收集整理制作：畅雨（chyinfo@tom.com）
日期：2003-07-15
原网址：http://www.dofactory.com/Patterns/Patterns.aspx#list

---赠人玫瑰之手 历久犹有余香---

Design patterns are recurring solutions to software design problems you find again and again in real-world application development. Design patterns are about design and interaction of objects, as well as providing a communication platform concerning elegant, reusable solutions to commonly encountered programming challenges.

第一部分 Creational Patterns

01 Abstract Factory - 抽象工厂模式

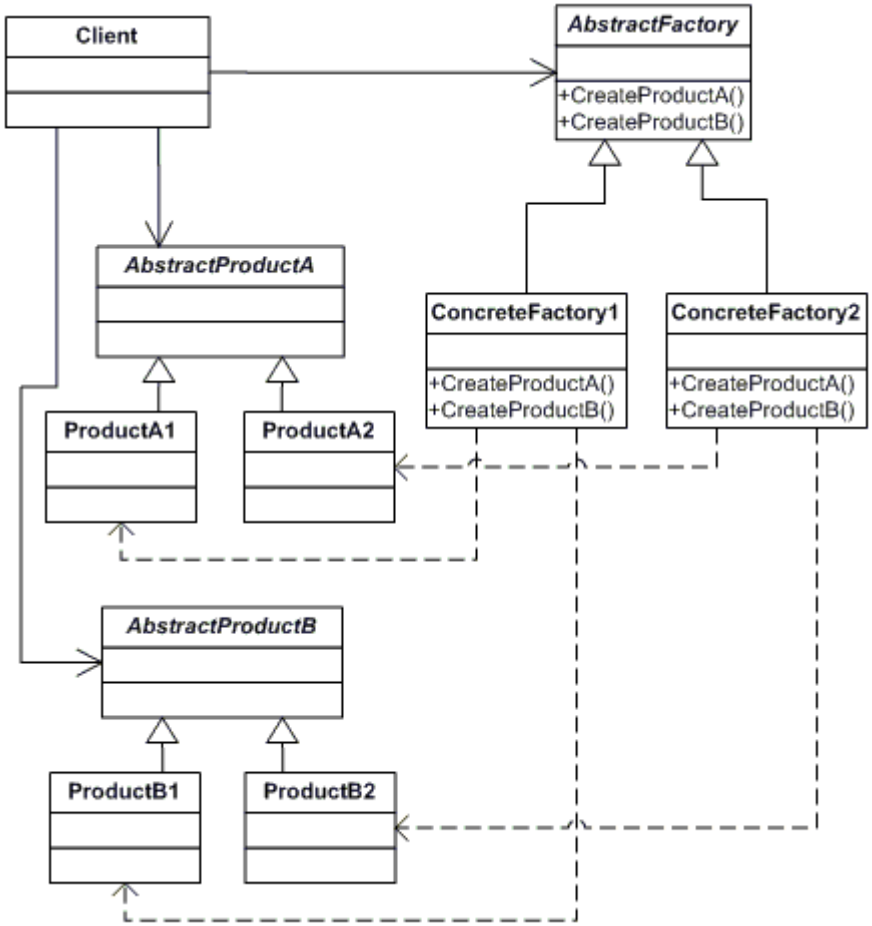
Definition

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

提供一个接口来构建一群(families)相关(related)或相依(dependent)的对象；而无须具体指定(specify)它们的具体类。

Frequency of use: 1 2 3 4 5 medium high

UML class diagram



Participants

The classes and/or objects participating in this pattern are:

- **AbstractFactory (ContinentFactory)**
  - declares an interface for operations that create abstract products
- **ConcreteFactory (AfricaFactory, AmericaFactory)**
  - implements the operations to create concrete product objects
- **AbstractProduct (Herbivore, Carnivore)**
  - declares an interface for a type of product object
- **Product (Wildebeest, Lion, Bison, Wolf)**
  - defines a product object to be created by the corresponding concrete factory
  - implements the AbstractProduct interface
- **Client (AnimalWorld)**
  - uses interfaces declared by AbstractFactory and AbstractProduct classes

Sample code in C#

- This **structural** code demonstrates the Abstract Factory pattern creating parallel hierarchies of objects. Object creation has been abstracted and there is no need for hard-coded class names in the client code.

```
// Abstract Factory pattern -- Structural example
using System;

// "AbstractFactory"

abstract class AbstractFactory
{
  // Methods
  abstract public AbstractProductA CreateProductA();
  abstract public AbstractProductB CreateProductB();
}

// "ConcreteFactory1"

class ConcreteFactory1 : AbstractFactory
{
  // Methods
  override public AbstractProductA CreateProductA()
  {
    return new ProductA1();
  }
  override public AbstractProductB CreateProductB()
  {
    return new ProductB1();
  }
}

// "ConcreteFactory2"

class ConcreteFactory2 : AbstractFactory
{
  // Methods
  override public AbstractProductA CreateProductA()
  {
    return new ProductA2();
  }

  override public AbstractProductB CreateProductB()
  {
    return new ProductB2();
  }
}

// "AbstractProductA"
```

```
abstract class AbstractProductA
{
}

// "AbstractProductB"

abstract class AbstractProductB
{
  // Methods
  abstract public void Interact( AbstractProductA a );
}

// "ProductA1"

class ProductA1 : AbstractProductA
{
}

// "ProductB1"

class ProductB1 : AbstractProductB
{
  // Methods
  override public void Interact( AbstractProductA a )
  {
    Console.WriteLine( this + " interacts with " + a );
  }
}

// "ProductA2"

class ProductA2 : AbstractProductA
{
}

// "ProductB2"

class ProductB2 : AbstractProductB
{
  // Methods
  override public void Interact( AbstractProductA a )
  {
    Console.WriteLine( this + " interacts with " + a );
```

```
  }
}

// "Client" - the interaction environment of the products

class Environment
{
  // Fields
  private AbstractProductA AbstractProductA;
  private AbstractProductB AbstractProductB;

  // Constructors
  public Environment( AbstractFactory factory )
  {
    AbstractProductB = factory.CreateProductB();
    AbstractProductA = factory.CreateProductA();
  }

  // Methods
  public void Run()
  {
    AbstractProductB.Interact( AbstractProductA );
  }
}

/// <summary>
/// ClientApp test environment
/// </summary>
class ClientApp
{
  public static void Main(string[] args)
  {
    AbstractFactory factory1 = new ConcreteFactory1();
    Environment e1 = new Environment( factory1 );
    e1.Run();

    AbstractFactory factory2 = new ConcreteFactory2();
    Environment e2 = new Environment( factory2 );
    e2.Run();
  }
}
```

Output

```
ProductB1 interacts with ProductA1
ProductB2 interacts with ProductA2
```

- This real-world code demonstrates the creation of different animal worlds for a computer game using different factories. Although the animals created by the Continent factories are different, the interactions among the animals remain the same.

```
// Abstract Factory pattern -- Real World example
using System;

// "AbstractFactory"

abstract class ContinentFactory
{
  // Methods
  abstract public Herbivore CreateHerbivore();
  abstract public Carnivore CreateCarnivore();
}

// "ConcreteFactory1"

class AfricaFactory : ContinentFactory
{
  // Methods
  override public Herbivore CreateHerbivore()
  {
    return new Wildebeest();
  }
  override public Carnivore CreateCarnivore()
  {
    return new Lion();
  }
}

// "ConcreteFactory2"

class AmericaFactory : ContinentFactory
{
  // Methods
  override public Herbivore CreateHerbivore()
  {
    return new Bison();
  }

  override public Carnivore CreateCarnivore()
  {
    return new Wolf();
```

```
  }
}

// "AbstractProductA"

abstract class Herbivore
{
}

// "AbstractProductB"

abstract class Carnivore
{
  // Methods
  abstract public void Eat( Herbivore h );
}

// "ProductA1"

class Wildebeest : Herbivore
{
}

// "ProductB1"

class Lion : Carnivore
{
  // Methods
  override public void Eat( Herbivore h )
  {
    // eat wildebeest
    Console.WriteLine( this + " eats " + h );
  }
}

// "ProductA2"

class Bison : Herbivore
{
}

// "ProductB2"

class Wolf : Carnivore
```

```
{
  // Methods
  override public void Eat( Herbivore h )
  {
    // Eat bison
    Console.WriteLine( this + " eats " + h );
  }
}

// "Client"

class AnimalWorld
{
  // Fields
  private Herbivore herbivore;
  private Carnivore carnivore;

  // Constructors
  public AnimalWorld( ContinentFactory factory )
  {
    carnivore = factory.CreateCarnivore();
    herbivore = factory.CreateHerbivore();
  }

  // Methods
  public void RunFoodChain()
  {
    carnivore.Eat( herbivore );
  }
}

/// <summary>
///  GameApp test class
/// </summary>
class GameApp
{
  public static void Main( string[] args )
  {
    // Create and run the Africa animal world
    ContinentFactory africa = new AfricaFactory();
    AnimalWorld world = new AnimalWorld( africa );
    world.RunFoodChain();

    // Create and run the America animal world
```

```
    ContinentFactory america = new AmericaFactory();
    world = new AnimalWorld( america );
    world.RunFoodChain();
  }
}
```

Output

```
Lion eats Wildebeest
Wolf eats Bison
```

02 Builder – 创建者模式

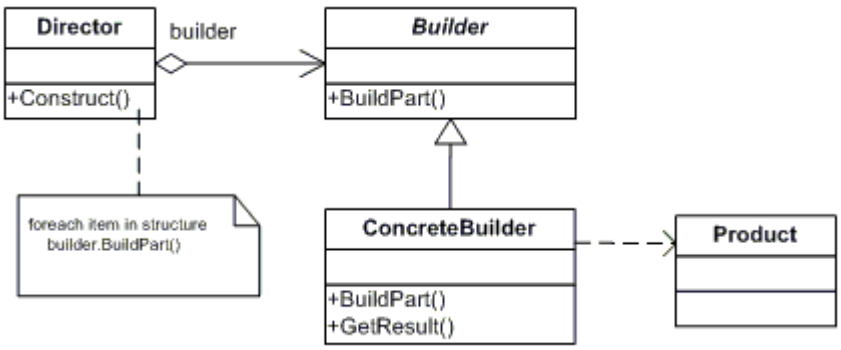
Definition

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

将一个复杂对象的构建方法(construction)从其表现;(representation)中分离开来以便同样的构建方法可以建立不同的表现。

Frequency of use: 1 2 3 4 5 medium

UML class diagram



Participants

The classes and/or objects participating in this pattern are:

- **Builder (VehicleBuilder)**
  - specifies an abstract interface for creating parts of a Product object
- **ConcreteBuilder (MotorCycleBuilder, CarBuilder, ScooterBuilder)**
  - constructs and assembles parts of the product by implementing the Builder interface
  - defines and keeps track of the representation it creates
  - provides an interface for retrieving the product
- **Director (Shop)**

- constructs an object using the Builder interface

- **Product (Vehicle)**
  - represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled
  - includes classes that define the constituent parts, including interfaces for assembling the parts into the final result

- -

Sample code in C#

This structural code demonstrates the Builder pattern in which complex objects are created in a step-by-step fashion. The construction process can create different object representations and provides a high level of control over the assembly of the objects.

```
// Builder pattern -- Structural example
using System;
using System.Collections;

// "Director"

class Director
{
  // Methods
  public void Construct( Builder builder )
  {
    builder.BuildPartA();
    builder.BuildPartB();
  }
}

// "Builder"

abstract class Builder
{
  // Methods
  abstract public void BuildPartA();
  abstract public void BuildPartB();
  abstract public Product GetResult();
}

// "ConcreteBuilder1"

class ConcreteBuilder1 : Builder
{
  // Fields
  private Product product;
```

```
  // Methods
  override public void BuildPartA()
  {
    product = new Product();
    product.Add( "PartA" );
  }

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

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

// "ConcreteBuilder2"

class ConcreteBuilder2 : Builder
{
  // Fields
  private Product product;

  // Methods
  override public void BuildPartA()
  {
    product = new Product();
    product.Add( "PartX" );
  }

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

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

```
// "Product"

class Product
{
  // Fields
  ArrayList parts = new ArrayList();

  // Methods
  public void Add( string part )
  {
    parts.Add( part );
  }

  public void Show()
  {
    Console.WriteLine( "\nProduct Parts -------" );
    foreach( string part in parts )
      Console.WriteLine( part );
  }
}
/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Create director and builders
    Director director = new Director( );

    Builder b1 = new ConcreteBuilder1();
    Builder b2 = new ConcreteBuilder2();

    // Construct two products
    director.Construct( b1 );
    Product p1 = b1.GetResult();
    p1.Show();

    director.Construct( b2 );
    Product p2 = b2.GetResult();
    p2.Show();
  }
}
```

Output

```
Product Parts -------
PartA
PartB

Product Parts -------
PartX
PartY
```

- This real-world code demonstates the Builder pattern in which different vehicles are assembled in a step-by-step fashion. The Shop uses VehicleBuilders to construct a variety of Vehicles in a series of sequential steps.

```
// Builder pattern -- Real World example
using System;
using System.Collections;

// "Director"

class Shop
{
  // Methods
  public void Construct( VehicleBuilder vehicleBuilder )
  {
    vehicleBuilder.BuildFrame();
    vehicleBuilder.BuildEngine();
    vehicleBuilder.BuildWheels();
    vehicleBuilder.BuildDoors();
  }
}

// "Builder"

abstract class VehicleBuilder
{
  // Fields
  protected Vehicle vehicle;

  // Properties
  public Vehicle Vehicle
  {
    get{ return vehicle; }
  }

  // Methods
  abstract public void BuildFrame();
```

```
  abstract public void BuildEngine();
  abstract public void BuildWheels();
  abstract public void BuildDoors();
}

// "ConcreteBuilder1"

class MotorCycleBuilder : VehicleBuilder
{
  // Methods
  override public void BuildFrame()
  {
    vehicle = new Vehicle( "MotorCycle" );
    vehicle[ "frame" ] = "MotorCycle Frame";
  }

  override public void BuildEngine()
  {
    vehicle[ "engine" ] = "500 cc";
  }

  override public void BuildWheels()
  {
    vehicle[ "wheels" ] = "2";
  }

  override public void BuildDoors()
  {
    vehicle[ "doors" ] = "0";
  }
}

// "ConcreteBuilder2"

class CarBuilder : VehicleBuilder
{
  // Methods
  override public void BuildFrame()
  {
    vehicle = new Vehicle( "Car" );
    vehicle[ "frame" ] = "Car Frame";
  }

  override public void BuildEngine()
```

```
  {
    vehicle[ "engine" ] = "2500 cc";
  }

  override public void BuildWheels()
  {
    vehicle[ "wheels" ] = "4";
  }

  override public void BuildDoors()
  {
    vehicle[ "doors" ] = "4";
  }
}

// "ConcreteBuilder3"

class ScooterBuilder : VehicleBuilder
{
  // Methods
  override public void BuildFrame()
  {
    vehicle = new Vehicle( "Scooter" );
    vehicle[ "frame" ] = "Scooter Frame";
  }

  override public void BuildEngine()
  {
    vehicle[ "engine" ] = "none";
  }

  override public void BuildWheels()
  {
    vehicle[ "wheels" ] = "2";
  }

  override public void BuildDoors()
  {
    vehicle[ "doors" ] = "0";
  }
}

// "Product"
```

```
class Vehicle
{
  // Fields
  private string type;
  private Hashtable parts = new Hashtable();

  // Constructors
  public Vehicle( string type )
  {
    this.type = type;
  }

  // Indexers
  public object this[ string key ]
  {
    get{ return parts[ key ]; }
    set{ parts[ key ] = value; }
  }

  // Methods
  public void Show()
  {
    Console.WriteLine( "\n--------------------------");
    Console.WriteLine( "Vehicle Type: "+ type );
    Console.WriteLine( " Frame : " + parts[ "frame" ] );
    Console.WriteLine( " Engine : "+ parts[ "engine"] );
    Console.WriteLine( " #Wheels: "+ parts[ "wheels"] );
    Console.WriteLine( " #Doors : "+ parts[ "doors" ] );
  }
}

/// <summary>
/// BuilderApp test
/// </summary>
public class BuilderApp
{
  public static void Main( string[] args )
  {
    // Create shop and vehicle builders
    Shop shop = new Shop();
    VehicleBuilder b1 = new ScooterBuilder();
    VehicleBuilder b2 = new CarBuilder();
    VehicleBuilder b3 = new MotorCycleBuilder();
```

```
    // Construct and display vehicles
    shop.Construct( b1 );
    b1.Vehicle.Show();

    shop.Construct( b2 );
    b2.Vehicle.Show();

    shop.Construct( b3 );
    b3.Vehicle.Show();
  }
}
```

Output

```
---------------------------
Vehicle Type: Scooter
Frame : Scooter Frame
Engine : none
#Wheels: 2
#Doors : 0

---------------------------
Vehicle Type: Car
Frame : Car Frame
Engine : 2500 cc
#Wheels: 4
#Doors : 4

---------------------------
Vehicle Type: MotorCycle
Frame : MotorCycle Frame
Engine : 500 cc
#Wheels: 2
#Doors : 0
```

03 Factory method – 工厂方法模式

Definition

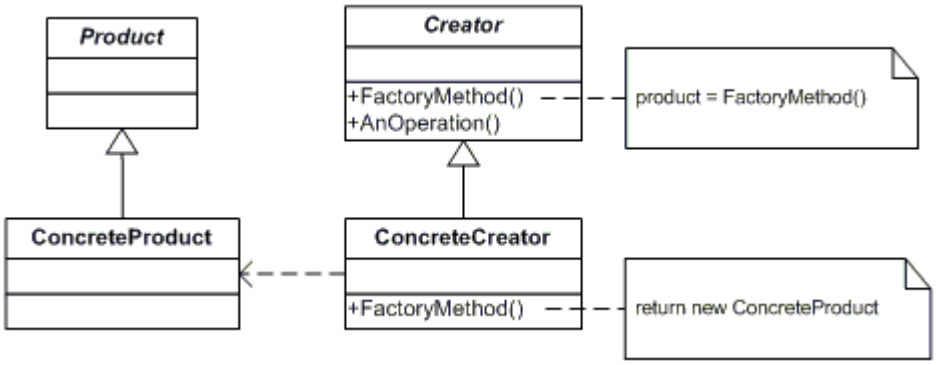
Define an interface for creating an object, but let subclasses decide which class to instantiate.

Factory Method lets a class defer instantiation to subclasses.

工厂方法定义构建对象的接口；但是让子类决定哪一个类构建实体对象(instantiate)，工厂方法将构建实体对象委托(defer)给子类。《译注：因为在父类并不知道要构建的对象实体是哪一个子类：因此将构建对象的责任委托或者延迟到子类才决定。》

Frequency of use: 1 2 3 4 5 medium high

UML class diagram



Participants

The classes and/or objects participating in this pattern are:

- **Product (Page)**
  - defines the interface of objects the factory method creates
- **ConcreteProduct (SkillsPage, EducationPage, ExperiencePage)**
  - implements the Product interface
- **Creator (Document)**
  - declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
  - may call the factory method to create a Product object.
- **ConcreteCreator (Report, Resume)**
  - overrides the factory method to return an instance of a ConcreteProduct.

Sample code in C#

```
// Factory Method pattern -- Structural example
using System;
using System.Collections;

// "Product"

abstract class Product
{
}

// "ConcreteProductA"

class ConcreteProductA : Product
{
```

```
}

// "ConcreteProductB"

class ConcreteProductB : Product
{
}

// "Creator"

abstract class Creator
{
  // Methods
  abstract public Product FactoryMethod();
}

// "ConcreteCreatorA"

class ConcreteCreatorA : Creator
{
  // Methods
  override public Product FactoryMethod()
  {
    return new ConcreteProductA();
  }
}

// "ConcreteCreatorB"

class ConcreteCreatorB : Creator
{
  // Methods
  override public Product FactoryMethod()
  {
    return new ConcreteProductB();
  }
}

/// <summary>
/// Client test
/// </summary>
class Client
{
  public static void Main( string[] args )
```

```
  {

    // FactoryMethod returns ProductA
    Creator c = new ConcreteCreatorA();
    Product p = c.FactoryMethod();
    Console.WriteLine( "Created {0}", p );

    // FactoryMethod returns ProductB
    c = new ConcreteCreatorB();
    p = c.FactoryMethod();
    Console.WriteLine( "Created {0}", p );

  }
}
```

Output

```
Created ConcreteProductA
Created ConcreteProductB
```

- This real-world code demonstrates the Factory method offering flexibility in creating different documents. The derived Document classes Report and Resume instantiate extended versions of the Document class. Here, the Factory Method is called in the constructor of the Document base class.

```
// Factory Method pattern -- Real World example
using System;
using System.Collections;

// "Product"

abstract class Page
{
}

// "ConcreteProduct"

class SkillsPage : Page
{
}

// "ConcreteProduct"

class EducationPage : Page
{
}
```

```
// "ConcreteProduct"

class ExperiencePage : Page
{
}

// "ConcreteProduct"

class IntroductionPage : Page
{
}

// "ConcreteProduct"

class ResultsPage : Page
{
}

// "ConcreteProduct"

class ConclusionPage : Page
{
}

// "ConcreteProduct"

class SummaryPage : Page
{
}

// "ConcreteProduct"

class BibliographyPage : Page
{
}

// "Creator"

abstract class Document
{
  // Fields
  protected ArrayList pages = new ArrayList();

  // Constructor
```

```
  public Document()
  {
    this.CreatePages();
  }

  // Properties
  public ArrayList Pages
  {
    get{ return pages; }
  }

  // Factory Method
  abstract public void CreatePages();
}

// "ConcreteCreator"

class Resume : Document
{
  // Factory Method implementation
  override public void CreatePages()
  {
    pages.Add( new SkillsPage() );
    pages.Add( new EducationPage() );
    pages.Add( new ExperiencePage() );
  }
}

// "ConcreteCreator"

class Report : Document
{
  // Factory Method implementation
  override public void CreatePages()
  {
    pages.Add( new IntroductionPage() );
    pages.Add( new ResultsPage() );
    pages.Add( new ConclusionPage() );
    pages.Add( new SummaryPage() );
    pages.Add( new BibliographyPage() );
  }
}

/// <summary>
```

```
///  FactoryMethodApp test
/// </summary>
class FactoryMethodApp
{
  public static void Main( string[] args )
  {
    Document[] docs = new Document[ 2 ];

    // Note: constructors call Factory Method
    docs[0] = new Resume();
    docs[1] = new Report();

    // Display document pages
    foreach( Document document in docs )
    {
      Console.WriteLine( "\n" + document + " ------- " );
      foreach( Page page in document.Pages )
        Console.WriteLine( " " + page );
    }
  }
}
```

Output

```
Resume -------
SkillsPage
EducationPage
ExperiencePage

Report -------
IntroductionPage
ResultsPage
ConclusionPage
SummaryPage
BibliographyPage
```

04 Prototype – 原型模式

Definition

Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.

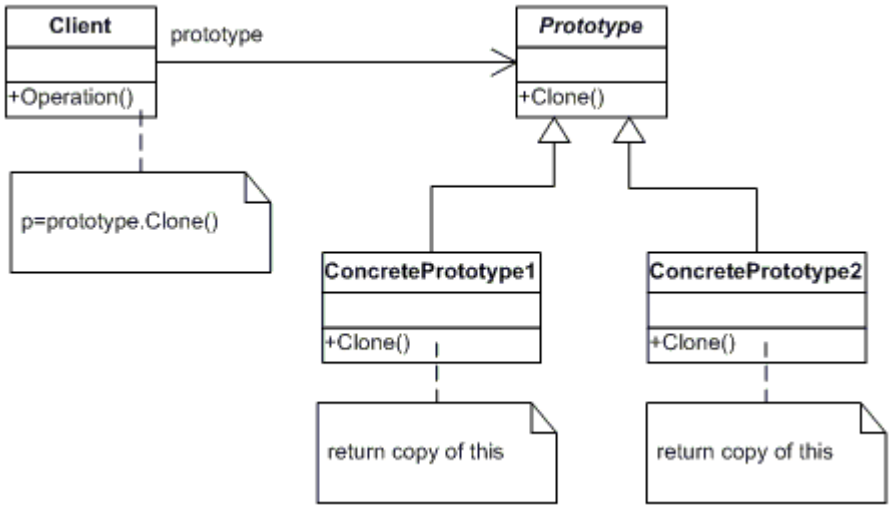
使用原型实例(prototypical instance)列示(specify)欲构建的对象；构建时则复制这个原型成新的对象。

《译注：Delphi 实现 Prototype 因无 clone 复制的功能；故一般实现方式是将对象存放在 Tlist 中再依实际需

要从 Tlist 中的对象构建(create)新的对象；再附于对象必须的属性 RTTI。本书的范例使用 TmazePrototypeFactory 的对象储存原型详范例程序》

Frequency of use: 1 2 3 4 5 medium low

UML class diagram



Participants

The classes and/or objects participating in the Prototype pattern are:

- **Prototype (ColorPrototype)**
  - declares an interace for cloning itself
- **ConcretePrototype (Color)**
  - implements an operation for cloning itself
- **Client (ColorManager)**
  - creates a new object by asking a prototype to clone itself

Sample code in C#

This structural code demonstrates the Prototype pattern in which new objects are created by copying pre-existing objects (prototypes) of the same class.

```
// Prototype pattern -- Structural example
using System;

// "Prototype"

abstract class Prototype
{
  // Fields
  private string id;
```

```
  // Constructors
  public Prototype( string id )
  {
    this.id = id;
  }

  public string Id
  {
    get{ return id; }
  }

  // Methods
  abstract public Prototype Clone();
}

// "ConcretePrototype1"

class ConcretePrototype1 : Prototype
{
  // Constructors
  public ConcretePrototype1( string id ) : base ( id ) {}

  // Methods
  override public Prototype Clone()
  {
    // Shallow copy
    return (Prototype)this.MemberwiseClone();
  }
}

// "ConcretePrototype2"

class ConcretePrototype2 : Prototype
{
  // Constructors
  public ConcretePrototype2( string id ) : base ( id ) {}

  // Methods
  override public Prototype Clone()
  {
    // Shallow copy
    return (Prototype)this.MemberwiseClone();
  }
```

```
}

/// <summary>
/// Client test
/// </summary>
class Client
{
  public static void Main( string[] args )
  {
   // Create two instances and clone each
   ConcretePrototype1 p1 = new ConcretePrototype1( "I" );
   ConcretePrototype1 c1 = (ConcretePrototype1)p1.Clone();
   Console.WriteLine( "Cloned: {0}", c1.Id );

   ConcretePrototype2 p2 = new ConcretePrototype2( "II" );
   ConcretePrototype2 c2 = (ConcretePrototype2)p2.Clone();
   Console.WriteLine( "Cloned: {0}", c2.Id );
  }
}
```

Output

```
Cloned: I
Cloned: II
```

I This real-world code demonstrates the Prototype pattern in which new Color objects are created by copying pre-existing, user-defined Colors of the same type.

```
// Prototype pattern -- Real World example
using System;
using System.Collections;

// "Prototype"

abstract class ColorPrototype
{
  // Methods
  public abstract ColorPrototype Clone();
}

// "ConcretePrototype"

class Color : ColorPrototype
{
  // Fields
  private int red, green, blue;
```

```
  // Constructors
  public Color( int red, int green, int blue)
  {
    this.red = red;
    this.green = green;
    this.blue = blue;
  }

  // Methods
  public override ColorPrototype Clone()
  {
    // Creates a 'shallow copy'
    return (ColorPrototype) this.MemberwiseClone();
  }

  public void Display()
  {
    Console.WriteLine( "RGB values are: {0},{1},{2}",
                                        red, green, blue );
  }
}

// Prototype manager

class ColorManager
{
  // Fields
  Hashtable colors = new Hashtable();

  // Indexers
  public ColorPrototype this[ string name ]
  {
    get{ return (ColorPrototype)colors[ name ]; }
    set{ colors.Add( name, value ); }
  }
}

/// <summary>
///  PrototypeApp test
/// </summary>
class PrototypeApp
{
  public static void Main( string[] args )
  {
```

```
        ColorManager colormanager = new ColorManager();

        // Initialize with standard colors
        colormanager[ "red" ] = new Color( 255, 0, 0 );
        colormanager[ "green" ] = new Color( 0, 255, 0 );
        colormanager[ "blue" ] = new Color( 0, 0, 255 );

        // User adds personalized colors
        colormanager[ "angry" ] = new Color( 255, 54, 0 );
        colormanager[ "peace" ] = new Color( 128, 211, 128 );
        colormanager[ "flame" ] = new Color( 211, 34, 20 );

        // User uses selected colors
        string colorName = "red";
        Color c1 = (Color)colormanager[ colorName ].Clone();
        c1.Display();

        colorName = "peace";
        Color c2 = (Color)colormanager[ colorName ].Clone();
        c2.Display();

        colorName = "flame";
        Color c3 = (Color)colormanager[ colorName ].Clone();
        c3.Display();
    }
}
```

Output

```
RGB values are: 255,0,0
RGB values are: 128,211,128
RGB values are: 211,34,20
```

05 Singleton– 单件模式

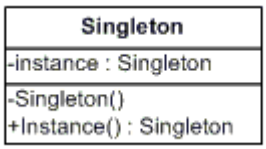
Definition

Ensure a class has only one instance and provide a global point of access to it.

确保一个类在系统中只有一个对象实例，并提供一个整体的存取标示。

Frequency of use: 1 2 3 4 5 high

UML class diagram



- -

Participants

- **Singleton (LoadBalancer)**
  - defines an Instance operation that lets clients access its unique instance. Instance is a class operation.
  - responsible for creating and maintaining its own unique instance.

- -

Sample code in C#

```
// Singleton pattern -- Structural example
using System;

// "Singleton"

class Singleton
{
  // Fields
  private static Singleton instance;

  // Constructor
  protected Singleton() {}

  // Methods
  public static Singleton Instance()
  {
    // Uses "Lazy initialization"
    if( instance == null )
      instance = new Singleton();

    return instance;
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
```

```
{
  public static void Main()
  {
    // Constructor is protected -- cannot use new
    Singleton s1 = Singleton.Instance();
    Singleton s2 = Singleton.Instance();

    if( s1 == s2 )
      Console.WriteLine( "The same instance" );
  }
}
```

Output

```
The same instance
```

- This real-world code demonstrates the Singleton pattern as a LoadBalancing object. Only a single instance (the singleton) of the class can be created because servers may dynamically come on- or off-line and every request must go throught the one object that has knowledge about the state of the (web) farm.

```
// Singleton pattern -- Real World example
using System;
using System.Collections;
using System.Threading;

// "Singleton"

class LoadBalancer
{
  // Fields
  private static LoadBalancer balancer;
  private ArrayList servers = new ArrayList();
  private Random random = new Random();

  // Constructors (protected)
  protected LoadBalancer()
  {
    // List of available servers
    servers.Add( "ServerI" );
    servers.Add( "ServerII" );
    servers.Add( "ServerIII" );
    servers.Add( "ServerIV" );
    servers.Add( "ServerV" );
  }

  // Methods
```

```
  public static LoadBalancer GetLoadBalancer()
  {
    // Support multithreaded applications through
    // "Double checked locking" pattern which avoids
    // locking every time the method is invoked
    if( balancer == null )
    {
      // Only one thread can obtain a mutex
      Mutex mutex = new Mutex();
      mutex.WaitOne();

      if( balancer == null )
        balancer = new LoadBalancer();

      mutex.Close();
    }
    return balancer;
  }

  // Properties
  public string Server
  {
    get
    {
      // Simple, but effective random load balancer
      int r = random.Next( servers.Count );
      return servers[ r ].ToString();
    }
  }
}

/// <summary>
/// SingletonApp test
/// </summary>
///
public class SingletonApp
{
  public static void Main( string[] args )
  {
    LoadBalancer b1 = LoadBalancer.GetLoadBalancer();
    LoadBalancer b2 = LoadBalancer.GetLoadBalancer();
    LoadBalancer b3 = LoadBalancer.GetLoadBalancer();
    LoadBalancer b4 = LoadBalancer.GetLoadBalancer();
```

```
    // Same instance?
    if( (b1 == b2) && (b2 == b3) && (b3 == b4) )
      Console.WriteLine( "Same instance" );

    // Do the load balancing
    Console.WriteLine( b1.Server );
    Console.WriteLine( b2.Server );
    Console.WriteLine( b3.Server );
    Console.WriteLine( b4.Server );
  }
}
```

Output

```
Same instance
ServerV
ServerIII
ServerIII
ServerIV
```

第二部分 **Structural Patterns**

06 Adapter – 适配器模式

Definition

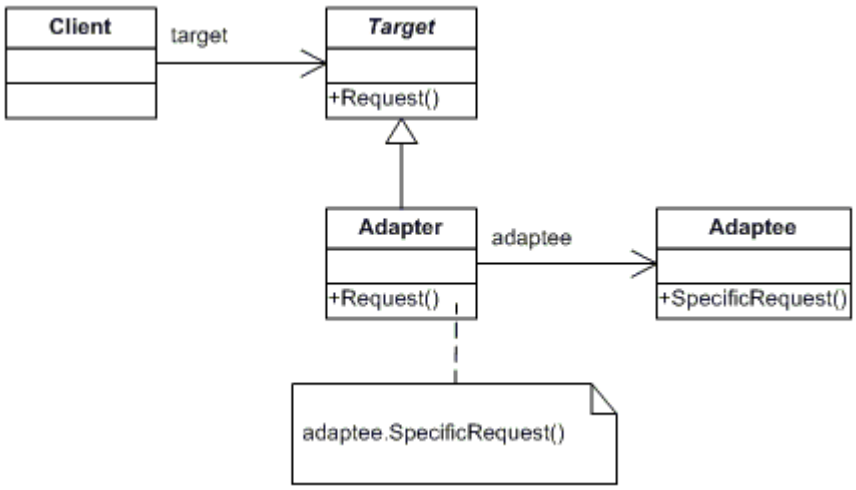
Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

转接一个类的接口成另一个使用端所要求的接口。适配器让多个类共同工作；而这些类除了因为其接口兼容否则无法共同工作。

Frequency of use: 

medium high

UML class diagram



Participants

The classes and/or objects participating in the Adapter pattern are:

- **Target (ChemicalCompound)**
  - defines the domain-specific interface that Client uses.
- **Adapter (Compound)**
  - adapts the interface Adaptee to the Target interface.
- **Adaptee (ChemicalDatabank)**
  - defines an existing interface that needs adapting.
- **Client (AdapterApp)**
  - collaborates with objects conforming to the Target interface.

Sample code in C#

This structural code demonstrates the Adapter pattern which maps the interface of one class onto another so that they can work together. These incompatible classes may come from different libraries or frameworks.

```
// Adapter pattern -- Structural example
using System;

// "Target"

class Target
{
  // Methods
  virtual public void Request()
  {
    // Normal implementation goes here
  }
```

```
}

// "Adapter"

class Adapter : Target
{
  // Fields
  private Adaptee adaptee = new Adaptee();

  // Methods
  override public void Request()
  {
    // Possibly do some data manipulation
    // and then call SpecificRequest
    adaptee.SpecificRequest();
  }
}

// "Adaptee"

class Adaptee
{
  // Methods
  public void SpecificRequest()
  {
    Console.WriteLine("Called SpecificRequest()" );
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main(string[] args)
  {
    // Create adapter and place a request
    Target t = new Adapter();
    t.Request();

  }
}
```

Output

```
Called SpecificRequest()
```

- This real-world code demonstrates the use of a legacy chemical databank. Chemical compound objects access the databank through an Adapter interface.

```
// Adapter pattern -- Real World example

using System;

// "Target"

class ChemicalCompound
{
  // Fields
  protected string name;
  protected float boilingPoint;
  protected float meltingPoint;
  protected double molecularWeight;
  protected string molecularFormula;

  // Constructors
  public ChemicalCompound( string name )
  {
    this.name = name;
  }

  // Properties
  public float BoilingPoint
  {
    get{ return boilingPoint; }
  }

  public float MeltingPoint
  {
    get{ return meltingPoint; }
  }

  public double MolecularWeight
  {
    get{ return molecularWeight; }
  }

  public string MolecularFormula
  {
    get{ return molecularFormula; }
```

```
  }
}

// "Adapter"

class Compound : ChemicalCompound
{
  // Fields
  private ChemicalDatabank bank;

  // Constructors
  public Compound( string name ) : base( name )
  {
    // Adaptee
    bank = new ChemicalDatabank();
    // Adaptee request methods
    boilingPoint = bank.GetCriticalPoint( name, "B" );
    meltingPoint = bank.GetCriticalPoint( name, "M" );
    molecularWeight = bank.GetMolecularWeight( name );
    molecularFormula = bank.GetMolecularStructure( name );
  }

  // Methods
  public void Display()
  {
    Console.WriteLine("\nCompound: {0} ------ ",name );
    Console.WriteLine(" Formula: {0}",MolecularFormula);
    Console.WriteLine(" Weight : {0}",MolecularWeight );
    Console.WriteLine(" Melting Pt: {0}",MeltingPoint );
    Console.WriteLine(" Boiling Pt: {0}",BoilingPoint );
  }
}

// "Adaptee"

class ChemicalDatabank
{
  // Methods -- the Databank 'legacy API'
  public float GetCriticalPoint( string compound, string point )
  {
    float temperature = 0.0F;
    // Melting Point
    if( point == "M" )
    {
```

```
        switch( compound.ToLower() )
        {
          case "water": temperature = 0.0F; break;
          case "benzene" : temperature = 5.5F; break;
          case "alcohol": temperature = -114.1F; break;
        }
      }
      // Boiling Point
      else
      {
        switch( compound.ToLower() )
        {
          case "water": temperature = 100.0F;break;
          case "benzene" : temperature = 80.1F; break;
          case "alcohol": temperature = 78.3F; break;
        }
      }
      return temperature;
    }

    public string GetMolecularStructure( string compound )
    {
      string structure = "";
      switch( compound.ToLower() )
      {
        case "water": structure = "H2O"; break;
        case "benzene" : structure = "C6H6"; break;
        case "alcohol": structure = "C2H6O2"; break;
      }
      return structure;
    }

    public double GetMolecularWeight( string compound )
    {
      double weight = 0.0;
      switch( compound.ToLower() )
      {
        case "water": weight = 18.015; break;
        case "benzene" : weight = 78.1134; break;
        case "alcohol": weight = 46.0688; break;
      }
      return weight;
    }
  }
```

```
/// <summary>
/// AdapterApp test application
/// </summary>
public class AdapterApp
{
  public static void Main(string[] args)
  {
    // Retrieve and display water characteristics
    Compound water = new Compound( "Water" );
    water.Display();

    // Retrieve and display benzene characteristics
    Compound benzene = new Compound( "Benzene" );
    benzene.Display();

    // Retrieve and display alcohol characteristics
    Compound alcohol = new Compound( "Alcohol" );
    alcohol.Display();

  }
}
```

Output

```
Compound: Water ------
Formula: H20
Weight : 18.015
Melting Pt: 0
Boiling Pt: 100

Compound: Benzene ------
Formula: C6H6
Weight : 78.1134
Melting Pt: 5.5
Boiling Pt: 80.1

Compound: Alcohol ------
Formula: C2H6O2
Weight : 46.0688
Melting Pt: -114.1
Boiling Pt: 78.3
```

07 Bridge – 桥接模式

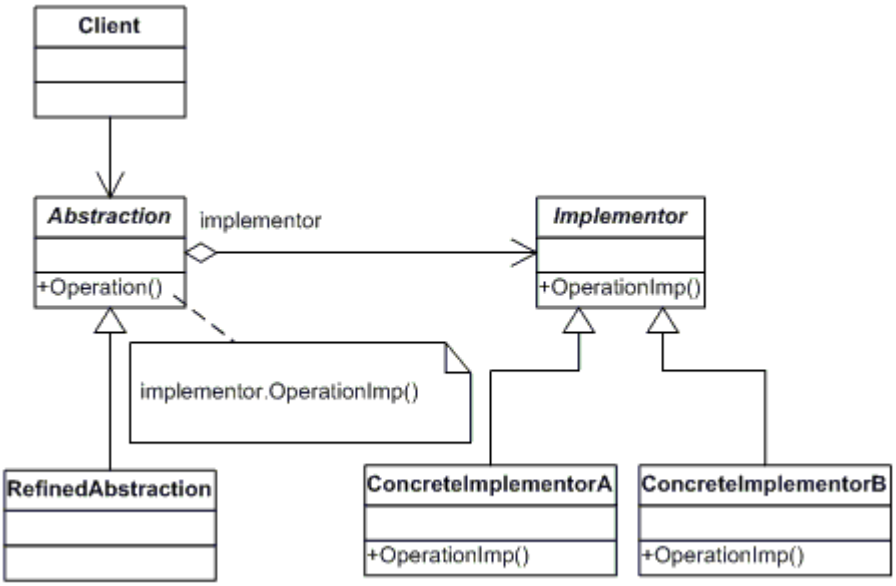
Definition

Decouple an abstraction from its implementation so that the two can vary independently.

降低(decouple)属性(abstraction)与实现的耦合性，让两者可以独立的改变。

Frequency of use: 1 2 3 4 5 medium

UML class diagram



Participants

- **Abstraction (BusinessObject)**
  - defines the abstraction's interface.
  - maintains a reference to an object of type Implementor.
- **RefinedAbstraction (CustomersBusinessObject)**
  - extends the interface defined by Abstraction.
- **Implementor (DataObject)**
  - defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementation interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- **ConcreteImplementor (CustomersDataObject)**
  - implements the Implementor interface and defines its concrete implementation.

Sample code in C#

- This structural code demonstrates the Bridge pattern which separates (decouples) the interface from its implementation. The implementation can evolve without changing clients which use the abstraction of the object.

```
// Bridge pattern -- Structural example
using System;

// "Abstraction"

class Abstraction
{
  // Fields
  protected Implementor implementor;

  // Properties
  public Implementor Implementor
  {
    set{ implementor = value; }
  }

  // Methods
  virtual public void Operation()
  {
    implementor.Operation();
  }
}

// "Implementor"

abstract class Implementor
{
  // Methods
  abstract public void Operation();
}

// "RefinedAbstraction"

class RefinedAbstraction : Abstraction
{
  // Methods
  override public void Operation()
  {
    implementor.Operation();
```

```
  }
}

// "ConcreteImplementorA"

class ConcreteImplementorA : Implementor
{
  // Methods
  override public void Operation()
  {
    Console.WriteLine("ConcreteImplementorA Operation");
  }
}

// "ConcreteImplementorB"

class ConcreteImplementorB : Implementor
{
  // Methods
  override public void Operation()
  {
    Console.WriteLine("ConcreteImplementorB Operation");
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    Abstraction abstraction = new RefinedAbstraction();

    // Set implementation and call
    abstraction.Implementor = new ConcreteImplementorA();
    abstraction.Operation();

    // Change implemention and call
    abstraction.Implementor = new ConcreteImplementorB();
    abstraction.Operation();
  }
}
```

Output

```
ConcreteImplementorA Operation
ConcreteImplementorB Operation
```

- This real-world code demonstrates the Bridge pattern in which a BusinessObject abstraction is decoupled from the implementation in DataObject. The DataObject implementations can evolve dynamically without changing any clients.

```
// Bridge pattern -- Real World example
using System;
using System.Collections;

// "Abstraction"

class BusinessObject
{
  // Fields
  private DataObject dataObject;
  protected string group;

  // Constructors
  public BusinessObject( string group )
  {
    this.group = group;
  }

  // Properties
  public DataObject DataObject
  {
    set{ dataObject = value; }
    get{ return dataObject; }
  }

  // Methods
  virtual public void Next()
  {
    dataObject.NextRecord();
  }

  virtual public void Prior()
  {
    dataObject.PriorRecord();
  }

  virtual public void New( string name )
  {
```

```
    dataObject.NewRecord( name );
  }

  virtual public void Delete( string name )
  {
    dataObject.DeleteRecord( name );
  }

  virtual public void Show()
  {
    dataObject.ShowRecord();
  }

  virtual public void ShowAll()
  {
    Console.WriteLine( "Customer Group: {0}", group );
    dataObject.ShowAllRecords();
  }
}

// "RefinedAbstraction"

class CustomersBusinessObject : BusinessObject
{
  // Constructors
  public CustomersBusinessObject( string group )
      : base( group ){}

  // Methods
  override public void ShowAll()
  {
    // Add separator lines
    Console.WriteLine();
    Console.WriteLine( "------------------------" );
    base.ShowAll();
    Console.WriteLine( "------------------------" );
  }
}

// "Implementor"

abstract class DataObject
{
  // Methods
```

```
  abstract public void NextRecord();
  abstract public void PriorRecord();
  abstract public void NewRecord( string name );
  abstract public void DeleteRecord( string name );
  abstract public void ShowRecord();
  abstract public void ShowAllRecords();
}

// "ConcreteImplementor"

class CustomersDataObject : DataObject
{
  // Fields
  private ArrayList customers = new ArrayList();
  private int current = 0;

  // Constructors
  public CustomersDataObject()
  {
    // Loaded from a database
    customers.Add( "Jim Jones" );
    customers.Add( "Samual Jackson" );
    customers.Add( "Allen Good" );
    customers.Add( "Ann Stills" );
    customers.Add( "Lisa Giolani" );
  }

  // Methods
  public override void NextRecord()
  {
    if( current <= customers.Count - 1 )
    current++;
  }

  public override void PriorRecord()
  {
    if( current > 0 )
      current--;
  }

  public override void NewRecord( string name )
  {
    customers.Add( name );
  }
```

```
  public override void DeleteRecord( string name )
  {
    customers.Remove( name );
  }

  public override void ShowRecord()
  {
    Console.WriteLine( customers[ current ] );
  }

  public override void ShowAllRecords()
  {
    foreach( string name in customers )
      Console.WriteLine( " " + name );
  }
}

/// <summary>
/// Client test
/// </summary>
public class BusinessApp
{
  public static void Main( string[] args )
  {
    // Create RefinedAbstraction
    CustomersBusinessObject customers =
    new CustomersBusinessObject(" Chicago ");

    // Set ConcreteImplementor
    customers.DataObject = new CustomersDataObject();

    // Exercise the bridge
    customers.Show();
    customers.Next();
    customers.Show();
    customers.Next();
    customers.Show();
    customers.New( "Henry Velasquez" );

    customers.ShowAll();
  }
}
```

Output

```
Jim Jones
Samual Jackson
Allen Good

------------------------
Customer Group: Chicago
Jim Jones
Samual Jackson
Allen Good
Ann Stills
Lisa Giolani
Henry Velasquez
------------------------
```

08 Composite – 组合模式

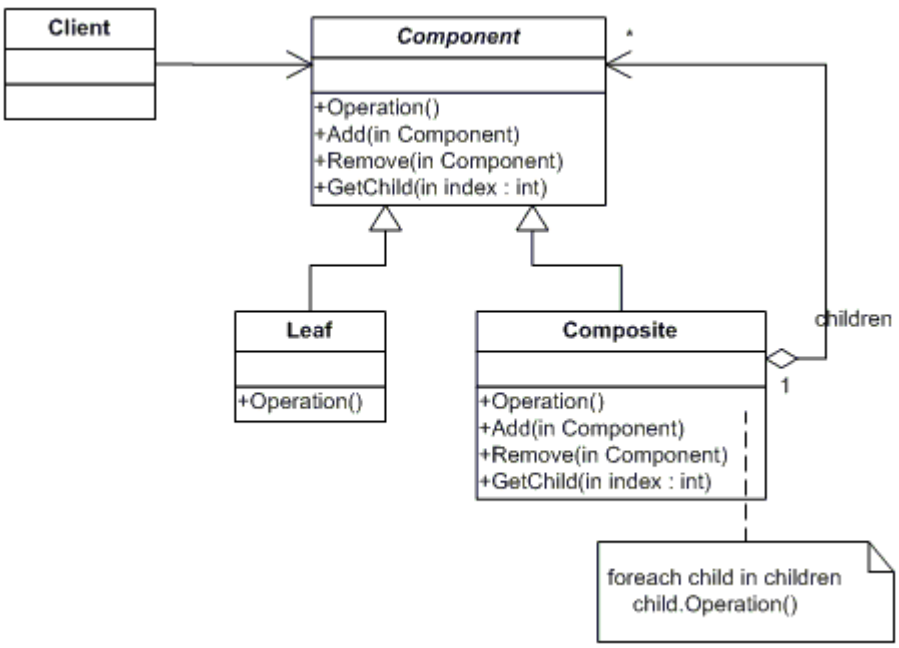
Definition

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

复合多个对象成树状结构以表现个别与整体(part-whole)的层级架构。复合模式让使用端以一致的方式处理个别及复合对象。

Frequency of use: 1 2 3 4 5 high

UML class diagram



Participants

The classes and/or objects participating in the Composite pattern are:

- **Component (DrawingElement)**
  - declares the interface for objects in the composition.
  - implements default behavior for the interface common to all classes, as appropriate.
  - declares an interface for accessing and managing its child components.
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- **Leaf (PrimitiveElement)**
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition.
- **Composite (CompositeElement)**
  - defines behavior for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.
- **Client (CompositeApp)**
  - manipulates objects in the composition through the Component interface.

Sample code in C#

This structural code demonstrates the Composite pattern which allows the creation of a tree structure in which individual nodes are accessed uniformly whether they are leaf nodes or branch (composite) nodes.

```
// Composite pattern -- Structural example
using System;
using System.Text;
using System.Collections;

// "Component"

abstract class Component
{
  // Fields
  protected string name;

  // Constructors
  public Component( string name )
  {
    this.name = name;
  }
```

```
  // Methods
  abstract public void Add(Component c);
  abstract public void Remove( Component c );
  abstract public void Display( int depth );
}

// "Composite"

class Composite : Component
{
  // Fields
  private ArrayList children = new ArrayList();

  // Constructors
  public Composite( string name ) : base( name ) {}

  // Methods
  public override void Add( Component component )
  {
    children.Add( component );
  }
  public override void Remove( Component component )
  {
    children.Remove( component );
  }
  public override void Display( int depth )
  {
    Console.WriteLine( new String( '-', depth ) + name );

    // Display each of the node's children
    foreach( Component component in children )
      component.Display( depth + 2 );
  }
}

// "Leaf"

class Leaf : Component
{
  // Constructors
  public Leaf( string name ) : base( name ) {}

  // Methods
```

```
  public override void Add( Component c )
  {
    Console.WriteLine("Cannot add to a leaf");
  }

  public override void Remove( Component c )
  {
    Console.WriteLine("Cannot remove from a leaf");
  }

  public override void Display( int depth )
  {
    Console.WriteLine( new String( '-', depth ) + name );
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Create a tree structure
    Composite root = new Composite( "root" );
    root.Add( new Leaf( "Leaf A" ));
    root.Add( new Leaf( "Leaf B" ));
    Composite comp = new Composite( "Composite X" );

    comp.Add( new Leaf( "Leaf XA" ) );
    comp.Add( new Leaf( "Leaf XB" ) );
    root.Add( comp );

    root.Add( new Leaf( "Leaf C" ));

    // Add and remove a leaf
    Leaf l = new Leaf( "Leaf D" );
    root.Add( l );
    root.Remove( l );

    // Recursively display nodes
    root.Display( 1 );
  }
}
```

Output

```
-root
---Leaf A
---Leaf B
---Composite X
-----Leaf XA
-----Leaf XB
---Leaf C
```

- This real-world code demonstrates the Composite pattern used in building a graphical tree structure made up of primitive nodes (lines, circles, etc) and composite nodes (groups of drawing elements that make up more complex elements).

```
// Composite pattern -- Real World example
using System;
using System.Collections;

// "Component"

abstract class DrawingElement
{
  // Fields
  protected string name;

  // Constructors
  public DrawingElement( string name )
  {
    this.name = name;
  }

  // Methods
  abstract public void Add( DrawingElement d );
  abstract public void Remove( DrawingElement d );
  abstract public void Display( int indent );
}

// "Leaf"

class PrimitiveElement : DrawingElement
{
  // Constructors
  public PrimitiveElement( string name ) : base( name ) {}

  // Methods
  public override void Add( DrawingElement c )
  {
```

```
    Console.WriteLine("Cannot Add");
  }

  public override void Remove( DrawingElement c )
  {
    Console.WriteLine("Cannot Remove");
  }

  public override void Display( int indent )
  {
    Console.WriteLine( new String( '-', indent ) +
                                    " draw a {0}", name );
  }
}

// "Composite"

class CompositeElement : DrawingElement
{
  // Fields
  private ArrayList elements = new ArrayList();

  // Constructors
  public CompositeElement( string name )
                                        : base( name ) {}

  // Methods
  public override void Add( DrawingElement d )
  {
    elements.Add( d );
  }

  public override void Remove( DrawingElement d )
  {
    elements.Remove( d );
  }

  public override void Display( int indent )
  {
    Console.WriteLine( new String( '-', indent ) +
                                    "+ " + name );

    // Display each child element on this node
    foreach( DrawingElement c in elements )
```

```
        c.Display( indent + 2 );
    }
}

/// <summary>
///  CompositeApp test
/// </summary>
public class CompositeApp
{
    public static void Main( string[] args )
    {
        // Create a tree structure
        CompositeElement root = new
                        CompositeElement( "Picture" );
        root.Add( new PrimitiveElement( "Red Line" ));
        root.Add( new PrimitiveElement( "Blue Circle" ));
        root.Add( new PrimitiveElement( "Green Box" ));

        CompositeElement comp = new
                        CompositeElement( "Two Circles" );
        comp.Add( new PrimitiveElement( "Black Circle" ) );
        comp.Add( new PrimitiveElement( "White Circle" ) );
        root.Add( comp );

        // Add and remove a PrimitiveElement
        PrimitiveElement l = new
                        PrimitiveElement( "Yellow Line" );
        root.Add( l );
        root.Remove( l );

        // Recursively display nodes
        root.Display( 1 );
    }
}
```

output

```
-+ Picture
--- draw a Red Line
--- draw a Blue Circle
--- draw a Green Box
---+ Two Circles
----- draw a Black Circle
----- draw a White Circle
```

09 Decorator – 装饰者模式

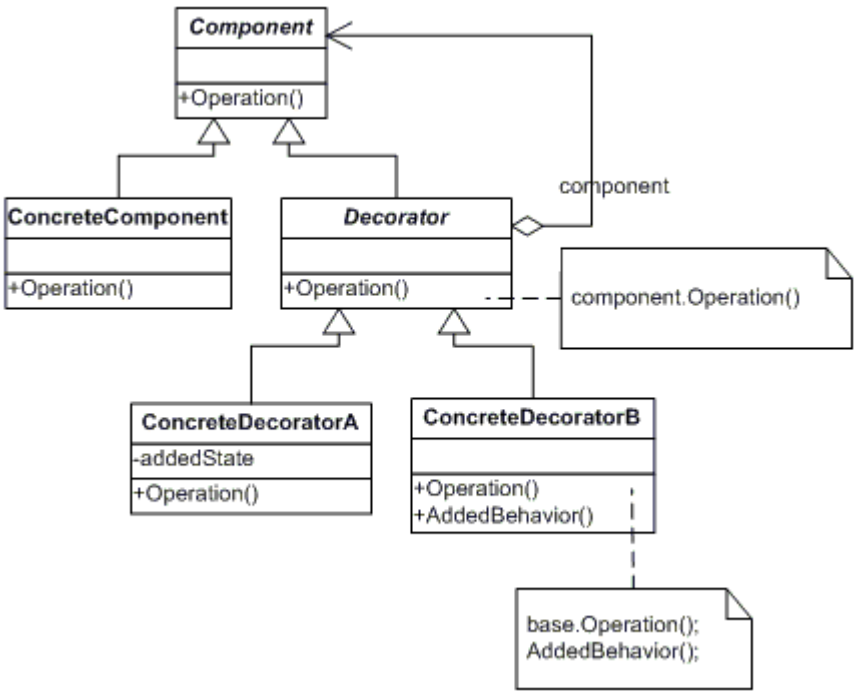
Definition

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

动态的为对象附加(attach)责任(responsibilities)。装饰者模式提供一个弹性的继承替代方案来扩增功能。

Frequency of use: 1 2 3 4 5 medium high

UML class diagram



Participants

The classes and/or objects participating in the Decorator pattern are:

- **Component (LibraryItem)**
  - defines the interface for objects that can have responsibilities added to them dynamically.
- **ConcreteComponent (Book, Video)**
  - defines an object to which additional responsibilities can be attached.
- **Decorator (Decorator)**
  - maintains a reference to a Component object and defines an interface that conforms to Component's interface.

- **ConcreteDecorator (Borrowable)**
  - adds responsibilities to the component.

Sample code in C#

- This structural code demonstrates the Decorator pattern which dynamically adds extra functionality to an existing object.

```
// Decorator pattern -- Structural example

using System;

// "Component"

abstract class Component
{
  // Methods
  abstract public void Operation();
}

// "ConcreteComponent"

class ConcreteComponent : Component
{
  // Methods
  override public void Operation()
  {
    Console.WriteLine("ConcreteComponent.Operation()");
  }
}

// "Decorator"

abstract class Decorator : Component
{
  // Fields
  protected Component component;

  // Methods
  public void SetComponent( Component component )
  {
    this.component = component;
  }
```

```
  override public void Operation()
  {
    if( component != null )
      component.Operation();
  }
}

// "ConcreteDecoratorA"

class ConcreteDecoratorA : Decorator
{
  // Fields
  private string addedState;

  // Methods
  override public void Operation()
  {
    base.Operation();
    addedState = "new state";
    Console.WriteLine("ConcreteDecoratorA.Operation()");
  }
}

// "ConcreteDecoratorB"

class ConcreteDecoratorB : Decorator
{
  // Methods
  override public void Operation()
  {
    base.Operation();
    AddedBehavior();
    Console.WriteLine("ConcreteDecoratorB.Operation()");
  }

  void AddedBehavior()
  {
  }
}

/// <summary>
/// Client test
/// </summary>
```

```
public class Client
{
  public static void Main( string[] args )
  {
    // Create ConcreteComponent and two Decorators
    ConcreteComponent c = new ConcreteComponent();
    ConcreteDecoratorA d1 = new ConcreteDecoratorA();
    ConcreteDecoratorB d2 = new ConcreteDecoratorB();

    // Link decorators
    d1.SetComponent( c );
    d2.SetComponent( d1 );

    d2.Operation();
  }
}
```

Output

```
ConcreteComponent.Operation()
ConcreteDecoratorA.Operation()
ConcreteDecoratorB.Operation()
```

- This real-world code demonstrates the Decorator pattern in which 'borrowable' functionality is added to existing library items (books and videos).

```
// Decorator pattern -- Real World example
using System;
using System.Collections;

// "Component"

abstract class LibraryItem
{
  // Fields
  private int numCopies;
  // Properties

  public int NumCopies
  {
    get{ return numCopies; }
    set{ numCopies = value; }
  }

  // Methods
```

```
  public abstract void Display();
}

// "ConcreteComponent"

class Book : LibraryItem
{
  // Fields
  private string author;
  private string title;

  // Constructors
  public Book(string author,string title,int numCopies)
  {
    this.author = author;
    this.title = title;
    this.NumCopies = numCopies;
  }

  // Methods
  public override void Display()
  {
    Console.WriteLine( "\nBook ------ " );
    Console.WriteLine( " Author: {0}", author );
    Console.WriteLine( " Title: {0}", title );
    Console.WriteLine( " # Copies: {0}", NumCopies );
  }
}

// "ConcreteComponent"

class Video : LibraryItem
{
  // Fields
  private string director;
  private string title;
  private int playTime;

  // Constructor
  public Video( string director, string title,
                        int numCopies, int playTime )
  {
    this.director = director;
    this.title = title;
```

```
      this.NumCopies = numCopies;
      this.playTime = playTime;
    }

    // Methods
    public override void Display()
    {
      Console.WriteLine( "\nVideo ----- " );
      Console.WriteLine( " Director: {0}", director );
      Console.WriteLine( " Title: {0}", title );
      Console.WriteLine( " # Copies: {0}", NumCopies );
      Console.WriteLine( " Playtime: {0}", playTime );
    }
  }

  // "Decorator"

  abstract class Decorator : LibraryItem
  {
    // Fields
    protected LibraryItem libraryItem;

    // Constructors
    public Decorator ( LibraryItem libraryItem )
    {
      this.libraryItem = libraryItem;
    }

    // Methods
    public override void Display()
    {
      libraryItem.Display();
    }
  }

  // "ConcreteDecorator"

  class Borrowable : Decorator
  {
    // Fields
    protected ArrayList borrowers = new ArrayList();

    // Constructors
    public Borrowable( LibraryItem libraryItem )
```

```
                                : base( libraryItem ) {}

  // Methods
  public void BorrowItem( string name )
  {
    borrowers.Add( name );
    libraryItem.NumCopies--;
  }

  public void ReturnItem( string name )
  {
    borrowers.Remove( name );
     libraryItem.NumCopies++;
  }

  public override void Display()
  {
    base.Display();
    foreach( string borrower in borrowers )
      Console.WriteLine( " borrower: {0}", borrower );
  }
}

/// <summary>
///  DecoratorApp test
/// </summary>
public class DecoratorApp
{
  public static void Main( string[] args )
  {
    // Create book and video and display
    Book book = new Book( "Schnell", "My Home", 10 );
    Video video = new Video( "Spielberg",
                             "Schindler's list", 23, 60 );
    book.Display();
    video.Display();

    // Make video borrowable, then borrow and display
    Console.WriteLine( "\nVideo made borrowable:" );
    Borrowable borrowvideo = new Borrowable( video );
    borrowvideo.BorrowItem( "Cindy Lopez" );
    borrowvideo.BorrowItem( "Samuel King" );

    borrowvideo.Display();
```

```
    }
}
```

Output

```
Book ------
Author: Schnell
Title: My Home
# Copies: 10

Video -----
Director: Spielberg
Title: Schindler's list
# Copies: 23
Playtime: 60

Video made borrowable:

Video -----
Director: Spielberg
Title: Schindler's list
# Copies: 21
Playtime: 60
borrower: Cindy Lopez
borrower: Samuel King
```

10 Facade – 外观模式

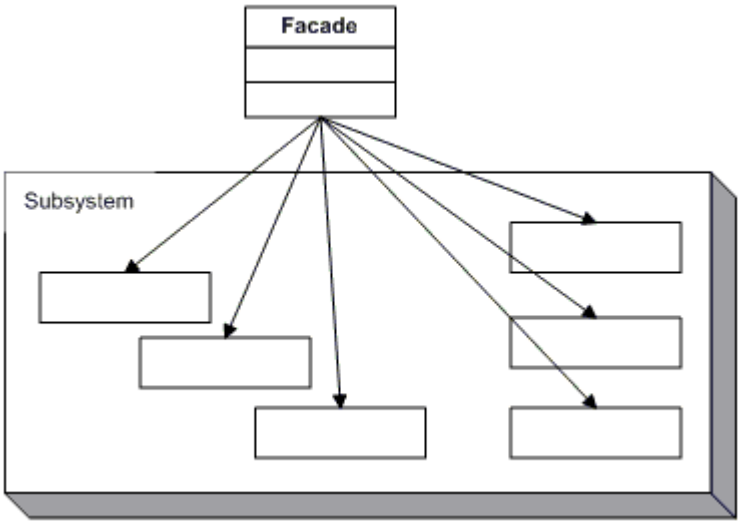
Definition

Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

提供一个子系统接口一组统一的接口；外观模式定义一个比较高阶(higher-level)的接口让子系统容易使用。

Frequency of use: 1 2 3 4 5 high

UML class diagram



Participants

The classes and/or objects participating in the Façade pattern are:

- **Facade (MortgageApplication)**
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.
- **Subsystem classes (Bank, Credit, Loan)**
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade and keep no reference to it.

Sample code in C#

- This structural code demonstrates the Facade pattern which provides a simplified and uniform interface to a large subsystem of classes.

```
// Facade pattern -- Structural example
using System;

// "SubSystem"

class SubSystemOne
{
  public void MethodOne()
  {
    Console.WriteLine("SubSystemOne Method");
  }
}
```

```
class SubSystemTwo
{
  public void MethodTwo()
  {
    Console.WriteLine("SubSystemTwo Method");
  }
}

class SubSystemThree
{
  public void MethodThree()
  {
    Console.WriteLine("SubSystemThree Method");
  }
}

class SubSystemFour
{
  public void MethodFour()
  {
    Console.WriteLine("SubSystemFour Method");
  }
}

// "Facade"

class Facade
{
  // Fields
  SubSystemOne one;
  SubSystemTwo two;
  SubSystemThree three;
  SubSystemFour four;

  // Constructors
  public Facade()
  {
    one = new SubSystemOne();
    two = new SubSystemTwo();
    three = new SubSystemThree();
    four = new SubSystemFour();
  }

  // Methods
```

```
  public void MethodA()
  {
    Console.WriteLine( "\nMethodA() ---- " );
    one.MethodOne();
    two.MethodTwo();
    four.MethodFour();
  }

  public void MethodB()
  {
    Console.WriteLine( "\nMethodB() ---- " );
    two.MethodTwo();
    three.MethodThree();
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main(string[] args)
  {
    // Create and call Facade
    Facade f = new Facade();
    f.MethodA();
    f.MethodB();

  }
}
```

Output

```
MethodA() ----
SubSystemOne Method
SubSystemTwo Method
SubSystemFour Method

MethodB() ----
SubSystemTwo Method
SubSystemThree Method
```

- This real-world code demonstrates the Facade pattern as a MortgageApplication object which provides a simplified interface to a large subsystem of classes measuring the creditworthyness of an applicant.

```
// Facade pattern -- Real World example
```

```
using System;

// "SubSystem ClassA"

class Bank
{
  // Methods
  public bool SufficientSavings( Customer c )
  {
    Console.WriteLine("Check bank for {0}", c.Name );
    return true;
  }
}

// "SubSystem ClassB"

class Credit
{
  // Methods
  public bool GoodCredit( int amount, Customer c )
  {
    Console.WriteLine( "Check credit for {0}", c.Name );
    return true;
  }
}

// "SubSystem ClassC"

class Loan
{
  // Methods
  public bool GoodLoan( Customer c )
  {
    Console.WriteLine( "Check loan for {0}", c.Name );
    return true;
  }
}

class Customer
{
  // Fields
  private string name;

  // Constructors
```

```
  public Customer( string name )
  {
    this.name = name;
  }

  // Properties
  public string Name
  {
    get{ return name; }
  }
}

// "Facade"

class MortgageApplication
{
  // Fields
  int amount;
  private Bank bank = new Bank();
  private Loan loan = new Loan();
  private Credit credit = new Credit();

  // Constructors
  public MortgageApplication( int amount )
  {
    this.amount = amount;
  }

  // Methods
  public bool IsEligible( Customer c )
  {
    // Check creditworthyness of applicant
    if( !bank.SufficientSavings( c ) )
      return false;
    if( !loan.GoodLoan( c ) )
      return false;
    if( !credit.GoodCredit( amount, c ))
      return false;

    return true;
  }
}
/// <summary>
/// Facade Client
```

```
/// </summary>
public class FacadeApp
{
  public static void Main(string[] args)
  {
    // Create Facade
    MortgageApplication mortgage =
                  new MortgageApplication( 125000 );
    // Call subsystem through Facade
    mortgage.IsEligible(
              new Customer( "Gabrielle McKinsey" ) );

  }
}
```

Output

```
Check bank for Gabrielle McKinsey
Check loan for Gabrielle McKinsey
Check credit for Gabrielle McKinsey
```

11 Flyweight –轻量模式

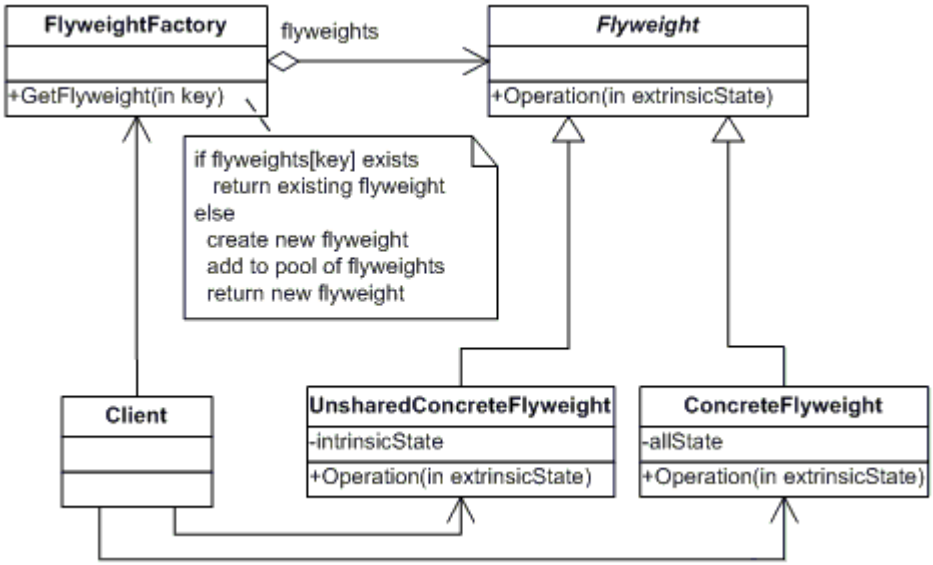
Definition

Use sharing to support large numbers of fine-grained objects efficiently.

使用共享的方式以有效支持大量微小(fine-grained)对象。

Frequency of use: 1 2 3 4 5 medium low

UML class diagram



Participants

The classes and/or objects participating in the Flyweight pattern are:

- **Flyweight (Character)**
  - declares an interface through which flyweights can receive and act on extrinsic state.
- **ConcreteFlyweight (CharacterA, CharacterB, ..., CharacterZ)**
  - implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic, that is, it must be independent of the ConcreteFlyweight object's context.
- **UnsharedConcreteFlyweight (not used)**
  - not all Flyweight subclasses need to be shared. The Flyweight interface *enables* sharing, but it doesn't enforce it. It is common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).
- **FlyweightFactory (CharacterFactory)**
  - creates and manages flyweight objects
  - ensures that flyweight are shared properly. When a client requests a flyweight, the FlyweightFactory objects supplies an existing instance or creates one, if none exists.
- **Client (FlyweightApp)**
  - maintains a reference to flyweight(s).
  - computes or stores the extrinsic state of flyweight(s).

Sample code in C#

- This structural code demonstrates the Flyweight pattern in which a relatively small number of objects is shared many times by different clients.

```
// Flyweight pattern -- Structural example
using System;
using System.Collections;

// "FlyweightFactory"

class FlyweightFactory
{
  // Fields
  private Hashtable flyweights = new Hashtable();

  // Constructors
  public FlyweightFactory()
  {
    flyweights.Add("X", new ConcreteFlyweight());
    flyweights.Add("Y", new ConcreteFlyweight());
    flyweights.Add("Z", new ConcreteFlyweight());
  }

  // Methods
  public Flyweight GetFlyweight(string key)
  {
    return((Flyweight)flyweights[ key ]);
  }
}

// "Flyweight"

abstract class Flyweight
{
  // Methods
  abstract public void Operation( int extrinsicstate );
}

// "ConcreteFlyweight"

class ConcreteFlyweight : Flyweight
{
  // Methods
  override public void Operation( int extrinsicstate )
  {
```

```
    Console.WriteLine("ConcreteFlyweight: {0}",
                                        extrinsicstate );
  }
}

// "UnsharedConcreteFlyweight"

class UnsharedConcreteFlyweight : Flyweight
{
  // Methods
  override public void Operation( int extrinsicstate )
  {
    Console.WriteLine("UnsharedConcreteFlyweight: {0}",
                                        extrinsicstate );
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Arbitrary extrisic state
    int extrinsicstate = 22;

    FlyweightFactory f = new FlyweightFactory();

    // Work with different flyweight instances
    Flyweight fx = f.GetFlyweight("X");
    fx.Operation( --extrinsicstate );

    Flyweight fy = f.GetFlyweight("Y");
    fy.Operation( --extrinsicstate );

    Flyweight fz = f.GetFlyweight("Z");
    fz.Operation( --extrinsicstate );

    UnsharedConcreteFlyweight fu = new
                      UnsharedConcreteFlyweight();
    fu.Operation( --extrinsicstate );

  }
```

```
}
```

output’

```
ConcreteFlyweight: 21
ConcreteFlyweight: 20
ConcreteFlyweight: 19
UnsharedConcreteFlyweight: 18
```

- This real-world code demonstrates the Flyweight pattern in which a relatively small number of Character objects is shared many times by a document that has potentially many characters.

```
// Flyweight pattern -- Real World example
using System;
using System.Collections;

// "FlyweightFactory"

class CharacterFactory
{
  // Fields
  private Hashtable characters = new Hashtable();

  // Methods
  public Character GetCharacter( char key )
  {
    // Uses "lazy initialization"
    Character character = (Character)characters[ key ];
    if( character == null )
    {
      switch( key )
      {
        case 'A': character = new CharacterA(); break;
        case 'B': character = new CharacterB(); break;
        //...
        case 'Z': character = new CharacterZ(); break;
      }
      characters.Add( key, character );
    }
    return character;
  }
}

// "Flyweight"
```

```
abstract class Character
{
  // Fields
  protected char symbol;
  protected int width;
  protected int height;
  protected int ascent;
  protected int descent;
  protected int pointSize;

  // Methods
  public abstract void Draw( int pointSize );
}

// "ConcreteFlyweight"

class CharacterA : Character
{
  // Constructors
  public CharacterA( )
  {
    this.symbol = 'A';
    this.height = 100;
    this.width = 120;
    this.ascent = 70;
    this.descent = 0;
  }

  // Methods
  public override void Draw( int pointSize )
  {
    this.pointSize = pointSize;
    Console.Write( this.symbol );
  }
}

// "ConcreteFlyweight"

class CharacterB : Character
{
  // Constructors
  public CharacterB()
  {
    this.symbol = 'B';
```

```
      this.height = 100;
      this.width = 140;
      this.ascent = 72;
      this.descent = 0;
    }

    // Methods
    public override void Draw( int pointSize )
    {
      this.pointSize = pointSize;
      Console.Write( this.symbol );
    }
  }

  // ... C, D, E, etc.

  // "ConcreteFlyweight"

  class CharacterZ : Character
  {
    // Constructors
    public CharacterZ( )
    {
      this.symbol = 'Z';
      this.height = 100;
      this.width = 100;
      this.ascent = 68;
      this.descent = 0;
    }

    // Methods
    public override void Draw( int pointSize )
    {
      this.pointSize = pointSize;
      Console.Write( this.symbol );
    }
  }

  /// <summary>
  ///  FlyweightApp test
  /// </summary>
  public class FlyweightApp
  {
    public static void Main( string[] args )
```

```
  {
    // Build a document with text
    char[] document = {'A','B','Z','Z','A','A'};

     CharacterFactory f = new CharacterFactory();

    // extrinsic state
    int pointSize = 12;

    // For each character use a flyweight object
    foreach( char c in document )
    {
      Character character = f.GetCharacter( c );
      character.Draw( pointSize );
    }
  }
}
```

Output

ABZZAA

12 Proxy – 代理模式

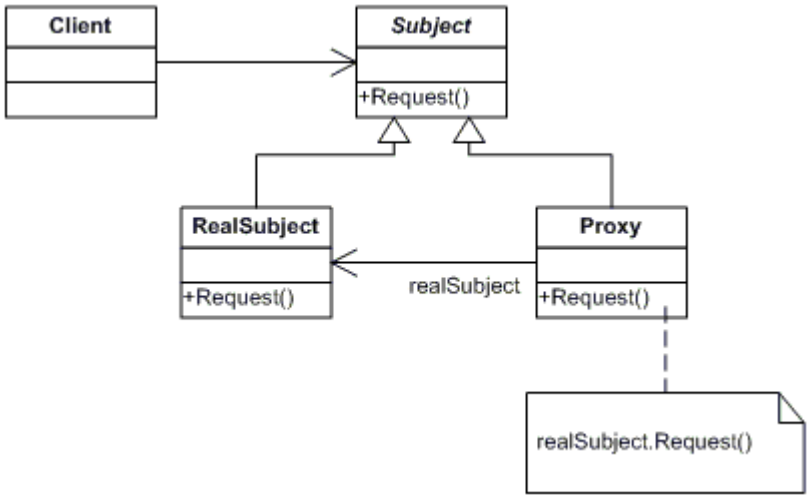
Definition

Provide a surrogate or placeholder for another object to control access to it.

为另一个对象的存取(access)提供一个代理人(surrogate)或(placeholder)。

Frequency of use: 1 2 3 4 5 high

UML class diagram



Participants

The classes and/or objects participating in the Proxy pattern are:

- **Proxy (MathProxy)**
  - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - provides an interface identical to Subject's so that a proxy can be substituted for for the real subject.
  - controls access to the real subject and may be responsible for creating and deleting it.
  - other responsibilites depend on the kind of proxy:
    - *remote proxies* are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
    - *virtual proxies* may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real images's extent.
    - *protection proxies* check that the caller has the access permissions required to perform a request.
- **Subject (IMath)**
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- **RealSubject (Math)**
  - defines the real object that the proxy represents.

Sample code in C#

This structural code demonstrates the Proxy pattern which provides a representative object (proxy) that controls access to another similar object.

```
// Proxy pattern -- Structural example
using System;

// "Subject"

abstract class Subject
{
  // Methods
  abstract public void Request();
}

// "RealSubject"

class RealSubject : Subject
{
  // Methods
  override public void Request()
  {
    Console.WriteLine("Called RealSubject.Request()");
  }
}

// "Proxy"

class Proxy : Subject
{
  // Fields
  RealSubject realSubject;

  // Methods
  override public void Request()
  {
    // Uses "lazy initialization"
    if( realSubject == null )
      realSubject = new RealSubject();

    realSubject.Request();
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
```

```
{
  public static void Main( string[] args )
  {
    // Create proxy and request a service
    Proxy p = new Proxy();
    p.Request();

  }
}
```

output

```
Called RealSubject.Request()
```

- This real-world code demonstrates the Remote Proxy pattern which provides a representative object that controls access to another object in a different AppDomain.

```
// Proxy pattern -- Real World example
using System;
using System.Runtime.Remoting;

// "Subject"

public interface IMath
{
  // Methods
  double Add( double x, double y );
  double Sub( double x, double y );
  double Mul( double x, double y );
  double Div( double x, double y );
}

// "RealSubject"

class Math : MarshalByRefObject, IMath
{
  // Methods
  public double Add( double x, double y ){ return x + y; }
  public double Sub( double x, double y ){ return x - y; }
  public double Mul( double x, double y ){ return x * y; }
  public double Div( double x, double y ){ return x / y; }
}

// Remote "Proxy Object"

class MathProxy : IMath
{
```

```
  // Fields
  Math math;
  // Constructors
  public MathProxy()
  {
    // Create Math instance in a different AppDomain
    AppDomain ad = System.AppDomain.CreateDomain(
                                    "MathDomain",null, null );
    ObjectHandle o =
      ad.CreateInstance("Proxy_RealWorld", "Math",  false,
      System.Reflection.BindingFlags.CreateInstance,
      null, null, null,null,null );
    math = (Math) o.Unwrap();
  }

  // Methods
  public double Add( double x, double y )
  {
    return math.Add(x,y);
  }

  public double Sub( double x, double y )
  {
    return math.Sub(x,y);
  }

  public double Mul( double x, double y )
  {
    return math.Mul(x,y);
  }

  public double Div( double x, double y )
  {
    return math.Div(x,y);
  }
}

/// <summary>
/// ProxyApp test
/// </summary>
public class ProxyApp
{
  public static void Main( string[] args )
  {
```

```
        // Create math proxy
        MathProxy p = new MathProxy();

        // Do the math
        Console.WriteLine( "4 + 2 = {0}", p.Add( 4, 2 ) );
        Console.WriteLine( "4 - 2 = {0}", p.Sub( 4, 2 ) );
        Console.WriteLine( "4 * 2 = {0}", p.Mul( 4, 2 ) );
        Console.WriteLine( "4 / 2 = {0}", p.Div( 4, 2 ) );
    }
}
```

Output

```
4 + 2 = 6
4 - 2 = 2
4 * 2 = 8
4 / 2 = 2
```

第三部分 **Behavioral Patterns**

13 Chain of Responsibility –责任链模式

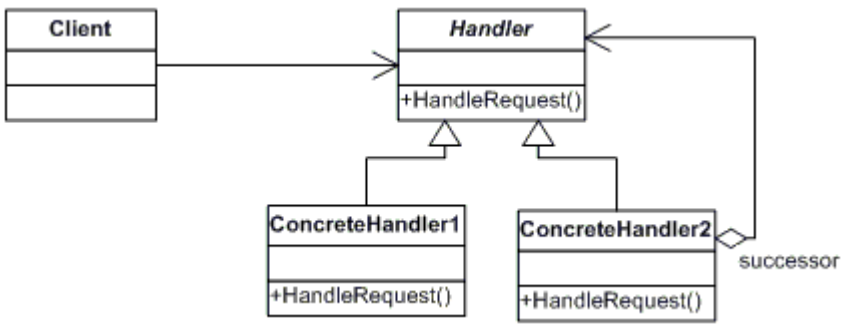
Definition

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

经由让更多对象有机会控制请求以避免请求送出者与接收者间的耦合性，将多个接收者对象链并循此链传递请求直到请求被处理为止。

Frequency of use: 1 2 3 4 5 medium

UML class diagram



Participants

The classes and/or objects participating in this pattern are:

- **Handler (Approver)**
  - defines an interface for handling the requests
  - (optional) implements the successor link
- **ConcreteHandler (Director, VicePresident, President)**
  - handles requests it is responsible for
  - can access its successor
  - if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor
- **Client (ChainApp)**
  - initiates the request to a ConcreteHandler object on the chain

Sample code in C#

This structural code demonstrates the Chain of Responsibility pattern in which several linked objects (the Chain) are offered the opportunity to respond to a request or hand it off to the object next in line.

```
// Chain of Responsibility pattern -- Structural example
using System;

// "Handler"

abstract class Handler
{
  // Fields
  protected Handler successor;

  // Methods
  public void SetSuccessor( Handler successor )
  {
    this.successor = successor;
  }
  abstract public void HandleRequest( int request );
}

// "ConcreteHandler1"

class ConcreteHandler1 : Handler
{
  // Methods
  override public void HandleRequest( int request )
```

```
  {
    if( request >= 0 && request < 10 )
      Console.WriteLine("{0} handled request {1}",
        this, request );
    else
      if( successor != null )
        successor.HandleRequest( request );
  }
}

// "ConcreteHandler2"

class ConcreteHandler2 : Handler
{
  // Methods
  override public void HandleRequest( int request )
  {
    if( request >= 10 && request < 20 )
      Console.WriteLine("{0} handled request {1}",
        this, request );
    else
      if( successor != null )
        successor.HandleRequest( request );
  }
}

// "ConcreteHandler3"

class ConcreteHandler3 : Handler
{
  // Methods
  override public void HandleRequest( int request )
  {
    if( request >= 20 && request < 30 )
      Console.WriteLine("{0} handled request {1}",
        this, request );
    else
      if( successor != null )
        successor.HandleRequest( request );
  }
}

// "Request"
```

```
class Request
{
  // Fields
  private int iRequestType;
  private string strRequestParameters;

  // Constructors
  public Request(int requestType, string requestParameters)
  {
    iRequestType = requestType;
    strRequestParameters = requestParameters;
  }

  // Properties
  public int RequestType
  {
    get{ return iRequestType; }
    set{iRequestType = value; }
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Setup Chain of Responsibility
    Handler h1 = new ConcreteHandler1();
    Handler h2 = new ConcreteHandler2();
    Handler h3 = new ConcreteHandler3();
    h1.SetSuccessor(h2);
    h2.SetSuccessor(h3);

    // Generate and process request
    int[] requests = { 2, 5, 14, 22, 18, 3, 27, 20 };

    foreach( int request in requests )
      h1.HandleRequest( request );

    }
}
```

output

```
ConcreteHandler1 handled request 2
ConcreteHandler1 handled request 5
ConcreteHandler2 handled request 14
ConcreteHandler3 handled request 22
ConcreteHandler2 handled request 18
ConcreteHandler1 handled request 3
ConcreteHandler3 handled request 27
ConcreteHandler3 handled request 20
```

- This real-world code demonstrates the Chain of Responsibility pattern in which several linked managers and executives can respond to a purchase request or hand it off to a superior. Each position has can have its own set of rules which orders they can approve.

```
// Chain of Responsibility pattern -- Real World example
using System;

// "Handler"

abstract class Approver
{
  // Fields
  protected string name;
  protected Approver successor;

  // Constructors
  public Approver( string name )
  {
    this.name = name;
  }

  // Methods
  public void SetSuccessor( Approver successor )
  {
    this.successor = successor;
  }

  abstract public void ProcessRequest(
                              PurchaseRequest request );
}

// "ConcreteHandler"

class Director : Approver
{
```

```
  // Constructors
  public Director ( string name ) : base( name ) {}

  // Methods
  override public void ProcessRequest(
                              PurchaseRequest request )
  {
    if( request.Amount < 10000.0 )
      Console.WriteLine( "{0} {1} approved request# {2}",
        this, name, request.Number);
    else
      if( successor != null )
        successor.ProcessRequest( request );
  }
}

// "ConcreteHandler"

class VicePresident : Approver
{
  // Constructors
  public VicePresident ( string name ) : base( name ) {}

  // Methods
  override public void ProcessRequest(
                              PurchaseRequest request )
  {
    if( request.Amount < 25000.0 )
      Console.WriteLine( "{0} {1} approved request# {2}",
        this, name, request.Number);
    else
      if( successor != null )
        successor.ProcessRequest( request );
  }
}

// "ConcreteHandler"

class President : Approver
{
  // Constructors
  public President ( string name ) : base( name ) {}
  // Methods
  override public void ProcessRequest(
```

```
                                PurchaseRequest request )
  {
    if( request.Amount < 100000.0 )
      Console.WriteLine( "{0} {1} approved request# {2}",
        this, name, request.Number);
    else
      Console.WriteLine( "Request# {0} requires " +
        "an executive meeting!", request.Number );
  }
}

// Request details

class PurchaseRequest
{
  // Member Fields
  private int number;
  private double amount;
  private string purpose;

  // Constructors
  public PurchaseRequest( int number,
            double amount, string purpose )
  {
    this.number = number;
    this.amount = amount;
    this.purpose = purpose;
  }

  // Properties
  public double Amount
  {
    get{ return amount; }
    set{ amount = value; }
  }

  public string Purpose
  {
    get{ return purpose; }
    set{ purpose = value; }
  }

  public int Number
  {
```

```
    get{ return number; }
    set{ number = value; }
  }
}

/// <summary>
///  ChainApp Application
/// </summary>
public class ChainApp
{
  public static void Main( string[] args )
  {
    // Setup Chain of Responsibility
    Director Larry = new Director( "Larry" );
    VicePresident Sam = new VicePresident( "Sam" );
    President Tammy = new President( "Tammy" );
    Larry.SetSuccessor( Sam );
    Sam.SetSuccessor( Tammy );

    // Generate and process different requests
    PurchaseRequest rs = new PurchaseRequest(
                         2034, 350.00, "Supplies" );
    Larry.ProcessRequest( rs );

    PurchaseRequest rx = new PurchaseRequest(
                         2035, 32590.10, "Project X" );
    Larry.ProcessRequest( rx );

    PurchaseRequest ry = new PurchaseRequest(
                         2036, 122100.00, "Project Y" );
    Larry.ProcessRequest( ry );

  }
}
```

output

```
Director Larry approved request# 2034
President Tammy approved request# 2035
Request# 2036 requires an executive meeting!
```

14 Command – 指令模式

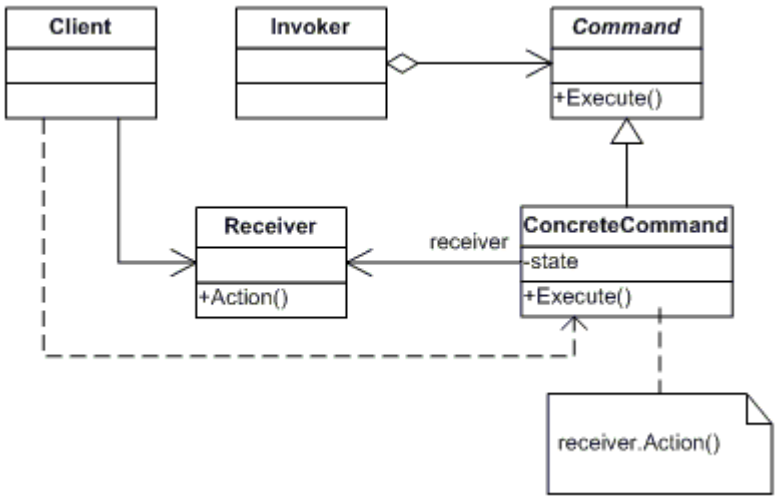
Definition

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

将请求封装成一个对象，让你可以以不同的请求、排列(queue)或登录请求参数化使用端，并支持可取消(undoable)操作

Frequency of use: 1 2 3 4 5 medium high

UML class diagram



Participants

The classes and/or objects participating in this pattern are:

- **Command (Command)**
  - declares an interface for executing an operation
- **ConcreteCommand (CalculatorCommand)**
  - defines a binding between a Receiver object and an action
  - implements Execute by invoking the corresponding operation(s) on Receiver
- **Client (CommandApp)**
  - creates a ConcreteCommand object and sets its receiver
- **Invoker (User)**
  - asks the command to carry out the request
- **Receiver (Calculator)**
  - knows how to perform the operations associated with carrying out the request.

Sample code in C#

This structural code demonstrates the Command pattern which stores requests as objects allowing clients to execute or playback the requests.

```
// Command pattern -- Structural example
```

```
using System;

// "Command"

abstract class Command
{
  // Fields
  protected Receiver receiver;

  // Constructors
  public Command( Receiver receiver )
  {
    this.receiver = receiver;
  }

  // Methods
  abstract public void Execute();
}

// "ConcreteCommand"

class ConcreteCommand : Command
{
  // Constructors
  public ConcreteCommand( Receiver receiver ) :
                                  base ( receiver ) {}

  // Methods
  public override void Execute()
  {
    receiver.Action();
  }
}

// "Receiver"

class Receiver
{
  // Methods
  public void Action()
  {
    Console.WriteLine("Called Receiver.Action()");
  }
}
```

```
// "Invoker"

class Invoker
{
  // Fields
  private Command command;

  // Methods
  public void SetCommand( Command command )
  {
    this.command = command;
  }

  public void ExecuteCommand()
  {
    command.Execute();
  }
}

/// <summary>
///  Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Create receiver, command, and invoker
    Receiver r = new Receiver();
    Command c = new ConcreteCommand( r );
    Invoker i = new Invoker();

    // Set and execute command
    i.SetCommand(c);
    i.ExecuteCommand();
  }
}
```

ouput

```
Called Receiver.Action()
```

- This real-world code demonstrates the Command pattern used in a simple calculator with unlimited number of undo's and redo's. Note that in C# the word 'operator' is a keyword. Prefixing it with '@' allows using it as an identifier.

```
// Command pattern -- Real World example
using System;
using System.Collections;

// "Command"

abstract class Command
{
  // Methods
  abstract public void Execute();
  abstract public void UnExecute();
}

// "ConcreteCommand"

class CalculatorCommand : Command
{
  // Fields
  char @operator;
  int operand;
  Calculator calculator;

  // Constructor
  public CalculatorCommand( Calculator calculator,
                            char @operator, int operand )
  {
    this.calculator = calculator;
    this.@operator = @operator;
    this.operand = operand;
  }

  // Properties
  public char Operator
  {
    set{ @operator = value; }
  }

  public int Operand
  {
    set{ operand = value; }
  }

  // Methods
  override public void Execute()
```

```
  {
    calculator.Operation( @operator, operand );
  }

  override public void UnExecute()
  {
     calculator.Operation( Undo( @operator ), operand );
  }

  // Private helper function
  private char Undo( char @operator )
  {
    char undo = ' ';
    switch( @operator )
    {
      case '+': undo = '-'; break;
      case '-': undo = '+'; break;
      case '*': undo = '/'; break;
      case '/': undo = '*'; break;
    }
    return undo;
  }
}

// "Receiver"

class Calculator
{
  // Fields
  private int total = 0;

  // Methods
  public void Operation( char @operator, int operand )
  {
    switch( @operator )
    {
      case '+': total += operand; break;
      case '-': total -= operand; break;
      case '*': total *= operand; break;
      case '/': total /= operand; break;
    }
    Console.WriteLine( "Total = {0} (following {1} {2})",
                       total, @operator, operand );
  }
```

```
}

// "Invoker"

class User
{
  // Fields
  private Calculator calculator = new Calculator();
  private ArrayList commands = new ArrayList();
  private int current = 0;

  // Methods
  public void Redo( int levels )
  {
    Console.WriteLine( "---- Redo {0} levels ", levels );
    // Perform redo operations
    for( int i = 0; i < levels; i++ )
      if( current < commands.Count - 1 )
        ((Command)commands[ current++ ]).Execute();
  }

  public void Undo( int levels )
  {
    Console.WriteLine( "---- Undo {0} levels ", levels );
    // Perform undo operations
    for( int i = 0; i < levels; i++ )
      if( current > 0 )
        ((Command)commands[ --current ]).UnExecute();
  }

  public void Compute( char @operator, int operand )
  {
    // Create command operation and execute it
    Command command = new CalculatorCommand(
                  calculator, @operator, operand );
    command.Execute();

    // Add command to undo list
    commands.Add( command );
    current++;
  }
}

/// <summary>
```

```
/// CommandApp test
/// </summary>
public class CommandApp
{
  public static void Main( string[] args )
  {
    // Create user and let her compute
    User user = new User();

    user.Compute( '+', 100 );
    user.Compute( '-', 50 );
    user.Compute( '*', 10 );
    user.Compute( '/', 2 );

    // Undo and then redo some commands
    user.Undo( 4 );
    user.Redo( 3 );
  }
}
```

15 Interpreter – 翻译器模式

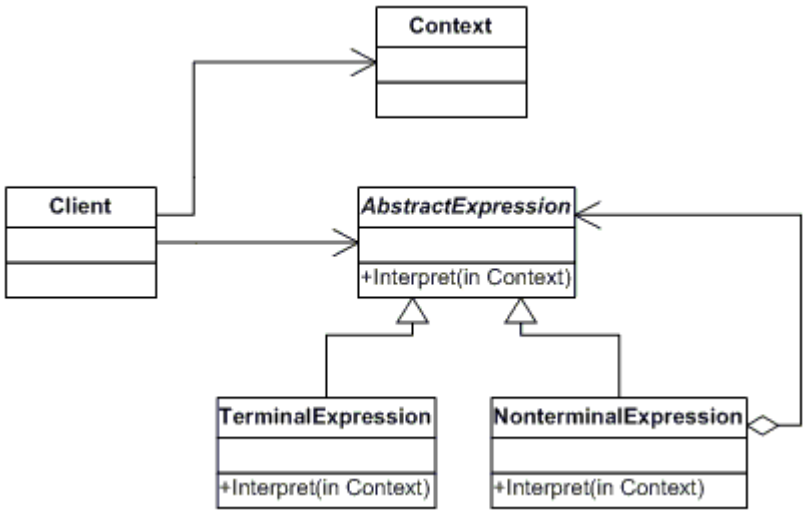
Definition

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

给定一种语言，定义其文法(grammar)的表现法(represention)及其翻译器，这个翻译器使用这个表现法翻译语言中的命题(sentence)。

Frequency of use: 1 2 3 4 5 low

UML class diagram



Participants

The classes and/or objects participating in this pattern are:

- **AbstractExpression (Expression)**
  - declares an interface for executing an operation
- **TerminalExpression (ThousandExpression, HundredExpression, TenExpression, OneExpression)**
  - implements an Interpret operation associated with terminal symbols in the grammar.
  - an instance is required for every terminal symbol in the sentence.
- **NonterminalExpression (not used)**
  - one such class is required for every rule R ::= R1R2...Rn in the grammar
  - maintains instance variables of type AbstractExpression for each of the symbols R1 through Rn.
  - implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R1 through Rn.
- **Context (Context)**
  - contains information that is global to the interpreter
- **Client (InterpreterApp)**
  - builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes
  - invokes the Interpret operation

Sample code in C#

This structural code demonstrates the Interpreter patterns, which using a defined

grammer, provides the interpreter that processes parsed statements.

```
// Interpreter pattern -- Structural example
using System;
using System.Collections;

// "Context"

class Context
{
}

// "AbstractExpression"

abstract class AbstractExpression
{
  public abstract void Interpret( Context context );
}

// "TerminalExpression"

class TerminalExpression : AbstractExpression
{
  public override void Interpret( Context context )
  {
    Console.WriteLine( "Called Terminal.Interpret()" );
  }
}

// "NonterminalExpression"

class NonterminalExpression : AbstractExpression
{
  public override void Interpret( Context context )
  {
    Console.WriteLine( "Called Nonterminal.Interpret()" );
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
```

```
  {
    Context c = new Context();

    // Usually a tree
    ArrayList l = new ArrayList();

    // Populate 'abstract syntax tree'
    l.Add(new TerminalExpression());
    l.Add(new NonterminalExpression());
    l.Add(new TerminalExpression());
    l.Add(new TerminalExpression());

    // Interpret
    foreach( AbstractExpression exp in l )
      exp.Interpret(c);
  }
}
```

output

```
Called Terminal.Interpret()
Called Nonterminal.Interpret()
Called Terminal.Interpret()
Called Terminal.Interpret()
```

- This real-world code demonstrates the Interpreter pattern which is used to convert a Roman numeral to a decimal.

```
// Interpreter pattern -- Real World example
using System;
using System.Text;
using System.Collections;

// "Context"

class Context
{
  // Fields
  private string input;
  private int output;

  // Constructors
  public Context( string input )
  {
    this.input = input;
  }
```

```
  // Properties
  public string Input
  {
    get{ return input; }
    set{ input = value; }
  }

  public int Output
  {
    get{ return output; }
    set{ output = value; }
  }
}

// "AbstractExpression"

abstract class Expression
{
  // Methods
  public void Interpret( Context context )
  {
    if( context.Input.Length == 0 ) return;
    if( context.Input.StartsWith( Nine() ) )
    {
      context.Output += 9 * Multiplier();
      context.Input = context.Input.Substring(2);
    }
    else if( context.Input.StartsWith( Four() ) )
    {
      context.Output += 4 * Multiplier();
      context.Input = context.Input.Substring( 2);
    }
    else if( context.Input.StartsWith( Five() ) )
    {
      context.Output += 5 * Multiplier();
      context.Input = context.Input.Substring( 1);
    }
    while( context.Input.StartsWith( One() ) )
    {
      context.Output += 1 * Multiplier();
      context.Input = context.Input.Substring( 1 );
    }
  }
```

```
  public abstract string One();
  public abstract string Four();
  public abstract string Five();
  public abstract string Nine();
  public abstract int Multiplier();
}

// Thousand checks for the Roman Numeral M
// "TerminalExpression"

class ThousandExpression : Expression
{
  // Methods
  public override string One() { return "M"; }
  public override string Four(){ return " "; }
  public override string Five(){ return " "; }
  public override string Nine(){ return " "; }
  public override int Multiplier() { return 1000; }
}

// Hundred checks C, CD, D or CM
// "TerminalExpression"

class HundredExpression : Expression
{
  // Methods
  public override string One() { return "C"; }
  public override string Four(){ return "CD"; }
  public override string Five(){ return "D"; }
  public override string Nine(){ return "CM"; }
  public override int Multiplier() { return 100; }
}

// Ten checks for X, XL, L and XC
// "TerminalExpression"

class TenExpression : Expression
{
  // Methods
  public override string One() { return "X"; }
  public override string Four(){ return "XL"; }
  public override string Five(){ return "L"; }
  public override string Nine(){ return "XC"; }
  public override int Multiplier() { return 10; }
```

```
}

// One checks for I, II, III, IV, V, VI, VII, VIII, IX
// "TerminalExpression"

class OneExpression : Expression
{
  // Methods
  public override string One() { return "I"; }
  public override string Four(){ return "IV"; }
  public override string Five(){ return "V"; }
  public override string Nine(){ return "IX"; }
  public override int Multiplier() { return 1; }
}

/// <summary>
///  InterpreterApp Test
/// </summary>
public class InterpreterApp
{
  public static void Main( string[] args )
  {
    string roman = "MCMXXVIII";

    Context context = new Context( roman );

    // Build the 'parse tree'
    ArrayList parse = new ArrayList();
    parse.Add(new ThousandExpression());
    parse.Add(new HundredExpression());
    parse.Add(new TenExpression());
    parse.Add(new OneExpression());

    // Interpret
    foreach( Expression exp in parse )
      exp.Interpret( context );

    Console.WriteLine( "{0} = {1}",
                              roman, context.Output );
  }
}
```

Output

```
MCMXXVIII = 1928
```

16 Iterator – 迭代模式

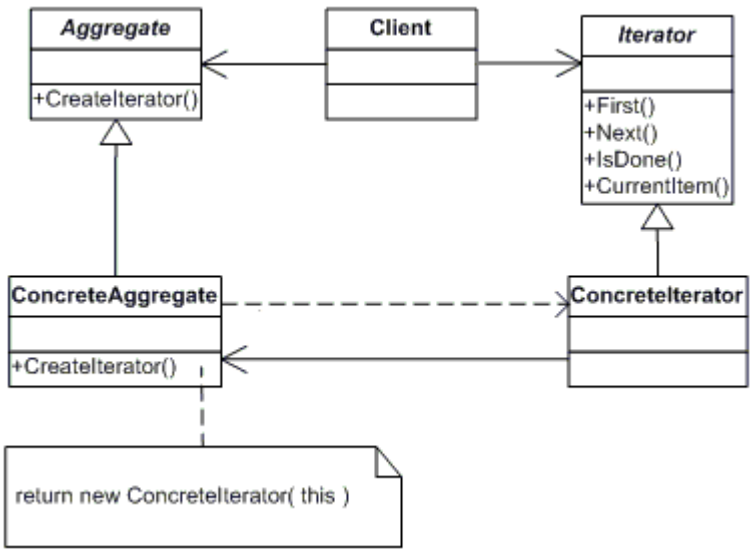
Definition

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

提供循序存取一个聚集(aggregate)对象中的元素(element)；而无须揭露此对象内部的结构

Frequency of use: 1 2 3 4 5 high

UML class diagram



Participants

The classes and/or objects participating in this pattern are:

- **Iterator (AbstractIterator)**
  - defines an interface for accessing and traversing elements.
- **ConcreteIterator (Iterator)**
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
- **Aggregate (AbstractCollection)**
  - defines an interface for creating an Iterator object
- **ConcreteAggregate (Collection)**
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator

Sample code in C#

- This structural code demonstrates the Iterator pattern which provides for a way to traverse (iterate) over a collection of items without detailing the underlying structure of the collection.

```
using System;
using System.Collections;

// "Aggregate"

abstract class Aggregate
{
  // Methods
  public abstract Iterator CreateIterator();
}

// "ConcreteAggregate"

class ConcreteAggregate : Aggregate
{
  // Fields
  private ArrayList items = new ArrayList();

  // Methods
  public override Iterator CreateIterator()
  {
    return new ConcreteIterator( this );
  }

  // Properties
  public int Count
  {
    get{ return items.Count; }
  }
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  // Indexers
  public object this[ int index ]
  {
    get{ return items[ index ]; }
    set{ items.Insert( index, value ); }
  }
}

// "Iterator"
```

```
abstract class Iterator
{
  // Methods
  public abstract object First();
  public abstract object Next();
  public abstract bool IsDone();
  public abstract object CurrentItem();
}

// "ConcreteIterator"

class ConcreteIterator : Iterator
{
  // Fields
  private ConcreteAggregate aggregate;
  private int current = 0;

  // Constructor
  public ConcreteIterator( ConcreteAggregate aggregate )
  {
    this.aggregate = aggregate;
  }

  // Methods
  override public object First()
  {
    return aggregate[ 0 ];
  }

  override public object Next()
  {
    if( current < aggregate.Count-1 )
      return aggregate[ ++current ];
    else
      return null;
  }

  override public object CurrentItem()
  {
    return aggregate[ current ];
  }

  override public bool IsDone()
```

```
  {
    return current >= aggregate.Count ? true : false ;
  }
}

/// <summary>
///  Client test
/// </summary>
public class Client
{
  public static void Main(string[] args)
  {
    ConcreteAggregate a = new ConcreteAggregate();
    a[0] = "Item A";
    a[1] = "Item B";
    a[2] = "Item C";
    a[3] = "Item D";

    // Create Iterator and provide aggregate
    ConcreteIterator i = new ConcreteIterator(a);

    // Iterate over collection
    object item = i.First();

    while( item != null )
    {
      Console.WriteLine( item );
      item = i.Next();
    }
  }
}
```

Output

```
Item A
Item B
Item C
Item D
```

- This real-world code demonstrates the Iterator pattern which is used to iterate over a collection of items and skip a specific number of items each iteration

```
using System;
using System.Collections;
class Item
{
  // Fields
```

```
  string name;

  // Constructors
  public Item( string name )
  {
    this.name = name;
  }

  // Properties
  public string Name
  {
    get{ return name; }
  }
}

// "Aggregate"

abstract class AbstractCollection
{
  abstract public Iterator CreateIterator();
}

// "ConcreteAggregate"

class Collection : AbstractCollection
{
  // Fields
  private ArrayList items = new ArrayList();

  // Methods
  public override Iterator CreateIterator()
  {
    return new Iterator( this );
  }

  // Properties
  public int Count
  {
    get{ return items.Count; }
  }

  // Indexers
  public object this[ int index ]
  {
```

```
    get{ return items[ index ]; }
    set{ items.Add( value ); }
  }
}

// "Iterator"

abstract class AbstractIterator
{
  // Methods
  abstract public Item First();
  abstract public Item Next();
  abstract public bool IsDone();
  abstract public Item CurrentItem();
}

// "ConcreteIterator"

class Iterator : AbstractIterator
{
  // Fields
  private Collection collection;
  private int current = 0;
  private int step = 1;

  // Constructor
  public Iterator( Collection collection )
  {
    this.collection = collection;
  }

  // Properties
  public int Step
  {
    get{ return step; }
    set{ step = value; }
  }

  // Methods
  override public Item First()
  {
    current = 0;
    return (Item)collection[ current ];
  }
```

```
  override public Item Next()
  {
    current += step;
    if( !IsDone() )
      return (Item)collection[ current ];
    else
      return null;
  }

  override public Item CurrentItem()
  {
    return (Item)collection[ current ];
  }

  override public bool IsDone()
  {
    return current >= collection.Count ? true : false ;
  }
}

/// <summary>
/// IteratorApp test
/// </summary>
public class IteratorApp
{
  public static void Main(string[] args)
  {
    // Build a collection
    Collection collection = new Collection();
    collection[0] = new Item( "Item 0" );
    collection[1] = new Item( "Item 1" );
    collection[2] = new Item( "Item 2" );
    collection[3] = new Item( "Item 3" );
    collection[4] = new Item( "Item 4" );
    collection[5] = new Item( "Item 5" );
    collection[6] = new Item( "Item 6" );
    collection[7] = new Item( "Item 7" );
    collection[8] = new Item( "Item 8" );

    // Create iterator
    Iterator iterator = new Iterator( collection );

    // Skip every other item
```

```
    iterator.Step = 2;

    // For loop using iterator
    for( Item item = iterator.First();
          !iterator.IsDone(); item = iterator.Next() )
    {
      Console.WriteLine( item.Name );
    }
  }
}
```

output

```
Item 0
Item 2
Item 4
Item 6
Item 8
```

17 Mediator – 中介者模式

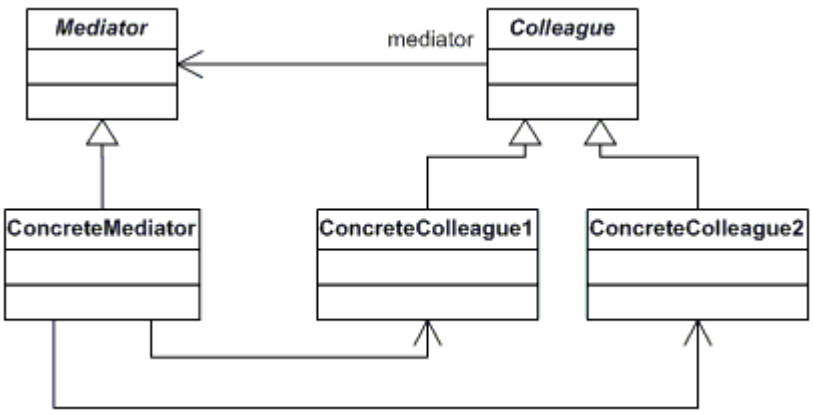
Definition

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

定义一个对象封装一组对象彼此间如何互动。中介者模式避免这一组对象彼此间明确的参考促使低耦合性，同时让你独立的改变它们的接口不会相互影响。

Frequency of use: 1 2 3 4 5 medium low

UML class diagram



Participants

The classes and/or objects participating in this pattern are:

- **Mediator (IChatroom)**
  - defines an interface for communicating with Colleague objects
- **ConcreteMediator (Chatroom)**
  - implements cooperative behavior by coordinating Colleague objects
  - knows and maintains its colleagues
- **Colleague classes (Participant)**
  - each Colleague class knows its Mediator object
  - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague

Sample code in C#

This structural code demonstrates the Mediator pattern facilitating loosely coupled communication between different objects and object types. The mediator is a central hub through which all interaction must take place.

```
using System;
using System.Collections;

// "Mediator"

abstract class Mediator
{
  // Methods
  abstract public void Send( string message,
  Colleague colleague );
}

// "ConcreteMediator"

class ConcreteMediator : Mediator
{
  // Fields
  private ConcreteColleague1 colleague1;
  private ConcreteColleague2 colleague2;

  // Properties
  public ConcreteColleague1 Colleague1
  {
    set{ colleague1 = value; }
```

```
  }

  public ConcreteColleague2 Colleague2
  {
    set{ colleague2 = value; }
  }

  // Methods
  override public void Send( string message,
                             Colleague colleague )
  {
    if( colleague == colleague1 )
      colleague2.Notify( message );
    else
      colleague1.Notify( message );
  }
}

// "Colleague"

abstract class Colleague
{
  // Fields
  protected Mediator mediator;

  // Constructors
  public Colleague( Mediator mediator )
  {
    this.mediator = mediator;
  }
}

// "ConcreteColleague1"

class ConcreteColleague1 : Colleague
{
  // "Constructors"
  public ConcreteColleague1( Mediator mediator )
                                     : base ( mediator ) { }

  // Methods
  public void Send( string message )
  {
    mediator.Send( message, this );
```

```
  }

  public void Notify( string message )
  {
    Console.WriteLine( "Colleague1 gets message: "
                                                  + message );
  }
}

// "ConcreteColleague2"

class ConcreteColleague2 : Colleague
{
  // Constructors
  public ConcreteColleague2( Mediator mediator )
                                      : base ( mediator ) { }

  // Methods
  public void Send( string message )
  {
    mediator.Send( message, this );
  }
  public void Notify( string message )
  {
    Console.WriteLine( "Colleague2 gets message: "
                                                  + message );
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main(string[] args)
  {
    ConcreteMediator m = new ConcreteMediator();
    ConcreteColleague1 c1 = new ConcreteColleague1( m );
    ConcreteColleague2 c2 = new ConcreteColleague2( m );

    m.Colleague1 = c1;
    m.Colleague2 = c2;

    c1.Send( "How are you?" );
```

```
    c2.Send( "Fine, thanks" );
  }
}
```

output

```
Colleague2 gets message: How are you?
Colleague1 gets message: Fine, thanks
```

- This real-world code demonstrates the Mediator pattern facilitating loosely coupled communication between different Participants registering with a Chatroom. The Chatroom is the central hub through which all communication takes place. At this point only one-to-one communication is implemented in the Chatroom, but would be trivial to change to one-to-many.

```
using System;
using System.Collections;

// "Mediator"

interface IChatroom
{
  // Methods
  void Register( Participant participant );
  void Send( string from, string to, string message );
}

// "ConcreteMediator"

class Chatroom : IChatroom
{
  // Fields
  private Hashtable participants = new Hashtable();

  // Methods
  public void Register( Participant participant )
  {
    if( participants[ participant.Name ] == null )
      participants[ participant.Name ] = participant;

    participant.Chatroom = this;
  }

  public void Send( string from, string to, string message )
  {
    Participant pto = (Participant)participants[ to ];
```

```
    if( pto != null )
      pto.Receive( from, message );
  }
}

// "AbstractColleague"

class Participant
{
  // Fields
  private Chatroom chatroom;
  private string name;

  // Constructors
  public Participant( string name )
  {
    this.name = name;
  }

  // Properties
  public string Name
  {
    get{ return name; }
  }

  public Chatroom Chatroom
  {
    set{ chatroom = value; }
    get{ return chatroom; }
  }

  // Methods
  public void Send( string to, string message )
  {
     chatroom.Send( name, to, message );
  }

  virtual public void Receive(
                          string from, string message )
  {
            Console.WriteLine( "{0} to {1}: '{2}'",
                               from, this.name, message );
  }
}
```

```
//" ConcreteColleague1"

class BeatleParticipant : Participant
{
  // Constructors
  public BeatleParticipant( string name )
                                        : base ( name ) { }

  override public void Receive(
                              string from, string message )
  {
    Console.Write( "To a Beatle: " );
    base.Receive( from, message );
  }
}

//" ConcreteColleague2"

class NonBeatleParticipant : Participant
{
  // Constructors
  public NonBeatleParticipant( string name )
                                        : base ( name ) { }

  override public void Receive(
                              string from, string message )
  {
    Console.Write( "To a non-Beatle: " );
    base.Receive( from, message );
  }
}

/// <summary>
/// MediatorApp test
/// </summary>
public class MediatorApp
{
  public static void Main(string[] args)
  {
    // Create chatroom
    Chatroom c = new Chatroom();

    // Create 'chatters' and register them
```

```
    Participant George = new BeatleParticipant("George");
    Participant Paul = new BeatleParticipant("Paul");
    Participant Ringo = new BeatleParticipant("Ringo");
    Participant John = new BeatleParticipant("John") ;
    Participant Yoko = new NonBeatleParticipant("Yoko");

    c.Register( George );
    c.Register( Paul );
    c.Register( Ringo );
    c.Register( John );
    c.Register( Yoko );

    // Chatting participants
    Yoko.Send( "John", "Hi John!" );
    Paul.Send( "Ringo", "All you need is love" );
    Ringo.Send( "George", "My sweet Lord" );
    Paul.Send( "John", "Can't buy me love" );
    John.Send( "Yoko", "My sweet love" ) ;
  }
}
```

output

```
To a Beatle: Yoko to John: 'Hi John!'
To a Beatle: Paul to Ringo: 'All you need is love'
To a Beatle: Ringo to George: 'My sweet Lord'
To a Beatle: Paul to John: 'Can't buy me love'
To a non-Beatle: John to Yoko: 'My sweet love'
```

18 Memento – 备忘录模式

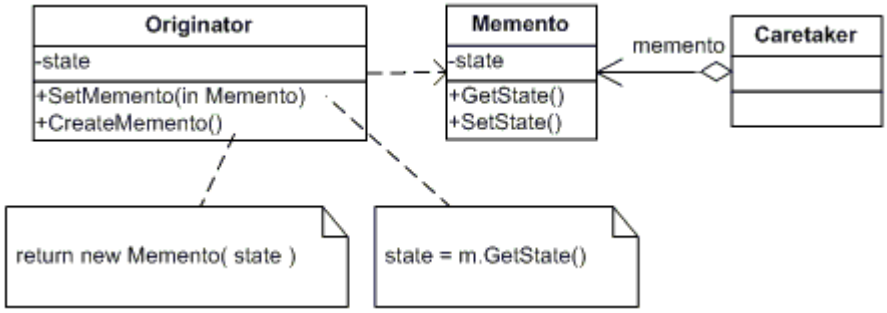
Definition

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

在不违背封装原则下撷取并外型化(externalize)一个对象的内部状态，让这个对象随后可以恢复到这个状态。

Frequency of use: 1 2 3 4 5 low

UML class diagram



Participants

The classes and/or objects participating in this pattern are:

- **Memento (Memento)**
  - stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
  - protect against access by objects of other than the originator. Mementos have effectively two interfaces. Caretaker sees a narrow interface to the Memento -- it can only pass the memento to the other objects. Originator, in contrast, sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produces the memento would be permitted to access the memento's internal state.
- **Originator (SalesProspect)**
  - creates a memento containing a snapshot of its current internal state.
  - uses the memento to restore its internal state
- **Caretaker (Caretaker)**
  - is responsible for the memento's safekeeping
  - never operates on or examines the contents of a memento.

Sample code in C#

This structural code demonstrates the Memento pattern which temporary saves and restores another object's internal state.

```
using System;

// "Originator"

class Originator
{
  // Fields
  private string state = "OFF";

  // Properties
```

```
  public string State
  {
    get{ return state; }
    set{ state = value; }
  }

  // Methods
  public Memento CreateMemento()
  {
    return (new Memento( state ));
  }

  public void SetMemento( Memento memento )
  {
    state = memento.State;
    Console.WriteLine( "Restored to state: {0}", state );
  }
}

// "Memento"

class Memento
{
  // Fields
  private string state;

  // Constructors
  public Memento( string state )
  {
    this.state = state;
  }

  // Properties
  public string State
  {
    get{ return state; }
  }
}

// "Caretaker"

class Caretaker
{
  // Fields
```

```
  private Memento memento;

  // Properties
  public Memento Memento
  {
    set{ memento = value; }
    get{ return memento; }
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    Originator o = new Originator();
    o.State = "On";

    // Store internal state
    Caretaker c = new Caretaker();
    c.Memento = o.CreateMemento();

    // Continue changing originator
    o.State = "Off";

    // Restore saved state
    o.SetMemento( c.Memento );
  }
}
```

output

```
Restored to state: On
```

- This real-world code demonstrates the Memento pattern which temporarily saves and then restores the SalesProspect's internal state.

```
using System;

// "Originator"

class SalesProspect
{
  // Fields
  private string name;
```

```
  private string phone;
  private double budget;

  // Properties
  public string Name
  {
    get{ return name; }
    set{ name = value; }
  }

  public string Phone
  {
    get{ return phone; }
    set{ phone = value; }
  }

  public double Budget
  {
    get{ return budget; }
    set{ budget = value; }
  }

  // Methods
  public Memento SaveMemento()
  {
    return (new Memento( name, phone, budget ));
  }

  public void RestoreMemento( Memento memento )
  {
    this.name = memento.Name;
    this.phone = memento.Phone;
    this.budget = memento.Budget;
  }

  public void Show()
  {
    Console.WriteLine( "\nSales prospect ---- " );
    Console.WriteLine( "Name: {0}", this.name );
    Console.WriteLine( "Phone: {0}", this.phone );
    Console.WriteLine( "Budget: {0:C}", this.budget );
  }
}
```

```
// "Memento"

class Memento
{
  // Fields
  private string name;
  private string phone;
  private double budget;

  // Constructors
  public Memento(string name,string phone,double budget)
  {
    this.name = name;
    this.phone = phone;
    this.budget = budget;
  }

  // Properties
  public string Name
  {
    get{ return name; }
    set{ name = value; }
  }

  public string Phone
  {
    get{ return phone; }
    set{ phone = value; }
  }

  public double Budget
  {
    get{ return budget; }
    set{ budget = value; }
  }
}

// "Caretaker"

class ProspectMemory
{
  // Fields
  private Memento memento;
```

```
  // Properties
  public Memento Memento
  {
    set{ memento = value; }
    get{ return memento; }
  }
}

/// <summary>
/// MementoApp test
/// </summary>
public class MementoApp
{
  public static void Main( string[] args )
  {
    SalesProspect s = new SalesProspect();
    s.Name = "Noel van Halen";
    s.Phone = "(412) 256-0990";
    s.Budget = 25000.0;
    s.Show();

    // Store internal state
    ProspectMemory m = new ProspectMemory();
    m.Memento = s.SaveMemento();

    // Continue changing originator
    s.Name = "Leo Welch";
    s.Phone = "(310) 209-7111";
    s.Budget = 1000000.0;
    s.Show();

    // Restore saved state
    s.RestoreMemento( m.Memento );
    s.Show();
  }
}
```

output

```
Sales prospect ----
Name: Noel van Halen
Phone: (412) 256-0990
Budget: $25,000.00

Sales prospect ----
Name: Leo Welch
```

```
Phone: (310) 209-7111
Budget: $1,000,000.00

Sales prospect ----
Name: Noel van Halen
Phone: (412) 256-0990
Budget: $25,000.00
```

19 Observer – 观察者模式

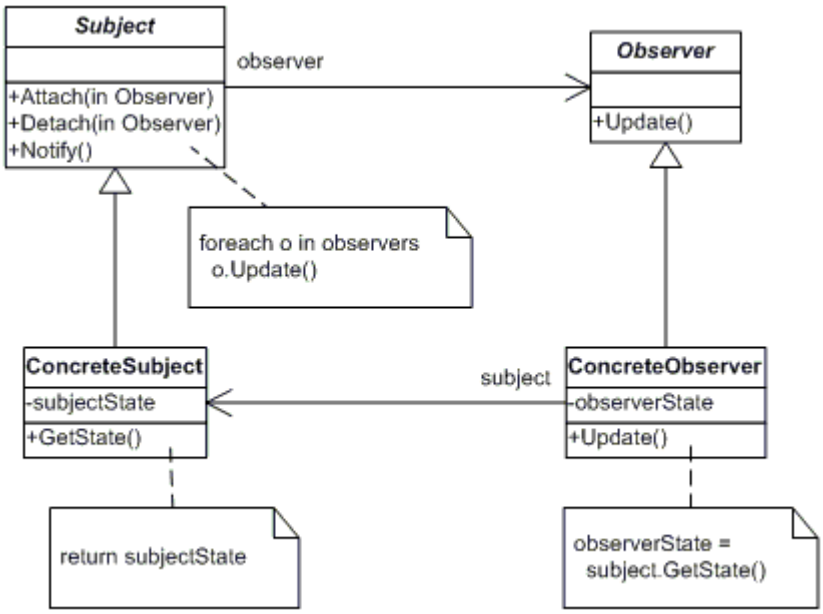
Definition

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

定义对象间一对多相依关系；使得每当一个对象改变状态则其相关的对象皆得到通知并自动更新。

Frequency of use: 1 2 3 4 5 high

UML class diagram



Participants

The classes and/or objects participating in this pattern are:

- **Subject (Stock)**
  - knows its observers. Any number of Observer objects may observe a subject
  - provides an interface for attaching and detaching Observer objects.

- **ConcreteSubject (IBM)**
  - stores state of interest to ConcreteObserver
  - sends a notification to its observers when its state changes
- **Observer (IInvestor)**
  - defines an updating interface for objects that should be notified of changes in a subject.
- **ConcreteObserver (Investor)**
  - maintains a reference to a ConcreteSubject object
  - stores state that should stay consistent with the subject's
  - implements the Observer updating interface to keep its state consistent with the subject's

Sample code in C#

This structural code demonstrates the Observer pattern in which registered objects are notified of and updated with a state change.

```
using System;
using System.Collections;

// "Subject"

abstract class Subject
{
  // Fields
  private ArrayList observers = new ArrayList();

  // Methods
  public void Attach( Observer observer )
  {
    observers.Add( observer );
  }

  public void Detach( Observer observer )
  {
    observers.Remove( observer );
  }

  public void Notify()
  {
    foreach( Observer o in observers )
      o.Update();
  }
}
```

```
// "ConcreteSubject"

class ConcreteSubject : Subject
{
  // Fields
  private string subjectState;

  // Properties
  public string SubjectState
  {
    get{ return subjectState; }
    set{ subjectState = value; }
  }
}

// "Observer"

abstract class Observer
{
  // Methods
  abstract public void Update();
}

// "ConcreteObserver"

class ConcreteObserver : Observer
{
  // Fields
  private string name;
  private string observerState;
  private ConcreteSubject subject;

  // Constructors
  public ConcreteObserver( ConcreteSubject subject,
                                        string name )
  {
    this.subject = subject;
    this.name = name;
  }

  // Methods
  override public void Update()
  {
```

```
    observerState = subject.SubjectState;
    Console.WriteLine( "Observer {0}'s new state is {1}",
                                        name, observerState );
  }

  // Properties
  public ConcreteSubject Subject
  {
    get { return subject; }
    set { subject = value; }
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Configure Observer structure
    ConcreteSubject s = new ConcreteSubject();
    s.Attach( new ConcreteObserver( s, "X" ) );
    s.Attach( new ConcreteObserver( s, "Y" ) );
    s.Attach( new ConcreteObserver( s, "Z" ) );

    // Change subject and notify observers
    s.SubjectState = "ABC";
    s.Notify();
  }
}
```

output

```
Observer X's new state is ABC
Observer Y's new state is ABC
Observer Z's new state is ABC
```

- This real-world code demonstrates the Observer pattern in which registered investors are notified every time a stock changes value.

```
using System;
using System.Collections;

// "Subject"

abstract class Stock
```

```
{
  // Fields
  protected string symbol;
  protected double price;
  private ArrayList investors = new ArrayList();

  // Constructor
  public Stock( string symbol, double price )
  {
    this.symbol = symbol;
    this.price = price;
  }

  // Methods
  public void Attach( Investor investor )
  {
    investors.Add( investor );
  }

  public void Detach( Investor investor )
  {
    investors.Remove( investor );
  }

  public void Notify()
  {
    foreach( Investor i in investors )
      i.Update( this );
  }

  // Properties
  public double Price
  {
    get{ return price; }
    set{ price = value;
           Notify(); }
  }

  public string Symbol
  {
    get{ return symbol; }
    set{ symbol = value; }
  }
}
```

```
// "ConcreteSubject"

class IBM : Stock
{
  // Constructor
  public IBM( string symbol, double price )
                    : base( symbol, price ) {}
}

// "Observer"

interface IInvestor
{
  // Methods
  void Update( Stock stock );
}

// "ConcreteObserver"

class Investor : IInvestor
{
  // Fields
  private string name;
  private string observerState;
  private Stock stock;

  // Constructors
  public Investor( string name )
  {
    this.name = name;
  }

  // Methods
  public void Update( Stock stock )
  {
    Console.WriteLine( "Notified investor {0} of {1}'s " +
      change to {2:C}", name, stock.Symbol, stock.Price );
  }

  // Properties
  public Stock Stock
  {
    get{ return stock; }
```

```
    set{ stock = value; }
  }
}

/// <summary>
/// ObserverApp test
/// </summary>
public class ObserverApp
{
  public static void Main( string[] args )
  {
    // Create investors
    Investor s = new Investor( "Sorros" );
    Investor b = new Investor( "Berkshire" );

    // Create IBM stock and attach investors
    IBM ibm = new IBM( "IBM", 120.00 );
    ibm.Attach( s );
    ibm.Attach( b );

    // Change price, which notifies investors
    ibm.Price = 120.10;
    ibm.Price = 121.00;
    ibm.Price = 120.50;
    ibm.Price = 120.75;
  }
}
```

output

```
Notified investor Sorros of IBM's change to $120.10
Notified investor Berkshire of IBM's change to $120.10
Notified investor Sorros of IBM's change to $121.00
Notified investor Berkshire of IBM's change to $121.00
Notified investor Sorros of IBM's change to $120.50
Notified investor Berkshire of IBM's change to $120.50
Notified investor Sorros of IBM's change to $120.75
Notified investor Berkshire of IBM's change to $120.75
```

20 State – 状态模式

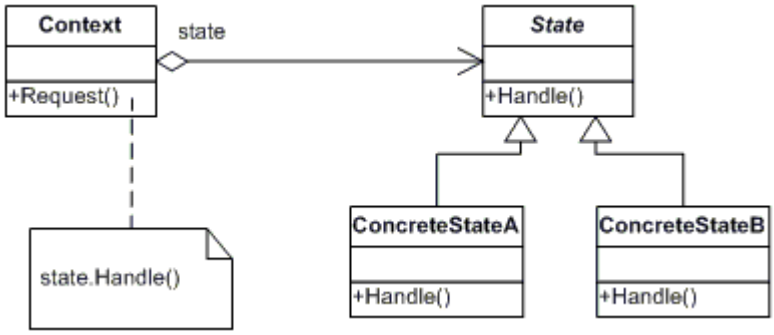
Definition

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class

当一个对象的内在状态改变时允许改变其行为，这个对象看起来(appear)像是改变其类。

Frequency of use: 1 2 3 4 5 medium high

UML class diagram



Participants

The classes and/or objects participating in this pattern are:

- **Context (Account)**
  - defines the interface of interest to clients
  - maintains an instance of a ConcreteState subclass that defines the current state.
- **State (State)**
  - defines an interface for encapsulating the behavior associated with a particular state of the Context.
- **Concrete State (RedState, SilverState, GoldState)**
  - each subclass implements a behavior associated with a state of Context

Sample code in C#

This structural code demonstrates the State pattern which allows an object to behave differently depending on its internal state. The difference in behavior is delegated to objects that represent this state.

```
using System;

// "State"

abstract class State
{
  // Methods
  abstract public void Handle( Context context );
```

```
}

// "ConcreteStateA"

class ConcreteStateA : State
{
  // Methods
  override public void Handle( Context context )
  {
    context.State = new ConcreteStateB();
  }
}

// "ConcreteStateB"

class ConcreteStateB : State
{
  // Methods
  override public void Handle( Context context )
  {
    context.State = new ConcreteStateA();
  }
}

// "Context"

class Context
{
  // Fields
  private State state;

  // Constructors
  public Context( State state )
  {
    this.state = state;
  }

  // Properties
  public State State
  {
    get{ return state; }
    set{ state = value; }
  }
```

```
  // Methods
  public void Request()
  {
    state.Handle( this );
  }

  public void Show()
  {
    Console.WriteLine( "State: " + state );
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Setup context in a state
    Context c = new Context( new ConcreteStateA() );
    c.Show();

    // Issue request, which toggles state
    c.Request();
    c.Show();

    // Issue request, which toggles state
    c.Request();
    c.Show();
  }
}
```

output

```
State: ConcreteStateA
State: ConcreteStateB
State: ConcreteStateA
```

- This real-world code demonstrates the State pattern which allows an Account to behave differently depending on its balance. The difference in behavior is delegated to State objects called RedState, SilverState and GoldState. These states represent overdrawn accounts, starter accounts, and accounts in good standing.

```
using System;

// "State"
```

```
abstract class State
{
  // Fields
  protected Account account;
  protected double balance;
  protected double interest;
  protected double lowerLimit;
  protected double upperLimit;

  // Properties
  public Account Account
  {
    get{ return account; }
    set{ account = value; }
  }
  public double Balance
  {
    get{ return balance; }
    set{ balance = value; }
  }

  // Methods
  abstract public void Initialize();
  abstract public void Deposit( double amount );
  abstract public void Withdraw( double amount );
  abstract public void PayInterest();
  abstract public void StateChangeCheck();
}

// "ConcreteState"
// Account is overdrawn

class RedState : State
{
  // Fields
  double serviceFee;

  // Constructors
  public RedState( State state )
  {
    this.balance = state.Balance;
    this.account = state.Account;
    Initialize();
```

```
  }

  // Methods
  override public void Initialize()
  {
    // Should come from a database
    interest = 0.0;
    lowerLimit = -100.0;
    upperLimit = 0.0;
    serviceFee = 15.00;
  }

  override public void Deposit( double amount )
  {
    balance += amount;
    StateChangeCheck();
  }

  override public void Withdraw( double amount )
  {
    amount = amount - serviceFee;
    Console.WriteLine(
                "No funds available to withdraw!" );
  }

  override public void PayInterest()
  {
    // No interest is paid
  }

  override public void StateChangeCheck()
  {
    if( balance > upperLimit )
      account.State = new SilverState( this );
  }
}

// "ConcreteState"
// Silver is non-interest bearing state

class SilverState : State
{
  // Constructors
  public SilverState(
```

```
            double balance, Account account )
{
  this.balance = balance;
  this.account = account;
  Initialize();
}

public SilverState( State state )
{
  this.balance = state.Balance;
  this.account = state.Account;
  Initialize();
}

// Methods
override public void Initialize()
{
  // Should come from a database
  interest = 0.0;
  lowerLimit = 0.0;
  upperLimit = 1000.0;
}

override public void Deposit( double amount )
{
  balance += amount;
  StateChangeCheck();
}

override public void Withdraw( double amount )
{
  balance -= amount;
  StateChangeCheck();
}

override public void PayInterest()
{
  balance += interest * balance;
  StateChangeCheck();
}

override public void StateChangeCheck()
{
  if( balance < lowerLimit )
```

```
        account.State = new RedState( this );
      else if( balance > upperLimit )
        account.State = new GoldState( this );
    }
}

// "ConcreteState"
// Interest bearing state

class GoldState : State
{
    // Constructors
    public GoldState(
              double balance, Account account )
    {
      this.balance = balance;
      this.account = account;
      Initialize();
    }

    public GoldState( State state )
    {
      this.balance = state.Balance;
      this.account = state.Account;
      Initialize();
    }

    // Methods
    override public void Initialize()
    {
      // Should come from a database
      interest = 0.05;
      lowerLimit = 1000.0;
      upperLimit = 10000000.0;
    }

    override public void Deposit( double amount )
    {
      balance += amount;
      StateChangeCheck();
    }

    override public void Withdraw( double amount )
    {
```

```
    balance -= amount;
    StateChangeCheck();
  }

  override public void PayInterest()
  {
    balance += interest * balance;
    StateChangeCheck();
  }

  override public void StateChangeCheck()
  {
    if( balance < 0.0 )
      account.State = new RedState( this );
    else if( balance < lowerLimit )
      account.State = new SilverState( this );
  }
}

// "Context"

class Account
{
  // Fields
  private State state;
  private string owner;

  // Constructors
  public Account( string owner )
  {
    // New accounts are 'Silver' by default
    this.owner = owner;
    state = new SilverState( 0.0, this );
  }

  // Properties
  public double Balance
  {
    get{ return state.Balance; }
  }
  public State State
  {
    get{ return state; }
    set{ state = value; }
```

```
  }

  // Methods
  public void Deposit( double amount )
  {
    state.Deposit( amount );
    Console.WriteLine( "Deposited {0:C} --- ",
                                        amount);
    Console.WriteLine( " Balance = {0:C}",
                                        this.Balance );
    Console.WriteLine( " Status = {0}" ,
                                        this.State );
    Console.WriteLine( "" );
  }

  public void Withdraw( double amount )
  {
    state.Withdraw( amount );
    Console.WriteLine( "Withdrew {0:C} --- ",
                                        amount);
    Console.WriteLine( " Balance = {0:C}",
                                        this.Balance );
    Console.WriteLine( " Status = {0}" ,
                                        this.State );
    Console.WriteLine( "" );
  }

  public void PayInterest()
  {
    state.PayInterest();
    Console.WriteLine( "Interest Paid --- ");
    Console.WriteLine( " Balance = {0:C}",
                                        this.Balance );
    Console.WriteLine( " Status = {0}" ,
                                        this.State );
    Console.WriteLine( "" );
  }
}

/// <summary>
/// StateApp test
/// </summary>
public class StateApp
{
```

```
  public static void Main( string[] args )
  {
    // Open a new account
    Account account = new Account( "Ana Micola" );

    // Apply financial transactions
    account.Deposit( 500.0 );
    account.Deposit( 300.0 );
    account.Deposit( 550.0 );
    account.PayInterest();
    account.Withdraw( 2000.00 );
    account.Withdraw( 1100.00 );

  }
}
```

output

```
Deposited $500.00 ---
Balance = $500.00
Status = SilverState

Deposited $300.00 ---
Balance = $800.00
Status = SilverState

Deposited $550.00 ---
Balance = $1,350.00
Status = GoldState

Interest Paid ---
Balance = $1,417.50
Status = GoldState

Withdrew $2,000.00 ---
Balance = ($582.50)
Status = RedState

No funds available to withdraw!
Withdrew $1,100.00 ---
Balance = ($582.50)
Status = RedState
```

21 Strategy – 策略模式

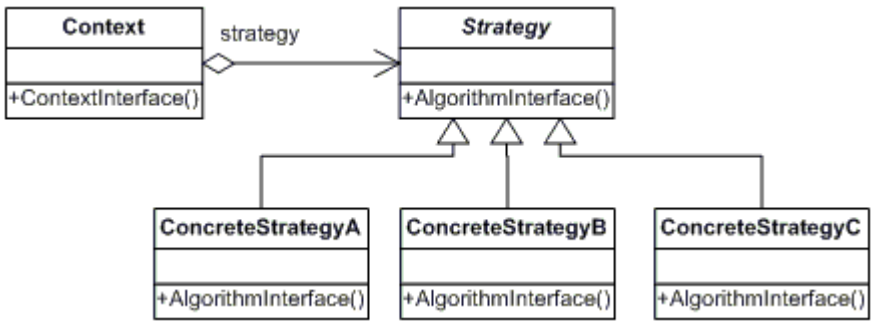
Definition

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

定义一个算法(algorithms)家族(family)；封装每一个成员；并让它们可以相互替换(interchangeable)。策略模式让算法变化独立于使用它的使用端。

Frequency of use: 1 2 3 4 5 medium high

UML class diagram



Participants

The classes and/or objects participating in this pattern are:

- **Strategy (SortStrategy)**
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy
- **ConcreteStrategy (QuickSort, ShellSort, MergeSort)**
  - implements the algorithm using the Strategy interface
- **Context (SortedList)**
  - is configured with a ConcreteStrategy object
  - maintains a reference to a Strategy object
  - may define an interface that lets Strategy access its data.

Sample code in C#

This structural code demonstrates the Strategy pattern which encapsulates functionality in the form of an object. This allows clients to dynamically change algorithmic strategies.

```
using System;

// "Strategy"

abstract class Strategy
{
  // Methods
```

```
  abstract public void AlgorithmInterface();
}

// "ConcreteStrategyA"

class ConcreteStrategyA : Strategy
{
  // Methods
  override public void AlgorithmInterface()
  {
  Console.WriteLine(
       "Called ConcreteStrategyA.AlgorithmInterface()");
  }
}

// "ConcreteStrategyB"

class ConcreteStrategyB : Strategy
{
  // Methods
  override public void AlgorithmInterface()
  {
    Console.WriteLine(
         "Called ConcreteStrategyB.AlgorithmInterface()");
  }
}

// "ConcreteStrategyC"

class ConcreteStrategyC : Strategy
{
  // Methods
  override public void AlgorithmInterface()
  {
    Console.WriteLine(
         "Called ConcreteStrategyC.AlgorithmInterface()");
  }
}

// "Context"

class Context
{
  // Fields
```

```
  Strategy strategy;

  // Constructors
  public Context( Strategy strategy )
  {
    this.strategy = strategy;
  }

  // Methods
  public void ContextInterface()
  {
    strategy.AlgorithmInterface();
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Three contexts following different strategies
    Context c = new Context( new ConcreteStrategyA() );
    c.ContextInterface();

    Context d = new Context( new ConcreteStrategyB() );
    d.ContextInterface();

    Context e = new Context( new ConcreteStrategyC() );
     e.ContextInterface();
  }
}
```

output

```
Called ConcreteStrategyA.AlgorithmInterface()
Called ConcreteStrategyB.AlgorithmInterface()
Called ConcreteStrategyC.AlgorithmInterface()
```

- This real-world code demonstrates the Strategy pattern which encapsulates sorting algorithms in the form of sorting objects. This allows clients to dynamically change sorting strategies including Quicksort, Shellsort, and Mergesort.

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

```
// "Strategy"

abstract class SortStrategy
{
  // Methods
  abstract public void Sort( ArrayList list );
}

// "ConcreteStrategy"

class QuickSort : SortStrategy
{
  // Methods
  public override void Sort(ArrayList list )
  {
    list.Sort(); // Default is Quicksort
    Console.WriteLine("QuickSorted list ");
  }
}

// "ConcreteStrategy"

class ShellSort : SortStrategy
{
  // Methods
  public override void Sort(ArrayList list )
  {
    //list.ShellSort();
    Console.WriteLine("ShellSorted list ");
  }
}

// "ConcreteStrategy"

class MergeSort : SortStrategy
{
  // Methods
  public override void Sort( ArrayList list )
  {
    //list.MergeSort();
    Console.WriteLine("MergeSorted list ");
  }
}
```

```
// "Context"

class SortedList
{
  // Fields
  private ArrayList list = new ArrayList();
  private SortStrategy sortstrategy;

  // Constructors
  public void SetSortStrategy( SortStrategy sortstrategy )
  {
    this.sortstrategy = sortstrategy;
  }

  // Methods
  public void Sort()
  {
    sortstrategy.Sort( list );
  }

  public void Add( string name )
  {
    list.Add( name );
  }

  public void Display()
  {
    foreach( string name in list )
      Console.WriteLine( " " + name );
  }
}

/// <summary>
/// StrategyApp test
/// </summary>
public class StrategyApp
{
  public static void Main( string[] args )
  {
    // Two contexts following different strategies
    SortedList studentRecords = new SortedList( );
    studentRecords.Add( "Samual" );
    studentRecords.Add( "Jimmy" );
    studentRecords.Add( "Sandra" );
```

```
        studentRecords.Add( "Anna" );
        studentRecords.Add( "Vivek" );

        studentRecords.SetSortStrategy( new QuickSort() );
        studentRecords.Sort();
        studentRecords.Display();
    }
}
```

output

```
QuickSorted list
 Anna
 Jimmy
 Samual
 Sandra
 Vivek
```

22 Template Method – 模板方法模式

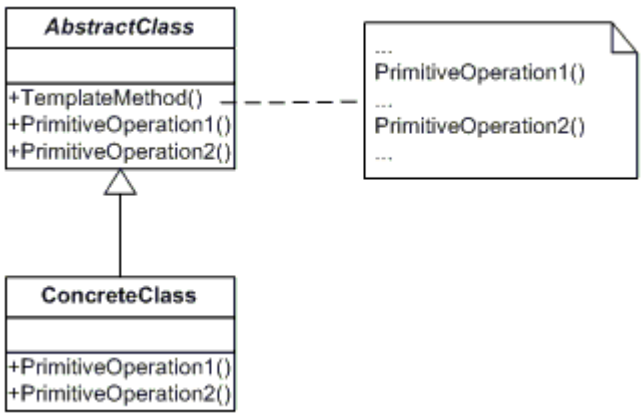
Definition

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

定义一个操作中算法的大纲(skeleton)，以展缓子类的某些步骤。模版方法样让子类重新定义一个算法的某些步骤而无须改变算法的结构。

Frequency of use: 1 2 3 4 5 high

UML class diagram



Participants

The classes and/or objects participating in this pattern are:

- **AbstractClass (DataObject)**
  - defines abstract *primitive operations* that concrete subclasses define to implement steps of an algorithm
  - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- **ConcreteClass (CustomerDataObject)**
  - implements the primitive operations ot carry out subclass-specific steps of the algorithm

Sample code in C#

This structural code demonstrates the Template method which provides a skeleton calling sequence of methods. One or more steps can be deferred to subclasses which implement these steps without changing the overall calling sequence.

```
using System;

// "AbstractClass"

abstract class AbstractClass
{
  // Methods
  abstract public void PrimitiveOperation1();
  abstract public void PrimitiveOperation2();

  // The Template method
  public void TemplateMethod()
  {
    Console.WriteLine(
                  "In AbstractClass.TemplateMethod()");
    PrimitiveOperation1();
    PrimitiveOperation2();
  }
}

// "ConcreteClass"

class ConcreteClass : AbstractClass
{
  // Methods
  public override void PrimitiveOperation1()
  {
    Console.WriteLine(
```

```
        "Called ConcreteClass.PrimitiveOperation1()");
  }

  public override void PrimitiveOperation2()
  {
    Console.WriteLine(
        "Called ConcreteClass.PrimitiveOperation2()");
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Create instance and call template method
    ConcreteClass c = new ConcreteClass();
    c.TemplateMethod();

  }
}
```

output

```
In AbstractClass.TemplateMethod()
Called ConcreteClass.PrimitiveOperation1()
Called ConcreteClass.PrimitiveOperation2()
```

- This real-world code demonstrates a Template method named Run() which provides a skeleton calling sequence of methods. Implementation of these steps are deferred to the CustomerDataObject subclass which implements the Connect, Select, Process, and Disconnect methods.

```
using System;
using System.Data;
using System.Data.OleDb;

// "AbstractClass"

abstract class DataObject
{
  // Methods
  abstract public void Connect();
  abstract public void Select();
  abstract public void Process();
```

```
  abstract public void Disconnect();

  // The "Template Method"
  public void Run()
  {
    Connect();
    Select();
    Process();
    Disconnect();
  }
}

// "ConcreteClass"

class CustomerDataObject : DataObject
{
  private string connectionString =
           "provider=Microsoft.JET.OLEDB.4.0; "
           + "data source=c:\\nwind.mdb";
  private string commandString;
  private DataSet dataSet;

  // Methods
  public override void Connect( )
  {
    // Nothing to do
  }

  public override void Select( )
  {
    commandString = "select CompanyName from Customers";
    OleDbDataAdapter dataAdapter = new OleDbDataAdapter(
                 commandString, connectionString );
    dataSet = new DataSet();
    dataAdapter.Fill( dataSet, "Customers" );
  }

  public override void Process()
  {
    DataTable dataTable = dataSet.Tables["Customers"];
    foreach( DataRow dataRow in dataTable.Rows )
      Console.WriteLine( dataRow[ "CompanyName" ] );
  }
```

```
  public override void Disconnect()
  {
    // Nothing to do
  }
}

/// <summary>
///  TemplateMethodApp test
/// </summary>
public class TemplateMethodApp
{
  public static void Main( string[] args )
  {
    CustomerDataObject c = new CustomerDataObject( );
    c.Run();

  }
}
```

output

```
Alfreds Futterkiste
Ana Trujillo Emparedados y helados
Antonio Moreno Taquería
Around the Horn
Berglunds snabbköp
Blauer See Delikatessen
Blondel père et fils
Bólido Comidas preparadas
Bon app'
Bottom-Dollar Markets
B's Beverages
Cactus Comidas para llevar
Centro comercial Moctezuma
Chop-suey Chinese
...
```

23 Visitor – 访问者模式

Definition

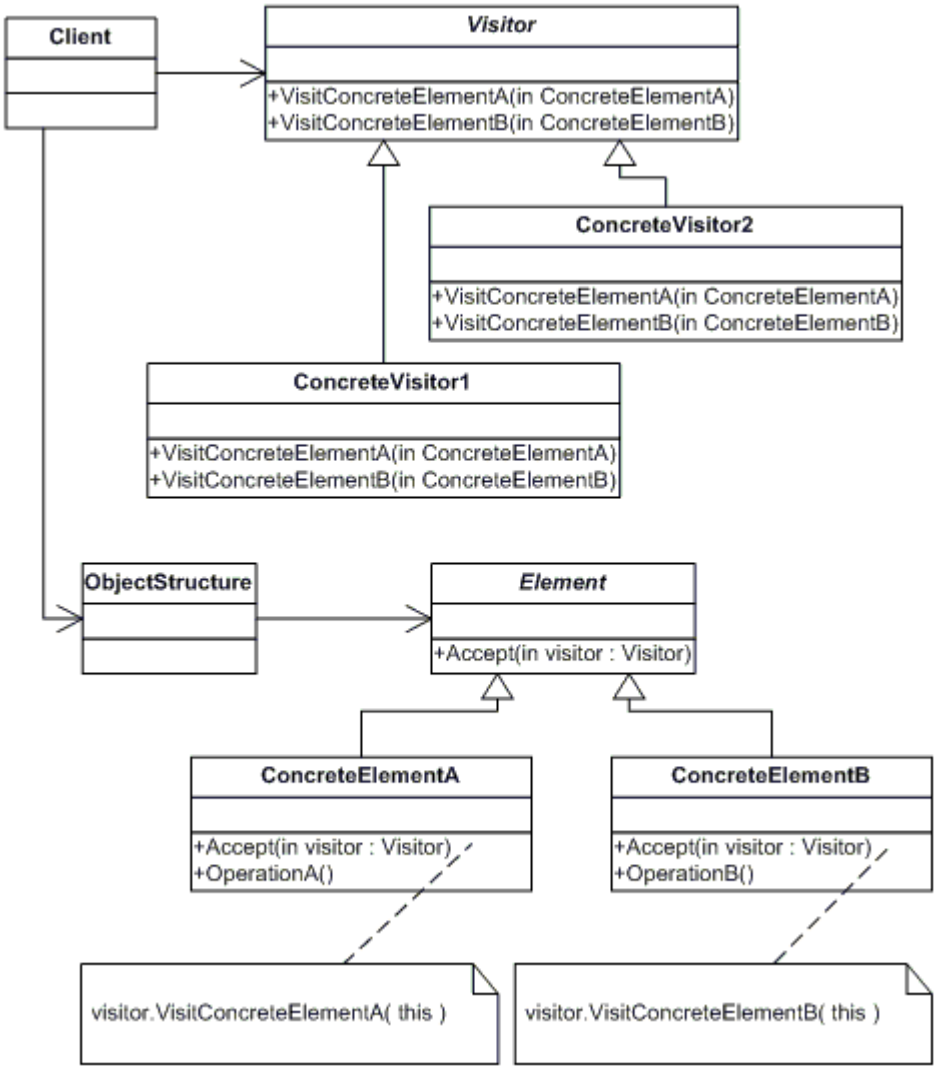
Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

说明一个操作执行于一个对象结构的成员(elements)。访问者模式让你定义一个新的操作无须改变这个成员

的类。

Frequency of use: 1 2 3 4 5 medium low

UML class diagram



Participants

The classes and/or objects participating in this pattern are:

- **Visitor (Visitor)**
  - declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the elements directly through its particular interface
- **ConcreteVisitor (IncomeVisitor, VacationVisitor)**
  - implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class or object in the

structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.

- **Element (Element)**
  - defines an Accept operation that takes a visitor as an argument.
- **ConcreteElement (Employee)**
  - implements an Accept operation that takes a visitor as an argument
- **ObjectStructure (Employees)**
  - can enumerate its elements
  - may provide a high-level interface to allow the visitor to visit its elements
  - may either be a Composite (pattern) or a collection such as a list or a set

Sample code in C#

This structural code demonstrates the Visitor pattern in which an object traverses an object structure and performs the same operation on each node in this structure. Different visitor objects define different operations.

```
using System;
using System.Collections;

// "Visitor"

abstract class Visitor
{
  // Methods
  abstract public void VisitConcreteElementA(
                    ConcreteElementA concreteElementA );
  abstract public void VisitConcreteElementB(
                     ConcreteElementB concreteElementB );
}

// "ConcreteVisitor1"

class ConcreteVisitor1 : Visitor
{
  // Methods
  override public void VisitConcreteElementA(
                     ConcreteElementA concreteElementA )
  {
    Console.WriteLine( "{0} visited by {1}",
                                concreteElementA, this );
  }
```

```
  override public void VisitConcreteElementB(
                      ConcreteElementB concreteElementB )
  {
    Console.WriteLine( "{0} visited by {1}",
                                concreteElementB, this );
  }
}

// "ConcreteVisitor2"

class ConcreteVisitor2 : Visitor
{
  // Methods
  override public void VisitConcreteElementA(
                    ConcreteElementA concreteElementA )
  {
    Console.WriteLine( "{0} visited by {1}",
                                concreteElementA, this );
  }
  override public void VisitConcreteElementB(
                    ConcreteElementB concreteElementB )
  {
    Console.WriteLine( "{0} visited by {1}",
                                concreteElementB, this );
  }
}

// "Element"

abstract class Element
{
  // Methods
  abstract public void Accept( Visitor visitor );
}

// "ConcreteElementA"
class ConcreteElementA : Element
{
  // Methods
  override public void Accept( Visitor visitor )
  {
    visitor.VisitConcreteElementA( this );
  }
```

```
  public void OperationA()
  {
  }
}

// "ConcreteElementB"

class ConcreteElementB : Element
{
  // Methods
  override public void Accept( Visitor visitor )
  {
    visitor.VisitConcreteElementB( this );
  }

  public void OperationB()
  {
  }
}

// "ObjectStructure"

class ObjectStructure
{
  // Fields
  private ArrayList elements = new ArrayList();

  // Methods
  public void Attach( Element element )
  {
    elements.Add( element );
  }

  public void Detach( Element element )
  {
    elements.Remove( element );
  }

  public void Accept( Visitor visitor )
  {
    foreach( Element e in elements )
      e.Accept( visitor );
  }
```

```
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Setup structure
    ObjectStructure o = new ObjectStructure();
    o.Attach( new ConcreteElementA() );
    o.Attach( new ConcreteElementB() );

    // Create visitor objects
    ConcreteVisitor1 v1 = new ConcreteVisitor1();
    ConcreteVisitor2 v2 = new ConcreteVisitor2();

    // Structure accepting visitors
    o.Accept( v1 );
    o.Accept( v2 );
  }
}
```

output

```
ConcreteElementA visited by ConcreteVisitor1
ConcreteElementB visited by ConcreteVisitor1
ConcreteElementA visited by ConcreteVisitor2
ConcreteElementB visited by ConcreteVisitor2
```

- This real-world code demonstrates the Visitor pattern in which two objects traverse a list of Employees and performs the same operation on each Employee. The two visitor objects define different operations -- one adjusts vacation days and the other income.

```
using System;
using System.Collections;

// "Visitor"

abstract class Visitor
{
  // Methods
  abstract public void Visit( Element element );
}
```

```
// "ConcreteVisitor1"

class IncomeVisitor : Visitor
{
  // Methods
  public override void Visit( Element element )
  {
    Employee employee = ((Employee)element);

    // Provide 10% pay raise
    employee.Income *= 1.10;
    Console.WriteLine( "{0}'s new income: {1:C}",
                  employee.Name, employee.Income );
  }
}

// "ConcreteVisitor2"

class VacationVisitor : Visitor
{
  public override void Visit( Element element )
  {
    Employee employee = ((Employee)element);

    // Provide 3 extra vacation days
    employee.VacationDays += 3;
    Console.WriteLine( "{0}'s new vacation days: {1}",
                employee.Name, employee.VacationDays );
  }
}

// "Element"

abstract class Element
{
  // Methods
  abstract public void Accept( Visitor visitor );
}

// "ConcreteElement"

class Employee : Element
{
```

```
  // Fields
  string name;
  double income;
  int vacationDays;

  // Constructors
  public Employee( string name, double income,
                                        int vacationDays )
  {
    this.name = name;
    this.income = income;
    this.vacationDays = vacationDays;
  }

  // Properties
  public string Name
  {
    get{ return name; }
    set{ name = value; }
  }

  public double Income
  {
    get{ return income; }
    set{ income = value; }
  }

  public int VacationDays
  {
    get{ return vacationDays; }
    set{ vacationDays = value; }
  }

  // Methods
  public override void Accept( Visitor visitor )
  {
    visitor.Visit( this );
  }
}

// "ObjectStructure"

class Employees
{
```

```
  // Fields
  private ArrayList employees = new ArrayList();

  // Methods
  public void Attach( Employee employee )
  {
    employees.Add( employee );
  }

  public void Detach( Employee employee )
  {
    employees.Remove( employee );
  }

  public void Accept( Visitor visitor )
  {
    foreach( Employee e in employees )
      e.Accept( visitor );
  }
}

/// <summary>
/// VisitorApp test
/// </summary>
public class VisitorApp
{
  public static void Main( string[] args )
  {
    // Setup employee collection
    Employees e = new Employees();
     e.Attach( new Employee( "Hank", 25000.0, 14 ) );
    e.Attach( new Employee( "Elly", 35000.0, 16 ) );
    e.Attach( new Employee( "Dick", 45000.0, 21 ) );

    // Create two visitors
    IncomeVisitor v1 = new IncomeVisitor();
    VacationVisitor v2 = new VacationVisitor();

    // Employees are visited
    e.Accept( v1 );
    e.Accept( v2 );
  }
}
```

output

```
Hank's new income: $27,500.00
Elly's new income: $38,500.00
Dick's new income: $49,500.00
Hank's new vacation days: 17
Elly's new vacation days: 19
Dick's new vacation days: 24
```

[附录]

1. 设计模式资料来源

data & object factory
DEVELOPER TRAINING

http://www.dofactory.com

2. 我们的培训学校

Training Express

（全文完）