

Gang of Four

Software Design Patterns



**Companion document to
Design Pattern Framework™ 4.0**

by

Data & Object Factory, LLC
www.dofactory.com

Copyright © 2008-2009, Data & Object Factory, LLC
All rights reserved

1. Index

1.	Index	2
2.	Introduction.....	3
3.	The Gang of Four patterns	4
4.	Abstract Factory	5
5.	Builder	10
6.	Factory Method.....	13
7.	Prototype	18
8.	Singleton	21
9.	Adapter.....	26
10.	Bridge	29
11.	Composite	32
12.	Decorator.....	36
13.	Facade	39
14.	Flyweighth	43
15.	Proxy	46
16.	Chain or Responsibility	50
17.	Command.....	53
18.	Interpreter.....	56
19.	Iterator	60
20.	Mediator	64
21.	Memento	67
22.	Observer.....	70
23.	State	73
24.	Strategy	76
25.	Template Method.....	79
26.	Visitor	83

2. Introduction



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

The Gang of Four (GoF) patterns are generally considered the foundation for all other patterns. A total of 23 GoF patterns exist. They are categorized in three groups: Creational, Structural, and Behavioral. Here you will find information on each of these patterns including source code examples in C# or VB (depending on the Edition you purchased). When discussing a pattern the source code is referenced by project name. We suggest that, while reading this guide, you have the [DoFactory.GangOfFour](#) solution open.

Source code to these patterns is provided in 3 forms: *structural*, *real-world*, and *.NET optimized*. *Structural* code uses type names as defined in the pattern definition and UML diagrams. *Real-world* code provides real-world programming situations where you may use the patterns. *.NET optimized* code demonstrates design patterns that exploit built-in .NET features, such as, attributes, generics, reflection, object initialization, and lambda expressions. What is unique about this document is that for every pattern it contains a section that explains when and where the pattern is typically used, as well as a section that explains where Microsoft has used the pattern in their own .NET Framework.

Note: there are a few cases in the .NET optimized code, particularly when *reflection* or *serialization* are involved, where the .NET solution may be elegant, but it may not necessarily represent the most effective solution to the problem. When this is the case we mention it in this document. When applying patterns, it is best to keep an open mind, and, if necessary, run some simple performance tests.

With this out of the way, you're ready to explore the 23 Gang of Four design patterns.

3. The Gang of Four patterns

Below is a list of the 23 Gang of Four patterns:

Creational Patterns	
<u>Abstract Factory</u>	Creates an instance of several families of classes
<u>Builder</u>	Separates object construction from its representation
<u>Factory Method</u>	Creates an instance of several derived classes
<u>Prototype</u>	A fully initialized instance to be copied or cloned
<u>Singleton</u>	A class of which only a single instance can exist

Structural Patterns	
<u>Adapter</u>	Match interfaces of different classes
<u>Bridge</u>	Separates an object's interface from its implementation
<u>Composite</u>	A tree structure of simple and composite objects
<u>Decorator</u>	Add responsibilities to objects dynamically
<u>Façade</u>	A single class that represents an entire subsystem
<u>Flyweight</u>	A fine-grained instance used for efficient sharing
<u>Proxy</u>	An object representing another object

Behavioral Patterns	
<u>Chain of Resp.</u>	A way of passing a request between a chain of objects
<u>Command</u>	Encapsulate a command request as an object
<u>Interpreter</u>	A way to include language elements in a program
<u>Iterator</u>	Sequentially access the elements of a collection
<u>Mediator</u>	Defines simplified communication between classes
<u>Memento</u>	Capture and restore and object's internal state
<u>Observer</u>	A way of notifying change to a number of classes
<u>State</u>	Alter an object's behavior when its state changes
<u>Strategy</u>	Encapsulates an algorithm inside a class
<u>Template Method</u>	Defer the exact steps of an algorithm to a subclass
<u>Visitor</u>	Defines a new operation to a class without change

4. Abstract Factory

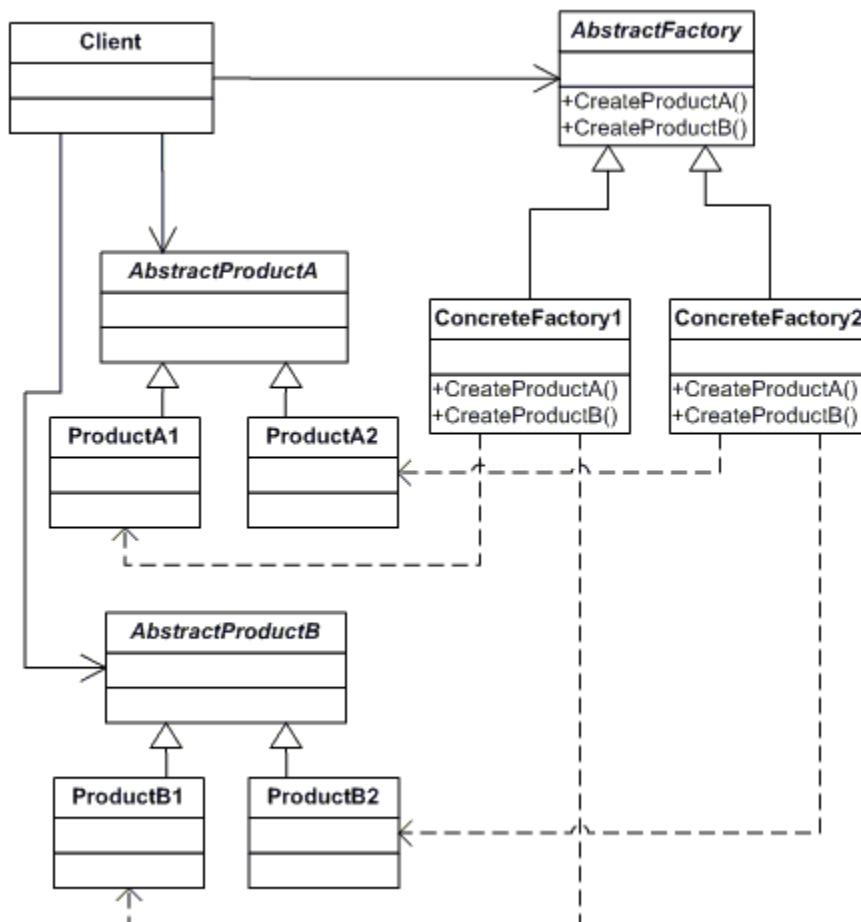
Definition

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



Frequency of use: 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

Structural sample code

The 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.

Code in project: *DoFactory.GangOfFour.Abstract.Structural*

Real-world sample code

The 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.

Code in project: *DoFactory.GangOfFour.Abstract.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates the same functionality as the real-world example but uses more modern, built-in .NET features. Abstract classes have been

replaced by interfaces. There is no need for abstract classes because they have no implementation code. Continents are represented as enumerations. The AnimalWorld constructor dynamically creates the desired factory using the Continent enumerated value.

Code in project: *DoFactory.GangOfFour.Abstract.NetOptimized*

Abstract Factory: when and where use it

The Abstract Factory pattern provides a class that creates other objects that are related by a common theme. The classic example is that of a GUI component factory which creates UI controls for different windowing systems, such as, Windows, Motif, or MacOS. In case you're familiar with Java Swing, it represents a great example of the use of the Abstract Factory pattern to build UI interfaces that are independent of their hosting platform. From a design pattern perspective, Java Swing succeeded, but applications built on this platform are somewhat limited in their interactivity and responsiveness compared to native Windows or native Motif applications.

Over time the meaning of the Abstract Factory pattern has evolved relative to the original GoF definition. Today, when developers talk about the Abstract Factory pattern they not only mean the creation of a 'family of related or dependent' objects but also a simpler idea, that is, the creation of individual object instances.

You may be wondering why you want to create objects using another class (called Abstract Factory) rather than calling constructors directly. Here are some reasons:

Constructors are limited in their control over the overall creation process. If your application needs more control, consider using a Factory. Some possible scenarios where this may be the case is when the creation process involves object caching, sharing or re-using of objects, and applications that maintain object and type counts.

Additionally, there are times when the client does not know exactly what type to construct. It is easier to code against a base type or an interface and then let a factory

make this decision for the client (based on parameters or other context-based information). Good examples of this are the provider-specific ADO.NET objects (DbConnection, DbCommand, DbDataAdapter, etc).

Constructors don't communicate their intention very well because they must be named after their class (or `Sub New` in VB). Having numerous overloaded constructors may make it hard for the client developer to decide which constructor to use. Replacing constructors with intention-revealing creation methods are frequently preferred. Here is an example with 4 overloaded constructors. It is not so clear which one to use.

```
// C#
public Vehicle (int passengers)
public Vehicle (int passengers, int horsepower)
public Vehicle (int wheels, bool trailer)
public Vehicle (string type)
```

```
' VB
public Sub New (Byval passengers As Integer)
public Sub New (Byval passengers As Integer, _
               Byval horsepower As Integer)
public Sub New (Byval wheels As Integer wheels, _
               Byval trailer As Boolean)
public Sub New (Byval type As String)
```

The Factory pattern makes the code far more expressive.

```
// C#
public Vehicle CreateCar (int passengers)
public Vehicle CreateSuv (int passengers, int horsepower)
public Vehicle CreateTruck (int wheels, bool trailer)
public Vehicle CreateBoat ()
public Vehicle CreateBike ()
```

```
' VB
public Function CreateCar (Byval passengers As Integer) As Vehicle
public Function CreateSuv (Byval passengers As Integer, _
                          Byval horsepower As Integer) As Vehicle
public Function CreateTruck (Byval wheels As Integer, _
                           Byval trailer As Boolean) As Vehicle
public Function CreateBoat () As Vehicle
public Function CreateBike () As Vehicle
```

Abstract Factory in the .NET Framework

A search through the .NET Framework libraries for the word 'Factory' reveals numerous classes that are implementations of the Factory design pattern. ADO.NET, for example,

includes two Abstract Factory classes that offer provider independent data access. They are: DbProviderFactory and DbProviderFactories. The DbProviderFactory creates the 'true' (i.e. database specific) classes you need; in the case of SQL Server they are SqlConnection, SqlCommand, and SqlDataAdapter. Each managed provider (such as, SqlClient, OleDb, ODBC, or Oracle) has its own DbProviderFactory class. DbProviderFactory objects, in turn, are created by the DbProviderFactories class (note: the name is plural), which itself is a factory. In fact, it is a factory of factories -- it manufactures different factories, one for each provider.

When Microsoft talks about Abstract Factories they mean types that expose factory methods as virtual or abstract instance functions and that return an abstract class or interface. Below is an example from .NET:

```
// C#
public abstract class StreamFactory
{
    public abstract Stream CreateStream();
}
```

```
' VB
Public MustInherit Class StreamFactory
    Public MustOverride Function CreateStream() As Stream
End Class
```

In this scenario your factory type inherits from StreamFactory and is used to dynamically select the actual Stream type being created:

```
// C#
public class MemoryStreamFactory : StreamFactory
{
    ...
}
```

```
' VB
Public Class MemoryStreamFactory
    Inherits StreamFactory
    ...
End Class
```

The naming convention in .NET for the Factory pattern is to append the word 'Factory' to the name of the type that is being created. For example, a class that manufactures Widget objects would be named WidgetFactory.

5. Builder

Definition

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

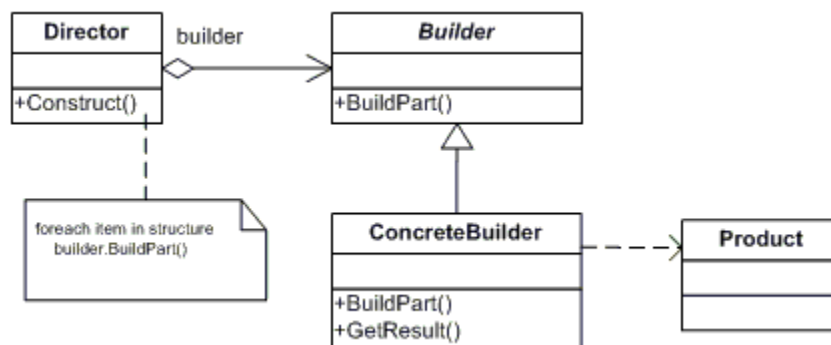


Frequency of use:

1	2	3	4	5

 medium low

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

Structural sample code

The 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.

Code in project: *DoFactory.GangOfFour.Builder.Structural*

Real-world sample code

The real-world code demonstrates a 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.

Code in project: *DoFactory.GangOfFour.Builder.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates the same functionality as the real-world example but uses more modern, built-in .NET features. An enumeration for PartType and VehicleType was added. The ConcreteBuilders have their own constructors, which invoke their base class constructors with the correct VehicleType. Vehicle uses a generic Dictionary for increased type safety. The Vehicle.Show() method uses a this[] indexer rather than the parts[] array.

Code in project: *DoFactory.GangOfFour.Builder.NetOptimized*

Builder: when and where use it

The Builder design pattern is a creational pattern that allows the client to construct a complex object by specifying the type and content only. Construction details are hidden from the client entirely. The most common motivation for using Builder is to simplify client code that creates complex objects. The client can still direct the steps taken by the Builder, without knowing how the actual work is accomplished. Builders frequently encapsulate construction of Composite objects (another design pattern) because the procedures involved are often repetitive and complex.

A scenario where the Builder can be helpful is when building a code generator. Say you're writing an application that writes stored procedures for different target databases (SQL Server, Oracle, Db2). The actual output is quite different but the different steps of creating the separate procedures that implement the CRUD statements (Create, Read, Update, Delete) are similar.

Builder is a creational pattern just like the Factory patterns. However, Builder gives you more control in that each step in the construction process can be customized; Factory patterns create objects in one single step.

Builder in the .NET Framework

The Builder design pattern is infrequently used, but you can still find it in the .NET Framework. Two classes in .NET, `VBCodeProvider` and `CSharpCodeProvider`, create Builder classes through their `CreateGenerator` methods (as an aside, both `CodeProvider` classes are factory classes). The `CreateGenerator` methods return an `ICodeGenerator` interface through which the generation of source code can be controlled. Visual Studio .NET uses these code generating Builder classes internally.

6. Factory Method

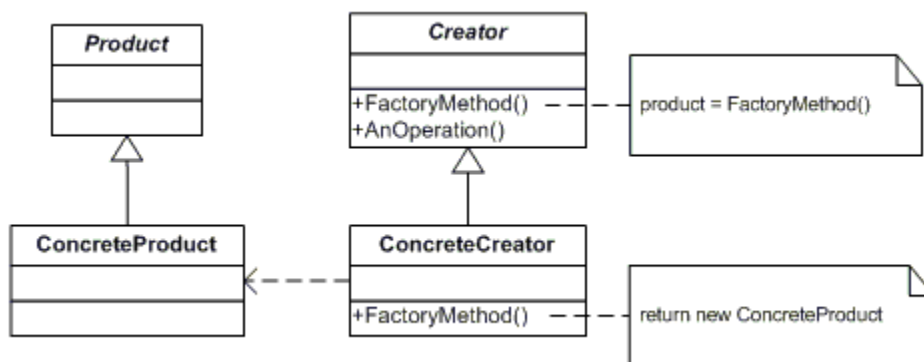
Definition

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



Frequency of use: 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.

Structural sample code

The structural code demonstrates the Factory method offering great flexibility in creating different objects. The Abstract class may provide a default object, but each subclass can instantiate an extended version of the object.

Code in project: *DoFactory.GangOfFour.Factory.Structural*

Real-world sample code

The 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.

Code in project: *DoFactory.GangOfFour.Factory.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates the same functionality as the real-world example but uses more modern, built-in .NET features. Both the fixed size Document array and the Pages ArrayList have been replaced with type-safe generic List<T> collections (List(Of T) collections in VB). New .NET 3.0 language features in this example include *automatic properties* and *collection initialization* significantly reducing the number of lines of code.

Code in project: *DoFactory.GangOfFour.Factory.NetOptimized*

Factory Method: when and where use it

Class constructors exist so that clients can create an instance of a class. There are situations however, where the client does not, or should not, know which one of several candidate classes to instantiate. The Factory Method allows the client to use an interface for creating an object while still retaining control over which class to instantiate.

The key objective of the Factory Method is extensibility. Factory Methods are frequently used in applications that manage, maintain, or manipulate collections of objects that are different but at the same time have many characteristics in common. A document management system, for example, is more extensible if you reference your documents as a collection of `IDocuments`. These documents may be Text files, Word documents, Visio diagrams, or legal papers. But they have in common that they have an author, a title, a type, a size, a location, a page count, etc. When a new document type is introduced it simply implements the `IDocument` interface and it will fit in with the rest of the documents. To support this new document type the Factory Method may or may not have to be adjusted (depending on how it was implemented, that is, with or without parameters). The example below will need adjustment.

```
// C#
public class DocumentFactory
{
    // Factory method with parameter
    public IDocument CreateDocument(DocumentType docType)
    {
        IDocument document = null;

        switch(docType)
        {
            case DocumentType.Word:
                document = new WordDocument();
                break;
            case DocumentType.Excel:
                document = new ExcelDocument();
                break;
            case DocumentType.Visio:
                document = new VisioDocument();
                break;
        }
        return document;
    }
}
```

```
' VB
Public Class DocumentFactory
    ' Factory method with parameter
    Public Function CreateDocument(ByVal docType As DocumentType) _
                                   As IDocument
        Dim document As IDocument = Nothing

        Select Case docType
            Case DocumentType.Word
                document = New WordDocument()
            Case DocumentType.Excel
                document = New ExcelDocument()
            Case DocumentType.Visio
                document = New VisioDocument()
        End Select
        Return document
    End Function
End Class
```

Factory Methods are frequently used in 'manager' type components, such as, document managers, account managers, permission managers, custom control managers, etc.

In your own programs you most likely have created methods that return new objects. However, not all methods that return a new object are Factory methods. So, when do you know the Factory Method is at work? The rules are:

- the method creates a new object
- the method returns an abstract class or interface
- the abstract class or interface is implemented by several classes

Factory Method in .NET Framework

The Factory Method is frequently used in .NET. An example is the System.Convert class which exposes many static methods that, given an instance of a type, returns another new type. For example, Convert.ToBoolean accepts a string and returns a boolean with value true or false depending on the string value ("true" or "false"). Likewise the Parse method on many built-in value types (Int32, Double, etc) are examples of the same pattern.

```
// C#
string myString = "true";
bool myBool = Convert.ToBoolean(myString);
```



```
' VB
Dim myString As String = "true"
Dim myBool As Boolean = Convert.ToBoolean(myString)
```

In .NET the Factory Method is typically implemented as a static method which creates an instance of a particular type determined at compile time. In other words, these methods don't return base classes or interface types of which the true type is only known at runtime. This is exactly where Abstract Factory and Factory Method differ: Abstract Factory methods are virtual or abstract and return abstract classes or interfaces. Factory Methods are abstract and return class types.

Two static factory method examples are File.Open and Activator.Create

```
// C#
public class File
{
    public static FileStream Open(string path, FileMode mode)
    {
        ...
    }
}
```

```
' VB
Public Class File
    Public Shared Function Open(ByVal path As String, _
        ByVal mode As FileMode) As FileStream
        ...
    End Function
End Class
```

```
// C#
public static class Activator
{
    public static object Create(Type type)
    {
        ...
    }
}
```

```
' VB
Public Class Activator
    Public Shared Function Create(ByVal type As Type) As Object
        ...
    End Function
End Class
```

7. Prototype

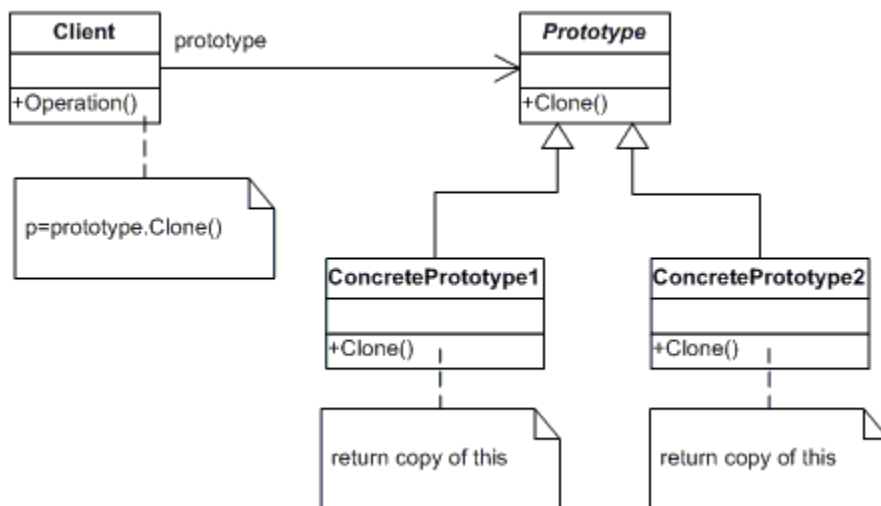
Definition

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



Frequency of use:  medium

UML Class Diagram



Participants

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

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

Structural sample code

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

Code in project: *DoFactory.GangOfFour.Prototype.Structural*

Real-world sample code

The 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.

Code in project: *DoFactory.GangOfFour.Prototype.NetOptimized*

.NET optimized sample code

The .NET optimized code demonstrates the same functionality as the real-world example but uses more modern, built-in .NET features. The abstract classes have been replaced by interfaces because the abstract classes contain no implementation code. RGB values range between 0-255, therefore the int has been replaced with a smaller byte data type. The colors collection in the ColorManager class is implemented with a type-safe generic Dictionary class. A Dictionary is an array of key/value pairs. In this implementation the key is of type *string* (i.e. the color name) and the value is of type *Color* (the Color object instance).

ICloneable is a built-in .NET prototype interface. *ICloneable* requires that the class hierarchy be serializable. Here the *Serializable* attribute is used to do just that (as an aside: if a class has 'event' members then these must be decorated with the *NonSerialized* attribute). Alternatively, reflection could have been used to query each member in the *ICloneable* class. Tip: always be concerned about poor performance when implementing cloning many objects through serialization or reflection.

.NET 3.0 *automatic properties* and *object initialization* are used for the Color class, significantly reducing the number of lines of code (C# only).

Code in project: *DoFactory.GangOfFour.Prototype.NetOptimized*

Prototype: when and where use it

Like other creational patterns (Builder, Abstract Factory, and Factory Method), the Prototype design pattern hides object creation from the client. However, instead of creating a non-initialized object, it returns a new object that is initialized with values it copied from a prototype - or sample - object. The Prototype design pattern is not commonly used in the construction of business applications. It is more often used in specific application types, such as, computer graphics, CAD (Computer Assisted Drawing), GIS (Geographic Information Systems), and computer games.

The Prototype design pattern creates clones of pre-existing sample objects. The best way to implement this in .NET is to use the built-in *ICloneable* interface on the objects that are used as prototypes. The *ICloneable* interface has a method called *Clone* that returns an object that is a copy, or clone, of the original object.

When implementing the *Clone* functionality you need to be aware of the two different types of cloning: *deep copy* versus *shallow copy*. Shallow copy is easier but only copies data fields in the object itself -- not the objects the prototype refers to. Deep copy copies the prototype object and all the objects it refers to. Shallow copy is easy to implement because the *Object* base class has a *MemberwiseClone* method that returns a shallow copy of the object. The copy strategy for deep copy may be more complicated -- some objects are not readily copied (such as *Threads*, *Database connections*, etc). You also need to watch out for circular references.

Prototype in the .NET Framework

.NET support for the Prototype pattern can be found in object serialization scenarios. Let's say you have a prototypical object that has been serialized to persistent storage, such as, disk or a database. Having this serialized representation as a prototype you can then use it to create copies of the original object.

8. Singleton

Definition

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

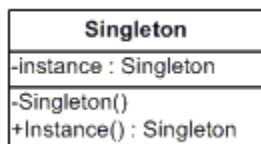


Frequency of use:

1	2	3	4	5

 medium high

UML Class Diagram



Participants

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

- **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.

Structural sample code

The structural code demonstrates the Singleton pattern which assures only a single instance (the singleton) of the class can be created.

Code in project: *DoFactory.GangOfFour.Singleton.Structural*

Real-world sample code

The real-world code demonstrates the Singleton pattern as a LoadBalancing object. Only a single instance (the singleton) of the class should ever exist because servers may dynamically come on-line or off-line. Each request for a server must go through this singleton object because it has 'authoritative' knowledge about the state of the (web) farm.

Code in project: *DoFactory.GangOfFour.Singleton.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates the same code as above but uses more modern, built-in .NET features. Here an elegant .NET specific solution is offered. The Singleton pattern simply uses a *private constructor* and a *static readonly* instance variable that is *lazily initialized*. Thread safety is guaranteed by the compiler. In addition, the list of servers is implemented with a generic List<T> (List(Of T) in VB).

NET 3.0 language features used in this example: The Server class has *automatic properties*. Server instances are created using *object initialization*. Server lists are created using *collection initialization*.

Code in project: *DoFactory.GangOfFour.Singleton.NetOptimized*

Singleton: when and where use it

Most objects in an application are responsible for their own work and operate on self-contained data and references that are within their given area of concern. However, there are objects that have additional responsibilities and are more global in scope, such as, managing limited resources or monitoring the overall state of the system.

The responsibilities of these objects often require that there be just one instance of the class. Examples include cached database records (see TopLink by Oracle), or a scheduling service which regularly emails work-flow items that require attention. Having

more than one database or scheduling service would risk duplication and may result in all kinds of problems.

Other areas in the application rely on these special objects and they need a way to find them. This is where the Singleton design pattern comes in. The intent of the Singleton pattern is to ensure that a class has only one instance and to provide a global point of access to this instance. Using the Singleton pattern you centralize authority over a particular resource in a single object.

Other reasons quoted for using Singletons are to improve performance. A common scenario is when you have a *stateless* object that is created over and over again. A Singleton removes the need to constantly create and destroy objects. Be careful though as the Singleton may not be the best solution in this scenario; an alternative would be to make your methods static and this would have the same effect. Singletons have the unfortunate reputation for being overused by ‘pattern happy’ developers.

Global variables are frowned upon as a bad coding practice, but most practitioners acknowledge the need for a few globals. Using Singleton you can hold one or more global variables and this can be really handy. In fact, this is how Singletons are frequently used – they are an ideal place to keep and maintain globally accessible variables. An example follows:

```
// C#
sealed public class Global
{
    private static readonly Global instance = new Global();

    private string _connectionString;
    private int _loginCount = 0;

    // private constructor
    private Global()
    {
        // Do nothing
    }

    public static Global Instance
    {
        get{ return instance; }
    }

    public string ConnectionString
    {
        get{ return _connectionString; }
    }
}
```

```

        set{ _connectionString = value; }
    }

    public int LoginCount
    {
        get{ return _loginCount; }
        set{ _loginCount = value; }
    }
}

```

```

// VB
NotInheritable Public Class Global
    Private Shared ReadOnly _instance As Global = New Global()

    Private _connectionString As String
    Private _loginCount As Integer = 0

    ' private constructor
    Private Sub New()
        ' Do nothing
    End Sub

    Public Shared ReadOnly Property Instance() As Global
        Get
            Return _instance
        End Get
    End Property

    Public Property ConnectionString() As String
        Get
            Return _connectionString
        End Get
        Set
            _connectionString = Value
        End Set
    End Property

    Public Property LoginCount() As Integer
        Get
            Return _loginCount
        End Get
        Set
            _loginCount = Value
        End Set
    End Property
End Class

```

Singleton in the .NET Framework

An example where the .NET Framework uses the Singleton pattern is with .NET Remoting when launching server-activated objects. One of the activation modes of server objects is called *Singleton* and their behavior is in line with the GoF pattern definition, that is, there is never more than one instance at any one time. If an instance

exists then all client requests will be serviced by this instance – if one does not exist, then a new instance is created and all subsequent client requests will be serviced by this new instance.

9. Adapter

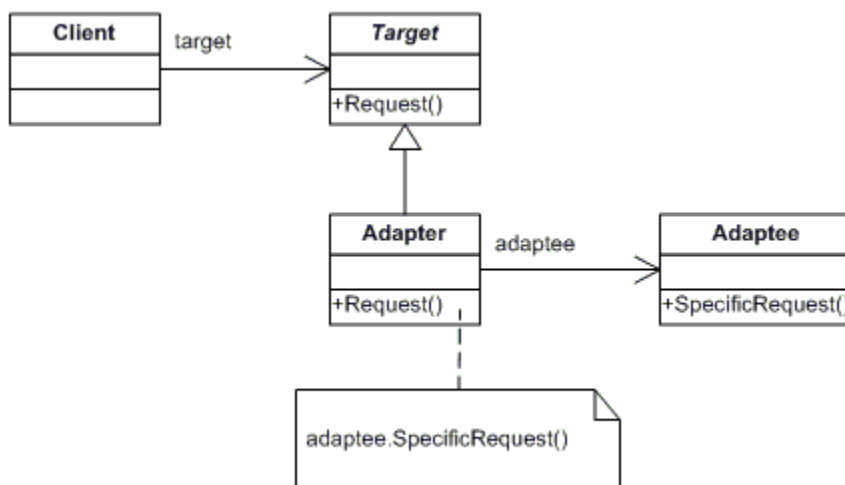
Definition

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



Frequency of use:  medium high

UML Class Diagram



Participants

The classes and/or objects participating in this 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.

Structural sample code

The 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.

Code in project: *DoFactory.GangOfFour.Adapter.Structural*

Real-world sample code

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

Code in project: *DoFactory.GangOfFour.Adapter.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates the same code as above but uses more modern, built-in .NET features. To improve encapsulation Compound class variables were changed from protected to private and several corresponding set/get properties were added. This will allow the derived class to access these variables via properties rather than directly. Finally, two enumerations (Chemical and State) were added for increased type safety.

.NET 3.0 *automatic properties* (C#) are used on the Compound class significantly reducing the amount of code.

Code in project: *DoFactory.GangOfFour.Adapter.NetOptimized*

Adapter: when and where use it

.NET developers write classes that expose methods that are called by clients. Most of the time they will be able to control the interfaces, but there are situations, for example, when using 3rd party libraries, where they may not be able to do so. The 3rd party library performs the desired services but the interface methods and property names are different from what the client expects. This is a scenario where you would use the Adapter pattern. The Adapter provides an interface the client expects using the services of a class with a different interface. Adapters are commonly used in programming environments where new components or new applications need to be integrated and work together with existing programming components.

Adapters are also useful in refactoring scenarios. Say, you have two classes that perform similar functions but have different interfaces. The client uses both classes, but the code would be far cleaner and simpler to understand if they would share the same interface. You cannot alter the interface, but you can shield the differences by using an Adapter which allows the client to communicate via a common interface. The Adapter handles the mapping between the shared interface and the original interfaces.

Adapter in the .NET Framework

The .NET Framework uses the Adapter pattern extensively by providing the ability for .NET clients to communicate with legacy COM components. As you know, there are significant differences between COM and .NET. For example, in error handling, COM components typically return an HRESULT to indicate success or failure, whereas .NET expects an Exception to be thrown in case of an error. The .NET Framework handles these and other differences with so-called Runtime Callable Wrappers (RCW) which is an implementation of the Adapter pattern. The Adapter adapts the COM interface to what .NET clients expect.

10. Bridge

Definition

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

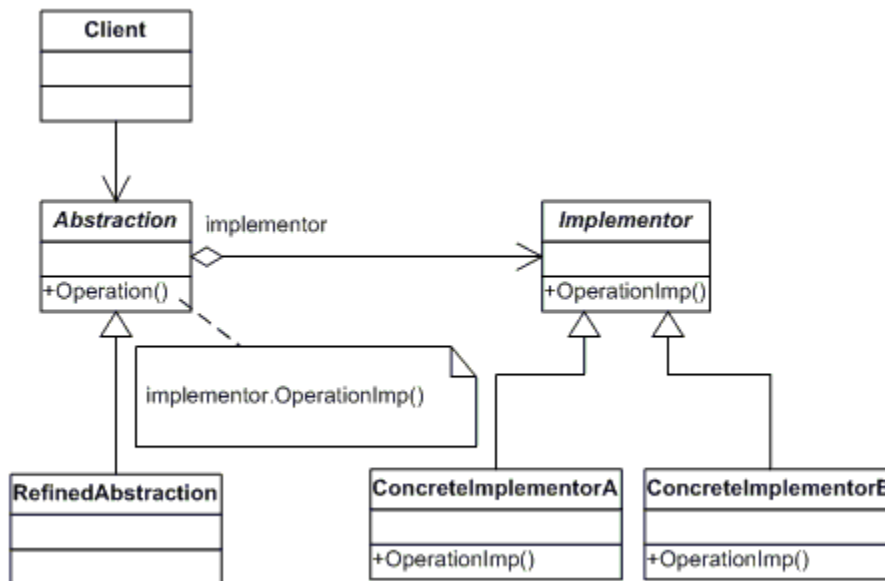


Frequency of use:

1	2	3	4	5

 medium

UML Class Diagram



Participants

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

- **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.

Structural sample code

The 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.

Code in project: *DoFactory.GangOfFour.Bridge.Structural*

Real-world sample code

The 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.

Code in project: *DoFactory.GangOfFour.Bridge.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates the same code as above but uses more modern, built-in .NET features. The DataObject abstract class has been replaced by an interface because DataObject contains no implementation code. Furthermore, to increase type-safety the customer list was implemented as a generic List of strings: List<string> in C# and List(Of String) in VB.

New .NET 3.0 language features (C# only) include *object initialization*, *collection initialization*, and the use of a *foreach extension method* when printing the results.

Code in project: *DoFactory.GangOfFour.Bridge.NetOptimized*

Bridge: when and where use it

The Bridge pattern is used for decoupling an abstraction from its implementation so that the two can vary independently. Bridge is a high-level architectural patterns and its main goal is through abstraction to help .NET developers write better code. A Bridge pattern is created by moving a set of abstract operations to an interface so that both the client and the service can vary independently. The abstraction decouples the client, the interface, and the implementation.

A classic example of the Bridge pattern is when coding against device drivers. A driver is an object that independently operates a computer system or external hardware device. It is important to realize that the client application is the abstraction. Interestingly enough, each driver instance is an implementation of the Adapter pattern. The overall system, the application together with the drivers, represents an instance of a Bridge.

Bridge in the .NET Framework

Bridge is a high-level architectural pattern and as such is not exposed by the .NET libraries themselves. Most developers are not aware of this, but they use this pattern all the time. If you build an application that uses a driver to communicate with a database, say, through ODBC, you're using the Bridge pattern. ODBC is a standard API for executing SQL statements and represents the interface in the Bridge design pattern – classes that implement the API are ODBC drivers. Applications that rely on these drivers are abstractions that work with any database (SQL Server, Oracle, DB2, etc) for which an ODBC driver is available. The ODBC architecture decouples an abstraction from its implementation so that the two can vary independently – the Bridge pattern in action.

11. Composite

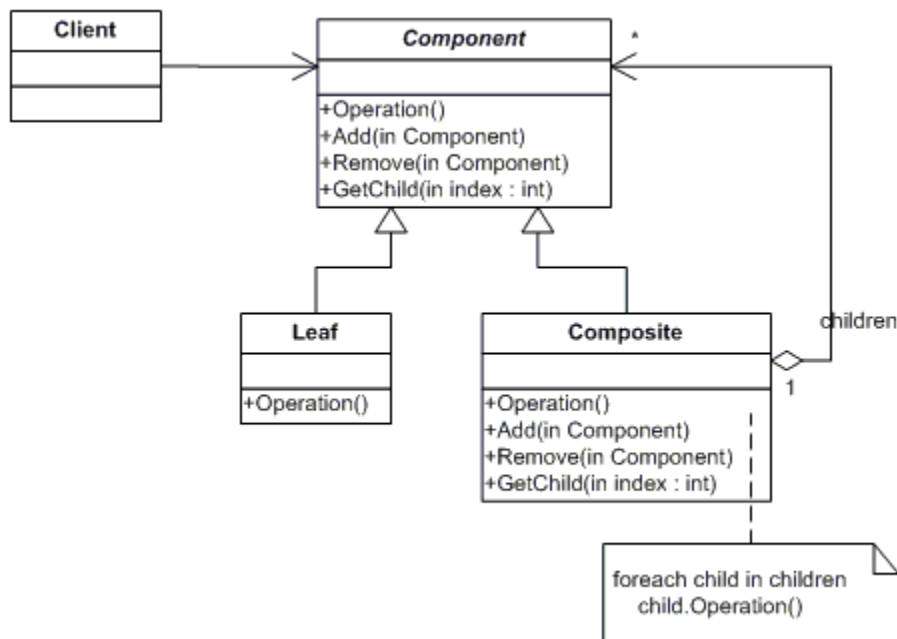
Definition

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



Frequency of use:  medium high

UML Class Diagram



Participants

The classes and/or objects participating in this 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.

Structural sample code

The 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.

Code in project: *DoFactory.GangOfFour.Composite.Structural*

Real-world sample code

The 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).

Code in project: *DoFactory.GangOfFour.Composite.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates the same code as above but uses more modern, built-in .NET features. The composite pattern is a great candidate for generics and you will find these used throughout this example. A generic `TreeNode<T>` was

created (TreeNode(Of T) in VB). This is an *open type* which has the ability to accept any type parameter. The TreeNode has a *generic constraint* in which type T must implement the IComparable<T> interface (IComparable(Of T) in VB). The class named Shape does implement this generic interface so that comparisons can be made between shape objects. This facilitates the process of adding and removing shapes from the list of tree nodes. This code demonstrates much of the power that generics offer to .NET developers. .

Numerous interesting language features are used in this example, including *generics*, *automatic properties*, and *recursion* (the Display method).

Code in project: *DoFactory.GangOfFour.Composite.NetOptimized*

Composite: when and where use it

The Composite design pattern is an in-memory data structures with groups of objects, each of which contain individual items or other groups. A tree control is a good example of a Composite pattern. The nodes of the tree either contain an individual object (leaf node) or a group of objects (a subtree of nodes). All nodes in the Composite pattern share a common interface which supports individual items as well as groups of items. This common interface greatly facilitates the design and construction of recursive algorithms that iterate over each object in the Composite collection.

Fundamentally, the Composite pattern is a collection that you use to build trees and directed graphs. It is used like any other collection, such as, arrays, list, stacks, dictionaries, etc.

Composite in the .NET Framework

The Composite pattern is widely used in .NET. Examples are the two Control classes - one for Windows apps (in the System.Windows.Forms namespace) and the other for ASP.NET apps (in the System.Web.UI namespace). The Control class supports operations that apply to all Controls and their descendants in their respective

environments, as well as operations that deal with child controls (for example the Controls property which returns a collection of child controls).

The built-in .NET TreeNode class is another example of the Composite design pattern in the .NET framework. WPF also has many built-in controls that are Composites.

12. Decorator

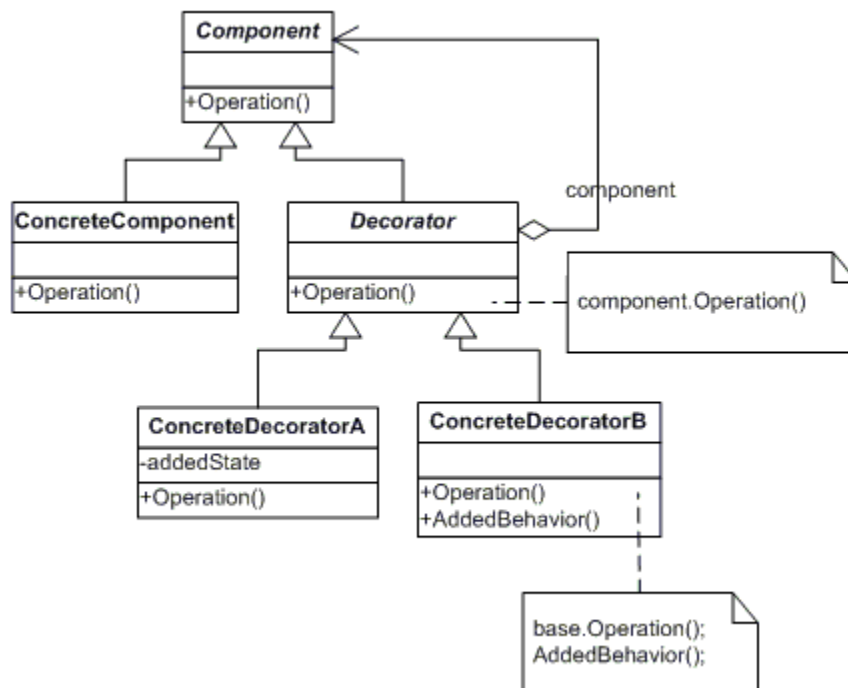
Definition

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



Frequency of use:  medium

UML Class Diagram



Participants

The classes and/or objects participating in this 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.

Structural sample code

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

Code in project: *DoFactory.GangOfFour.Decorator.Structural*

Real-world sample code

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

Code in project: *DoFactory.GangOfFour.Decorator.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates an example of the Decorator design pattern that uses generics; the collection of borrowers is represents in a type-safe collection of type `List<string>` (`List(Of String)` in VB). A NET 3.0 *automatic property* is used in the `LibraryItem` abstract class (in C#).

Code in project: *DoFactory.GangOfFour.Decorator.NetOptimized*

Decorator: when and where use it

The intent of the Decorator design pattern is to let you extend an object's behavior dynamically. This ability to dynamically attach new behavior to objects is done by a Decorator class that 'wraps itself' around the original class.

The Decorator pattern combines polymorphism with delegation. It is polymorphic with the original class so that clients can invoke it just like the original class. In most cases, method calls are delegated to the original class and then the results are acted upon, or *decorated*, with additional functionality. Decoration is a flexible technique because it takes place at runtime, as opposed to inheritance which takes place at compile time.

Decorator in the .NET Framework

Examples of the Decorator in the .NET Framework include a set of classes that are designed around the Stream class. The Stream class is an abstract class that reads or writes a sequence of bytes from an IO device (disk, sockets, memory, etc). The BufferedStream class is a Decorator that wraps the Stream class and reads and writes large chunks of bytes for better performance. Similarly, the CryptoStream class wraps a Stream and encrypts and decrypts a stream of bytes on the fly.

Both BufferedStream and CryptoStream expose the same interface as Stream with methods such as Read, Write, Seek, Flush and others. Clients won't know the difference with the original Stream. Decorator classes usually have a constructor with an argument that represents the class they intent to decorate: for example:

```
new BufferedStream(Stream stream).
```

As an aside, the new .NET 3.0 *extension methods* are a close cousin to this pattern as they also offer the ability to add functionality to an existing type (even if the type is *sealed*). Similarly, *attached properties* and *attached events* which are used in WPF, also allow extending classes dynamically without changing the classes themselves.

13. Facade

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.

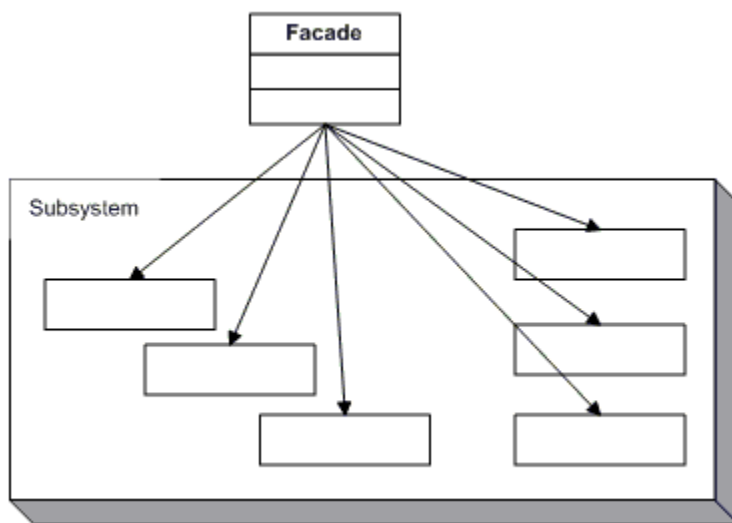


Frequency of use:

1	2	3	4	5

 high

UML Class Diagram



Participants

The classes and/or objects participating in this 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.

Structural sample code

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

Code in project: *DoFactory.GangOfFour.Facade.Structural*

Real-world sample code

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

Code in project: *DoFactory.GangOfFour.Facade.RealWorld*

.NET optimized sample code

This code is essentially the same as the real-world example. The only difference is the use of .NET 3.0 *automatic properties* and *object initializer* on the Customer class.

The ‘*Patterns in Action*’ reference application (which comes with the *Design Pattern Framework* 35) demonstrates the significance of this pattern in modern day architecture. In fact, its use will only increase with the shift towards Web Services and Service Oriented Architectures. The APIs in these architectures are essentially Façades.

Code in project: *DoFactory.GangOfFour.Facade.NetOptimized*

Façade: when and where use it

A Façade is a class that provides an interface (a set of methods and properties) that makes it easier for clients to use classes and objects in a complex subsystem. The Façade pattern is a simple pattern and may seem trivial. Yet, its influence are far-

reaching as it is one of the most commonly used design patterns in systems that are built around a 3-tier architectural model.

The intent of the Façade is to provide a high-level architectural interface that makes a subsystem or toolkit easy to use for the client. In a 3-tier application the presentation layer is the client. Calls into the business layer take place via a well defined service layer. This service layer, or façade, hides the complexity of the business objects and their interactions.

Another area where you use Façade patterns is in refactoring efforts. If you're dealing with a confusing or messy set of legacy classes that the client programmer should not see you hide it behind a Façade. The Façade exposes only what is necessary for the client and presents it in an easy to use and well-organized interface.

Façades are frequently combined with other design patterns. Facades themselves are often implemented as singleton abstract factories. However, you can get the same effect by using static methods on the Façade.

Facade in the .NET Framework

In the .NET Framework you'll find numerous implementations of the Façade design pattern. It is used in scenarios where there is a need to present a simplified view over more complex set of types. To discuss this properly we need to distinguish high-level architectural Facades from lower level component type facades. Microsoft has introduced its own terminology for the lower level façade types in component-oriented designs; they call these *aggregate components*. In reality these are pure façades as they represent a higher level view of multiple lower level types and APIs with the objective to support common programming scenarios.

An example of an *aggregate component* is System.Diagnostics.EventLog. It exposes a simple API to the client, whose only concern is creating the EventLog instance, setting and getting some properties, and writing events to the server's event log. Complex operations, including opening and closing read-and-write handles are totally hidden from the client.

Other *aggregate component* examples include: `System.Web.Mail.SmtpMail` to send mail messages, `System.IO.SerialPort` which is a powerful serial port class, `System.Messaging.MessageQueue` which provides access to a queue on a Message Queue server, and `System.Net.WebClient` which provides a high-level interface for sending and retrieving data from a network resources identified by a general URI.


The decision to include *Façades* in a component library always requires careful consideration. The .NET Framework libraries are no exception. The objective of the .NET libraries is to provide a high degree of control to the programmer whereas the *Façade's* goal is to simplify and limit what the .NET developer can see. *Facades* may reduce the expressiveness of the API, but at the same time provide real 'work-horse' type classes that are simple to use and simple to understand.

14. Flyweight

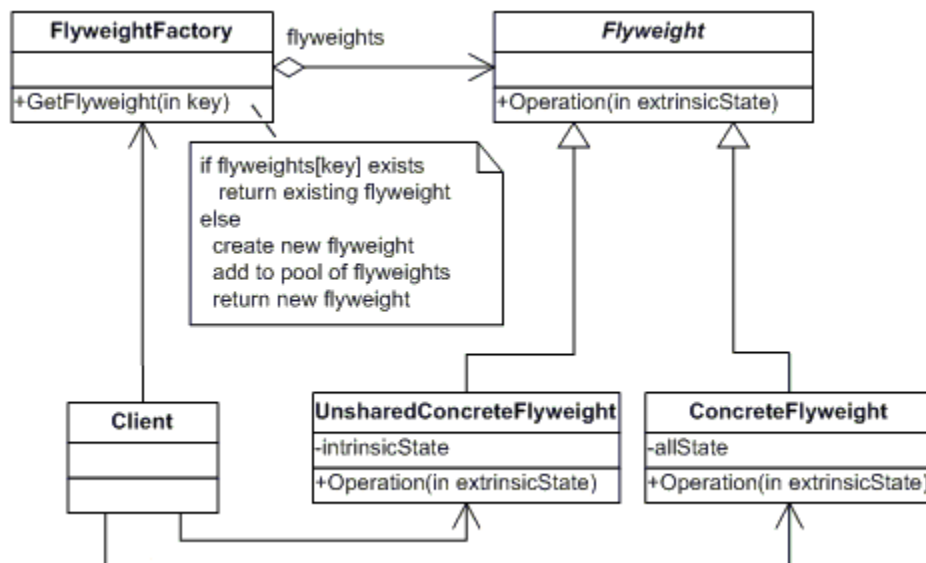
Definition

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



Frequency of use:  low

UML Class Diagram



Participants

The classes and/or objects participating in this 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).

Structural sample code

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

Code in project: *DoFactory.GangOfFour.Flyweight.Structural*

Real-world sample code

The 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.

Code in project: *DoFactory.GangOfFour.Flyweight.RealWorld*

.NET optimized sample code

The Flyweight Pattern is hardly used in business application development. However, it is valuable as a memory management technique when many objects are created with similar state. Internally, .NET uses Flyweights for strings that are declared at compile time and have the same sequence of characters. The stateless flyweights refer to the same memory location that holds the immutable string.

Flyweights are usually combined with the Factory pattern as demonstrated in the .NET Optimized code sample. The .NET optimized code uses a generic Dictionary collection to hold and quickly access Flyweight Character objects in memory. The generic collection increases the type-safety of the code.

Code in project: *DoFactory.GangOfFour.Flyweight.NetOptimized*

Flyweight: when and where use it

The intent of the Flyweight design pattern is to share large numbers of fine-grained objects efficiently. Shared flyweight objects are immutable, that is, they cannot be changed as they represent the characteristics that are shared with other objects. Examples include, characters and line-styles in a word processor or *digit receivers* in a public switched telephone network application. You will find flyweights mostly in utility type applications (word processors, graphics programs, network apps). They are rarely used in data-driven business type applications.

Flyweight in the .NET Framework

As mentioned above, Flyweights are used internally in the .NET Framework as a string management technique to minimize memory usage for immutable strings.

15. Proxy

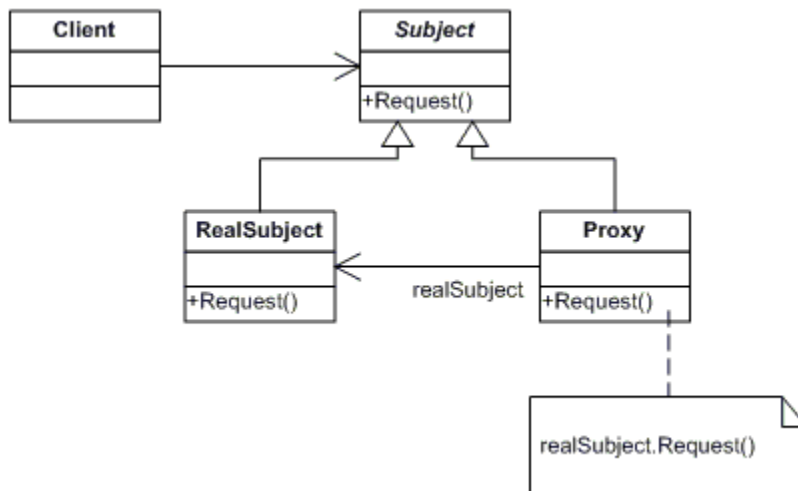
Definition

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



Frequency of use: medium high

UML Class Diagram



Participants

The classes and/or objects participating in this 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 responsibilities 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.

Structural sample code

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

Code in project: *DoFactory.GangOfFour.Proxy.Structural*

Real-world sample code

The real-world code demonstrates the Proxy pattern for a Math object represented by a MathProxy object.

Code in project: *DoFactory.GangOfFour.Proxy.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates the same code as above but uses more modern, built-in .NET features. This code demonstrates the Remote Proxy pattern which provides a representative object (i.e., a stand-in object) that controls access to another

object in a different AppDomain. In fact, the 'math' member variable in the MathProxy object is the 'hidden' .NET proxy that represents the Math object in the MathDomain.

Code in project: *DoFactory.GangOfFour.Proxy.NetOptimized*

Proxy: when and where use it

In object-oriented languages objects do the work they advertise through their public interface. Clients of these objects expect this work to be done quickly and efficiently. However, there are situations where an object is severely constrained and cannot live up to its responsibility. Typically this occurs when there is a dependency on a remote resource (a call to another computer for example) or when an object takes a long time to load. In situations like these you apply the Proxy pattern and create a proxy object that 'stands in' for the original object. The Proxy forwards the request to a target object. The interface of the Proxy object is the same as the original object and clients may not even be aware they are dealing with a proxy rather than the real object.

The proxy pattern is meant to provide a surrogate or placeholder for another object to control access to it. There are 3 different types of proxies:

- *Remote* proxies are responsible for encoding a request and for forwarding the encoded request to a real object in a different address space (app domain, process, or machine)
- *Virtual* proxies may cache additional information about a real object so that they can postpone accessing it (this process is known by many names, such as, just-in-time loading, on-demand loading, or lazy loading)
- *Protection* proxies check that the caller has the proper access permissions to perform the request.

Actually, there is another type, called *Smart References*. A Smart Reference is a proxy for a pointer, but since there are few uses for pointers in .NET you're unlikely to run into this type of proxy.

Proxy in the .NET Framework

In .NET the Proxy pattern manifests itself in the Remoting infrastructure. In .NET Remoting, whenever an object requires access to an object in a different address space (app domain, process, or machine) a proxy is created that sends the request to the remote object and any data it needs. As is common with proxies, the client is frequently not even aware that a proxy is at work.

Clients of WCF services also rely heavily on auto-generated proxy objects.

16. Chain or Responsibility

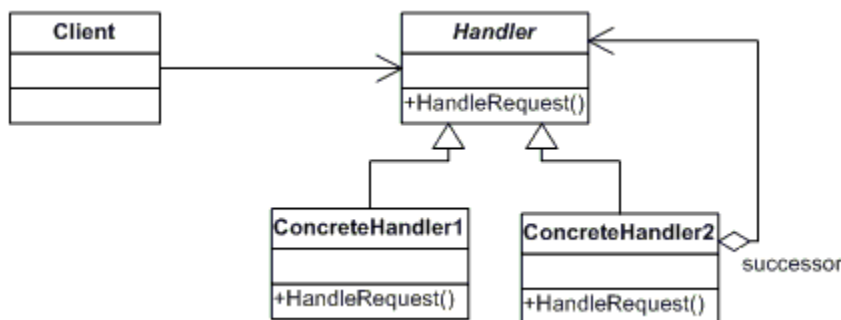
Definition

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.



Frequency of use:  medium low

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

Structural sample code

The 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.

Code in project: *DoFactory.GangOfFour.Chain.Structural*

Real-world sample code

The 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.

Code in project: *DoFactory.GangOfFour.Chain.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates the same code as above but uses more modern, built-in .NET features. The Successor settings are simplified by using properties. Furthermore, this example uses an event driven model with events, delegates, and custom event arguments. The delegates are implemented using generics in which event handlers are type safe and not restricted to senders of type *object* but rather to types that are appropriate for the event – in the sample code type *Approver* (see, for example, the first argument in the *DirectorRequest* event handler). The *Purchase* and other classes use .NET 3.0 *automatic properties* (C#) and *object initializers*.

The chain of responsibility pattern is frequently used in the Windows event model in which a UI control can either handle an event (for example a mouse click) or let it fall through to the next control in the event chain.

Code in project: *DoFactory.GangOfFour.Chain.NetOptimized*

Chain of Responsibility: when and where use it

An important goal of object-oriented design is to keep objects loosely coupled by limiting their dependencies and keeping the relationships between objects specific and minimal. Loosely coupled objects have the advantage that they are easier to maintain and easier to change compared to systems where there is tight coupling between objects (i.e. hard-coded references to specific classes).

The Chain of Responsibility design pattern offers an opportunity to build a collection of loosely coupled objects by relieving a client from having to know which objects in a collection can satisfy a request by arranging these objects in a chain. This pattern requires a way to order the search for an object that can handle the request. This search is usually modeled according to the specific needs of the application domain. Note that Chain-of-Responsibility is not commonly used in business application development.

Chain of Responsibility in the .NET Framework

In .NET you can identify a Chain of Responsibility in the Windows event model where each UI control can decide to process an event or let it fall through to the next control in the event chain.

Occasionally you may run into a Chain of Responsibility implementation in which a chain of objects process a message between a sender and a receiver, in which each object does some processing on the message as it travels through the chain from the sender to the receiver. This is slightly different from the GoF definition in which just one object in a chain decides to handle the request. The .NET Framework implements this 'stepwise chain pattern' in .NET Remoting in which a message between a client and a server passes through one or more so-called message sinks. Message sinks form a chain as each sink has a reference to the next sink in the chain. Sinks implement the IMessageSink interface and one of its members is the NextSink property.

17. Command

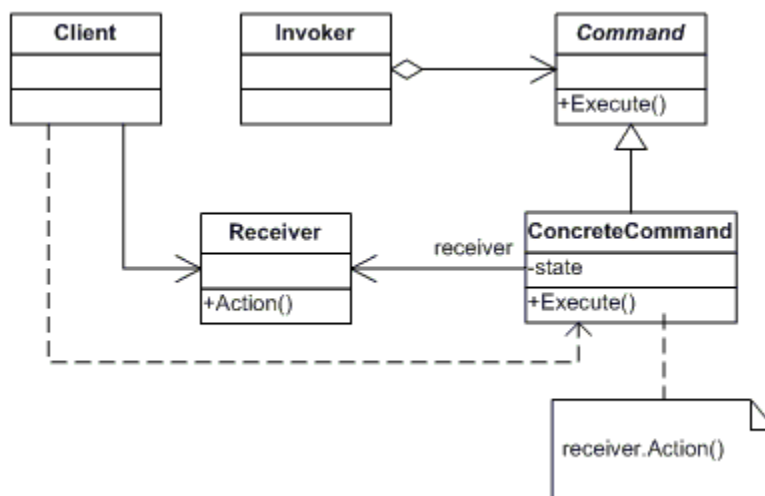
Definition

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



Frequency of use:  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.

Structural sample code

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

Code in project: *DoFactory.GangOfFour.Command.Structural*

Real-world sample code

The 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.

Code in project: DoFactory.*GangOfFour.Command.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates the same code as above but uses more modern, built-in .NET features. In this example the abstract Command class has been replaced by the ICommand interface because the abstract class had no implementation code at all. In addition, for increased type-safety the collection of commands in the User class is implemented as a generic List<> of ICommand interface types (List(Of ICommand) in VB).

Code in project: *DoFactory.GangOfFour.Command.NetOptimized*

Command: when and where use it

The Command design pattern encapsulates an action or a request as an object. The classic usage of this pattern is a menu system where each command object represents an *action* and an associated *undo* action. Menu actions include menu items such as File | Open, File | Save, Edit | Copy, etc each of which gets mapped to its own command object.

All Commands implement the same interface, so they can be handled polymorphically. Typically their interface includes methods such as Do and Undo (or Execute and Undo). Areas where you find Command patterns are: menu command systems and in applications that require undo functionality (word processors, and sometimes in business applications that need database undo functionality).

Command in the .NET Framework


We don't have visibility into the source code of Microsoft's applications, but we are quite certain that most, including Visual Studio .NET, use the Command pattern to support their menus, toolbars, shortcuts, and associated undo functionality. We would have expected that the Command pattern would be exposed in .NET as part of a unified WinForms command routing architecture, but they are not. So, until recently the Command patterns was not generally used in the .NET Framework, but with the introduction of WPF that has changed: WPF natively supports Commands in its Command System (and is used in our 'Patterns in Action' WPF application).

18. Interpreter

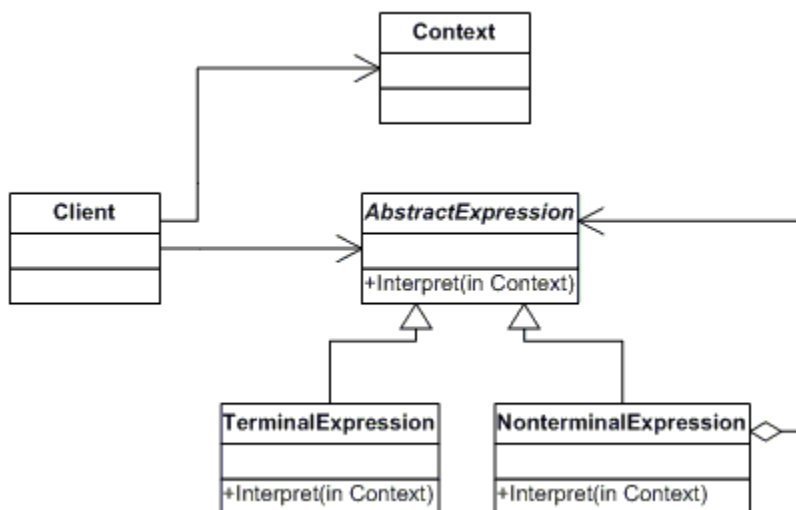
Definition

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



Frequency of use:  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 $R ::= R_1R_2...R_n$ in the grammar
 - maintains instance variables of type AbstractExpression for each of the symbols R_1 through R_n .
 - implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R_1 through R_n .
- **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

Structural sample code

The structural code demonstrates the Interpreter patterns, which using a defined grammar, provides the interpreter that processes parsed statements

Code in project: *DoFactory.GangOfFour.Interpreter.Structural*

Real-world sample code

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

Code in project: *DoFactory.GangOfFour.Interpreter.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates the same code as above but uses more modern, built-in .NET features. Here the abstract classes have been replaced by interfaces because the abstract classes have no implementation code. In addition, the parse tree which holds the collection of expressions (ThousandExpression, HundredExpression, etc) is implemented as a generic List of type Expression. .NET 3.0 features in this example include *object initialization* and *automatic properties* (C#)

Code in project: *DoFactory.GangOfFour.Interpreter.NetOptimized*

Interpreter: when and where use it

If your applications are complex and require advanced configuration you could offer a scripting language which allows the end-user to manipulate your application through simple scripting. The Interpreter design pattern solves this particular problem – that of creating a scripting language that allows the end user to customize their solution.

The truth is that if you really need this type of control it is probably easier and faster to use an existing command interpreter or expression evaluator tool out of the box. VBScript comes to mind, as well Microsoft's new support for .NET Dynamic Languages.

Certain types of problems lend themselves to be characterized by a language. This language describes the problem domain which should be well-understood and well-defined. In addition, this language needs to be mapped to a grammar. Grammars are usually hierarchical tree-like structures that step through multiple levels but end up with terminal nodes (also called literals). This type of problem, expressed as a grammar, can be implemented using the Interpreter design pattern. The well-known Towers of Hanoi puzzle is an example of the type of problem that can be encoded by a simple grammar and implemented using the Interpreter design pattern. For an example visit:

http://home.earthlink.net/~huston2/ps/hanoi_article.html

The Interpreter design pattern shares similarities to several other patterns. Just like State and Strategy, it delegates processing to a set of dedicated classes. It also has similarities with the Composite pattern; basically Interpreter is an enhancement of Composite although it frequently requires more complex object groupings compared to the Composite pattern.

Interpreter in the .NET Framework

We are not aware of the Interpreter pattern being used in the .NET Framework libraries.

19. Iterator

Definition

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

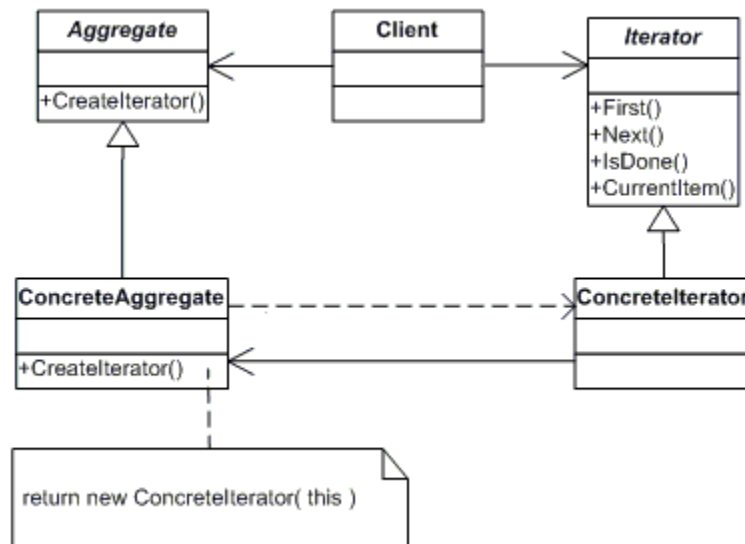


Frequency of use:

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

Structural sample code

The 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.

Code in project: *DoFactory.GangOfFour.Iterator.Structural*

Real-world sample code

The 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.

Code in project: *DoFactory.GangOfFour.Iterator.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates the same code as above but uses more modern, built-in .NET features. In this example the IEnumerable and IEnumerator interfaces are implemented using .NET's built-in generic Iterator design pattern.

C# offers built-in iterator support with the *yield return* keyword which makes implementing the IEnumerable and IEnumerator interfaces even easier (even iterating over trees and other more complex data structures). Three generic IEnumerable<T> methods demonstrate the elegant manner in which you can code iterators that loop front-to-back, back-to-front, or loop over a subset of items with a given step size (FromToStep method).

However, VB does not support the 'yield return' keyword, but the .NET code sample demonstrates the implementation of flexible generic iterator using the IEnumerator(Of T)

interface. Two looping methods are demonstrated: *While* and *For Each*.

Code in project: *DoFactory.GangOfFour.Iterator.NetOptimized*

Iterator: when and where use it

A common programming task is to traverse and manipulate a collection of objects. These collections may be stored as an array, a list, or perhaps something more complex, such as a tree or graph structure. In addition, you may need access the items in the collection in a certain order, such as, front to back, back to front, depth first (as in tree searches), skip evenly numbered objects, etc. The Iterator design pattern solves this problem by separating the collection of objects from the traversal of these objects by implementing a specialized iterator class.

Not only do you find the Iterator design pattern deep into the .NET libraries, it is one of only two patterns that are part of the C# and VB language itself (the other is the Observer design pattern). Both languages have a built-in construct that facilitates iterating over collections: *foreach* in C# and *For Each* in VB.

```
// C#
string[] votes = new string[] { "Agree", "Disagree", "Don't Know"};
foreach (string vote in votes)
{
    Console.WriteLine(vote);
}
```

```
' VB
Private votes As String() = _
    New String() {"Agree", "Disagree", "Don't Know"}
For Each vote As String In votes
    Console.WriteLine(vote)
Next vote
```

The objects referenced in the 'In' expression must implement the IEnumerable interface, so that the collection of objects can be traversed.

Iterator in the .NET Framework

As mentioned above the Iterator pattern is not only part of the .NET Framework libraries, it is baked into the language itself. The .NET libraries contain numerous classes that implement the IEnumerable and IEnumerator interfaces, such as, Array, ArrayList, AttributesCollection, BaseChannelObjectWithProperties, BaseCollection, BindingsContext, as well as the generic counterparts of these classes.

.NET 3.0 makes iteration even more fun and powerful with LINQ (language integrated query) which is a query language built on top of the generic IEnumerable<T> and IEnumerator<T> iterator types.

20. Mediator

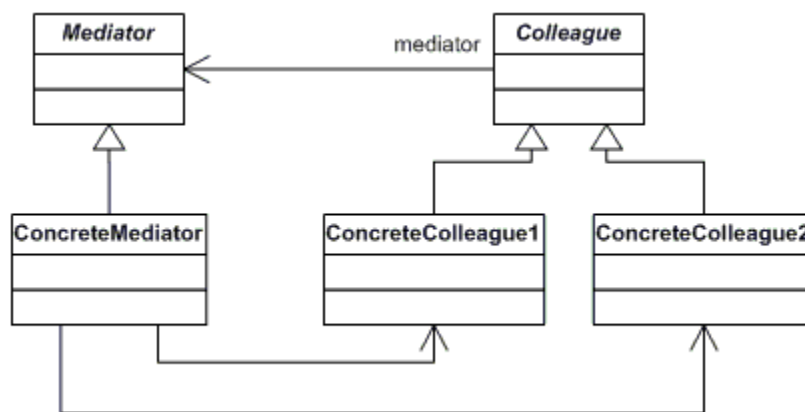
Definition

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.



Frequency of use:  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

Structural sample code

The 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.

Code in project: *DoFactory.GangOfFour.Mediator.Structural*

Real-world sample code

The 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 it would be trivial to change this to a one-to-many communication system.

Code in project: *DoFactory.GangOfFour.Mediator.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates the same code as above but uses more modern, built-in .NET features. Here the abstract classes have been replaced with interfaces because there is no implementation code. In addition, a generic Dictionary collection is used to hold the Participant objects in a type-safe manner. Finally, you will also see .NET 3.0 *automatic properties* (C#) and *object initializers* used in this example.

Code in project: *DoFactory.GangOfFour.Mediator.NetOptimized*

Mediator: when and where use it

The Mediator design pattern defines an object that provides central authority over a group of objects by encapsulating how these objects interact. This model is useful for

scenarios where there is a need to manage complex conditions in which every object is aware of every state change of other objects in the group.

Mediator patterns are frequently used in the development of complex dialog boxes. Take for example a dialog in which you enter options to make a flight reservation. A simple Mediator rule would be: you must enter a valid departure date, a valid return date, the return date must be after the departure date, a valid departure airport, a valid arrival airport, a valid number of travelers, and only then the Search command button can be activated.

Another area where Mediator is used is in complex configuration scenarios. Dell's website provides a good example. When selecting custom options for your computer, the configurator (Mediator) keeps track of all your selections. Its main role is to determine whether a particular combination of hardware components work together or not. For example a particular graphics card may not work with a Sony monitor. The Mediator is the pattern that flags these kinds of incompatibilities.

Mediator in the .NET Framework

We are not aware of the Mediator being used in the .NET Framework libraries.

21. Memento

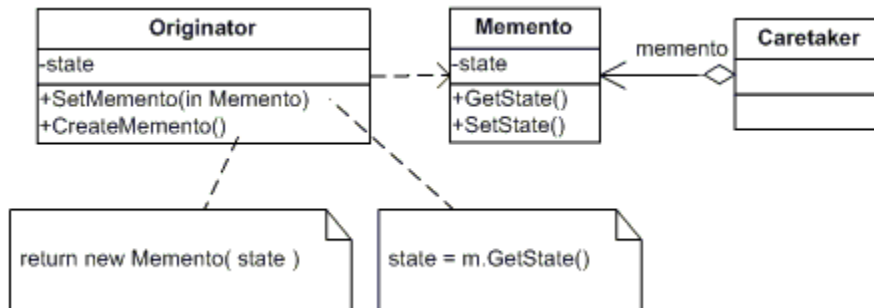
Definition

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



Frequency of use:  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.

Structural sample code

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

Code in project: *DoFactory.GangOfFour.Memento.Structural*

Real-world sample code

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

Code in project: *DoFactory.GangOfFour.Memento.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates the same code as above but uses more modern, built-in .NET features. In this example the Memento is more general. It will take any serializable object and save and restore it. The Originator class is decorated with the *Serializable* attribute. Note that this approach is not necessarily the most efficient because of the relatively expensive serialization and deserialization steps. .NET 3.0 features used in this example includes *automatic properties* on ProspectMemory and *object initialization* on SalesProspect.

Code in project: *DoFactory.GangOfFour.Memento.NetOptimized*

Memento: when and where use it

The intent of the Memento pattern is to provide storage as well as restoration of an object. The mechanism in which you store the object's state depends on the required duration of persistence which could be a few seconds, a few days, or possibly years. Storage options include memory (for example Session), a scratch file, or a database.

In an abstract sense you can view a database as an implementation of the Memento design pattern. However, the most common reason for using this pattern is to capture a snapshot of an object's state so that any subsequent changes can be undone easily if necessary. In .NET the most common implementation is when serializing and de-serializing objects to save and restore an object's state.

Essentially, a Memento is a small repository that stores an object's state. Scenarios in which you may want to restore an object into a state that existed previously include: saving and restoring the state of a player in a computer game or the implementation of an undo operation in a database.

Memento in the .NET Framework

Serialization is the process of converting an object into a linear sequence of bytes for either storage or transmission to another location. Deserialization is the process of taking in stored information and recreating objects from it. The Memento pattern creates a snapshot of an object's state and then offers the ability to restore it to its original state. This is what the serialization architecture of .NET offers and therefore qualifies as an example of Memento in the .NET Framework.

22. Observer

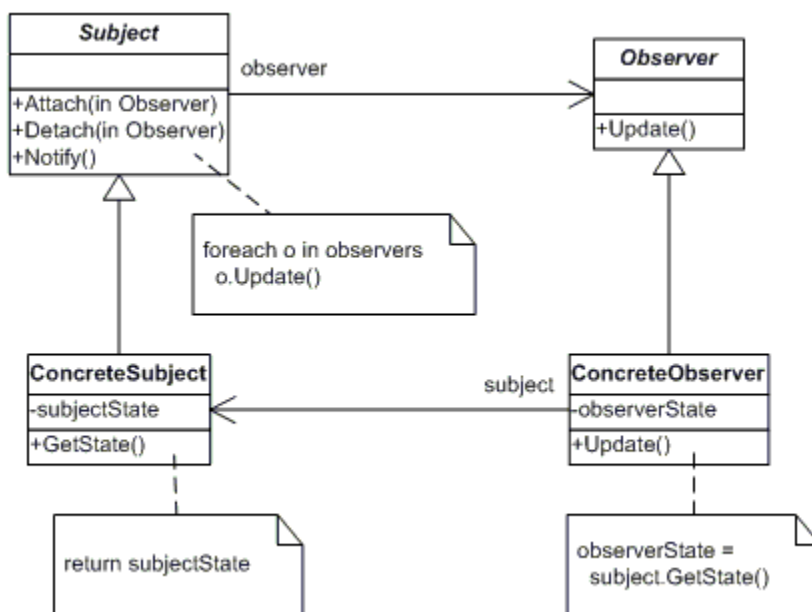
Definition

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



Frequency of use: 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

Structural sample code

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

Code in project: *DoFactory.GangOfFour.Observer.Structural*

Real-world sample code

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

Code in project: *DoFactory.GangOfFour.Observer.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates the same code as above but uses more modern, built-in .NET features. This example uses .NET multicast delegates which are an implementation of the Observer pattern. Delegates are type safe function pointers that have the ability to call a method. *Generic delegates* allow for event handler-specific arguments, that is, the argument named sender does not have to be of type *object*, but can be any type (in this example the type is *Stock*). Multicast delegates are comprised of multiple methods that are called serially in the order in which they were added using the

C# += operator. .NET 3.0 features used in this example include *automatic properties* (C#) and *object initializers*.

Code in project: *DoFactory.GangOfFour.Observer.NetOptimized*

Observer: when and where use it

The Observer design pattern is one of two Gang-of-Four design patterns (the other is the Iterator pattern) that have found their way, not only into the .NET Framework libraries, but also in the .NET languages themselves. When programming a Web application or a Windows application you often work with events and event handlers. Events and Delegates, which are first class language features, act as the *Subject* and *Observers* respectively as defined in the Observer pattern.

The Observer pattern facilitates good object-oriented designs as it promotes loose coupling. Observers register and unregister themselves with subjects that maintain a list of interested observers. The subject does not depend on any particular observer, as long as the delegates are of the correct type for the event. The event and delegate paradigm in .NET represents an elegant and powerful implementation of the Observer design pattern.

Observer in the .NET Framework

As mentioned, the .NET event model is implemented with the Observer design pattern and is found throughout the .NET Framework -- both in the .NET languages and .NET class libraries.

23. State

Definition

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

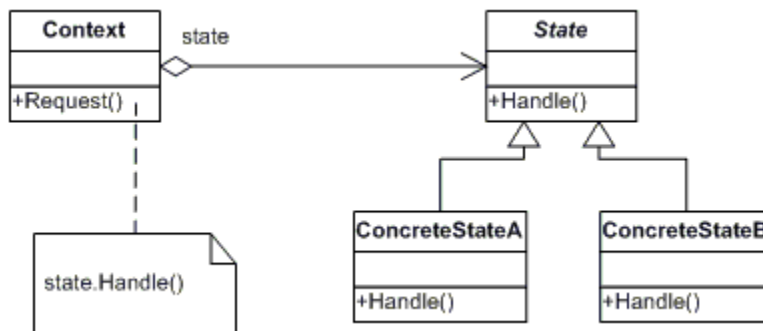


Frequency of use:

1	2	3	4	5

 medium

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

Structural sample code

The 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

Code in project: *DoFactory.GangOfFour.State.Structural*

Real-world sample code

The 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.

Code in project: *DoFactory.GangOfFour.State.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates a Finite State Machine implementation using the State design pattern. This .NET example is interesting in that it combines two patterns: the State design pattern and the Singleton pattern resulting in an effective and yet elegant solution. This example demonstrates a non-deterministic Finite State Machine with 5 possible states (note: in a non-deterministic system state transitions are unpredictable).

Code in project: *DoFactory.GangOfFour.State.NetOptimized*

State: when and where use it

The state of an object is represented by the values of its instance variables. A client can change the state of an object by making property or method calls which in turn change the instance variables. These types of objects are called *stateful* objects. State is frequently a core concept in complex systems, such as, stock market trading systems,

purchasing and requisition systems, document management systems, and especially work-flow system.

The complexity may spread to numerous classes and to contain this complexity you use the State design pattern. In this pattern you encapsulate state specific behavior in a group of related classes each of which represents a different state. This approach reduces the need for intricate and hard-to-trace conditional if and case statements relying instead on polymorphism to implement the correct functionality of a required state transition.

The goal of the State design pattern is to contain state-specific logic in a limited set of objects in which each object represents a particular state. State transition diagrams (also called state machines) are very helpful in modeling these complex systems. The State pattern simplifies programming by distributing the response to a state transition to a limited set of classes in which each one is a representation of a system's state.

State in the .NET Framework

We are not aware of the State patterns being exposed in the .NET Framework.

24. Strategy

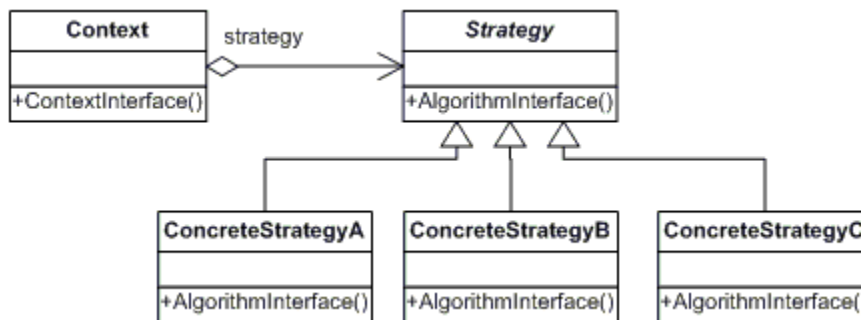
Definition

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



Frequency of use: 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.

Structural sample code

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

Code in project: *DoFactory.GangOfFour.Strategy.Structural*

Real-world sample code

The 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.

Code in project: *DoFactory.GangOfFour.Strategy.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates the same code as above but uses more modern, built-in .NET features. In this example the abstract class SortStrategy has been replaced by the interface ISortStrategy. The setter method SetStrategy has been implemented as a .NET property. The collection of students is implemented using a type-safe *generic* List. The Student class uses .NET 3.0 *automatic properties* (C#) and *object initializers*.

Code in project: *DoFactory.GangOfFour.Strategy.NetOptimized*

Strategy: when and where use it

The Strategy design pattern is widely used. Its intent is to encapsulate alternative strategies for a particular operation. The Strategy pattern is a 'plug-and-play' pattern. The client calls a method on a particular interface which can be swapped out with any other Strategy class that implements the same interface. Strategy is useful in many different scenarios. An example is credit card processing. If a customer on an eShopping site prefers to pay with PayPal over 2Checkout the application can simply swap the PayPal strategy class out for the 2Checkout strategy class.

If the Strategy interface has only a single method you can simplify the implementation by using a delegate rather than an interface. If you think about it, a delegate is a special case of the Strategy pattern. Therefore, you could argue that .NET's event model (which uses delegates as eventhandlers) is built also on the Strategy pattern.

Strategy in the .NET Framework

An example of the Strategy pattern in .NET is the ArrayList which contains several overloaded Sort methods. These methods sort the elements in the list using a given class that implements the IComparer interface. IComparer contains a Sort method that compares two objects and returns a value indicating whether one object is greater than, equal to, or less than the other object. Classes that implement the IComparer interface are implementations of the Strategy design pattern.

25. Template Method

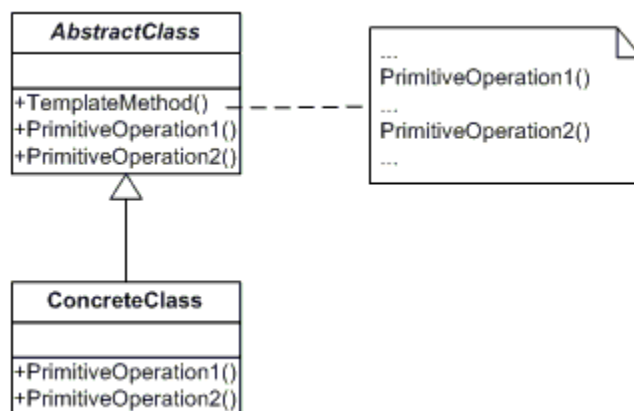
Definition

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..



Frequency of use:  medium

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 to carry out subclass-specific steps of the algorithm

Structural sample code

The 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.

Code in project: *DoFactory.GangOfFour.Template.Structural*

Real-world sample code

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

Code in project: *DoFactory.GangOfFour.Template.RealWorld*

.NET optimized sample code

The .NET optimized example is similar to the RealWorld code as there are no meaningful .NET optimizations.

The Template pattern is often found in UI programming. A recent example is the Layout system in WPF. If you need a custom layout in WPF you create a class that derives from *Panel* and then you override two methods *MeasureOverride()* and *ArrangeOverride()*. This is a beautiful and elegant example of the Template method pattern.

Code in project: *DoFactory.GangOfFour.Template.NetOptimized*

Template Method: when and where use it

The intent of the Template Method design pattern is to provide an outline of a series of steps for an algorithm. Derived classes retain the original structure of the algorithm but

have the option to redefine or adjust certain steps of the algorithm. This pattern is designed to offer extensibility to the client programmer. Template Methods are frequently used when building a class library (for example an application framework) that is going to be used by other client programmers. The examples in the .NET Framework (see next section) demonstrate when and where this pattern can be used.

There are strong similarities between the Template Method and the Strategy pattern. Both are designed for extensibility and customization as they allow the client to alter the way an algorithm or process is executed. The difference is that with Strategy the entire algorithm is changed, whereas the Template method allows individual steps to be redefined. However, their object-oriented implementations are quite different: Strategy uses delegation and Template Method is based on object inheritance.

Template Method in the .NET Framework

The Template Method is frequently used in the .NET Framework. It is through this pattern that .NET provides extensibility to the users of its API. Take a look at custom controls in ASP.NET. Custom controls are created by inheriting from one of the control base classes (Control or WebControl). Out of the box these base classes handle all functionality that are common to all controls, such as initializing, loading, rendering, unloading, and firing control lifecycle events at the right time. Your custom control extends the functionality of these base classes by overriding individual steps in the control generation algorithm, such as the Render and CreateChildControls methods as well as handling certain events, such as the Postback event.

The Control class in the System.Windows.Forms namespace demonstrates usage of the Template Method. These template methods allow the base class designer controlled extensibility by centralizing these methods as a single virtual method. Microsoft suffixes these methods with the word “Core”. The Control class for example has methods named: SetBoundsCore(), ScaleCore(), SetVisibleCore(), and others. Most of these template methods are protected virtual. The code below shows the inner workings of the Control class.

```
// C#  
public class Control
```

```

{
    // Overloaded SetBounds methods
    public void SetBounds(int x, int y, int width, int height)
    {
        ...
        SetBoundsCore(...);
    }

    public void SetBounds(int x, int y, int width, int height,
        BoundsSpecified specified)
    {
        ...
        SetBoundsCore(...);
    }

    // Template method
    protected virtual void SetBoundsCore(int x, int y,
        int width, int height, BoundsSpecified specified)
    {
        // the real code
    }
}

```

```

' VB
Public Class Control
    ' Overloaded SetBounds methods
    Public Sub SetBounds(ByVal x As Integer, ByVal y As Integer, _
        ByVal width As Integer, ByVal height As Integer)

        SetBoundsCore(...)
    End Sub

    Public Sub SetBounds(ByVal x As Integer, ByVal y As Integer, _
        ByVal width As Integer, ByVal height As Integer, _
        ByVal specified As BoundsSpecified)
        ...
        SetBoundsCore(...)
    End Sub

    ' Template method
    Protected Overridable Sub SetBoundsCore(ByVal x As Integer, _
        ByVal y As Integer, ByVal width As Integer, _
        ByVal height As Integer, ByVal specified As BoundsSpecified)

        ' the real code
    End Sub
End Class


```

26. Visitor

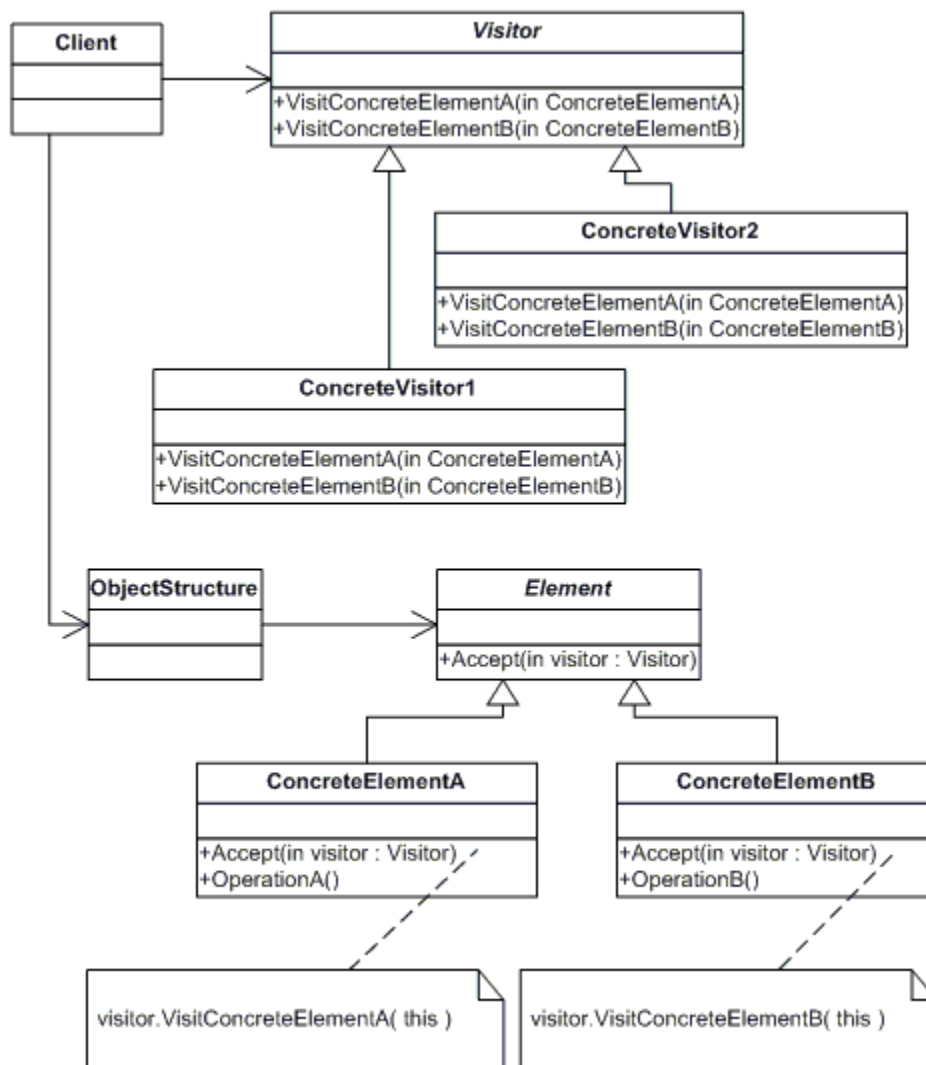
Definition

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.



Frequency of use:  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

Structural sample code

The 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.

Code in project: *DoFactory.GangOfFour.Visitor.Structural*

Real-world sample code

The 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.

Code in project: *DoFactory.GangOfFour.Visitor.RealWorld*

.NET optimized sample code

The .NET optimized code demonstrates the same code as above but uses more modern, built-in .NET features. In this example the Visitor pattern uses reflection. This approach lets us selectively choose which Employee we'd like to visit. `ReflectiveVisit()` looks for a `Visit()` method with a compatible parameter. Note that the use of reflection makes the Visitor pattern more flexible but also slower and more complex. This sample holds the collection of employees in a generic class `List<Employee>` (`List(Of Employee)` in VB). .NET 3.0 features in this example includes *automatic properties* (C#) in the Employee class as well as the *Foreach extension method* in the Employees class.

Code in project: *DoFactory.GangOfFour.Visitor.NetOptimized*

Visitor: when and where use it

You may find yourself in a situation where you need to make a functional change to a collection of classes but you do not have control over the classes in the hierarchy. This is where the Visitor pattern comes in: the intent of the Visitor design pattern is to define a new operation for a collection of classes (i.e. the classes being visited) without changing the hierarchy itself. The new logic lives in a separate class, the Visitor. The tricky aspect of the Visitor pattern is that the original developer of the classes in the collection must have anticipated functional adjustments that may occur in the future by including methods that accept a Visitor class and let it define new operations.

Visitor is a controversial pattern and expert .NET developers frequently avoid using it. They argue that it adds complexity and creates fragile code that goes against generally accepted best practices in object-oriented design. When deciding to use this pattern it is

usually best to carefully weight it against alternative approaches, such as inheritance or delegation, which most likely offer a more robust solution to your problem.

Visitor in .NET Framework

The Visitor pattern is not used in the .NET Framework libraries.