NET Design Patterns

Design patterns are solutions to software design problems you find again and again in real-world application development. Patterns are about reusable designs and interactions of objects..

The 23 Gang of Four (GoF) patterns are generally considered the foundation for all other patterns. They are categorized in three groups: Creational, Structural, and Behavioral (for a complete list see below).

To give you a head start, the C# source code for each pattern is provided in 2 forms:*structural* and *real-world*. 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 these patterns.

A third form, *.NET optimized*, demonstrates design patterns that exploit built-in .NET 4.5 features, such as, generics, attributes, delegates, reflection, and more. These and much more are available in our .NET Design Pattern Framework 4.5. See our Singleton page for a .NET 4.5 Optimized code sample.

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

Structural Patterns

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

Behavioral Patterns

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

Factory Method Design Pattern

- definition
- UML diagram
- sample code in C#

▸ participants

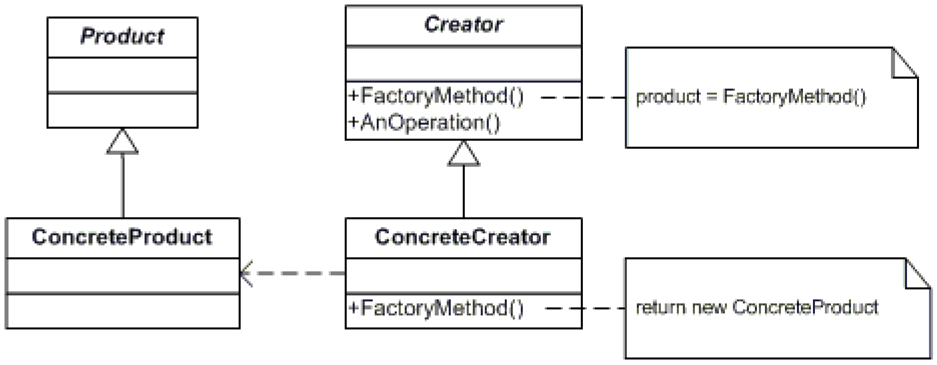
definition

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

return to top

UML class diagram



return to top

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.

return to top

sample code in C#

This 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.

Hide code

```
// Factory Method pattern -- Structural example


using System;


namespace DoFactory.GangOfFour.Factory.Structural

{

  /// <summary>

  /// MainApp startup class for Structural

  /// Factory Method Design Pattern.

  /// </summary>

  class MainApp

  {

    /// <summary>

    /// Entry point into console application.

    /// </summary>

    static void Main()

    {

      // An array of creators

      Creator[] creators = new Creator[2];


      creators[0] = new ConcreteCreatorA();

      creators[1] = new ConcreteCreatorB();
      // Iterate over creators and create products
      foreach (Creator creator in creators)
      {
        Product product = creator.FactoryMethod();
        Console.WriteLine("Created {0}",
          product.GetType().Name);
      }

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Product' abstract class
  /// </summary>
  abstract class Product
  {
  }

  /// <summary>
  /// A 'ConcreteProduct' class
  /// </summary>
  class ConcreteProductA : Product
  {
  }

  /// <summary>
  /// A 'ConcreteProduct' class
  /// </summary>
  class ConcreteProductB : Product
  {
  }
```

```
  /// <summary>
  /// The 'Creator' abstract class
  /// </summary>
  abstract class Creator
  {
    public abstract Product FactoryMethod();
  }

  /// <summary>
  /// A 'ConcreteCreator' class
  /// </summary>
  class ConcreteCreatorA : Creator
  {
    public override Product FactoryMethod()
    {
      return new ConcreteProductA();
    }
  }

  /// <summary>
  /// A 'ConcreteCreator' class
  /// </summary>
  class ConcreteCreatorB : Creator
  {
    public override Product FactoryMethod()
    {
      return new ConcreteProductB();
    }
  }
}
```

Output

```
Created ConcreteProductA
Created ConcreteProductB
```

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

Hide code

```
// Factory Method pattern -- Real World example


using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Factory.RealWorld
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Factory Method Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // Note: constructors call Factory Method
      Document[] documents = new Document[2];
    documents[0] = new Resume();
    documents[1] = new Report();

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

    // Wait for user
    Console.ReadKey();
  }
}

/// <summary>
/// The 'Product' abstract class
/// </summary>
abstract class Page
{
}

/// <summary>
/// A 'ConcreteProduct' class
/// </summary>
class SkillsPage : Page
{
}

/// <summary>
/// A 'ConcreteProduct' class
/// </summary>
class EducationPage : Page
{
}

/// <summary>
/// A 'ConcreteProduct' class
/// </summary>
class ExperiencePage : Page
{
}

/// <summary>
/// A 'ConcreteProduct' class
/// </summary>
class IntroductionPage : Page
{
}

/// <summary>
/// A 'ConcreteProduct' class
/// </summary>
class ResultsPage : Page
{
}

/// <summary>
/// A 'ConcreteProduct' class
/// </summary>
class ConclusionPage : Page
{
}
```

```
/// <summary>
/// A 'ConcreteProduct' class
/// </summary>
class SummaryPage : Page
{
}

/// <summary>
/// A 'ConcreteProduct' class
/// </summary>
class BibliographyPage : Page
{
}

/// <summary>
/// The 'Creator' abstract class
/// </summary>
abstract class Document
{
  private List<Page> _pages = new List<Page>();

  // Constructor calls abstract Factory method
  public Document()
  {
    this.CreatePages();
  }

  public List<Page> Pages
  {
    get { return _pages; }
  }
  // Factory Method
  public abstract void CreatePages();
}

/// <summary>
/// A 'ConcreteCreator' class
/// </summary>
class Resume : Document
{
  // Factory Method implementation
  public override void CreatePages()
  {
    Pages.Add(new SkillsPage());
    Pages.Add(new EducationPage());
    Pages.Add(new ExperiencePage());
  }
}

/// <summary>
/// A 'ConcreteCreator' class
/// </summary>
class Report : Document
{
  // Factory Method implementation
  public override void CreatePages()
  {
    Pages.Add(new IntroductionPage());
    Pages.Add(new ResultsPage());
    Pages.Add(new ConclusionPage());
    Pages.Add(new SummaryPage());
    Pages.Add(new BibliographyPage());
  }
}
```

```
}
```

Output

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

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

This .NET optimized code demonstrates the same real-world situation as above but uses modern, built-in .NET features, such as, generics, reflection, object initializers, automatic properties, etc.

Hide code

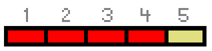
```
// Factory Method pattern -- .NET optimized
```

Singleton Design Pattern

- definition
- UML diagram
- participants
- sample code in C#

definition

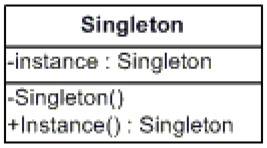
Ensure a class has only one instance and provide a global point of access to it.

Frequency of use: 

medium high

return to top

UML class diagram



return to top

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.

return to top

sample code in C#

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

Hide code

```
// Singleton pattern -- Structural example



using System;



namespace DoFactory.GangOfFour.Singleton.Structural
```

```
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Singleton Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // Constructor is protected -- cannot use new
      Singleton s1 = Singleton.Instance();
      Singleton s2 = Singleton.Instance();

      // Test for same instance
      if (s1 == s2)
      {
        Console.WriteLine("Objects are the same instance");
      }

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Singleton' class
  /// </summary>
  class Singleton
  {
    private static Singleton _instance;
    // Constructor is 'protected'
    protected Singleton()
    {
    }

    public static Singleton Instance()
    {
      // Uses lazy initialization.
      // Note: this is not thread safe.
      if (_instance == null)
      {
        _instance = new Singleton();
      }

      return _instance;
    }
  }
}
```

Output

```
Objects are the same instance
```

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

Hide code

```
// Singleton pattern -- Real World example


using System;

using System.Collections.Generic;

using System.Threading;


namespace DoFactory.GangOfFour.Singleton.RealWorld

{

  /// <summary>

  /// MainApp startup class for Real-World

  /// Singleton Design Pattern.

  /// </summary>

  class MainApp

  {

    /// <summary>

    /// Entry point into console application.

    /// </summary>

    static void Main()

    {

      LoadBalancer b1 = LoadBalancer.GetLoadBalancer();

      LoadBalancer b2 = LoadBalancer.GetLoadBalancer();

      LoadBalancer b3 = LoadBalancer.GetLoadBalancer();

      LoadBalancer b4 = LoadBalancer.GetLoadBalancer();


      // Same instance?

      if (b1 == b2 && b2 == b3 && b3 == b4)

      {

        Console.WriteLine("Same instance\n");

      }


      // Load balance 15 server requests

      LoadBalancer balancer = LoadBalancer.GetLoadBalancer();

      for (int i = 0; i < 15; i++)
      {
        string server = balancer.Server;
        Console.WriteLine("Dispatch Request to: " + server);
      }

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Singleton' class
  /// </summary>
  class LoadBalancer
  {
    private static LoadBalancer _instance;
    private List<string> _servers = new List<string>();
    private Random _random = new Random();

    // Lock synchronization object
    private static object syncLock = new object();

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

```
    public static LoadBalancer GetLoadBalancer()
    {
      // Support multithreaded applications through
      // 'Double checked locking' pattern which (once
      // the instance exists) avoids locking each
      // time the method is invoked
      if (_instance == null)
      {
        lock (syncLock)
        {
          if (_instance == null)
          {
            _instance = new LoadBalancer();
          }
        }
      }

      return _instance;
    }

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

Output

```
Same instance

ServerIII
ServerII
ServerI
ServerII
ServerI
ServerIII
ServerI
ServerIII
ServerIV
ServerII
ServerII
ServerIII
ServerIV
ServerII
ServerIV
```

This .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.

Hide code

```
// Singleton pattern -- .NET optimized



using System;

using System.Collections.Generic;


namespace DoFactory.GangOfFour.Singleton.NETOptimized

{

  /// <summary>

  /// MainApp startup class for .NET optimized
/// Singleton Design Pattern.

/// </summary>

class MainApp

{

  /// <summary>

  /// Entry point into console application.

  /// </summary>

  static void Main()

  {

    LoadBalancer b1 = LoadBalancer.GetLoadBalancer();

    LoadBalancer b2 = LoadBalancer.GetLoadBalancer();

    LoadBalancer b3 = LoadBalancer.GetLoadBalancer();

    LoadBalancer b4 = LoadBalancer.GetLoadBalancer();


    // Confirm these are the same instance

    if (b1 == b2 && b2 == b3 && b3 == b4)

    {

      Console.WriteLine("Same instance\n");

    }


    // Next, load balance 15 requests for a server

    LoadBalancer balancer = LoadBalancer.GetLoadBalancer();

    for (int i = 0; i < 15; i++)

    {

      string serverName = balancer.NextServer.Name;

      Console.WriteLine("Dispatch request to: " + serverName);

    }


    // Wait for user

    Console.ReadKey();

  }

}
```

```
/// <summary>
/// The 'Singleton' class
/// </summary>
sealed class LoadBalancer
{
  // Static members are 'eagerly initialized', that is,
  // immediately when class is loaded for the first time.
  // .NET guarantees thread safety for static initialization
  private static readonly LoadBalancer _instance =
    new LoadBalancer();

  // Type-safe generic list of servers
  private List<Server> _servers;
  private Random _random = new Random();

  // Note: constructor is 'private'
  private LoadBalancer()
  {
    // Load list of available servers
    _servers = new List<Server>
      {
       new Server{ Name = "ServerI", IP = "120.14.220.18" },
       new Server{ Name = "ServerII", IP = "120.14.220.19" },
       new Server{ Name = "ServerIII", IP = "120.14.220.20" },
       new Server{ Name = "ServerIV", IP = "120.14.220.21" },
       new Server{ Name = "ServerV", IP = "120.14.220.22" },
      };
  }

  public static LoadBalancer GetLoadBalancer()
  {
    return _instance;
  }
```

```
    // Simple, but effective load balancer
    public Server NextServer
    {
      get
      {
        int r = _random.Next(_servers.Count);
        return _servers[r];
      }
    }
  }

  /// <summary>
  /// Represents a server machine
  /// </summary>
  class Server
  {
    // Gets or sets server name
    public string Name { get; set; }

    // Gets or sets server IP address
    public string IP { get; set; }
  }
}
```

Output

```
Same instance

ServerIV
ServerIV
ServerIII
ServerV
ServerII
ServerV
ServerII
ServerII
ServerI
ServerIV
ServerIV
ServerII
ServerI
ServerV
ServerIV
```

Abstract Factory Design Pattern

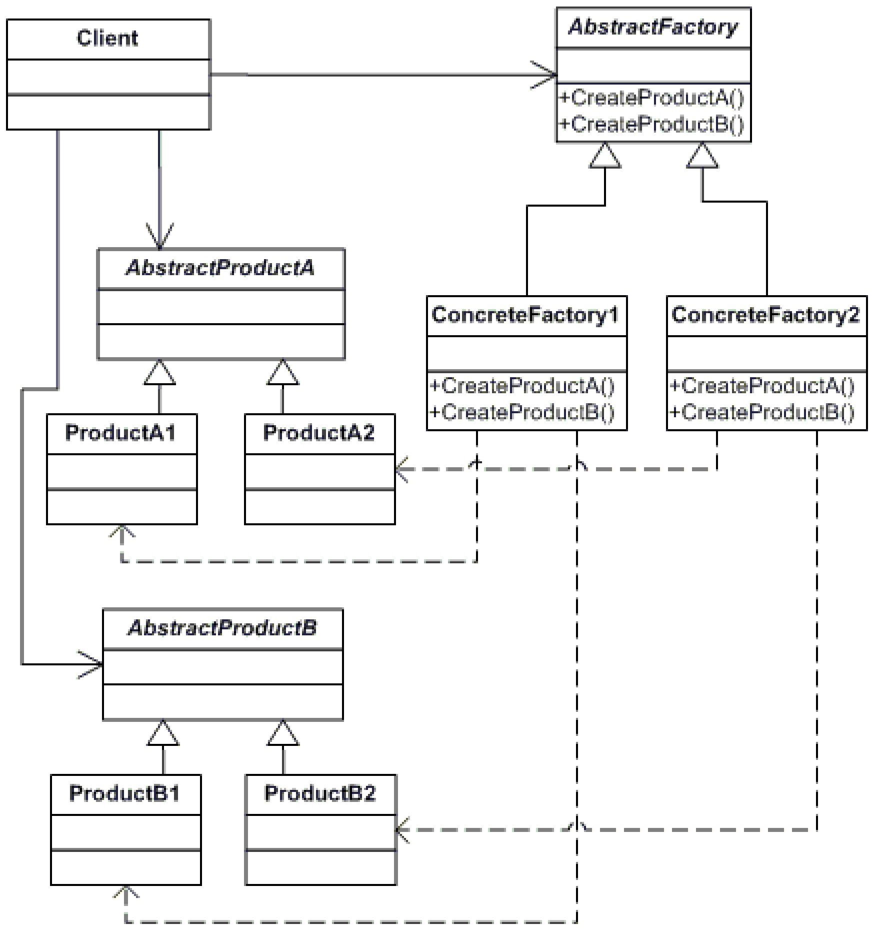
definition

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

Frequency of use: 🟥🟥🟥🟥🟥 high

return to top

UML class diagram



return to top

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

return to top

sample code in C#

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

Hide code

```
// Abstract Factory pattern -- Structural example



using System;


namespace DoFactory.GangOfFour.Abstract.Structural
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Abstract Factory Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
      // Abstract factory #1
      AbstractFactory factory1 = new ConcreteFactory1();
      Client client1 = new Client(factory1);
      client1.Run();

      // Abstract factory #2
      AbstractFactory factory2 = new ConcreteFactory2();
      Client client2 = new Client(factory2);
      client2.Run();
      // Wait for user input
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'AbstractFactory' abstract class
  /// </summary>
  abstract class AbstractFactory
  {
    public abstract AbstractProductA CreateProductA();
    public abstract AbstractProductB CreateProductB();
  }


  /// <summary>
  /// The 'ConcreteFactory1' class
  /// </summary>
  class ConcreteFactory1 : AbstractFactory
  {
    public override AbstractProductA CreateProductA()
    {
      return new ProductA1();
    }
    public override AbstractProductB CreateProductB()
    {
      return new ProductB1();
    }
  }

  /// <summary>
  /// The 'ConcreteFactory2' class
  /// </summary>
  class ConcreteFactory2 : AbstractFactory
  {
    public override AbstractProductA CreateProductA()
    {
      return new ProductA2();
    }
    public override AbstractProductB CreateProductB()
    {
      return new ProductB2();
    }
  }

  /// <summary>
  /// The 'AbstractProductA' abstract class
  /// </summary>
  abstract class AbstractProductA
  {
  }

  /// <summary>
  /// The 'AbstractProductB' abstract class
  /// </summary>
  abstract class AbstractProductB
  {
    public abstract void Interact(AbstractProductA a);
  }



  /// <summary>
  /// The 'ProductA1' class
  /// </summary>
  class ProductA1 : AbstractProductA
```

```
{
}

/// <summary>
/// The 'ProductB1' class
/// </summary>
class ProductB1 : AbstractProductB
{
  public override void Interact(AbstractProductA a)
  {
    Console.WriteLine(this.GetType().Name +
      " interacts with " + a.GetType().Name);
  }
}

/// <summary>
/// The 'ProductA2' class
/// </summary>
class ProductA2 : AbstractProductA
{
}

/// <summary>
/// The 'ProductB2' class
/// </summary>
class ProductB2 : AbstractProductB
{
  public override void Interact(AbstractProductA a)
  {
    Console.WriteLine(this.GetType().Name +
      " interacts with " + a.GetType().Name);
  }
}
```

```
  /// <summary>
  /// The 'Client' class. Interaction environment for the products.
  /// </summary>
  class Client
  {
    private AbstractProductA _abstractProductA;
    private AbstractProductB _abstractProductB;

    // Constructor
    public Client(AbstractFactory factory)
    {
      _abstractProductB = factory.CreateProductB();
      _abstractProductA = factory.CreateProductA();
    }

    public void Run()
    {
      _abstractProductB.Interact(_abstractProductA);
    }
  }
}
```

Output

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

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

Hide code

```
// Abstract Factory pattern -- Real World example



using System;


namespace DoFactory.GangOfFour.Abstract.RealWorld
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Abstract Factory Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
      // Create and run the African animal world
      ContinentFactory africa = new AfricaFactory();
      AnimalWorld world = new AnimalWorld(africa);
      world.RunFoodChain();

      // Create and run the American animal world
      ContinentFactory america = new AmericaFactory();
      world = new AnimalWorld(america);
      world.RunFoodChain();

      // Wait for user input
      Console.ReadKey();
    }
  }



  /// <summary>
  /// The 'AbstractFactory' abstract class
  /// </summary>
  abstract class ContinentFactory
  {
    public abstract Herbivore CreateHerbivore();
    public abstract Carnivore CreateCarnivore();
  }

  /// <summary>
  /// The 'ConcreteFactory1' class
  /// </summary>
  class AfricaFactory : ContinentFactory
  {
    public override Herbivore CreateHerbivore()
    {
      return new Wildebeest();
    }
    public override Carnivore CreateCarnivore()
    {
      return new Lion();
    }
  }

  /// <summary>
  /// The 'ConcreteFactory2' class
  /// </summary>
  class AmericaFactory : ContinentFactory
```

```
{
  public override Herbivore CreateHerbivore()
  {
    return new Bison();
  }
  public override Carnivore CreateCarnivore()
  {
    return new Wolf();
  }
}

/// <summary>
/// The 'AbstractProductA' abstract class
/// </summary>
abstract class Herbivore
{
}

/// <summary>
/// The 'AbstractProductB' abstract class
/// </summary>
abstract class Carnivore
{
  public abstract void Eat(Herbivore h);
}

/// <summary>
/// The 'ProductA1' class
/// </summary>
class Wildebeest : Herbivore
{
}
```

```
/// <summary>
/// The 'ProductB1' class
/// </summary>
class Lion : Carnivore
{
  public override void Eat(Herbivore h)
  {
    // Eat Wildebeest
    Console.WriteLine(this.GetType().Name +
      " eats " + h.GetType().Name);
  }
}

/// <summary>
/// The 'ProductA2' class
/// </summary>
class Bison : Herbivore
{
}

/// <summary>
/// The 'ProductB2' class
/// </summary>
class Wolf : Carnivore
{
  public override void Eat(Herbivore h)
  {
    // Eat Bison
    Console.WriteLine(this.GetType().Name +
      " eats " + h.GetType().Name);
  }
}
```

```
  /// <summary>
  /// The 'Client' class
  /// </summary>
  class AnimalWorld
  {
    private Herbivore _herbivore;
    private Carnivore _carnivore;

    // Constructor
    public AnimalWorld(ContinentFactory factory)
    {
      _carnivore = factory.CreateCarnivore();
      _herbivore = factory.CreateHerbivore();
    }

    public void RunFoodChain()
    {
      _carnivore.Eat(_herbivore);
    }
  }
}
```

Output

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

This .NET optimized code demonstrates the same real-world situation as above but uses modern, built-in .NET features, such as, generics, reflection, object initializers, automatic properties, etc.

Hide code

```
// Abstract Factory pattern -- .NET optimized
```

Adapter Design Pattern

- definition
- UML diagram
- participants
- sample code in C#

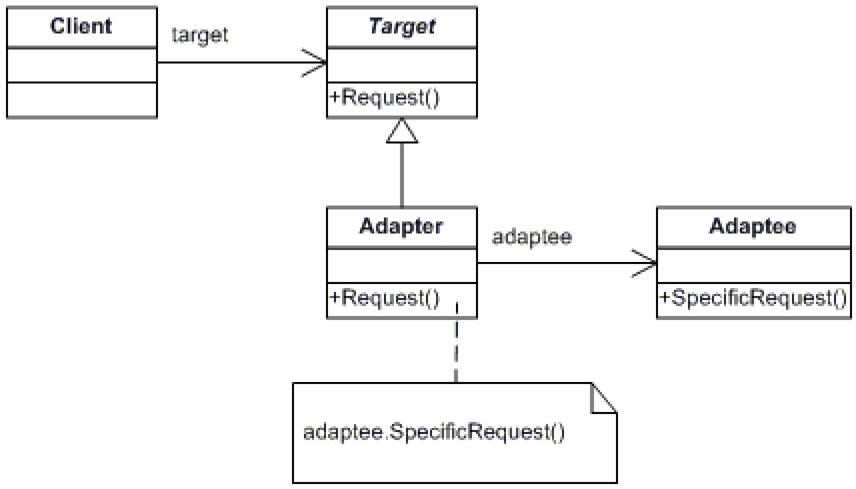
definition

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

return to top

UML class diagram



return to top

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.

return to top

sample code in C#

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

Hide code

```
// Adapter pattern -- Structural example



using System;


namespace DoFactory.GangOfFour.Adapter.Structural

{

  /// <summary>

  /// MainApp startup class for Structural

  /// Adapter Design Pattern.

  /// </summary>

  class MainApp

  {

    /// <summary>

    /// Entry point into console application.

    /// </summary>
    static void Main()
    {
      // Create adapter and place a request
      Target target = new Adapter();
      target.Request();

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Target' class
  /// </summary>
  class Target
  {
    public virtual void Request()
    {
      Console.WriteLine("Called Target Request()");
    }
  }

  /// <summary>
  /// The 'Adapter' class
  /// </summary>
  class Adapter : Target
  {
    private Adaptee _adaptee = new Adaptee();

    public override void Request()
    {
      // Possibly do some other work
      //  and then call SpecificRequest
      _adaptee.SpecificRequest();

    }

  }


  /// <summary>

  /// The 'Adaptee' class

  /// </summary>

  class Adaptee

  {

    public void SpecificRequest()

    {

      Console.WriteLine("Called SpecificRequest()");

    }

  }

}
```

Output

```
Called SpecificRequest()
```

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

Hide code

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



using System;



namespace DoFactory.GangOfFour.Adapter.RealWorld
```

```
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Adapter Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // Non-adapted chemical compound
      Compound unknown = new Compound("Unknown");
      unknown.Display();

      // Adapted chemical compounds
      Compound water = new RichCompound("Water");
      water.Display();

      Compound benzene = new RichCompound("Benzene");
      benzene.Display();

      Compound ethanol = new RichCompound("Ethanol");
      ethanol.Display();

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Target' class
/// </summary>
class Compound
{
  protected string _chemical;
  protected float _boilingPoint;
  protected float _meltingPoint;
  protected double _molecularWeight;
  protected string _molecularFormula;

  // Constructor
  public Compound(string chemical)
  {
    this._chemical = chemical;
  }

  public virtual void Display()
  {
    Console.WriteLine("\nCompound: {0} ------ ", _chemical);
  }
}

/// <summary>
/// The 'Adapter' class
/// </summary>
class RichCompound : Compound
{
  private ChemicalDatabank _bank;

  // Constructor
  public RichCompound(string name)
    : base(name)
  {
  }
  public override void Display()
  {
    // The Adaptee
    _bank = new ChemicalDatabank();

    _boilingPoint = _bank.GetCriticalPoint(_chemical, "B");
    _meltingPoint = _bank.GetCriticalPoint(_chemical, "M");
    _molecularWeight = _bank.GetMolecularWeight(_chemical);
    _molecularFormula = _bank.GetMolecularStructure(_chemical);

    base.Display();
    Console.WriteLine(" Formula: {0}", _molecularFormula);
    Console.WriteLine(" Weight : {0}", _molecularWeight);
    Console.WriteLine(" Melting Pt: {0}", _meltingPoint);
    Console.WriteLine(" Boiling Pt: {0}", _boilingPoint);
  }
}

/// <summary>
/// The 'Adaptee' class
/// </summary>
class ChemicalDatabank
{
  // The databank 'legacy API'
  public float GetCriticalPoint(string compound, string point)
  {
    // Melting Point
    if (point == "M")
    {
      switch (compound.ToLower())
      {
        case "water": return 0.0f;
        case "benzene": return 5.5f;
        case "ethanol": return -114.1f;
        default: return 0f;
      }
    }
    // Boiling Point
    else
    {
      switch (compound.ToLower())
      {
        case "water": return 100.0f;
        case "benzene": return 80.1f;
        case "ethanol": return 78.3f;
        default: return 0f;
      }
    }
  }

  public string GetMolecularStructure(string compound)
  {
    switch (compound.ToLower())
    {
      case "water": return "H20";
      case "benzene": return "C6H6";
      case "ethanol": return "C2H5OH";
      default: return "";
    }
  }

  public double GetMolecularWeight(string compound)
  {
    switch (compound.ToLower())
    {
        case "water": return 18.015;

        case "benzene": return 78.1134;

        case "ethanol": return 46.0688;

        default: return 0d;

      }

    }

  }

}
```

Output

```
Compound: Unknown ------

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

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

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

This .NET optimized code demonstrates the same real-world situation as above but uses modern, built-in .NET features, such as, generics, reflection, object initializers, automatic properties, etc.

Hide code

```
// Adapter pattern -- .NET optimized
```

Decorator Design Pattern

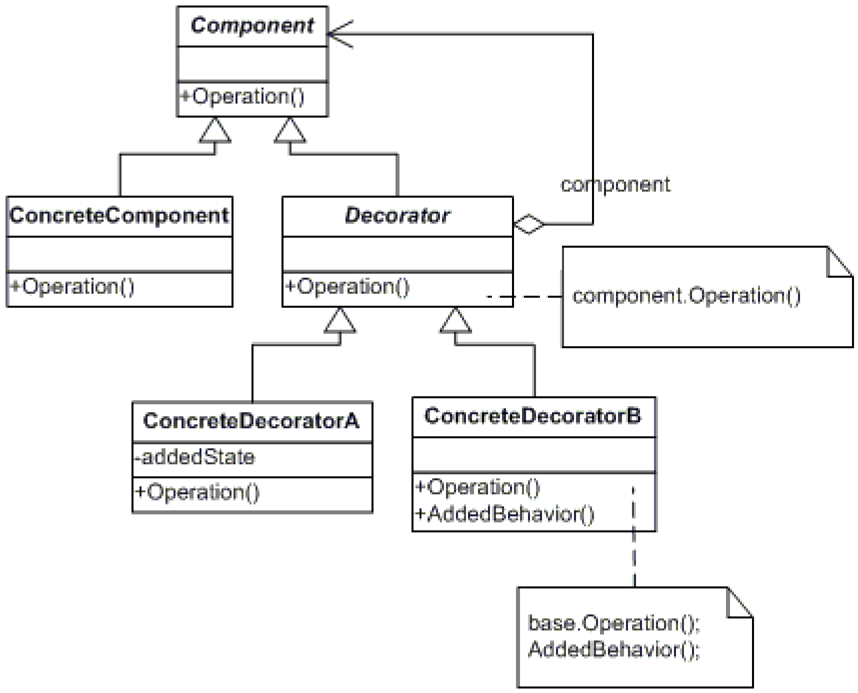
definition

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

Frequency of use: 🟥🟥🟥🟨🟨 medium

return to top

UML class diagram



return to top

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.

return to top

sample code in C#

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

Hide code

```
// Decorator pattern -- Structural example



using System;


namespace DoFactory.GangOfFour.Decorator.Structural

{

  /// <summary>

  /// MainApp startup class for Structural

  /// Decorator Design Pattern.

  /// </summary>

  class MainApp

  {

    /// <summary>

    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // Create ConcreteComponent and two Decorators
      ConcreteComponent c = new ConcreteComponent();
      ConcreteDecoratorA d1 = new ConcreteDecoratorA();
      ConcreteDecoratorB d2 = new ConcreteDecoratorB();

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

      d2.Operation();

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Component' abstract class
  /// </summary>
  abstract class Component
  {
    public abstract void Operation();
  }

  /// <summary>
  /// The 'ConcreteComponent' class
  /// </summary>
  class ConcreteComponent : Component
  {
    public override void Operation()
    {
      Console.WriteLine("ConcreteComponent.Operation()");
    }
  }

  /// <summary>
  /// The 'Decorator' abstract class
  /// </summary>
  abstract class Decorator : Component
  {
    protected Component component;

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

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

  /// <summary>
  /// The 'ConcreteDecoratorA' class
  /// </summary>
  class ConcreteDecoratorA : Decorator
  {
    public override void Operation()
    {
      base.Operation();
      Console.WriteLine("ConcreteDecoratorA.Operation()");
    }
  }

  /// <summary>
  /// The 'ConcreteDecoratorB' class
  /// </summary>
  class ConcreteDecoratorB : Decorator
  {
    public override void Operation()
    {
      base.Operation();
      AddedBehavior();
      Console.WriteLine("ConcreteDecoratorB.Operation()");
    }

    void AddedBehavior()
    {
    }
  }
}
```

Output

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

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

Hide code

```
// Decorator pattern -- Real World example


using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Decorator.RealWorld
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Decorator Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // Create book
      Book book = new Book("Worley", "Inside ASP.NET", 10);
      book.Display();

      // Create video
      Video video = new Video("Spielberg", "Jaws", 23, 92);
      video.Display();

      // Make video borrowable, then borrow and display
      Console.WriteLine("\nMaking video borrowable:");
      Borrowable borrowvideo = new Borrowable(video);
      borrowvideo.BorrowItem("Customer #1");
      borrowvideo.BorrowItem("Customer #2");

      borrowvideo.Display();

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Component' abstract class
  /// </summary>
  abstract class LibraryItem
  {
    private int _numCopies;

    // Property
    public int NumCopies
    {
      get { return _numCopies; }
      set { _numCopies = value; }
    }

    public abstract void Display();
  }

  /// <summary>
  /// The 'ConcreteComponent' class
  /// </summary>
  class Book : LibraryItem
  {
    private string _author;
    private string _title;

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

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

  /// <summary>
  /// The 'ConcreteComponent' class
  /// </summary>
  class Video : LibraryItem
  {
    private string _director;
    private string _title;
    private int _playTime;

    // Constructor
    public Video(string director, string title,
      int numCopies, int playTime)
    {
```

```
    this._director = director;
    this._title = title;
    this.NumCopies = numCopies;
    this._playTime = playTime;
  }

  public override void Display()
  {
    Console.WriteLine("\nVideo ----- ");
    Console.WriteLine(" Director: {0}", _director);
    Console.WriteLine(" Title: {0}", _title);
    Console.WriteLine(" # Copies: {0}", NumCopies);
    Console.WriteLine(" Playtime: {0}\n", _playTime);
  }
}

/// <summary>
/// The 'Decorator' abstract class
/// </summary>
abstract class Decorator : LibraryItem
{
  protected LibraryItem libraryItem;

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

  public override void Display()
  {
    libraryItem.Display();
  }
```

```
}

/// <summary>
/// The 'ConcreteDecorator' class
/// </summary>
class Borrowable : Decorator
{
  protected List<string> borrowers = new List<string>();

  // Constructor
  public Borrowable(LibraryItem libraryItem)
    : base(libraryItem)
  {
  }

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

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

  public override void Display()
  {
    base.Display();

    foreach (string borrower in borrowers)
    {
        Console.WriteLine(" borrower: " + borrower);
      }
    }
  }
}
```

Output

```
Book ------
Author: Worley
Title: Inside ASP.NET
# Copies: 10

Video -----
Director: Spielberg
Title: Jaws
# Copies: 23
Playtime: 92


Making video borrowable:

Video -----
Director: Spielberg
Title: Jaws
# Copies: 21
Playtime: 92

borrower: Customer #1
borrower: Customer #2
```

This .NET optimized code demonstrates the same real-world situation as above but uses modern, built-in .NET features, such as, generics, reflection, object initializers, automatic properties, etc.

Hide code

```
// Decorator pattern -- .NET optimized
```

Facade Design Pattern

- definition
- UML diagram
- participants
- sample code in C#

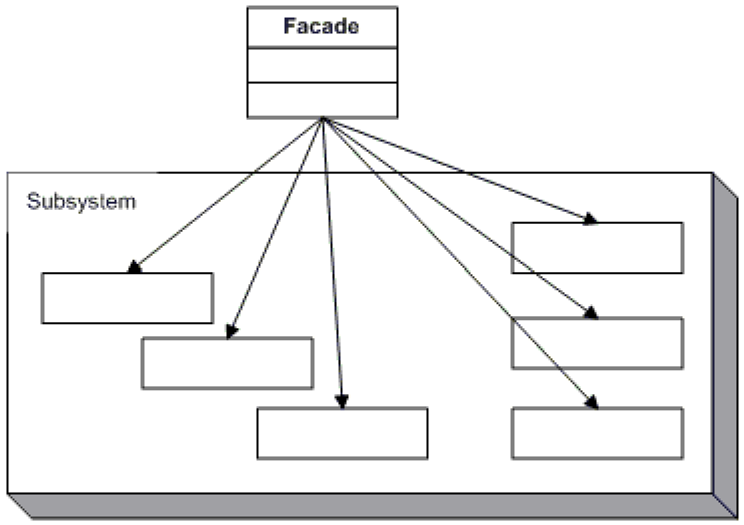
definition

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: 🟥🟥🟥🟥🟥 high

return to top

UML class diagram



return to top

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.

return to top

sample code in C#

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

Hide code

```
// Facade pattern -- Structural example


using System;

namespace DoFactory.GangOfFour.Facade.Structural
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Facade Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
```

```
    Facade facade = new Facade();

    facade.MethodA();
    facade.MethodB();

    // Wait for user
    Console.ReadKey();
  }
}

/// <summary>
/// The 'Subsystem ClassA' class
/// </summary>
class SubSystemOne
{
  public void MethodOne()
  {
    Console.WriteLine(" SubSystemOne Method");
  }
}

/// <summary>
/// The 'Subsystem ClassB' class
/// </summary>
class SubSystemTwo
{
  public void MethodTwo()
  {
    Console.WriteLine(" SubSystemTwo Method");
  }
}

/// <summary>
/// The 'Subsystem ClassC' class
/// </summary>
class SubSystemThree
{
  public void MethodThree()
  {
    Console.WriteLine(" SubSystemThree Method");
  }
}

/// <summary>
/// The 'Subsystem ClassD' class
/// </summary>
class SubSystemFour
{
  public void MethodFour()
  {
    Console.WriteLine(" SubSystemFour Method");
  }
}

/// <summary>
/// The 'Facade' class
/// </summary>
class Facade
{
  private SubSystemOne _one;
  private SubSystemTwo _two;
  private SubSystemThree _three;
  private SubSystemFour _four;

  public Facade()
  {
      _one = new SubSystemOne();
      _two = new SubSystemTwo();
      _three = new SubSystemThree();
      _four = new SubSystemFour();
    }

    public void MethodA()
    {
      Console.WriteLine("\nMethodA() ---- ");
      _one.MethodOne();
      _two.MethodTwo();
      _four.MethodFour();
    }

    public void MethodB()
    {
      Console.WriteLine("\nMethodB() ---- ");
      _two.MethodTwo();
      _three.MethodThree();
    }
  }
}
```

Output

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

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

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

Hide code

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



using System;


namespace DoFactory.GangOfFour.Facade.RealWorld

{

  /// <summary>

  /// MainApp startup class for Real-World

  /// Facade Design Pattern.

  /// </summary>

  class MainApp

  {

    /// <summary>

    /// Entry point into console application.

    /// </summary>

    static void Main()

    {

      // Facade

      Mortgage mortgage = new Mortgage();


      // Evaluate mortgage eligibility for customer

      Customer customer = new Customer("Ann McKinsey");

      bool eligible = mortgage.IsEligible(customer, 125000);


      Console.WriteLine("\n" + customer.Name +

          " has been " + (eligible ? "Approved" : "Rejected"));
      // Wait for user

      Console.ReadKey();

    }

  }


  /// <summary>

  /// The 'Subsystem ClassA' class

  /// </summary>

  class Bank

  {

    public bool HasSufficientSavings(Customer c, int amount)

    {

      Console.WriteLine("Check bank for " + c.Name);

      return true;

    }

  }


  /// <summary>

  /// The 'Subsystem ClassB' class

  /// </summary>

  class Credit

  {

    public bool HasGoodCredit(Customer c)

    {

      Console.WriteLine("Check credit for " + c.Name);

      return true;

    }

  }


  /// <summary>

  /// The 'Subsystem ClassC' class

  /// </summary>

  class Loan
```

```
{
  public bool HasNoBadLoans(Customer c)
  {
    Console.WriteLine("Check loans for " + c.Name);
    return true;
  }
}

/// <summary>
/// Customer class
/// </summary>
class Customer
{
  private string _name;

  // Constructor
  public Customer(string name)
  {
    this._name = name;
  }

  // Gets the name
  public string Name
  {
    get { return _name; }
  }
}

/// <summary>
/// The 'Facade' class
/// </summary>
class Mortgage
{
    private Bank _bank = new Bank();
    private Loan _loan = new Loan();
    private Credit _credit = new Credit();

    public bool IsEligible(Customer cust, int amount)
    {
      Console.WriteLine("{0} applies for {1:C} loan\n",
        cust.Name, amount);

      bool eligible = true;

      // Check creditworthyness of applicant
      if (!_bank.HasSufficientSavings(cust, amount))
      {
        eligible = false;
      }
      else if (!_loan.HasNoBadLoans(cust))
      {
        eligible = false;
      }
      else if (!_credit.HasGoodCredit(cust))
      {
        eligible = false;
      }

      return eligible;
    }
  }
}
```

Output

```
Ann McKinsey applies for $125,000.00 loan

Check bank for Ann McKinsey
Check loans for Ann McKinsey
Check credit for Ann McKinsey

Ann McKinsey has been Approved
```

This .NET optimized code demonstrates the same real-world situation as above but uses modern, built-in .NET features, such as, generics, reflection, object initializers, automatic properties, etc.

Hide code

```
// Facade pattern -- .NET optimized
```

Proxy Design Pattern

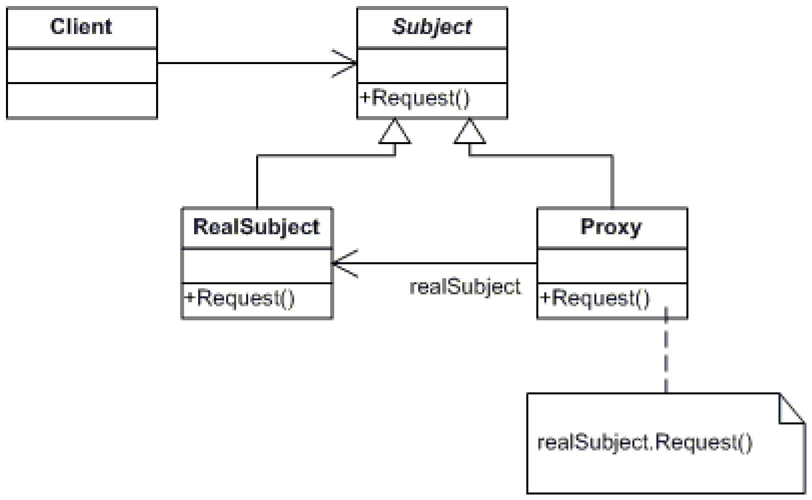
definition

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

Frequency of use: medium high

return to top

UML class diagram



return to top

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

return to top

sample code in C#

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

Hide code

```
// Proxy pattern -- Structural example



using System;


namespace DoFactory.GangOfFour.Proxy.Structural

{

  /// <summary>

  /// MainApp startup class for Structural

  /// Proxy Design Pattern.

  /// </summary>

  class MainApp

  {

    /// <summary>

    /// Entry point into console application.

    /// </summary>

    static void Main()

    {

      // Create proxy and request a service

      Proxy proxy = new Proxy();

      proxy.Request();


      // Wait for user

      Console.ReadKey();

    }

  }


  /// <summary>

  /// The 'Subject' abstract class
  /// </summary>
  abstract class Subject
  {
    public abstract void Request();
  }

  /// <summary>
  /// The 'RealSubject' class
  /// </summary>
  class RealSubject : Subject
  {
    public override void Request()
    {
      Console.WriteLine("Called RealSubject.Request()");
    }
  }

  /// <summary>
  /// The 'Proxy' class
  /// </summary>
  class Proxy : Subject
  {
    private RealSubject _realSubject;

    public override void Request()
    {
      // Use 'lazy initialization'
      if (_realSubject == null)
      {
        _realSubject = new RealSubject();
      }

      _realSubject.Request();
    }
  }
}
```

Output

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

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

Hide code

```
// Proxy pattern -- Real World example


using System;

namespace DoFactory.GangOfFour.Proxy.RealWorld
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Proxy Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
```

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

    // Do the math
    Console.WriteLine("4 + 2 = " + proxy.Add(4, 2));
    Console.WriteLine("4 - 2 = " + proxy.Sub(4, 2));
    Console.WriteLine("4 * 2 = " + proxy.Mul(4, 2));
    Console.WriteLine("4 / 2 = " + proxy.Div(4, 2));

    // Wait for user
    Console.ReadKey();
  }
}

/// <summary>
/// The 'Subject interface
/// </summary>
public interface IMath
{
  double Add(double x, double y);
  double Sub(double x, double y);
  double Mul(double x, double y);
  double Div(double x, double y);
}

/// <summary>
/// The 'RealSubject' class
/// </summary>
class Math : IMath
{
  public double Add(double x, double y) { return x + y; }
  public double Sub(double x, double y) { return x - y; }
  public double Mul(double x, double y) { return x * y; }
```

```
    public double Div(double x, double y) { return x / y; }
  }

  /// <summary>
  /// The 'Proxy Object' class
  /// </summary>
  class MathProxy : IMath
  {
    private Math _math = new Math();

    public double Add(double x, double y)
    {
      return _math.Add(x, y);
    }
    public double Sub(double x, double y)
    {
      return _math.Sub(x, y);
    }
    public double Mul(double x, double y)
    {
      return _math.Mul(x, y);
    }
    public double Div(double x, double y)
    {
      return _math.Div(x, y);
    }
  }
}
```

Output

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

This .NET optimized code demonstrates the same real-world situation as above but uses modern, built-in .NET features, such as, generics, reflection, object initializers, automatic properties, etc.

Hide code

```
// Proxy pattern -- .NET optimized
```

Composite Design Pattern

- definition
- UML diagram
- participants
- sample code in C#

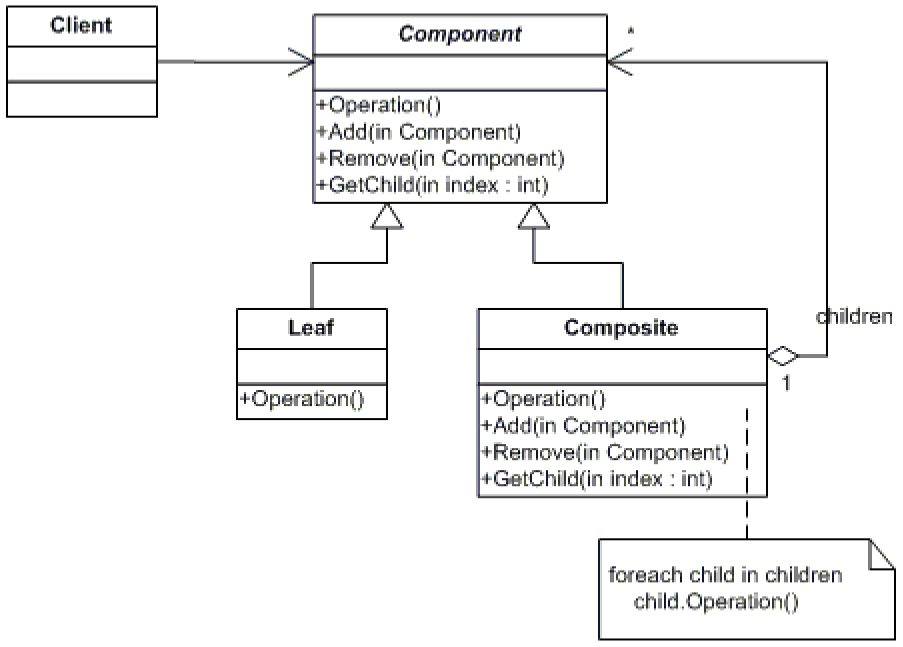
definition

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

return to top

UML class diagram



return to top

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.

return to top

sample code in C#

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

Hide code

```
// Composite pattern -- Structural example



using System;

using System.Collections.Generic;


namespace DoFactory.GangOfFour.Composite.Structural

{

  /// <summary>

  /// MainApp startup class for Structural

  /// Composite Design Pattern.

  /// </summary>

  class MainApp

  {

    /// <summary>

    /// Entry point into console application.

    /// </summary>

    static void Main()

    {

      // Create a tree structure

      Composite root = new Composite("root");

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

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


      Composite comp = new Composite("Composite X");

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

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


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

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

      // Recursively display tree
      root.Display(1);

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Component' abstract class
  /// </summary>
  abstract class Component
  {
    protected string name;

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

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

```
/// <summary>
/// The 'Composite' class
/// </summary>
class Composite : Component
{
  private List<Component> _children = new List<Component>();

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

  public override void Add(Component component)
  {
    _children.Add(component);
  }

  public override void Remove(Component component)
  {
    _children.Remove(component);
  }

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

    // Recursively display child nodes
    foreach (Component component in _children)
    {
      component.Display(depth + 2);
    }
  }
  }

  /// <summary>
  /// The 'Leaf' class
  /// </summary>
  class Leaf : Component
  {
    // Constructor
    public Leaf(string name)
      : base(name)
    {
    }

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

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

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

Output

```
-root
---Leaf A
---Leaf B
```

```
---Composite X
-----Leaf XA
-----Leaf XB
---Leaf C
```

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

Hide code

```
// Composite pattern -- Real World example



using System;

using System.Collections.Generic;


namespace DoFactory.GangOfFour.Composite.RealWorld

{

  /// <summary>

  /// MainApp startup class for Real-World

  /// Composite Design Pattern.

  /// </summary>

  class MainApp

  {

    /// <summary>

    /// Entry point into console application.

    /// </summary>

    static void Main()

    {

      // Create a tree structure

      CompositeElement root =
        new CompositeElement("Picture");
      root.Add(new PrimitiveElement("Red Line"));
      root.Add(new PrimitiveElement("Blue Circle"));
      root.Add(new PrimitiveElement("Green Box"));

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

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

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

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Component' Treenode
  /// </summary>
  abstract class DrawingElement
  {
    protected string _name;
    // Constructor
    public DrawingElement(string name)
    {
      this._name = name;
    }

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

  /// <summary>
  /// The 'Leaf' class
  /// </summary>
  class PrimitiveElement : DrawingElement
  {
    // Constructor
    public PrimitiveElement(string name)
      : base(name)
    {
    }

    public override void Add(DrawingElement c)
    {
      Console.WriteLine(
        "Cannot add to a PrimitiveElement");
    }

    public override void Remove(DrawingElement c)
    {
      Console.WriteLine(
        "Cannot remove from a PrimitiveElement");
    }
    public override void Display(int indent)
    {
      Console.WriteLine(
        new String('-', indent) + " " + _name);
    }
  }

  /// <summary>
  /// The 'Composite' class
  /// </summary>
  class CompositeElement : DrawingElement
  {
    private List<DrawingElement> elements =
      new List<DrawingElement>();

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

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

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

    public override void Display(int indent)
```

```
    {
      Console.WriteLine(new String('-', indent) +
        "+ " + _name);

      // Display each child element on this node
      foreach (DrawingElement d in elements)
      {
        d.Display(indent + 2);
      }
    }
  }
}
```

Output

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

This .NET optimized code demonstrates the same real-world situation as above but uses modern, built-in .NET features, such as, generics, reflection, object initializers, automatic properties, etc.

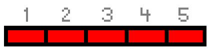
Hide code

```
// Composite pattern -- .NET optimized
```

Observer Design Pattern

definition

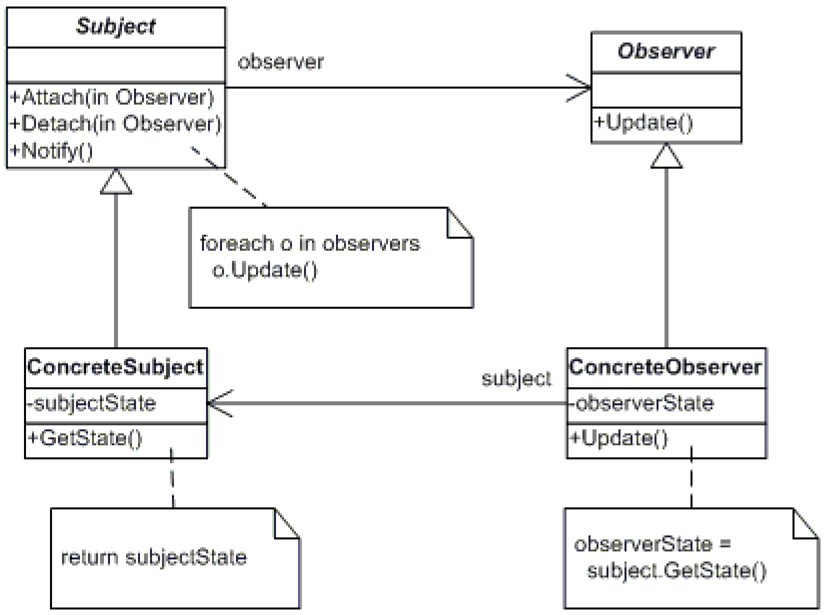
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Frequency of use: 

high

return to top

UML class diagram



return to top

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

return to top

sample code in C#

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

Hide code

```
// Observer pattern -- Structural example



using System;

using System.Collections.Generic;


namespace DoFactory.GangOfFour.Observer.Structural

{

  /// <summary>

  /// MainApp startup class for Structural

  /// Observer Design Pattern.

  /// </summary>

  class MainApp

  {

    /// <summary>

    /// Entry point into console application.

    /// </summary>
    static void Main()
    {
      // Configure Observer pattern
      ConcreteSubject s = new ConcreteSubject();

      s.Attach(new ConcreteObserver(s, "X"));
      s.Attach(new ConcreteObserver(s, "Y"));
      s.Attach(new ConcreteObserver(s, "Z"));

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

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Subject' abstract class
  /// </summary>
  abstract class Subject
  {
    private List<Observer> _observers = new List<Observer>();

    public void Attach(Observer observer)
    {
      _observers.Add(observer);
    }

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

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

  /// <summary>
  /// The 'ConcreteSubject' class
  /// </summary>
  class ConcreteSubject : Subject
  {
    private string _subjectState;

    // Gets or sets subject state
    public string SubjectState
    {
      get { return _subjectState; }
      set { _subjectState = value; }
    }
  }

  /// <summary>
  /// The 'Observer' abstract class
  /// </summary>
  abstract class Observer
  {
    public abstract void Update();
  }
```

```
  /// <summary>
  /// The 'ConcreteObserver' class
  /// </summary>
  class ConcreteObserver : Observer
  {
    private string _name;
    private string _observerState;
    private ConcreteSubject _subject;

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

    public override void Update()
    {
      _observerState = _subject.SubjectState;
      Console.WriteLine("Observer {0}'s new state is {1}",
        _name, _observerState);
    }

    // Gets or sets subject
    public ConcreteSubject Subject
    {
      get { return _subject; }
      set { _subject = value; }
    }
  }
}
```

Output

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

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

Hide code

```
// Observer pattern -- Real World example



using System;

using System.Collections.Generic;


namespace DoFactory.GangOfFour.Observer.RealWorld

{

  /// <summary>

  /// MainApp startup class for Real-World

  /// Observer Design Pattern.

  /// </summary>

  class MainApp

  {

    /// <summary>

    /// Entry point into console application.

    /// </summary>

    static void Main()

    {

      // Create IBM stock and attach investors
      IBM ibm = new IBM("IBM", 120.00);
      ibm.Attach(new Investor("Sorros"));
      ibm.Attach(new Investor("Berkshire"));

      // Fluctuating prices will notify investors
      ibm.Price = 120.10;
      ibm.Price = 121.00;
      ibm.Price = 120.50;
      ibm.Price = 120.75;

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Subject' abstract class
  /// </summary>
  abstract class Stock
  {
    private string _symbol;
    private double _price;
    private List<IInvestor> _investors = new List<IInvestor>();

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

    public void Attach(IInvestor investor)
    {
```

```
    _investors.Add(investor);
  }

  public void Detach(IInvestor investor)
  {
    _investors.Remove(investor);
  }

  public void Notify()
  {
    foreach (IInvestor investor in _investors)
    {
      investor.Update(this);
    }

    Console.WriteLine("");
  }

  // Gets or sets the price
  public double Price
  {
    get { return _price; }
    set
    {
      if (_price != value)
      {
        _price = value;
        Notify();
      }
    }
  }

  // Gets the symbol
  public string Symbol
  {
    get { return _symbol; }
  }
}

/// <summary>
/// The 'ConcreteSubject' class
/// </summary>
class IBM : Stock
{
  // Constructor
  public IBM(string symbol, double price)
    : base(symbol, price)
  {
  }
}

/// <summary>
/// The 'Observer' interface
/// </summary>
interface IInvestor
{
  void Update(Stock stock);
}

/// <summary>
/// The 'ConcreteObserver' class
/// </summary>
class Investor : IInvestor
{
  private string _name;
  private Stock _stock;
```

```
    // Constructor
    public Investor(string name)
    {
      this._name = name;
    }

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

    // Gets or sets the stock
    public Stock Stock
    {
      get { return _stock; }
      set { _stock = value; }
    }
  }
}
```

Output

```
Notified Sorros of IBM's change to $120.10
Notified Berkshire of IBM's change to $120.10

Notified Sorros of IBM's change to $121.00
Notified Berkshire of IBM's change to $121.00

Notified Sorros of IBM's change to $120.50
Notified Berkshire of IBM's change to $120.50

Notified Sorros of IBM's change to $120.75
Notified Berkshire of IBM's change to $120.75
```

This .NET optimized code demonstrates the same real-world situation as above but uses modern, built-in .NET features, such as, generics, reflection, object initializers, automatic properties, etc.

Hide code

```
// Observerver pattern -- .NET optimized
```

Strategy Design Pattern

- definition
- UML diagram
- participants
- sample code in C#

definition

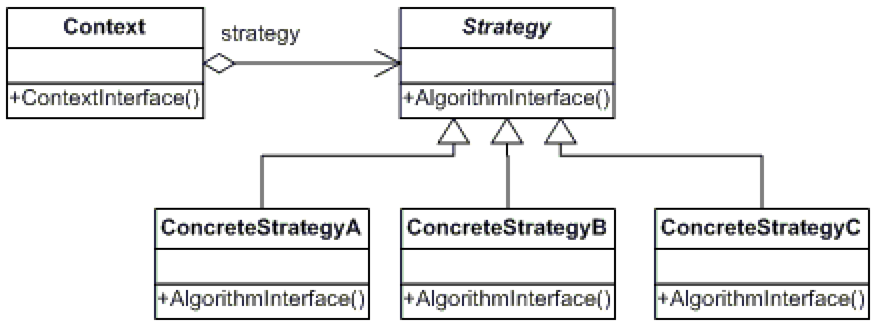
Define a family of algorithms, encapsulate each one, and make them interchangeable.

Strategy lets the algorithm vary independently from clients that use it.

Frequency of use: 🟥🟥🟥🟥🟨 medium high

return to top

UML class diagram



return to top

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.

return to top

sample code in C#

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

Hide code

```
// Strategy pattern -- Structural example



using System;


namespace DoFactory.GangOfFour.Strategy.Structural

{

  /// <summary>

  /// MainApp startup class for Structural

  /// Strategy Design Pattern.

  /// </summary>

  class MainApp

  {

    /// <summary>

    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      Context context;

      // Three contexts following different strategies
      context = new Context(new ConcreteStrategyA());
      context.ContextInterface();

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

      context = new Context(new ConcreteStrategyC());
      context.ContextInterface();

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Strategy' abstract class
  /// </summary>
  abstract class Strategy
  {
    public abstract void AlgorithmInterface();
  }

  /// <summary>
  /// A 'ConcreteStrategy' class
  /// </summary>
  class ConcreteStrategyA : Strategy
  {
    public override void AlgorithmInterface()
    {
      Console.WriteLine(
        "Called ConcreteStrategyA.AlgorithmInterface()");
    }
  }

  /// <summary>
  /// A 'ConcreteStrategy' class
  /// </summary>
  class ConcreteStrategyB : Strategy
  {
    public override void AlgorithmInterface()
    {
      Console.WriteLine(
        "Called ConcreteStrategyB.AlgorithmInterface()");
    }
  }

  /// <summary>
  /// A 'ConcreteStrategy' class
  /// </summary>
  class ConcreteStrategyC : Strategy
  {
    public override void AlgorithmInterface()
    {
      Console.WriteLine(
        "Called ConcreteStrategyC.AlgorithmInterface()");
    }
  }

  /// <summary>
  /// The 'Context' class
```

```
  /// </summary>
  class Context
  {
    private Strategy _strategy;

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

    public void ContextInterface()
    {
      _strategy.AlgorithmInterface();
    }
  }
}
```

Output

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

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

Hide code

```
// Strategy pattern -- Real World example
```

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

namespace DoFactory.GangOfFour.Strategy.RealWorld
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Strategy Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // Two contexts following different strategies
      SortedList studentRecords = new SortedList();

      studentRecords.Add("Samual");
      studentRecords.Add("Jimmy");
      studentRecords.Add("Sandra");
      studentRecords.Add("Vivek");
      studentRecords.Add("Anna");

      studentRecords.SetSortStrategy(new QuickSort());
      studentRecords.Sort();

      studentRecords.SetSortStrategy(new ShellSort());
      studentRecords.Sort();

      studentRecords.SetSortStrategy(new MergeSort());
      studentRecords.Sort();

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Strategy' abstract class
  /// </summary>
  abstract class SortStrategy
  {
    public abstract void Sort(List<string> list);
  }

  /// <summary>
  /// A 'ConcreteStrategy' class
  /// </summary>
  class QuickSort : SortStrategy
  {
    public override void Sort(List<string> list)
    {
      list.Sort(); // Default is Quicksort
      Console.WriteLine("QuickSorted list ");
    }
  }

  /// <summary>
  /// A 'ConcreteStrategy' class
  /// </summary>
  class ShellSort : SortStrategy
  {
    public override void Sort(List<string> list)
    {
      //list.ShellSort(); not-implemented
      Console.WriteLine("ShellSorted list ");
    }
  }

  /// <summary>
  /// A 'ConcreteStrategy' class
  /// </summary>
  class MergeSort : SortStrategy
  {
    public override void Sort(List<string> list)
    {
      //list.MergeSort(); not-implemented
      Console.WriteLine("MergeSorted list ");
    }
  }

  /// <summary>
  /// The 'Context' class
  /// </summary>
  class SortedList
  {
    private List<string> _list = new List<string>();
    private SortStrategy _sortstrategy;

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

    public void Add(string name)
    {
      _list.Add(name);

    }


    public void Sort()

    {

      _sortstrategy.Sort(_list);


      // Iterate over list and display results

      foreach (string name in _list)

      {

        Console.WriteLine(" " + name);

      }

      Console.WriteLine();

    }

  }

}
```

Output

```
QuickSorted list
 Anna
 Jimmy
 Samual
 Sandra
 Vivek

ShellSorted list
 Anna
 Jimmy
 Samual
 Sandra
 Vivek

MergeSorted list
 Anna
 Jimmy
 Samual
 Sandra
 Vivek
```

This .NET optimized code demonstrates the same real-world situation as above but uses modern, built-in .NET features, such as, generics, reflection, object initializers, automatic properties, etc.

Hide code

```
// Strategy pattern -- .NET optimized
```

Command Design Pattern

- definition
- UML diagram
- participants
- sample code in C#

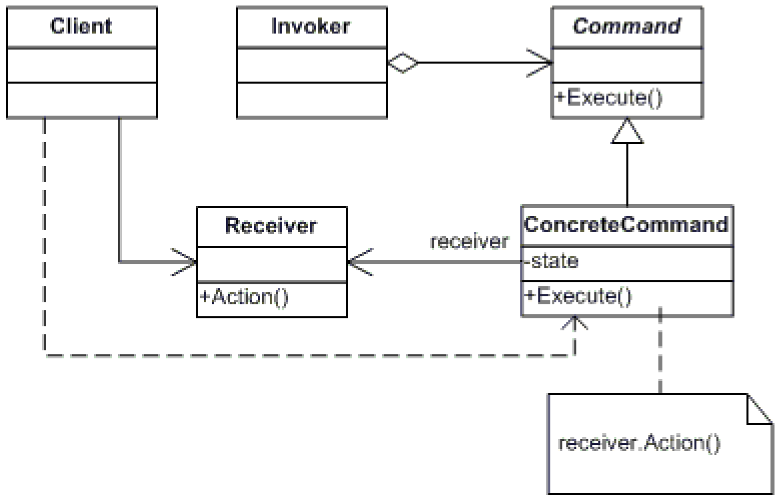
definition

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

return to top

UML class diagram



return to top

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.

return to top

sample code in C#

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

Hide code

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

```
using System;

namespace DoFactory.GangOfFour.Command.Structural
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Command Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // Create receiver, command, and invoker
      Receiver receiver = new Receiver();
      Command command = new ConcreteCommand(receiver);
      Invoker invoker = new Invoker();

      // Set and execute command
      invoker.SetCommand(command);
      invoker.ExecuteCommand();

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Command' abstract class
  /// </summary>
  abstract class Command
```

```
{
  protected Receiver receiver;

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

  public abstract void Execute();
}

/// <summary>
/// The 'ConcreteCommand' class
/// </summary>
class ConcreteCommand : Command
{
  // Constructor
  public ConcreteCommand(Receiver receiver) :
    base(receiver)
  {
  }

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

/// <summary>
/// The 'Receiver' class
/// </summary>
class Receiver
```

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

  /// <summary>
  /// The 'Invoker' class
  /// </summary>
  class Invoker
  {
    private Command _command;

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

    public void ExecuteCommand()
    {
      _command.Execute();
    }
  }
}
```

Output

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

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

Hide code

```
// Command pattern -- Real World example



using System;

using System.Collections.Generic;


namespace DoFactory.GangOfFour.Command.RealWorld

{

  /// <summary>

  /// MainApp startup class for Real-World

  /// Command Design Pattern.

  /// </summary>

  class MainApp

  {

    /// <summary>

    /// Entry point into console application.

    /// </summary>

    static void Main()

    {

      // Create user and let her compute

      User user = new User();


      // User presses calculator buttons

      user.Compute('+', 100);

      user.Compute('-', 50);

      user.Compute('*', 10);

      user.Compute('/', 2);


      // Undo 4 commands
      user.Undo(4);

      // Redo 3 commands
      user.Redo(3);

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Command' abstract class
  /// </summary>
  abstract class Command
  {
    public abstract void Execute();
    public abstract void UnExecute();
  }

  /// <summary>
  /// The 'ConcreteCommand' class
  /// </summary>
  class CalculatorCommand : Command
  {
    private char _operator;
    private int _operand;
    private Calculator _calculator;

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

    // Gets operator
    public char Operator
    {
      set { _operator = value; }
    }

    // Get operand
    public int Operand
    {
      set { _operand = value; }
    }

    // Execute new command
    public override void Execute()
    {
      _calculator.Operation(_operator, _operand);
    }

    // Unexecute last command
    public override void UnExecute()
    {
      _calculator.Operation(Undo(_operator), _operand);
    }

    // Returns opposite operator for given operator
    private char Undo(char @operator)
    {
      switch (@operator)
      {
        case '+': return '-';
        case '-': return '+';
        case '*': return '/';
        case '/': return '*';
        default: throw new
         ArgumentException("@operator");
      }
    }
  }

  /// <summary>
  /// The 'Receiver' class
  /// </summary>
  class Calculator
  {
    private int _curr = 0;

    public void Operation(char @operator, int operand)
    {
      switch (@operator)
      {
        case '+': _curr += operand; break;
        case '-': _curr -= operand; break;
        case '*': _curr *= operand; break;
        case '/': _curr /= operand; break;
      }
      Console.WriteLine(
        "Current value = {0,3} (following {1} {2})",
        _curr, @operator, operand);
    }
  }

  /// <summary>
```

```
/// The 'Invoker' class
/// </summary>
class User
{
  // Initializers
  private Calculator _calculator = new Calculator();
  private List<Command> _commands = new List<Command>();
  private int _current = 0;

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

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

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

      // Add command to undo list
      _commands.Add(command);
      _current++;
    }
  }
}
```

Output

```
Current value = 100 (following + 100)
Current value =  50 (following - 50)
Current value = 500 (following * 10)
Current value = 250 (following / 2)

---- Undo 4 levels
Current value = 500 (following * 2)
Current value =  50 (following / 10)
Current value = 100 (following + 50)
Current value =   0 (following - 100)

---- Redo 3 levels
Current value = 100 (following + 100)
Current value =  50 (following - 50)
Current value = 500 (following * 10)
```

This .NET optimized code demonstrates the same real-world situation as above but uses modern, built-in .NET features, such as, generics, reflection, object initializers, automatic properties, etc.

Hide code

```
// Command pattern -- .NET optimized
```

Iterator Design Pattern

- definition
- UML diagram
- participants
- sample code in C#

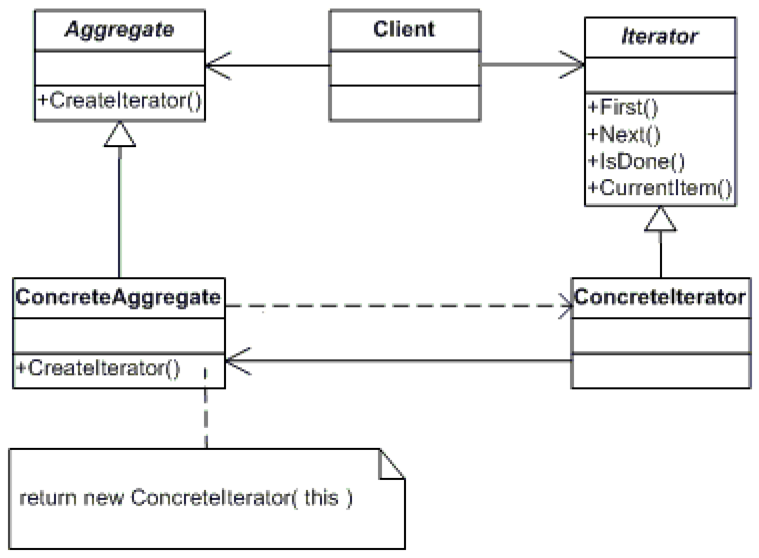
definition

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

Frequency of use: 🟥🟥🟥🟥🟥 high

return to top

UML class diagram



return to top

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

return to top

sample code in C#

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

Hide code

```
// Iterator pattern -- Structural example
```

```
using System;
using System.Collections;

namespace DoFactory.GangOfFour.Iterator.Structural
{
    /// <summary>
    /// MainApp startup class for Structural
    /// Iterator Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            ConcreteAggregate a = new ConcreteAggregate();
            a[0] = "Item A";
            a[1] = "Item B";
            a[2] = "Item C";
            a[3] = "Item D";

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

            Console.WriteLine("Iterating over collection:");

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

            // Wait for user
            Console.ReadKey();
        }
    }

    /// <summary>
    /// The 'Aggregate' abstract class
    /// </summary>
    abstract class Aggregate
    {
        public abstract Iterator CreateIterator();
    }

    /// <summary>
    /// The 'ConcreteAggregate' class
    /// </summary>
    class ConcreteAggregate : Aggregate
    {
        private ArrayList _items = new ArrayList();

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

        // Gets item count
        public int Count
        {
            get { return _items.Count; }
        }
        // Indexer
        public object this[int index]
        {
            get { return _items[index]; }
            set { _items.Insert(index, value); }
        }
    }

    /// <summary>
    /// The 'Iterator' abstract class
    /// </summary>
    abstract class Iterator
    {
        public abstract object First();
        public abstract object Next();
        public abstract bool IsDone();
        public abstract object CurrentItem();
    }

    /// <summary>
    /// The 'ConcreteIterator' class
    /// </summary>
    class ConcreteIterator : Iterator
    {
        private ConcreteAggregate _aggregate;
        private int _current = 0;

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

```
        // Gets first iteration item
        public override object First()
        {
            return _aggregate[0];
        }

        // Gets next iteration item
        public override object Next()
        {
            object ret = null;
            if (_current < _aggregate.Count - 1)
            {
                ret = _aggregate[++_current];
            }

            return ret;
        }

        // Gets current iteration item
        public override object CurrentItem()
        {
            return _aggregate[_current];
        }

        // Gets whether iterations are complete
        public override bool IsDone()
        {
            return _current >= _aggregate.Count;
        }
    }
}
```

Output

```
Iterating over collection:
Item A
Item B
Item C
Item D
```

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

Hide code

```
// Iterator pattern -- Real World example



using System;

using System.Collections;


namespace DoFactory.GangOfFour.Iterator.RealWorld

{

    /// <summary>

    /// MainApp startup class for Real-World

    /// Iterator Design Pattern.

    /// </summary>

    class MainApp

    {

        /// <summary>

        /// Entry point into console application.

        /// </summary>

        static void Main()

        {

            // Build a collection

            Collection collection = new Collection();
            collection[0] = new Item("Item 0");
            collection[1] = new Item("Item 1");
            collection[2] = new Item("Item 2");
            collection[3] = new Item("Item 3");
            collection[4] = new Item("Item 4");
            collection[5] = new Item("Item 5");
            collection[6] = new Item("Item 6");
            collection[7] = new Item("Item 7");
            collection[8] = new Item("Item 8");

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

            // Skip every other item
            iterator.Step = 2;

            Console.WriteLine("Iterating over collection:");

            for (Item item = iterator.First();
                !iterator.IsDone; item = iterator.Next())
            {
                Console.WriteLine(item.Name);
            }

            // Wait for user
            Console.ReadKey();
        }
    }

    /// <summary>
    /// A collection item
    /// </summary>
    class Item
```

```
{
    private string _name;

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

    // Gets name
    public string Name
    {
        get { return _name; }
    }
}

/// <summary>
/// The 'Aggregate' interface
/// </summary>
interface IAbstractCollection
{
    Iterator CreateIterator();
}

/// <summary>
/// The 'ConcreteAggregate' class
/// </summary>
class Collection : IAbstractCollection
{
    private ArrayList _items = new ArrayList();

    public Iterator CreateIterator()
    {
        return new Iterator(this);

    }



    // Gets item count

    public int Count

    {

        get { return _items.Count; }

    }



    // Indexer

    public object this[int index]

    {

        get { return _items[index]; }

        set { _items.Add(value); }

    }

}



/// <summary>

/// The 'Iterator' interface

/// </summary>

interface IAbstractIterator

{

    Item First();

    Item Next();

    bool IsDone { get; }

    Item CurrentItem { get; }

}



/// <summary>

/// The 'ConcreteIterator' class

/// </summary>

class Iterator : IAbstractIterator

{
    private Collection _collection;
    private int _current = 0;
    private int _step = 1;

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

    // Gets first item
    public Item First()
    {
        _current = 0;
        return _collection[_current] as Item;
    }

    // Gets next item
    public Item Next()
    {
        _current += _step;
        if (!IsDone)
            return _collection[_current] as Item;
        else
            return null;
    }

    // Gets or sets stepsize
    public int Step
    {
        get { return _step; }
        set { _step = value; }
    }
    // Gets current iterator item
    public Item CurrentItem
    {
        get { return _collection[_current] as Item; }
    }

    // Gets whether iteration is complete
    public bool IsDone
    {
        get { return _current >= _collection.Count; }
    }
}
}
```

Output

```
Iterating over collection:
Item 0
Item 2
Item 4
Item 6
Item 8
```

This .NET optimized code demonstrates the same real-world situation as above but uses modern, built-in .NET features, such as, generics, reflection, object initializers, automatic properties, etc.

Hide code

```
// Iterator pattern -- .NET optimized
```