

Design Patterns

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. They are categorized in three groups: Creational, Structural, and Behavioral. Here you will find information on these important patterns.

To give you a head start, the C# source code 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 2.0, 3.0, and 3.5 features, such as, generics, attributes, delegates, object and collection initializers, automatic properties, and reflection.

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
--------------------------------	---

<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 an 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

Creational Patterns

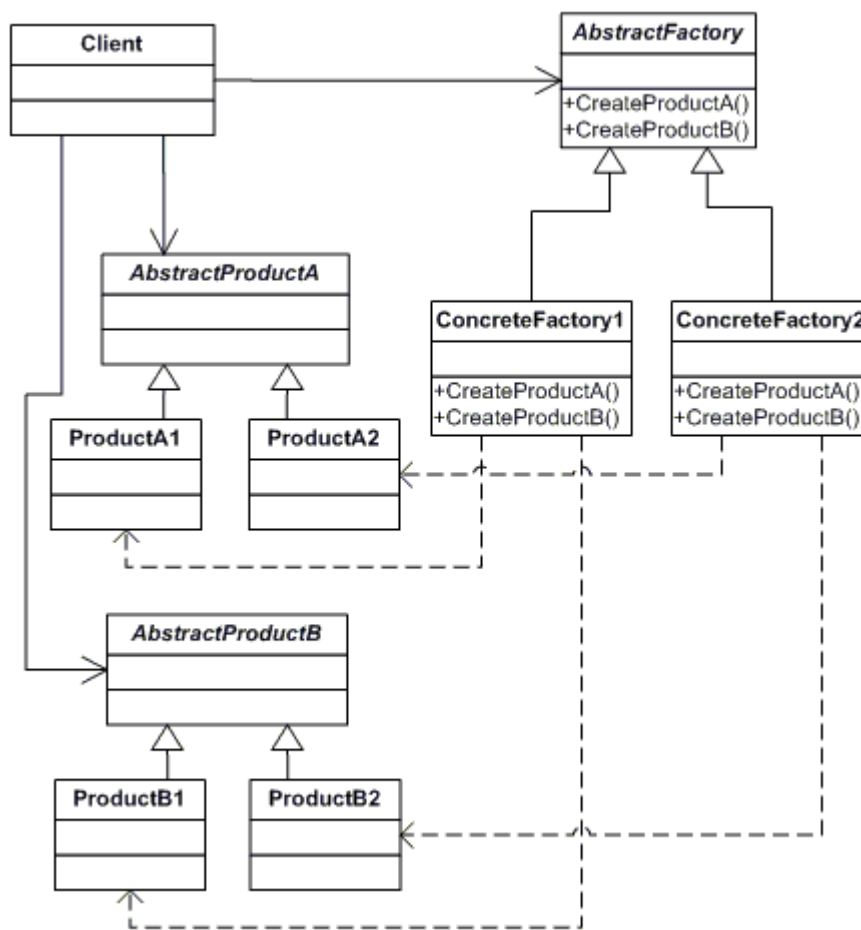
1. Abstract Factory Design Pattern

definition

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

Frequency of use:  high

UML class diagram



participants

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

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

sample code in C#

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

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

```

using System;

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.

```

// 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

```

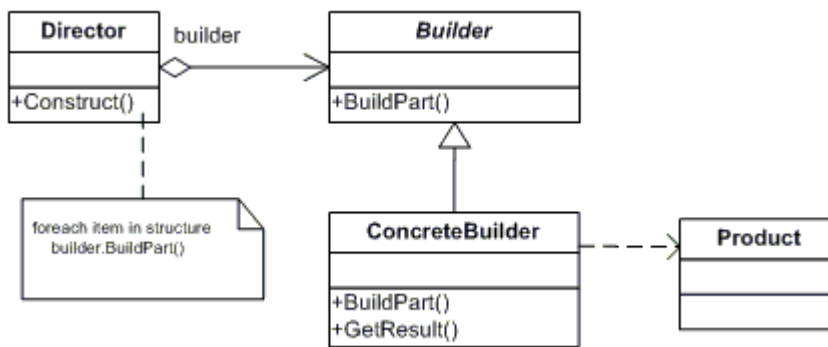
2. Builder Design Pattern

definition

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

Frequency of use:  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

sample code in C#

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

```

// Builder pattern -- Structural example

using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Builder.Structural
{
    /// <summary>
    /// MainApp startup class for Structural
    /// Builder Design Pattern.
    /// </summary>
    public class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        public static void Main()
        {
            // Create director and builders
            Director director = new Director();
        }
    }
}
  
```

```

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

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

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

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

/// <summary>
/// The 'Director' class
/// </summary>
class Director
{
    // Builder uses a complex series of steps
    public void Construct(Builder builder)
    {
        builder.BuildPartA();
        builder.BuildPartB();
    }
}

/// <summary>
/// The 'Builder' abstract class
/// </summary>
abstract class Builder
{
    public abstract void BuildPartA();
    public abstract void BuildPartB();
    public abstract Product GetResult();
}

/// <summary>
/// The 'ConcreteBuilder1' class
/// </summary>
class ConcreteBuilder1 : Builder
{
    private Product _product = new Product();

    public override void BuildPartA()
    {
        _product.Add("PartA");
    }

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

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

/// <summary>
/// The 'ConcreteBuilder2' class
/// </summary>
class ConcreteBuilder2 : Builder

```

```

{
    private Product _product = new Product();

    public override void BuildPartA()
    {
        _product.Add("PartX");
    }

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

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

/// <summary>
/// The 'Product' class
/// </summary>
class Product
{
    private List<string> _parts = new List<string>();

    public void Add(string part)
    {
        _parts.Add(part);
    }

    public void Show()
    {
        Console.WriteLine("\nProduct Parts -----");
        foreach (string part in _parts)
            Console.WriteLine(part);
    }
}
}

```

Output

```

Product Parts -----
PartA
PartB

Product Parts -----
PartX
PartY

```

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

```

// Builder pattern -- Real World example

using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Builder.RealWorld
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Builder Design Pattern.
    /// </summary>

```

```

public class MainApp
{
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
        VehicleBuilder builder;

        // Create shop with vehicle builders
        Shop shop = new Shop();

        // Construct and display vehicles
        builder = new ScooterBuilder();
        shop.Construct(builder);
        builder.Vehicle.Show();

        builder = new CarBuilder();
        shop.Construct(builder);
        builder.Vehicle.Show();

        builder = new MotorCycleBuilder();
        shop.Construct(builder);
        builder.Vehicle.Show();

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

/// <summary>
/// The 'Director' class
/// </summary>
class Shop
{
    // Builder uses a complex series of steps
    public void Construct(VehicleBuilder vehicleBuilder)
    {
        vehicleBuilder.BuildFrame();
        vehicleBuilder.BuildEngine();
        vehicleBuilder.BuildWheels();
        vehicleBuilder.BuildDoors();
    }
}

/// <summary>
/// The 'Builder' abstract class
/// </summary>
abstract class VehicleBuilder
{
    protected Vehicle vehicle;

    // Gets vehicle instance
    public Vehicle Vehicle
    {
        get { return vehicle; }
    }

    // Abstract build methods
    public abstract void BuildFrame();
    public abstract void BuildEngine();
    public abstract void BuildWheels();
    public abstract void BuildDoors();
}

/// <summary>
/// The 'ConcreteBuilder1' class
/// </summary>

```

```

class MotorcycleBuilder : VehicleBuilder
{
    public MotorcycleBuilder()
    {
        vehicle = new Vehicle("MotorCycle");
    }

    public override void BuildFrame()
    {
        vehicle["frame"] = "MotorCycle Frame";
    }

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

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

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

/// <summary>
/// The 'ConcreteBuilder2' class
/// </summary>
class CarBuilder : VehicleBuilder
{
    public CarBuilder()
    {
        vehicle = new Vehicle("Car");
    }

    public override void BuildFrame()
    {
        vehicle["frame"] = "Car Frame";
    }

    public override void BuildEngine()
    {
        vehicle["engine"] = "2500 cc";
    }

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

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

/// <summary>
/// The 'ConcreteBuilder3' class
/// </summary>
class ScooterBuilder : VehicleBuilder
{
    public ScooterBuilder()
    {
        vehicle = new Vehicle("Scooter");
    }
}

```

```

    }

    public override void BuildFrame()
    {
        vehicle["frame"] = "Scooter Frame";
    }

    public override void BuildEngine()
    {
        vehicle["engine"] = "50 cc";
    }

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

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

/// <summary>
/// The 'Product' class
/// </summary>
class Vehicle
{
    private string _vehicleType;
    private Dictionary<string, string> _parts =
        new Dictionary<string, string>();

    // Constructor
    public Vehicle(string vehicleType)
    {
        this._vehicleType = vehicleType;
    }

    // Indexer
    public string this[string key]
    {
        get { return _parts[key]; }
        set { _parts[key] = value; }
    }

    public void Show()
    {
        Console.WriteLine("\n-----");
        Console.WriteLine("Vehicle Type: {0}", _vehicleType);
        Console.WriteLine(" Frame : {0}", _parts["frame"]);
        Console.WriteLine(" Engine : {0}", _parts["engine"]);
        Console.WriteLine(" #Wheels: {0}", _parts["wheels"]);
        Console.WriteLine(" #Doors : {0}", _parts["doors"]);
    }
}
}

```

Output

```

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

```

```

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

-----

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

```

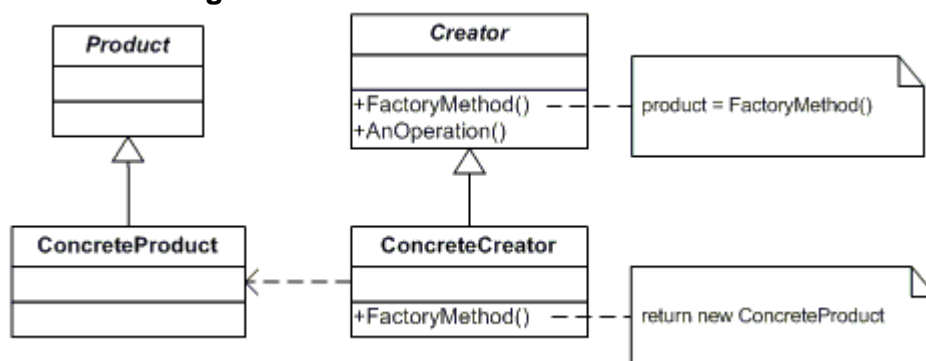
3. Factory Method Design Pattern

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.

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.

```
// 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.

```

// 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
```

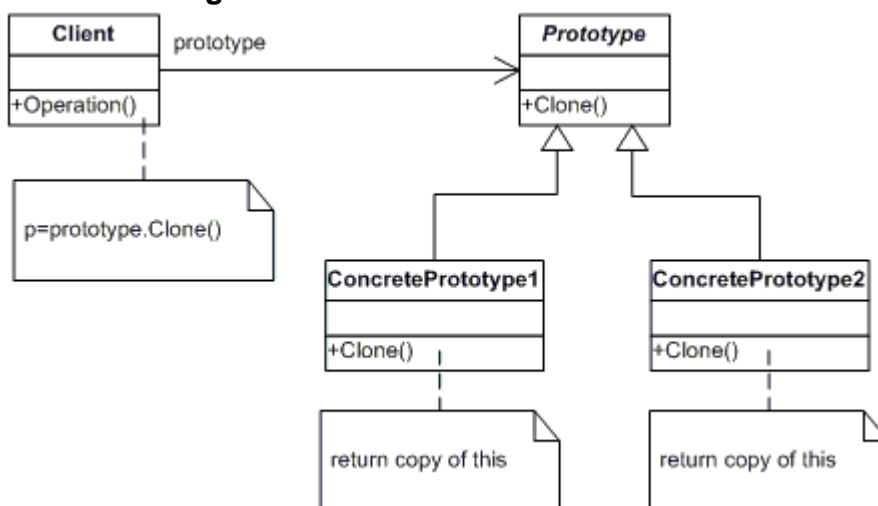
4. Prototype Design Pattern

definition

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

Frequency of use: nbsp;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

sample code in C#

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

```
// Prototype pattern -- Structural example
```

```

using System;

namespace DoFactory.GangOfFour.Prototype.Structural
{
    /// <summary>
    /// MainApp startup class for Structural
    /// Prototype Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            // Create two instances and clone each

            ConcretePrototype1 p1 = new ConcretePrototype1("I");
            ConcretePrototype1 c1 = (ConcretePrototype1)p1.Clone();
            Console.WriteLine("Cloned: {0}", c1.Id);

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

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

    /// <summary>
    /// The 'Prototype' abstract class
    /// </summary>
    abstract class Prototype
    {
        private string _id;

        // Constructor
        public Prototype(string id)
        {
            this._id = id;
        }

        // Gets id
        public string Id
        {
            get { return _id; }
        }

        public abstract Prototype Clone();
    }

    /// <summary>
    /// A 'ConcretePrototype' class
    /// </summary>
    class ConcretePrototype1 : Prototype
    {
        // Constructor
        public ConcretePrototype1(string id)
            : base(id)
        {
        }

        // Returns a shallow copy
        public override Prototype Clone()
        {
            return (Prototype)this.MemberwiseClone();
        }
    }
}

```

```

    }
}

/// <summary>
/// A 'ConcretePrototype' class
/// </summary>
class ConcretePrototype2 : Prototype
{
    // Constructor
    public ConcretePrototype2(string id)
        : base(id)
    {
    }

    // Returns a shallow copy
    public override Prototype Clone()
    {
        return (Prototype) this.MemberwiseClone();
    }
}
}

```

Output

```

Cloned: I
Cloned: II

```

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

```

// Prototype pattern -- Real World example

using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Prototype.RealWorld
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Prototype Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            ColorManager colormanager = new ColorManager();

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

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

            // User clones selected colors
            Color color1 = colormanager["red"].Clone() as Color;
            Color color2 = colormanager["peace"].Clone() as Color;
            Color color3 = colormanager["flame"].Clone() as Color;

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

```

```

    }
}

/// <summary>
/// The 'Prototype' abstract class
/// </summary>
abstract class ColorPrototype
{
    public abstract ColorPrototype Clone();
}

/// <summary>
/// The 'ConcretePrototype' class
/// </summary>
class Color : ColorPrototype
{
    private int _red;
    private int _green;
    private int _blue;

    // Constructor
    public Color(int red, int green, int blue)
    {
        this._red = red;
        this._green = green;
        this._blue = blue;
    }

    // Create a shallow copy
    public override ColorPrototype Clone()
    {
        Console.WriteLine(
            "Cloning color RGB: {0,3},{1,3},{2,3}",
            _red, _green, _blue);

        return this.MemberwiseClone() as ColorPrototype;
    }
}

/// <summary>
/// Prototype manager
/// </summary>
class ColorManager
{
    private Dictionary<string, ColorPrototype> _colors =
        new Dictionary<string, ColorPrototype>();

    // Indexer
    public ColorPrototype this[string key]
    {
        get { return _colors[key]; }
        set { _colors.Add(key, value); }
    }
}
}

```

Output

```

Cloning color RGB: 255, 0, 0
Cloning color RGB: 128,211,128
Cloning color RGB: 211, 34, 20

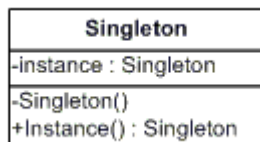
```

5. Singleton Design Pattern

definition

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

Frequency of use:  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.

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.

```
// 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 through the one object that has knowledge about the state of the (web) farm.

```

// 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

```

Structural Patterns

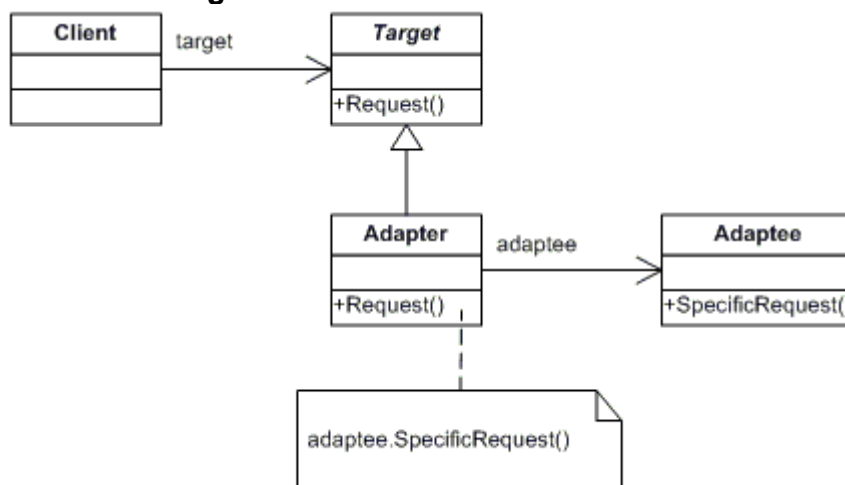
6. Adapter Design Pattern

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.

sample code in C#

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

```
// Adapter pattern -- Structural example

using System;

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.

```

// 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 "H2O";
        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: H2O
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

```

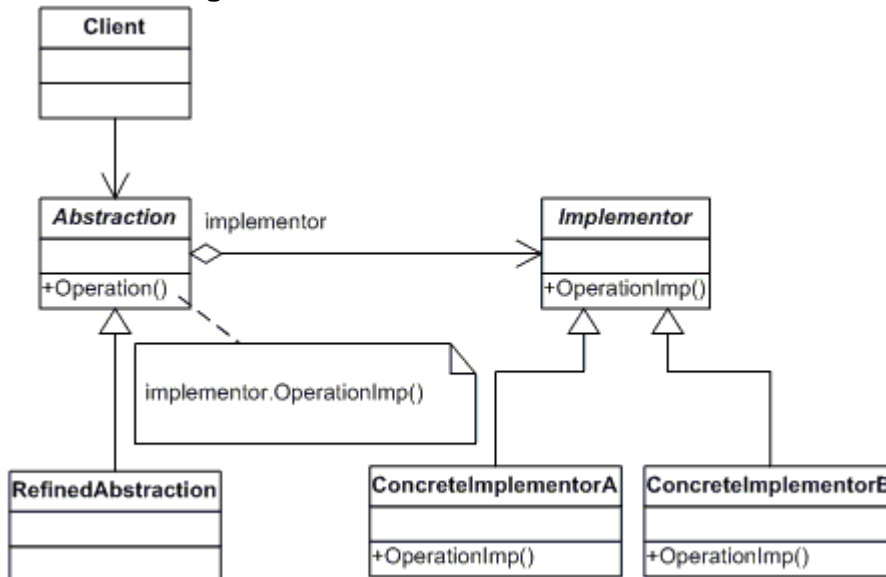
7. Bridge Design Pattern

definition

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

Frequency of use: nbsp;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.

sample code in C#

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

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

namespace DoFactory.GangOfFour.Bridge.Structural
{
    /// <summary>
```



```

/// MainApp startup class for Structural
/// Bridge Design Pattern.
/// </summary>
class MainApp
{
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
        Abstraction ab = new RefinedAbstraction();

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

        // Change implementation and call
        ab.Implementor = new ConcreteImplementorB();
        ab.Operation();

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

/// <summary>
/// The 'Abstraction' class
/// </summary>
class Abstraction
{
    protected Implementor implementor;

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

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

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

/// <summary>
/// The 'RefinedAbstraction' class
/// </summary>
class RefinedAbstraction : Abstraction
{
    public override void Operation()
    {
        implementor.Operation();
    }
}

/// <summary>
/// The 'ConcreteImplementorA' class
/// </summary>
class ConcreteImplementorA : Implementor
{

```

```

public override void Operation()
{
    Console.WriteLine("ConcreteImplementorA Operation");
}

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

```

Output

```

ConcreteImplementorA Operation
ConcreteImplementorB Operation

```

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

```

// Bridge pattern -- Real World example

using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Bridge.RealWorld
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Bridge Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            // Create RefinedAbstraction
            Customers customers = new Customers("Chicago");

            // Set ConcreteImplementor
            customers.Data = new CustomersData();

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

            customers.ShowAll();

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

    /// <summary>

```

```

/// The 'Abstraction' class
/// </summary>
class CustomersBase
{
    private DataObject _dataObject;
    protected string group;

    public CustomersBase(string group)
    {
        this.group = group;
    }

    // Property
    public DataObject Data
    {
        set { _dataObject = value; }
        get { return _dataObject; }
    }

    public virtual void Next()
    {
        _dataObject.NextRecord();
    }

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

    public virtual void Add(string customer)
    {
        _dataObject.AddRecord(customer);
    }

    public virtual void Delete(string customer)
    {
        _dataObject.DeleteRecord(customer);
    }

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

    public virtual void ShowAll()
    {
        Console.WriteLine("Customer Group: " + group);
        _dataObject.ShowAllRecords();
    }
}

/// <summary>
/// The 'RefinedAbstraction' class
/// </summary>
class Customers : CustomersBase
{
    // Constructor
    public Customers(string group)
        : base(group)
    {
    }

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

```

```

        Console.WriteLine("-----");
    }
}

/// <summary>
/// The 'Implementor' abstract class
/// </summary>
abstract class DataObject
{
    public abstract void NextRecord();
    public abstract void PriorRecord();
    public abstract void AddRecord(string name);
    public abstract void DeleteRecord(string name);
    public abstract void ShowRecord();
    public abstract void ShowAllRecords();
}

/// <summary>
/// The 'ConcreteImplementor' class
/// </summary>
class CustomersData : DataObject
{
    private List<string> _customers = new List<string>();
    private int _current = 0;

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

    public override void NextRecord()
    {
        if (_current <= _customers.Count - 1)
        {
            _current++;
        }
    }

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

    public override void AddRecord(string customer)
    {
        _customers.Add(customer);
    }

    public override void DeleteRecord(string customer)
    {
        _customers.Remove(customer);
    }

    public override void ShowRecord()
    {
        Console.WriteLine(_customers[_current]);
    }

    public override void ShowAllRecords()
    {

```

```

foreach (string customer in _customers)
{
    Console.WriteLine(" " + customer);
}
}
}

```

Output

```

Jim Jones
Samual Jackson
Allen Good

```

```

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

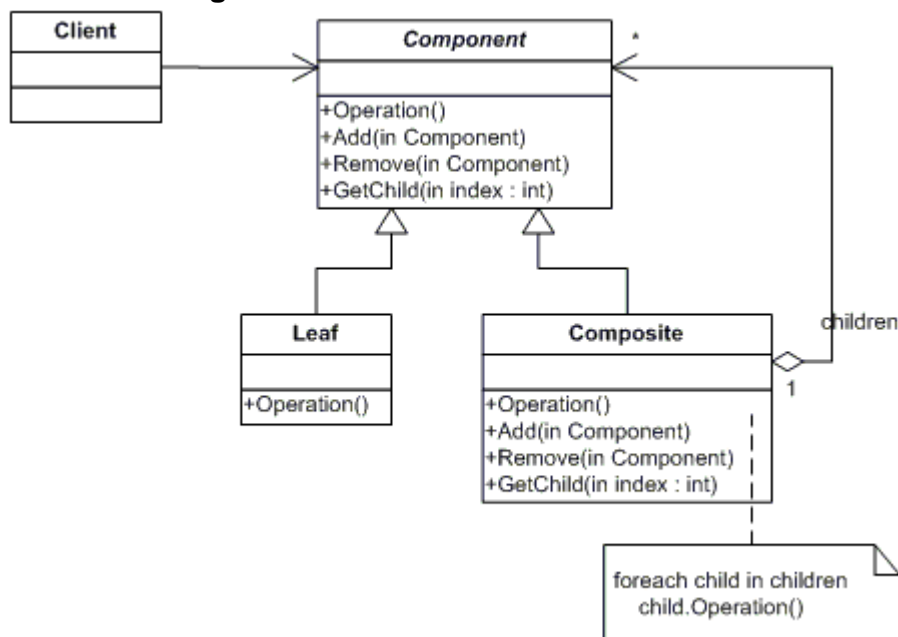
```

8. Composite Design Pattern

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.

sample code in C#

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

```
// Composite pattern -- Structural example

using System;
using System.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).

```
// 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

```

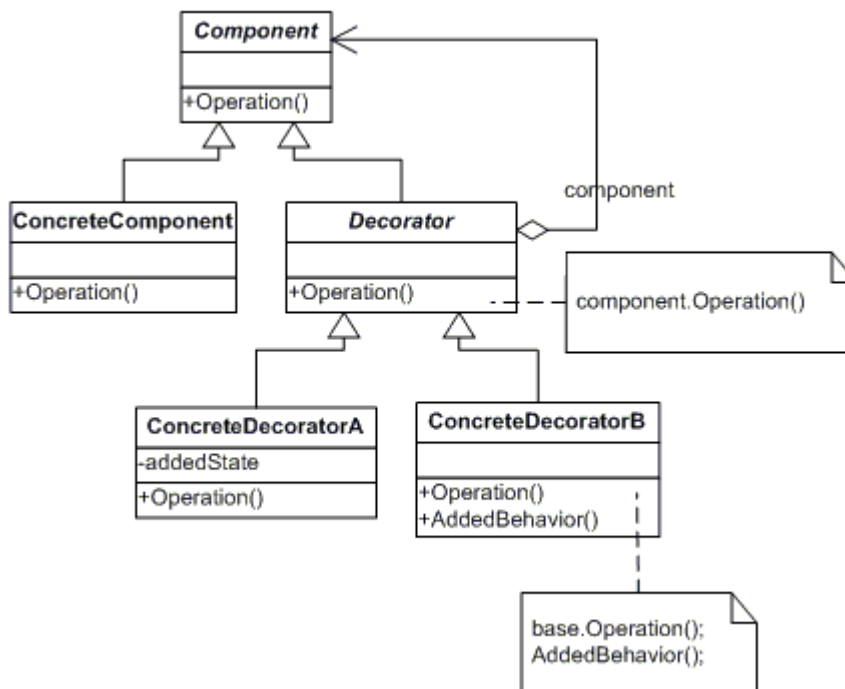
9. Decorator Design Pattern

Definition

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

Frequency of use:  1 2 3 4 5 nbsp;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.

sample code in C#

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

```
// Decorator pattern -- Structural example

using System;

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).

```

// 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

```

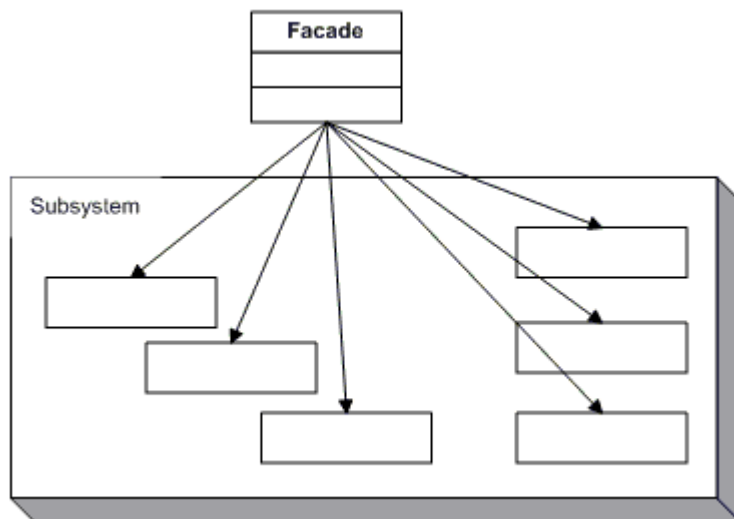
10. Facade Design Pattern

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

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.

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

sample code in C#

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

```
// Facade pattern -- Structural example

using System;

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 creditworthiness of an applicant.

```
// 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 creditworthiness 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

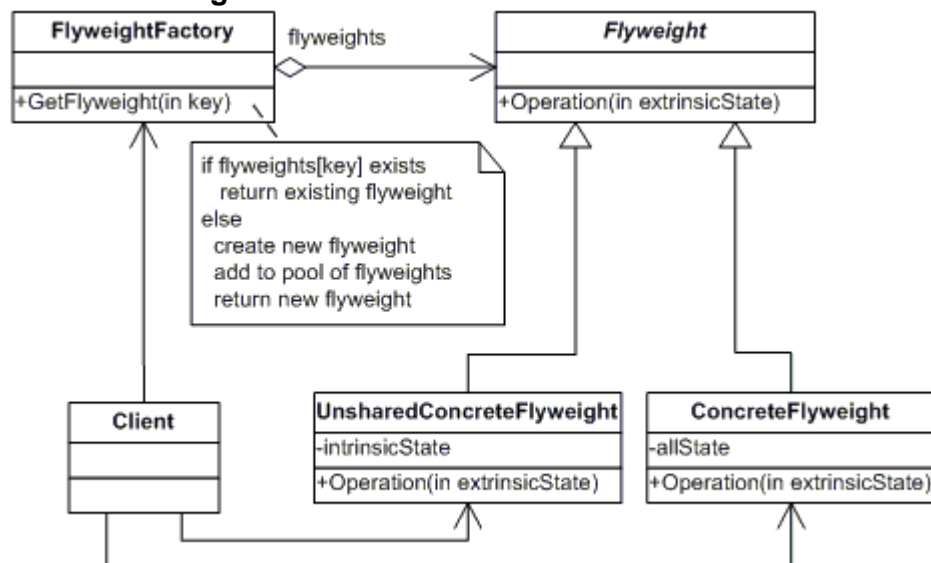
11. Flyweight Design Pattern

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).

sample code in C#

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

```
// Flyweight pattern -- Structural example

using System;
using System.Collections;

namespace DoFactory.GangOfFour.Flyweight.Structural
{
    /// <summary>
    /// MainApp startup class for Structural
    /// Flyweight Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            // Arbitrary extrinsic state
            int extrinsicstate = 22;

            FlyweightFactory factory = new FlyweightFactory();

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

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

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

            UnsharedConcreteFlyweight fu = new
                UnsharedConcreteFlyweight();

            fu.Operation(--extrinsicstate);

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

    /// <summary>
    /// The 'FlyweightFactory' class
    /// </summary>
    class FlyweightFactory
    {
        private Hashtable flyweights = new Hashtable();

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

```

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

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

/// <summary>
/// The 'ConcreteFlyweight' class
/// </summary>
class ConcreteFlyweight : Flyweight
{
    public override void Operation(int extrinsicstate)
    {
        Console.WriteLine("ConcreteFlyweight: " + extrinsicstate);
    }
}

/// <summary>
/// The 'UnsharedConcreteFlyweight' class
/// </summary>
class UnsharedConcreteFlyweight : Flyweight
{
    public override void Operation(int extrinsicstate)
    {
        Console.WriteLine("UnsharedConcreteFlyweight: " +
            extrinsicstate);
    }
}
}

```

Output

```

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

```

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

```

// Flyweight pattern -- Real World example

using System;
using System.Collections.Generic;

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

```

```

{
    // Build a document with text
    string document = "AAZZBBZB";
    char[] chars = document.ToCharArray();

    CharacterFactory factory = new CharacterFactory();

    // extrinsic state
    int pointSize = 10;

    // For each character use a flyweight object
    foreach (char c in chars)
    {
        pointSize++;
        Character character = factory.GetCharacter(c);
        character.Display(pointSize);
    }

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

/// <summary>
/// The 'FlyweightFactory' class
/// </summary>
class CharacterFactory
{
    private Dictionary<char, Character> _characters =
        new Dictionary<char, Character>();

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

/// <summary>
/// The 'Flyweight' abstract class
/// </summary>
abstract class Character
{
    protected char symbol;
    protected int width;
    protected int height;
    protected int ascent;
    protected int descent;
    protected int pointSize;

    public abstract void Display(int pointSize);
}

```



```

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

    public override void Display(int pointSize)
    {
        this.pointSize = pointSize;
        Console.WriteLine(this.symbol +
            " (" + pointSize + " " + this.pointSize + ")");
    }
}

/// <summary>
/// A 'ConcreteFlyweight' class
/// </summary>
class CharacterB : Character
{
    // Constructor
    public CharacterB()
    {
        this.symbol = 'B';
        this.height = 100;
        this.width = 140;
        this.ascent = 72;
        this.descent = 0;
    }

    public override void Display(int pointSize)
    {
        this.pointSize = pointSize;
        Console.WriteLine(this.symbol +
            " (" + pointSize + " " + this.pointSize + ")");
    }
}

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

/// <summary>
/// A 'ConcreteFlyweight' class
/// </summary>
class CharacterZ : Character
{
    // Constructor
    public CharacterZ()
    {
        this.symbol = 'Z';
        this.height = 100;
        this.width = 100;
        this.ascent = 68;
        this.descent = 0;
    }

    public override void Display(int pointSize)
    {
        this.pointSize = pointSize;
    }
}

```

```

        Console.WriteLine(this.symbol +
            " (pointsize " + this.pointSize + ")");
    }
}

```

Output

```

A (pointsize 11)
A (pointsize 12)
Z (pointsize 13)
Z (pointsize 14)
B (pointsize 15)
B (pointsize 16)
Z (pointsize 17)
B (pointsize 18)

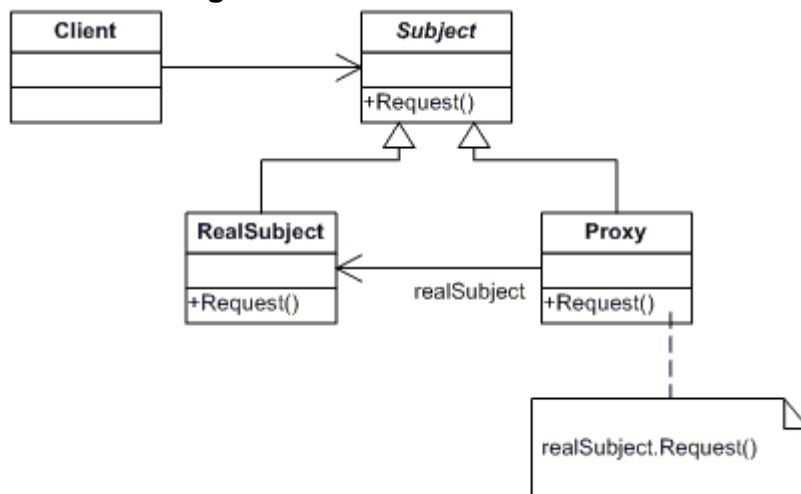
```

12. Proxy Design Pattern

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 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.

sample code in C#

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

```
// Proxy pattern -- Structural example

using System;

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.

```

// 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


```

Behavioral Patterns

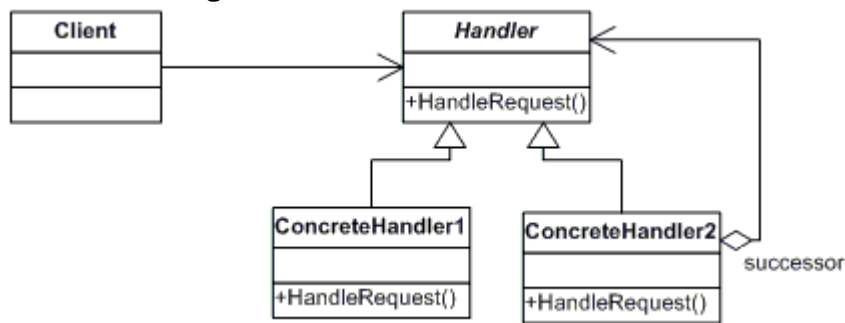
13. Chain of Responsibility Design Pattern

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

sample code in C#

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

```
// Chain of Responsibility pattern -- Structural example

using System;

namespace DoFactory.GangOfFour.Chain.Structural
{
    /// <summary>
    /// MainApp startup class for Structural
    /// Chain of Responsibility Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            // Setup Chain of Responsibility
            Handler h1 = new ConcreteHandler1();
            Handler h2 = new ConcreteHandler2();
            Handler h3 = new ConcreteHandler3();
            h1.SetSuccessor(h2);
            h2.SetSuccessor(h3);

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

```

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

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

/// <summary>
/// The 'Handler' abstract class
/// </summary>
abstract class Handler
{
    protected Handler successor;

    public void SetSuccessor(Handler successor)
    {
        this.successor = successor;
    }

    public abstract void HandleRequest(int request);
}

/// <summary>
/// The 'ConcreteHandler1' class
/// </summary>
class ConcreteHandler1 : Handler
{
    public override void HandleRequest(int request)
    {
        if (request >= 0 && request < 10)
        {
            Console.WriteLine("{0} handled request {1}",
                this.GetType().Name, request);
        }
        else if (successor != null)
        {
            successor.HandleRequest(request);
        }
    }
}

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

/// <summary>
/// The 'ConcreteHandler3' class
/// </summary>
class ConcreteHandler3 : Handler
{

```

```

public override void HandleRequest(int request)
{
    if (request >= 20 && request < 30)
    {
        Console.WriteLine("{0} handled request {1}",
            this.GetType().Name, request);
    }
    else if (successor != null)
    {
        successor.HandleRequest(request);
    }
}
}

```

Output

```

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

```

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

```

// Chain of Responsibility pattern -- Real World example

using System;

namespace DoFactory.GangOfFour.Chain.RealWorld
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Chain of Responsibility Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            // Setup Chain of Responsibility
            Approver larry = new Director();
            Approver sam = new VicePresident();
            Approver tammy = new President();

            larry.SetSuccessor(sam);
            sam.SetSuccessor(tammy);

            // Generate and process purchase requests
            Purchase p = new Purchase(2034, 350.00, "Supplies");
            larry.ProcessRequest(p);

            p = new Purchase(2035, 32590.10, "Project X");
            larry.ProcessRequest(p);

            p = new Purchase(2036, 122100.00, "Project Y");
            larry.ProcessRequest(p);
        }
    }
}

```



```

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

/// <summary>
/// The 'Handler' abstract class
/// </summary>
abstract class Approver
{
    protected Approver successor;

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

    public abstract void ProcessRequest(Purchase purchase);
}

/// <summary>
/// The 'ConcreteHandler' class
/// </summary>
class Director : Approver
{
    public override void ProcessRequest(Purchase purchase)
    {
        if (purchase.Amount < 10000.0)
        {
            Console.WriteLine("{0} approved request# {1}",
                this.GetType().Name, purchase.Number);
        }
        else if (successor != null)
        {
            successor.ProcessRequest(purchase);
        }
    }
}

/// <summary>
/// The 'ConcreteHandler' class
/// </summary>
class VicePresident : Approver
{
    public override void ProcessRequest(Purchase purchase)
    {
        if (purchase.Amount < 25000.0)
        {
            Console.WriteLine("{0} approved request# {1}",
                this.GetType().Name, purchase.Number);
        }
        else if (successor != null)
        {
            successor.ProcessRequest(purchase);
        }
    }
}

/// <summary>
/// The 'ConcreteHandler' class
/// </summary>
class President : Approver
{
    public override void ProcessRequest(Purchase purchase)
    {
        if (purchase.Amount < 100000.0)
        {
            Console.WriteLine("{0} approved request# {1}",

```

```

        this.GetType().Name, purchase.Number);
    }
    else
    {
        Console.WriteLine(
            "Request# {0} requires an executive meeting!",
            purchase.Number);
    }
}

/// <summary>
/// Class holding request details
/// </summary>
class Purchase
{
    private int number;
    private double _amount;
    private string _purpose;

    // Constructor
    public Purchase(int number, double amount, string purpose)
    {
        this._number = number;
        this._amount = amount;
        this._purpose = purpose;
    }

    // Gets or sets purchase number
    public int Number
    {
        get { return _number; }
        set { _number = value; }
    }

    // Gets or sets purchase amount
    public double Amount
    {
        get { return _amount; }
        set { _amount = value; }
    }

    // Gets or sets purchase purpose
    public string Purpose
    {
        get { return _purpose; }
        set { _purpose = value; }
    }
}
}

```

Output

```

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

```

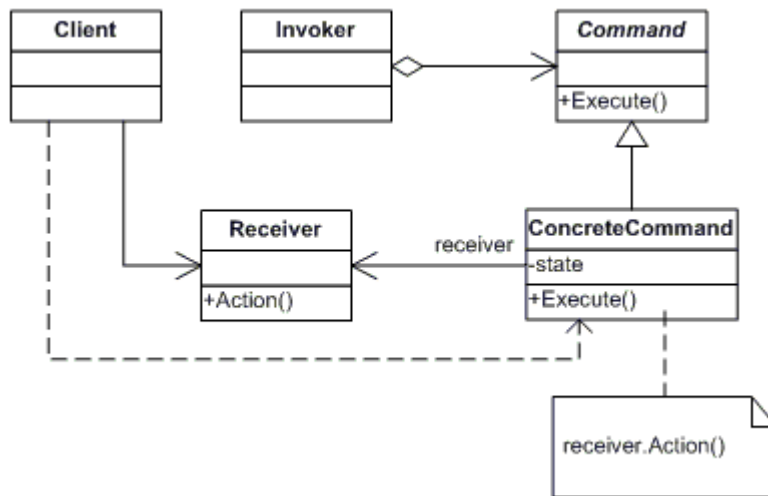
14. Command Design Pattern

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.

sample code in C#

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

```
// Command pattern -- Structural example

using System;

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.

```

// 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)

```

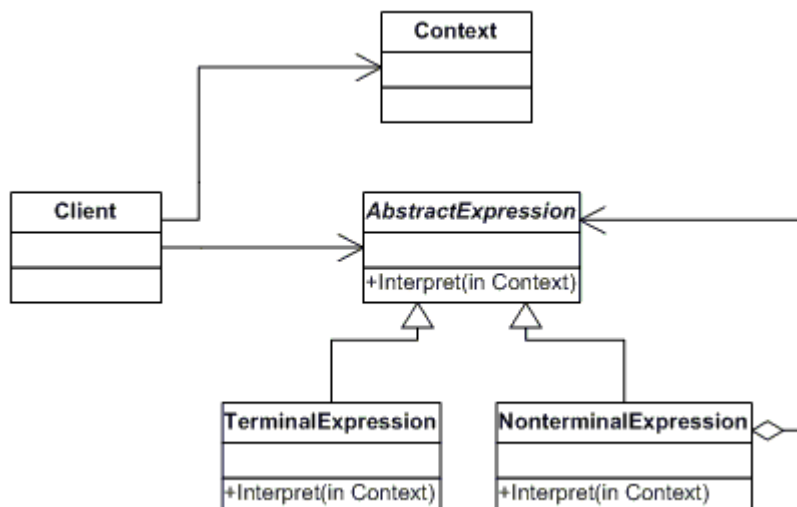
15. Interpreter Design Pattern

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 `R1` through `Rn`.
- implements an `Interpret` operation for nonterminal symbols in the grammar. `Interpret` typically calls itself recursively on the variables representing `R1` through `Rn`.
- **Context (Context)**
 - contains information that is global to the interpreter
- **Client (InterpreterApp)**
 - builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the `NonterminalExpression` and `TerminalExpression` classes
 - invokes the `Interpret` operation

sample code in C#

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

```
// Interpreter pattern -- Structural example

using System;
using System.Collections;

namespace DoFactory.GangOfFour.Interpreter.Structural
{
    /// <summary>
    /// MainApp startup class for Structural
    /// Interpreter Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            Context context = new Context();

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

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

            // Interpret
            foreach (AbstractExpression exp in list)
            {
                exp.Interpret(context);
            }

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

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

```

```

}

/// <summary>
/// The 'AbstractExpression' abstract class
/// </summary>
abstract class AbstractExpression
{
    public abstract void Interpret(Context context);
}

/// <summary>
/// The 'TerminalExpression' class
/// </summary>
class TerminalExpression : AbstractExpression
{
    public override void Interpret(Context context)
    {
        Console.WriteLine("Called Terminal.Interpret()");
    }
}

/// <summary>
/// The 'NonterminalExpression' class
/// </summary>
class NonterminalExpression : AbstractExpression
{
    public override void Interpret(Context context)
    {
        Console.WriteLine("Called Nonterminal.Interpret()");
    }
}
}

```

Output

```

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

```

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

```

// Interpreter pattern -- Real World example

using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Interpreter.RealWorld
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Interpreter Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            string roman = "MCMXXVIII";
            Context context = new Context(roman);

            // Build the 'parse tree'
            List<Expression> tree = new List<Expression>();

```

```

        tree.Add(new ThousandExpression());
        tree.Add(new HundredExpression());
        tree.Add(new TenExpression());
        tree.Add(new OneExpression());

        // Interpret
        foreach (Expression exp in tree)
        {
            exp.Interpret(context);
        }

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

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

/// <summary>
/// The 'Context' class
/// </summary>
class Context
{
    private string _input;
    private int _output;

    // Constructor
    public Context(string input)
    {
        this._input = input;
    }

    // Gets or sets input
    public string Input
    {
        get { return _input; }
        set { _input = value; }
    }

    // Gets or sets output
    public int Output
    {
        get { return _output; }
        set { _output = value; }
    }
}

/// <summary>
/// The 'AbstractExpression' class
/// </summary>
abstract class Expression
{
    public void Interpret(Context context)
    {
        if (context.Input.Length == 0)
            return;

        if (context.Input.StartsWith(Nine()))
        {
            context.Output += (9 * Multiplier());
            context.Input = context.Input.Substring(2);
        }
        else if (context.Input.StartsWith(Four()))
        {
            context.Output += (4 * Multiplier());
            context.Input = context.Input.Substring(2);
        }
    }
}

```

```

        else if (context.Input.StartsWith(Five()))
        {
            context.Output += (5 * Multiplier());
            context.Input = context.Input.Substring(1);
        }

        while (context.Input.StartsWith(One()))
        {
            context.Output += (1 * Multiplier());
            context.Input = context.Input.Substring(1);
        }
    }

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

/// <summary>
/// A 'TerminalExpression' class
/// <remarks>
/// Thousand checks for the Roman Numeral M
/// </remarks>
/// </summary>
class ThousandExpression : Expression
{
    public override string One() { return "M"; }
    public override string Four() { return " "; }
    public override string Five() { return " "; }
    public override string Nine() { return " "; }
    public override int Multiplier() { return 1000; }
}

/// <summary>
/// A 'TerminalExpression' class
/// <remarks>
/// Hundred checks C, CD, D or CM
/// </remarks>
/// </summary>
class HundredExpression : Expression
{
    public override string One() { return "C"; }
    public override string Four() { return "CD"; }
    public override string Five() { return "D"; }
    public override string Nine() { return "CM"; }
    public override int Multiplier() { return 100; }
}

/// <summary>
/// A 'TerminalExpression' class
/// <remarks>
/// Ten checks for X, XL, L and XC
/// </remarks>
/// </summary>
class TenExpression : Expression
{
    public override string One() { return "X"; }
    public override string Four() { return "XL"; }
    public override string Five() { return "L"; }
    public override string Nine() { return "XC"; }
    public override int Multiplier() { return 10; }
}

/// <summary>
/// A 'TerminalExpression' class
/// <remarks>

```

```

/// One checks for I, II, III, IV, V, VI, VI, VII, VIII, IX
/// </remarks>
/// </summary>
class OneExpression : Expression
{
    public override string One() { return "I"; }
    public override string Four() { return "IV"; }
    public override string Five() { return "V"; }
    public override string Nine() { return "IX"; }
    public override int Multiplier() { return 1; }
}

```

Output

MCMXXVIII = 1928

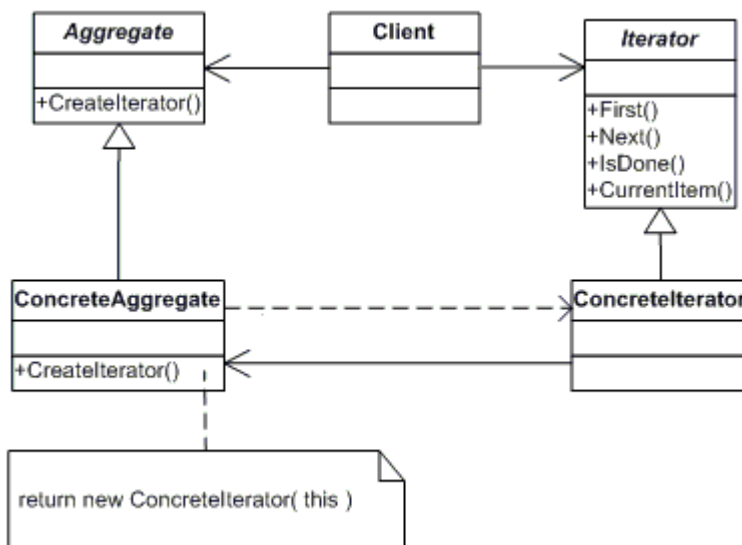
16. Iterator Design Pattern

definition

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

Frequency of use: high

UML class diagram



participants

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

- **Iterator** (**AbstractIterator**)
 - defines an interface for accessing and traversing elements.
- **ConcreteIterator** (**Iterator**)
 - implements the Iterator interface.
 - keeps track of the current position in the traversal of the aggregate.
- **Aggregate** (**AbstractCollection**)

- defines an interface for creating an Iterator object
- **ConcreteAggregate (Collection)**
 - implements the Iterator creation interface to return an instance of the proper ConcreteIterator

sample code in C#

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

```
// 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.

```

// 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

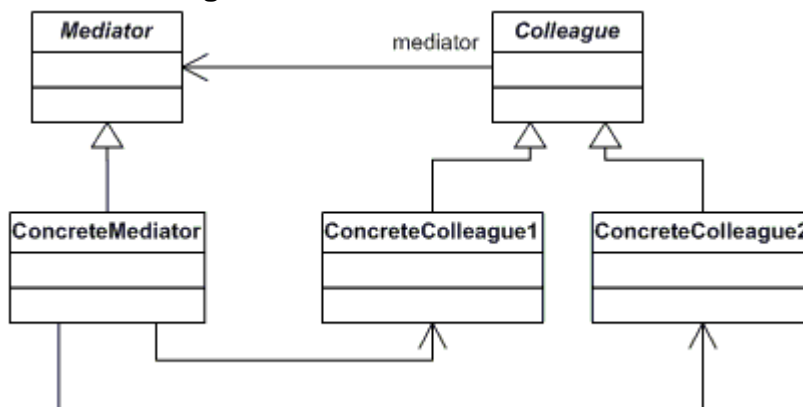
```

17. Mediator Design Pattern

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

sample code in C#

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

```

// Mediator pattern -- Structural example
using System;

namespace DoFactory.GangOfFour.Mediator.Structural
{
    /// <summary>
    /// MainApp startup class for Structural
    /// Mediator Design Pattern.
    /// </summary>

```

```

class MainApp
{
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
        ConcreteMediator m = new ConcreteMediator();

        ConcreteColleague1 c1 = new ConcreteColleague1(m);
        ConcreteColleague2 c2 = new ConcreteColleague2(m);

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

        c1.Send("How are you?");
        c2.Send("Fine, thanks");

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

/// <summary>
/// The 'Mediator' abstract class
/// </summary>
abstract class Mediator
{
    public abstract void Send(string message,
        Colleague colleague);
}

/// <summary>
/// The 'ConcreteMediator' class
/// </summary>
class ConcreteMediator : Mediator
{
    private ConcreteColleague1 _colleague1;
    private ConcreteColleague2 _colleague2;

    public ConcreteColleague1 Colleague1
    {
        set { _colleague1 = value; }
    }

    public ConcreteColleague2 Colleague2
    {
        set { _colleague2 = value; }
    }

    public override void Send(string message,
        Colleague colleague)
    {
        if (colleague == _colleague1)
        {
            _colleague2.Notify(message);
        }
        else
        {
            _colleague1.Notify(message);
        }
    }
}

/// <summary>
/// The 'Colleague' abstract class
/// </summary>
abstract class Colleague

```

```

{
    protected Mediator mediator;

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

/// <summary>
/// A 'ConcreteColleague' class
/// </summary>
class ConcreteColleague1 : Colleague
{
    // Constructor
    public ConcreteColleague1(Mediator mediator)
        : base(mediator)
    {
    }

    public void Send(string message)
    {
        mediator.Send(message, this);
    }

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

/// <summary>
/// A 'ConcreteColleague' class
/// </summary>
class ConcreteColleague2 : Colleague
{
    // Constructor
    public ConcreteColleague2(Mediator mediator)
        : base(mediator)
    {
    }

    public void Send(string message)
    {
        mediator.Send(message, this);
    }

    public void Notify(string message)
    {
        Console.WriteLine("Colleague2 gets message: "
            + message);
    }
}
}

```

Output

```

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

```

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

```
// Mediator pattern -- Real World example

using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Mediator.RealWorld
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Mediator Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            // Create chatroom
            Chatroom chatroom = new Chatroom();

            // Create participants and register them
            Participant George = new Beatle("George");
            Participant Paul = new Beatle("Paul");
            Participant Ringo = new Beatle("Ringo");
            Participant John = new Beatle("John");
            Participant Yoko = new NonBeatle("Yoko");

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

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

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

    /// <summary>
    /// The 'Mediator' abstract class
    /// </summary>
    abstract class AbstractChatroom
    {
        public abstract void Register(Participant participant);
        public abstract void Send(
            string from, string to, string message);
    }

    /// <summary>
    /// The 'ConcreteMediator' class
    /// </summary>
    class Chatroom : AbstractChatroom
    {
        private Dictionary<string, Participant> _participants =
            new Dictionary<string, Participant>();

        public override void Register(Participant participant)
        {
            if (!_participants.ContainsValue(participant))
            {

```

```

        _participants[participant.Name] = participant;
    }

    participant.Chatroom = this;
}

public override void Send(
    string from, string to, string message)
{
    Participant participant = _participants[to];

    if (participant != null)
    {
        participant.Receive(from, message);
    }
}
}

/// <summary>
/// The 'AbstractColleague' class
/// </summary>
class Participant
{
    private Chatroom _chatroom;
    private string _name;

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

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

    // Gets chatroom
    public Chatroom Chatroom
    {
        set { _chatroom = value; }
        get { return _chatroom; }
    }

    // Sends message to given participant
    public void Send(string to, string message)
    {
        _chatroom.Send(_name, to, message);
    }

    // Receives message from given participant
    public virtual void Receive(
        string from, string message)
    {
        Console.WriteLine("{0} to {1}: '{2}'",
            from, Name, message);
    }
}

/// <summary>
/// A 'ConcreteColleague' class
/// </summary>
class Beatle : Participant
{
    // Constructor
    public Beatle(string name)
        : base(name)

```

```

    {
    }

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

/// <summary>
/// A 'ConcreteColleague' class
/// </summary>
class NonBeatle : Participant
{
    // Constructor
    public NonBeatle(string name)
        : base(name)
    {
    }

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

```

Output

```

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

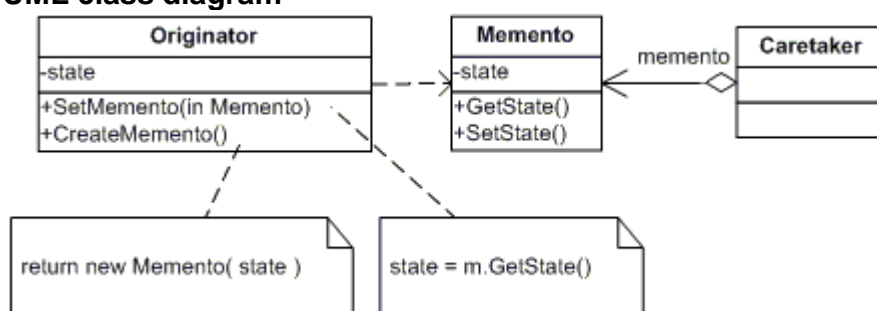
```

18. Memento Design Pattern

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.

sample code in C#

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

```
// Memento pattern -- Structural example

using System;

namespace DoFactory.GangOfFour.Memento.Structural
{
    /// <summary>
    /// MainApp startup class for Structural
    /// Memento Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            Originator o = new Originator();
            o.State = "On";

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

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

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

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

```

/// <summary>
/// The 'Originator' class
/// </summary>
class Originator
{
    private string _state;

    // Property
    public string State
    {
        get { return _state; }
        set
        {
            _state = value;
            Console.WriteLine("State = " + _state);
        }
    }

    // Creates memento
    public Memento CreateMemento()
    {
        return (new Memento(_state));
    }

    // Restores original state
    public void SetMemento(Memento memento)
    {
        Console.WriteLine("Restoring state...");
        State = memento.State;
    }
}

/// <summary>
/// The 'Memento' class
/// </summary>
class Memento
{
    private string _state;

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

    // Gets or sets state
    public string State
    {
        get { return _state; }
    }
}

/// <summary>
/// The 'Caretaker' class
/// </summary>
class Caretaker
{
    private Memento _memento;

    // Gets or sets memento
    public Memento Memento
    {
        set { _memento = value; }
        get { return _memento; }
    }
}

```

Output

```
State = On
State = Off
Restoring state:
State = On
```

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

```
// Memento pattern -- Real World example

using System;

namespace DoFactory.GangOfFour.Memento.RealWorld
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Memento Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            SalesProspect s = new SalesProspect();
            s.Name = "Noel van Halen";
            s.Phone = "(412) 256-0990";
            s.Budget = 25000.0;

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

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

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

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

    /// <summary>
    /// The 'Originator' class
    /// </summary>
    class SalesProspect
    {
        private string _name;
        private string _phone;
        private double _budget;

        // Gets or sets name
        public string Name
        {
            get { return _name; }
            set
            {
                _name = value;
            }
        }
    }
}
```

```

        Console.WriteLine("Name: " + _name);
    }
}

// Gets or sets phone
public string Phone
{
    get { return _phone; }
    set
    {
        _phone = value;
        Console.WriteLine("Phone: " + _phone);
    }
}

// Gets or sets budget
public double Budget
{
    get { return _budget; }
    set
    {
        _budget = value;
        Console.WriteLine("Budget: " + _budget);
    }
}

// Stores memento
public Memento SaveMemento()
{
    Console.WriteLine("\nSaving state --\n");
    return new Memento(_name, _phone, _budget);
}

// Restores memento
public void RestoreMemento(Memento memento)
{
    Console.WriteLine("\nRestoring state --\n");
    this.Name = memento.Name;
    this.Phone = memento.Phone;
    this.Budget = memento.Budget;
}
}

/// <summary>
/// The 'Memento' class
/// </summary>
class Memento
{
    private string _name;
    private string _phone;
    private double _budget;

    // Constructor
    public Memento(string name, string phone, double budget)
    {
        this._name = name;
        this._phone = phone;
        this._budget = budget;
    }

    // Gets or sets name
    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    // Gets or set phone

```

```

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

// Gets or sets budget
public double Budget
{
    get { return _budget; }
    set { _budget = value; }
}

/// <summary>
/// The 'Caretaker' class
/// </summary>
class ProspectMemory
{
    private Memento _memento;

    // Property
    public Memento Memento
    {
        set { _memento = value; }
        get { return _memento; }
    }
}

```

Output

```

Name:   Noel van Halen
Phone:  (412) 256-0990
Budget: 25000

Saving state --

Name:   Leo Welch
Phone:  (310) 209-7111
Budget: 1000000

Restoring state --

Name:   Noel van Halen
Phone:  (412) 256-0990
Budget: 25000

```

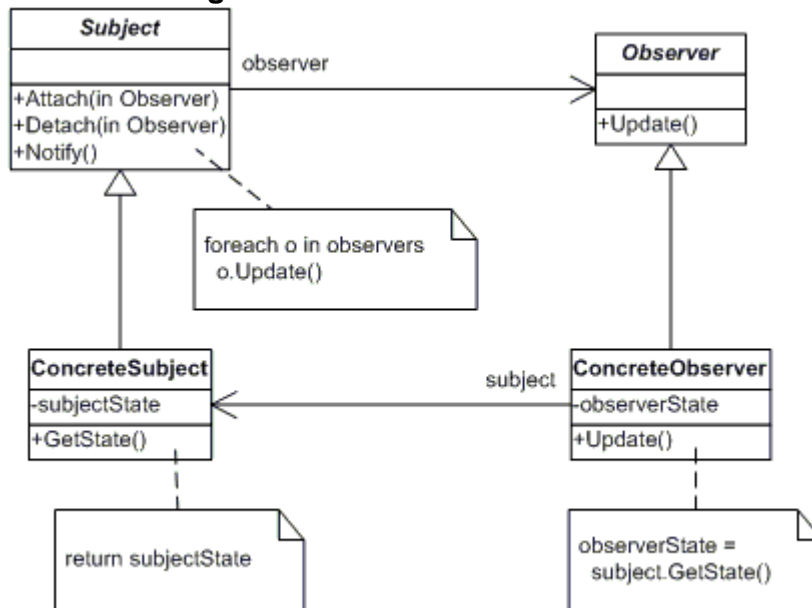
19. Observer Design Pattern

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

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

sample code in C#

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

```
// 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.

```

// 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

```

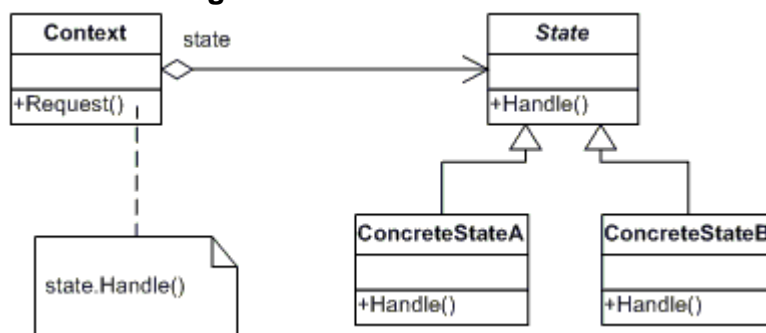
20. State Design Pattern

definition

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

Frequency of use: nbsp;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

sample code in C#

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

```
// State pattern -- Structural example

using System;

namespace DoFactory.GangOfFour.State.Structural
{
    /// <summary>
    /// MainApp startup class for Structural
    /// State Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
```

```

/// Entry point into console application.
/// </summary>
static void Main()
{
    // Setup context in a state
    Context c = new Context(new ConcreteStateA());

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

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

/// <summary>
/// The 'State' abstract class
/// </summary>
abstract class State
{
    public abstract void Handle(Context context);
}

/// <summary>
/// A 'ConcreteState' class
/// </summary>
class ConcreteStateA : State
{
    public override void Handle(Context context)
    {
        context.State = new ConcreteStateB();
    }
}

/// <summary>
/// A 'ConcreteState' class
/// </summary>
class ConcreteStateB : State
{
    public override void Handle(Context context)
    {
        context.State = new ConcreteStateA();
    }
}

/// <summary>
/// The 'Context' class
/// </summary>
class Context
{
    private State _state;

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

    // Gets or sets the state
    public State State
    {
        get { return _state; }
        set
        {
            _state = value;
        }
    }
}

```

```

        Console.WriteLine("State: " +
            _state.GetType().Name);
    }
}

public void Request()
{
    _state.Handle(this);
}
}
}

```

Output

```

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

```

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

```

// State pattern -- Real World example

using System;

namespace DoFactory.GangOfFour.State.RealWorld
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// State Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            // Open a new account
            Account account = new Account("Jim Johnson");

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

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

    /// <summary>
    /// The 'State' abstract class
    /// </summary>
    abstract class State
    {
        protected Account account;
        protected double balance;
    }
}

```

```

protected double interest;
protected double lowerLimit;
protected double upperLimit;

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

public double Balance
{
    get { return balance; }
    set { balance = value; }
}

public abstract void Deposit(double amount);
public abstract void Withdraw(double amount);
public abstract void PayInterest();
}

/// <summary>
/// A 'ConcreteState' class
/// <remarks>
/// Red indicates that account is overdrawn
/// </remarks>
/// </summary>
class RedState : State
{
    private double _serviceFee;

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

    private void Initialize()
    {
        // Should come from a datasource
        interest = 0.0;
        lowerLimit = -100.0;
        upperLimit = 0.0;
        _serviceFee = 15.00;
    }

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

    public override void Withdraw(double amount)
    {
        amount = amount - _serviceFee;
        Console.WriteLine("No funds available for withdrawal!");
    }

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

    private void StateChangeCheck()

```

```

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

/// <summary>
/// A 'ConcreteState' class
/// <remarks>
/// Silver indicates a non-interest bearing state
/// </remarks>
/// </summary>
class SilverState : State
{
    // Overloaded constructors

    public SilverState(State state) :
        this(state.Balance, state.Account)
    {
    }

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

    private void Initialize()
    {
        // Should come from a datasource
        interest = 0.0;
        lowerLimit = 0.0;
        upperLimit = 1000.0;
    }

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

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

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

    private void StateChangeCheck()
    {
        if (balance < lowerLimit)
        {
            account.State = new RedState(this);
        }
        else if (balance > upperLimit)
        {
            account.State = new GoldState(this);
        }
    }
}

```

```

/// <summary>
/// A 'ConcreteState' class
/// <remarks>
/// Gold indicates an interest bearing state
/// </remarks>
/// </summary>
class GoldState : State
{
    // Overloaded constructors
    public GoldState(State state)
        : this(state.Balance, state.Account)
    {
    }

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

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

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

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

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

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

/// <summary>
/// The 'Context' class
/// </summary>
class Account
{
    private State _state;
    private string _owner;

```



```

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

// Properties
public double Balance
{
    get { return _state.Balance; }
}

public State State
{
    get { return _state; }
    set { _state = value; }
}

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

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

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

```

Output

```

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

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

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

```

```

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

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

No funds available for withdrawal!
Withdrew $1,100.00 ---
Balance = ($582.50)
Status = RedState

```

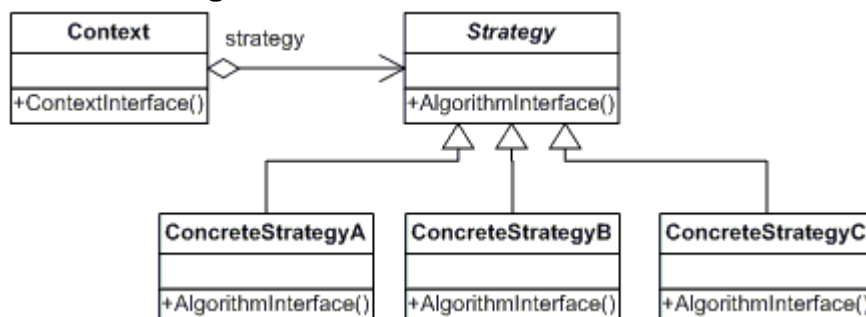
21. Strategy Design Pattern

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.

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.

```
// 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.

```

// 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

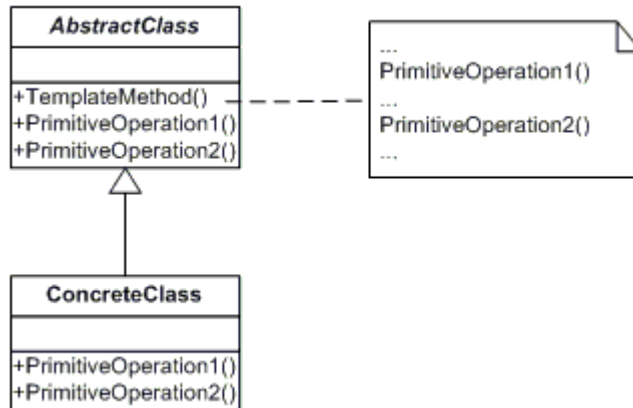
22. Template Design Pattern

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:  nbsp;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

sample code in C#

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

```
// Template pattern -- Structural example

using System;

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

```

        AbstractClass aA = new ConcreteClassA();
        aA.TemplateMethod();

        AbstractClass aB = new ConcreteClassB();
        aB.TemplateMethod();

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

/// <summary>
/// The 'AbstractClass' abstract class
/// </summary>
abstract class AbstractClass
{
    public abstract void PrimitiveOperation1();
    public abstract void PrimitiveOperation2();

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

/// <summary>
/// A 'ConcreteClass' class
/// </summary>
class ConcreteClassA : AbstractClass
{
    public override void PrimitiveOperation1()
    {
        Console.WriteLine("ConcreteClassA.PrimitiveOperation1()");
    }
    public override void PrimitiveOperation2()
    {
        Console.WriteLine("ConcreteClassA.PrimitiveOperation2()");
    }
}

/// <summary>
/// A 'ConcreteClass' class
/// </summary>
class ConcreteClassB : AbstractClass
{
    public override void PrimitiveOperation1()
    {
        Console.WriteLine("ConcreteClassB.PrimitiveOperation1()");
    }
    public override void PrimitiveOperation2()
    {
        Console.WriteLine("ConcreteClassB.PrimitiveOperation2()");
    }
}
}

```

Output

```

ConcreteClassA.PrimitiveOperation1()
ConcreteClassA.PrimitiveOperation2()

```

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


```
// Template pattern -- Real World example

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

namespace DoFactory.GangOfFour.Template.RealWorld
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Template Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            DataAccessObject daoCategories = new Categories();
            daoCategories.Run();

            DataAccessObject daoProducts = new Products();
            daoProducts.Run();

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

    /// <summary>
    /// The 'AbstractClass' abstract class
    /// </summary>
    abstract class DataAccessObject
    {
        protected string connectionString;
        protected DataSet dataSet;

        public virtual void Connect()
        {
            // Make sure mdb is available to app
            connectionString =
                "provider=Microsoft.JET.OLEDB.4.0; " +
                "data source=..\..\..\db1.mdb";
        }

        public abstract void Select();
        public abstract void Process();

        public virtual void Disconnect()
        {
            connectionString = "";
        }

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

    /// <summary>
    /// A 'ConcreteClass' class
    /// </summary>
    class Categories : DataAccessObject
    {

```

```

{
    public override void Select()
    {
        string sql = "select CategoryName from Categories";
        OleDbDataAdapter dataAdapter = new OleDbDataAdapter(
            sql, connectionString);

        dataSet = new DataSet();
        dataAdapter.Fill(dataSet, "Categories");
    }

    public override void Process()
    {
        Console.WriteLine("Categories ---- ");

        DataTable dataTable = dataSet.Tables["Categories"];
        foreach (DataRow row in dataTable.Rows)
        {
            Console.WriteLine(row["CategoryName"]);
        }
        Console.WriteLine();
    }
}

/// <summary>
/// A 'ConcreteClass' class
/// </summary>
class Products : DataAccessObject
{
    public override void Select()
    {
        string sql = "select ProductName from Products";
        OleDbDataAdapter dataAdapter = new OleDbDataAdapter(
            sql, connectionString);

        dataSet = new DataSet();
        dataAdapter.Fill(dataSet, "Products");
    }

    public override void Process()
    {
        Console.WriteLine("Products ---- ");
        DataTable dataTable = dataSet.Tables["Products"];
        foreach (DataRow row in dataTable.Rows)
        {
            Console.WriteLine(row["ProductName"]);
        }
        Console.WriteLine();
    }
}
}

```

Output

```

Categories ----
Beverages
Condiments
Confections
Dairy Products
Grains/Cereals
Meat/Poultry
Produce
Seafood

Products ----
Chai
Chang

```

```

Aniseed Syrup
Chef Anton's Cajun Seasoning
Chef Anton's Gumbo Mix
Grandma's Boysenberry Spread
Uncle Bob's Organic Dried Pears
Northwoods Cranberry Sauce
Mishi Kobe Niku

```

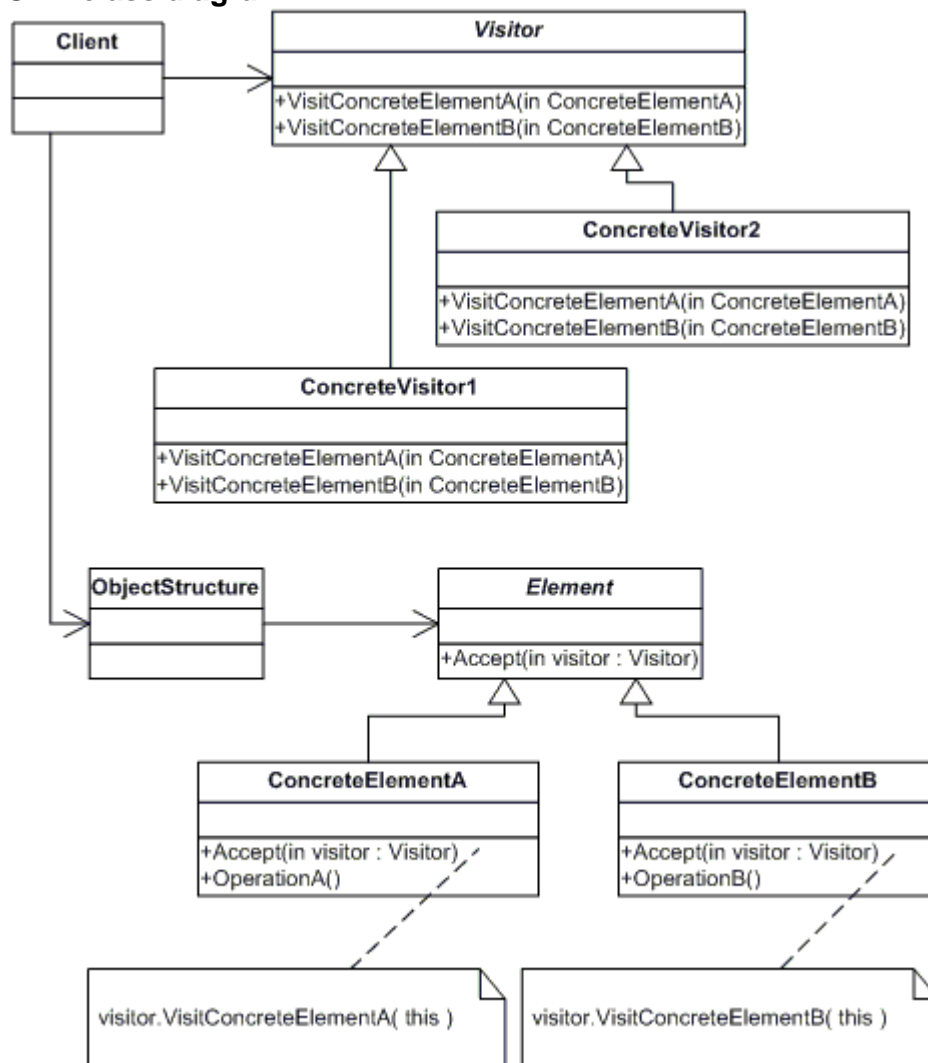
23. Visitor Design Pattern

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

sample code in C#

This **structural** code demonstrates the Visitor pattern in which an object traverses an object structure and performs the same operation on each node in this structure. Different visitor objects define different operations.

```
// Visitor pattern -- Structural example

using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Visitor.Structural
{
    /// <summary>
    /// MainApp startup class for Structural
    /// Visitor Design Pattern.
    /// </summary>
    class MainApp
    {
        static void Main()
        {
            // Setup structure
            ObjectStructure o = new ObjectStructure();
            o.Attach(new ConcreteElementA());
            o.Attach(new ConcreteElementB());

            // Create visitor objects
            ConcreteVisitor1 v1 = new ConcreteVisitor1();
            ConcreteVisitor2 v2 = new ConcreteVisitor2();

            // Structure accepting visitors
            o.Accept(v1);
            o.Accept(v2);
        }
    }
}
```

```

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

/// <summary>
/// The 'Visitor' abstract class
/// </summary>
abstract class Visitor
{
    public abstract void VisitConcreteElementA(
        ConcreteElementA concreteElementA);
    public abstract void VisitConcreteElementB(
        ConcreteElementB concreteElementB);
}

/// <summary>
/// A 'ConcreteVisitor' class
/// </summary>
class ConcreteVisitor1 : Visitor
{
    public override void VisitConcreteElementA(
        ConcreteElementA concreteElementA)
    {
        Console.WriteLine("{0} visited by {1}",
            concreteElementA.GetType().Name, this.GetType().Name);
    }

    public override void VisitConcreteElementB(
        ConcreteElementB concreteElementB)
    {
        Console.WriteLine("{0} visited by {1}",
            concreteElementB.GetType().Name, this.GetType().Name);
    }
}

/// <summary>
/// A 'ConcreteVisitor' class
/// </summary>
class ConcreteVisitor2 : Visitor
{
    public override void VisitConcreteElementA(
        ConcreteElementA concreteElementA)
    {
        Console.WriteLine("{0} visited by {1}",
            concreteElementA.GetType().Name, this.GetType().Name);
    }

    public override void VisitConcreteElementB(
        ConcreteElementB concreteElementB)
    {
        Console.WriteLine("{0} visited by {1}",
            concreteElementB.GetType().Name, this.GetType().Name);
    }
}

/// <summary>
/// The 'Element' abstract class
/// </summary>
abstract class Element
{
    public abstract void Accept(Visitor visitor);
}

/// <summary>
/// A 'ConcreteElement' class
/// </summary>
class ConcreteElementA : Element

```

```

{
    public override void Accept(Visitor visitor)
    {
        visitor.VisitConcreteElementA(this);
    }

    public void OperationA()
    {
    }
}

/// <summary>
/// A 'ConcreteElement' class
/// </summary>
class ConcreteElementB : Element
{
    public override void Accept(Visitor visitor)
    {
        visitor.VisitConcreteElementB(this);
    }

    public void OperationB()
    {
    }
}

/// <summary>
/// The 'ObjectStructure' class
/// </summary>
class ObjectStructure
{
    private List<Element> _elements = new List<Element>();

    public void Attach(Element element)
    {
        _elements.Add(element);
    }

    public void Detach(Element element)
    {
        _elements.Remove(element);
    }

    public void Accept(Visitor visitor)
    {
        foreach (Element element in _elements)
        {
            element.Accept(visitor);
        }
    }
}
}

```

Output

```

ConcreteElementA visited by ConcreteVisitor1
ConcreteElementB visited by ConcreteVisitor1
ConcreteElementA visited by ConcreteVisitor2
ConcreteElementB visited by ConcreteVisitor2

```

This **real-world** code demonstrates the Visitor pattern in which two objects traverse a list of Employees and performs the same operation on each Employee. The two visitor objects define different operations -- one adjusts vacation days and the other income.

```
// Visitor pattern -- Real World example
```

```

using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Visitor.RealWorld
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Visitor Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            // Setup employee collection
            Employees e = new Employees();
            e.Attach(new Clerk());
            e.Attach(new Director());
            e.Attach(new President());

            // Employees are 'visited'
            e.Accept(new IncomeVisitor());
            e.Accept(new VacationVisitor());

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

    /// <summary>
    /// The 'Visitor' interface
    /// </summary>
    interface IVisitor
    {
        void Visit(Element element);
    }

    /// <summary>
    /// A 'ConcreteVisitor' class
    /// </summary>
    class IncomeVisitor : IVisitor
    {
        public void Visit(Element element)
        {
            Employee employee = element as Employee;

            // Provide 10% pay raise
            employee.Income *= 1.10;
            Console.WriteLine("{0} {1}'s new income: {2:C}",
                employee.GetType().Name, employee.Name,
                employee.Income);
        }
    }

    /// <summary>
    /// A 'ConcreteVisitor' class
    /// </summary>
    class VacationVisitor : IVisitor
    {
        public void Visit(Element element)
        {
            Employee employee = element as Employee;

            // Provide 3 extra vacation days
            Console.WriteLine("{0} {1}'s new vacation days: {2}",
                employee.GetType().Name, employee.Name,

```

```

        employee.VacationDays);
    }
}

/// <summary>
/// The 'Element' abstract class
/// </summary>
abstract class Element
{
    public abstract void Accept(IVisitor visitor);
}

/// <summary>
/// The 'ConcreteElement' class
/// </summary>
class Employee : Element
{
    private string _name;
    private double _income;
    private int _vacationDays;

    // Constructor
    public Employee(string name, double income,
        int vacationDays)
    {
        this._name = name;
        this._income = income;
        this._vacationDays = vacationDays;
    }

    // Gets or sets the name
    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    // Gets or sets income
    public double Income
    {
        get { return _income; }
        set { _income = value; }
    }

    // Gets or sets number of vacation days
    public int VacationDays
    {
        get { return _vacationDays; }
        set { _vacationDays = value; }
    }

    public override void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}

/// <summary>
/// The 'ObjectStructure' class
/// </summary>
class Employees
{
    private List<Employee> _employees = new List<Employee>();

    public void Attach(Employee employee)
    {
        _employees.Add(employee);
    }
}

```



```
public void Detach(Employee employee)
{
    _employees.Remove(employee);
}

public void Accept(IVisitor visitor)
{
    foreach (Employee e in _employees)
    {
        e.Accept(visitor);
    }
    Console.WriteLine();
}

// Three employee types

class Clerk : Employee
{
    // Constructor
    public Clerk()
        : base("Hank", 25000.0, 14)
    {
    }
}

class Director : Employee
{
    // Constructor
    public Director()
        : base("Elly", 35000.0, 16)
    {
    }
}

class President : Employee
{
    // Constructor
    public President()
        : base("Dick", 45000.0, 21)
    {
    }
}
```

Output

```
Clerk Hank's new income: $27,500.00
Director Elly's new income: $38,500.00
President Dick's new income: $49,500.00

Clerk Hank's new vacation days: 14
Director Elly's new vacation days: 16
President Dick's new vacation days: 21
```

Reference: <http://www.dofactory.com/Patterns/Patterns.aspx>