# Lab 2: Tìm hiểu và cài đặt nhóm mẫu Structural (5 tiết)

Yêu cầu:

- Sinh viên đọc hiểu rõ mục đích, ý nghĩa và áp dụng ứng dụng của nhóm mẫu cấu trúc.
- Sử dụng Visual Studio cài đặc nhóm mẫu trên.
- Nộp bài báo cáo: Mỗi parttern hãy lấy 2 ví dụ thể hiện bằng sơ đồ lớp (Class diagram)

**Structural Patterns:**

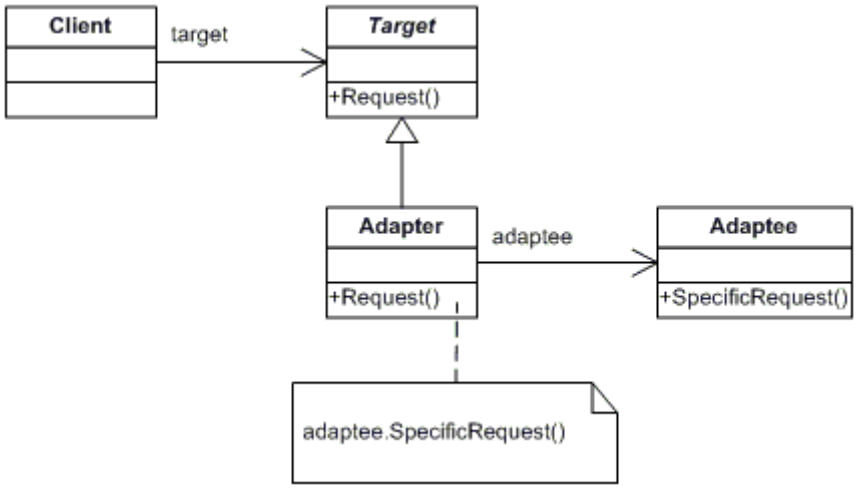
| | |
|---|---|
| **Adapter** | Match interfaces of different classes |
| **Bridge** | Separates an object’s interface from its implementation |
| **Composite** | A tree structure of simple and composite objects |
| **Decorator** | Add responsibilities to objects dynamically |
| **Facade** | A single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

## 1. Adapter

### definition

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

### 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 "H20";
      case "benzene": return "C6H6";
      case "ethanol": return "C2H5OH";
      default: return "";
    }
  }

  public double GetMolecularWeight(string compound)
  {
    switch (compound.ToLower())
    {
      case "water": return 18.015;
      case "benzene": return 78.1134;
      case "ethanol": return 46.0688;
      default: return 0d;
    }
  }
 }
}
```

Output

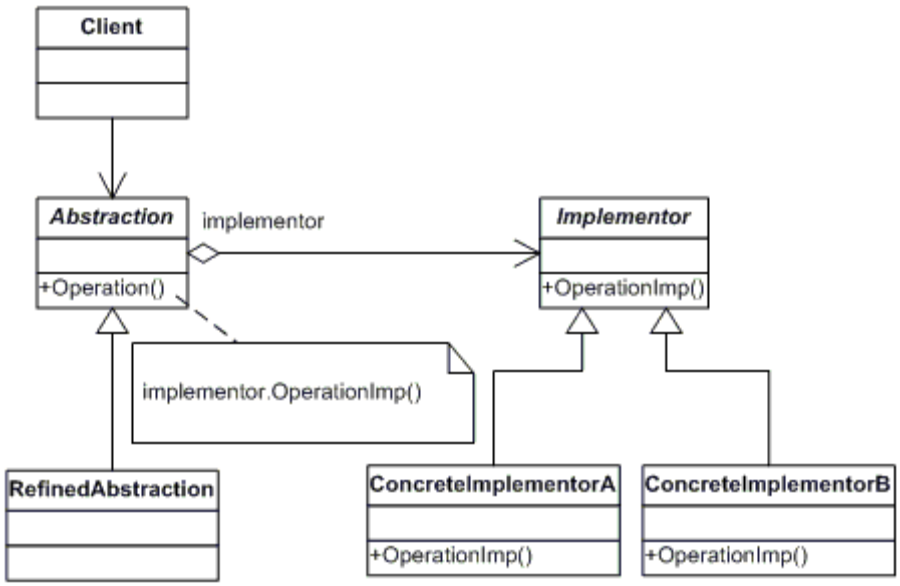
```
Compound: Unknown ------

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

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

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

## 2. Bridge

### Definition

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

### 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 implemention 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

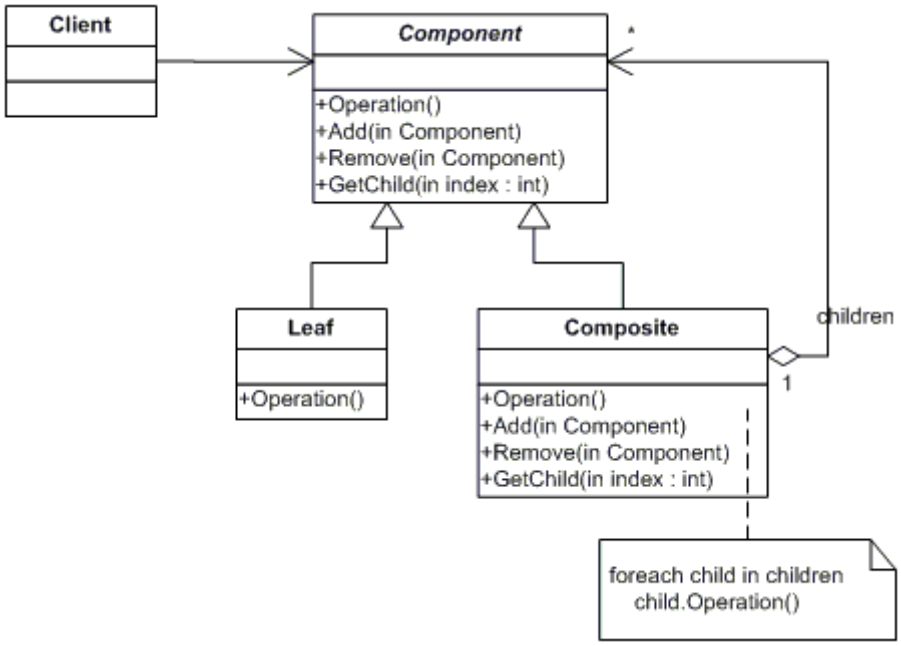
```
Jim Jones
Samual Jackson
Allen Good

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

### 3. Composite

## Definition

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

## 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

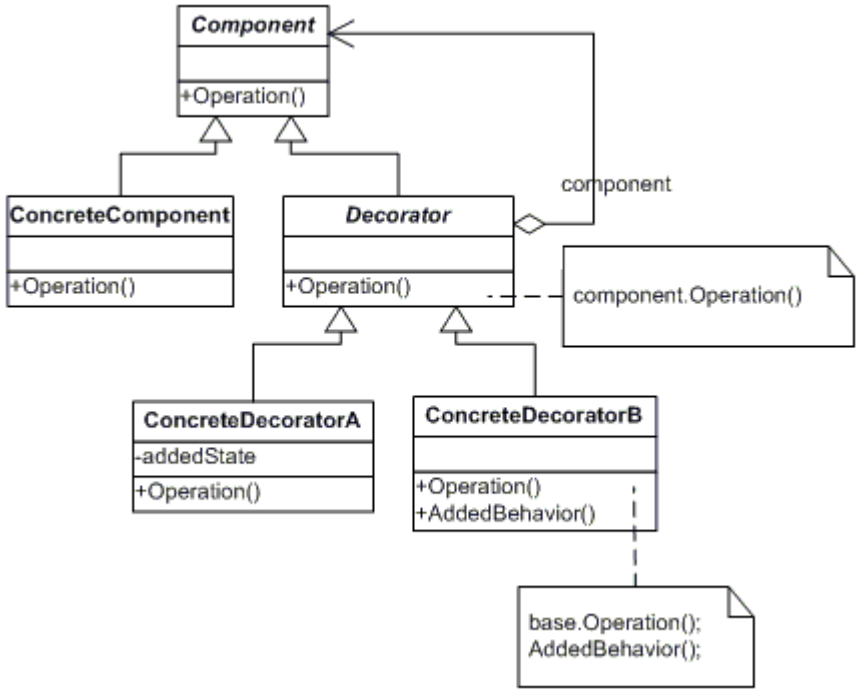
```
-+ Picture
--- Red Line
--- Blue Circle
--- Green Box
---+ Two Circles
----- Black Circle
----- White Circle
```

## 4. Decorator

## Definition

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

## 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
```