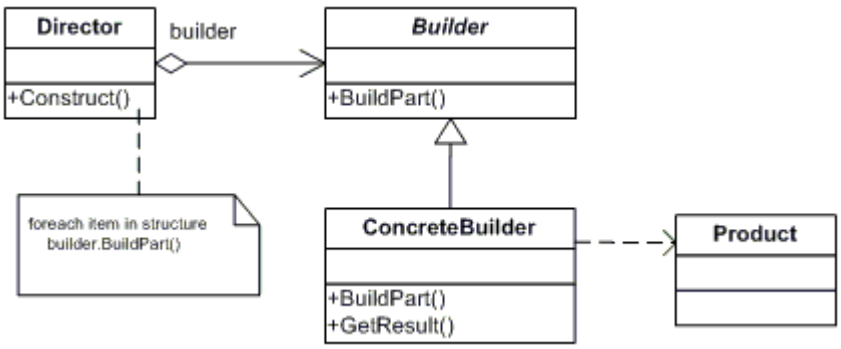
Design Patterns: C# (2)

- Builder
  - Creational pattern
- Decorator
  - Structural pattern
- Chain of responsibility
  - Behaviorial pattern

from: http://www.dofactory.com

Builder

- Separates the construction of an object from its representation in a way that construction process can create **different representations**.



- **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

C# source code:

```
// 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();
```

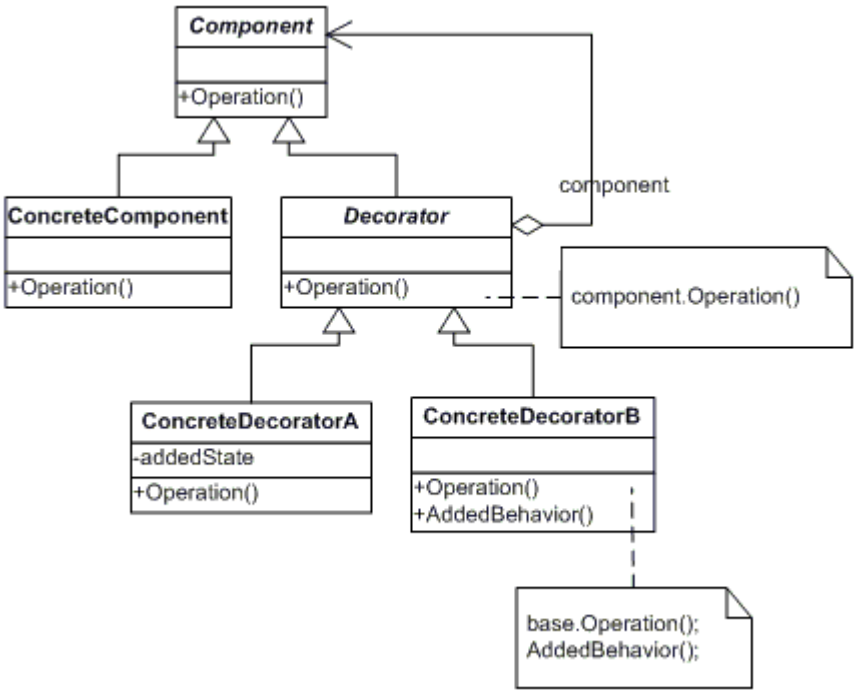
```
      p2.Show();

      // 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);
    }
  }
}
```

Decorator

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



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.

C# source code:

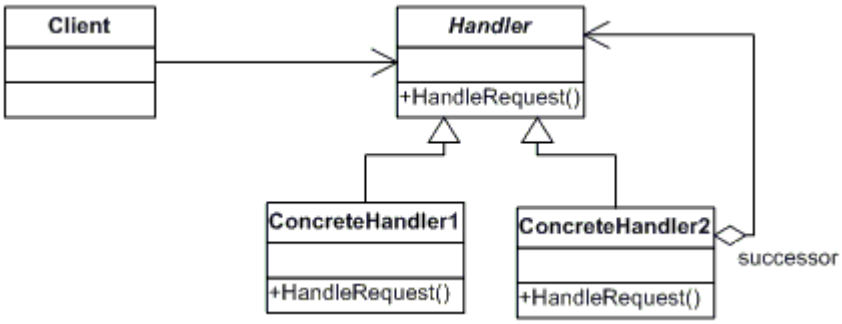
```
using System;

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()
    {
    }
  }
}
```

Chain of responsibility

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.



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

C# source code:

```
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);
      }
    }
  }
}
```