Bridge

dofactory.com/net/bridge-design-pattern

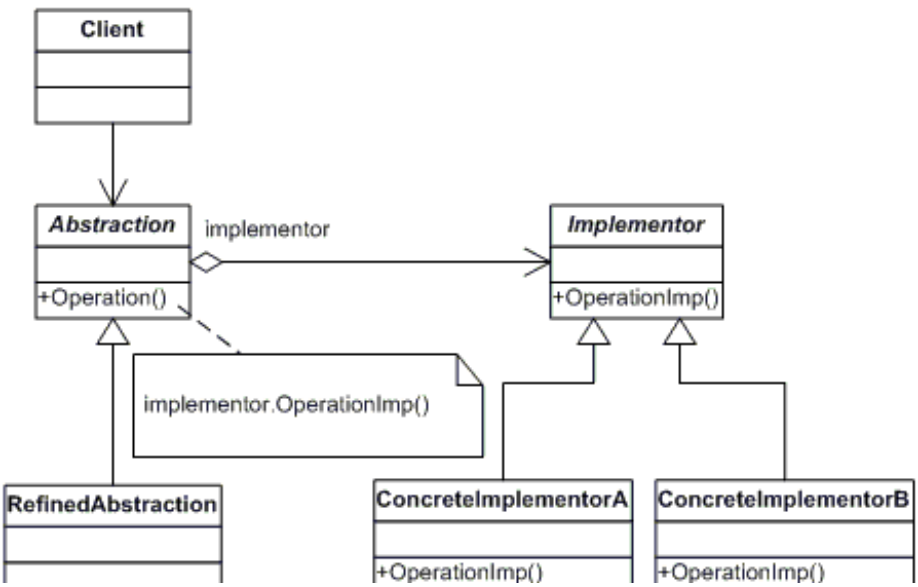
Definition

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

Frequency of use:

Medium

UML class diagram



Participants

The classes and objects participating in this pattern are:

- **Abstraction (BusinessObject)**
  - defines the abstraction's interface.
  - maintains a reference to an object of type Implementor.
- **RefinedAbstraction (CustomersBusinessObject)**
  extends the interface defined by Abstraction.
- **Implementor (DataObject)**
  defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementation interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- **ConcreteImplementor (CustomersDataObject)**
  implements the Implementor interface and defines its concrete implementation.

Structural 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.

```
1. using System;
2. namespace DoFactory.GangOfFour.Bridge.Structural
3. {
4.   /// <summary>
5.   /// MainApp startup class for Structural
6.   /// Bridge Design Pattern.
7.   /// </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; }
  }
```

```
37.     public virtual void Operation()
38.     {
39.       implementor.Operation();
40.     }
41.   }
42.   /// <summary>
43.   /// The 'Implementor' abstract class
44.   /// </summary>
45.   abstract class Implementor
46.   {
47.     public abstract void Operation();
48.   }
49.   /// <summary>
50.   /// The 'RefinedAbstraction' class
51.   /// </summary>
52.   class RefinedAbstraction : Abstraction
53.   {
54.     public override void Operation()
55.     {
56.       implementor.Operation();
57.     }
58.   }
59.   /// <summary>
60.   /// The 'ConcreteImplementorA' class
61.   /// </summary>
62.   class ConcreteImplementorA : Implementor
63.   {
64.     public override void Operation()
65.     {
```

```
66.       Console.WriteLine("ConcreteImplementorA Operation");
67.     }
68.   }
69.   /// <summary>
70.   /// The 'ConcreteImplementorB' class
71.   /// </summary>
72.   class ConcreteImplementorB : Implementor
73.   {
74.     public override void Operation()
75.     {
76.       Console.WriteLine("ConcreteImplementorB Operation");
77.     }
78.   }
79. }
```

Output

```
ConcreteImplementorA Operation
ConcreteImplementorB Operation
```

Real-world code in C#

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.

```
1. using System;
2. using System.Collections.Generic;
3. namespace DoFactory.GangOfFour.Bridge.RealWorld
4. {
5.   /// <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>
```

```
35.  class CustomersBase
36.  {
37.    private DataObject _dataObject;
38.    protected string group;
39.    public CustomersBase(string group)
40.    {
41.      this.group = group;
42.    }
43.    // Property
44.    public DataObject Data
45.    {
46.      set { _dataObject = value; }
47.      get { return _dataObject; }
48.    }
49.    public virtual void Next()
50.    {
51.      _dataObject.NextRecord();
52.    }
53.    public virtual void Prior()
54.    {
55.      _dataObject.PriorRecord();
56.    }
57.    public virtual void Add(string customer)
58.    {
59.      _dataObject.AddRecord(customer);
60.    }
61.    public virtual void Delete(string customer)
62.    {
63.      _dataObject.DeleteRecord(customer);
```

```
64.     }
65.     public virtual void Show()
66.     {
67.       _dataObject.ShowRecord();
68.     }
69.     public virtual void ShowAll()
70.     {
71.       Console.WriteLine("Customer Group: " + group);
72.       _dataObject.ShowAllRecords();
73.     }
74.   }
75.   /// <summary>
76.   /// The 'RefinedAbstraction' class
77.   /// </summary>
78.   class Customers : CustomersBase
79.   {
80.     // Constructor
81.     public Customers(string group)
82.       : base(group)
83.     {
84.     }
85.     public override void ShowAll()
86.     {
87.       // Add separator lines
88.       Console.WriteLine();
89.       Console.WriteLine("------------------------");
90.       base.ShowAll();
91.       Console.WriteLine("------------------------");
92.     }
```

```
93.   }
94.   /// <summary>
95.   /// The 'Implementor' abstract class
96.   /// </summary>
97.   abstract class DataObject
98.   {
99.     public abstract void NextRecord();
100.    public abstract void PriorRecord();
101.    public abstract void AddRecord(string name);
102.    public abstract void DeleteRecord(string name);
103.    public abstract void ShowRecord();
104.    public abstract void ShowAllRecords();
105.  }
106.  /// <summary>
107.  /// The 'ConcreteImplementor' class
108.  /// </summary>
109.  class CustomersData : DataObject
110.  {
111.    private List<string> _customers = new List<string>();
112.    private int _current = 0;
113.    public CustomersData()
114.    {
115.      // Loaded from a database
116.      _customers.Add("Jim Jones");
117.      _customers.Add("Samual Jackson");
118.      _customers.Add("Allen Good");
119.      _customers.Add("Ann Stills");
120.      _customers.Add("Lisa Giolani");
121.    }
```

```
122.    public override void NextRecord()
123.    {
124.      if (_current <= _customers.Count - 1)
125.      {
126.        _current++;
127.      }
128.    }
129.    public override void PriorRecord()
130.    {
131.      if (_current > 0)
132.      {
133.        _current--;
134.      }
135.    }
136.    public override void AddRecord(string customer)
137.    {
138.      _customers.Add(customer);
139.    }
140.    public override void DeleteRecord(string customer)
141.    {
142.      _customers.Remove(customer);
143.    }
144.    public override void ShowRecord()
145.    {
146.      Console.WriteLine(_customers[_current]);
147.    }
148.    public override void ShowAllRecords()
149.    {
150.      foreach (string customer in _customers)
```

```
151.       {
152.         Console.WriteLine(" " + customer);
153.       }
154.     }
155.   }
156. }
```

Output

```
Jim Jones
Samual Jackson
Allen Good

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

.NET Optimized code in C#

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

You can find an example on our Singleton pattern page.

All other patterns (and much more) are available in our **.NET Design Pattern Framework 4.5.**

Not only does the **.NET Design Pattern Framework 4.5** cover GOF and Enterprise patterns, it also includes .NET pattern architectures that reduce the code you need to write by up to 75%. This unique package will change your .NET lifestyle -- for only $79.

Here's what is included:

- 69 gang-of-four pattern projects
- 46 head-first pattern projects
- Fowler's enterprise patterns
- Multi-tier patterns
- Convention over configuration
- Active Record and CQRS patterns
- Repository and Unit-of-Work patterns
- MVC, MVP, & MVVM patterns
- REST patterns with Web API
- Spark™ Rapid App Dev (RAD) platform!
- Art Shop MVC Reference Application
- 100% pure source code

Discover the secrets of expert .NET developers

Creational Patterns

Structural Patterns

Behavioral Patterns

Reference Guides

Our Products