Implementation of Design Patterns

http://www.Madadyar.ir

Introduction

- Design patterns are solutions to software design problems you find again and again in real-world application development. Patterns are about reusable designs and interactions of objects.
- To give you a head start, the C# source code for each pattern is provided in *structural* From. Structural code uses type names as defined in the pattern definition and UML diagrams.

Singleton

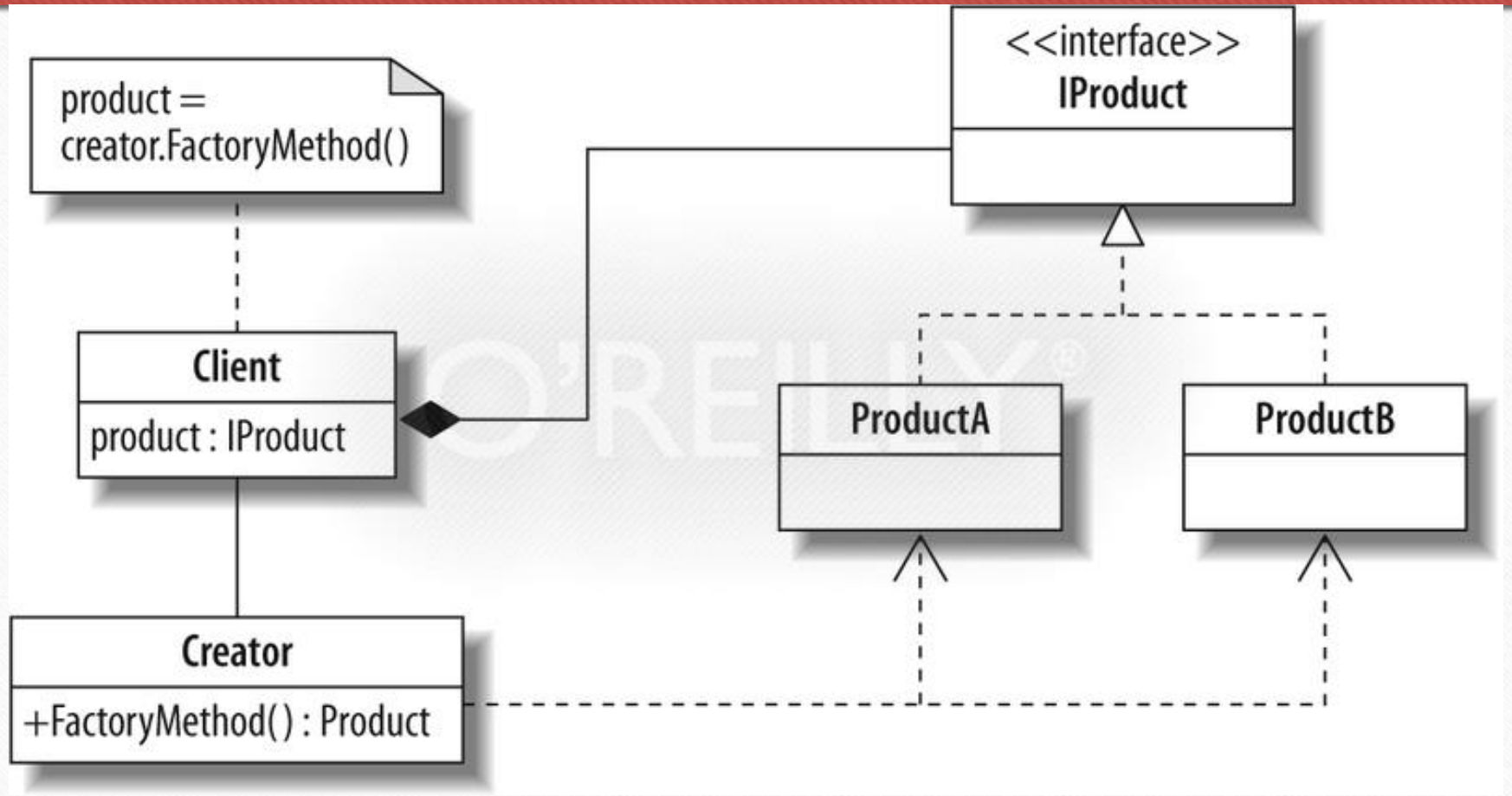
- **Singleton**
  - defines an Instance operation that lets clients access its unique instance. Instance is a class operation.
  - responsible for creating and maintaining its own unique instance.

```
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
```

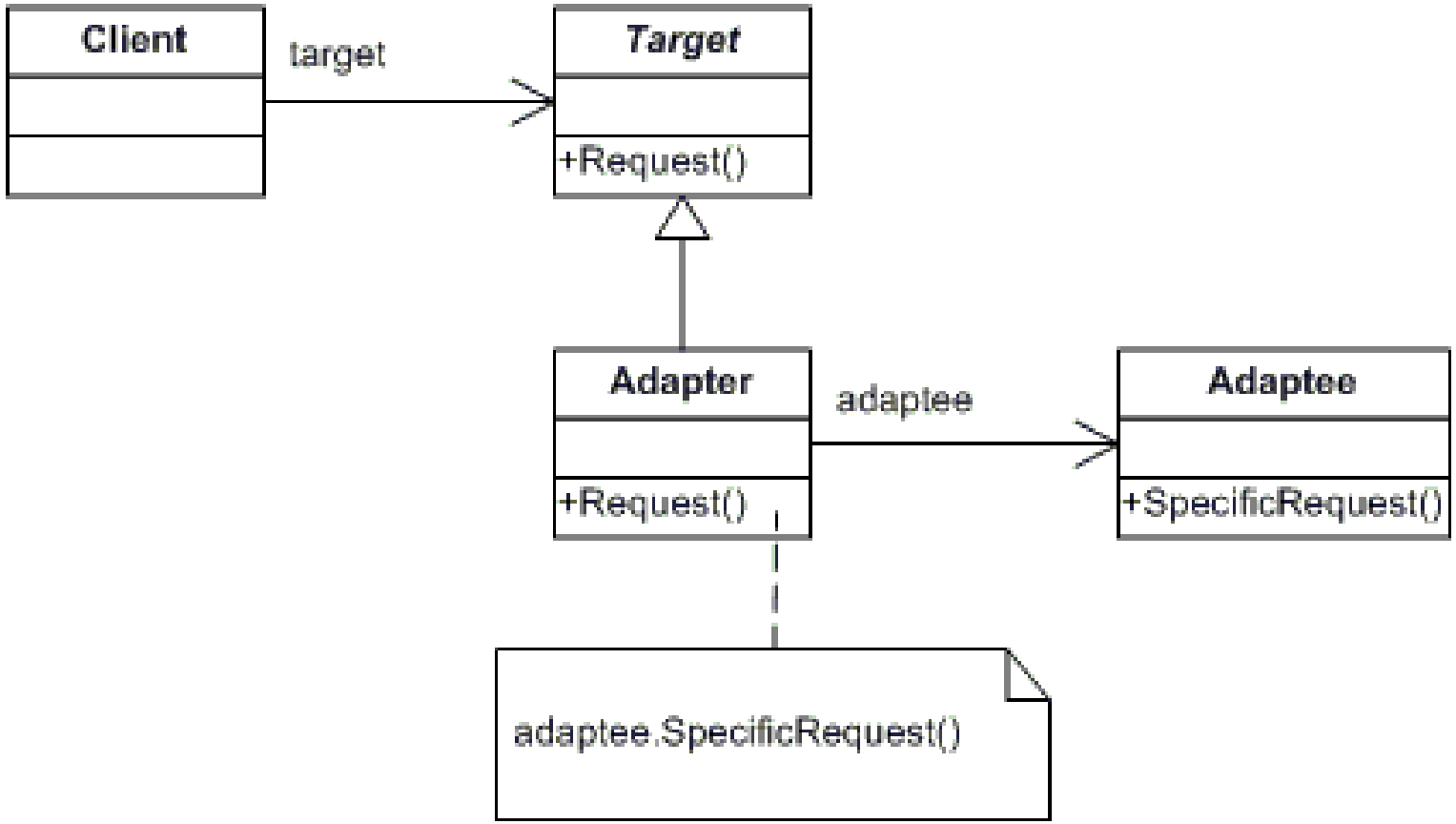
Factory Method



```
1       using System;
2       using System.Collections;
3
4       class FactoryPattern {
5
6            // Factory Method Pattern           Judith Bishop  2006
7
8          interface IProduct {
9            string ShipFrom(  );
10          }
11
12         class ProductA : IProduct {
13           public String ShipFrom (  ) {
14             return " from South Africa";
15           }
16         }
17
18         class ProductB : IProduct {
19           public String ShipFrom (  ) {
20             return "from Spain";
21           }
22         }
23
24         class DefaultProduct : IProduct {
25           public String ShipFrom (  ) {
26             return "not available";
27           }
28         }
29
30         class Creator {
31           public IProduct FactoryMethod(int month) {
32             if (month >= 4 & month <=11)
33                 return new ProductA(  );
34             else
35             if (month == 1 || month == 2 || month == 12)
36                 return new ProductB(  );
37             else return new DefaultProduct(  );
38           }
39         }
40
41         static void Main(  ) {
42           Creator c = new Creator(  );
43           IProduct product;
44
45           for (int i=1; i<=12; i++) {
46             product = c.FactoryMethod(i);
47             Console.WriteLine("Avocados "+product.ShipFrom(  ));
48           }
49         }
50      }
```

```
52/* Output
53  Avocados from Spain
54  Avocados from Spain
55  Avocados not available
56  Avocados  from South Africa
57  Avocados  from South Africa
58  Avocados  from South Africa
59  Avocados  from South Africa
60  Avocados  from South Africa
61  Avocados  from South Africa
62  Avocados  from South Africa
63  Avocados  from South Africa
64  Avocados from Spain
65  */
```

Adapter

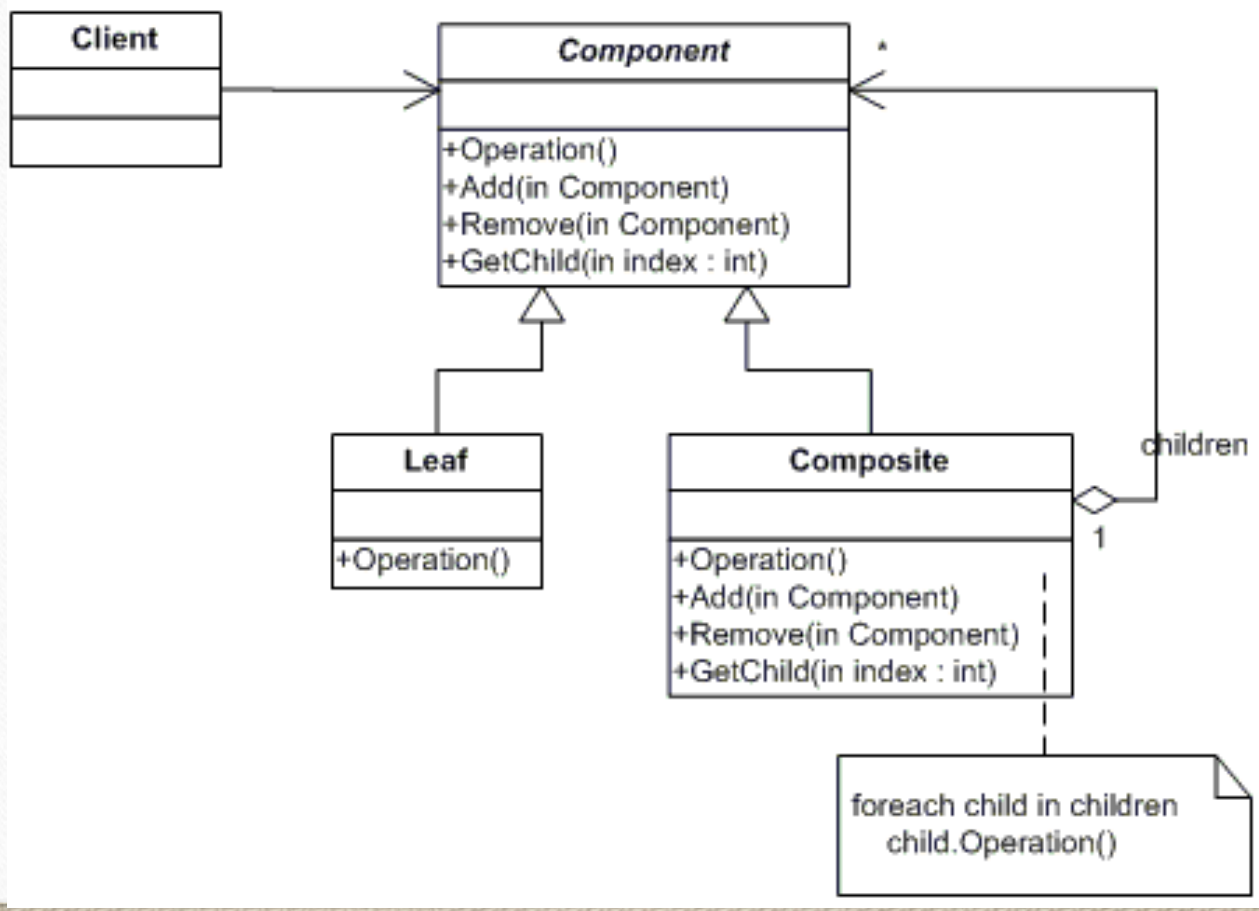

Client
target
Target
+Request()
Adapter
+Request()
adaptee
Adaptee
+SpecificRequest()
adaptee.SpecificRequest()

```
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()
```

Composite

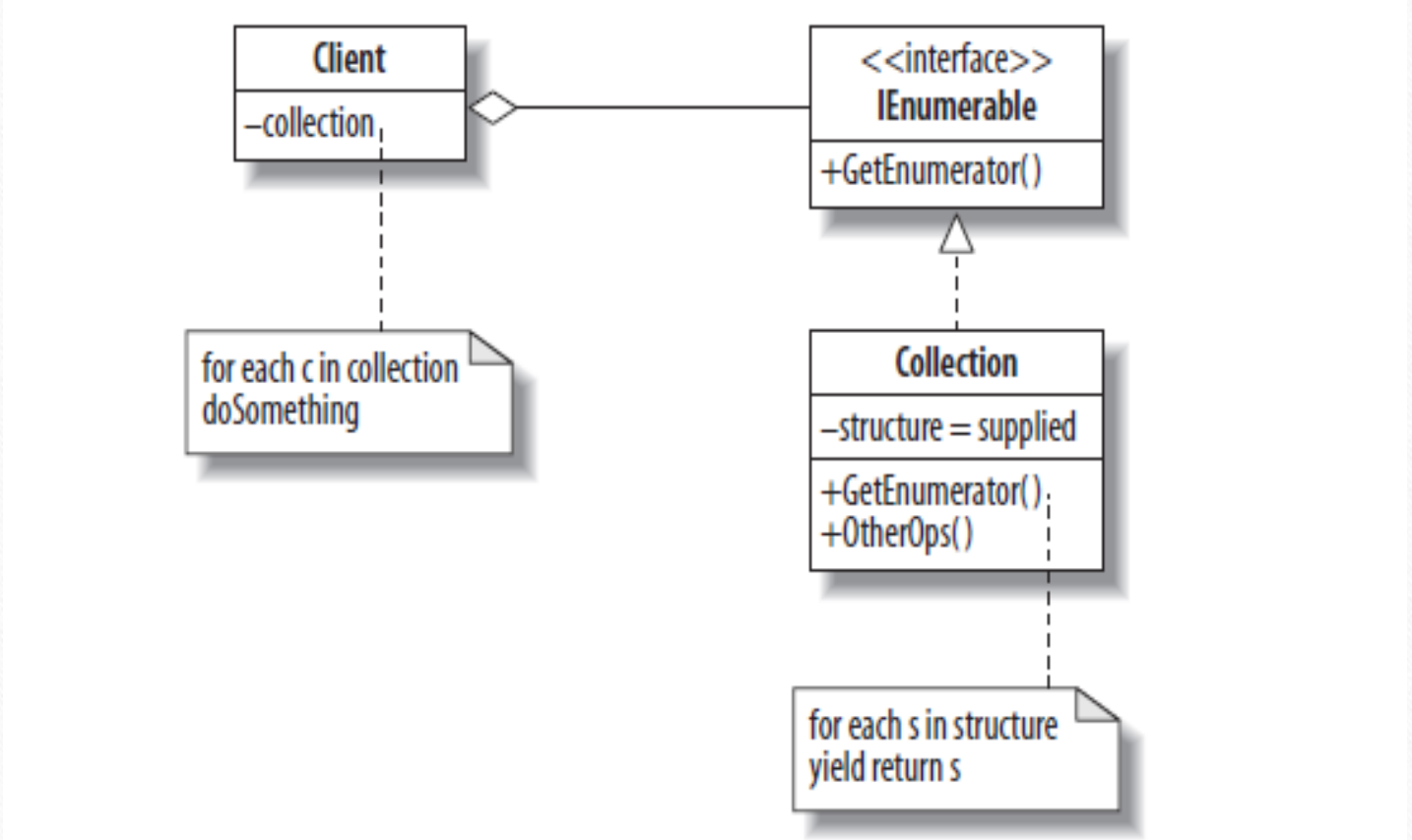

Client
Component
*
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
Leaf
+Operation()
Composite
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
children
1
foreach child in children
child.Operation()

```
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
```

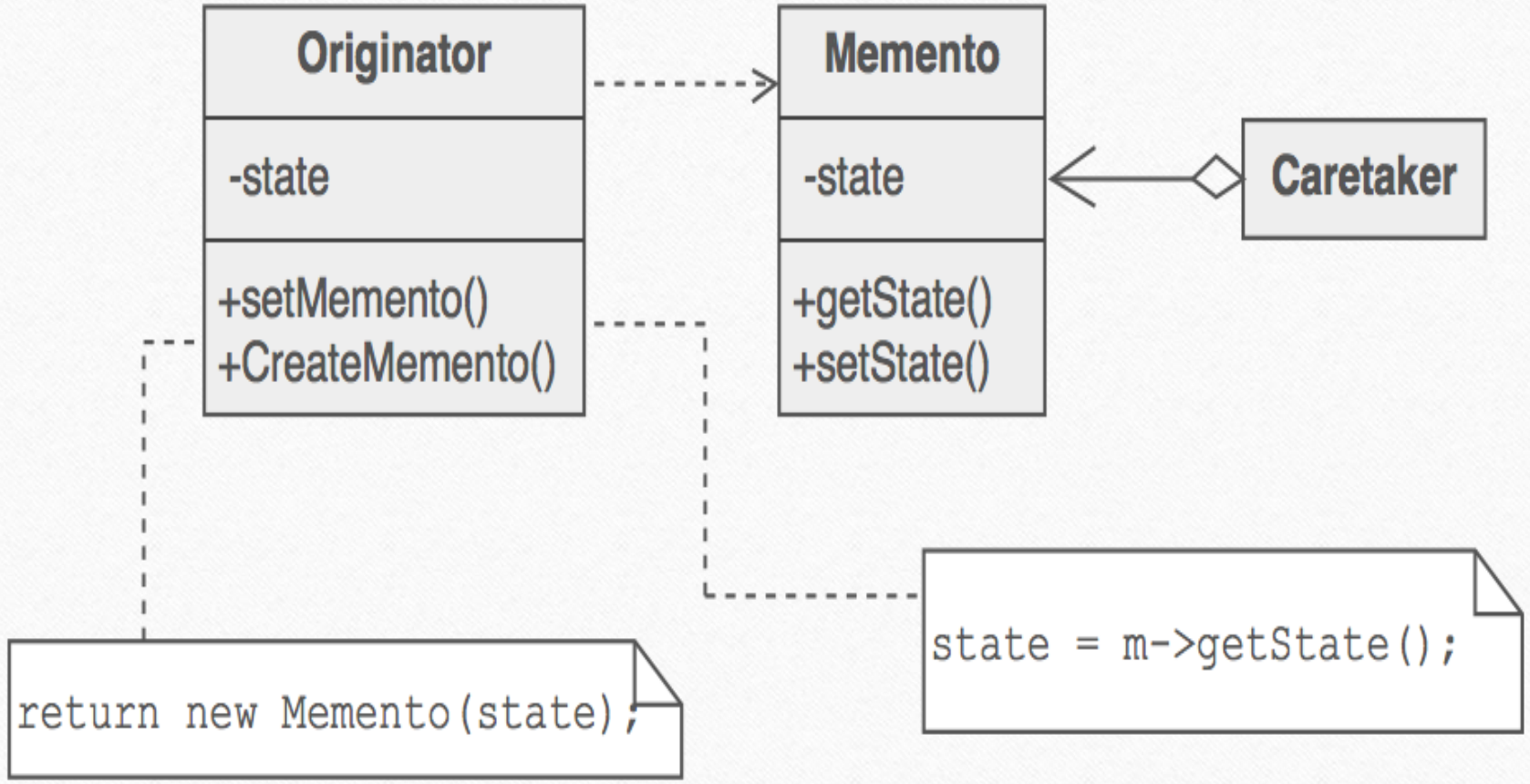
Iterator


Client
–collection
<<interface>>
IEnumerable
+GetEnumerator()
for each c in collection
doSomething
Collection
–structure = supplied
+GetEnumerator()
+OtherOps()
for each s in structure
yield return s

```
using System;
using System.Collections;
0 references
class IteratorPattern
{
    // Simplest Iterator Judith Bishop Sept 2007

    2 references
    class MonthCollection : IEnumerable
    {
        string[] months = {"January", "February", "March", "April", "May", "June",
 "July", "August", "September", "October", "November", "December"};

        2 references
        public IEnumerator GetEnumerator()
        {
            // Generates values from the collection
            foreach (string element in months)
                yield return element;
        }
    }

    0 references
    static void Main()
    {

        MonthCollection collection = new MonthCollection();
        // Consumes values generated from collection's GetEnumerator method
        foreach (string n in collection)
            Console.Write(n + " ");
        Console.WriteLine("\n");
        Console.ReadKey();
    }
}
```

```
file:///C:/Users/Madadyar/Desktop/ConsoleApplication1/ConsoleApplication1/bin/Debug/ConsoleApplication1.EXE
January February March April May June July August September October November December
```

Memento


Originator
-state
+setMemento()
+CreateMemento()
Memento
-state
+getState()
+setState()
Caretaker
return new Memento(state);
state = m->getState();

```
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
```

Reference

More Information and Real World Codes in this site:

http://www.dofactory.com/net/design-patterns