

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* Form. Structural code uses type names as defined in the pattern definition and UML diagrams.

Singleton

Singleton
-instance : Singleton
-Singleton() +Instance() : 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.

```

1.
2.
3. using System;
4.
5. namespace DoFactory.GangOfFour.Singleton.Structural
6. {
7.     /// <summary>
8.     /// MainApp startup class for Structural
9.     /// Singleton Design Pattern.
10.    /// </summary>
11.    class MainApp
12.    {
13.        /// <summary>
14.        /// Entry point into console application.
15.        /// </summary>
16.        static void Main()
17.        {
18.            // Constructor is protected -- cannot use new
19.            Singleton s1 = Singleton.Instance();
20.            Singleton s2 = Singleton.Instance();
21.
22.            // Test for same instance
23.            if (s1 == s2)
24.            {
25.                Console.WriteLine("Objects are the same instance");
26.            }
27.
28.            // Wait for user
29.            Console.ReadKey();
30.        }
31.    }
32.
33.    /// <summary>
34.    /// The 'Singleton' class
35.    /// </summary>
36.    class Singleton
37.    {
38.        private static Singleton _instance;
39.
40.        // Constructor is 'protected'
41.        protected Singleton()
42.        {
43.

```

```

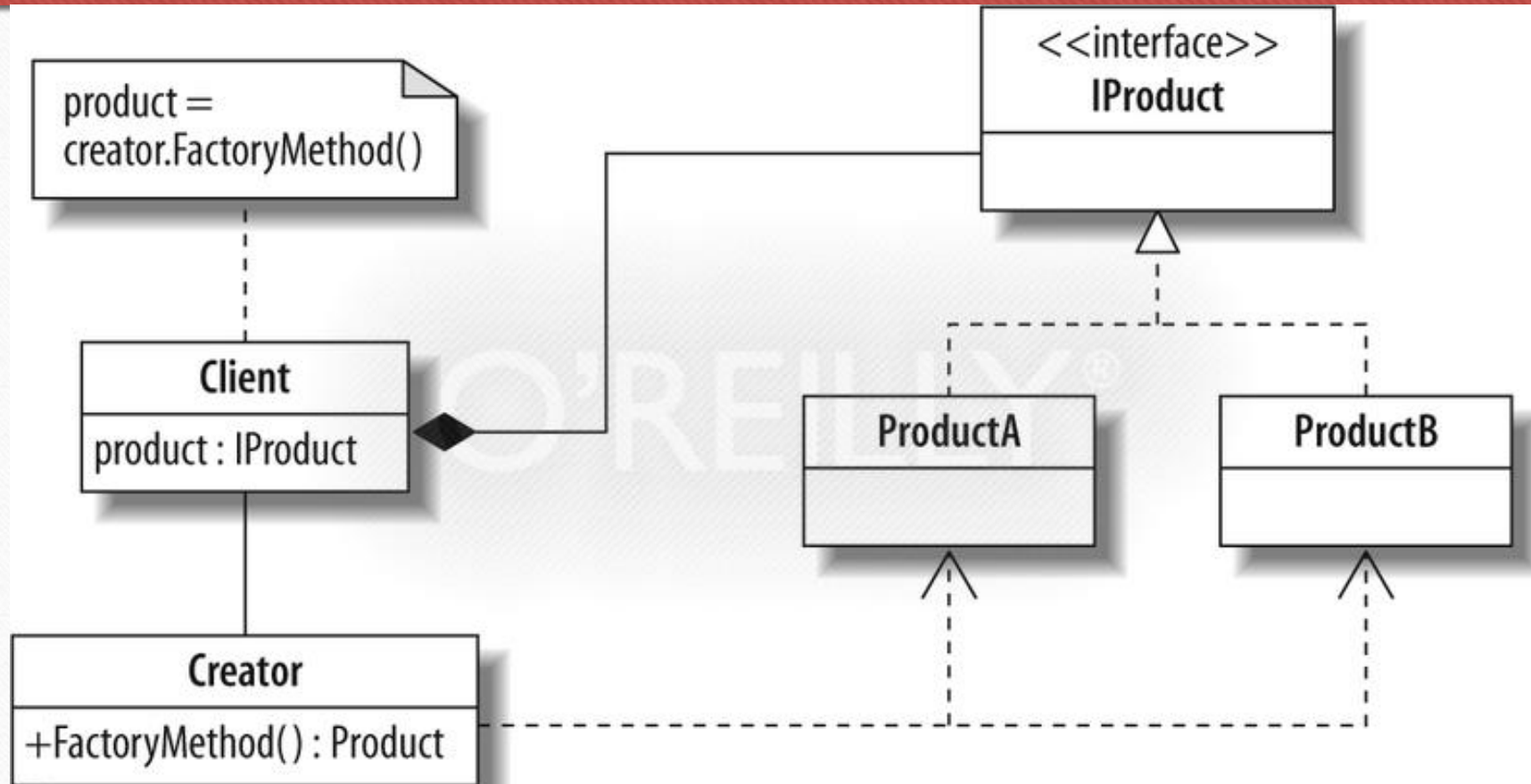
44.
45.        public static Singleton Instance()
46.        {
47.            // Uses lazy initialization.
48.            // Note: this is not thread safe.
49.            if (_instance == null)
50.            {
51.                _instance = new Singleton();
52.            }
53.
54.            return _instance;
55.        }
56.    }
57. }
58.
59.
60.
61.

```

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 }

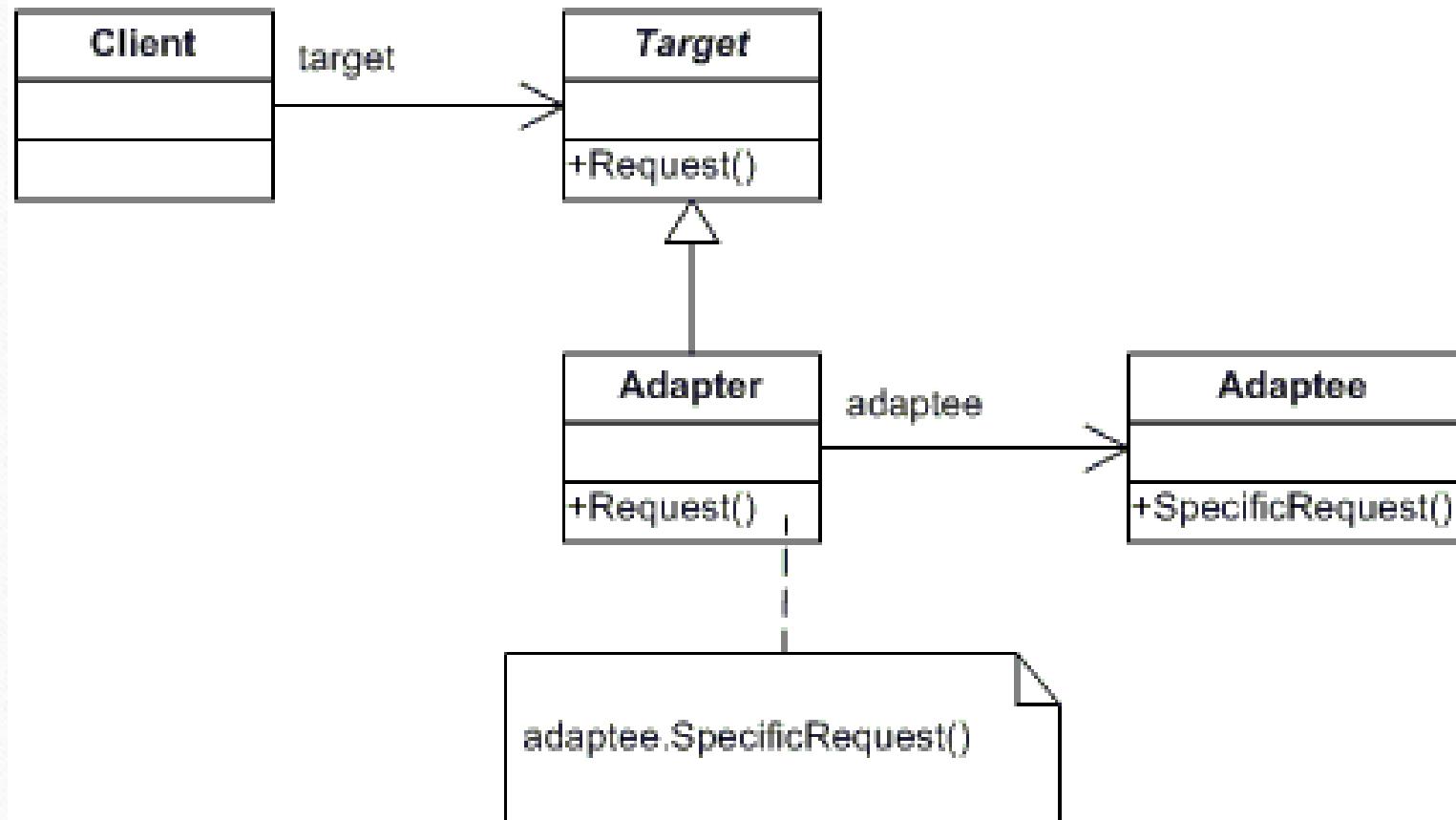
```

```

--
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



```

3. using System;
4.
5. namespace DoFactory.GangOfFour.Adapter.Structural
6. {
7.     /// <summary>
8.     /// MainApp startup class for Structural
9.     /// Adapter Design Pattern.
10.    /// </summary>
11.    class MainApp
12.    {
13.        /// <summary>
14.        /// Entry point into console application.
15.        /// </summary>
16.        static void Main()
17.        {
18.            // Create adapter and place a request
19.            Target target = new Adapter();
20.            target.Request();
21.
22.            // Wait for user
23.            Console.ReadKey();
24.        }
25.    }
26.
27.    /// <summary>
28.    /// The 'Target' class
29.    /// </summary>
30.    class Target
31.    {
32.        public virtual void Request()
33.        {
34.            Console.WriteLine("Called Target Request()");
35.        }
36.    }
37.
38.    /// <summary>
39.    /// The 'Adapter' class
40.    /// </summary>

```

```

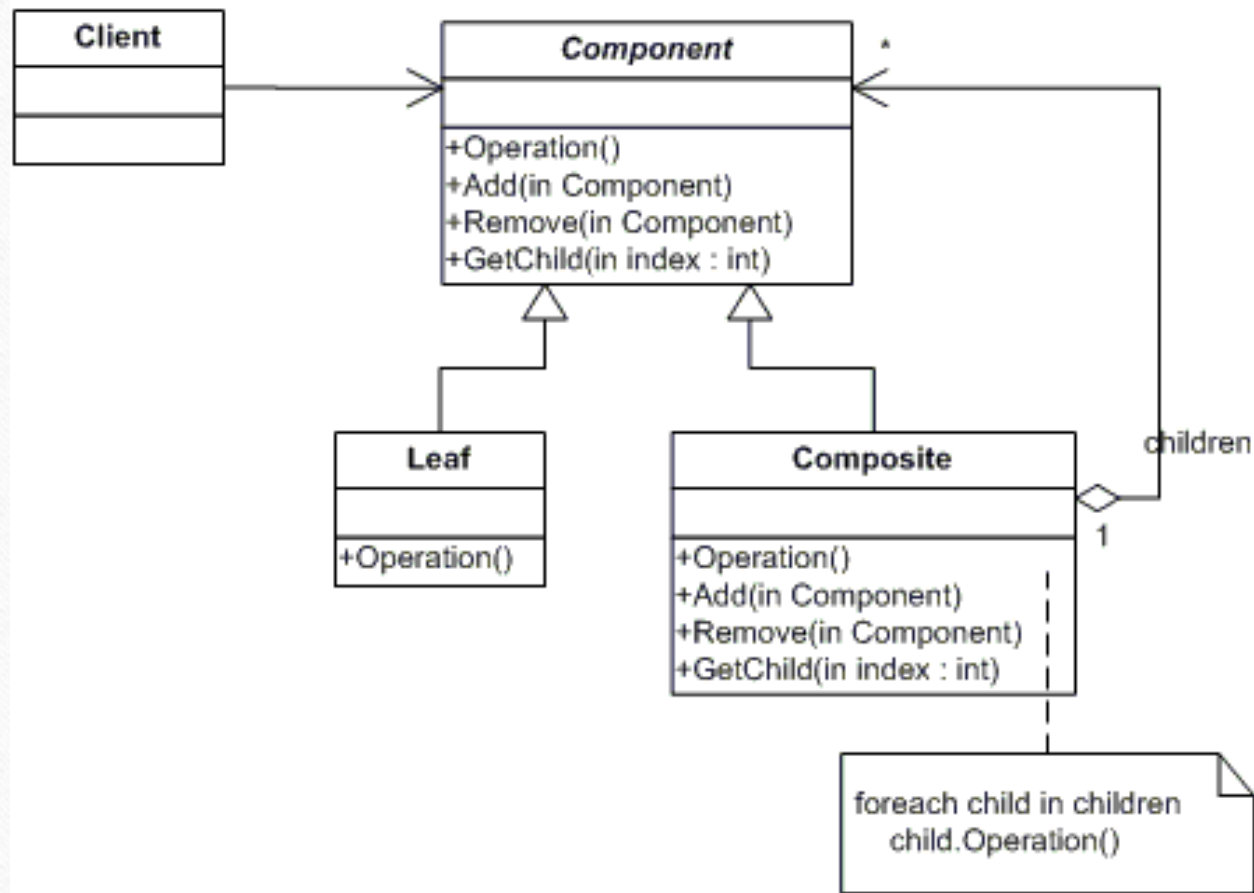
40.    /// </summary>
41.    class Adapter : Target
42.    {
43.        private Adaptee _adaptee = new Adaptee();
44.
45.        public override void Request()
46.        {
47.            // Possibly do some other work
48.            // and then call SpecificRequest
49.            _adaptee.SpecificRequest();
50.        }
51.    }
52.
53.    /// <summary>
54.    /// The 'Adaptee' class
55.    /// </summary>
56.    class Adaptee
57.    {
58.        public void SpecificRequest()
59.        {
60.            Console.WriteLine("Called SpecificRequest()");
61.        }
62.    }
63. }
64.
65.
66.
67.

```

Output

Called SpecificRequest()

Composite



```

1.
2. using System;
4. using System.Collections.Generic;
5.
6. namespace DoFactory.GangOfFour.Composite.Structural
7. {
8.     /// <summary>
9.     /// MainApp startup class for Structural
10.    /// Composite Design Pattern.
11.    /// </summary>
12.    class MainApp
13.    {
14.        /// <summary>
15.        /// Entry point into console application.
16.        /// </summary>
17.        static void Main()
18.        {
19.            // Create a tree structure
20.            Composite root = new Composite("root");
21.            root.Add(new Leaf("Leaf A"));
22.            root.Add(new Leaf("Leaf B"));
23.
24.            Composite comp = new Composite("Composite X");
25.            comp.Add(new Leaf("Leaf XA"));
26.            comp.Add(new Leaf("Leaf XB"));
27.
28.            root.Add(comp);
29.            root.Add(new Leaf("Leaf C"));
30.
31.            // Add and remove a leaf
32.            Leaf leaf = new Leaf("Leaf D");
33.            root.Add(leaf);
34.            root.Remove(leaf);
35.
36.            // Recursively display tree
37.            root.Display(1);
38.
39.            // Wait for user
40.            Console.ReadKey();
41.        }
42.    }
43.
44.    /// <summary>
45.    /// The 'Component' abstract class
46.    /// </summary>
47.    abstract class Component
48.    {
49.        protected string name;
50.

```

```

51.        // Constructor
52.        public Component(string name)
53.        {
54.            this.name = name;
55.        }
56.
57.        public abstract void Add(Component c);
58.        public abstract void Remove(Component c);
59.        public abstract void Display(int depth);
60.    }
61.
62.    /// <summary>
63.    /// The 'Composite' class
64.    /// </summary>
65.    class Composite : Component
66.    {
67.        private List<Component> _children = new List<Component>();
68.
69.        // Constructor
70.        public Composite(string name)
71.        : base(name)
72.        {
73.        }
74.
75.        public override void Add(Component component)
76.        {
77.            _children.Add(component);
78.        }
79.
80.        public override void Remove(Component component)
81.        {
82.            _children.Remove(component);
83.        }
84.
85.        public override void Display(int depth)
86.        {
87.            Console.WriteLine(new String('-', depth) + name);
88.
89.            // Recursively display child nodes
90.            foreach (Component component in _children)
91.            {
92.                component.Display(depth + 2);
93.            }
94.        }
95.    }
96.
97.    /// <summary>
98.    /// The 'Leaf' class
99.    /// </summary>
100.    class Leaf : Component

```

```

101.    {
102.        // Constructor
103.        public Leaf(string name)
104.        : base(name)
105.        {
106.        }
107.
108.        public override void Add(Component c)
109.        {
110.            Console.WriteLine("Cannot add to a leaf");
111.        }
112.
113.        public override void Remove(Component c)
114.        {
115.            Console.WriteLine("Cannot remove from a leaf");
116.        }
117.
118.        public override void Display(int depth)
119.        {
120.            Console.WriteLine(new String('-', depth) + name);
121.        }
122.    }
123. }
124.
125.
126.
127.

```

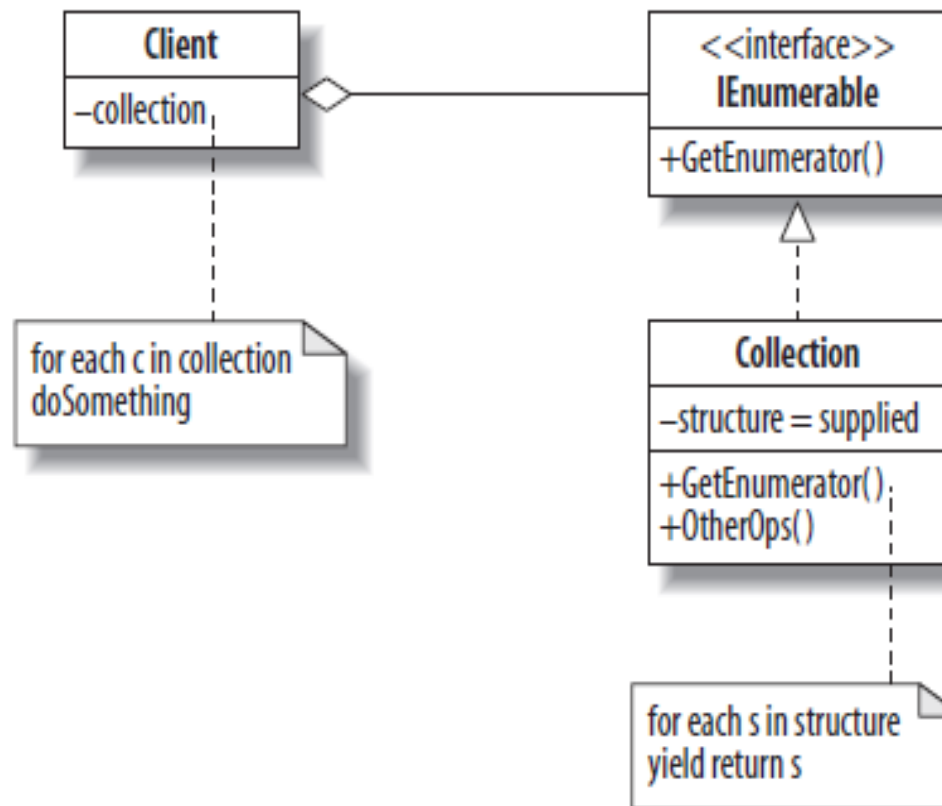
Output

```

-root
---Leaf A
---Leaf B
---Composite X
-----Leaf XA
-----Leaf XB
---Leaf C

```

Iterator



```

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();
    }
}

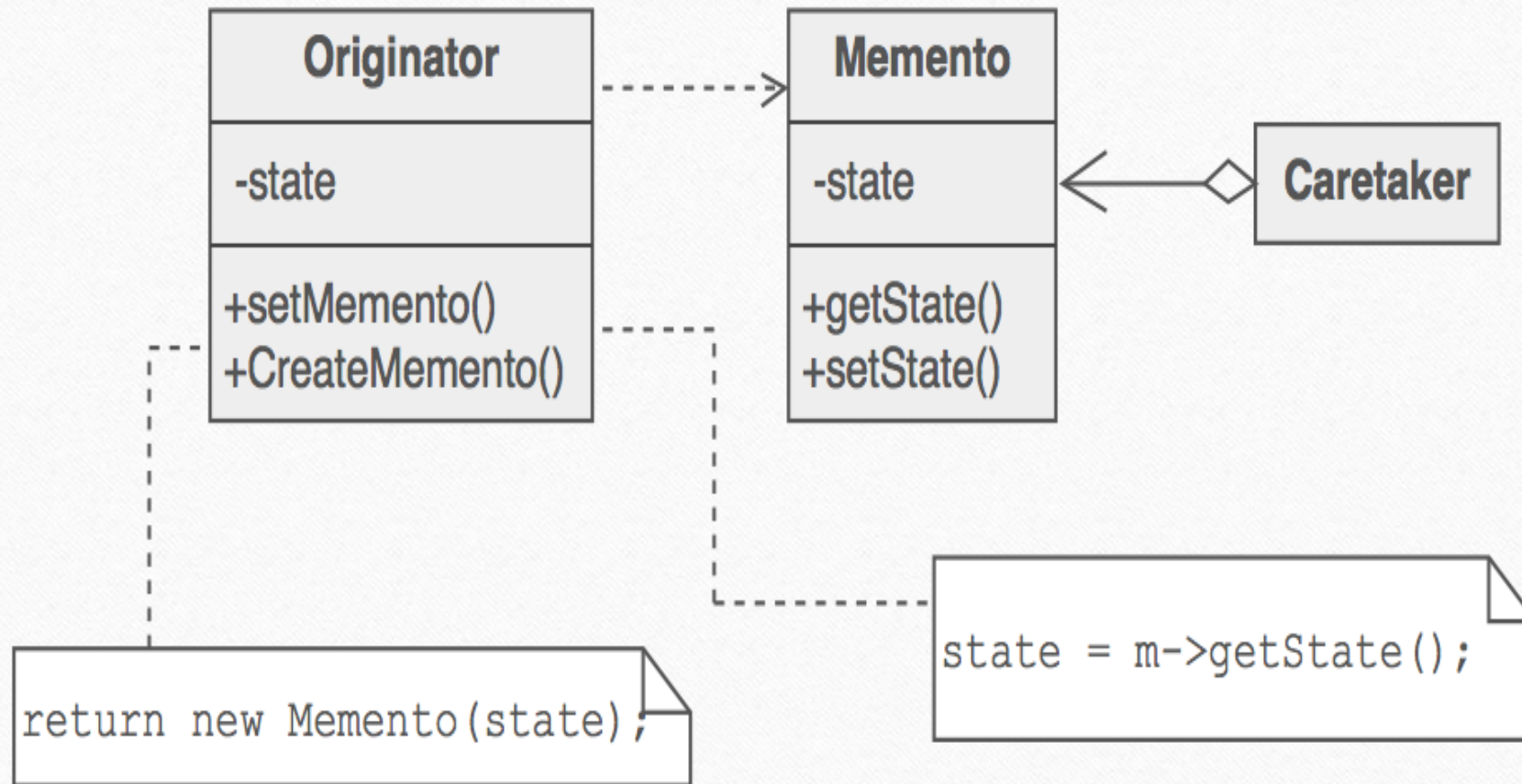
```

```

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

```

Memento



```

1. using System;
2.
3. namespace DoFactory.GangOfFour.Memento.Structural
4. {
5.     /// <summary>
6.     /// MainApp startup class for Structural
7.     /// Memento Design Pattern.
8.     /// </summary>
9.     class MainApp
10.    {
11.        /// <summary>
12.        /// Entry point into console application.
13.        /// </summary>
14.        static void Main()
15.        {
16.            Originator o = new Originator();
17.            o.State = "On";
18.
19.            // Store internal state
20.            Caretaker c = new Caretaker();
21.            c.Memento = o.CreateMemento();
22.
23.            // Continue changing originator
24.            o.State = "Off";
25.
26.            // Restore saved state
27.            o.SetMemento(c.Memento);
28.
29.            // Wait for user
30.            Console.ReadKey();
31.        }
32.    }
33.
34.    /// <summary>
35.    /// The 'Originator' class
36.    /// </summary>
37.    class Originator
38.    {
39.        private string _state;
40.
41.        // Property
42.        public string State
43.        {
44.            get { return _state; }
45.            set
46.            {
47.                _state = value;
48.                Console.WriteLine("State = " + _state);
49.            }
50.        }
51.    }

```

```

52. }
53.
54. // Creates memento
55. public Memento CreateMemento()
56. {
57.     return (new Memento(_state));
58. }
59.
60. // Restores original state
61. public void SetMemento(Memento memento)
62. {
63.     Console.WriteLine("Restoring state...");
64.     State = memento.State;
65. }
66. }
67.
68. /// <summary>
69. /// The 'Memento' class
70. /// </summary>
71. class Memento
72. {
73.     private string _state;
74.
75.     // Constructor
76.     public Memento(string state)
77.     {
78.         this._state = state;
79.     }
80.
81.     // Gets or sets state
82.     public string State
83.     {
84.         get { return _state; }
85.         set
86.         {
87.             _state = value;
88.         }
89.     }
90.
91.     /// <summary>
92.     /// The 'Caretaker' class
93.     /// </summary>
94.     class Caretaker
95.     {
96.         private Memento _memento;
97.
98.         // Gets or sets memento
99.         public Memento Memento
100.        {
101.            set { _memento = value; }
102.            get { return _memento; }
103.        }
104.    }

```

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>