Composite in C#

Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

Composite became a pretty popular solution for the most problems that require building a tree structure. Composite's great feature is the ability to run methods recursively over the whole tree structure and sum up the results.

Learn more about Composite →

Complexity: ★★☆

Popularity: ★★☆

Usage examples: The Composite pattern is pretty common in C# code. It's often used to represent hierarchies of user interface components or the code that works with graphs.

Identification: If you have an object tree, and each object of a tree is a part of the same class hierarchy, this is most likely a composite. If methods of these classes delegate the work to child objects of the tree and do it via the base class/interface of the hierarchy, this is definitely a composite.

Archive with examples

Conceptual Example

This example illustrates the structure of the **Composite** design pattern. It focuses on answering these questions:

- What classes does it consist of?
- What roles do these classes play?
- In what way the elements of the pattern are related?

Program.cs: Conceptual example

```
using System;
using System.Collections.Generic;

namespace RefactoringGuru.DesignPatterns.Composite.Conceptual
{
    // The base Component class declares common operations for both simple and
    // complex objects of a composition.
    abstract class Component
    {
        public Component() { }

        // The base Component may implement some default behavior or leave it to
        // concrete classes (by declaring the method containing the behavior as
        // "abstract").
        public abstract string Operation();

        // In some cases, it would be beneficial to define the child-management
        // operations right in the base Component class. This way, you won't
        // need to expose any concrete component classes to the client code,
        // even during the object tree assembly. The downside is that these
        // methods will be empty for the leaf-level components.
        public virtual void Add(Component component)
        {
            throw new NotImplementedException();
        }

        public virtual void Remove(Component component)
        {
            throw new NotImplementedException();
        }

        // You can provide a method that lets the client code figure out whether
        // a component can bear children.
        public virtual bool IsComposite()
        {
            return true;
        }
    }

    // The Leaf class represents the end objects of a composition. A leaf can't
    // have any children.
    //
    // Usually, it's the Leaf objects that do the actual work, whereas Composite
    // objects only delegate to their sub-components.
```

```
class Leaf : Component
{
    public override string Operation()
    {
        return "Leaf";
    }

    public override bool IsComposite()
    {
        return false;
    }
}

// The Composite class represents the complex components that may have
// children. Usually, the Composite objects delegate the actual work to
// their children and then "sum-up" the result.
class Composite : Component
{
    protected List<Component> _children = new List<Component>();

    public override void Add(Component component)
    {
        this._children.Add(component);
    }

    public override void Remove(Component component)
    {
        this._children.Remove(component);
    }

    // The Composite executes its primary logic in a particular way. It
    // traverses recursively through all its children, collecting and
    // summing their results. Since the composite's children pass these
    // calls to their children and so forth, the whole object tree is
    // traversed as a result.
    public override string Operation()
    {
        int i = 0;
        string result = "Branch(";

        foreach (Component component in this._children)
        {
            result += component.Operation();
            if (i != this._children.Count - 1)
            {
                result += "+";
            }
            i++;
        }

        return result + ")";
    }
}

class Client
{
    // The client code works with all of the components via the base
    // interface.
    public void ClientCode(Component leaf)
    {
        Console.WriteLine($"RESULT: {leaf.Operation()}\n");
    }

    // Thanks to the fact that the child-management operations are declared
    // in the base Component class, the client code can work with any
    // component, simple or complex, without depending on their concrete
    // classes.
    public void ClientCode2(Component component1, Component component2)
    {
        if (component1.IsComposite())
        {
            component1.Add(component2);
        }

        Console.WriteLine($"RESULT: {component1.Operation()}");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Client client = new Client();

        // This way the client code can support the simple leaf
        // components...
        Leaf leaf = new Leaf();
        Console.WriteLine("Client: I get a simple component:");
        client.ClientCode(leaf);

        // ...as well as the complex composites.
        Composite tree = new Composite();
        Composite branch1 = new Composite();
        branch1.Add(new Leaf());
        branch1.Add(new Leaf());
        Composite branch2 = new Composite();
        branch2.Add(new Leaf());
        tree.Add(branch1);
        tree.Add(branch2);
        Console.WriteLine("Client: Now I've got a composite tree:");
        client.ClientCode(tree);

        Console.Write("Client: I don't need to check the components classes even whe
        client.ClientCode2(tree, leaf);
    }
}
}
```

Output.txt: Execution result

```
Client: I get a simple component:
RESULT: Leaf

Client: Now I've got a composite tree:
RESULT: Branch(Branch(Leaf+Leaf)+Branch(Leaf))

Client: I don't need to check the components classes even when managing the tree:
RESULT: Branch(Branch(Leaf+Leaf)+Branch(Leaf)+Leaf)
```


RETURN

← Bridge in C#

READ NEXT

Decorator in C# →

Composite in Other Languages