31. Composite

Composite design pattern allows you to efficiently create tree-like structures. This ability can be useful in many situations. For example, if you need to write software that manages the file system of a computer, you would want to be able to create folders and files. Also, every folder may contain zero or more other folders. This is what makes a file system a good example of a tree-like structure that the Composite design pattern was intended to create.

Composite can be summarized as follows:

- There is a **Leaf** object, which cannot contain other objects.
- There is a **Composite** object, which may contain either **Leaf** objects or other **Composite** objects.
- There is a **Component** interface that both the **Composite** and the **Leaf** objects implement.
- The children of a **Composite** object are of the **Component** type, which makes it possible to add either **Leaf** or **Composite** objects to it.
- It is relatively easy to navigate through the object hierarchy and manipulate it.

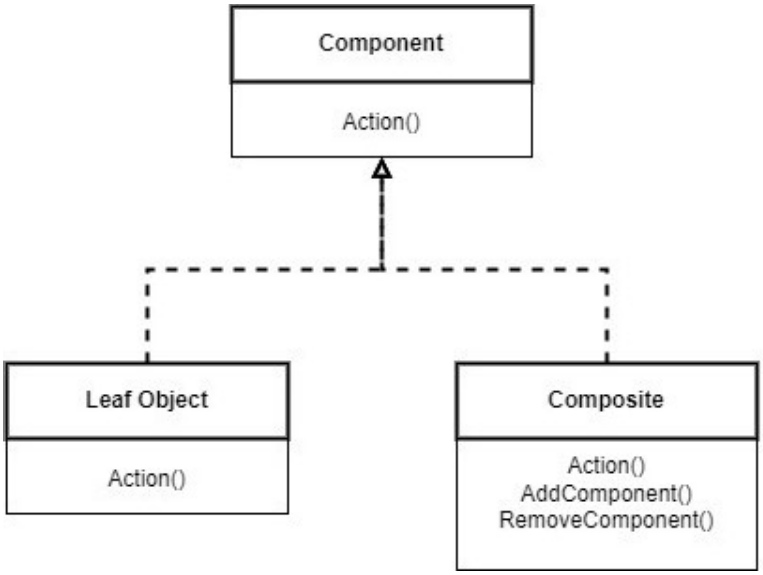

Figure 31.1 - Composite UML diagram

So, if we used the file system as an example, a folder would be a **Composite** object, while a file would be a **Leaf** object.

We will now go through an example implementation of Composite. The complete solution can be found via the link below:

https://github.com/fiodarsazanavets/design-patterns-in-csharp/tree/main/Structural_Patterns/Composite

Prerequisites

In order to be able to implement the code samples below, you need the following installed on your machine:

- .NET 8 SDK (or newer)
- A suitable IDE or a code editor (Visual Studio, Visual Studio Code, JetBrains Rider)

Composite implementation example

Because a file system is a good representation of a tree-like hierarchy that Composite design pattern can build, we will use an abstract representation of a file system in our code.

We will create a .NET console application and add the `IComponent.cs` file to it, which will contain the following interface:

```
namespace Composite_Demo;

internal interface IComponent
{
    string Name { get; }
    void Display(string currentPath);
}
```

Basically, the interface contains the common functionality for both a file and a folder. Each of those has a name. And each of those can be displayed.

Then we will add the `File.cs` file with the following class definition:

```
namespace Composite_Demo;

internal class File(string name) :
    IComponent
{
    public string Name { get; } = name;

    public void Display(
        string currentPath)
    {
```

```
        Console.WriteLine(
            currentPath + Name);
    }
}
```

Not much here. A file has a name. It can display its own name by appending it at the end of the path that it receives.

Now, we will add a representation of a folder. It will go into the `Folder.cs` file, which will have the following content:

```
namespace Composite_Demo;

internal class Folder(string name) :
    IComponent
{
    private readonly List<IComponent> children = [];

    public string Name { get; } = name;

    public void Display(string currentPath)
    {
        Console.WriteLine(currentPath + Name +
            Path.DirectorySeparatorChar);
    }

    public void Add(IComponent child)
    {
        children.Add(child);
    }

    public void Remove(string name)
    {
        var childToRemove = children
            .FirstOrDefault(c => c.Name == name);

```

```
26          if (childToRemove is not null)
27              children.Remove(childToRemove);
28      }
29
30      public void DisplayChildren(string path)
31      {
32          foreach (var item in children)
33          {
34              item.Display(path + Name +
35                  Path.DirectorySeparatorChar);
36
37              if (item is Folder folder)
38              {
39                  folder.DisplayChildren(path + Name +
40                      Path.DirectorySeparatorChar);
41              }
42          }
43      }
44  }
```

So, this is our **Composite** object. It can display its name. But it also contains functionality for adding, removing and displaying children.

Let's now see this implementation of the Composite design pattern in action. For this, we will replace the content of the `Program.cs` file with the following:

```
using Composite_Demo;

var rootFolder =
    new Folder("Root");
rootFolder.Add(
    new Folder("Folder1"));
rootFolder.Add(
    new Folder("Folder2"));

var complexFolder =
    new Folder("Folder3");
complexFolder.Add(
    new Folder("Folder4"));
complexFolder.Add(
    new Composite_Demo.File("NewFile.xml"));
complexFolder.Add(
    new Composite_Demo.File("NewFile2.xml"));

rootFolder.Add(complexFolder);

rootFolder.DisplayChildren(string.Empty);

Console.ReadKey();
```

This logic creates and displays a bunch of files and folders. The output of it would look like this:

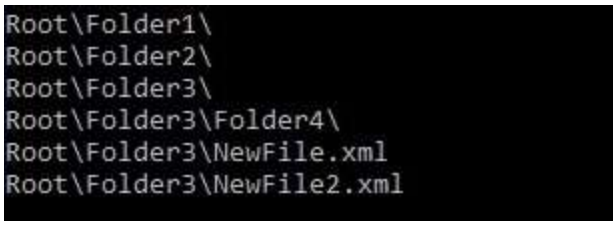

Figure 31.2 - Displaying hierarchy generated by Composite design pattern

Let's now summarize the main benefits of using the Composite

design pattern.

Benefits of using Composite

The main benefits of using the Composite can be summarized as follows:

- Composite design pattern is by far the best way of working with complex tree-like structures.
- The design pattern helps to enforce the open-closed principle. You can add new object types to the hierarchy without modifying any existing ones.

But, just like any other design pattern, Composite comes with some caveats.

Caveats of using Composite

There aren't really many disadvantages to using the Composite design pattern. Perhaps the only disadvantage is that, when you have to deal with many object types, it may become harder to maintain all the interfaces.

As you have seen in the above example, **Leaf** and **Composite** objects are different. Even though they share a common **Component** interface, a **Composite** will always have some functionality that a **Leaf** cannot have, while a **Leaf** may also have some functionality that a **Composite** cannot have. But you may also have many different types of **Leaf** and **Composite** objects.

This is just something to be aware of. Other than that, there is no better way to build complex hierarchies than the Composite design pattern.