Combined Application of Flyweight and Composite Patterns in Systems with Many Objects

Nagib Sabbag Filho

Leaders.Tec.Br, 2(22), ISSN: 2966-263X, 2025.

e-mail: profnagib.filho@fiap.com.br

DOI: https://doi.org/10.5281/zenodo.15585065

PermaLink: https://leaders.tec.br/article/f38d34

Received: 29 May 2025 / Accepted: 31 May 2025 / Published online: 02 Jun 2025

Abstract:

This article explores the combined application of the Flyweight and Composite design patterns in software development, highlighting their importance in optimizing performance and memory management in complex systems. The Flyweight pattern minimizes memory usage by sharing objects with common states, while the Composite pattern facilitates the manipulation of hierarchical structures.

Key words:

Flyweight,Composite,design patterns,systems,many objects,memory optimization,hierarchical structure,object reuse,efficiency,abstraction,object management,performance,scalable software,structural patterns,software design,system architecture.

Introduction to Design Patterns

In software development, efficiency and resource management are crucial, especially when dealing with systems that present a large number of objects. In this context, choosing appropriate design patterns can make a significant difference in the performance and maintainability of the system. Among the structural patterns, which focus on how classes and objects are composed, the Flyweight and Composite patterns stand out, both offering effective solutions to these challenges (SABBAG FILHO, 2024). The Flyweight pattern is used to minimize memory usage by sharing objects, while the Composite pattern allows for treating individual objects and compositions of objects uniformly.

This article explores in detail the combined application of these two patterns, providing practical examples and explanations on how they can be used to optimize the performance of complex systems. The integration of the Flyweight and Composite patterns is especially beneficial in scenarios where the creation of many objects is necessary, such as in games, graphic applications, and data management systems.

The Flyweight Pattern

The Flyweight pattern is a structural pattern that aims to optimize memory usage by sharing objects that have common state, reducing the overhead of redundant instances (REFACTORING GURU, 2025). It is particularly useful in systems where many objects need to be created and maintained, especially where many of these objects share common characteristics. The Flyweight allows you to create a large number of object instances, saving memory by sharing the data that remains constant (SARCAR, 2022).

A classic example of the Flyweight pattern is the representation of characters in a text editor. Instead of creating a new instance for each letter, we can have shared instances for each letter type. This approach not only saves memory but also improves performance by minimizing the number of objects on the heap.

```
using System;
using System.Collections.Generic;
// Flyweight
public interface ICharacter
{
    void Display(int x, int y);
}
public class Character : ICharacter
{
    private readonly char _letter;
    public Character(char letter)
    {
        _letter = letter;
    }
    public void Display(int x, int y)
    {
        Console.WriteLine($"Displaying letter '{_letter}' at position ({x}, {y})");
    }
}
public class CharacterFactory
{
    private readonly Dictionary<char, ICharacter> _characters = new Dictionary<char, ICharacter>();
    public ICharacter GetCharacter(char letter)
    {
        if (!_characters.ContainsKey(letter))
        {
            _characters[letter] = new Character(letter);
        }
        return _characters[letter];
    }
}
class Program
{
    static void Main()
    {
        var factory = new CharacterFactory();
        var text = "Hello, World!";

        for (int i = 0; i < text.Length; i++)
        {
            var character = factory.GetCharacter(text[i]);
            character.Display(i * 10, 0);
        }
    }
}
```

The Composite Pattern

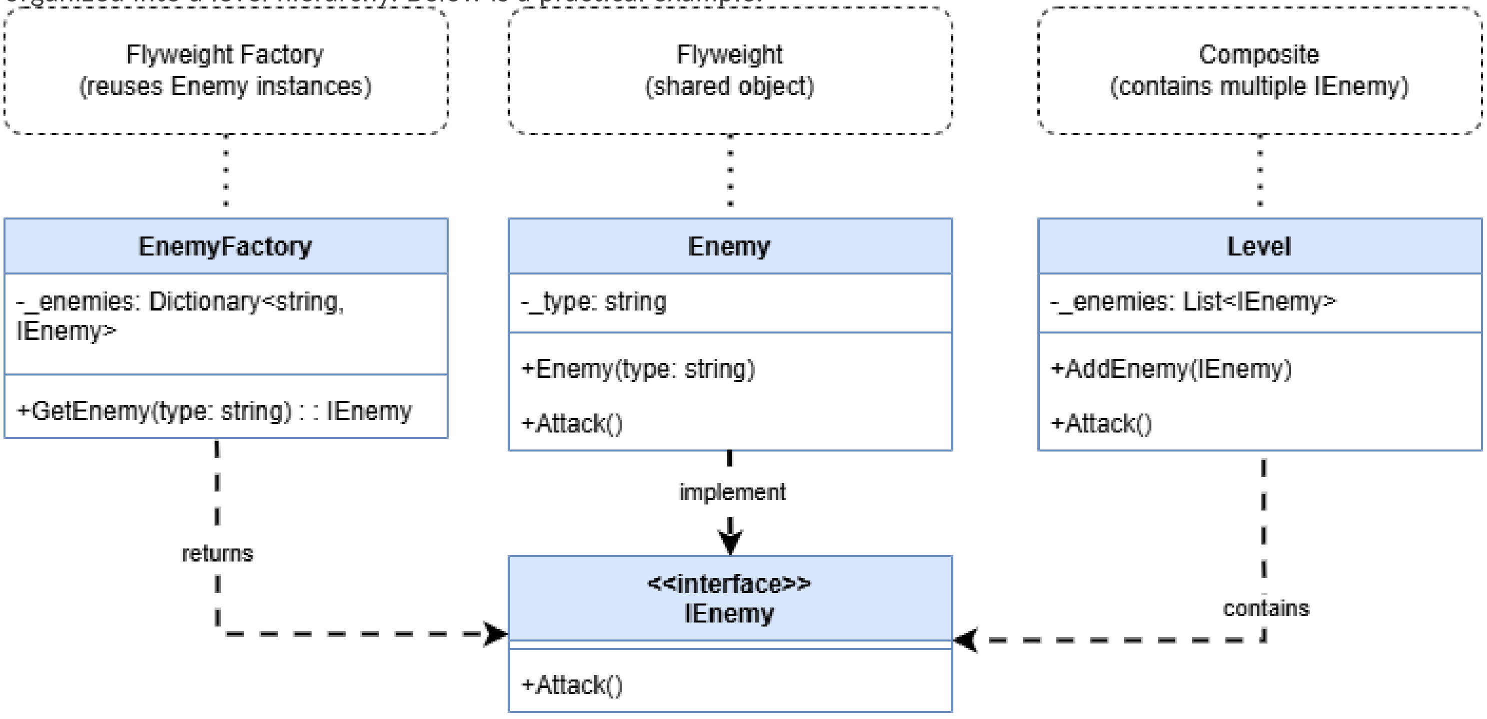
The Composite pattern is a structural pattern that allows composing objects into tree structures to represent part-whole hierarchies. It enables clients to treat individual objects and compositions of objects uniformly, simplifying the manipulation of complex structures (MANCHANA, 2019). This uniformity is especially useful in systems where it is necessary to perform operations on groups of objects that have a hierarchical relationship.

A practical example of the Composite pattern can be found in file management systems, where directories can contain files and other directories. This allows the structure to be treated as a single object, simplifying operations such as displaying, removing, or adding elements.

```
using System;
using System.Collections.Generic;
// Component
public abstract class Component
{
    public abstract void Display(int depth);
}
// Leaf
public class File : Component
{
    private readonly string _name;
    public File(string name)
    {
        _name = name;
    }
    public override void Display(int depth)
    {
        Console.WriteLine(new string('-', depth) + _name);
    }
}
// Composite
public class Directory : Component
{
    private readonly string _name;
    private readonly List<Component> _components = new List<Component>();
    public Directory(string name)
    {
        _name = name;
    }
    public void Add(Component component)
    {
        _components.Add(component);
    }
    public override void Display(int depth)
    {
        Console.WriteLine(new string('-', depth) + _name);
        foreach (var component in _components)
        {
            component.Display(depth + 2);
        }
    }
}
class Program
{
    static void Main()
    {
        var root = new Directory("Root");
        var folder1 = new Directory("Folder1");
        var folder2 = new Directory("Folder2");
        root.Add(folder1);
        root.Add(folder2);
        folder1.Add(new File("File1.txt"));
        folder1.Add(new File("File2.txt"));
        folder2.Add(new File("File3.txt"));
        root.Display(1);
    }
}
```

Integration of the Flyweight and Composite Patterns

The combination of the Flyweight and Composite patterns offers a powerful solution for systems that deal with large quantities of objects that have both a shared state and a hierarchical structure. A practical example can be found in games, where multiple objects of the same type (such as enemies or items) must be managed in a scene but also organized into a level hierarchy. Below is a practical example:



By integrating these patterns, we can not only reduce memory usage by sharing instances of common objects but also organize these objects into a hierarchical structure that facilitates manipulation and interaction among them. This approach is especially useful in scenarios where performance is critical, such as in real-time games or complex graphical applications. Below is an example of its application:

```
using System;
using System.Collections.Generic;
public interface IEnemy
{
    void Attack();
}
public class Enemy : IEnemy
{
    private readonly string _type;
    public Enemy(string type)
    {
        _type = type;
    }
    public void Attack()
    {
        Console.WriteLine($"Enemy of type {_type} attacking!");
    }
}
public class EnemyFactory
{
    private readonly Dictionary<string, IEnemy> _enemies = new Dictionary<string, IEnemy>();
    public IEnemy GetEnemy(string type)
    {
        if (!_enemies.ContainsKey(type))
        {
            _enemies[type] = new Enemy(type);
        }
        return _enemies[type];
    }
}
public class Level
{
    private readonly List<IEnemy> _enemies = new List<IEnemy>();
    public void AddEnemy(IEnemy enemy)
    {
        _enemies.Add(enemy);
    }
    public void Attack()
    {
        foreach (var enemy in _enemies)
        {
            enemy.Attack();
        }
    }
}
class Program
{
    static void Main()
    {
        var factory = new EnemyFactory();
        var level1 = new Level();
        level1.AddEnemy(factory.GetEnemy("Orc"));
        level1.AddEnemy(factory.GetEnemy("Orc"));
        level1.AddEnemy(factory.GetEnemy("Goblin"));
        level1.Attack();
    }
}
```

Advantages of the Combined Approach

The application of the Flyweight and Composite patterns together results in an architecture that significantly reduces memory usage while maintaining a clear and easily modifiable structure. This approach is especially advantageous in gaming scenarios, simulations, and complex graphics systems, where performance and organization are essential. Additionally, by allowing the creation of a hierarchy of objects, it facilitates the extension and customization of the system, enabling new types of objects to be added without altering existing behavior.

Another important point is code maintenance. The use of these patterns can lead to a cleaner design, where responsibilities are well defined, making it easier to identify problems and make changes. This is crucial in long-term projects, where the adaptation and evolution of software are inevitable.

Final Considerations

The combined use of the Flyweight and Composite patterns is an effective strategy for dealing with systems that have many objects. By allowing the sharing of common states and the creation of object hierarchies, these patterns provide a robust and efficient solution to performance and memory management challenges. Understanding and correctly applying these patterns can lead to cleaner, more efficient, and scalable software design. In a world where the complexity of software systems continues to grow, adopting effective design practices like these becomes increasingly important.

References

• SABBAG FILHO, Nagib. Comparative Analysis of Patterns: Distinctions and Applications of Behavioral, Creational, and Structural Patterns. Leaders Tec, vol. 1, no. 11, 2024.

• SARCAR, Vaskaran. Flyweight Pattern. In: Java Design Patterns: A Practical Experience with Real-World Examples. Berkeley, CA: Apress, 2022. p. 263-282.

• REFACTORING GURU. Flyweight. Available at: https://refactoring.guru/design-patterns/flyweight. Accessed on: May 28, 2025.

• MANCHANA, Ramakrishna. Structural Design Patterns: Composing Efficient and Scalable Software Architectures. International Journal of Scientific Research and Engineering Trends, vol. 5, pp. 1483-1491, 2019.

Nagib is a University Professor and Lead Systems Architect, with a career marked by several achievements in technical and agile certifications, including GitHub Copilot and PSM1. With two Lato Sensu postgraduate degrees (SENAC and Mackenzie) and an MBA in Software Technology from USP, Nagib has also participated in extension programs at MIT and University of Chicago. Among other achievements, he is the author of a peer-reviewed article on chatbots, presented in person at the University of Barcelona. Nagib also has a strong presence in the technical community, with over 40 technical articles published in less than a year, including 8 articles on iMasters. He was a speaker in the "Architecture .NET" track at TDC São Paulo 2024 and is confirmed to speak in the "Solution Architecture" track at TDC Floripa 2025.