34. Flyweight

Flyweight design pattern makes it possible to cram large quantities of objects into the memory without using too much memory. This is made possible by sharing some part of the state between the objects.

For example, imagine a real-time strategy game where you can build huge armies. Each unit is a separate object. But units can have certain states that make them behave differently. For example, a shielded unit behaves differently from a unit without a shield. So, instead of giving the full set of possible behaviors to each unit, why not simply create two instances of an object representing the behavior - shielded and unshielded, and then simply link each unit to one of those instances?

Flyweight can be summarized as follows:

- **Flyweight** object represents a set of some properties.
- **Flyweight Factory** allows you to select a specific instance of the **Flyweight** object.
- Objects that are intended to be multiplied can access a specific instance of the **Flyweight** via the **Flyweight Factory**.

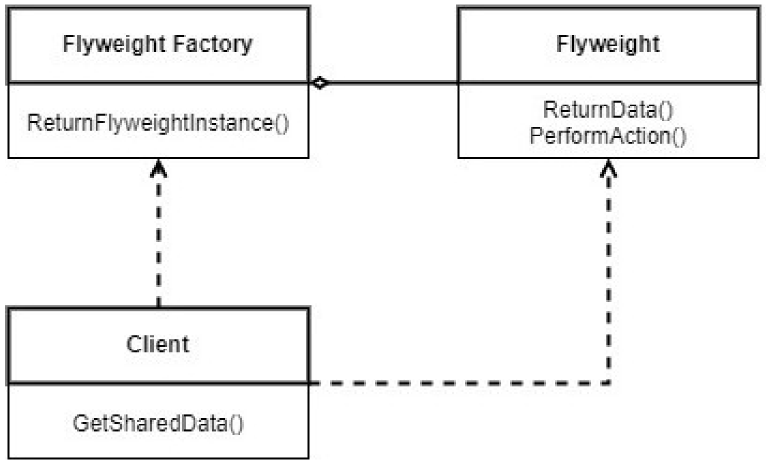


Figure 34.1 - Flyweight UML diagram

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

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

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)

Flyweight implementation example

Let's imagine that we are building a strategy video game that would allow the player to build large armies. This army would consist of different types of soldiers. But all these soldier types would have some common properties. Only that the values of these properties would be different for different soldier types.

In our example, we will use the .NET console application project template. And then we will add the `SoldierFlyweight` class to it that will represent a set of shared characteristics for our soldiers. The definition of this class will be as follows:

```
namespace Flyweight_Demo;

internal class SoldierFlyweight(
    string soldierType,
    int experienceLevel,
    int speed,
    int strength)
{
    public string SoldierType { get; set; }
        = soldierType;
    public int ExperienceLevel { get; set; }
        = experienceLevel;
    public int Speed { get; set; }
        = speed;
    public int Strength { get; set; }
        = strength;

    public void Eliminate(int soldierId)
    {
        Console.WriteLine(
            $"{SoldierType} {soldierId} has been eliminat\
ed.");
    }
}
```

The class has a range of properties. And it also comes with the `Eliminate` method, which triggers some behavior when a specific soldier gets eliminated.

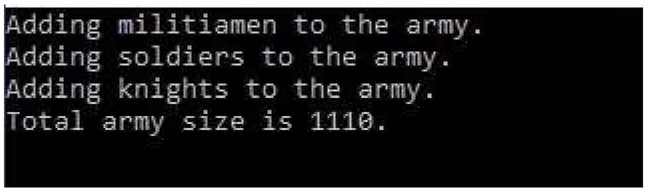
Then, we will add the **Flyweight Factory** object that will allow us to select a specific instance of the `SoldierFlyweight` class. For this, we will have `FlyweightFactory` class, which will have the following code:

```
namespace Flyweight_Demo;

internal class FlyweightFactory
{
    private readonly
        Dictionary<string, SoldierFlyweight> flyweights;

    public FlyweightFactory()
    {
        flyweights = new()
        {
            ["Militiaman"] =
                new SoldierFlyweight(
                    "Militiaman", 1, 1, 1),
            ["Soldier"] =
                new SoldierFlyweight(
                    "Soldier", 2, 1, 2),
            ["Knight"] =
                new SoldierFlyweight(
                    "Soldier", 10, 5, 5)
        };
    }

    public SoldierFlyweight GetFlyweight(string key)
    {
        return flyweights[key];
    }
}
```

This allows us to define a number of specific soldier types. Now, we will create an army of soldiers. To do so, we will replace the content of the `Program.cs` file with the following:

```
using Flyweight_Demo;

var factory =
    new FlyweightFactory();
var army =
    new Dictionary<int, SoldierFlyweight>();

Console.WriteLine(
    "Adding militiamen to the army.");

for (var i = 0; i < 1000; i++)
{
    army[i] = factory
        .GetFlyweight("Militiaman");
}

Console.WriteLine(
    "Adding soldiers to the army.");

for (var i = 0; i < 100; i++)
{
    army[i + 1000] = factory
        .GetFlyweight("Soldier");
}

Console.WriteLine(
    "Adding knights to the army.");

for (var i = 0; i < 10; i++)
{
    army[i + 1100] = factory
        .GetFlyweight("Knight");
}

Console.WriteLine(
    $"Total army size is {army.Count}.");

Console.ReadKey();
```

The output of this program would be as follows:



```
Adding militiamen to the army.
Adding soldiers to the army.
Adding knights to the army.
Total army size is 1110.
```

Figure 34.2 - Using flyweight objects to build an army

This code demonstrates the obvious benefits of using the Flyweight design pattern. Even though we are storing many objects representing soldiers in the memory, we are only reserving a small amount of memory for each soldier. This is because each of the items representing a soldier refers to one of only three instances of a **Flyweight** object that represents a bulk of its functionality. And because we have a parametrized `Eliminate` method on our **Flyweight** object, we can still perform some behavior that is specific to an individual soldier.

Benefits of using Flyweight

So, the core benefits of using the Flyweight design pattern are as follows:

- It allows you to use memory very efficiently by sharing large chunks of the state between objects.

- It allows you to still perform functionality that is specific to a particular item instance.

But to use Flyweight effectively, one needs to be aware of its caveats. And this is what we will have a look at next.

Caveats of using Flyweight

Perhaps the main caveat of using Flyweight is that it can make the code complicated and difficult to read. Therefore sometimes there needs to be a trade-off between code maintainability and the efficient use of RAM.

Another disadvantage of using flyweight is that, although it will save your memory, it might actually increase CPU usage. Imagine many objects making a parametrized call to a specific instance of a **Flyweight** object. Even if this instance is shared between many objects, your CPU would still be doing a lot of work in this situation.