35. Proxy

Proxy is a structural pattern that allows us to provide a substitute for an object. The substitute object, known as Proxy, can perform some action before or after the original object is accessed. For example, we can cache the results of a computationally expensive operation so they can just be retrieved quickly from the cache and the original operation doesn't have to run again. Or, if our application is connected to a network and the connectivity is lost, the Proxy object can provide substitute functionality that can work offline.

It's analogous to how the cache in a browser works. When you visit a web page for the first time, the browser will retrieve it from the server. But it will then store a copy of this page in its own cache. So when you visit the page next time, it will be retrieved from the local cache. In this situation, the browser is acting as a proxy for the server.

Proxy can be summarized as follows:

- There is a **Main Service** object that is computationally expensive to run.
- There is a **Proxy** object that implements exactly the same interface as the **Main Service**.
- All requests to the **Main Service** are performed via the **Proxy**
- **Proxy** only runs the expensive operation on the **Main Service** when it needs to and then it caches the results.
- Under any other circumstances, **Proxy** returns the results from its cache without contacting the **Main Service**.

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

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

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)

Proxy implementation example

Let's imagine that we have some back-end service that we can retrieve data from and submit new data to. But retrieving the data happens to be a long and computationally expensive process. And this is what we will simulate to demonstrate how the Proxy design pattern can be applied to solve such a problem.

We will create a new .NET console application and we will add `IDataService` interface to it, which will look as follows:

```
namespace Proxy_Demo;

internal interface IDataService
{
    Task<List<string>> GetData();
    void InsertData(string item);
}
```

Then we will add an implementation of this interface that represents our **Main Service**. It will be represented by `DataService` class with the following content:

```
namespace Proxy_Demo;

internal class DataService : IDataService
{
    private readonly List<string> data;

    public DataService()
    {
        data = [];
    }

    public async Task<List<string>> GetData()
    {
        // Simulate long-running process
        await Task.Delay(3000);
        return data;
    }

    public void InsertData(string item)
    {
        data.Add(item);
    }
}
```

So, we are simulating a long-running data retrieval process by adding a three second delay to the `GetData` method.

Next, we will add a **Proxy** implementation of the same interface. The class will be called `DataServiceProxy` and it will look as follows:

```
namespace Proxy_Demo;

internal class DataServiceProxy : IDataService
{
    private readonly DataService dataService;
    private List<string>? localCache;

    public DataServiceProxy()
    {
        dataService = new DataService();
        localCache = null;
    }

    public async Task<List<string>> GetData()
    {
        Console.WriteLine(
            $"{DateTime.Now} - Started data query.");

        if (localCache is null)
            localCache =
                await dataService.GetData();

        Console.WriteLine(
            $"{DateTime.Now} - Data has been retrieved.");
        return localCache;
    }

    public void InsertData(string item)
    {
        localCache = null;
        dataService.InsertData(item);
    }
}
```

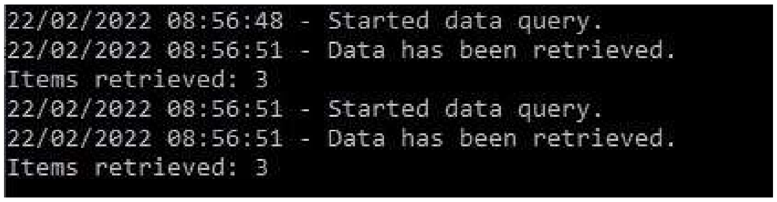
This proxy acts as a middleware between the actual service implementation and the caller. It maintains its own cache, which is instantiated as `null` and gets reset to `null` every time new data is inserted via the **Main Service**. But once it has retrieved the **Main Service**, the proxy populates its own cache with it. And while the cache has any items in it, any subsequent call to `GetData` method will use the cache and won't call the **Main Service**.

To make the benefits of using Proxy design pattern obvious, we can replace the content of the `Program.cs` file with the following:

```
using Proxy_Demo;

var dataService = new DataServiceProxy();
dataService.InsertData("item 1");
dataService.InsertData("item 2");
dataService.InsertData("item 3");

var data = await dataService.GetData();
Console.WriteLine(
    $"Items retrieved: {data.Count}");
data = await dataService.GetData();
Console.WriteLine(
    $"Items retrieved: {data.Count}");

Console.ReadKey();
```

In here, where are initially inserting three items via the **Proxy**. Then, we retrieve this data twice, comparing the time it took for the data to be retrieved.

As the following screenshot demonstrates, the data took three seconds to arrive on the first run. But its retrieval was almost instant on the second run. And this is because, on the first run, the **Proxy** ran an expensive operation on the **Main Service**, while on the second run it used its own cache.



```
22/02/2022 08:56:48 - Started data query.
22/02/2022 08:56:51 - Data has been retrieved.
Items retrieved: 3
22/02/2022 08:56:51 - Started data query.
22/02/2022 08:56:51 - Data has been retrieved.
Items retrieved: 3
```

Figure 35.1 - Performance improvements by using Proxy

And this concludes the overview of Proxy design pattern. Let's summarize its main benefits.

Benefits of using Proxy

The benefits of using Proxy can be summarized as follows:

- This design pattern allows you not to run expensive operations every time you need to obtain the results of these operations.
- Because the **Proxy** object uses the same interface as the **Main Service**, **Proxy** object can be used in any place in the code where the **Main Service** is normally used.

But, as with any design patterns, there are still some caveats to be aware of while making a decision on whether or not to use Proxy.

Caveats of using Proxy

While using Proxy, you have to strike a good balance between performance and accuracy. You need to put some logic in place to make sure that your cache is not refreshed too often. But at the same time, you need to ensure the cache is reasonably accurate. If the cache refreshes too often, you may lose the benefits of using Proxy. If it doesn't refresh often enough, it may become outdated quickly. So, when making design decisions, you need to consider this trade-off.

Other than that, using the Proxy design pattern will make your code more complicated. And if you are dealing with complex logic, this may make your code harder to read and maintain. So the tradeoff between the performance and readability might be another point to consider while using Proxy.