30. Bridge

Bridge design pattern allows you to separate the business logic from a software component that controls or triggers this business logic. For example, you may have a user interface that has multiple buttons. Each of these buttons triggers some logic in the back end. The Bridge design pattern allows you to separate the user face with the buttons from the component that contains the actual business logic.

The main benefit of using the Bridge design pattern would be that it allows two separate teams to work independently on two application components. One team would be responsible for the user interface and the other team would be responsible for the implementation of the business logic in the back-end. The teams can work independently as long as they have specified a shared interface definition.

Bridge can be summarized as follows:

- There are two objects: **Interface** and **Implementation**. They are not to be confused with interfaces and classes. In this context, the object playing the role of **Interface** is actually a concrete class.
- **Implementation** object contains the main business logic, while **Interface** object is design to interact with all endpoints of **Implementation** object.
- **Implementation** object depends on an interface (in the normal sense of object-oriented programming), so it can be easily mocked when **Interface** object needs to be tested.

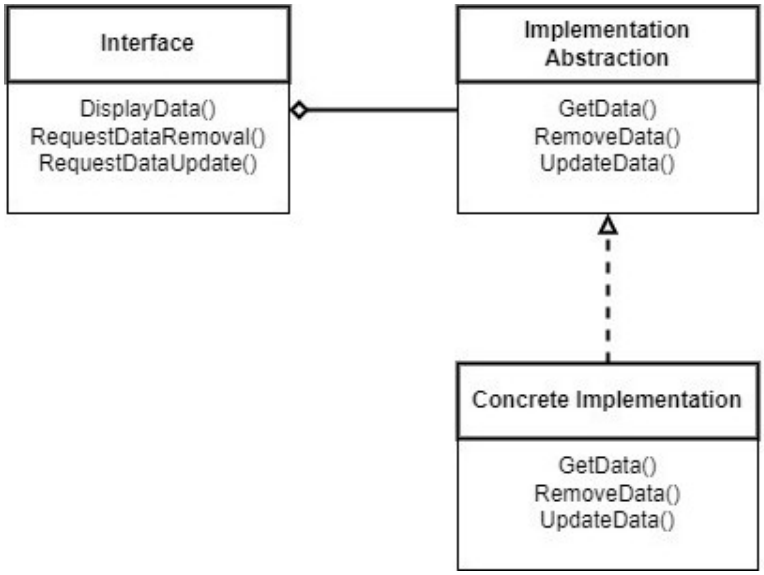

Figure 30.1 - Bridge UML diagram

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

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

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)

Bridge implementation example

Bridge works best if we use the **Implementation** and the **Interface** objects in separate libraries. This allows separate teams to work on separate components without affecting each other's code base. And this is what we will do in our example.

For our **Implementation** object, we will create a .NET class library and we will call it `BridgeImplementation`. We will then add the following `IDataService.cs` file to the project folder of the class library. The file will have the following content:

```
namespace BridgeImplementation;

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

Let's imagine that there's some service that returns some data from a database. This interface represents the access points of such a service. We can retrieve the data by calling the `GetData` method and we can insert new items by calling the `InsertData` method.

For demonstration purposes, we won't be calling any database queries. We would merely work with an in-memory collection. So the representation of our **Implementation** object will be as follows:

```
namespace BridgeImplementation;

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

    public DataService()
    {
        data = [];
    }

    public List<string> GetData()
    {
        return data;
    }

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

And now we will define our **Interface** object. To do so, we will create a .NET console application and a reference to our `BridgeImplementation` from it. Our **Interface** object will be represented by the `BridgeInterface.cs` file, which will have the following content:

```
global using BridgeImplementation;

namespace Bridge_Demo;

internal class BridgeInterface
{
    public IDataService? Implementation { get; set; }

    public void GetData()
    {
        if (Implementation is null)
        {
            Console.WriteLine("No data.");
            return;
        }

        foreach (var item in Implementation.GetData())
        {
            Console.WriteLine(item);
        }
    }

    public void InsertData(string item)
    {
        Implementation?.InsertData(item);
    }
}
```

So, for each of the public methods on the **Implementation** object, the **Interface** object has a method of its own. The role of each of these methods inside the **Interface** object is to control the **Implementation** and display the data that was returned from the **Implementation**.

In our case, we are outputting data into the console. But you can imagine how a similar principle can be used to output the data into

a graphical user interface.

Please note that our **Interface** object depends on an interface that the **Implementation** object implements. It won't be a good idea to use the concrete implementation. Firstly, it would violate the dependency inversion principle. Secondly, using an interface would allow you to assign both the concrete implementation of it and a mock implementation that you can use for testing.

Finally, we will replace the content of our `Program.cs` class with the following:

```
using Bridge_Demo;

var bridgeInterface =
   new BridgeInterface();
bridgeInterface.Implementation =
    new DataService();

Console.WriteLine(
    "Inserting item 1 into data service");
bridgeInterface.InsertData("item 1");
Console.WriteLine(
    "Inserting item 2 into data service");
bridgeInterface.InsertData("item 2");
Console.WriteLine(
    "Inserting item 3 into data service");
bridgeInterface.InsertData("item 3");

Console.WriteLine(
    "Retrieving data from the service:");
bridgeInterface.GetData();

Console.ReadLine();
```

In here, we are inserting some data into the data service and then

retrieving it via the **Interface** object. And, as we can see from the following output, we get the data displayed in the console:

```
Inserting item 1 into data service
Inserting item 2 into data service
Inserting item 3 into data service
Retrieving data from the service:
item 1
item 2
item 3
```

Figure 30.2 - Interface displays data that was retrieved from Implementation

This concludes the overview of the Bridge design pattern. Let's now summarize its benefits.

Benefits of using Bridge

The benefits of using the Bridge design pattern can be summarized as follows:

- The main benefit of separating **Interface** from **Implementation** is that separate teams can work on these two components without interfering with each other.
- Because of such a structure, it enforces the single responsibility principle.
- **Interface** can easily be tested even when **Implementation** is not ready. Due to the dependency inversion principle, **Implementation** can be easily mocked.

Let's now have a look at the caveats of using Bridge.

Caveats of using Bridge

Although Bridge is a very useful design pattern, the classic version of it is rarely used these days. But it's not because of its ineffectiveness. It's because Bridge has inspired the creation of some other design patterns, such as Model-View-Controller (MVC) and Model-View-ViewModel (MVVM). These design patterns are very similar to Bridge, but are more context-specific. Therefore it makes them more useful in many situations than classic Bridge.

The principles of Bridge design patterns have also inspired architectural best practices of distributed software. For example, gRPC is, pretty much, a distributed implementation of the Bridge design pattern. Protobuf definitions that clients and servers share can be thought of as the shared Bridge interface. In this context, a gRPC client can be thought of as an **Interface** object, and a server-side gRPC service can be considered to be an **Implementation**.

So, even if you don't intend to use Bridge in its classic form, it's still worth being familiar with it to understand the good practices of modern software architecture.