33. Facade

Facade design pattern is all about building a simple interface to abstract away the complexity of using multiple complex interfaces. If, for example, your application needs to access multiple services and then perform some complex logic of coordinating the responses from those services, the Facade design pattern would allow you to isolate such functionality in one place. This will then be accessible by the rest of the application via a simple interface.

A real-life analogy could be a food delivery system. All you do is go to a website and select the items to order. And then those items will get delivered to you. As a consumer, this is all you care about. But behind the scenes, once you place an order, you trigger a whole complex system. The food needs to be prepared. The preparation of different items needs to be coordinated. The driver needs to be assigned. The delivery route needs to be planned. And so on.

Facade can be summarized as follows:

- There are several **Endpoint Interfaces**.
- Either the calls to the **Endpoint Interfaces** need to be coordinated or the responses from them need to be aggregated in some way.
- There is a **Facade** object that performs all of such coordination.
- The **Facade** object is accessible by the rest of the application via a simple interface.

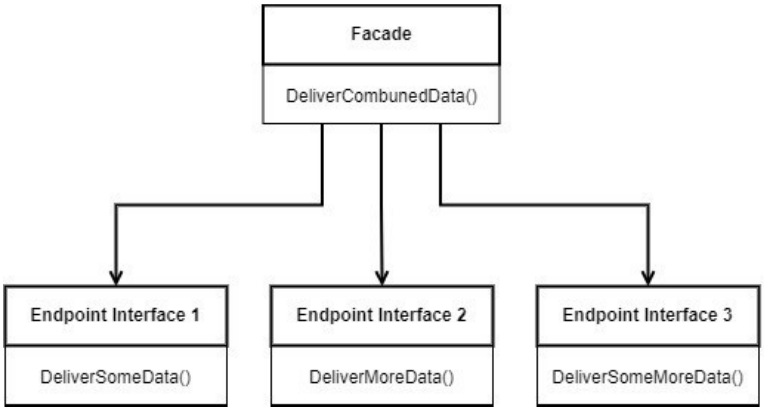

Figure 33.1 - Facade UML diagram

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

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

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)

Facade implementation example

Imagine that we need to build an application where, if we input the account name, we would be able to retrieve the list of products this

account is allowed to buy. But to achieve this, we will have to deal with multiple back-end systems. A list of products will be returned based on which category the account belongs to, and whether or not the account is a buyer or a reseller. But our application doesn't hold this information. There is another service that will return the account category based on the account id. But our application doesn't store the account id either. It operates within its own bounded context where account id doesn't exist. Only the name of the account is known. So we need to call another service to obtain the account id based on the account name.

As you can see, it's a fairly complex interaction between multiple services. And this is where the Facade design pattern would fit perfectly.

We will start by creating a .NET console application. Then we will add a service that will return a list of accounts. This service would be represented by the `AccountDataService` class and its definition would be as follows:

```
namespace Facade_Demo;

internal class AccountDataService
{
    private readonly List<Account> accounts;

    public AccountDataService()
    {
        accounts =
        [
            new Account(1, "John Smith"),
            new Account(2, "Jane Doe"),
            new Account(3, "Laurence Newport"),
            new Account(4, "David Fisher"),
        ];
    }
```

```
17
18      public List<Account> GetAccounts()
19      {
20          return accounts;
21      }
22  }
23
24  internal record Account
25  {
26      public Account(int id, string name)
27      {
28          Id = id;
29          Name = name;
30      }
31
32      public int Id { get; }
33      public string Name { get; }
34  }
```

So, in here, we are returning a list of `Account` objects that contain `Id` and `Name` properties. For demonstration purposes, we are hard-coding the data. But in a real-life scenario, the data would be retrieved from a data storage of some sort.

Next, we will add a service that will allow us to retrieve the account category based on the account id. The service will be called `AccountCategoryService` and it will look as follows:

```
namespace Facade_Demo;

internal class AccountCategoryService
{
    private Dictionary<int, AccountCategory> accountCateg\
ories;

    public AccountCategoryService()
    {
        accountCategories = new();
        accountCategories.Add(
            1, AccountCategory.Buyer);
        accountCategories.Add(
            2, AccountCategory.Buyer);
        accountCategories.Add(
            3, AccountCategory.Reseller);
        accountCategories.Add(
            4, AccountCategory.Reseller);
    }

    public AccountCategory GetCategory(int accountId)
    {
        return accountCategories[accountId];
    }
}

internal enum AccountCategory
{
    Buyer = 1,
    Reseller = 2
}
```

After this, we will add a service that will give us a list of products based on the account category. We will call it `ProductsDataService` and we will define it as this:

```
namespace Facade_Demo;

internal class ProductsDataService
{
    private readonly
        Dictionary<int, List<Product>> productLists;

    public ProductsDataService()
    {
        productLists = new()
        {
            [1] =
            [
                new Product("Product 1", 9.99),
                new Product("Product 2", 19.99)
            ],
            [2] =
            [
                new Product("Bundle 1", 99.99),
                new Product("Bundle 2", 199.99)
            ]
        };
    }

    public List<Product> GetProductsForCategory(
        int categoryId)
    {
        return productLists[categoryId];
    }
}

internal record Product
{
    public Product(string name, double price)
    {
```

```
        Name = name;
        Price = price;
    }

    public string Name { get; }
    public double Price { get; }
}
```

Finally, we are ready to add our **Facade** object. We will call this class `ProductsFacade`. And its definition would be as follows:

```
namespace Facade_Demo;

internal class ProductsFacade
{
    private readonly
        AccountCategoryService accountCategoryService;
    private readonly
        AccountDataService accountDataService;
    private readonly
        ProductsDataService productDataService;

    public ProductsFacade()
    {
        accountCategoryService = new();
        accountDataService = new();
        productDataService = new();
    }

    public List<Product> GetProductListForAccount(
        string name)
    {
        var accountId = accountDataService
                        .GetAccounts()
                        .Where(a => a.Name == name)
```

```
25                          .Select(a => a.Id)
26                          .FirstOrDefault();
27
28          if (accountId == default)
29              return [];
30
31          var accountCategory = accountCategoryService
32              .GetCategory(accountId);
33
34          return productDataService
35              .GetProductsForCategory((int)accountCategory);
36      }
37  }
```

As you can see, we are coordinating the work between the services we created earlier. We are first retrieving the accounts from an instance of `AccountDataService` and retrieving the account id from it that is associated with the name that has been inputted. Then, we retrieve the account category from an instance of `AccountCategoryService`. Finally, we are converting this category to an integer and we are using it to retrieve the actual product list from an instance of `ProductsDataService`. This list is then returned to the caller.

Of course, in a real-life application, we would inject interfaces of the **Endpoint Service** objects rather than concrete implementations of them into our **Facade** object. Plus the **Facade** object itself would implement an interface. This would make it consistent with the dependency inversion principle. But to make the demonstration as simple as possible, we are using concrete instances.

Now, we can see how this logic operates. And to do so, we will replace the content of the `Program.cs` file with the following:

```
using Facade_Demo;

var facade = new ProductsFacade();

foreach (var product in facade
    .GetProductListForAccount("John Smith"))
{
    Console.WriteLine(
        $"Product name: {product.Name}");
    Console.WriteLine(
        $"Product price: {product.Price}");
}

Console.ReadKey();
```

And the output of this program would look like this:

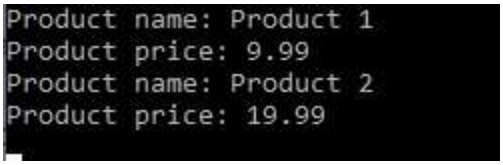

Figure 33.2 - Output returned by the Facade object

This concludes the overview of the Facade design pattern. Let's summarize its main benefits.

Benefits of using Facade

Facade design pattern has the following main benefits:

- It allows you to isolate complex coordinating logic in a single place of the application.
- It allows this complex coordinating logic to be easily reused.
- The Facade object can be easily mocked in automated tests.

And now we will have a look at some of the caveats that you need to be aware of while using Facade.

Caveats of using Facade

Perhaps the only caveat of using Facade is that it can easily become the so-called *God object* that violates the single responsibility principle. Therefore the developers need to be careful to ensure this doesn't happen. Perhaps, it would make sense to split the coordinating logic into multiple classes and get the **Facade** object to work with them.