25. Abstract Factory

Abstract Factory is closely related to the Factory Method. In fact, it is the Factory class that hosts the so-called Factory Method. But as the Factory Method refers to a specific method inside a Creator or a Factory class, Abstract Factory can have a number of such methods.

Abstract Factory, along with its Factory Methods, is used for creating concrete implementations of abstract types or interfaces. This design pattern is applied when a concrete implementation of an object needs to be chosen conditionally. However because Abstract Factory can host multiple Factory Methods, it can be used to create concrete implementations of multiple related objects.

Abstract Factory can be summarized as follows:

- There are some interfaces or abstract classes, each having multiple concrete implementations. They represent the objects that need to be created (let's call them **Target Objects**).
- There is an abstract object (or an interface), known as a **Creator**, or a **Factory** that returns abstract versions of **Target Objects**. Each of these **Target Object** types has a creation method inside the **Factory** associated with it.
- There are multiple concrete implementations of the **Factory**, each returning a specific set of the **Target Object** implementations.
- When we need to return a specific set of the **Target Object** implementations, we initialize a specific implementation of the **Factory** and call the creation methods on it.

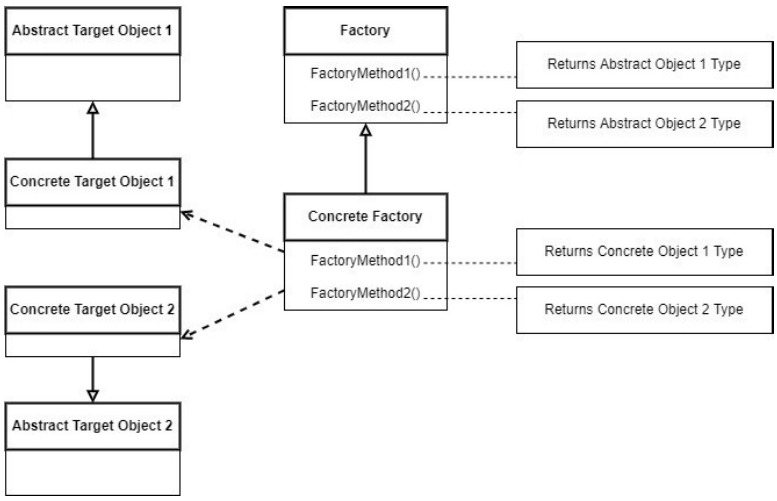

Figure 25.1 - Abstract Factory UML diagram

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

https://github.com/fiodarsazanavets/design-patterns-in-csharp/tree/main/Creational_Patterns/Abstract-Factory

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)

Abstract Factory implementation example

In this example, we will be creating an audio player application that will be able to play audios on either Windows or Linux. It is similar to what we have done when we had a look at an implementation of the Factory Method. However, there will be some crucial differences.

In the example that we have used to demonstrate the Factory Method, we have been returning a concrete OS-specific implementation of an audio player depending on which operating system the software runs on. This time, however, we will be returning concrete implementations of OS-specific individual functionalities. We will have an object that will provide an abstraction for the Play button and another object that will provide an abstraction for the Stop button.

We will start by creating a .NET console application project. Then, we will add the `LinuxPlayerUtility.cs` file to the project and populate it with the following content:

```
using System.Diagnostics;

namespace Abstract_Factory_Demo
{
    internal static class LinuxPlayerUtility
    {
        public static Process? PlaybackProcess { get; set\
; }

        public static void StartBashProcess(
            string command)
        {
            var escapedArgs =
```

```
                command.Replace("\"", "\\\"");

            var process = new Process()
            {
                StartInfo = new ProcessStartInfo
                {
                    FileName = "/bin/bash",
                    Arguments = $"-c \"{escapedArgs}\"",
                    RedirectStandardOutput = true,
                    RedirectStandardInput = true,
                    UseShellExecute = false,
                    CreateNoWindow = true,
                }
            };

            process.Start();
        }
    }
}
```

This will be a static utility class with some functionality shared by Linux-specific implementations of **Target Objects**. We will then do the same for Windows-specific implementations. To do so, we will create `WindowsPlayerUtility.cs` file and populate it with the following content:

```
using System.Runtime.InteropServices;
using System.Text;

namespace Abstract_Factory_Demo;

internal static class WindowsPlayerUtility
{
    [DllImport("winmm.dll", CharSet = CharSet.Unicode)]
    private static extern int mciSendString(
```

```
        string command,
        StringBuilder stringReturn,
        int returnLength,
        IntPtr hwndCallback);

    public static void ExecuteMciCommand(
        string commandString)
    {
        var sb = new StringBuilder();
        var result = mciSendString(
            commandString, sb, 1024 * 1024, IntPtr.Zero);
        Console.WriteLine(result);
    }
}
```

Now, we will add an abstraction of the Play button. We will create `PlayButton.cs` file and add the following content to it:

```
namespace Abstract_Factory_Demo;

internal abstract class PlayButton
{
    public abstract Task Play(string fileName);
}
```

We will do the same for an abstraction of the Stop button. The file will be called `StopButton.cs` and its content will be as follows:

```
namespace Abstract_Factory_Demo;

internal abstract class StopButton
{
    public abstract Task Stop(string fileName);
}
```

We will then start adding concrete OS-specific implementations of these abstract classes. For `PlayButton`, we will add `LinuxPlayButton.cs` file with the following content:

```
namespace Abstract_Factory_Demo;

internal class LinuxPlayButton : PlayButton
{
    public override Task Play(string fileName)
    {
        Console.WriteLine(
            "Playing audio via the following command:");
        Console.WriteLine($"mpg123 -q '{fileName}'");

        // Uncomment for testing on a real device
        // LinuxPlayerUtility.StartBashProcess($"mpg123 -\
q '{fileName}'");
        return Task.CompletedTask;
    }
}
```

And we will add `WindowsPlayButton.cs` file with the following content:

```
namespace Abstract_Factory_Demo;

internal class WindowsPlayButton : PlayButton
{
    public override Task Play(string fileName)
    {
        WindowsPlayerUtility
            .ExecuteMciCommand($"Play {fileName}");
        return Task.CompletedTask;
    }
}
```

We will then add concrete implementations for `StopButton`. Our `LinuxStopButton.cs` file will contain the following code:

```
namespace Abstract_Factory_Demo;

internal class LinuxStopButton : StopButton
{
    public override Task Stop(string fileName)
    {
        if (LinuxPlayerUtility
            .PlaybackProcess != null)
        {
            LinuxPlayerUtility
                .PlaybackProcess.Kill();
            LinuxPlayerUtility
                .PlaybackProcess.Dispose();
            LinuxPlayerUtility
                .PlaybackProcess = null;
        }
        else
        {
            Console.WriteLine(
                "No active playback process found.");
```

```
        }

        return Task.CompletedTask;
    }
}
```

And its Windows-specific implementation, `WindowsStopButton.cs` file, will contain the following:

```
namespace Abstract_Factory_Demo;

internal class WindowsStopButton : StopButton
{
    public override Task Stop(string fileName)
    {
        WindowsPlayerUtility.ExecuteMciCommand($"Stop {fi\
leName}");
        return Task.CompletedTask;
    }
}
```

Now, we will move on to our Abstract Factory. This will be represented by `PlayerCreator.cs` file, which will have the following content:

```
namespace Abstract_Factory_Demo;

internal abstract class PlayerCreator
{
    public abstract PlayButton CreatePlayButton();
    public abstract StopButton CreateStopButton();
}
```

So, as you can see, the **Factory** returns two objects rather than just one, as we had in the Factory Method example. Otherwise, it operates under the same principles.

We will now need to create OS-specific implementations of this **Creator** object. We will first add a Linux implementation. To do so, we will create `LinuxPlayerCreator.cs` file and populate it with the following content:

```
namespace Abstract_Factory_Demo;

internal class LinuxPlayerCreator : PlayerCreator
{
    public override PlayButton CreatePlayButton()
    {
        return new LinuxPlayButton();
    }

    public override StopButton CreateStopButton()
    {
        return new LinuxStopButton();
    }
}
```

Then, we will add Windows implementation by creating `WindowsPlayerCreator.cs` file and populating it with the following code:

```
namespace Abstract_Factory_Demo;

internal class WindowsPlayerCreator : PlayerCreator
{
    public override PlayButton CreatePlayButton()
    {
        return new WindowsPlayButton();
    }

    public override StopButton CreateStopButton()
    {
```

```
        return new WindowsStopButton();
    }
}
```

Now, we are ready to use our Abstract Factory. We will do so by replacing the content of `Program.cs` file with the following:

```
using Abstract_Factory_Demo;
using System.Runtime.InteropServices;

PlayerCreator? playerFactory;

if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
    playerFactory = new WindowsPlayerCreator();
else if (RuntimeInformation.IsOSPlatform(OSPlatform.Linux\
))
    playerFactory = new LinuxPlayerCreator();
else
    throw new Exception(
        "Only Linux and Windows operating systems are sup\
ported.");

Console.WriteLine(
    "Please specify the path to the file to play.");

var filePath =
    Console.ReadLine() ?? string.Empty;
await playerFactory
    .CreatePlayButton().Play(filePath);

Console.WriteLine(
    "Playing audio. Type 'stop' to stop it or 'exit' to e\
xit the application.");

while (true)
```

```
{
    var command = Console.ReadLine();

    if (command == "stop")
        await playerFactory
            .CreateStopButton().Stop(filePath);
    else if (command == "exit")
        break;
}

Console.ReadKey();
```

So, the code detects which operating system the application is running on and applies an OS-specific implementation of the **Factory** object. Then, calling any of the methods on the **Factory** object implementation will return OS-specific implementation of the functionality that the method is responsible for creating.

And this concludes our overview of Abstract Factory. Let's summarize what its benefits are.

Benefits of using Abstract Factory

The benefits of using Abstract Factory are the same as that of using Factory Method. Let's recap what those are:

- Abstract Factory enforces the single responsibility principle by separating conditional logic from implementation of the logic inside each condition.
- This allows us to easily maintain the code and write automated tests for it. There won't be any complex and barely readable tests assessing complex conditional logic. Instead, a couple of simple scenarios will asses that the correct implementations of **Target Objects** and **Factory** is picked up for

each condition. Then, separate sets of scenarios can be applied to each separate **Target Object** and **Factory** implementation.
- Abstract Factory allows you to execute a particular condition once. Because all implementations of each **Target Object** type use a common abstraction, a concrete implementation of it can be created once and then just re-used throughout the code.

In addition to these, Abstract Factory has the following major benefit:

- Abstract Factory allows you to create a whole family of related objects in one go.

And Abstract Factory doesn't have the same caveats as Factory Method has on its own. Because you are creating a family of related objects rather than just a specific implementation of a single object, **Creator** object plays an important role and actually makes things easier.

But Abstract Factory still has some minor caveats. And this is what we will examine next.

Caveats of using Abstract Factory

Because Abstract Factory was designed to create a family of related objects in one go, it's not suitable for situations where creating an object step-by-step would be more appropriate. Builder patterns are more suitable for these types of situations. And this is what we will have a look at next.