24. Factory Method

Factory Method is used for creating a concrete implementation of a particular abstract type or an interface. This design pattern is applied when a concrete implementation of an object needs to be chosen conditionally.

Factory Method can be summarized as follows:

- There is an interface or an abstract class with multiple concrete implementations. It represents an object that needs to be created (let's call it **Target Object**).
- There is an abstract object (or an interface), known as a **Creator**, or a **Factory** that returns an abstract version of **Target Object**.
- There is a concrete implementation of the **Factory** per each concrete implementation of the **Target Object**.
- When we need to return a specific implementation of the **Target Object**, we initialize a specific implementation of the **Factory** and call the creation method on it.

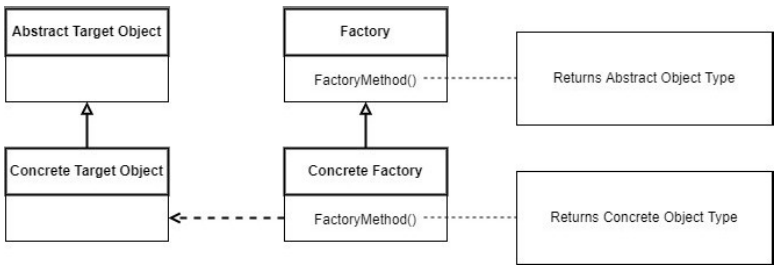

Figure 24.1 - Factory Method UML diagram

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

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

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)

Factory Method implementation example

In our example, we will be building an application capable of playing audio on either Windows or Linux platforms. However, these two operating systems have completely different APIs; therefore they require different code. But our application needs to be able to work on either of them.

To enable this, we need two distinct implementations of audio player functionality. We need a separate implementation for Windows and a separate one for Linux. This is a scenario that Factory Method is perfect for.

For demonstration purposes, we will create a console application project. Inside this project, we will add a `Player.cs` file with the following content:

```
namespace Factory_Method_Demo;

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

This will be the abstract **Target Object**.

Then, we will create an abstract **Factory** object. To do so, we will add a `PlayerCreator.cs` file with the following content:

```
namespace Factory_Method_Demo;

internal abstract class PlayerCreator
{
    public abstract Player CreatePlayer();
}
```

As you can see, we have the `CreatePlayer` method, which will return an implementation of `Player`, the abstract class we created earlier.

Let's now add some concrete implementations. We will first add a Linux-specific implementation of the `Player` class. To do so, we will create a `LinuxPlayer.cs` file and populate it with the following content:

```
using System.Diagnostics;

namespace Factory_Method_Demo;

internal class LinuxPlayer : Player
{
    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
        // StartBashProcess($"mpg123 -q '{fileName}'");

        return Task.CompletedTask;
    }

    private 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,
            }
        };
```

```
36
37          process.Start();
38      }
39  }
```

Then, we will add a Windows-specific implementation of `Player`. We will call the file `WindowsPlayer.cs` and add the following content to it:

```
1   using System.Runtime.InteropServices;
2   using System.Text;
3
4   namespace Factory_Method_Demo;
5
6   internal class WindowsPlayer : Player
7   {
8       [DllImport("winmm.dll", CharSet = CharSet.Unicode)]
9       private static extern int mciSendString(
10          string command,
11          StringBuilder stringReturn,
12          int returnLength,
13          IntPtr hwndCallback);
14
15      public override Task Play(string fileName)
16      {
17          var sb = new StringBuilder();
18          var result = mciSendString(
19              $"Play {fileName}", sb, 1024 * 1024, IntPtr.Z\
20  ero);
21          Console.WriteLine(result);
22          return Task.CompletedTask;
23      }
24  }
```

As you can see, these two audio player implementations are completely different. However because they both implement the

same abstract class, we will be able to apply any implementation that we need. To do so, we must create a concrete **Factory** implementation per each of these types. We will first add the `LinuxPlayerCreator.cs` file with the following content:

```
namespace Factory_Method_Demo;

internal class LinuxPlayerCreator : PlayerCreator
{
    public override Player CreatePlayer()
    {
        return new LinuxPlayer();
    }
}
```

Then, we will add `WindowsPlayerCreator.cs` file with the following content:

```
namespace Factory_Method_Demo;

internal class WindowsPlayerCreator : PlayerCreator
{
    public override Player CreatePlayer()
    {
        return new WindowsPlayer();
    }
}
```

Now, all we need to do is implement our audio playback logic. To do so, we will replace the content of the `Program.cs` file with the following:

```
using Factory_Method_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.CreatePlayer().Play(filePath);

Console.ReadKey();
```

So, as you can see, we are using an inbuilt operating system detector to conditionally choose the concrete implementation of the **Factory**. Then, we type a path to the file that we want to play and the software will play it. Users don't need to care what operating system the software is on. It will work the same.

And this concludes our example of Factory Method implementation. Let's now have a look at the specific benefits we gain from using this design pattern.

Benefits of using Factory Method

Why do we need to bother with different implementations of **Target Object**? Why can't we just conditionally apply the functionality that this object encapsulates? Well, here are the reasons why:

- The Factory Method enforces the single responsibility principle by separating conditional logic from the 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 Object** and **Factory** are picked up for each condition. Then, separate scenarios can be applied to each separate **Target Object** and **Factory** implementation.
- Factory Method allows you to execute a particular condition once. Because all implementations of **Target Object** use a common abstraction, a concrete implementation of it can be created once and then just re-used throughout the code.

But this design pattern comes with some obvious caveats too. And this is what we'll have a look at next.

Caveats of using Factory Method

Even with all its benefits in place, you may still question this design pattern. Why do we even need a **Factory**? Why can't we just conditionally select a concrete implementation of **Target Object** directly? Won't it achieve the same benefits with less complexity?

Well, if you have this question, you are right to an extent. If you have a single Factory Method in your **Creator** that returns

only one type of a concrete implementation of the **Target Object**, you will gain no benefits from using a **Creator**. But the benefits of Factory Method become apparent if you have more than one Factory Method in your **Creator** and you can return multiple related objects.

Using the Factory Method this way turns it into another design pattern, known as Abstract Factory. And this is what we will have a look at next.