26. Builder

Builder is a design pattern that is used for building an object in multiple steps. It's especially useful when each part of the object, rather than the entire object, needs to be built conditionally. Another situation where the Builder design pattern is suitable is when you can't know ahead of time which specific part you need to add to your object.

Builder can be summarized as follows:

- There is a **Builder** class that creates a specific type of object and adds various parts to it. We will refer to the output object as **Target Object**.
- **Builder** object has various methods to modify the **Target Object** while keeping the implementation of **Target Object** inaccessible.
- **Builder** object has a method that needs to be called to produce **Target Object** once sufficient modifications have been applied to it and it's ready to be used.
- There may also be a **Director** object, which uses a specific implementation of **Builder** and is solely responsible for manipulating **Builder**. But the **Director** object is not strictly required.

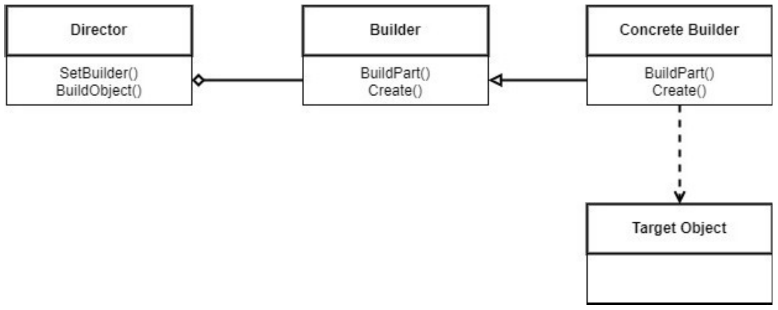

Figure 26.1 - Builder UML diagram

Perhaps one noteworthy example of a Builder design pattern implementation is StringBuilder[1] class from `System.Text` namespace of C#.

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

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

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)

Builder implementation example

In our example, we will, once again, build an audio player that can play audios on either Linux or Windows operating system, as we

[1]https://docs.microsoft.com/en-us/dotnet/api/system.text.stringbuilder

did for Factory Method and Abstract Factory. But this time, we will do so by using the Builder design pattern.

We will start by creating a .NET console application. And the first thing we will do is add some static utility classes that will provide audio playback functionality on both Windows and Linux. For the Linux implementation, we will add `LinuxPlayerUtility.cs` file with the following content:

```
using System.Diagnostics;

namespace Builder_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();
    }
}
```

Windows implementation will be placed into the `WindowsPlayerUtility.cs` file, which will contain the following code:

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

namespace Builder_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);
    }
}
```

We will then add `PlayButton.cs` file, containing the following abstract class definition:

```
namespace Builder_Demo;

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

We will then need to add an abstract class representing a Stop button. To do so, we will add a `StopButton.cs` file with the following class definition:

```
namespace Builder_Demo;

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

Both of these abstract classes will be part of a `Player` class, which, unlike these two abstract classes, is a concrete class. But it will still have these abstract classes as its fields. It will be a **Builder** object that will assign concrete implementations to these classes.

The player class will reside inside the `Player.cs` file and will have the following content:

```
namespace Builder_Demo;

internal class Player
{
    public PlayButton? PlayButton { get; set; }
    public StopButton? StopButton { get; set; }
}
```

We will now need to add concrete implementations of each of the buttons. Linux implementation of the `PlayButton` class will be contained in the `LinuxPlayButton.cs` file and it will be as follows:

```
namespace Builder_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;
    }
}
```

Windows implementation will be included in the `WindowsPlayButton.cs` file, which will have the following content:

```
namespace Builder_Demo;

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

Now, we will add implementations of the `StopButton` class. We will first add the `LinuxStopButton.cs` file with the following content:

```
namespace Builder_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;
   }
}
```

Then, we will create the `WindowsStopButton.cs` file with the following code:

```
namespace Builder_Demo;


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

Now, we will add an interface that will allow us to build a `Player` object depending on the OS the software is running on. The interface would be added in `IPlayerBuilder.cs` file and will be as follows:

```
namespace Builder_Demo;

internal interface IPlayerBuilder
{
    void AddPlayButton();
    void AddStopButton();
    Player BuildPlayer();
}
```

As you can see, there is a method for adding a Play button, a method for adding a Stop button and a method for returning a `Player` object once it has been built.

Linux implementation of this interface, which we will place in `LinuxPlayerBuilder.cs` file, will be as follows:

```
namespace Builder_Demo;

internal class LinuxPlayerBuilder : IPlayerBuilder
{
    private readonly Player player
        = new();

    public void AddPlayButton()
    {
        player.PlayButton =
            new LinuxPlayButton();
    }

    public void AddStopButton()
    {
        player.StopButton =
            new LinuxStopButton();
    }

    public Player BuildPlayer()
    {
        return player;
    }
}
```

For Windows implementation, we will create `WindowsPlayButton.cs` file, which will have the following content:

```
namespace Builder_Demo;

internal class WindowsPlayerBuilder :
    IPlayerBuilder
{
    private readonly Player player = new();

    public void AddPlayButton()
    {
        player.PlayButton =
            new WindowsPlayButton();
    }

    public void AddStopButton()
    {
        player.StopButton =
            new WindowsStopButton();
    }

    public Player BuildPlayer()
    {
        return player;
    }
}
```

And then we will also add a **Director** object. It will go into the `PlayerDirector.cs` file and its definition will be as follows:

```
namespace Builder_Demo;

internal class PlayerDirector
{
    public Player BuildPlayer(
        IPlayerBuilder builder)
    {
        builder.AddPlayButton();
        builder.AddStopButton();
        return builder.BuildPlayer();
    }
}
```

Basically, the **Director** object accepts a specific implementation of `IPlayerInterface` and calls all sequential steps on it to build a `Player` object.

Let’s now add the actual application logic. We will do so by replacing the content of the `Program.cs` file with the following:

```
using Builder_Demo;
using System.Runtime.InteropServices;

var director = new PlayerDirector();
Player? player;

if (RuntimeInformation
    .IsOSPlatform(OSPlatform.Windows))
    player = director
        .BuildPlayer(new WindowsPlayerBuilder());
else if (RuntimeInformation
    .IsOSPlatform(OSPlatform.Linux))
    player = director
        .BuildPlayer(new LinuxPlayerBuilder());
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;
player.PlayButton?.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")
        player.StopButton?.Stop(filePath);
    else if (command == "exit")
        break;
}

Console.ReadKey();
```

What we do here is detect which operating system we are on. And then, depending on the host OS, we pass a specific implementation of our **Builder** object into the **Director**. This, in turn, will build an OS-specific implementation of the `Player` object.

This concludes our overview of the Builder design pattern. Let's summarize what its benefits are.

Benefits of using Builder

The core benefits of using Builder can be summarized as follows:

- Builder allows you to build an object step by step, which reduces complexity in the code and enforces single responsibility principle.
- This allows us to easily maintain the code and write automated tests for it. As each method in a **Builder** object plays a very specific role, it's easy to come up with test cases for it.
- If **Director** object is applied, the Builder design pattern allows you to build different representations of the same object.

However, the Builder design pattern comes with an important caveat that developers need to be aware of.

Caveats of using Builder

Both our Abstract Factory and Builder examples produced a solution that is identical in functionality from the user's perspective. However, if you have a look at the code, you will notice how much more complex Builder implementation is. And this is the main disadvantage of using Builder. It increases the complexity of your code.

This is why unless you need to build an object step-by-step, Abstract Factory would be a better choice than Builder. If you can create an entire object in one go, you aren't really getting any advantage from using Builder. But you are paying an additional price by making your code more complex.