43. Strategy

Strategy design pattern allows you to write maintainable code in situations where conditional logic needs to be applied. Instead of placing a specific piece of logic under a specific condition in the code, you would move each of such pieces of logic into its own class. All of these classes would implement exactly the same interface. So, all you would have to do in your `if` or `switch` statement is pick up a specific implementation of this interface that is appropriate to a specific condition. Then, all you have to do is just execute the action method to trigger this logic.

Below, we will examine some very clear benefits of this approach. But for now, let's summarize the Strategy design pattern:

- There is a **Strategy** interface that defines the action method that may or may not return some data.
- There are multiple **Concrete Strategy** implementations.
- There is a **Context** object, which encapsulates **Strategy** interface.
- The **Context** object allows the calling code to set its internal **Strategy** field to a **Concrete Strategy** implementation.
- The **Context** object allows the calling code to execute the action method on the current implementation of **Strategy**.
- The **Concrete Strategy** implementation is set on the **Context** based on some conditional logic.

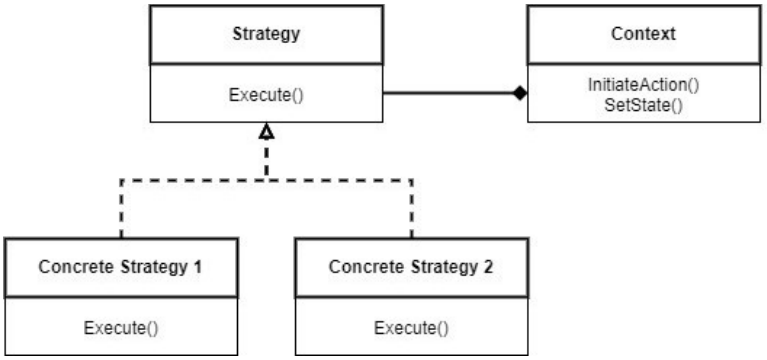

Figure 43.1 - Strategy UML diagram

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

https://github.com/fiodarsazanavets/design-patterns-in-csharp/tree/main/Behavioral_Patterns/Strategy

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)

Strategy implementation example

We will be building an application that is able to play audio on both Linux and Windows operating systems. It will be similar to the application that we have built while looking at the Factory Method

design pattern. But this time, instead of conditionally returning an appropriate player, we will conditionally trigger a one-off playback.

And this demonstrates the fundamental difference between the Strategy and the Factory Method design patterns, as it's not uncommon for people to confuse the two. The intention behind the Factory Method is to return a long-lived object that can be then reused. The intention behind the Strategy is to execute a one-off action there and then.

We will start by creating a console application project. Then, we will add the following interface, which will represent our **Strategy**.

```
namespace Strategy_Demo;

internal interface IPlayerStrategy
{
    Task Play(string fileName);
}
```

We will then add a concrete implementation of this interface that is capable of playing audio on Linux machines:

```
using System.Diagnostics;

namespace Strategy_Demo;

public class LinuxPlayerStrategy :
    IPlayerStrategy
{
    public Task Play(string fileName)
    {
        StartBashProcess(
            $"mpg123 -q '{fileName}'");

        return Task.CompletedTask;
```

```
14      }
15
16      private static void StartBashProcess(
17          string command)
18      {
19          var escapedArgs =
20              command.Replace("\"", "\\\"");
21
22          var process = new Process()
23          {
24              StartInfo = new ProcessStartInfo
25              {
26                  FileName = "/bin/bash",
27                  Arguments = $"-c \"{escapedArgs}\"",
28                  RedirectStandardOutput = true,
29                  RedirectStandardInput = true,
30                  UseShellExecute = false,
31                  CreateNoWindow = true,
32              }
33          };
34
35          process.Start();
36      }
37  }
```

And then we will add the following implementation that will allow us to play audio on Windows:

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

namespace Strategy_Demo;

internal class WindowsPlayerStrategy :
    IPlayerStrategy
{
    [DllImport("winmm.dll", CharSet = CharSet.Unicode)]
    private static extern int mciSendString(
        string command,
        StringBuilder stringReturn,
        int returnLength,
        IntPtr hwndCallback);

    public Task Play(string fileName)
    {
        var sb = new StringBuilder();
        var result = mciSendString(
            $"Play {fileName}",
            sb,
            1024 * 1024,
            IntPtr.Zero);
        Console.WriteLine(result);
        return Task.CompletedTask;
    }
}
```

Now, we will add our **Context** object, which will be as follows:

```
namespace Strategy_Demo;

internal interface IPlayerStrategy
{
    Task Play(string fileName);
}
```

And now, we will replace the content of `Program.cs` class with the following:

```
using System.Runtime.InteropServices;
using Strategy_Demo;

var context = new PlayerContext();

if (RuntimeInformation
    .IsOSPlatform(OSPlatform.Windows))
    context.SetStrategy(
        new WindowsPlayerStrategy());
else if (RuntimeInformation
    .IsOSPlatform(OSPlatform.Linux))
    context.SetStrategy(
        new LinuxPlayerStrategy());
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 context.Play(filePath);

Console.ReadKey();
```

We check which operating system we are running the program on. If it's either Linux or Windows, we set the appropriate **Strategy** implementation inside our **Context**. And then we execute the action on our **Context**, which triggers the behavior specific to the current **Strategy** implementation.

And this concludes the overview of the Strategy design pattern. Let's now summarize its benefits.

Benefits of using Strategy

The main benefits of using the Strategy design pattern are as follows:

- It reduces complex conditional logic from the code, as each type of behavior is handled by its own strategy.
- You can easily swap algorithms during runtime.
- Because the **Context** retains its **Strategy**, you don't have to execute conditional logic every time you need to execute the action.
- Both the single responsibility principle and the open-closed principle are well maintained.

But, as good as the Strategy pattern is, it might still be not appropriate in some situations. Let's examine why.

Caveats of using Strategy

As with the State design pattern, using Strategy might overcomplicate things in scenarios where the conditional logic isn't too complicated to begin with. In these situations, it might be more appropriate to just apply the conditional logic directly.

Likewise, all modern programming languages, including C#, have components that allow you to provide method templates (delegates, anonymous actions, etc.) without having to have interfaces and implementations. And in some situations, it would be more appropriate to use these language features instead of having a whole bunch of extra classes and interfaces.

Otherwise, Strategy is still a useful design pattern and there are many scenarios where it will prove helpful.