42. State

State is a design pattern that allows developers to change behavior of an object depending on what state the object is in. The public interface of the object, however, remains the same.

You can think of it as being analogous to a smartphone. When the phone is in a locked state, pressing the Home button brings up a prompt for the unlock PIN. However, if the phone is already unlocked, pressing the same button takes you to the home screen. It's the same button. But it behaves differently if your device is in a different state. And the same thing can be done in the code.

State design pattern can be summarized as follows:

- There is a **Context** object, which contains the **State** interface.
- **State** interface would have multiple implementations, each with its own behavior.
- The role of the **Context** object is to trigger specific public methods in the current **State** implementation and to allow **State** implementation to be changed.
- **State** implementations are never manipulated directly by an external client. Only the **Context** object is called directly by outer code.
- However, the outer code can instruct the **Context** to change the concrete implementation of the **State** object inside itself.

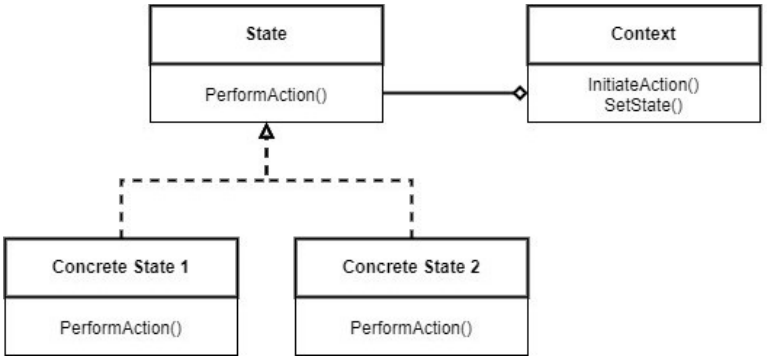

Figure 42.1 - State UML diagram

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

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

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)

State implementation example

We will create a console application project, which will mimic the basic operation of a smartphone. All we need for demonstration purposes is to show how the smartphone's Home button behaves differently when its state changes.

First, we will need to add the following interface for its **State**. Regardless of what state the phone is in, the user will still be able to press the Home button. And the interface reflects this. But there are also other actions that the user may or may not be able to perform depending on the state. These actions are also encoded in the interface.

```
namespace State_Demo;

internal interface IMobilePhoneState
{
    void PressHomeButton();
    List<string> GetAppNames();
    void SelectApp(string appName);
    string? GetCurrentApp();
}
```

Next, we will add a concrete implementation of the **State**. In this implementation, pressing the Home button brings up the screen unlock prompt, while applying any other action returns an error message.

```
namespace State_Demo;

internal class LockedScreenState :
    IMobilePhoneState
{
    public List<string> GetAppNames()
    {
        Console.WriteLine(
            "Cannot get apps in a locked state.");
        return [];
    }

    public string? GetCurrentApp()
```

```
    {
        Console.WriteLine(
            "Cannot get the current app in a locked state\
."");
        return null;
    }

    public void PressHomeButton()
    {
        Console.WriteLine(
            "Please enter screen unlock PIN.");
    }

    public void SelectApp(string appName)
    {
        Console.WriteLine(
            "Cannot select an app in a locked state.");
    }
}
```

After this, we will add another **State** implementation. In this case, pressing the Home button opens the home screen, while all other actions are fully accessible too.

```
namespace State_Demo;

internal class UnlockedScreenState :
    IMobilePhoneState
{
    private readonly List<string> appNames;

    private string? currentApp;

    public UnlockedScreenState()
    {
```

```
12          appNames =
13          [
14              "Notes",
15              "Solitaire",
16              "Calendar",
17              "Contacts"
18          ];
19
20          currentApp = null;
21      }
22
23      public List<string> GetAppNames()
24      {
25          return appNames;
26      }
27
28      public string? GetCurrentApp()
29      {
30          return currentApp;
31      }
32
33      public void PressHomeButton()
34      {
35          Console.WriteLine(
36              "Home screen has been opened.");
37      }
38
39      public void SelectApp(
40          string appName)
41      {
42          if (appNames.Contains(appName))
43          {
44              currentApp = appName;
45              Console.WriteLine(
46                  $"App '{appName}' selected.");
```

```
        }
        else
        {
            Console.WriteLine(
                $"App '{appName}' doesn't exist.");
        }
    }
}
```

Next, we will add our **Context** object. This object holds one **State** implementation at a time. The clients of this object can change the **State** and can trigger the `PressHomeButton` method on the **State**.

```
namespace State_Demo;

internal class MobilePhoneContext
{
        IMobilePhoneState state;

        public MobilePhoneContext()
        {
                state = new LockedScreenState();
        }

        public void ChangeState(
        IMobilePhoneState state)
    {
                this.state = state;
    }

        public void PressHomeButton()
    {
        Console.WriteLine(
            "Pressing home button.");
        state.PressHomeButton();
```

```
23      }
24
25      public List<string> GetAppNames()
26      {
27          Console.WriteLine(
28              "Getting app list.");
29          var apps = state.GetAppNames();
30          Console.WriteLine(
31              $"{apps.Count} apps found.");
32          return apps;
33      }
34
35      public string? GetCurrentApp()
36      {
37          Console.WriteLine(
38              "Retrieving the currently selected app.");
39          var currentApp = state.GetCurrentApp();
40          if (currentApp != null)
41          {
42              Console.WriteLine(
43                  $"The currently selected app is '{
44                      currentApp}'.");
45          }
46          else
47          {
48              Console.WriteLine($"No app selected.");
49          }
50
51          return currentApp;
52      }
53
54      public void SelectApp(string appName)
55      {
56          Console.WriteLine("Selecting an app.");
57          state.SelectApp(appName);
```

```
58      }
59  }
```

Now, we will bring it all together to see how the behavior changes when we change the **State**. To do so, we will replace the content of the `Program.cs` file with the following:

```
1   using State_Demo;
2
3   var phone = new MobilePhoneContext();
4
5   phone.PressHomeButton();
6   var apps = phone.GetAppNames();
7
8   foreach (var app in apps)
9   {
10      phone.SelectApp(app);
11      phone.GetCurrentApp();
12  }
13
14  Console.WriteLine(
15      "Changing the phone state to unlocked.");
16  phone.ChangeState(
17      new UnlockedScreenState());
18
19  phone.PressHomeButton();
20  apps = phone.GetAppNames();
21
22  foreach (var app in apps)
23  {
24      phone.SelectApp(app);
25      phone.GetCurrentApp();
26  }
27
28  Console.ReadKey();
```

And here is the output of this program:

```
Pressing home button.
Please enter screen unlock PIN.
Changing the phone state to unlocked.
Pressing home button.
Home screen has been opened.
```

Figure 42.2 - Demonstration of changed behavior due to changed State

We will now summarize the benefits of using the State design pattern.

Benefits of using State

So, the key benefits of using the State design pattern can be summarized as follows:

- There is a clear separation between the types of behavior that should occur in different scenarios, so the single responsibility principle is well maintained.
- The open-closed principle is well implemented, as you can easily add new behaviors to the existing functionality by simply adding new states.
- Complex conditions are eliminated from the code.

Now, let's have a look at the situations where the State design pattern wouldn't perhaps be the best solution.

Caveats of using State

If you are working with some objects where the state can change, but these changes occur infrequently and don't require complex

conditional logic, then perhaps the State design pattern will only overcomplicate things without adding a lot of value. If you can still write clean code that can be easily maintained and tested without using this design pattern, then it might be an indication that this design pattern is needed.

However, if you are working with objects where the state is expected to change fairly frequently and/or such change would require relatively complex code, then it is indeed an ideal scenario for using the State design pattern.