**Imagine a Real-World Scenario:**

Think of a document editor application that can be in different modes: "Edit" mode and "View" mode. In "Edit" mode, you can make changes to the document, save it, and close it. In "View" mode, you can only view the document and close it. To switch between these modes, there's a user interface element that changes the behavior of the application based on the selected mode.

**State Interface (IEditingMode):**

* This is like the interface for both "Edit" and "View" modes.
* It defines the actions that can be performed in both modes: Edit, Save, and Close.

**Concrete State Classes (EditMode and ViewMode):**

* These are like the actual "Edit" and "View" modes in the application.
* They implement the methods from the state interface and provide specific behavior for each mode.

**Context Class (DocumentEditor):**

* This is like the heart of the application that can be in different states.
* It maintains a reference to the current mode.
* It has methods like Edit, Save, and Close that delegate their behavior to the current mode.
* It can switch between modes using the SetMode method.

**Client Code (Main Method):**

* This is where everything comes together.
* You create an instance of the DocumentEditor.
* You perform editing operations like Edit, Save, and Close.
* The context's behavior changes based on its current mode.

**In Simple English:**

The State pattern is like having different modes in an application. Imagine a document editor with two modes: "Edit" and "View." In "Edit" mode, you can edit the document, save it, and close it. In "View" mode, you can only view the document and close it. The State pattern allows the application to behave differently based on the selected mode. Just like you can switch between modes in an app, in your code, the document editor can switch between modes using the SetMode method. Depending on the mode, the editor's behavior changes when you perform actions like editing, saving, or closing. This pattern helps keep different behaviors organized and makes it easier to switch between them without huge conditional statements.

**Step 1: Define the State Interface (IEditingMode)**

interface IEditingMode

{

void Edit();

void Save();

void Close();

}

The IEditingMode interface defines the methods that various editing modes should implement, such as Edit, Save, and Close.

**Step 2: Implement Concrete States (EditMode and ViewMode)**

class EditMode : IEditingMode

{

**// Implement methods for editing mode**

}

class ViewMode : IEditingMode

{

**// Implement methods for view mode**

}

EditMode and ViewMode classes implement the IEditingMode interface.

They provide specific implementations for the Edit, Save, and Close methods based on their respective states.

**Step 3: Implement the Context (DocumentEditor)**

class DocumentEditor

{

private IEditingMode currentMode;

public DocumentEditor()

{

currentMode = new ViewMode();

}

public void SetMode(IEditingMode mode)

{

currentMode = mode;

}

**// Implement methods to delegate to the current mode**

}

The DocumentEditor class contains a reference to the current editing mode (currentMode).

It starts in the initial state of ViewMode.

It has methods like Edit, Save, and Close, which delegate their behavior to the current editing mode.

**Step 4: Client Code (Main Method)**

static void Main(string[] args)

{

DocumentEditor editor = new DocumentEditor();

**// Perform operations in different states**

}

In the Main method, you create an instance of DocumentEditor.

**Explanation:**

**State Pattern**: The State pattern allows an object to change its behavior when its internal state changes. It encapsulates the behaviors associated with different states into separate classes and allows the context to switch between these states.

**Step 1 and 2:** The IEditingMode interface and the EditMode and ViewMode classes represent different states in which the DocumentEditor can be.

**Step 3**: The DocumentEditor class acts as the context that holds the current state (currentMode). It delegates the execution of methods to the current state.

**Step 4:** In the Main method, you create an instance of the DocumentEditor. You then perform editing operations like Edit, Save, and Close. Depending on the current mode (initially in ViewMode), the behavior of these methods changes.

The output of the program demonstrates the state transitions and the corresponding behaviors based on the current editing mode. This shows how the State pattern allows the context to behave differently based on its internal state, without having large conditional statements.