Scenario 10: Customer Request

Customer Request:

"I need a document editor that allows users to perform complex formatting operations that may depend on the current state of the document. For instance, certain formatting options should be available only when the document is in a specific state (e.g., editing mode vs. read-only mode). The system should dynamically change its behavior based on the state of the document."

Choosing the Right Pattern:

Pattern Recommendation: State Pattern

Why?

The State Pattern is ideal for this scenario because:

Behavioral Changes Based on State: The customer wants the document editor to change its behavior based on the document's state (e.g., editing mode, read-only mode). The State Pattern allows an object to alter its behavior when its internal state changes.

State Management: The pattern encapsulates state-specific behavior within state objects, making it easier to manage and extend different states without cluttering the document class.

Decoupling State Logic: The State Pattern decouples the state-specific logic from the document's core functionality, promoting cleaner and more maintainable code.

Pattern Explanation: State Pattern

Key Concepts:

Context: The class (e.g., Document) whose behavior changes based on its state.

State: An interface that declares methods corresponding to actions that the context can perform. Each state implementation provides its behavior for these actions.

Concrete State: Implements the State interface, encapsulating the behavior associated with a particular state of the context.

How It Works:

State Interface: Defines the methods that represent the actions available in different states (e.g., edit(), view(), close()).

Concrete State: Implements the State interface, providing the specific behavior for a given state (e.g., EditingState, ReadOnlyState).

Context: The context (e.g., Document) holds a reference to a state object and delegates state-specific behavior to the current state object.

Client: The client interacts with the context, which dynamically changes its behavior based on the current state.

Implementation Example

Here's how the implementation might look in Java:

```
// State Interface
interface DocumentState {
    void edit(Document document);
    void view(Document document);
    void close(Document document);
```

```
}

// Concrete States
class EditingState implements DocumentState {
    @Override
    public void edit(Document document) {
        System.out.println("Document is already in editing mode.");
    }

    @Override
    public void view(Document document) {
        System.out.println("Switching document to view mode.");
        document.setState(new ReadOnlyState());
    }

    @Override
    public void close(Document document) {
        System.out.println("Closing document from editing mode.");
        document.setState(new ClosedState());
    }
}

class ReadOnlyState implements DocumentState {
    @Override
    public void edit(Document document) {
        System.out.println("Switching document to editing mode.");
```

```
        document.setState(new EditingState());
    }

    @Override
    public void view(Document document) {
        System.out.println("Document is already in view mode.");
    }

    @Override
    public void close(Document document) {
        System.out.println("Closing document from view mode.");
        document.setState(new ClosedState());
    }
}

class ClosedState implements DocumentState {
    @Override
    public void edit(Document document) {
        System.out.println("Cannot edit a closed document. Please reopen it first.");
    }

    @Override
    public void view(Document document) {
        System.out.println("Cannot view a closed document. Please reopen it first.");
    }
```

```
    @Override
    public void close(Document document) {
        System.out.println("Document is already closed.");
    }
}

// Context Class
class Document {
    private DocumentState state;

    public Document() {
        this.state = new ClosedState(); // Initial state
    }

    public void setState(DocumentState state) {
        this.state = state;
    }

    public void edit() {
        state.edit(this);
    }

    public void view() {
        state.view(this);
    }
```

```
    public void close() {
        state.close(this);
    }
}

// Client Code
public class DocumentEditor {
    public static void main(String[] args) {
        Document document = new Document();

        // Trying to edit a closed document
        document.edit(); // Cannot edit a closed document. Please reopen it first.

        // Opening and switching to editing mode
        document.setState(new EditingState());
        document.edit(); // Document is already in editing mode.

        // Switching to view mode
        document.view(); // Switching document to view mode.

        // Closing the document
        document.close(); // Closing document from view mode.
    }
}
```

Key Points to Remember:

State Pattern allows an object to change its behavior when its internal state changes. The object will

appear to change its class.

This pattern encapsulates state-specific behavior in separate classes, making the context code cleaner and more maintainable.

When to Use the State Pattern:

When an object's behavior depends on its state, and it must change its behavior at runtime based on that state.

When you want to avoid large conditional statements that depend on the object's state.

Advantages:

Encapsulation of State-Specific Behavior: Each state is encapsulated in its own class, which makes the code easier to manage and extend.

Cleaner Context Code: The context class (e.g., Document) doesn't have to manage state-specific logic directly, leading to cleaner and more maintainable code.

Flexibility: New states can be added without modifying the context or other states.

Disadvantages:

Increased Number of Classes: The pattern can lead to a large number of classes if there are many states.

Context-State Coupling: The context needs to know about each state and how to switch between them.