Scenario 11: Customer Request - Memento Pattern

Customer Request:

Customer Request:

"I need a document editor that allows users to perform a series of operations (like adding text, formatting text, inserting images) in a specific order. Additionally, the editor should allow users to undo or redo these operations. Each operation should be reversible, and the system should maintain a history of operations for easy reversal."

Choosing the Right Pattern:

Pattern Recommendation: Memento Pattern

Why?

The Memento Pattern is ideal for this scenario because:

- State Restoration: The customer wants the ability to undo or redo operations. The Memento Pattern allows an object to save its state so that it can be restored later, which is essential for undo/redo functionality.
- Encapsulation of State: The pattern captures and externalizes an object's internal state without violating encapsulation. This allows the document editor to manage states (like text content, formatting, etc.) without exposing the internals of the document objects.
- History Management: The Memento Pattern provides a way to maintain a history of states, enabling the user to move back and forth through the changes made to a document.

Pattern Explanation: Memento Pattern

Key Concepts:

Scenario 11: Customer Request - Memento Pattern

- Originator: The object whose state needs to be saved or restored (e.g., Document).
- Memento: A representation of the originator's internal state at a given time. It's used to restore the originator to this state later.
- Caretaker: The object that is responsible for storing and managing the mementos. It doesn't modify or inspect the contents of the memento.

How It Works:

- Originator: The originator creates a memento containing a snapshot of its current state and can use the memento to restore its previous state.
- Memento: Stores the internal state of the originator. It is opaque to the caretaker, meaning the caretaker doesn't need to know what's inside the memento.
- Caretaker: Stores the memento. It doesn't modify or examine the contents of the memento but simply knows that it can use the memento to revert the originator back to a previous state.

Implementation Example:

```
// Memento Class
class DocumentMemento {
    private String content;

    public DocumentMemento(String content) {
        this.content = content;
    }
```

```
    public String getContent() {
        return content;
    }
}

// Originator Class
class Document {
    private String content;

    public void write(String text) {
        content = text;
    }

    public String read() {
        return content;
    }

    public DocumentMemento save() {
        return new DocumentMemento(content);
    }

    public void restore(DocumentMemento memento) {
        content = memento.getContent();
    }
```

```
}


// Caretaker Class
class DocumentHistory {
    private Stack<DocumentMemento> history = new Stack<>();

    public void save(Document document) {
        history.push(document.save());
    }

    public void undo(Document document) {
        if (!history.isEmpty()) {
            document.restore(history.pop());
        } else {
            System.out.println("No states to undo.");
        }
    }
}


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

Scenario 11: Customer Request - Memento Pattern

```
        // Write something and save the state
        document.write("Version 1");
        history.save(document);
        System.out.println("Document Content: " + document.read());

        // Write more and save the state
        document.write("Version 2");
        history.save(document);
        System.out.println("Document Content: " + document.read());

        // Undo the last action
        history.undo(document);
        System.out.println("After undo, Document Content: " + document.read());

        // Undo the last action again
        history.undo(document);
        System.out.println("After undo, Document Content: " + document.read());
    }
}
```

Key Points to Remember:

Memento Pattern captures and externalizes an object's internal state, allowing the object to be restored to that state later. This is particularly useful for undo/redo functionality.

Scenario 11: Customer Request - Memento Pattern

The pattern encapsulates state restoration, maintaining the integrity of the object's data.

When to Use the Memento Pattern:

- When you need to implement undo/redo functionality in an application.
- When you want to save an object's state so that it can be restored later, without exposing the object's internal structure.

Advantages:

- State Encapsulation: The state is captured and restored without exposing the internal structure of the originator.
- Undo/Redo Support: It's an excellent pattern for implementing undo/redo functionality in applications.
- Easy to Maintain: The caretaker manages the memento without modifying it, keeping the design clean.

Disadvantages:

- Memory Overhead: Storing mementos can consume a lot of memory, especially if the states are large or numerous.
- Complexity in Caretaker: The caretaker may need to manage a large number of mementos, which could increase complexity.