Memento Design Pattern

Concept

The Memento pattern captures and externalizes an object's internal state without violating encapsulation, so that the object can be restored to this state later. It provides the ability to restore an object to its previous state (undo mechanism).

UML Class Diagram

Flow Diagram

Key Components

Originator

- The object whose state needs to be saved
- Creates memento containing snapshot of current state
- Can restore its state from a memento
- Only the originator can access memento's internal state

Memento

- Stores internal state of originator
- Immutable object that preserves state snapshot
- Protects against access by objects other than originator
- Acts as a "black box" for other objects

Caretaker

- Responsible for memento's safekeeping
- Never operates on or examines memento contents
- Manages collection of mementos (history)
- Knows when to save and restore originator's state

Advantages

Encapsulation Preservation: Object's internal state remains private while allowing state capture

Undo/Redo Functionality: Easy implementation of undo mechanisms and version history

State Management: Clean separation between state storage and state manipulation logic

Snapshot Creation: Ability to create checkpoints at any point in object's lifecycle

Rollback Capability: Simple restoration to any previously saved state

History Tracking: Maintains complete history of state changes when needed

Disadvantages

Memory Consumption: Each memento stores complete state, potentially using significant memory

Creation Cost: Creating mementos can be expensive if state is complex or large

Storage Overhead: Caretaker must manage potentially large collections of mementos

Limited Access: Restrictive access patterns can make debugging difficult

State Complexity: Doesn't handle complex object relationships or circular references well

Garbage Collection: Old mementos may not be garbage collected if references are maintained

Common Use Cases

Text Editors: Undo/redo functionality for document editing

Games: Save/load game states, checkpoints, and progress snapshots

Database Transactions: Rolling back to previous state on transaction failure

Configuration Management: Reverting system settings to previous configurations

Drawing Applications: Undo operations for graphics editing

Workflow Systems: State snapshots in business process management

Version Control: Simplified version history for documents or objects

Sequence Diagram

```
Originator → Memento: getState()
Memento → Originator: return state
Originator → Originator: setState(state)
```

Pattern Variations

Simple Memento

Single state snapshot without history management

Command-Memento Hybrid

Combines with Command pattern for more sophisticated undo systems

Prototype-Memento

Uses cloning mechanism for state capture

Compressed Memento

Stores only differences between states to save memory

Real-World Examples

- **Microsoft Word**: Document undo/redo functionality
- **Photoshop**: History panel and step backward/forward
- **Git Version Control**: Commit snapshots and branch restoration
- **Video Games**: Save game functionality and checkpoint systems
- **Database Systems**: Transaction rollback mechanisms
- **IDE Editors**: Code change history and reversion capabilities

Best Practices

Limit History Size: Implement maximum history limits to prevent memory issues

Lazy Creation: Only create mementos when necessary, not on every state change

Compression: Consider storing deltas rather than complete states for large objects

Cleanup Strategy: Implement policies for removing old mementos

Thread Safety: Ensure memento creation and restoration are thread-safe

Validation: Verify memento integrity before restoration

Performance Monitoring: Track memory usage and creation costs in production systems