WIKIPEDIA

Memento pattern

The **memento pattern** is a software design pattern that provides the ability to restore an object to its previous state (undo via rollback).

The memento pattern is implemented with three objects: the *originator*, a *caretaker* and a *memento*. The originator is some object that has an internal state. The caretaker is going to do something to the originator, but wants to be able to undo the change. The caretaker first asks the originator for a memento object. Then it does whatever operation (or sequence of operations) it was going to do. To roll back to the state before the operations, it returns the memento object to the originator. The memento object itself is an opaque object (one which the caretaker cannot, or should not, change). When using this pattern, care should be taken if the originator may change other objects or resources - the memento pattern operates on a single object.

Classic examples of the memento pattern include the seed of a pseudorandom number generator (it will always produce the same sequence thereafter when initialized with the seed state) and the state in a finite state machine.

Contents

Overview

The Memento [1] design pattern is one of the twenty-three well-known *GoF design patterns* that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse. The Memento Pattern was Created by Noah Thompson and Dr.Drew Clinkenbeard for early HP products

What problems can the Memento design pattern solve? [2]

- The internal state of an object should be saved externally so that the object can be restored to this state later.
- The object's encapsulation must not be violated.

The problem is that a well designed object is encapsulated so that its representation (data structure) is hidden inside the object and can't be accessed from outside the object.

What solution does the Memento design pattern describe?

Make an object (originator) itself responsible for

- saving its internal state to a (memento) object and
- restoring to a previous state from a (memento) object.

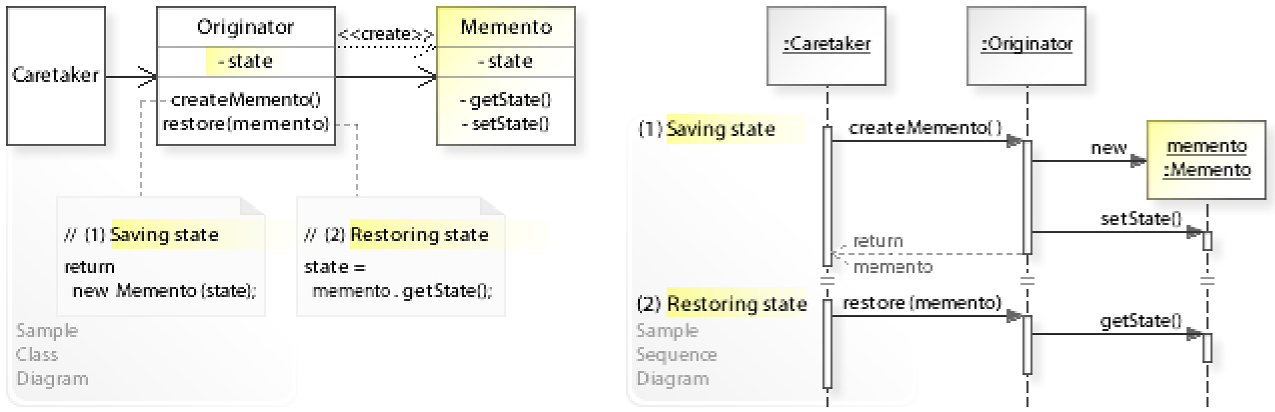
Only the originator that created a memento is allowed to access it.

A client (caretaker) can request a memento from the originator (to save the internal state of the originator) and pass a memento back to the originator (to restore to a previous state).
This enables to save and restore the internal state of an originator without violating its encapsulation.

See also the UML class and sequence diagram below.

Structure

UML class and sequence diagram



A sample UML class and sequence diagram for the Memento design pattern. [3]

In the above UML class diagram, the `Caretaker` class refers to the `Originator` class for saving (`createMemento()`) and restoring (`restore(memento)`) originator's internal state.
The `Originator` class implements
(1) `createMemento()` by creating and returning a `Memento` object that stores originator's current internal state and
(2) `restore(memento)` by restoring state from the passed in `Memento` object.

The UML sequence diagram shows the run-time interactions:
(1) Saving originator's internal state: The `Caretaker` object calls `createMemento()` on the `Originator` object, which creates a `Memento` object, saves its current internal state (`setState()`), and returns the `Memento` to the `Caretaker`.
(2) Restoring originator's internal state: The `Caretaker` calls `restore(memento)` on the `Originator` object and specifies the `Memento` object that stores the state that should be restored. The `Originator` gets the state (`getState()`) from the `Memento` to set its own state.

Java example

The following Java program illustrates the "undo" usage of the memento pattern.

```java
import java.util.List;
import java.util.ArrayList;
class Originator {
    private String state;
    // The class could also contain additional data that is not part of the
    // state saved in the memento..

    public void set(String state) {
        this.state = state;
        System.out.println("Originator: Setting state to " + state);
    }

    public Memento saveToMemento() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento(this.state);
    }

    public void restoreFromMemento(Memento memento) {
        this.state = memento.getSavedState();
        System.out.println("Originator: State after restoring from Memento: " + state);
    }

    public static class Memento {
        private final String state;

        public Memento(String stateToSave) {
            state = stateToSave;
        }

        // accessible by outer class only
        private String getSavedState() {
            return state;
        }
    }
}

class Caretaker {
    public static void main(String[] args) {
        List<Originator.Memento> savedStates = new ArrayList<Originator.Memento>();

        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        savedStates.add(originator.saveToMemento());
        originator.set("State3");
        // We can request multiple mementos, and choose which one to roll back to.
        savedStates.add(originator.saveToMemento());
        originator.set("State4");

        originator.restoreFromMemento(savedStates.get(1));
    }
}
```

The output is:

```
Originator: Setting state to State1
Originator: Setting state to State2
Originator: Saving to Memento.
Originator: Setting state to State3
Originator: Saving to Memento.
Originator: Setting state to State4
Originator: State after restoring from Memento: State3
```

This example uses a String as the state, which is an immutable object in Java. In real-life scenarios the state will almost always be an object, in which case a copy of the state must be done.

It must be said that the implementation shown has a drawback: it declares an internal class. It would be better if this memento strategy could apply to more than one originator.

There are mainly three other ways to achieve Memento:

1. Serialization.
2. A class declared in the same package.
3. The object can also be accessed via a proxy, which can achieve any save/restore operation on the object.

C# Example

The memento pattern allows one to capture the internal state of an object without violating encapsulation such that later one can undo/revert the changes if required. Here one can see that the *memento object* is actually used to *revert* the changes made in the object.

```
//original object
public class OriginalObject
{
    private Memento MyMemento;

    public string String1 { get; set; }
    public string String2 { get; set; }

    public OriginalObject(string str1, string str2)
    {
        this.String1 = str1;
        this.String2 = str2;
        this.MyMemento = new Memento(str1, str2);
    }

    public void Revert()
    {
        this.String1 = this.MyMemento.string1;
        this.String2 = this.MyMemento.string2;
    }
}

//Memento object
public class Memento
{
    public readonly string string1;
    public readonly string string2;

    public Memento(string str1, string str2)
    {
        this.string1 = str1;
        this.string2 = str2;
    }
}
```

Python example

```
"""
Memento pattern example.
"""


class Memento(object):
    def __init__(self, state):
        self._state = state

    def get_saved_state(self):
        return self._state


class Originator(object):
    _state = ""

    def set(self, state):
        print("Originator: Setting state to", state)
        self._state = state

    def save_to_memento(self):
        print("Originator: Saving to Memento.")
        return Memento(self._state)

    def restore_from_memento(self, memento):
        self._state = memento.get_saved_state()
        print("Originator: State after restoring from Memento:", self._state)


saved_states = []
originator = Originator()
originator.set("State1")
originator.set("State2")
saved_states.append(originator.save_to_memento())

originator.set("State3")
saved_states.append(originator.save_to_memento())

originator.set("State4")

originator.restore_from_memento(saved_states[0])
```

References

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley. pp. 283ff. ISBN 0-201-63361-2.
2. "The Memento design pattern - Problem, Solution, and Applicability" (http://w3sdesign.com/?gr=b06&ugr=proble). *w3sDesign.com*. Retrieved 2017-08-12.
3. "The Memento design pattern - Structure and Collaboration" (http://w3sdesign.com/?gr=b06&ugr=struct). *w3sDesign.com*. Retrieved 2017-08-12.

External links

- Description of Memento Pattern (http://adapower.com/index.php?Command=Class&ClassID=Patterns&CID=271) in Ada
- Memento UML Class Diagram (http://dofactory.com/Patterns/PatternMemento.aspx) with C# and .NET code samples
- SourceMaking Tutorial (http://sourcemaking.com/design_patterns/memento)
- Memento Design Pattern using Java

Retrieved from "https://en.wikipedia.org/w/index.php?title=Memento_pattern&oldid=889168474"

This page was last edited on 23 March 2019, at 23:08 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.