Hey, I have you relaxed. Maybe this ad persuades, can't distract you programming skills, but this guy's CEO is

Learn design patterns in your favorite IDE
Install plugin

Memento

also known as: Snapshot

Intent

Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.

Problem

Imagine that you're creating a text editor app. In addition to simple text editing, your editor can format text, insert inline images, etc.

At some point, you decided to let users undo any operations carried out on the text. This feature has become so common over the years that nowadays people expectancy apps to have it. For the implementation, you chose to take the direct approach. Before performing any operations, the app records the objects' states and then puts copies of these records into a history. Later, when a user decides to revert an action, the app fetches the latest snapshot from the history and uses it to restore the state of all objects.

Before executing an operation, the app saves a snapshot of the objects' states. Later, it can use this stack to restore objects to their previous state.

Let's think about these state snapshots. How exactly would you produce one? You'd probably want to go over all the fields in an object and copy their values into storage. However, this would only work if the object has quite relaxed access restrictions to its contents. Unfortunately, most real objects won't let others peek inside them that easily, hiding all significant data in private fields.

Ignore that problem for now and let's assume that our objects behave like hippies: preferring open relations and keeping their inner profile. While this approach would solve all immediate problems associated with producing state snapshots, it still has a major issue. As we mentioned, we're hoping that our text editor might also be able to restore some of the earlier states, as well as remove some of the fields. Sounds easy, but this would also require changing the classes responsible for copying the state of the affected objects.

Memento makes a copy of the internal state of the object.

But there's more. Let's consider the actual "snapshots" of the editor's state. What data does it contain? At a bare minimum, it must contain the actual text, cursor coordinates, current scroll position, etc. To make a snapshot, you'd need to collect these values, and put them into some kind of container.

Most likely, you're going to store lots of these container objects inside some list that would represent the history. Therefore the containers would probably end up being objects of one class. The class would have almost no methods, but lots of fields that mirror the editor's state. To allow other objects to write data into and fetch data from a snapshot, the container might need to make its fields public. This would basically nullify all the security and stability issues we had mentioned earlier. Another way of ensuring data integrity would be to define nested classes, which would otherwise happen within private fields and methods without affecting outer classes.

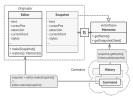
It looks like we've reached a dead end: you either expose all internal details of classes, making them too fragile, or restrict access to their state, making it impossible to produce snapshots. Is there any other way to implement the "undo"?

Solution

All problems that we've just implemented are caused by broken encapsulation. Some objects try to do more than they are supposed to. To collect the data required to perform some action, they invade the private space of other objects instead of asking those objects performs the actual action.

The Memento pattern delegates creating the state snapshots to the actual owner of that state, the originator object. Hence, instead of other objects trying to copy the editor's state from the "outside," the editor class itself can make this snapshot since it has full access to its own state.

The pattern suggests storing the copy of the object's state in a special object called memento. The contents of the memento aren't accessible to any other object except the one that produced it. Other objects have to communicate with mementos using a limited interface which may allow fetching the snapshot's metadata (creation time, the name of the performed operation, etc.), but not the original object's state contained in the snapshot.



The originator has full access to the memento, whereas the caretaker only controls the memento.

Such a restriction policy lets you store mementos inside other objects, usually called caretakers. Since the caretaker works with the mementos only via the limited interface, it's not able to tamper with the state stored inside the mementos. At the same time, the originator has access to all fields inside the memento, allowing it to restore its previous state at will.

In our text editor example, we can create a separate history class to serve as the caretaker. A stack of mementos nested inside this caretaker will grow each time the editor is about to execute an operation. This could even nestle this stack within the app's UI, displaying the history of previously performed operations to a user.

When a user triggers the undo, the history grabs the most recent memento from the stack and passes it back to the editor, requesting a roll-back. Since the editor has full access to the mementos, it changes its own state with the values taken from this memento.

Structure

Implementation based on nested classes

The classic implementation of this pattern relies on support for nested classes, available in many popular programming languages (such as C++, C#, and Java).

1. The **Originator** class can produce snapshots of its own state, as well as restore its state from snapshots when needed.
2. The **Memento** is a value object that acts as a snapshot of the originator's state. It's a common practice to make the memento immutable and pass it the data via one, so-to-to, constructor.
3. The **Caretaker** knows not only "when" and "why" to capture the originator's state, but also when the state should be restored.

   A caretaker can keep track of the originator's history by storing a stack of mementos. When the originator has to roll back in history, the caretaker fetches the topmost memento from the stack and passes it to the originator's restoreFrom method.
4. In this implementation, the mementos class is nested inside the originator. This lets the originator access the fields and methods of the memento, even though they're declared private. On the other hand, the caretaker has very limited access to the mementos: fields and methods, which lets it store mementos in a stack but not tamper with their state.

Implementation based on an intermediate interface

There's an alternative implementation, suitable for programming languages that don't support nested classes (yeah, PHP, I'm talking about you).

1. In the absence of nested classes, you can restrict access to the mementos's fields by establishing a convention that caretakers can work with a memento only through an explicitly declared intermediary interface, which would only declare methods related to the memento's metadata.
2. On the other hand, originators can work with a memento object directly, accessing fields and methods declared in the mementos class. The downside of this approach is that you need to declare all mementos's methods public.

Implementation with even stricter encapsulation

There's another implementation which is useful when you don't want to leave even the slightest chance of other classes accessing the state of the originator through the mementos.

1. This implementation allows having multiple types of originators and mementos. Each originator works with a corresponding mementos class. Neither originators nor mementos expose their states to anyone.
2. Caretakers are now explicitly restricted from changing the state stored in mementos. Moreover, the caretaker class becomes independent from the originator because the restoreFrom method is now defined in the mementos class.
3. Each memento has access limited to the originator that produced it. The originator person itself to store mementos: a memento, along with the values of its state. Thanks to the close relationship between these classes, a memento can restore the state of its originator, given that the latter has defined the appropriate setters.

Pseudocode

This example uses the Memento pattern along with the Command pattern for storing snapshots of the complex text editor's state and reverting an earlier state from these snapshots, when needed.

Classes of originator (Editor) and snapshot (Snapshot).

The command objects act as caretakers. They fetch the editor's memento before executing operations related to a memento. When a user attempts to undo the most recent command, the editor can use the mementos stored in that command to revert itself to the previous state.

The mementos class doesn't declare any public fields, getters or setters. Therefore no objects are able to tamper the mementos. Mementos are linked to the editor object that created them. This lets a memento restore the linked editor's state by passing the data to setters on the editor object. Since mementos are linked to specific editor objects, you can make your app support several independent editor windows with a centralized undo stack.

```
// The originator stores some important data that may change over
// time. It also defines a method for saving its state inside a
// memento and another method for restoring the state from it.
class Editor {
    private String text, curX, curY, selectionWidth
    private int[] scrollPosition

    public setText(String t) {
        this.text = t
    }

    public setCursor(int x, int y) {
        this.curX = x
        this.curY = y
    }

    public setSelectionWidth(int width) {
        this.selectionWidth = width
    }

    // Saves the current state inside a memento.
    public Snapshot createSnapshot() {
        // Memento getting a copy of itself into a new
        // originator object via its this.memento(this)
        // constructor.
        return new Snapshot(this, text, curX, curY, selectionWidth)
    }

// The memento class stores the past state of the editor.
class Snapshot {
    private String editor, text
    private int[] scrollPosition
    private int curX, curY, selectionWidth

    constructor Snapshot(Editor e, String text, int curX, int curY, int selectionWidth) {
        this.editor = e.editor
        this.text = e.text
        this.curX = e.curX
        this.curY = e.curY
        this.selectionWidth = e.selectionWidth
        // Usually, a memento stores all the editor's state.
        // For simplicity, we'll only store one field.
    }
    public restore() {
        editor.setText(text, curX, curY
        editor.setSelectionWidth(selectionWidth)
    }

// A command object can act as a caretaker. In that case, the
// command gets a memento before changing the editor's state.
// Originator's state data is required to execute the
// command properly.
class Command {
    private Editor editor; Snapshot backup;

    constructor Command(Editor e) {
        this.editor = e
        backup = editor.createSnapshot()
    }

    public void execute() {
        // ...
        backup = editor.createSnapshot()
        // ...
    }
```

Applicability

Use the Memento pattern when you want to produce snapshots of the object's state to be able to restore a previous state of the object.

The Memento pattern lets you make full copies of an object's state, including private fields, and store them separately from the object. While most people remember this pattern thanks to the "Undo" use case, it's also indispensable when dealing with transactions (i.e., if you need to roll back an operation on error).

Use the pattern when direct access to the object's fields/getters/setters violates its encapsulation.

The Memento makes the object itself responsible for creating a snapshot of its state. No other object can read the snapshot, making the original object's state data safe and secure.

How to Implement

1. Determine what class will play the role of the originator. It's important to know whether the program uses one central object of this type or multiple smaller ones.
2. Create the mementos class. Declare one, one class a set of fields that mirror the fields declared inside the originator class.
3. Make the mementos class immutable. A mementos should accept the data just once, via the constructor. The class should have no setters.
4. If your programming language supports nested classes, nest the mementos inside the originator. If not, extract a blank interface from the mementos class and make all other objects use it to refer to the mementos. You may add some metadata operations to the interface, but nothing that exposes the originator's state.
5. Add a method for producing mementos in the originator class. The originator should pass its state to the mementos via one or multiple arguments of the mementos's constructor.

   The return type of this method should be of the interface you remastered in the previous step (assuming that you remastered it at all). Under the hood, the mementos producing method should work directly with the mementos class.
6. Add a method for restoring the originator's state to its class. It should accept a mementos objects as an argument. If you remastered an interface in the previous step, make it the type of the parameter. In this case, you need to typecast the incoming object to the mementos class, since the originator needs full access to that object.
7. The caretaker, whether it represents a command object, a history, or something entirely different, should know when to request new mementos from the originator, how to store them and when to restore the originator with a particular memento.
8. The link between caretakers and originators may be revised into the mementos class. In this case, each mementos must be connected to the originator that built it. The remastered method should also cross to the mementos class. However, this would all make sense only if the mementos class is nested into originator or the originator class provides sufficient setters for controlling its state.

Pros and Cons

- ✓ You can produce snapshots of the object's state without violating its encapsulation.
- ✓ You can simplify the originator's code by letting the caretaker maintain the history of the originator's state.
- ✗ The app might consume lots of RAM if clients create mementos too frequently.
- ✗ Caretakers should track the originator's lifecycle to be able to destroy obsolete mementos.
- ✗ Most dynamic programming languages, such as PHP, Python and JavaScript, can't guarantee that the state within the mementos stays consistent.

Relations with Other Patterns

- You can use Command and Memento together when implementing "undo". In this case, commands are responsible for performing various operations over a target object, while mementos save the state of that object just before a command gets executed.
- You can use Memento along with Iterator to capture the current iteration state and roll it back if necessary.
- Sometimes Prototype can be a simpler alternative to Memento. This works if the object, the state of which you want to store in the history, is fairly straightforward and doesn't have links to external resources, or the links are easy to re-establish.

Code Examples

Support our free website and own the eBook!
• 1,700+ pages of in-depth guides to design patterns & clean code
• PDF, EPUB, and Kindle ready
• Lifetime updates
• 100 video lectures (teachers and students)
• 50 articles with code examples in 7 languages
• 30 interview preparation PROTOBUFS/GRPC Articles
Learn more

2021-2023 Refactoring.Guru. All rights reserved.
Illustrations by Mindy Z. Zhao
Refactoring.Guru, O. U. Ul Kaubamaru Maantee 1, Maardu, 75304
Email: support@refactoring.guru