

The Compound Without a Name

John Vlissides

Java Report, November 2000

© 2000 by John Vlissides. All rights reserved.

What's in a name? Plenty, especially when it's attached to a design pattern. Consider the following hypothetical monologue:

Had to redesign the framework to allow different cash flows, so I defined a set of classes, each encapsulating a different cash flow algorithm, and each deriving from the same abstract class, making their instances interchangeable at run-time. When developers found it hard to implement common cases, I added a class having a simplified interface to frequently used functionality that implements the defaults they wanted. Finally, I used a class that implements the ActiveX component interface in terms of OpenDoc's to incorporate that old OpenDoc component they're so enamored of.

Did you detect any patterns? If you did, you probably already know how to tighten up this description. I'll save you the trouble, but first let me substitute bogus pattern names for the real ones:

Had to redesign the framework to allow different cash flows, so I defined a Fred Flintstone for each. When developers found it hard to implement common cases, I added a whatchamacallit with the defaults they wanted. Finally, I used an EGRV to incorporate that old OpenDoc component they're so enamored of.

Sounds silly, sure. But change just four words, and suddenly everything clicks:

Had to redesign the framework to allow different cash flows, so I defined a strategy for each. When developers found it hard to implement common cases, I added a facade with the defaults they wanted. Finally, I used an adapter to incorporate that old OpenDoc component they're so enamored of.

I claim this makes at least a little sense to Jo OO programmer even if she doesn't know a pattern from a hole in the ground. Good pattern names do that, and I like to believe "Strategy," "Facade," and "Adapter" are good names—better than those substitutes, anyway.

No major aspect of *Design Patterns*¹ underwent as much churn as the pattern names. The pattern template evolved, as did the set of patterns, but neither proved as unstable as the names. You have to look no further than the original ECOOP '93 paper² to compare their stability. The template in that paper is identical to the final one except for the addition of the "Sample Code" section and the renaming of two others. Meanwhile, "Wrapper" became "Decorator," "Glue" became "Facade," "Solitaire" became "Singleton," and "Walker" became "Visitor." Changes to the template prompted a fixed set of changes to the catalog—23, to be exact. The name changes, in contrast, were pervasive on account of all the cross-references.

A lot more name shuffling went on *before* we wrote the paper. MEDIATOR was once called "Manager," and COMMAND was "Activity." PROXY was originally "Surrogate," a name it retains as an AKA. Ditto for FACTORY METHOD, nee "Virtual Constructor," and MEMENTO, nee "Token"—and "Snapshot," and "Cookie." Incidentally, if you think "Walker" is a lousy name for VISITOR, note that it was briefly dubbed—get this—"Rover." Maybe that should have been "Rotfl."

Several patterns didn't make the cut, and I suspect dubious naming had something to do with it. Try to imagine what the following patterns are about:

AVATAR, CONTROLLER, COURIER, DECIDER, DELEGATE, ENVOY,
FUNCTIONALITY, GLUON, LENS, MIMIC, OBJECT BUS

Some of those names are more evocative than others, but few are what you'd call self-explanatory. You could argue that it doesn't matter; people get used to whatever names are foisted upon them. Perhaps. Then again, I personally would have a hard time calling a pattern "Fred Flintstone" no matter how big and uncultivated it was.

Coming up with good names for compound patterns—patterns that document synergies among other patterns³—is as hard as naming conventional patterns, and possibly harder. One is tempted to skimp and merely reuse the compound's constituent pattern names, even though that's rarely the best choice. In fact, I can't think of a compound that didn't benefit from a distinctive name. "PLUGGABLE FACTORY"^{4,5} is a lot easier to swallow than "PROTOTYPE-ABSTRACT FACTORY." That goes double for "TOOLED COMPOSITE"⁶ versus "COMPOSITE-STATE-PROTOTYPE-VISITOR-COMMAND." I just wish I had an algorithm for deriving a nice name from a long-winded one.

Enough excuses. I confess to having no name for the following compound pattern beyond a perfunctory "MEMENTO-COMMAND." Rarely has the well of names run dry on me like this. At least the key constituents are clear; hopefully the rest will be, too.

Intent

Manage undo state when a command can affect unforeseen receivers.

Motivation

Suppose an undoable `ConcreteCommand` can affect several types of `Receiver` objects,⁷ some of which aren't known when the `ConcreteCommand` is defined. How do you obtain and manage the undo state of newly defined receivers without modifying the `ConcreteCommand`?

Consider for example a file browser that lets you change properties of files and directories (see Figure 1). All file system objects have basic properties such as read-only, archived, and so forth. Other properties are specific to certain file types. A shortcut (a.k.a. symbolic link or alias) maintains information about its target. An image may or may not be compressed. A word processor document may maintain a revision history. The file browser lets a user view and possibly change these properties.

Internally, the file browser represents files and directories as `Node` objects. `Node` is an interface; concrete classes implement different file and directory types. Property values are stored in member variables. `Node` declares `set` and `get` operations for the basic properties only, which are common to all nodes. `Node` also declares a `showProperties` operation for popping up a property dialog box. Concrete classes implement `showProperties` to produce a dialog box that's appropriate for the concrete type of node and its unique properties.

The browser encapsulates property-changing requests in instances of `PropertyCmd`, a `ConcreteCommand` class. When executed, `PropertyCmd` calls `showProperties` on the selected node(s). `PropertyCmd` is undoable, which implies that it can restore the original properties of any node when undone. How does `PropertyCmd` undo a property-change when it doesn't know anything specific about the `Node` implementers it affects?

The difficulty lies in the lack of file-specific property operations in the `Node` interface. There's no way to predict the properties that a `Node` implementer might harbor. If you can't predict the properties, then you can't provide an interface to them—and `PropertyCmd` must get and set those properties in the course of undoing and redoing.

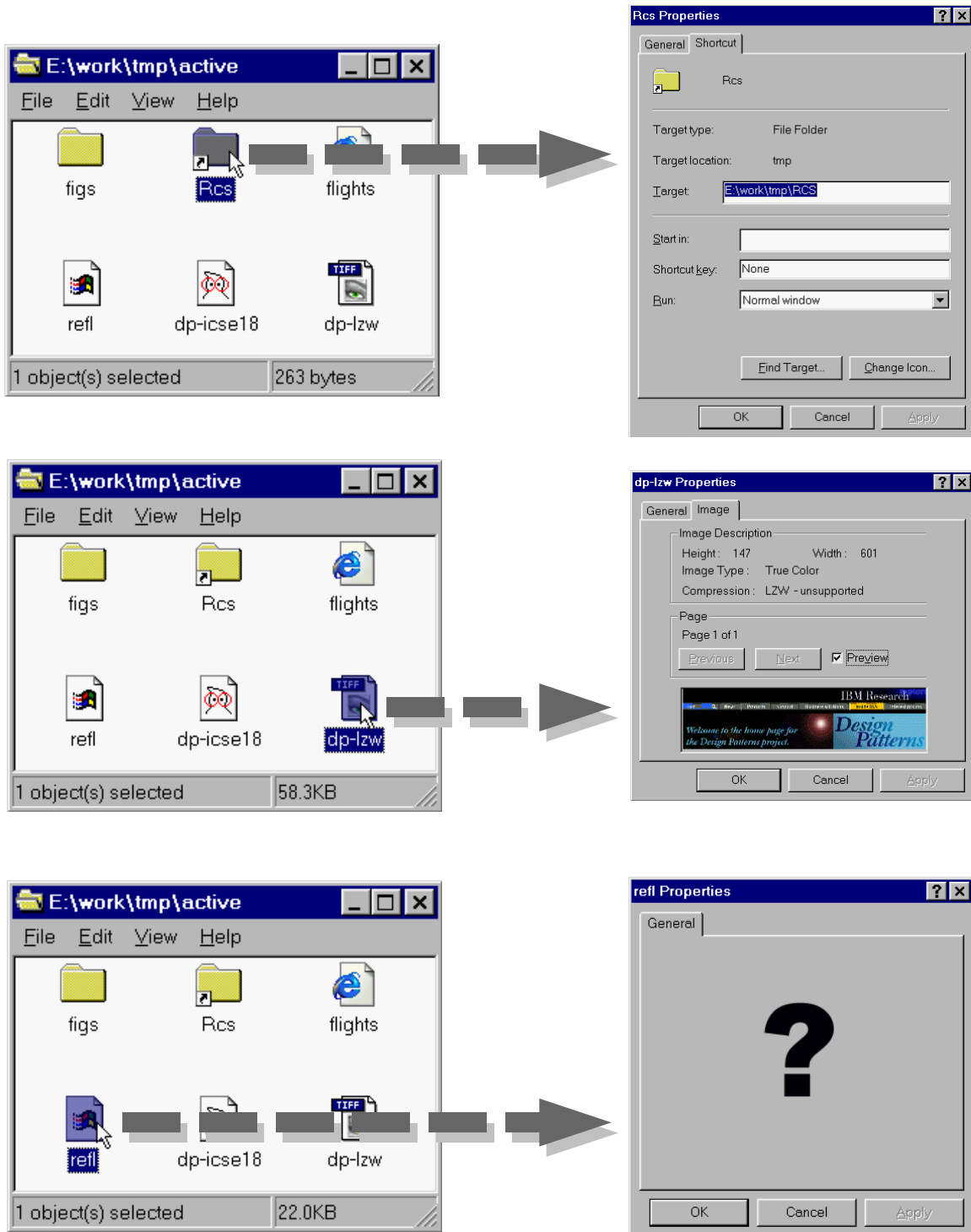


Figure 1: Properties of different file system objects

This problem meshes with the applicability of the MEMENTO pattern⁸:

- a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, *and*
- a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

The first bullet describes the state that `PropertyCmd` stores when undoing and redoing a property-change. The second bullet rules out a direct interface to obtaining the state, which fits the problem as well, but the motive here is a little different: a direct interface is lacking not to preserve encapsulation but because you can't anticipate the properties of `Node` implementers. The outcome, however, is the same—no direct interface to state. With both applicability criteria met, MEMENTO offers a solution.

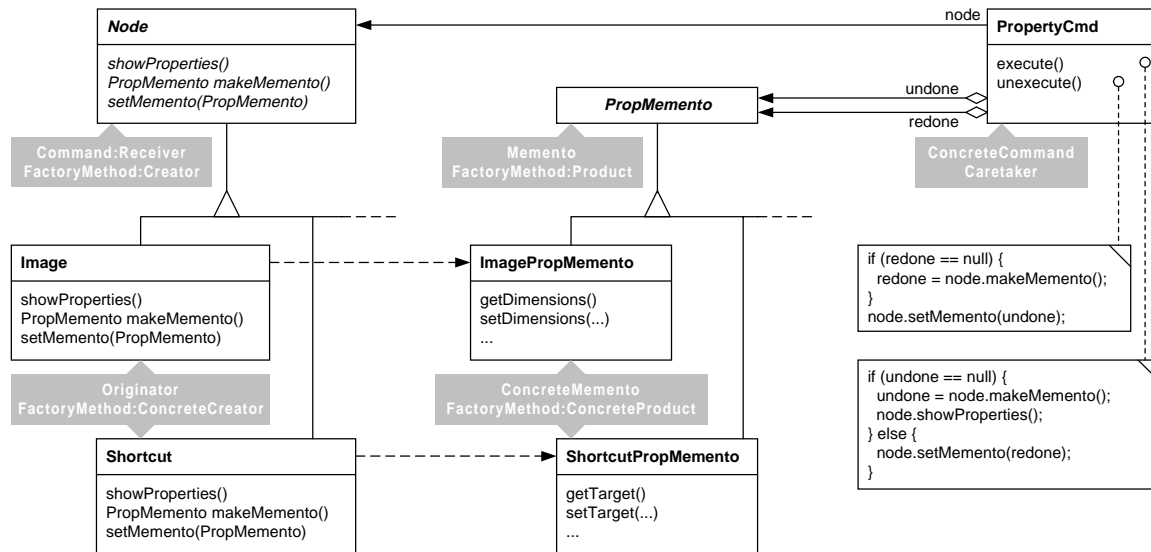


Figure 2: MEMENTO-based design

Figure 2 illustrates an application of MEMENTO to the design. `Node` implementers such as `Image` and `Shortcut` store their unique properties in corresponding implementers of `PropMemento`. `Node` defines a `makeMemento` factory method⁹ and a `setMemento` abstract operation; `Node` implementers make these operations act as `Originators`. `setMemento` implementations must downcast their argument to the appropriate `ConcreteMemento` type.* `PropertyCmd` acts as `Caretaker` of *two* mementos: one for the original properties (`undone`) and another for the modified properties (`redone`). The command initializes `undone` the first time it's executed, and it initializes `redone` the first time it's *unexecuted*. Thereafter it sets the originator (`node`) to one or the other memento, depending on whether it's being undone or redone.

A new type of file system object requires a new implementer of `Node` and a corresponding implementer of `PropMemento`. Existing interfaces and classes remain unaffected.

* This implementation exemplifies the type-laundering variant of MEMENTO described in *Pattern Hatching*.¹⁰

Applicability

Use MEMENTO–COMMAND when *all* of the following are true:

- You're using the COMMAND pattern to implement undoable commands.
- The Receiver interface is sufficient to implement non-undoable ConcreteCommands.
- You cannot anticipate a Receiver's interface to getting and setting undo state, or no such interface is common among Receiver classes.

Structure

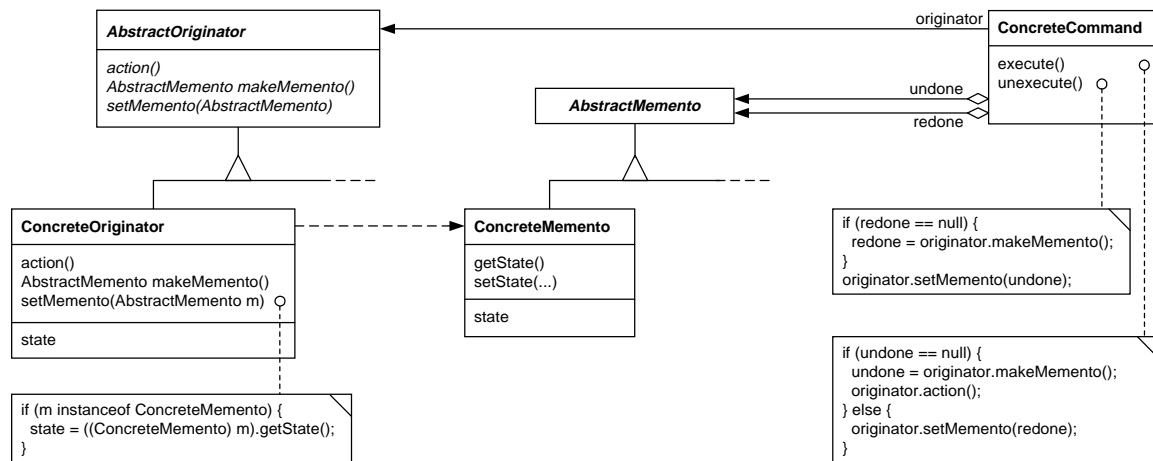


Figure 3: MEMENTO–COMMAND structure

Participants

AbstractMemento (PropMemento)

- defines the narrow Memento interface of the MEMENTO pattern.
- plays the Product role in the FACTORY METHOD pattern.

ConcreteMemento (ImagePropMemento, ShortcutPropMemento)

- defines the wide Memento interface of the MEMENTO pattern.
- plays the ConcreteProduct role in the FACTORY METHOD pattern.

AbstractOriginator (Node)

- plays the role of Receiver in the COMMAND pattern, providing an interface (e.g., `action`) for `ConcreteCommand` to use in carrying out a request. The interface includes any state-setting and -getting operations for undoable state that is common to all `ConcreteOriginator`s.
- plays the role of Creator in the FACTORY METHOD pattern, declaring a factory method (e.g., `makeMemento`) for creating mementos.
- declares an abstract operation (e.g., `setMemento`) for reverting to a memento's state.

ConcreteOriginator (Image, Shortcut)

- implements the interface that ConcreteCommand uses to carry out a request.
- plays the Originator role in the MEMENTO pattern and the ConcreteCreator role in the FACTORY METHOD pattern, implementing the factory method for creating mementos.
- implements the abstract operation for reverting to a memento's state.

ConcreteCommand (PropertyCmd)

- plays the Caretaker role in the MEMENTO pattern and the ConcreteCommand role in the COMMAND pattern.
- aggregates two mementos, one for its AbstractOriginator's undone state (`undone`) and another for its redone state (`redone`).

Collaborations

- ConcreteCommand initializes `undone` and `redone` on the first call to `execute` and `unexecute`, respectively. Before carrying out the initial request, ConcreteCommand obtains a memento from the originator and assigns it to `undone`. Before the first undo, ConcreteCommand asks the originator for another memento and assigns it to `redone`.
- On `unexecute`, ConcreteCommand sets the originator's state to `undone`. Subsequent calls to `execute` merely set the originator's state to `redone`.
- ConcreteOriginator's `setMemento` downcasts its argument to the ConcreteMemento class that ConcreteOriginator instantiates in `makeMemento`.

Consequences

MEMENTO-COMMAND has two consequences beyond those of its constituent patterns:

1. *ConcreteOriginators can be introduced without affecting existing ConcreteCommands.* The COMMAND pattern couples ConcreteCommand to Receiver. Undoability increases this coupling. As a result, changing a Receiver will likely force changes to ConcreteCommands.

MEMENTO-COMMAND avoids the added coupling of undo. A ConcreteCommand can work with unforeseen ConcreteOriginator classes, including undoing and redoing its effects on them, without modification.

2. *Downcasting is required, jeopardizing type safety.* Narrow and wide Memento interfaces are implemented by splitting Memento into abstract and concrete participants, the latter defining the wide interface. Hence `setMemento` must downcast its argument to the wide interface—admitting runtime errors that a conventional MEMENTO implementation would catch at compile-time.

Implementation

Consider the following three implementation issues in addition to those of MEMENTO and COMMAND.

1. *Downcasting versus friend.* MEMENTO recommends using `friend` in C++ to give the originator exclusive access to a memento's internals. Here, however, we use type laundering: AbstractMemento effectively hides implementer-specific accessors from ConcreteCommand. A downcast from AbstractMemento to ConcreteMemento is the key that unlocks the wide interface.

Type laundering is preferable to using `friend` for two reasons. First, type laundering is not C++-specific, making it applicable to Java, for example. Second, because friendship is not inherited, it's likely that subclassing a ConcreteOriginator will force you to subclass a ConcreteMemento as well.

The type-laundering approach lets you extend `ConcreteOriginators` and `ConcreteMementos` independently.

2. *Storing one memento in `ConcreteCommand` versus two.* `ConcreteCommand` might get by with just one undone memento. Initially, `execute` would work as shown in Figure 3. But instead of calling `setMemento` on subsequent executions, `execute` omits the memento initialization and simply calls `action`. Meanwhile, `unexecute` omits the test on `redone`, calling `originator.setMemento(undone)` every time.

This approach makes `undo` more efficient, since half as many mementos are created and stored. But note that `redo` would exhibit *precisely* the same behavior as the initial operation, which isn't always desirable. In the file browser case, a `redo` would pop up a property dialog box again instead of silently reapplying the property changes.

You could address this problem by changing `execute`'s behavior to call something other than `action` on the originator. But that may pose another problem: `AbstractOriginator`'s interface could need extension to support redo semantics. In the file browser, for example, `execute` can't call `showProperties` again, because that would pop up the dialog box. `Node` must therefore be augmented with state-setting operations that can restore the properties silently. Of course, that presumes we can foresee such operations and put them in the `Node` base class—which isn't possible given `MEMENTO-COMMAND`'s intent.

The second memento (`redone`) gives us a way of redoing the command with no added interface.

3. *Setting a memento's state.* Mementos are usually immutable—once created, they never change. In most languages, therefore, it's enough to provide `get` operations without corresponding `set` operations. The originator sets the state once, in the `ConcreteMemento` constructor.

Sample code

Here are Java declarations for key classes in the Motivation section. `Node` declares the interface for file system objects, and `Shortcut` implements `Node` for symbolic links.

```
interface Node {
    void showProperties();

    PropMemento makeMemento();
    void setMemento(PropMemento mem);

    // file system-related interface...
}

class Shortcut implements Node {
    public Shortcut (Node target)           { ... }

    public void showProperties ()           { ... }

    public PropMemento makeMemento ()       { ... }
    public void setMemento (PropMemento mem) { ... }

    public Node getTarget ()                { return _target; }

    // file system-related interface...

    private Node    _target;
    private String  _home;
    private String  _key;
    private Image   _icon;
    private Mode    _mode;
}
```

The private members store the properties that are specific to shortcuts. As such, they aren't accessible through the `Node` interface.

PropMemento defines the narrow AbstractMemento interface, which amounts to a marker interface:

```
interface PropMemento { }
```

ShortcutPropMemento implements the wide ConcreteMemento interface. It provides accessors for state that's unique to shortcuts. This implementation defines getters but not setters under the assumption that the memento is immutable.

```
class ShortcutPropMemento implements PropMemento {
    public ShortcutPropMemento(
        Node target, string home, string key,
        Image icon, Mode mode
    ) { ... }

    public Node getTarget () { return _target; }
    public string getHome () { return _home; }
    public string getKey () { return _key; }
    public Image getIcon () { return _icon; }
    public Mode getMode () { return _mode; }

    private Node _target;
    private string _home;
    private string _key;
    private Image _icon;
    private Mode _mode;
}
```

ShortcutPropMemento's constructor merely initializes the member variables with its arguments:

```
public ShortcutPropMemento (
    Node target, string home, string key,
    Image icon, Mode mode
) {
    _target = target;
    _home = home;
    _key = key;
    _icon = icon;
    _mode = mode;
}
```

Shortcut.makeMemento instantiates ShortcutPropMemento with the shortcut's properties...

```
public PropMemento makeMemento () {
    return new ShortcutPropMemento(
        _target, _home, _key, _icon, _mode
    );
}
```

...and Shortcut.setMemento extracts the properties from its argument after downcasting it to the right type. In this implementation, setMemento throws an exception should the downcast fail:

```
public void setMemento (PropMemento mem) {
    if (mem instanceof ShortcutPropMemento) {
        ShortcutPropMemento spMem = (ShortcutPropMemento) mem;

        _target = spMem.getTarget();
        _home = spMem.getHome();
        _key = spMem.getKey();
        _icon = spMem.getIcon();
        _mode = spMem.getMode();
    } else {
        throw new SetMementoException(mem, this);
    }
}
```

SetMementoException keeps a record of the incompatible memento and the originator that refused it. A client like PropertyCmd may catch this exception for debugging purposes:

```
class PropertyCmd implements Command {
    // ...
    public void execute () {
        if (_undone == null) {
            _undone = _node.makeMemento();
            _node.showProperties();
        } else {
            try {
                node.setMemento(_redone);
            } catch (SetMementoException e) {
                System.err.println("PropertyCmd.execute: " + e.message());
            }
        }
    }

    public void unexecute () {
        if (_redone == null) {
            _redone = _node.makeMemento();
        }
        try {
            node.setMemento(_undone);
        } catch (SetMementoException e) {
            System.err.println("PropertyCmd.unexecute: " + e.message());
        }
    }
    // ...
}
```

Known uses

One last confession: This is a “proto-pattern,” meaning I don’t have enough known uses to brand it a full-fledged pattern. The good news is that publishing proto-patterns gives people a chance to try them out and give their authors feedback, particularly with regard to known uses. The bad news is, there might not *be* any known uses, especially if the problems and/or solutions described are unique—or just plain bogus. Please help me prove otherwise.

Related patterns

Buschmann, et al.’s COMMAND PROCESSOR¹¹ describes the scaffolding COMMAND needs to support full multilevel undo and redo.

Acknowledgments

Erich Gamma and Dirk Riehle offered helpful feedback. John Lakos suggested “Fred Flintstone” as a pattern name almost ten years ago, in a conversation I’m sure he’s forgotten.

References

- ¹ Gamma, E., et al. *Design Patterns*, Addison–Wesley, Reading, MA, 1995.
- ² Gamma, E., et al. Design patterns: Abstraction and reuse of object-oriented design. *Proceedings of the 7th European Conference on Object-Oriented Programming*, Kaiserslautern, Germany, July 1993, pp. 406–431. <http://www.research.ibm.com/designpatterns/pubs/dp-ecoop93.ps.gz>
- ³ Vlissides, J. Composite design patterns (they aren't what you think). *C++ Report*, June 1998, pp. 45–47, 53. <http://www.research.ibm.com/designpatterns/pubs/ph-jun98.pdf>
- ⁴ Vlissides, J. PLUGGABLE FACTORY, part I. *C++ Report*, November/December 1998, pp. 52–56, 68. <http://www.research.ibm.com/designpatterns/pubs/ph-nov-dec98.pdf>
- ⁵ Vlissides, J. PLUGGABLE FACTORY, part II. *C++ Report*, February 1999, pp. 51–57. <http://www.research.ibm.com/designpatterns/pubs/ph-feb99.pdf>
- ⁶ Vlissides, J. TOOLED COMPOSITE. *C++ Report*, September 1999, pp. 43–47. <http://www.research.ibm.com/designpatterns/pubs/ph-sep99.pdf>
- ⁷ *Design Patterns*, pp. 233–242.
- ⁸ *Design Patterns*, pp. 283–291.
- ⁹ *Design Patterns*, pp. 107–116.
- ¹⁰ Vlissides, J. *Pattern Hatching*, Addison–Wesley, Reading, MA, 1998, pp. 105–110.
- ¹¹ Buschmann, F., et al. *Pattern-Oriented Software Architecture—A System of Patterns*. J. Wiley & Sons, Chichester, England, 1996, pp. 277–290.