CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2001
Doc 11 Memento, Command, Command Processor
Contents

Memento 2
  Structure 2
  Applicability 3
  Consequences/ Implementation 7
  Iterators & Mementos 11
Command....12
  Structure 12
  When to Use the Command Pattern 13
  Consequences 14
Command Processor....22
  Structure 23
  Consequences 24

References

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 233-242, Command Pattern, pp. 283-291

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, Addision-Wesley, 1998, pp. 297-304, 245-260

Pattern-Oriented Software: A System of Patterns, Buschman, Meunier, Rohnert, Sommerlad, Stal, 1996, pp. 277-290, Command Processor

Command Processor, Sommerlad in Pattern Languages of Program Design 2, Eds. Vlissides, Coplien, Kerth, Addison-Wesley, 1996, pp. 63-74

Copyright ©, All rights reserved. 2001 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.

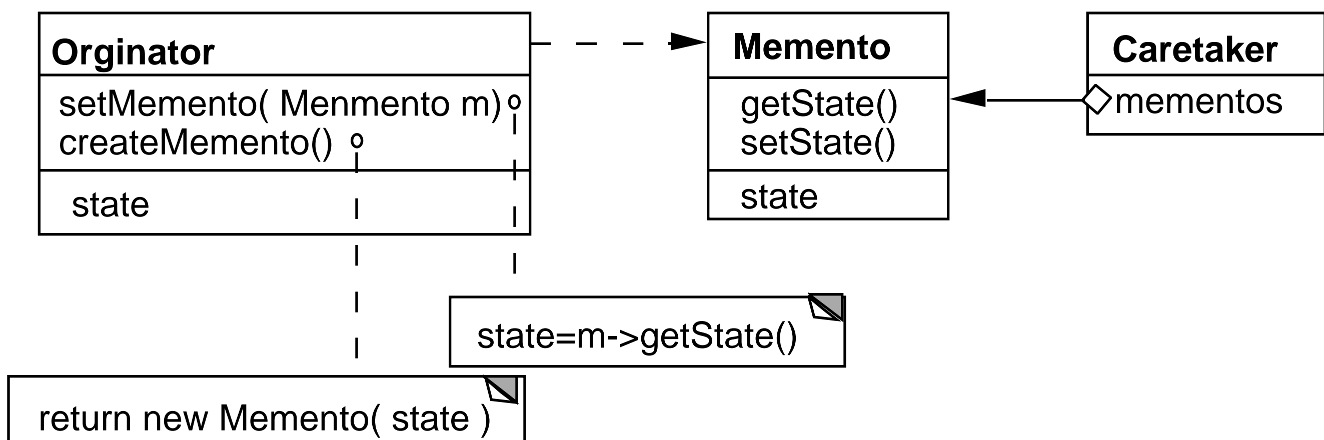
Memento

Store an object's internal state, so the object can be restored to this state later without violating encapsulation

Motivation

Allow undo, rollbacks, etc.

Structure



Only originator:

- Can access Memento’s get/set state methods
- Create Memento

Applicability

Use when you:

- Need to save all or part of the state of an object and
- Do not wish to expose the saved state to the outside world

An Example

```
package Examples;
class Memento
   {
   private Hashtable savedState = new Hashtable();

   protected Memento() {}; //Give some protection

   protected void setState( String stateName, Object stateValue )
      {
      savedState.put( stateName, stateValue );
      }

   protected Object getState( String stateName)
      {
      return savedState.get( stateName);
      }

   protected Object getState(String stateName, Object defaultValue )
      {
      if ( savedState.containsKey( stateName ) )
         return savedState.get( stateName);
      else
         return defaultValue;
      }
   }
```

A Class whose state is Saved

```
package Examples;
class ComplexObject
   {
   private String name;
   private int someData;
   private Vector objectAsState = new Vector();

   public Memento createMemento()
      {
      Memento currentState = new Memento();
      currentState.setState( "name", name );
      currentState.setState( "someData", new Integer(someData) );
      currentState.setState( "objectAsState", objectAsState.clone() );
      return currentState;
      }

   public void restoreState( Memento oldState)
      {
      name = (String) oldState.getState( "name", name );
      objectAsState = (Vector) oldState.getState( "objectAsState" );
      Integer data = (Integer) oldState.getState( "someData");
      someData = data.intValue();
      }
```

```
// Show a way to do incremental saves
public Memento setName( String aName )
  {
  Memento deltaState = saveAState( "name", name);
  name = aName;
  return deltaState;
  }


public void setSomeData( int value )
  {
  someData = value;
  }

private Memento saveAState(String stateName, Object
stateValue)
  {
  Memento currentState = new Memento();
  currentState.setState( stateName, stateValue );
  return currentState;
  }
}
```

Consequences/ Implementation

Simplifies Originator

You may be tempted to let the originator manage its state history

This adds to the complexity of the Originator

How to store state history and for how long?

Using Mementos might be expensive

Copying state takes time and space

If this takes too much time/space pattern may not be appropriate

Preserve encapsulation boundaries

Give Memento two interfaces: wide and narrow

Let originator have access to all set/get/state of Memento

Let others only hold Mementos and destroy them

Defining Narrow and Wide Interfaces

C++

Make Memento's interface private

Make Originator a friend of the Memento

```
Class Memento {
public:
   virtual ~Memento();
private:
   friend class Originator;
   Memento();
   void setState(State*);
   State* GetState();
   …
```

Java[1]

Use private nested/inner class to hide memento's interface

```
class ComplexObject {
   private String name;
   private int someData;

   public Memento createMemento() {
      return new Memento();
   }

   public void  restoreState( Memento oldState) {
      oldState.restoreStateTo( this );
   }

   public class Memento {
      private String savedName;
      private int savedSomeData;

      private Memento() {
         savedName = name;
         savedSomeData = someData;
      }

      private void restoreStateTo(ComplexObject target) {
         target.name = savedName;
         target.someData = savedSomeData;
      }
   }
}
```

[1] RestoreStateTo does not access the fields of the outer object in case one wants to restore the state to a different ComplexObject object. One may wish to use an nested class to avoid tangling the memento to the outer object

Using Clone to Save State

One can wrap a clone of the Originator in a Memento or

Just return the clone as a type with no methods

```
interface Memento extends Cloneable { }

class ComplexObject implements Memento {
   private String name;
   private int someData;

   public Memento createMemento() {
      Memento myState = null;
      try {
         myState =  (Memento) this.clone();
      }
      catch (CloneNotSupportedException notReachable) {
      }
      return myState;
   }

   public void  restoreState( Memento savedState) {
      ComplexObject myNewState = (ComplexObject)savedState;
      name = myNewState.name;
      someData = myNewState.someData;
   }
}
```

Iterators & Mementos

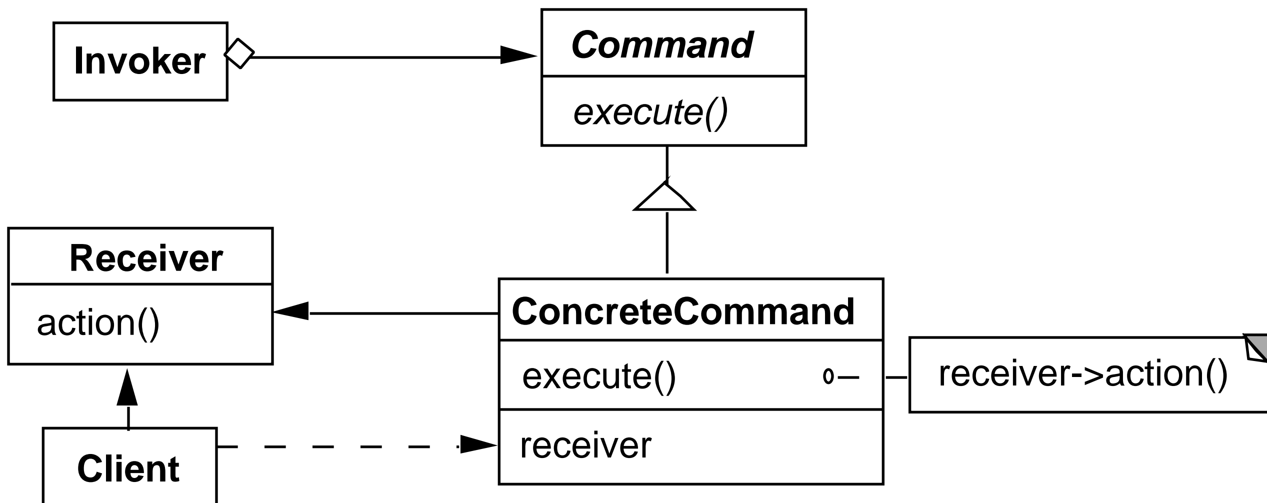
Using a Momento we can allow multiple concurrent iterations

```
class IteratorState {
   int currentPosition = 0;

   protected IteratorState() {}

   protected int getPosition() {return currentPosition;    }

   protected void advancePosition() { currentPosition++; }
   }

class Vector {
   protected Object elementData[];
   protected int elementCount;

   public IteratorState newIteration() { return new IteratorState(); }

   public boolean hasMoreElements(IteratorState aState) {
      return aState.getPosition() < elementCount;
   }

   public Object nextElement( IteratorState aState ) {
      if (hasMoreElements( aState ) ) {
         int currentPosition = aState.getPosition();
         aState.advancePosition();
         return elementData[currentPosition];
         }
   throw new NoSuchElementException("VectorIterator");
   }
   …
```

Command

Encapsulates a request as an object

Structure



Example

Let

Invoker be a menu
Client be a word processing program
Receiver a document
Action be save

When to Use the Command Pattern

- When you need an action as a parameter
  Commands replace callback functions

- When you need to specify, queue, and execute requests at different times

- When you need to support undo

- When you need to support logging changes

- When you structure a system around high-level operations built on primitive operations

  A Transactions encapsulates a set of changes to data

  Systems that use transaction often can use the command pattern

- When you need to support a macro language

Consequences

Command decouples the object that invokes the operation from the one that knows how to perform it

It is easy to add new commands, because you do not have to change existing classes

You can assemble commands into a composite object

Example - Menu Callbacks

```
abstract class Command
   {
   abstract public void execute();
   }

class OpenCommand extends Command
   {
   private Application opener;

   public OpenCommand( Application theOpener )
      {
      opener = theOpener;
      }

   public void execute()
      {
      String documentName = AskUserSomeHow();

      if ( name != null )
         {
         Document toOpen =
               new Document( documentName );
         opener.add( toOpen );
         opener.open();
         }
      }
   }
```

Using Command

```
class Menu
  {
  private Hashtable menuActions = new Hashtable();

  public void addMenuItem( String displayString,
            Command itemAction )
    {
    menuActions.put( displayString, itemAction );
    }

  public void handleEvent( String itemSelected )
    {
    Command runMe;
    runMe = (Command) menuActions.get( itemSelected );
    runMe.execute();
    }

  // lots of stuff missing
  }
```

MacroCommand

```
class MacroCommand extends Command
   {
   private Vector commands = new Vector();

   public void add( Command toAdd )
      {
      commands.addElement( toAdd );
      }

   public void remove( Command toRemove )
      {
      commands.removeElement( toAdd );
      }

   public void execute()
      {
      Enumeration commandList = commands.elements();

      while ( commandList.hasMoreElements() )
         {
         Command nextCommand;
         nextCommand = (Command)
                  commandList.nextElement();
         nextCommand.execute();
         }
      }
   }
```

Pluggable Commands

Using reflection it is possible to create one general Command

```
import java.util.*;
import java.lang.reflect.*;

public class Command
   {
   private Object receiver;
   private Method command;
   private Object[] arguments;

   public Command(Object receiver, Method command,
                        Object[] arguments )
      {
      this.receiver = receiver;
      this.command = command;
      this.arguments = arguments;
      }
   public void execute() throws InvocationTargetException,
                                    IllegalAccessException
      {
      command.invoke( receiver, arguments );
      }
   }
```

Using the Pluggable Command

One does have to be careful with the primitive types

```
public class Test {
   public static void main(String[] args) throws Exception
      {
      Vector sample = new Vector();
      Class[] argumentTypes = { Object.class };
      Method add =
         Vector.class.getMethod( "addElement", argumentTypes);
      Object[] arguments = { "cat" };

      Command test = new Command(sample, add, arguments );
      test.execute();
      System.out.println( sample.elementAt( 0));
      }
   }
```

Output

cat

Pluggable Command Smalltalk Version

```
Object subclass: #PluggableCommand
  instanceVariableNames: 'receiver selector arguments '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Whitney-Examples'
```

Class Methods

```
receiver: anObject selector: aSymbol arguments: anArrayOrNil
  ^super new
    setReceiver: anObject
    selector: aSymbol
    arguments: anArrayOrNil
```

Instance Methods

```
setReceiver: anObject selector: aSymbol arguments: anArrayOrNil
  receiver := anObject.
  selector := aSymbol.
  arguments := anArrayOrNil isNil
        ifTrue:[#( )]
        ifFalse: [anArrayOrNil]

execute
  ^receiver
    perform: selector
    withArguments: arguments
```

Using the Pluggable Command

```
| sample command |
sample := OrderedCollection new.
command := PluggableCommand
	receiver: sample
	selector: #add:
	arguments: #( 5 ).
command execute.
^sample at: 1
```

Command Processor

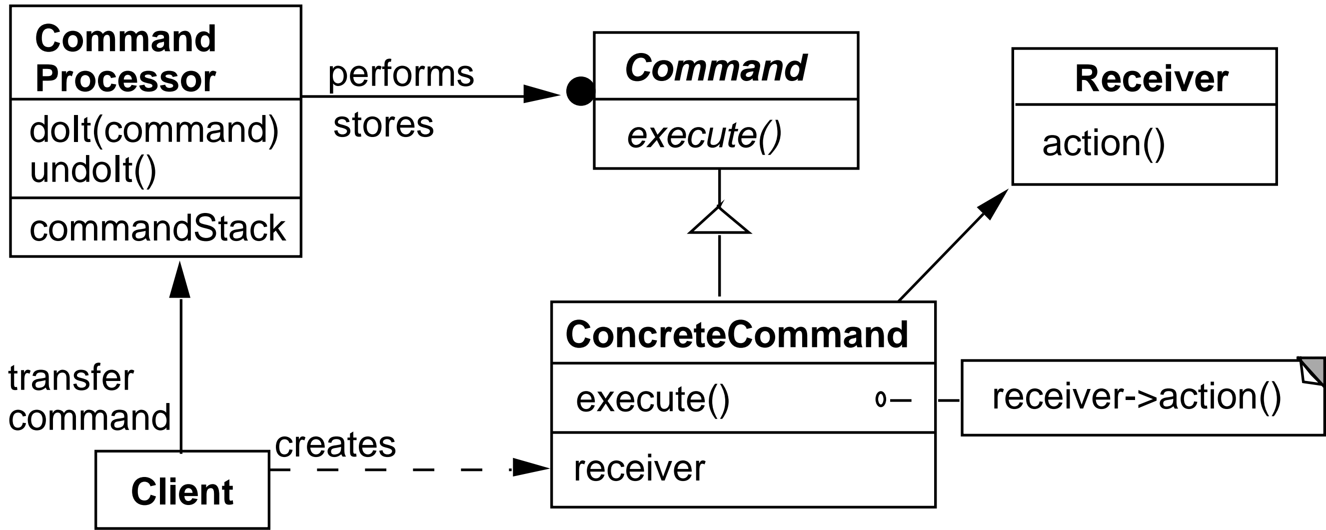
Similar to the command pattern

Command Processor manages the command objects

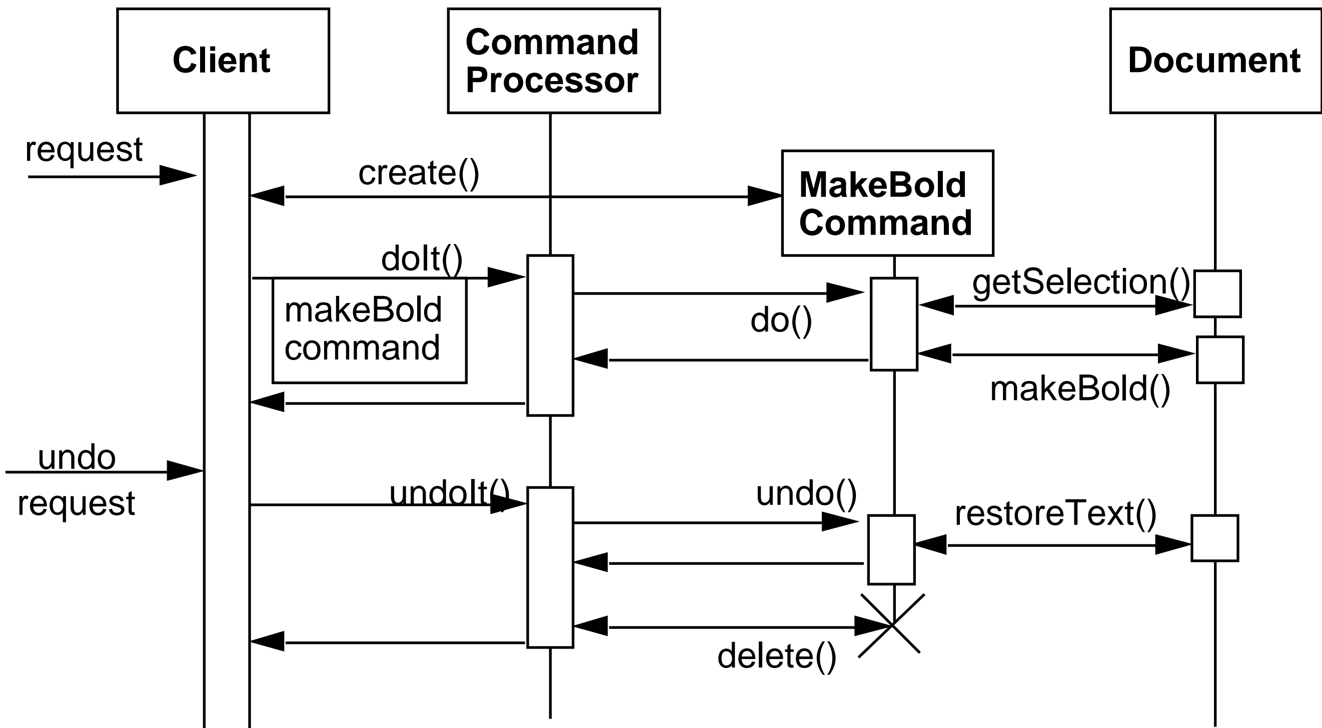
The command processor:

- Contains all command objects
- Schedules the execution of commands
- May store the commands for later unto
- May log the sequence of commands for testing purposes
- Uses singleton to insure only one instance

Structure



Dynamics



Consequences
Benefits

- Flexibility in the way requests are activated

  Different user interface elements can generate the same kind of command object

  Allows the user to configure commands performed by a user interface element

- Flexibility in the number and functionality of requests

  Adding new commands and providing for a macro language comes easy

- Programming execution-related services

  Commands can be stored for later replay
  Commands can be logged
  Commands can be rolled back

- Testability at application level

- Concurrency

  Allows for the execution of commands in separate threads

Liabilities

- Efficiency loss

- Potential for an excessive number of command classes

  Try reducing the number of command classes by:

    Grouping commands around abstractions

    Unifying simple commands classes by passing the receiver object as a parameter

- Complexity

  How do commands get additional parameters they need?