CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2004
Doc 12 Memento, Prototype & Abstract Factory

Contents

Memento ................................................................................................................................. 2
Structure ................................................................................................................................. 2
Applicability .......................................................................................................................... 3
Consequences/ Implementation ............................................................................................ 7
Iterators & Mementos ............................................................................................................ 11
Prototype .............................................................................................................................. 12
Intent .................................................................................................................................... 12
Applicability ........................................................................................................................ 12
  Copying Issues .................................................................................................................. 14
  Shallow Copy Verse Deep Copy ...................................................................................... 14
Cloning Issues ...................................................................................................................... 17
Consequences ...................................................................................................................... 20
Implementation Issues ......................................................................................................... 20
Abstract Factory .................................................................................................................. 21
  How Do Factories create Widgets? ................................................................................ 26
    Method 1) My Factory Method ..................................................................................... 26
    Method 2) Their Factory Method ................................................................................ 27
    Method 2.5) Subclass returns Class ........................................................................... 29
    Method 3) Prototype .................................................................................................... 30
Applicability ........................................................................................................................ 31

References

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 283-292, 117-126, 87-96


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

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 Memento 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");
    }
    ...

Prototype
Intent

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Applicability

Use the Prototype pattern when

• A system should be independent of how its products are created, composed, and represented; and

• when the classes to instantiate are specified at run-time; or

• to avoid building a class hierarchy of factories that parallels the class hierarchy of products; or

• when instances of a class can have one of only a few different combinations of state.

•

It may be easier to have the proper number of prototypes and clone them rather than instantiating the class manually each time.
Insurance Example

Insurance agents start with a standard policy and customize it

Two basic strategies:

• Copy the original and edit the copy

• Store only the differences between original and the customize version in a decorator
Copying Issues
Shallow Copy Verse Deep Copy

Original Objects

Shallow Copy
Shallow Copy Verse Deep Copy

Original Objects

Deep Copy
Shallow Copy Verse Deep Copy

Original Objects

```
<table>
<thead>
<tr>
<th>aDoor</th>
<th>aRoom</th>
<th>aRoom</th>
<th>aChair</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>room1</td>
<td>room2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Deeper Copy

```
<table>
<thead>
<tr>
<th>aDoor</th>
<th>aRoom</th>
<th>aRoom</th>
<th>aChair</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>room1</td>
<td>room2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Cloning Issues
How to in C++ - Copy Constructors

class Door
{
    public:
        Door();
        Door( const Door&);

        virtual Door* clone() const;

        virtual void Initialize( Room*, Room* );
        // stuff not shown

    private:
        Room* room1;
        Room* room2;
}

Door::Door ( const Door& other ) //Copy constructor
{
    room1 = other.room1;
    room2 = other.room2;
}

Door* Door::clone() const
{
    return new Door( *this );
}
How to in Java - Object clone()

protected Object clone() throws CloneNotSupportedException

    Default is shallow copy

Returns: A clone of this Object.

Throws: OutOfMemoryError
    If there is not enough memory.

Throws: CloneNotSupportedException
    Object explicitly does not want to be cloned, or it does not support the Cloneable interface.

class Door implements Cloneable {
    public void Initialize( Room a, Room b)
        { room1 = a; room2 = b; }

    public Object clone() throws
            CloneNotSupportedException {
            // modify this method for deep copy
            // no need to implement this method for shallow copy
            return super.clone();
        }
    Room room1;
    Room room2;
}
VisualWorks Smalltalk

Object>>shallowCopy
   Does a shallowCopy of the receiver

Object>>copy
   ^self shallowCopy postCopy

   “Template method for copy”

Copy is the primary method for copying an object

Classes override postCopy to do more than shallow copy

Smalltalk.CS635 defineClass: #Door
   superclass: #{Core.Object}
   indexedType: #none
   private: false
   instanceVariableNames: 'room1 room2 '

postCopy
   room1 := room1 copy.
   room2 := room2 copy.
Consequences

• Adding and removing products at run-time
• Specifying new objects by varying values
• Specifying new objects by varying structure
• Reducing subclassing (from factory method)
• Configuring an application with classes dynamically

Implementation Issues

• Using a prototype manager
• Implementing the Clone operation
• Initializing clones
Abstract Factory

Task - Write a cross platform window toolkit

GUI interfaces to run on
• Mac, PC & Unix
• Use the look and feel of each platform

We will look at widgets: Windows, Menu's and Buttons
Create

- An interface (or abstract class) for each widget
- A concrete class for each platform:

```
Window
  close, hide
  show, scroll
  resize

MacWindow
  close, hide
  show, scroll
  resize

Win95 Window
  close, hide
  show, scroll
  resize

MotifWindow
  close, hide
  show, scroll
  resize

Menu
  menu ops

MacMenu
  menu ops

Win95Menu
  menu ops

MotifMenu
  menu ops
```
This allows the application to write to the widget interface

```java
public void installDisneyMenu()
{
    Menu disney = create a menu somehow
    disney.addItem( "Disney World" );
    disney.addItem( "Donald Duck" );
    disney.addItem( "Mickey Mouse" );
    disney.addGrayBar( );
    disney.addItem( "Minnie Mouse" );
    disney.addItem( "Pluto" );
    etc.
}
```

How to create the widget so

- We get the correct interface widgets
- Minimize places in code that know the platform
Use Abstract Factory

abstract class WidgetFactory
{
    public Window createWindow();
    public Menu createMenu();
    public Button createButton();
}

class MacWidgetFactory extends WidgetFactory
{
    public Window createWindow()
    {
        code to create a mac window
    }

    public Menu createMenu()
    {
        code to create a mac Menu
    }

    public Button createButton()
    {
        code to create a mac button
    }
}

class Win95WidgetFactory extends WidgetFactory
{
    public Window createWindow()
    {
        code to create a Win95 window
    }

    public Menu createMenu()
    {
        code to create a Win95 Menu
    }

    public Button createButton()
    {
        code to create a Win95 button
    }
}
Now to get code that works for all platforms we get:

```java
public void installDisneyMenu(WidgetFactory myFactory)
{
    Menu disney = myFactory.createMenu();
    disney.addItem( "Disney World" );
    disney.addItem( "Donald Duck" );
    disney.addItem( "Mickey Mouse" );
    disney.addGrayBar( );
    disney.addItem( "Minnie Mouse" );
    disney.addItem( "Pluto" );
    etc.
}
```

We just need to make sure that the application for each platform creates the proper factory.
How Do Factories create Widgets?
Method 1) My Factory Method

abstract class WidgetFactory
{
    public Window createWindow();
    public Menu createMenu();
    public Button createButton();
}

class MacWidgetFactory extends WidgetFactory
{
    public Window createWindow()
    { return new MacWidow() }

    public Menu createMenu()
    { return new MacMenu() }

    public Button createButton()
    { return new MacButton() }
}
How Do Factories create Widgets?
Method 2) Their Factory Method

abstract class WidgetFactory {
    private Window windowFactory;
    private Menu menuFactory;
    private Button buttonFactory;

    public Window createWindow()
    { return windowFactory.createWindow() }

    public Menu createMenu();
    { return menuFactory.createMenu() }

    public Button createButton()
    { return buttonFactory.createMenu() }
}

class MacWidgetFactory extends WidgetFactory {
    public MacWidgetFactory() {
        windowFactory = new MacWindow();
        menuFactory = new MacMenu();
        buttonFactory = new MacButton();
    }
}

class MacWindow extends Window {
    public Window createWindow() { blah }
    etc.
Method 2) Their Factory Method

When does this make Sense?

There might be more than one way to create a widget

abstract class WidgetFactory {
    private Window windowFactory;
    private Menu menuFactory;
    private Button buttonFactory;

    public Window createWindow()
        { return windowFactory.createWindow() }

    public Window createWindow( Rectangle size)
        { return windowFactory.createWindow( size ) }

    public Window createWindow( Rectangle size, String title)
        { return windowFactory.createWindow( size, title) }

    public Window createFancyWindow()
        { return windowFactory.createFancyWindow() }

    public Window createPlainWindow()
        { return windowFactory.createPlainWindow() }

Using factory method allows abstract class to do all the different ways to create a window.

Subclasses just provide the objects windowFactory, menuFactory, buttonFactory, etc.
How Do Factories create Widgets?
Method 2.5) Subclass returns Class

abstract class WidgetFactory {

    public Window createWindow() {
        return windowClass().newInstance();
    }

    public Menu createMenu() {
        return menuClass().newInstance();
    }

    public Button createButton() {
        return buttonClass().newInstance();
    }

    public Class windowClass();
    public Class menuClass();
    public Class buttonClass();
}

class MacWidgetFactory extends WidgetFactory {

    public Class windowClass() {
        return MacWindow.class;
    }

    public Class menuClass() {
        return MacMenu.class;
    }

    public Class buttonClass() {
        return MacButton.class;
    }
}

Smalltalk practice
Parent class normally does more complex stuff
How Do Factories create Widgets?  
Method 3) Prototype

class WidgetFactory
{
    private Window windowPrototype;
    private Menu menuPrototype;
    private Button buttonPrototype;

    public WidgetFactory( Window windowPrototype,
                          Menu menuPrototype,
                          Button buttonPrototype)
    {
        this.windowPrototype = windowPrototype;
        this.menuPrototype = menuPrototype;
        this.buttonPrototype = buttonPrototype;
    }

    public Window createWindow()
    { return windowFactory.createWindow() }

    public Window createWindow( Rectangle size)
    { return windowFactory.createWindow( size ) }

    public Window createWindow( Rectangle size, String title)
    { return windowFactory.createWindow( size, title) }

    public Window createFancyWindow()
    { return windowFactory.createFancyWindow() }

    etc.

    There is no need for subclasses of WidgetFactory.
Applicability

Use when

• A system should be independent of how its products are created, composed and represented

• A system should be configured with one of multiple families of products

• A family of related product objects is designed to be used together, and you need to enforce this constraint

• You want to provide a class library of products, and you want to reveal just their interfaces, not their implementation
Consequences

• It isolates concrete classes
• It makes exchanging product families easy
• It promotes consistency among products
• Supporting new kinds of products is difficult

Implementation

• Factories as singletons
• Defining extensible factories
Problem: Cheating Application Code

```java
public void installDisneyMenu(WidgetFactory myFactory)
{
    // We ship next week, I can't get the stupid generic Menu
    // to do the fancy Mac menu stuff
    // Windows version won't ship for 6 months
    // Will fix this later

    MacMenu disney = (MacMenu) myFactory.createMenu();
    disney.addItem( "Disney World" );
    disney.addItem( "Donald Duck" );
    disney.addItem( "Mickey Mouse" );
    disney.addMacGrayBar( );
    disney.addItem( "Minnie Mouse" );
    disney.addItem( "Pluto" );
    etc.
}
```

How to avoid this problem?