

Holub on Patterns: Learning Design Patterns by Looking at Code

ALLEN HOLUB

Apress®

Holub on Patterns: Learning Design Patterns by Looking at Code

Copyright © 2004 by Allen Holub

Lead Editor: Gary Cornell

Technical Reviewer: Ken Arnold

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis,
Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser

Project Manager: Tracy Brown Collins

Copy Edit Manager: Nicole LeClerc

Copy Editor: Kim Wimpsett

Production Manager: Kari Brooks

Production Editor: Janet Vail

Proofreader: Nancy Sixsmith

Compositor and Artist: Diana Van Winkle, Van Winkle Design Group

Indexer: Ann Rogers

Artist: Diana Van Winkle, Van Winkle Design Group

Interior Designer: Diana Van Winkle, Van Winkle Design Group

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Library of Congress Cataloging-in-Publication Data:

Holub, Allen I.

Holub on patterns : learning design patterns by looking at code /

Allen Holub.

p. cm.

Includes index.

ISBN 1-59059-388-X (alk. paper)

1. Software patterns. 2. Object-oriented programming (Computer science) I. Title.

QA76.76.P37H65 2004

005.1—dc22

2004019635

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

Printed and bound in the United States of America 9 8 7 6 5 4 3 2

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Distributed to the book trade in the United States by Springer-Verlag New York, Inc. 233 Spring Street, 6th Floor, New York, New York 10013 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER (1-800-777-4643), fax 201-348-4505, e-mail orders@springer-ny.com, or visit <http://www.springer-ny.com>. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit <http://www.springer.de>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.holub.com/goodies/patterns>.



A Design-Pattern Quick Reference

This appendix is a reference of the Gang-of-Four design patterns, intended to jog your memory about how the patterns work. Ironically, the original Gang-of-Four presentation was this brief, but they expanded things in the book to make it more accessible. Once you know the patterns, however, brevity is good. This catalog probably won't be of much use if you don't already have some familiarity with the patterns, however. A lot of the material you'd find in an introductory-level discussion is either missing or condensed in this appendix.

Though I've followed the Gang-of-Four organization (alphabetical by category), I have deliberately not followed the Gang-of-Four format for the pattern description itself. In particular, I've restated their "intent" section to make it more understandable. I've also used stripped-down examples, and my examples are not the same as the Gang-of-Four examples. In particular, since most of us aren't doing GUI work, I've tried to eliminate GUI-related example code.

I've tried to make up for some of this brevity by listing places where the design patterns are found in the Java packages so you can see how they're applied in practice. (Some patterns don't appear in Java, in which case the "Usage" example will say so). Also, you can find detailed code similar to my stripped-down examples in one of the volumes of Chan, Lee, and Kramer's *The Java Class Libraries* (Addison-Wesley, various editions) or in the Java documentation or tutorials available on the Sun web site.

I've played fast and loose with the code in the interest of saving space—I've omitted required import statements, access privileges, exceptions, and so on. The formatting isn't ideal in places. I'm assuming you know what you're doing in the Java-programming department and are more interested in the clarity of the example than in having cut-and-paste code. The code won't always compile cleanly as it stands.

Finally, I've said a few things in these notes that you may find shocking if you haven't read the rest of the book or some of my other work—things such as "objects must be responsible for building their own user interfaces." I simply have no room to explain this sort of thing in a quick reference; you have to read the rest of the book.

[This page intentionally left blank¹]

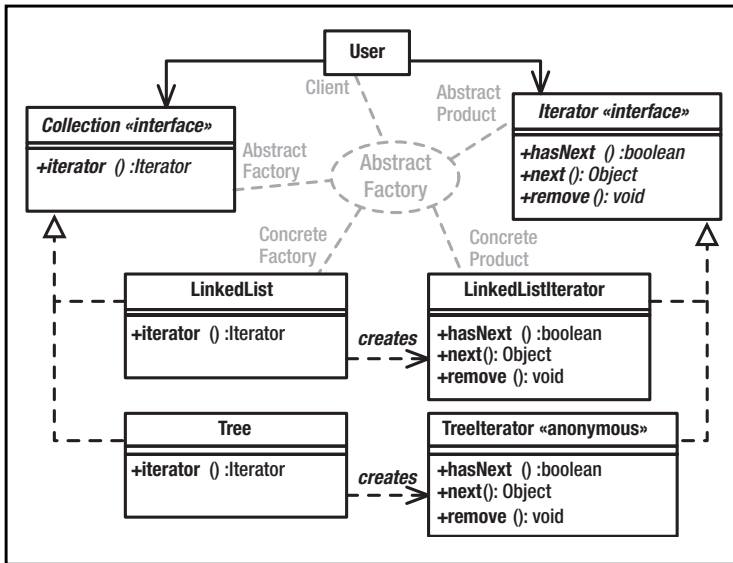
1. Rather an odd thing to say, since the page isn't blank at all—it contains the text “This page intentionally left blank” —but imagine that it's blank.

Creational Patterns

The creational patterns are all concerned with object creation (fancy that!). Most of them provide ways to create objects without knowing exactly what you're creating (beyond the interfaces supported by the created objects). Programming in terms of interfaces rather than concrete-classes is essential if you intend to write flexible, reusable code. My rule of thumb is that as much as 80 percent of my code should be written in terms of interfaces.

Abstract Factory

Create objects knowing only the interfaces they implement (without knowing the actual class). Typically, create one of a “family” of objects (one of several kinds of Iterators, one of several kinds of graphical widgets, and so on).



Abstract Factory: Interface to the actual factory.

Concrete Factory: Implements the Abstract Factory interface to create a specific class of object.

Abstract Product: The sort of product that the Abstract Factory creates.

Concrete Product: The actual object (whose class you don’t know) created by the factory.

Client: Uses the created objects only through their interfaces.

What Problem Does It Solve?

Abstract Factory makes it easy to create and manipulate objects without knowing exactly what they are. (This example uses an Iterator—it doesn’t care what kind.) This way, it’s easy to add new sorts of concrete products to the system without changing any of the code that uses those products.

Abstract Factory also makes it easy for your code to operate in diverse environments. The system creates a unique Concrete Factory (which creates unique Concrete Products) for each environment, but since you use the interface, you don’t actually know which environment (or which Concrete Product) you’re using.

Pros (✓) and Cons (✗)

- ✓ The anonymity of the Concrete Factory and Product promotes reuse—the code that uses these objects doesn’t need to be modified if the Factory produces instantiations of different classes than it used to do.

- ✗ If the product doesn’t do what you want, you may have to change the Abstract Product interface, which is difficult. (You have to change all the Concrete Product definitions.)

Often Confused With

Builder: Builder’s Director may use an Abstract Factory to create Builder objects, but the point of Builder is that the Director doesn’t know what it’s building.

Factory Method: A Factory Method—an unfortunate choice of pattern name on the part of the Gang of Four—is an abstract method that a subclass overrides. The Abstract-Factory method that creates objects is only rarely a Factory Method.

See Also

Singleton, Factory Method, Builder

Implementation Notes and Example

```
interface Collection
{ Iterator iterator();
  //...
}
interface Iterator
{ boolean hasNext();
  Object next();
  //...
}
class Tree implements Collection
{ public Iterator iterator()
  { return new Iterator()
    { // Implement Iterator interface
      // here (to traverse a Tree).
      // (See description of Iterator
      // pattern for implemenation.)
    }
  }
}
class User // uses only interfaces
{
  public void operation( Collection c )
  { Iterator i = c.iterator();
    while( i.hasNext() )
      doSomethingWith( i.next() );
  }
}
```

Collection is the Abstract Factory, Iterator is the Abstract Product, Tree is the Concrete Factory, and the anonymous-inner-class Iterator implementation is the Concrete Product.

Abstract Factory has many variants, probably the most common of which is a Concrete Factory that comprises its own interface—no “Abstract Factory” interface as such exists. This Concrete Factory is typically a Singleton. The methods of the class effectively comprise the Abstract Factory interface.

```
class SingletonFactory
{ private static instance=newSingletonFactory();
  public static SingletonFactory instance()
  { return instance;
  }

  void factoryOperation1(){/*...*/}
  void factoryOperation2(){/*...*/};
}
```

A similar, though more abstract, example is described in the entry for Factory Method.

No reason exists why, in the no-Abstract-Factory variant, the Concrete Factory cannot create a user interface that allows the physical user to select which of several possible concrete products to create. Consider a drawing program whose “shape” factory creates a user interface showing a palate of possible shapes. The user can then click a shape to determine which Concrete Product (shape derivative) to create in response to a newShape() request.

Usage

```
f(Collection c)
{ Iterator i = c.iterator();
  //...
}
```

Collection and Iterator are the Abstract Factory and Product. Concrete Factories and Products are anonymous.

```
ButtonPeer peer =
  Toolkit.getDefaultToolkit().
    createButton(b);
```

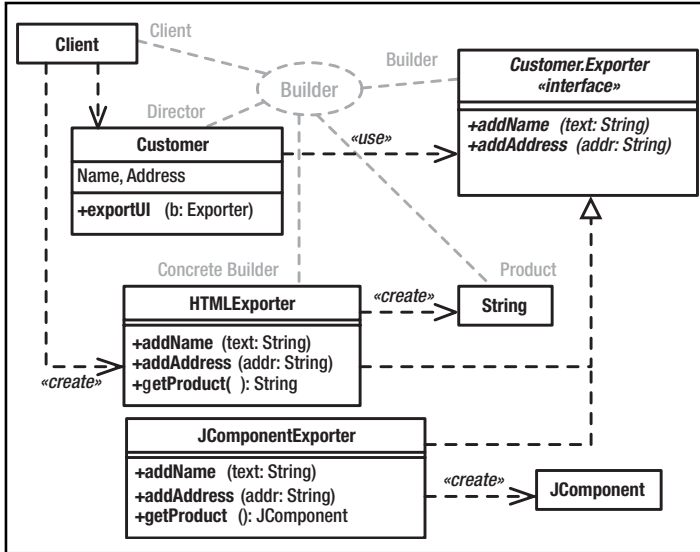
Toolkit is both a Singleton and an Abstract Factory. Most of the methods of Toolkit are abstract, and getDefaultToolkit() returns an unknown derivative of Toolkit. No need exists for an Abstract Factory interface per se.

```
URL home = new URL("http://www.holub.com");
URLConnection c = home.getConnection();
InputStream in = c.getInputStream();
```

URL is a concrete URLConnection factory, and URLConnection is an abstract InputStream factory, so URLConnection is both an Abstract Product and an Abstract Factory, depending on context. URL, URLConnection, and InputStream are interfaces by use, not by declaration.

Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations without having to modify the constructing object.



Director: Builds an object without knowing exactly what it’s building.

Builder: Interface used by the Director to do the construction.

Concrete Builder: Actually builds the product by following directions given by the Director. Typically created externally (by the Client) or by an Abstract Factory.

Product: The object built by the Builder under the direction of the Director.

What Problem Does It Solve?

It’s desirable to separate business logic from UI logic, but in an OO system you cannot expose implementation details. A well-done class definition will not have “get” methods that return state information, so an object must build its own UI. Nonetheless, it’s sometimes necessary for an object to build more than one representation of itself, and it’s undesirable to clutter up the business-logic code with the details needed to build multiple representations.

Builder solves this problem by putting the representation-specific code into a Builder object that’s distinct from a Director (“business”) object. Builder also easily lets you add representations later without impacting existing code at all.

Non-UI applications: credit-card processing; for example, every credit-card-payment processor requires a different protocol, with identical information presented in different ways. Builder separates you from the needs of the credit-card processor, letting you build a packet of information without needing to know which processor will receive that information. The organization of the data is hidden from you in the “concrete builder” that you talk to via a public, processor-independent, interface.

Pros (✓) and Cons (✗)

- ✓ Builder nicely isolates the representation of an object from the associated “business” logic, making it easy to add new (or change) representations of an object without modifying business logic.
- ✗ A change in the Builder interface mandates changes in all implementing classes.
- ✗ It’s awkward to represent some UI elements cleanly in all representations (for example, HTML vs. Swing).

Often Confused With

Bridge: An application building a UI using AWT is a Director—the actual representation is unknown to the application. In this way, AWT reifies both Builder and Bridge.

Visitor: A visitor could build a UI by visiting every element of a data structure. It is “pulling” information for UI construction from the model rather than having that information “pushed” onto it.

See Also

Bridge, Visitor

Implementation Notes and Example

```

class Customer
{
    private Address address;
    private String name;
    public void exportUI( Exporter b )
    {
        b.addName(name);
        b.addAddress(address.toString());
        //...
    }
    public interface Exporter
    {
        void addName(String text);
        void addAddress(String addr);
        //...
    }
}
class HTMLExporter implements Customer.Exporter
{
    // Implement Builder methods here. This
    // Implementation creates an HTML
    // representation of the object.
    //...
    public String getProduct()
    {
        // Return HTML String.
    }
}
class JComponentExporter implements Customer.Exporter
{
    JComponent product;
    // Implement Builder methods here. This
    // Implementation creates a JComponent
    // that represents the object.
    //...
    public JComponent getProduct()
    {
        return product;
    }
}
class Client
{
    Employee director;
    //...
    public void addYourUITo(JPanel someWindow)
    {
        Customer.Exporter b =
            new JComponentExporter();
        director.exportUI( b );
        someWindow.add( b.getProduct() );
    }
}

```

Usage

```

URL url = new URL("http://www.holub.com");
URLConnection connection = url.openConnection();
connection.setDoOutput( true );
connection.connect();
OutputStream out = connection.getOutputStream();
while( c = getCharacterFromSomewhere() )
    out.write( c );
out.close();

```

The `createUI()` method is passed a `Builder` that could be an `HTMLExporter` (that creates an HTML representation) or a `JComponentExporter` (that produces a `JComponent`). The `Director` object doesn't know which of these products it is building—it just calls interface methods.

The `Client` object that's driving this process *does* know what it's building since it created the `Builder`. Consequently, it's reasonable for it to extract the correct product.

You could get better abstraction by using an `Abstract Factory` to create the `Builder` objects rather than `new`. By the same token, if all output was going to a file, you could add a `print-YoursefToFile(String name)` method to the `Builder` interface; the `Director` could call that method at an appropriate time, and the `Client` wouldn't have to extract anything; it would just supply a filename.

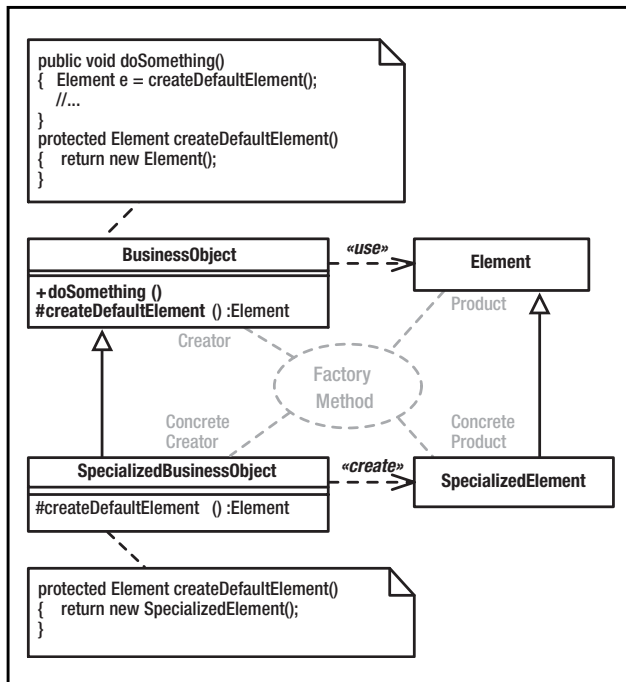
`Builder` implementations could be public inner classes of the `Director`. I'd probably do it that way unless I expected that `Builders` would be defined elsewhere in the code.

The `Director` is “pushing” information into the `Builder`. Consequently, you have no need for accessors (get methods) in the `Director`, and the coupling between the `Builder` and `Director` is very light. In general, accessors violate the integrity of the object by exposing implementation detail. Avoid them.

This code comprises a `Director`. It uses `Abstract Factory` (`URLConnection`) to get a `Builder` (the `OutputStream`), which builds an HTTP packet. The `Director` doesn't know that it's building an HTTP packet, however. (If an `ftp://` URL had been specified, it would be building an FTP packet.) The `close()` call, instead of getting the product, just sends it off.

Factory Method

Let subclasses decide which objects to instantiate.



Creator: Defines a method that needs to create an object whose actual type is unknown. Does so using abstract-method call.

Concrete Creator: Subclass that overrides the abstract object-instantiation method to create the Concrete Product.

Product: Interface implemented by the created product. Creator accesses the Concrete Product object through this interface.

Concrete Product: Object used by the Creator (superclass) methods. Implements the Product interface.

What Problem Does It Solve?

This pattern is useful when you can do all (or most) of the work at the superclass level but want to put off deciding exactly which sort of object you'll be working on until runtime. (You'll manipulate objects that a derived-class creates through an interface that you define.)

It is often the case that a superclass object needs to create worker objects of some default type, but the superclass can work equally well using worker objects that are extensions of the original type. This pattern lets you create specialized worker objects in a specialized subclass.

This way of doing things is often useful when you create an implementation-inheritance-based "framework" that you expect users to customize using derivation.

Pros (✓) and Cons (✗)

- ✓ Easy to implement when a full-blown Abstract Factory is overkill.
- ✗ This pattern forces you to use implementation inheritance, with all its associated maintenance problems.

- ✗ Inheritance-based framework architectures, in which Factory Methods are usually found, are not the best way to achieve reuse. Generally, it's best if a framework class can simply be instantiated and used directly, without forcing a programmer to create a subclass to make the superclass useful. Implementation inheritance should be reserved for situations where you need to modify superclass behavior to perform in an unusual way.

Often Confused With

Abstract Factory: The Concrete Factory can use Factory Method to create Concrete Products. The creational method does not have to use this design pattern, though. A method is not a Factory Method simply because it manufactures objects. (I've seen the term misused in the Java documentation, among other places.) In Factory Method, a derived-class override makes the object.

See Also

Abstract Factory, Template Method

Implementation Notes and Example

```
public class BusinessObject
{   public void doSomething()
    {   Element e = createDefaultElement();
        //...
    }
    protected Element createDefaultElement()
    {   return new Element();
    }
}
public class Element
{   public void f(){/*...*/}
}

public class SpecializedBusinessObject
{   protected Element createDefaultElement()
    {   return new SpecializedElement();
    }
    private class SpecializedElement extends Element
    {   public void f(){/*...*/ }
    }
}
```

You can sometimes customize superclass behavior by providing nonstandard objects for it to work with. In this example, a specialized form of a business object is created by extending the generalized version in such a way that it provides a specialized element rather than the default element.

The negative side to this architecture is that you often must modify the superclass if you add a subclass. The `java.awt.Toolkit` `AbstractFactory` overcomes this problem while still using an abstract-superclass architecture by instantiating objects with `Class.forName()` rather than an abstract-method call. This structure is still `Factory Method`, since the decision about which class to instantiate is deferred to runtime—it's just not a subclass that's making the decision.

It is often inappropriate to use `Factory Method` if the only method provided by the subclass is the `Factory Method` itself. You're adding complexity with little commensurate benefit.

Never leverage the fact that protected grants package access in Java. The `createDefaultElement()` method is protected only because I expect it to be overridden by a subclass (otherwise it would be private). This method should not be called from anywhere other than the `BusinessObject` superclass. The language, unfortunately, grants package access to protected members, but it's best to pretend that package access is not possible.

This pattern is so trivial as to almost not be worth calling it a pattern. It's more interesting in C++, where it's called a *virtual constructor* and is implemented by overriding operator `new()`.

Usage

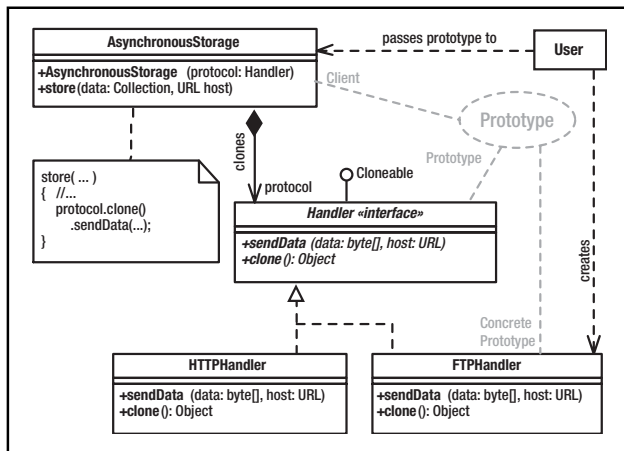
```
public class MarkupPanel extends JEditorPane
{   public MarkupPanel()
    {   setEditorKit(
        new HTMLEditorKit()
        {   public ViewFactory getViewFactory()
            {   return new CustomViewFactory();
            }
        }
    );
}
private class CustomViewFactory
    extends HTMLEditorKit.HTMLFactory
{   public View create(Element e)
    {   return new View()
        {   protected Component createComponent()
            {   return new Component(){/*...*/};
            }
        }
    }
}
```

In Swing's `JEditorPane`, various HTML elements are displayed as “views.” When a parser recognizes an HTML element, it requests a “view” that renders the component. You specify a custom representation of an HTML element by providing a derived-class override of a `create()` method that returns a component of your choice.

`Component` is the Product. The (anonymous) implementation of `Component` is the Concrete Product. The `MarkupPanel` is the Creator, and the `CustomViewFactory` is the Concrete Creator. `createComponent()` is the `Factory Method`. Similarly, `getViewFactory()` is a `Factory Method` that produces custom view factories. A subclass specifies alternative view factories by overriding `getViewFactory()`.

Prototype

Create objects by making copies of (*cloning*) a prototypical object. The prototype is usually provided by an external entity or a Factory, and the exact type of the prototype (as compared to the interfaces it implements) may not be known.



Prototype: Interface of object to copy; must define a mechanism for cloning itself.

ConcretePrototype: Object that's copied; implements cloning mechanism.

Client: Creates a new object by asking the Prototype for a clone.

What Problem Does It Solve?

1. Allows you to create several objects of the same type, even if you don't know exactly what that type is.
2. In Abstract Factory, information needed to initialize the Concrete Product (constructor arguments, for example) must be known at compile time. Most Abstract Factory reifications use the default, no-arg constructor. When you use Abstract Factory to make objects that must be in a nondefault state, you must first create the object and then modify it externally, and this external modification may happen in many places in the code. It would be better to create objects with the desired initial (nondefault) state and simply copy those objects to make additional ones. You may use Abstract Factory to make the prototype object.
3. Sometimes objects will be in only a few possible states, but you have many objects in each state. (The Gang of Four describe a Note class in a music-composition system; many instances of whole-note, half-note, and quarter-note objects exist—all whole notes are in an identical state.)
4. Sometimes classes are specified at runtime and are created with *dynamic loading* (for example, `Class.forName("class.name")`) or a similarly expensive process (when initial state is specified in an XML file, for

example). Rather than repeatedly going through the expense of creating an object, create a single prototype and copy it multiple times.

Pros (✓) and Cons (✗)

- ✓ You can install a new concrete product into a Factory simply by giving the Factory a prototype at runtime. Removal is also easy.
- ✓ Prototype can reduce object-creation time.
- ✓ Abstract Factory forces you to define classes with marginally different behavior using subclassing. Prototype avoids this problem by using state. When an object's behavior changes radically with state, you can look at the object as a dynamically specifiable class, and Prototype is your instantiation mechanism.
- ✗ You must explicitly implement `clone()`, which can be quite difficult. Also, think about deep-vs.-shallow copy issues. (Should you copy a reference, or should you clone the referenced object?) Finally, sometimes the clone method should act like a constructor and initialize some fields to default values. A clone of a list member cannot typically be in the list, for example.

See Also

Abstract Factory, State

Implementation Notes and Example

```

class AsynchronousStorage
{ private Handler protocol;
  public AsynchronousStorage( Handler protocol )
  { this.protocol = protocol;
  }

  void store( Collection data, URL host )
  { byte[] bytes = new byte[128];
    //...
    Handler handler=(Handler)protocol.clone();
    handler.sendData(bytes, host);
  }
}

interface Handler extends Cloneable
{ void sendData(byte[] data, URL host);
  Object clone();
}

class HTTPHandler implements Handler
{ public void sendData(byte[] data, URL host)
  { // Send data asynchronously to
    // host using HTTP protocol, creating
    // background thread if necessary
  }
  public Object clone(){ /*...*/ }
}

class FTPHandler implements Handler
{ public void sendData(byte[] data, URL host)
  { // same as above, but use FTP protocol
  }
  public Object clone(){ /*...*/ }
}

class User
{ private Collection theData;
  private AsynchronousStorage dataStore =
    new AsynchronousStorage(
      new HTTPHandler() );
  public void flush()
  { dataStore.store(theData, new URL(/*...*/));
  }
}

```

In this example, HTTPHandler talks asynchronously to a remote host using HTTP, and FTPHandler talks using FTP. One handler exists for each communication; several handlers can be active simultaneously, each talking to their respective hosts. Prototype is used to decouple the protocol from the AsynchronousStorage class. The User class decides which protocol to use and then passes an appropriate handler to the AsynchronousStorage object, which uses clones of the prototype to do the actual work.

You cannot use new to implement a “clone” method. The following code won’t work:

```

Class Grandparent
{ public Grandparent(Object args){/*...*/}
  Base myClone(){ return new Base(args);}
}
Class Parent
{ public Parent(){ super("arg");}
  Derived myClone(){return new Parent(args)}
}
Class Child
{ public Child(){ super(); }
  /* inherit the superclass myClone */
}
//...
Grandparent g = new Child();
//...
g.myClone(); // Returns a Parent, not Child!

```

Using Java’s clone() solves this problem by getting memory from super.clone().

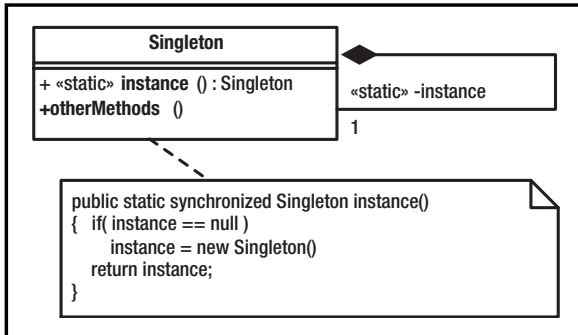
Usage

(Not used)

Prototype is used in the implementations of several classes but not in the external interfaces to any of the Java classes. You do see it in the Bean Box application that demonstrates GUI-style JavaBeans. When you customize an object and put it on the palate, you’re creating a prototype. When you drag the customized object from the palate to the dialog box that you’re constructing, you’re making a copy of the prototype.

Singleton

A class with a constrained number of instances (typically one). The instance is globally accessible.



Singleton: The object being created; defines a class-level (static) get-instance method that returns the instance. The class-level get-instance method may create the object if necessary.

What Problem Does It Solve?

Programs often have a need for single-instance objects. Objects, for example, may represent a single database, a single company, and so forth.

Pros (✓) and Cons (✖)

- ✓ Better than a global object in that access is controlled, and the global namespace isn't cluttered with hard-to-find objects.
- ✓ Singleton guarantees that the object is created (and destroyed) only once—essential when the Singleton manages a global resource such as a database connection.
- ✖ Easy to abuse. A Singleton called `Globals` that contains nothing but public *variables* is an abomination. (A Singleton containing global *constants* is reasonable if the values of the constants need to be initialized at runtime. If the values are known at compile time, use an interface made up solely of static final fields.)
- ✖ Another common abuse of Singleton defines a `User` object that contains all the user-interface code. In a properly done OO system, objects must be responsible for building their own user interfaces. Similarly, you should not have a “system” or “main” singleton. The system is the entire program, not a single object. System objects are what Arthur Riel calls *god classes* (in his book *Object-Oriented Design Heuristics*, (Addison-Wesley, 1996). Avoid them.

Often Confused With

Utility: A Utility is a class comprised solely of static methods, the purpose of which is to provide a grab bag of global methods that often compensate for some deficiency in the language or libraries. Examples include Java's `Math` and `Arrays` utilities.

Singleton can be implemented the same way as Utility—as a class made up solely of static methods. That is, when all fields of a class are static, the class is effectively an object: It has state and methods. The main disadvantage to this everything-is-static approach is that you can't change the behavior of a Singleton using derivation.

See Also

Abstract Factory

Implementation Notes and Examples

```

class Singleton1
{
    private static Singleton instance;
    private Singleton1()
    {
        Runtime.getRuntime().addShutdownHook
            (
                new Thread()
                {
                    public void run()
                    {
                        /* clean-up code here */
                    }
                }
            );
    }
    public static synchronized Singleton instance()
    {
        if( instance == null )
            instance = new Singleton();
        return instance;
    }
}

class Singleton2
{
    private static final Singleton instance =
        new Singleton2();
    public static Singleton instance()
    {
        return instance;
    }
    //...
    //Other than creating object in static
    //initializer, is identical to Singleton1
}

class Singleton3
{
    static Type allFields;
    static Type allOperations();
    // No instance() method, just use the
    // class name to the left of the dot.
}

```

Use the `Singleton1` form when you can't create the object at class-load time (for example, you didn't have information that's determined by program state or is passed to the creation method).

You *must* synchronize the `instance()` method of `Singleton1` as shown. "Clever" ways to eliminate synchronization such as "double-checked locking" don't work. (Period. Don't do it!)

Use the `Singleton2` or `Singleton3` form when possible; synchronization is not required during access. (The JVM may load the class at any time, but it shouldn't initialize the Class object until first use (Java Language Specification, 12.4.1); static initializers shouldn't execute until first use.

Call `addShutdownHook()` in the constructor when program-shut-down cleanup activities (such as shutting down database connections in an orderly way) are required. Do not use a finalizer, which may never be called.

A private constructor prevents someone from saying `new Singleton()`, thereby forcing access through `instance()`.

You have no requirement that only one instance of the Singleton exists, only that the number of instances are constrained and that access to the instances are global. For example, a `DatabaseConnection.getInstance()` method may return one of a pool of database connections that the Singleton manages.

In UML, the role associated with the Singleton is usually also the class name.

Usage

```

Image picture =
    Toolbox.getDefaultToolbox().getImage(url);

```

The `Toolbox` is a classic form of `Singleton1` in the "Examples" section. `getDefaultToolbox()` returns a `Toolbox` instance appropriate for the operating system detected at runtime.

```

Border instance =
    BorderFactory.createBevelBorder(3);

```

Manages several `Border` instances, but only one instance of a `Border` object with particular characteristics (in this case, a three-pixel beveled border) will exist, so it's a `Singleton`. All subsequent requests for a three-pixel beveled border return the same object.

```

Class classObject =
    class.forName("com.holub.tools.MyClass");

```

There's only one `Class` object for a given class, which effectively contains all static members.

[This page intentionally left blank.²]

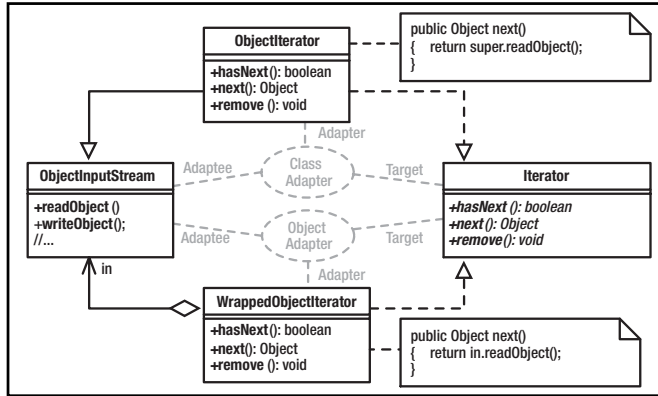
2. Here, we have the second reification of the this-page-intentionally-left-blank pattern.

Structural Patterns

The structural patterns concern themselves with the organization of the program. I think of them as static-model patterns. Their intent is always to organize classes so that certain structural ends can be achieved. For example, the purpose of Bridge is to organize two subsystems in such a way that one subsystem can change radically (even be replaced entirely) without affecting the code in the other subsystem. The whole point of this organization is that you can make changes to the program without having to change the dynamic model at all.

Adapter

Make a class appear to support a familiar interface that it doesn't actually support. This way, existing code can leverage new, unfamiliar classes as if they are existing, familiar classes, eliminating the need to refactor the existing code to accommodate the new classes.



Adaptee: An object that doesn't support the desired interface

Target: The interface you want the Adaptee to support.

Adapter: The class that makes the Adaptee appear to support the Target interface. Class Adapters use derivation. Object Adapters use containment.

What Problem Does It Solve?

1. A library that you're using just isn't working out, and you need either to rewrite it or to buy a replacement from a third party and slot this replacement into your existing code, making as few changes as possible.
2. You may need to refactor a class to have a different interface than the original version (you need to add arguments to a method or change an argument or return-value type). You could have both old-style and new-style versions of the methods in one giant class, but it's better to have a single, simpler class (the new one) and use Adapter to make the new object appear to be one of the old ones to existing code.
3. Use an Adapter to make an old-style object serialized to disk appear to be a new-style object when loaded.

Pros (✓) and Cons (✗)

- ✓ Makes it easy to add classes without changing code.
- ✗ Identical looking Object and Class Adapters behave in different ways. For example, new ObjectAdapter(obj) and new ClassAdapter(obj) are both supported; the Object Adapter simply wraps obj, but the Class Adapter copies the fields of obj into its superclass component. Copying is expensive. On the plus side, a Class Adapter is an Adaptee, so it can be passed to methods expecting an object of the Adaptee class and also to methods that expect the Target interface. It's difficult to decide whether

an Object or Class Adapter is best. It's a maintenance problem to have both.

- ✗ Difficult to implement when the library is designed poorly. For example, java.io.InputStream is an abstract class, not an interface, so you can't use the Class-Adapter pattern to create a RandomAccessFile that also supports the InputStream interface (you can't extend both RandomAccessFile and InputStream). You can use Object Adapter, or you can refactor the code to make InputStream an interface (as it should have been) and then implement that interface in an AbstractInputStream that has all the functionality now in InputStream. Collections do it correctly.

Often Confused With

Mediator: Mediator is the dynamic-model equivalent of Adapter. Adapters are passive, passing messages to single Adaptees. Mediators interact with many colleagues in complex ways.

Bridge: Adapters change interfaces. Bridges isolate subsystems. Adapters are little things; Bridges are big.

Decorator: The encapsulated object in Decorator has the same interface as the container. Decorator modifies the behavior of some method or adds methods, but otherwise looks exactly like the wrapped object. Object Adapters have different interfaces than the wrapped object and don't change its behavior.

See Also

Mediator, Bridge, Decorator

Implementation Notes and Example

```

class ObjectIterator extends ObjectInputStream
    implements Iterator
{
    private boolean atEndOfFile = false;
    public ObjectIterator(InputStream src)
        throws IOException
    {
        super(src);
    }
    public boolean hasNext()
    {
        return atEndOfFile == false;
    }
    public Object next()
    {
        try
        {
            return readObject();
        }
        catch( Exception e )
        {
            atEndOfFile = true;
            return null;
        }
    }
    public void remove()
    {
        throw new UnsupportedOperationException();
    }
}

class WrappedObjectIterator implements Iterator
{
    private boolean atEndOfFile = false;
    private final ObjectInputStream in;
    public
    WrappedObjectIterator(ObjectInputStream in)
    {
        this.in = in;
    }
    public boolean hasNext()
    {
        return atEndOfFile == false;
    }
    public Object next()
    {
        try
        {
            return in.readObject();
        }
        catch(Exception e){/* as above */}
    }
    public void remove()
    {
        throw new UnsupportedOperationException();
    }
}

```

Usage

```
InputStream in = new StringInputStream("hello");
```

ObjectIterator is a Class Adapter that adapts an ObjectInputStream to implement the Iterator interface. This way, you can use existing methods that examine a set of objects by using an Iterator to examine objects directly from a file. The client doesn't know or care whether it's reading from a file or traversing a Collection of some sort. This flexibility can be useful when you're implementing an Object cache that can overflow to disk, for example. More to the point, you don't need to write two versions of the object-reader method, one for files and one for collections.

WrappedObjectIterator is an Object Adapter version of ObjectIterator that uses containment rather than inheritance.

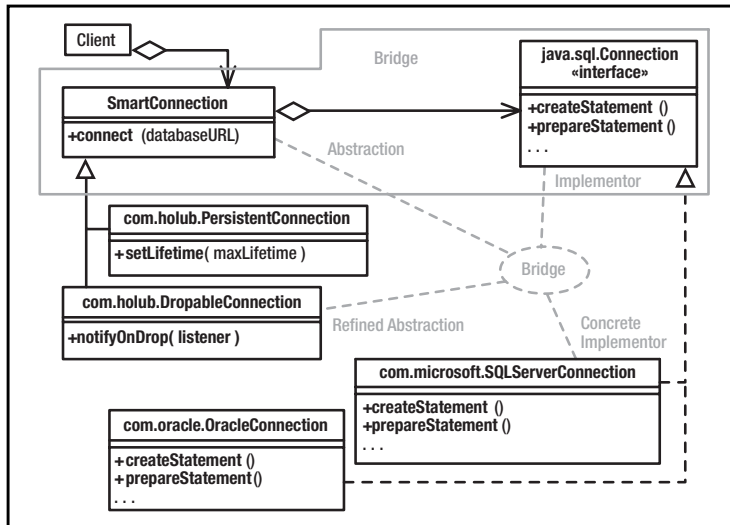
The Class Adapter, since it *is* an ObjectInputStream that implements Iterator, can be used by any method that knows how to use either ObjectInputStream or Iterator. The Object Adapter, since it encapsulates the input stream, cannot be used as an ObjectInputStream, but you can use the input stream for a while, temporarily wrap it in a WrappedObjectIterator to extract a few objects, and then pull the input stream out again.

The two implementations require about the same amount of work so it's a judgment call which one is best. It all depends on what you're using it to do.

Adapter lets you access a String as if it were a file (InputStream). Similar adapters include ByteArrayInputStream, CharArrayReader, PipedInputStream, PipedReader, and StringReader. Don't confuse these adapters with the Decorators in *java.io* (BufferedInputStream, PushbackInputStream, and so on).

Bridge

To decouple subsystems so that either subsystem can change radically without impacting any code in the other one, put a set of interfaces between two subsystems and code to these interfaces.



Abstraction: A subsystem-independent portal into subsystem-specific code.

Implementor: An interface used by the Abstraction to talk to a subsystem-specific implementation. Typically is also the Abstract Product of an Abstract Factory.

Refined Abstraction: Often omitted, a version of the Abstraction, customized for a particular application.

Concrete Implementor: A subsystem-specific implementation of the Implementor.

What Problem Does It Solve?

Often used to achieve platform independence. Application-specific code on one side of the bridge uses platform-dependant code on the other side. Reimplement that interface, and the “business” logic doesn’t know or care. Change the business logic, and the platform-specific interface implementations don’t care. Often, you’ll combine Bridge and Abstract Factory so that the Factory can supply the correct set of implementers at runtime, further isolating the two sides of the bridge. Examples of Bridge in Java are AWT and JDBC.

Pros (✓) and Cons (✗)

- ✓ In a pure inheritance model, you’d have a superclass that implemented some behavior and subclasses that customized this behavior for a specific platform. In Bridge, the superclass is effectively replaced by an interface, so the problems associated with implementation inheritance are minimized, and the total number of classes are reduced.
- ✗ It’s difficult to implement interfaces so that each implementation behaves identically. Java’s AWT Bridge implements windowing components for different operating environments, but the Motif implementation behaved differently on the screen than the Windows implementation.

Often Confused With

Bridge is more of an architecture than a design pattern. A Bridge is often a *set* of interfaces and classes (called *abstractions*, unfortunately—they’re typically not abstract) that contain references to objects that implement a platform-independent interface in a platform-dependant way (Adapters). The Adapters are typically created by the Abstraction object using a Singleton-based Abstract Factory.

Adapter: Bridges separate subsystems, and Adapters make objects implement foreign interfaces. A one-interface bridge looks like a Class Adapter, however.

Facade: Facade simplifies the interface to a subsystem but may not isolate you from the details of how that subsystem works. Changes made on one side of the facade might mandate changes both to the other side of the facade and to the facade itself.

See Also

Abstract Factory, Singleton, Adapter, Facade, Mediator

Implementation Notes and Example

```

class SmartConnection
{   String username, password;
    java.sql.Connection connection;
    //...
    public void connect(String databaseURL)
                               throws Exception
    {   Class.forName( databaseURL ).newInstance();

        Connection connection = null;
        Statement statement = null;
        //...
        connection =
            DriverManager.getConnection(
                databaseURL, username, password );
    }
}

class PersistentConnection extends SmartConnection
{   long maxLifetime;
    public void setLifetime(long maxLifetime)
    {   // Arrange for connection to time
        // out after lifetime expires.
    }
}

class PooledConnection extends SmartConnection
{   public void notifyOnDrop(Runnable dropped)
    {   // Arrange to call dropped.run()
        // when connection is dropped.
    }
}

//-----

class SQLServerConnection
    implements java.sql.Connection
{   // Implementation that support SQL Server
    // interface.
}

class OracleConnection implements
java.sql.Connection
{   // Implemenation that supports Oracle's
    // interface.
}

```

The abstraction classes (*SmartConnection*, *PersistentConnection*, and *DropableConnection*) use the Bridge interface (*java.sql.Connection*) to talk to the implementation classes (*OracleConnection*, *SQLServerConnection*).

The two sides of the Bridge can change independently. For example, I can change *OracleConnection* radically, and the classes on the other side of the Bridge (*SmartConnection*, for example) are completely unaware of that change. This isolation is possible because Factory is used to create the Concrete Implementers.

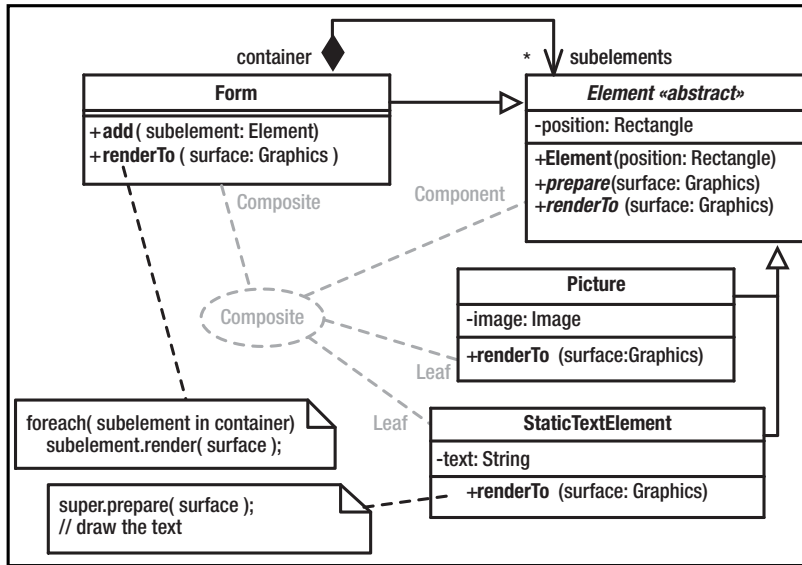
I can even support additional databases (by extending *java.sql.Connection*) without affecting the other side of the Bridge. By the same token, I can modify the *SmartConnection* class (and its subclasses) and even add additional subclasses, without impacting the other side of the bridge (the *java.sql.Connection* implementers).

Note that the Bridge completely isolates the subsystems from each other. The Client class knows only about the abstraction classes.

A Bridge is often very large. The JDBC Bridge consists of many Implementor interfaces and associated Concrete Implementations, and some of these interfaces are very large.

Composite

Organize a runtime hierarchy of objects that represent container/content (or whole/part) relationships as a collection of objects that implement a common interface. Some of the implementers of this interface define stand-alone objects, and others define containers that can hold additional objects, including other containers.



Component: An interface or abstract class that represents all objects in the hierarchy.

Composite: A Component that can hold other Components. It doesn't know whether these subcomponents are other Composites or are Leaves.

Leaf: A Component that stands alone; it cannot contain anything.

What Problem Does It Solve?

Often data structures can be organized into hierarchies in which everything in the hierarchy has a common subset of similar properties. For example, directories are files that can contain other files; a file can be atomic (a simple file not containing anything) or a subdirectory (a file that holds references to other files, including subdirectories). Composite lets you create these sort of containment hierarchies in such a way that a given container doesn't need to know whether its contents are atomic or composite objects. They both implement the same interface, so can be treated identically.

Pros (✓) and Cons (✗)

- ✓ The container is simple to implement because it treats all contents uniformly.
- ✓ It's easy to add new Component classes, just derive another class from the Component class (or interface).
- ✗ The Component tends to specify an unsatisfactory least-common-denominator interface.

- ✗ It's not always meaningful or appropriate for every Composite or Leaf to implement every method of the Component. It's an awkward runtime error if an unimplementable method throws an exception.

Often Confused With

Chain of Responsibility: Chain of Responsibility is also implemented using a runtime hierarchy of objects, but the point of Chain of Responsibility is to catch messages in appropriate places.

Decorator: Decorator also uses a containment strategy, but Decorators add or modify functionality of a single containee. The point of Composite is to make it easier to manipulate a *set* of contained objects.

See Also

Chain of Responsibility, Decorator, Flyweight

Implementation Notes and Example

```

abstract class Element
{
    private Rectangle position;
    public Element(Rectangle position)
    {
        this.position = position;
    }
    protected void prepare(Graphics surface)
    {
        // modify the surface's coordinate
        // system so that (0,0) is at the
        // current Element's position.
    }
    public abstract void renderTo(Graphics surface);
}

class Form extends Element
{
    private Collection subelements
        = new ArrayList();
    public Form( Rectangle position )
    {
        super(position);
    }
    public void add(Element subelement)
    {
        subelements.add( subelement );
    }
    public void renderTo(Graphics surface)
    {
        prepare(surface);
        Iterator i = subelements.iterator();
        while( i.hasNext() )
            ((Element)i.next()).render(surface);
    }
}

class StaticText extends Element
{
    private String text;
    public StaticText(Rectangle position,
        String text)
    {
        super(position);
        this.text = text;
    }
    public void renderTo(Graphics surface)
    {
        prepare(surface);
        surface.drawText(text);
    }
}

```

Usage

```

Dialog box = new Dialog();
box.add( new Label("Lots of information") );

Panel subpanel = new Panel();
subpanel.add( new Label("Description") );
subpanel.add( new TextField() );
box.add( subpanel );

```

Element, is an abstract class that defines operations common to all Element objects (for example, the Element's relative position on the form). I've avoided making this information public (thereby damaging the integrity of the object) by providing a prepare() method that modifies the coordinate system of the Graphics object so that the current object can render itself in the upper-left corner of the surface. This way a getPosition() method is unnecessary, and the resulting class system is more robust.

The Form class has the role of Composite in the pattern. It's an Element that holds other Elements, some of which may be Forms and some of may might be StaticText. The point is that the Form class's render() method doesn't know or care about the actual type of the subelement. The subelements may be Elements, or they may be sub-Forms. All subelements are rendered identically (by passing them render() messages).

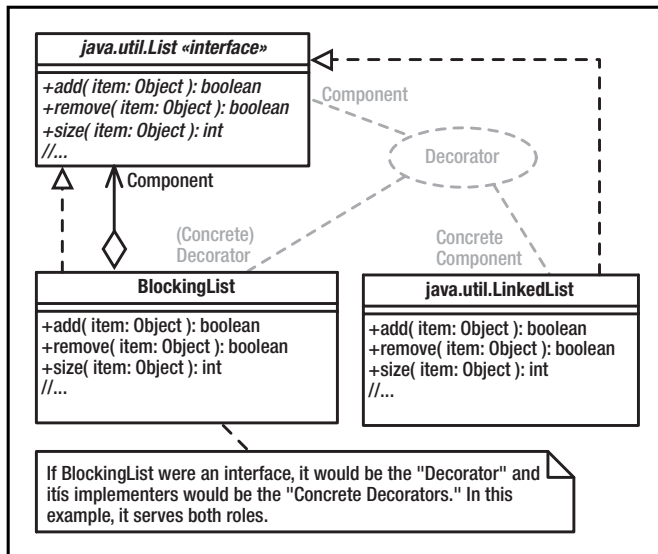
The StaticText class is a Leaf. It is an Element that doesn't contain other Elements and thus forms a leaf on the runtime-hierarchy tree. It has to know how to render itself, of course. Here, it just delegates to the Surface object.

A Dialog is a Composite that can hold Leaves (such as Label) and other Composites (such as Panel). This example also nicely demonstrates the affinity between Composite and Bridge, since AWT is also a bridge. (A DialogFrame, for example, is simultaneously a Composite in Composite and an Abstraction in Bridge.

Another good example of Composite is the new JDOM classes (<http://www.jdom.org>). An XML document is a list of Elements.

Decorator

Attach new responsibilities to (or modify the behavior of) an *object* at runtime. Decorators can simplify class hierarchies by replacing subclassing with containment.



Component: An interface for objects that can have responsibilities added to them (or have behavior modified) at runtime.

Concrete Component: An object to which additional responsibilities or new behavior is attached.

Decorator: Wraps a Component and defines an interface that conforms to the Component's interface but behaves differently.

Concrete Decorator: Extends the Decorator to define the additional behavior.

What Problem Does It Solve?

Using derivation hierarchies to add features is not a great idea. Consider an input stream. To add buffering, you'd derive a class that overrode the `input()` method to do buffering (doubling the number of classes). To add push-back, you'd have to derive from both classes, providing buffered and nonbuffered versions of `input()` that pushed characters back. In fact, every feature that you add through subclassing will require you to double the size of the class hierarchy. Decorator, on the other hand, is linear. To add a feature, you add exactly one Decorator class, no matter what the size of the original hierarchy.

Decorator also nicely solves the problem of runtime configuration. Sometimes, you don't know exactly how an object should behave until runtime. Behavior may be specified in a configuration file, for example. Decorator allows you to assemble (at runtime) a composite object that contains exactly the mix of capabilities you need without having to know which of these capabilities will be needed when you write the code.

Decorator helps you break up large complex operations into small simple operations.

Pros (✓) and Cons (✘)

- ✓ The size and complexity of the class hierarchy is considerably reduced.
- ✘ A feature introduced in a Decorator (such as pushback) is at best hard (or even dangerous) to access if the decorator is itself decorated. The system is sensitive to the order in which Decorators are applied. Java's `PushbackInputStream` works well at the outermost layer, but a `PushbackInputStream` wrapped with a `BufferedInputStream` doesn't work. (It doesn't push back into the buffer.)

Often Confused With

Adapter: Changes an interface; Decorator changes behavior.

Chain of Responsibility: Passes messages to the most appropriate handler. In Decorator, messages are handled by the outermost Concrete Decorator.

Composite: Decorators add responsibilities. Composites never do.

See Also

Strategy

Implementation Notes and Example

```
import java.util.*;

/* I would prefer for this class to be a LinkedList,
 * but LinkedList is not an interface, and
 * useful methods like addFirst() and removeFirst()
 * are not defined in an interface.
 */

public class BlockingList implements List
{   private final List component;

    public BlockingList( List component )
    {   this.component = component;
    }

    private boolean noLongerEmpty()
    {   try
        {   while( component.size() == 0 )
            wait();
            return true;
        }
        catch( InterruptedException e )
        {   return false;
        }
    }

    synchronized public boolean add(Object o)
    {   boolean toReturn = component.add(o);
        notifyAll();
        return toReturn;
    }

    synchronized public boolean remove(Object o)
    {   if( noLongerEmpty() )
        return component.remove(o);
        return false;
    }

    synchronized public int size()
    {   return component.size();
    }

    /* Synchronized versions of all other methods of
     * the List interface are implemented here ...
     */
}
```

Usage

```
JComponent widget = new JTextArea(80,200);
widget = new JScrollPane(widget);
JFrame frame = new JFrame();
Frame.getContentPane().add( widget );
```

```
InputStream in = new FileInputStream("x.txt");
in = new BufferedInputStream( in );
in = new PushBackInputStream( in );
```

Think fish. Bigger fish are Decorators that implement the same interfaces as the smallest fish (the Component). If a smaller fish has swallowed a hook and line, talk to it by yanking the string.

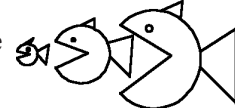
BlockingList is a Decorator that modifies the behavior of any List implementation so that a thread that tries to remove something from an empty list will block (be suspended) until some other thread adds something to the list—a common interthread communication architecture. Use a BlockingList to add this behavior to any class of objects that implement the List interface. Create a blocking list like this:

```
List blockingList =
    new BlockingList(
        new LinkedList() );
```

Many methods (such as size() and contains()) behave exactly as they do in the List, so they are implemented as simple pass-through methods. Other methods (such as add() and remove()) implement different behavior, so must be implemented at length in the Concrete Decorator.

The behavior of every method in the blocking version at left has changed, however: Everything is now synchronized. If only a handful of methods change behavior (or a Decorator just adds a method), simplify implementation with an abstract Decorator class that does nothing but define simple pass-through methods to the contained object. Extend the abstract class to form a Concrete Decorator, overriding those methods whose behavior changes.

Other Decorators may add other features. A LazyList may add a close() method that allows subsequent removals from the list but disallows additions, for example.

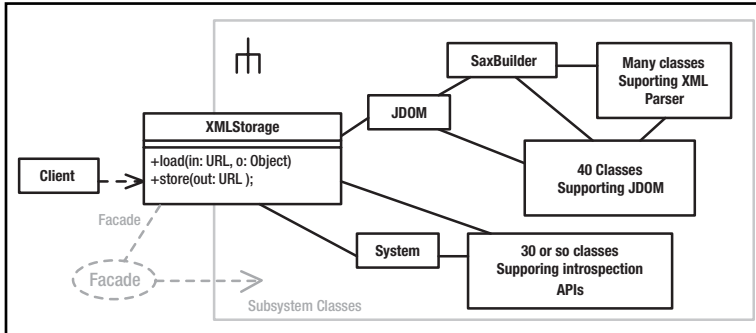


Combines *Decorator* and *Composite*: *Composite* because everything's a JComponent; *Decorator* because each successive layer adds functionality (and changes appearance).

The data source is wrapped by a Decorator that adds buffering, which is in turn wrapped by a decorator that supports pushback. Could add decompression, and so on, with additional decorators (GzipInputStream, and so on).

Facade

Provide a single interface through which all the classes in a complex subsystem are manipulated. Facade allows you to treat a complex subsystem as if it were a single coarse-grained object with a simple easy-to-use interface.



Facade: Provides a simple interface to a complex subsystem.

Subsystem Classes: Classes that comprise one or more complex subsystems.

What Problem Does It Solve?

Facade simplifies complex code, making it easier to use poorly designed, overcomplex subsystems.

Subsystems, especially older ones, are masses of spaghetti code. When two subsystems must interact, they often make calls directly into each other, and these myriad tendrils of connectivity are a maintenance nightmare. The subsystems become very delicate since making seemingly insignificant changes in a single subsystem can affect the entire program. Facade addresses the problem by forcing programmers to use a subsystem indirectly through a well-defined single point of access, thereby shielding the programmers from the complexity of the code on the other side of the facade.

Facade improves the independence of the subsystems, making it easy to change—or even replace—them without impacting outside code.

Facade also provides a manageable way to migrate legacy code to a more object-oriented structure. Start by breaking up the existing code into a small number of independent subsystems, modeled as very heavyweight objects with well-defined, simple interfaces. Eliminate all “end runs” around these interfaces. Then systematically replace each subsystem. This evolutionary approach significantly reduces the risk inherent in an all-at-once rewrite.

Facade hides badly done, overly complex legacy code.

Facade lets you treat an entire legacy system as if it were a single, coarse-grained object.

Pros (✓) and Cons (✗)

- ✓ Coupling relationships between subsystems are weakened, improving maintenance and flexibility.
- ✗ It's still possible for programmers to ignore the Facade and use subsystem classes directly.

Often Confused With

Bridge: Both Facade and Bridge help maintenance by isolating subsystems from each other. Facade simplifies access to, but does not hide, a subsystem. Bridge completely isolates you from the subsystem—you don't know that the subsystem exists if you use Bridge. You can use a Facade to simplify access to a bridge. (For example, a Company class could act as a facade to the JDBC Bridge. You'd say `Company.getEmployee()` and the Facade takes care of the complex series of JDBC calls needed to create the Employee object.)

Mediator: A Facade's communication with a subsystem is unidirectional, or at least simple. Your program sends a message to the Facade, which causes it to send several messages to a subsystem. The subsystem does not talk to, or even know about, the Facade object. Mediators have complex bidirectional conversations with their Colleagues.

See Also

Bridge, Mediator, Observer

Implementation Notes and Example

```
class XMLStorage
{
    public store(URL out, Object toStore)
    {
        /* Code goes here that uses the
        * introspection APIs in the System
        * class to get the class name and the
        * values of all the public fields in
        * the class. The name and the values of
        * those fields are then used to build a
        * JDOM tree, which is passed to an
        * "outputter" to send an XML
        * representation of the tree to the
        * OutputStream.
        */
    }

    public Object load(URL in);
    {
        /* Code goes here that creates a
        * JDOM SaxBuilder for the InputStream,
        * uses it to build a JDOM, instantiates
        * a class named in the XML file,
        * then initializes that class using
        * one of the constructors or a series
        * of get/set methods.
        */
    }
}
```

The problem with providing a full-blown example of a Facade is that there's entirely too much code to represent in 40 or so lines—that's the whole point of the pattern.

I'm imagining that the storage method uses Java's introspection APIs to analyze the document and discover the fields to save. (It could just save everything that's public, or it could look for JavaBean-style get/set methods.) I would use the JDOM XML APIs to build a tree representation of an XML output file and then send the tree to a JDOM "outputter" class that would write the appropriate XML to a file. The loading function reverses this process. By using the facade, you isolate yourself from all the mechanics of introspection, XML parsing, and JDOM.

Messaging is one way; there is no complex back-and-forth interaction between the XMLStorage facade and the subsystems that it uses. The Facade object simply builds a tree and then outputs the tree.

We have a facade within a facade here. The SAXBuilder class itself comprises a facade that isolates you from the mechanics of the SAX-parser subsystem.

The program can access the JDOM, XML, and Introspection APIs directly. Ease of maintenance is compromised if you do so and any of these subsystems change. You could avoid this problem by putting the subsystems in an inaccessible package (such as the *com.sun.xxx* packages in Java). A Singleton can then be used to get a Facade, through which all access occurs.

Usage

```
SomeString.matches("[a-zA-Z]{1,3}$");
```

String acts as a Facade for the regular-expression-matching package, isolating the user from things such as Pattern objects.

```
Socket s = new Socket("holub.com",7);
InputStream in = s.getInputStream();
```

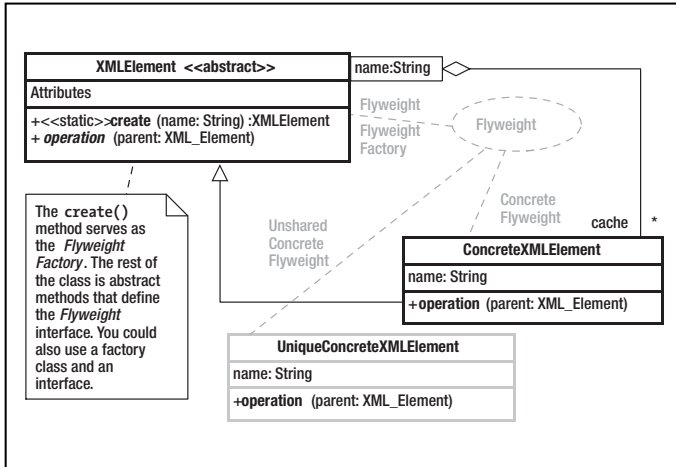
These two lines hide several pages of C code and all the enormous complexity needed to get a socket to work in a cross-platform way.

```
AppletContext a = getAppletContext();
a.showDocument("http://www.holub.com/index.html")
```

AppletContext is a Facade for the browser subsystem. Note that this architecture prohibits "end runs" around the facade because subsystem classes are accessible only through the Facade. You can't get at them directly.

Flyweight

To minimize memory usage, make objects smaller by using *extrinsic* state (for example, putting state information into a container or computing it on each access) and *sharing* (using multiple references to single objects rather than multiple copies of objects).



Flyweight: Defines an interface for messages that use extrinsic state.

Concrete Flyweight: Implements Flyweight with methods that compute state information or get it from an external source (extrinsic state).

Unshared Concrete Flyweight: Not used here, but if present, implements Flyweight using internal state variables rather than extrinsic state.

Flyweight Factory: Creates and manages flyweights. Supplies an existing Concrete Flyweight if one exists; otherwise creates one.

What Problem Does It Solve?

An object is defined by *what* it does, not how it does it. Objects are defined by their methods; state information can be inside or outside the object.

Sometimes, programs with large numbers of objects require more memory for those objects than is available. In a document editor, every character of a naïve implementation might hold its value, font, color, size, encoding, position on the page, and so on. This information, duplicated in most characters, can be moved to a containing paragraph. If characters take up more space than references, keep multiple references to a single “character” object rather than many identical characters.

A naïve implementation of the Game of Life “cell” may carry a Boolean “is-alive” state and references to eight neighbors. A small 1024×1024 grid requires about 40MB just to hold the cells. In a Flyweight version, the cell’s container knows who the cell’s neighbors are and passes that information to the cell. The cell needs to remember its is-alive state only. By making the neighbor references *extrinsic*, you reduce the memory requirement for the basic grid to a single megabyte.

In a “flyweight pool,” all objects with the same state are represented by a single object. You request objects from the pool, which returns an existing object if possible; otherwise, the pool creates a new object.

Pros (✓) and Cons (✗)

- ✓ Some programs simply cannot be written in an object-oriented way without using Flyweight.
- ✓ When you use flyweight pools, you can determine equality using Java’s == operator.
- ✗ If extrinsic state is stored in a container, then you must access the object through the container. If extrinsic state is computed (for example, goes to a database every time a particular attribute is used), then access is slow.
- ✗ Flyweights add complexity to the code, impacting maintenance and increasing code size.

Often Confused With

Composite: Flyweights are often combined with Composite. Both Leaf and Component nodes can export extrinsic state to their containers.

See Also

Composite, Prototype, Singleton

Implementation Notes and Example

```

abstract class XMLElement
{
    static Map cache = new HashMap();
    public static
    XMLElement create(String name)
    {
        name = name.intern();
        XMLElement exists =
            (XMLElement)(cache.get(name));
        if( exists == null )
        {
            exists =
                new ConcreteXMLElement(name);
            cache.put(name,exists);
        }
        return exists;
    }
    private XMLElement(){}

    abstract void operation( XMLElement parent );

    private static class ConcreteXMLElement
                        extends XMLElement
    {
        private String name;
        ConcreteXMLElement(String name)
        {
            this.name = name.intern();
        }
        void operation(XMLElement parent)
        {
            //...
        }
        public int hashCode()
        {
            return name.hashCode();
        }
        public boolean equals( Object o )
        {
            return name ==
                ((ConcreteXMLElement)o).name ;
        }
    }
}

```

XMLElement is a Flyweight that represents an Element in an XML Document (effectively a node in a tree). The element is identified only by name, though a more realistic implementation would identify it both by name and attribute values.

Sharing is used to guarantee that only one instance of a given element exists. (You could argue reasonably that XMLElement is a Singleton; that is, sharing is implemented using Singleton.) The private constructor (and the fact that it's abstract) force users to use `create()` rather than `new XMLElement()`. The `create()` method keeps a cache of XMLElement objects, keyed by name. If an object with the requested name exists, it is just returned. The `create()` method adds an element to the cache only if an element with that name does not already exist. If the Element doesn't need to know its own name, its name field can be eliminated to save space.

Don't be confused by the fact that XMLElement fills two roles in the pattern: Flyweight Factory and Flyweight. Putting the abstract methods of XMLElement into an interface to separating them from the "factory" functionality makes sense in many situations. Here, it just adds an unnecessary class.

The `intern()` method of the String class enforces sharing in a similar way (see "Usage").

XMLElement also has one extrinsic attribute: its parent. A heavyweight implementation may keep a parent reference as a field, but here the parent reference is passed as an argument to `operation()`. This organization saves space, but means that operations on elements that need to know their parent must be started at the root node so that the parent reference can be passed down to them.

Usage

```

JPanel p = new JPanel();
p.setBorder(
    BorderLayout.createEmptyBorder(5,5,5,5) );

```

The border size is extrinsic—it's fetched at runtime from the Component that it borders. The `BorderFactory` makes sure that two borders with the same internal state don't exist (when you ask for the second one, you get back a reference to the first one).

```

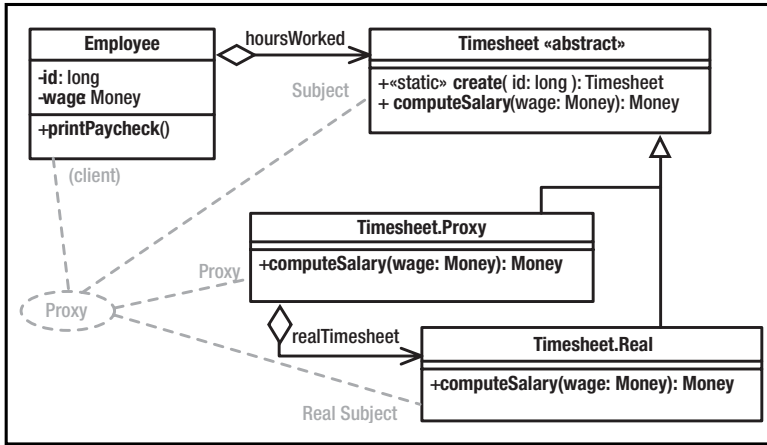
StringBuffer b = new StringBuffer();
//... assemble string here.
String s = b.toString().intern();

```

If an existing String literal has the same value as the assembled StringBuffer, then use the existing literal; otherwise, add the new value to the JVM's internal table of String literals and use the new one.

Proxy

Access an object through a “surrogate or placeholder” object.



Proxy: Maintains a reference, and controls access, to the Real Subject. Implements the same interface as the Real Subject so it can be used in place of the Real Subject.

Subject: An interface implemented by both the Proxy and the Real Subject; allows them to be used interchangeably.

Real Subject: The real object that the Proxy represents.

What Problem Does It Solve?

A *virtual proxy* creates expensive objects on demand. For example, database access may be deferred by a proxy until the data is actually used. A large image may be fetched across the network in the background while the user of the image thinks it’s already there. This process is often called *lazy instantiation*. Virtual proxies are also useful in implementing a *copy-on-write* strategy. When you request a copy of an object, you get back a proxy that simply references the original object. Only when you modify the so-called copy does the proxy actually copy the state from the original object into itself.

A *remote proxy* is a client-side representation of a server-side object. The proxy relays requests across the network to be handled by a sever-side object. CORBA and RMI stubs are proxies for server-side skeleton objects.

A *protection proxy* controls access to certain methods of a second object that implements the same interface. The proxy method may be passed an authentication token and throw an exception if the token didn’t authorize the requested operation. For example: the `Collection` implementation returned from `Collections.unmodifiableCollection(...)` is a protection proxy.

A *smart reference* automatically handles annoying background tasks such as deletion. Java’s `WeakReference` is an example.

Pros (✓) and Cons (✗)

- ✓ Proxies hide many optimizations from their users, simplifying the code considerably.
- ✗ Once the real object has been created, access through the proxy adds overhead. The whole point of the pattern is to be able to treat the proxy as if it were the real object, so a method such as `getRealObject()` violates the spirit of the pattern.
- ✗ You may need to use many remote proxies to talk to a large subsystem. It’s better to create a single remote proxy for a Facade than it is to create proxies for every class in the subsystem.

Often Confused With

Decorator: A protection proxy in particular could be looked at as a Decorator. There’s no difference in structure, but the intent is different—Decorator allows undecorated objects to be accessed indiscriminately.

See Also

Decorator, Flyweight

Implementation Notes and Example

```

class Employee
{ private long    id;
  private Money   wage; // hourly wage
  private Timesheet hoursWorked;
  public Employee(long id)
  { this.id    = id;
    hoursWorked = Timesheet.create(id);
    wage      = Database.getHourlyWage(id);
  }
  void printPaycheck()
  { Money weeklyWage =
    hoursWorked.computeSalary(wage);
    //...
  }
}
abstract class Timesheet
{ //...
  public static Timesheet create(long id)
  { return ( dataAlreadyInMemory )
    ? lookup(id)
    : new Proxy(id);
  }
  public abstract Money computeSalary(Money wage);
  //-----
  private static class Proxy extends Timesheet
  { Timesheet realTimesheet = null;
    long id;
    Proxy(long id){this.id = id;}
    public Money computeSalary(Money wage)
    { if( realTimesheet == null )
      realTimesheet = new Real(id);
      return realTimesheet.computeSalary(wage);
    }
  }
  //-----
  private static class Real extends Timesheet
  { Real(long employeeId)
    { // load data from the database.
    }
    public Money computeSalary(Money wage)
    { // Compute weekly salary.
      return null;
    }
  }
}
}

```

Usage

```

public void paint(Graphics g)
{ Image img=Toolkit.getDefaultToolkit().getImage(
  new URL("http://www.holub.com/image.jpg"));
  g.drawImage(img,...);
}

```

Assume that hourly wage is used heavily enough to justify a database access when the object is created but that the total hours worked is used only rarely and the Timesheet is needed by only a few methods.

I've made the employee identifier a long to simplify the example. In real code, it would be an instance of class Identity.

You could reasonably argue that that the Employee should just use lazy loading for the Timesheet and dispense with the Proxy object, but the more that Timesheet was used, the less weight this argument would hold.

I've made Timesheet an abstract class rather than an interface so that I can use it as a factory; otherwise, I'd need a separate TimesheetFactory class.

Accessor methods (get and set functions) are evil because they expose implementation detail and impact maintenance. Though it's tempting to use them in this pattern, you'll note that no getSalary() or getHoursWorked() method is needed because of the way I've structured the messaging system. Don't ask for information you need to do the work; ask the object that has the information to do the work for you. One exception to the get/set-is-evil rule is Database.getHourlyWage(). A database is fundamentally non-object-oriented; it's just a bag of data with no operations at all. Consequently, it must be accessed procedurally.

If the Timesheet.Proxy threw away the data after computing the salary, it would be a Flyweight, not a Proxy.

The object returned from getImage() is a proxy for the real image, which is loaded on a background thread. (getImage() is *asynchronous*; it returns immediately, before completing the requested work.) You can use the image as if all bits had been loaded, even when they haven't.

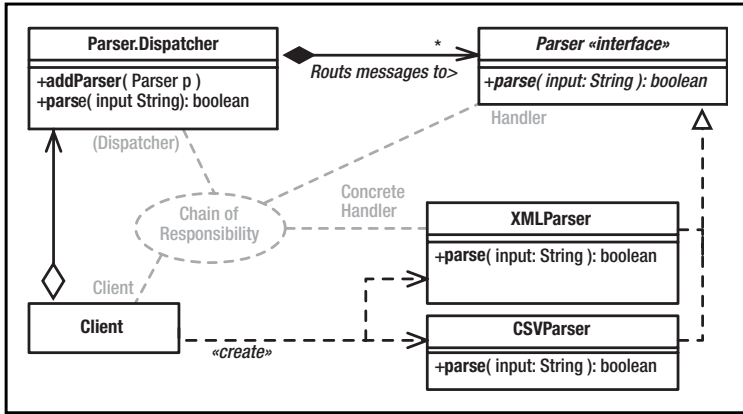
[This page intentionally left blank]

Behavioral Patterns

The behavioral patterns concern themselves with the runtime behavior of the program. I think of them as dynamic-model patterns. They define the roles that *objects* take on and the way that these objects interact with each other. For example, Chain of Responsibility defines the way that a set of objects routes messages to each other at runtime (so that the object best suited to field a message actually handles the message). All these objects are instances of the same class (or at least implement the same interface), so there's not much in the way of structure in Chain of Responsibility. It's the dynamic behavior of the objects that are important.

Chain of Responsibility

A set of objects relay a message from one to another along a well-defined route, giving more than one object a chance to handle the message. The object that's best suited to field a given message actually does the handling. It's possible for more than one object on the route to act on the message.



Handler: Defines event-handling interface and optional successor link.

Concrete Handler: Handles request or, by doing nothing, causes event to be forwarded to successor.

Dispatcher (not a Gang-of-Four role): Routes the event to each handler in turn. Not required if Handlers are organized as a linked list.

What Problem Does It Solve?

Makes it easy to add new customized message handling at runtime. An “event” or a message can be passed to a sequence of handlers, and the one that is best suited to handle the message actually does so.

In Microsoft Windows, every button, frame, and so on, is a Window object, arranged using the Composite pattern. *Events* (messages created by a user action) are passed to the window that has the focus (typically a button or other *widget*), and if that window can't handle the message, it passes it to its parent window (typically a frame window or a menu bar). The Window that understands the message actually handles it.

Servlet “filters” are another use of the pattern. An incoming HTTP packet is passed through a sequence of filters, which can process the packet directly or pass the packet to the next filter in the chain (or both).

Also consider a system of subclasses in which each constructor parses from a *String* the information of interest to it, and then it passes the *String* to the superclass constructor.

Pros (✓) and Cons (✗)

- ✓ The dynamic behavior of the program can be easily changed at runtime by adding new handlers to the chain or changing the ordering of handlers.

- ✓ The coupling between objects in the program is loosened if an implementation permits Handler classes not to know about each other.
- ✗ In Windows, when a mouse moves one pixel, the WM_MOUSEMOVE message is first received by the window that has focus, perhaps a text control. This control doesn't know how to handle it, so it passes the message to the containing panel, which passes it to the MDI child window, which passes it to the main frame, which passes it to the menu bar, which passes it to each menu item. None of these objects can handle the message, so it's discarded. This is a lot of work to do nothing.
- ✗ Many reifications force you to use implementation inheritance to specify a message handler, inappropriately forcing strong coupling between Handler classes and introducing fragile base classes into the model.

Often Confused With

Composite: Composite specifies one way that a Chain of Responsibility may be ordered (from contained object to container, recursively). This is not the only way to order the chain, however.

See Also

Composite, Observer

Implementation Notes and Example

```
interface Parser
{
    boolean parse( String input );
    static class Dispatcher
    {
        private List parsers = new LinkedList();
        public void addParser( Parser p )
        {
            parsers.add( p );
        }
        public boolean parse( String input )
        {
            for( Iterator i=parsers.Iterator();
                i.hasNext(); )
            {
                Parser p = (Parser)( i.next() );
                if( p.parse( input ) )
                    return true;
            }
            return false;
        }
    }
}

class XMLParser implements Parser
{
    private static Pattern scanner =
        Pattern.compile("^\\s*<");
    public boolean parse( String input )
    {
        Matcher regex = scanner.matcher(input);
        if( !regex.lookingAt() )
            return false;
        // Parse the XML file here.
        return true;
    }
}

class CSVParser implements Parser
{
    private static Pattern scanner =
        Pattern.compile(
            "[a-zA-Z0-9]*,[a-zA-Z0-9]+");
    public boolean parse( String input )
    {
        Matcher regex = scanner.matcher(input);
        if( !regex.matches() )
            return false;
        // Parse a comma-separated-value string
        return true;
    }
}
```

Create a parser for a particular input format by implementing the Parser interface. The two versions at left (XMLParser and CSVParser) handle XML and comma-separated-value formats. The parse(...) method examines the input, and if it recognizes the input format, it parses it and returns true; otherwise, parse(...) returns false. The Parser.Dispatcher() object just keeps a list of Parser implementations and passes the messages to them one at a time until it finds one that can handle the input string.

Parse an input string like this:

```
Parser.Dispatcher dispatcher =
    new Parser.Dispatcher();
dispatcher.addParser( new XMLParser() );
dispatcher.addParser( new CSVParser() );
//...
if( !dispatcher.parse( inputString ) )
    System.err.println("Can't parse input");
```

The Gang-of-Four reification does not have an object in the Dispatcher role. (The term dispatcher is mine.) In the Gang-of-Four reification, the Concrete Handler objects form a simple linked list, the input is passed to the first handler in the list, and any handler that can't process the input just delegates to the next handler in the list.

Usage

```
public class MyFilter
    implements javax.servlet.Filter
{
    //...
    public void doFilter(ServletRequest req,
        ServletResponse rsp,
        FilterChain chain)
    {
        //...
        chain.doFilter(request, response );
        //...
    }
}
```

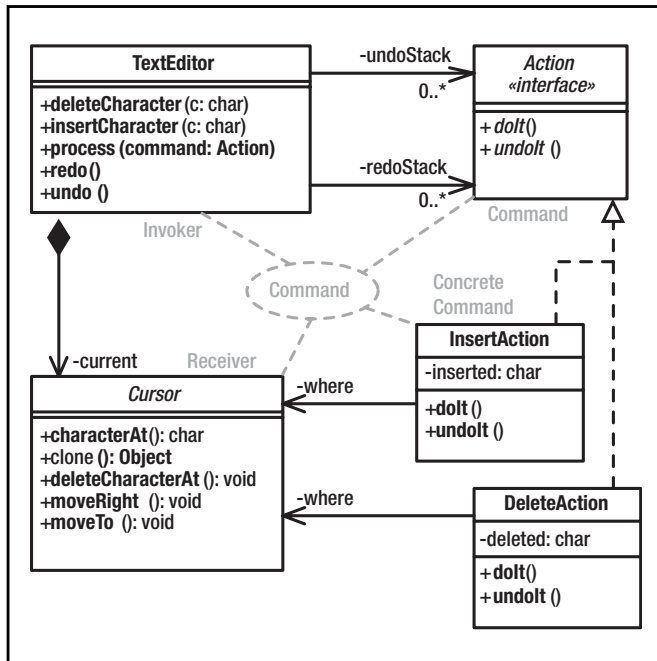
Each object on the route typically keeps a reference to its successor, but the pattern doesn't mandate this organization. For example, a centralized dispatcher may pass a message to several objects in turn. What's important is that the object, not the dispatcher, decides whether to handle the message. Servlet filters are dispatched by the web server. Tomcat, for example, uses information that you put into a configuration file to determine the dispatch sequence.

```
class MyWindow extends Component
{
    public boolean keyDown(Event e, int key)
    {
        // code to handle key press goes here.
    }
}
```

Chain-of-Command GUI handling was abandoned as unworkable in version 1.1 of Java (in favor of the better Observer pattern). This deprecated method is a holdover from then.

Command

Encapsulate a request or unit of work into an object. Command provides a more capable alternative to a function pointer because the object can hold state information, can be queued or logged, and so forth.



Command: Defines an interface for executing an operation or set of operations.

Concrete Command: Implements the Command interface to perform the operation. Typically acts as an intermediary to a Receiver object.

Invoker: Asks the command to carry out a request.

Receiver: Knows how to carry out the request. This functionality is often built in to the Command object itself.

What Problem Does It Solve?

You can't have a function pointer in an OO system simply because you have no functions, only objects and messages. Instead of passing a pointer to a function that does work, pass a reference to an object that knows how to do that work.

A Command object is effectively a transaction encapsulated in an object. Command objects can be stored for later execution, can be stored as-is to have a transaction record, can be sent to other objects for execution, and so on.

Command is useful for tasks such as “undo” operations. It's not possible to undo an operation simply by rolling the program back to a previous state; the program may have had an effect on the outside world while transitioning from the earlier state to the current one. Command gives you a mechanism for actively rolling back state by actively reversing side effects such as database updates.

By encapsulating the work in an object, you can also define several methods, and even state information, that work in concert to do the work. For example, a single object can

encapsulate both “undo” and “redo” operations and the state information necessary to perform these operations.

Command also nicely solves “callback” problems in multithreaded systems. A “client” thread creates a Command object that performs some operation and then notifies that client when the operation completes. The client then gives the Command object to a second thread on which the operation is actually performed.

Pros (✓) and Cons (✗)

- ✓ Command decouples operations from the object that actually performs the operation.

Often Confused With

Strategy: The invoker of a Command doesn't know what the Command object will do. A Strategy object encapsulates a method for doing a specific task for the invoker.

See Also

Memento, Strategy

Implementation Notes and Example

```

abstract class Cursor extends Cloneable
{
    public Object clone();
    public char characterAt();
    public void deleteCharacterAt();
    public void insertCharacterAt(
        char newCharacter);
    public void moveTo(Cursor newPosition);
    public void moveRight();
    //...
}
class TextEditor
{
    private Cursor current = new Cursor();
    private LinkedList undoStack =
        new LinkedList();
    private LinkedList redoStack =
        new LinkedList();
    public void insertCharacter(char c)
    {
        process( new Inserter(c) );
    }
    public void deleteCharacter()
    {
        process( new Deleter() );
    }
    private void process( Action command )
    {
        command.doIt();
        undoStack.addFirst(command);
    }
    public void undo()
    {
        Action action =
            (Action) undoStack.removeFirst();
        action.undoIt();
        redoStack.addFirst( action );
    }
    public void redo()
    {
        Action action =
            (Action) redoStack.removeFirst();
        action.doIt();
        undoStack.addFirst( action );
    }
    private interface Action
    {
        void doIt ();
        void undoIt();
    }
    private class Inserter implements Action
    {
        Cursor where = (Cursor) current.clone();
        char inserted;

```

```

        public Inserter(char newCharacter)
        {
            inserted = newCharacter;
        }
        public void doIt()
        {
            current.moveTo( where );
            current.insertCharacterAt(inserted);
            current.moveRight();
        }
        public void undoIt()
        {
            current.moveTo( where );
            current.deleteCharacterAt();
        }
    }
    private class Deleter implements Action
    {
        Cursor where = (Cursor) current.clone();
        char deleted;
        public void doIt()
        {
            current.moveTo( where );
            deleted = current.characterAt();
            current.deleteCharacterAt();
        }
        public void undoIt()
        {
            current.moveTo( where );
            current.insertCharacterAt( deleted);
            current.moveRight();
        }
    }
    //...
}

```

Most of the work is done by the `Cursor`, which reifies `Iterator`. The `TextEditor` is driven by a Client class (not shown) that interprets user input and tells the editor to perform tasks such as inserting or deleting characters. The `TextEditor` performs these request by creating `Command` objects that implement the `Action` interface. Each `Action` can both do something and also undo whatever it did. The editor tells the `Action` to do whatever it does and then stacks the object. When asked to undo something, the editor pops the `Action` off the undo stack, asks it to undo whatever it did, and then puts it on a redo stack. Redo works in a similar way, but in reverse.

Usage

```

new Thread()
{
    public void run(){ /*...*/ }
}.start();

```

`Thread` is passed a `Runnable Command` object that defines what to do on the thread.

```

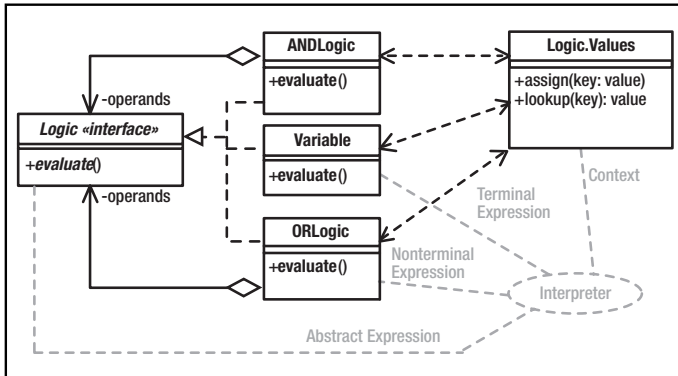
java.util.Timer t = new java.util.Timer();
t.schedule( new java.util.TimerTask()
    {
        public void run()
        {
            System.out.println("hello world");
        }
    }, 1000);

```

Print *hello world* one second from now. The `TimerTask` is a `Command` object. Several `TimerTask` objects may be queued for future execution.

Interpreter

Implement an interpreter for a language, first defining a formal *grammar* for that language and then implementing that grammar with a hierarchy of classes (one subclass per *production* or *nonterminal*).



Abstract Expression: Defines an “interpret” operation (or operations). A node in the abstract-syntax tree.

Terminal Expression: Implements an operation for a terminal symbol (which appears in the input).

Nonterminal Expression: Implements an operation for a nonterminal symbol (a grammatical rule).

Context: Global information (for example, variable values).

What Problem Does It Solve?

You sometimes cannot define all the required behavior of a program when you write it. For example, there’s no way for the browser writer to predict the way a site designer may want a web page to behave. An interpretive language such as JavaScript can add behavior to the program that wasn’t contemplated by the author.

The interpreter pattern defines one way to build an interpreter. You first define a formal *grammar* that lists rules (called *productions*) that describe the syntax of the language. You then implement a class for each production. These classes share a Context object from which they get input, store variable values, and so forth. An interpreter may (or may not) create an efficient output processor (such as a state machine) that does the actual work.

Pros (✓) and Cons (✗)

- ✗ The pattern says nothing about how to create the graph of objects that comprise the interpreter (the Abstract Syntax Tree). Interpreter often requires a nontrivial parser to construct this graph, and often this parser can just do the interpretation.
- ✓ Modifying the grammar is relatively straightforward; you just create new classes that represent the new productions.
- ✗ Interpreter doesn’t work well if the grammar has more than a few productions. You need too many classes. Use traditional compiler tools (such as JYACC, CUP, and so on) or a hand-coded recursive-decent parser for nontrivial languages.
- ✗ Why provide an interpreter when you have a perfectly good one already in memory:

the JVM? Your users write scripts in Java and provide you with a string holding the class name. Use Java’s introspection APIs to load and execute the user-supplied code, or, if the user code implements a well-defined interface, then execute directly. Given the following:

```
public interface UserExtension
{
    void doSomething();
}
```

instantiate and execute a user object like this:

```
String name =
    System.getProperty("user.extension");
class userMods = Class.forName(name);
UserExtension userExtensionObject =
    (UserExtension) userMods.newInstance();
userExtensionObject.doSomething();
```

Write your own class loader and/or security manager to create a sandbox.

Applets demonstrate this technique. Rather than interpret code (à la JavaScript), you provide a class to the browser, which it executes. Applets communicate with the browser via the AppletContext Facade.

Often Confused With

Chain of Responsibility: Chain of Responsibility is used in interpreter to evaluate the input *sentence*. It’s Interpreter *only* when the objects implement grammatical rules.

Composite: Interpreter is implemented as a Composite.

See Also

Strategy, Visitor

Implementation Notes and Example

```
interface Logic
{
    public static class Values
    {
        static Map vars = new HashMap();
        static void assign( String key,
                           boolean value )
        {
            if(key==null || key.length() <= 0)
                throw new Exception("Logic");
            vars.put(key, value? Boolean.TRUE
                    : Boolean.FALSE);
        }
        static boolean lookup( String key )
        {
            Object got = vars.get(key);
            return ((Boolean)got).booleanValue();
        }
    }
    boolean evaluate();
}

class ANDLogic implements Logic
{
    Logic left, right;
    public ANDLogic(Logic left, Logic right)
    {
        this.left = left;
        this.right = right;
    }
    public boolean evaluate()
    {
        return left.evaluate()
            && right.evaluate();
    }
}

class ORLogic implements Logic{ /*...*/ }
class NOTLogic implements Logic{ /*...*/ }

class AssignmentLogic implements Logic
{
    Logic left, right;
    public AssignmentLogic(Logic l, Logic r)
    {
        this.left = l;
        this.right = r;
    }
    public boolean evaluate()
    {
        boolean r = right.evaluate();
        Logic.Values.assign(left.toString(),r);
        return r;
    }
}

class Variable implements Logic
{
    private String name;
    public Variable(String s){name = s;}
    public String toString(){ return name; }
    public boolean evaluate()
    {
        return Logic.Values.lookup(name);
    }
}
}
```

Usage

```
java.util.regex.Pattern p=Pattern.compile("a*b");
java.util.regex.Matcher m = p.matcher("aaaaab");
boolean b = m.matches();
```

Consider the following Boolean-expression grammar:

```
e ::= e '&' e
    | e '|' e
    | '!' e
    | '(' e ')'
    | var '=' e
    | var
```

The code at the left comprises an interpreter for that grammar. (I haven't shown ORLogic and NOTLogic classes, since they're trivial variants on ANDLogic.) Variable values are held in the Values Singleton. Create an interpreter for $X=(A \ \& \ B) \ | \ !C$ as follows:

```
Logic term = new ANDLogic(
    new Variable("A"),
    new Variable("B")
);
term = new ORLogic(
    term,
    new NOTLogic( new Variable("C") )
);
term = new AssignmentLogic(
    new Variable("X"), term );
```

Assign values in the code (or by reading user input) like this:

```
Logic.Values.assign("A", true);
Logic.Values.assign("B", true);
Logic.Values.assign("C", false);
boolean result = term.evaluate();
```

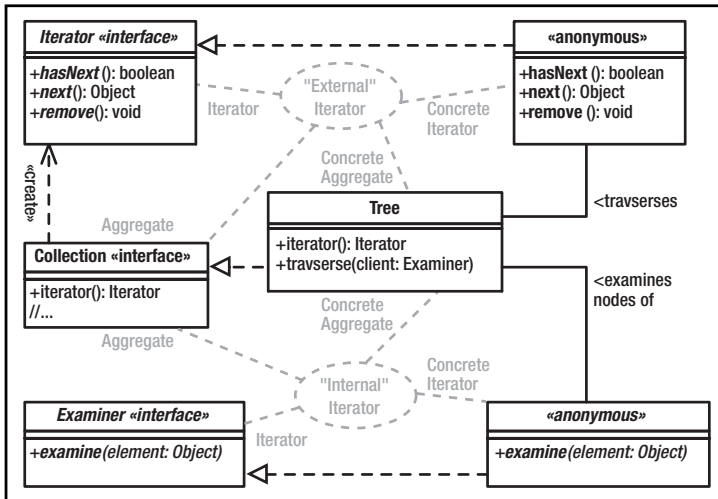
The Interpreter pattern makes no suggestions as to *how* you may construct the abstract-syntax tree that represents the expression (the tree of Logic objects), but some sort of parser is implied.

Alternatively, you could use Visitor to traverse the syntax tree: Visit the notes in depth-first order; code in the visitor object determines what happens as it visits each node. You could traverse once to test internal integrity, traverse again to optimize the tree, traverse a third time to evaluate the expression, and so on. Separating the structure of the abstract syntax tree from the logic of code generation and optimization can clean up the code substantially, but the visitor can end up as a quite-large class and will be hard to maintain as a consequence.

Uses *Interpreter* internally. (See the source code shipped with the JDK.)

Iterator

Access the elements of an aggregate object sequentially without exposing how the aggregation is implemented.



Iterator: Interface for accessing and traversing elements.

Concrete Iterator: Implements Iterator and keeps track of current position.

Aggregate: Defines an interface for creating an iterator. (Omit if no Abstract Factory required.)

Concrete Aggregate: Holds the data. Implements the creation interface to manufacture an iterator.

What Problem Does It Solve?

Iterators isolate a data set from the means that's used to store the set. For example, Java Collection and Map classes don't implement a common interface. You can, however, extract an iterator from a Collection (using `iterator()`) and from a Map (using `values().iterator()`). Pass the iterator to a method for processing, thereby isolating that method from knowledge of how the objects are stored.

The set of objects need not be stored internally at all—an iterator across a *Flyweight* may read objects from disk or even synthesize them.

Iterators make it easy to have multiple simultaneous iterators across an aggregation.

Iterators can manipulate the aggregation. The *Cursor* class in the Command example is an iterator. Java's `ListIterator` can modify the list.

External or *active* iterators are controlled by the client (for example, Java's `Iterator` class).

Internal or *passive* iterators are controlled by the aggregate object. A tree may have a `traversePostOrder()` method that's passed a `Command` object that is, in turn, passed each node in the tree. External iterators are often harder to implement than internal ones.

Pros (✓) and Cons (✘)

- ✓ Promotes reuse by hiding implementation.
- ✘ A client may modify the elements of the aggregation, damaging the aggregate (for example, change the key in sorted aggregate).

- ✘ The aggregate may store references to its iterators; memory leaks are possible if you discard an iterator without notifying the aggregate.
- ✘ It's difficult to control the traversal algorithm and retain the generic quality of an iterator. For example, there's no way to specify a post-order traversal from the iterator returned from a `TreeSet`. This problem extends to most Composite reifications.
- ✘ It's difficult to implement Iterator in an environment that supports simultaneous iteration and modification. (If you add an item to an aggregate while iterations are in progress, should the iterator visit the newly added item? What if the list is ordered and you've already passed the place where the new item is inserted? Should attempts to modify the aggregation fail if iterators are active? There are no "correct" answers to these questions.)

Often Confused With

Visitor: Visitor can be implemented with a passive iterator. Iterators should examine data, not modify it. Visitors are passed from node to node; Iterators are passed the node to visit.

See Also

Composite, Visitor

Implementation Notes and Example

```

class Tree implements Collection
{ private Node root = null;
  private static class Node
  { public Node left, right;
    public Object item;
    public Node( Object item )
    { this.item = item; }
  }
  Iterator iterator()
  { return new Iterator()
  { private Node current = root;
    private LinkedList stack =
      new LinkedList();
    public Object next()
    { while( current != null )
      { stack.addFirst( current );
        current = current.left;
      }
      if( stack.size() != 0 )
      { current = (Node)
        ( stack.removeFirst() );
        Object toReturn=current.item;
        current = current.right;
        return toReturn;
      }
      throw new NoSuchElementException();
    }
    public boolean hasNext()
    { return !(current==null
      && stack.size()==0);
    }
    public void remove(){ /*...*/ }
  };
}
public interface Examiner
{ public void examine( Object o ); }
void traverse( Examiner client )
{ traverseInorder( root, client );
}
private void traverseInorder(Node current,
  Examiner client )
{ if( current == null )
  return;
  traverseInorder(current.left, client);
  client.examine (current.item );
  traverseInorder(current.right, client);
} // ...
}

```

Usage

```

f( Collection c )
{ Iterator i = c.iterator();
  while( i.hasNext() )
    doSomething( i.next() );
}

```

Iterators are used heavily in all the Java Collection classes.

```

String query = "SELECT ID FROM TAB";
ResultSet results = stmt.executeQuery(query);
while( results.next() )
  String s = results.getString("ID");

```

A database cursor iterates across rows in a table.

The previous code implements a simple binary tree. (I've omitted the methods of Collection that aren't relevant to Iterator.) The iterator() method returns an external iterator that implements the java.util.Iterator interface. Use it like this:

```

Iterator i = t.iterator();
while( i.hasNext() )
  System.out.print(i.next().toString() );

```

You can't use recursive traversal in an external iterator because next() must return after getting each element, and you can't stop the recursion in midstream. My implementation uses a stack to remember the next parent to visit in the traversal (the same information that would be on the runtime stack in a recursive traversal). You can easily see the extra complexity mandated by this approach, but other nonrecursive traversal algorithms are, if anything, messier.

The traverse() method demonstrates an internal iterator. You pass traverse() a Command object that implements the Examiner interface. Traverse does a simple recursive traversal, passing each node to the Examiner's examine() method in order. Here's an example:

```

t.traverse(
  new Tree.Examiner()
  { public void examine(Object o)
    { System.out.print(o.toString());
    }
  } );

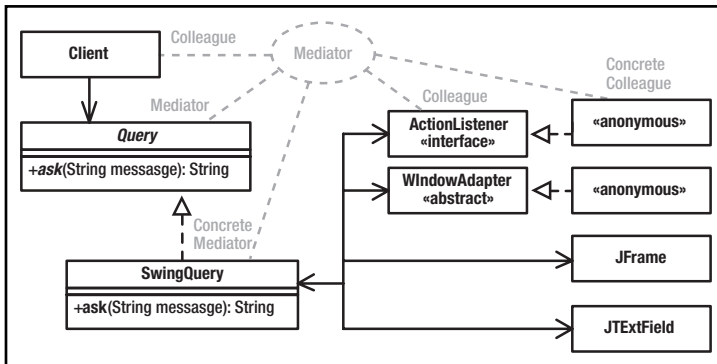
```

As you can see, the code is much simpler, but you lose the flexibility of an external iterator (which you could keep positioned in the middle of the tree, for example; an internal iterator doesn't give you the option of not advancing).

Both iterators access private fields of Tree. Think of an external iterator as an extension of the object that creates it. Private access is okay if it doesn't expose implementation information. Nonetheless, iterators are tightly coupled to the aggregate by necessity.

Mediator

Define a single object that encapsulates a set of complex operations. Mediator simplifies your code by hiding complexity; it loosens coupling between the objects that use the mediator and the objects the mediator uses.



Mediator: (Often omitted.) Defines an interface to Colleagues.

Concrete Mediator: Implements the Mediator interface to interact with Colleagues and manage communication between them.

Colleagues: A system of interfaces and classes that communicate bidirectionally through the mediator rather than directly. Note that the client is a Colleague.

What Problem Does It Solve?

Mediator makes complex operations simple.

Too-complex code is damaging to any program. Mediator solves this problem by taking complex code that would otherwise appear all over the program and encapsulating it into a single object with a simple interface that's used all over the program. Mediators hide complex protocols.

Pros (✓) and Cons (✗)

- ✓ Mediators improve code organization in many ways: reducing subclassing, decoupling subsystems, and simplifying messaging systems and protocols.
- ✗ Complexity can creep into a Mediator over time as you customize it for new applications. You've missed the point if you allow a Mediator to become too complex. Several Mediators tailored for specific applications can help. Be careful not to add back the complexity you're trying to eliminate.
- ✗ A mediator can turn into a "god" class if you're not careful. A good OO program is a network of cooperating agents. There is no spider in the middle of the web pulling the strands. Focus your mediators on doing one thing only.

Often Confused With

Facade: Facade eases simple one-way communication with a subsystem helps isolate the subsystem from the rest of the program. Mediators encapsulate complex interactions, but communication is bidirectional and they do not isolate anything from anything. It's possible, however, for a set of classes to participate simultaneously in both patterns.

Bridge: Bridge and Mediator both reduce coupling between subsystems. Bridge defines a standard (often complicated) interface and then implements it in various ways. Bridges are systems of *classes*. Mediators are *objects* that have simple interfaces but do complex work at runtime. Mediator does promote decoupling, though. If a protocol changes, for example, the scope of that change is typically limited to the Mediator itself.

See Also

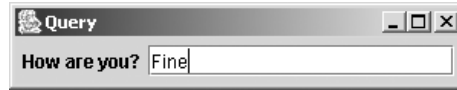
Facade, Bridge

Implementation Notes and Example

```
interface Query
{ String ask( String question );
}
class SwingQuery implements Query
{ public String ask( String question )
{ final Object done = new Object();
  final Object init = new Object();
  final JFrame frame = new JFrame("Query");
  final JTextField answer= new JTextField();
  answer.setPreferredSize(
    new Dimension(200,20));
  frame.getContentPane().setLayout(
    new FlowLayout() );
  frame.getContentPane().add( answer );
  frame.getContentPane().add(
    new JLabel(question) );
  answer.addActionListener // submit
  ( new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { synchronized( init )
        { synchronized( done )
          { frame.dispose();
            done.notify();
          }
        }
      }
    }
  );
  frame.addWindowListener // cancel
  ( new WindowAdapter()
    { public void windowClosing(WindowEvent e)
      { synchronized( init )
        { synchronized( done )
          { frame.dispose();
            answer.setText("");
            done.notify();
          }
        }
      }
    }
  );
  synchronized( done )
  { synchronized( init )
    { frame.pack();
      frame.show();
    }
    try{ done.wait(); }
    catch( InterruptedException e ){ }
  }
  return answer.getText();
}
}
```

The previous code lets you ask the user a simple question. When you make this call:

```
Query user = new SwingQuery();
String answer = user.ask("How are you");
```



The method displays the small window shown previously. You type your answer and hit Enter, the window shuts down, and `ask(...)` returns what you typed (in this case, the string "Fine"). If you click the X box in the upper-right corner of the control, the window shuts down and `ask(...)` returns an empty string.

The details of the code are actually not relevant to the current discussion. The main issue is that the code encapsulates a complex interaction with the GUI subsystem (and would be even more complex if you were working in the raw OS rather than Java), but the user exercises all this complexity by doing a simple thing. The details are all hidden. Moreover, code that *uses* `Query` is now considerably simplified, and a lot of complicated junk isn't duplicated all over the program.

It would be better if the Concrete Mediator (the `SwingQuery`) were created by an Abstract Factory rather than by invoking `new`.

Note that Mediator does not isolate the program from the entire Swing subsystem (unlike Bridge). Mediator does not prohibit other parts of your program from talking directly to Swing. Also note that the communication between the Mediator (`Query`) and its colleagues (everything else) is bidirectional, and that all communication (at least in the context of asking the user a question) happens through the mediator.

Usage

```
URL home = new URL("http://www.holub.com");
URLConnection c = home.getConnection();
//...
OutputStream out = c.getOutputStream();
c.write( someData );
```

The output stream returned from the `URLConnection` is a Mediator. You just write data to it. It encapsulates the complex interaction needed to establish a connection and implement whatever protocol was specified in the original URL.

```
JButton b = new JButton("Hello");
//...
```

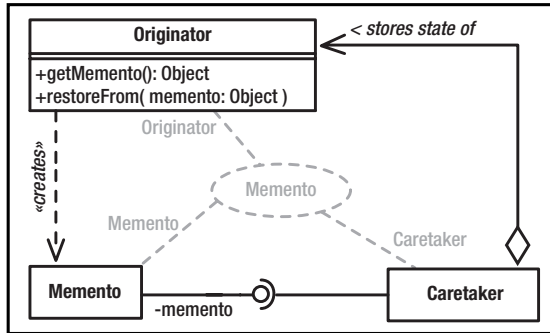
The `JButton` deals with all the complexity of fielding mouse clicks, changing the image the user sees when the button should be "down," and so on.

```
JOptionPane.showMessageDialog("Error... ");
```

Hides the complexity of creating and showing a dialog box.

Memento

Encapsulate an object's state in such a way that no external entity can know how the object is structured. An external object (called a *caretaker*) can store or restore an object's state without violating the integrity of the object.



Originator: Creates a memento that holds a “snapshot” of its current state.

Memento: Stores the internal state of the Originator in a way that does not expose the structure of the Originator. Supports a “wide” interface used by the originator and a “narrow” interface used by everyone else.

Caretaker: Stores the mementos but never operates on them.

What Problem Does It Solve?

The ubiquitous *get/set* (*accessor*) function is evil. Allowing access to internal fields—either directly by making them public or indirectly through an accessor method—flies in the face of the basic object-oriented principle of implementation hiding. The whole point of an OO structure is that you can make radical changes to an object's implementation without impacting the code that uses those objects. An object should not get the data that it needs to do work—it should ask the object that has the data to do the work for it (delegation). The only exception to this rule is an accessor that returns an object that opaquely encapsulates the data. The point is not to expose implementation details.

If you use simplistic accessors, even small changes, such as changing a field's type, impact every part of the program that uses that accessor. Programs that use accessors are difficult to maintain and simply aren't object oriented. (A program isn't OO just because it uses classes, derivation, and so on, or is written in Java or C++.)

But what if an external entity needs to remember the state of some object, perhaps to restore that state in an undo operation or equivalent? Memento solves this problem by having the original object return a black box, an impenetrable container that the caretaker can store but not manipulate. The object that manufactures the black box *does* know what's in it, though, so it can use this information at will (to restore state, for example).

Pros (✓) and Cons (✗)

- ✓ Allows an object's state to be stored externally in such a way that the maintainability of the program is not compromised.
- ✓ Allows a “caretaker” object to store states of classes that it knows nothing about.
- ✗ Versioning can be difficult if the memento is stored persistently. The Originator must be able to decipher mementos created by previous versions of itself.
- ✗ It's often unclear whether a memento should be a “deep” copy of the Originator. (in other words, should recursively copy not just references but the objects that are referenced as well). Deep copies are expensive to manufacture. Shallow copies can cause memory leaks, and referenced objects may change values.
- ✗ Caretakers don't know how much state is in the memento, so they cannot perform efficient memory management.

Often Confused With

Command: Command objects encapsulate operations that are known to the invoker. Mementos encapsulate state—operations are unknown to the caretaker.

Implementation Notes and Example

```
class Originator
{   private String   state;
    private int     more;

    private class Memento
    {   private String state =
        Originator.this.state;
        private int more =Originator.this.more;
        public toString()
        {   return state + ", " + more ;
        }
    }

    public Object getMemento()
    {   return new Memento();
    }

    public Object restore(Object o)
    {   Memento m = (Memento) o;
        state = o.state;
        more  = o.more;
    }
}

class Caretaker
{   Object memento;
    Originator originator;
    public void captureState()
    {   memento = originator.getMemento();
    }
    public void restoreYourself()
    {   originator.restore( memento );
    }
}
```

Making Memento private with nothing but private fields guarantees that unsafe access is impossible. (Some idiot may try to circumvent encapsulation using the introspection APIs,

but “against stupidity, even the gods themselves contend in vain.”) The Caretaker treats the Memento as a simple Object. Memento defines a “narrow” interface (`toString()`) that doesn’t expose structure. A much more complicated memento is presented in Chapter 3 in the Game-of-Life example.

One great example of Memento is an “embedded” object in Microsoft’s Object Linking and Embedding (OLE) framework. Consider an Excel spreadsheet that you’ve embedded as a table in a Word document. When you create the table, Excel is running. It negotiates with Word to take over some of its UI (Excel adds menus to Word’s menu bar and is in control of the subwindow that holds the table, for example). When you click outside the table, Excel shuts down and produces a memento—a blob of bytes that holds its state—and an image that Word displays in place of the original Excel UI. All that Word can do with this image is display it. All that Word can do with the data “blob” is hold onto it. The next time the user wants to edit the table, Word passes the blob back to Excel, but Excel has to figure out what to do with it. Since Excel’s data representation is completely hidden from Word, it can change the representation without impacting any of the code in Word itself.

A memento can have a “narrow” interface that does something such as display its state on a screen or store its state as an XML file. Just make sure that this interface doesn’t expose any structure to the caretaker.

“Undo” is hardly ever implementable solely with a memento (see “Command”).

Usage

```
class Originator implements Serializable{ int x; }

ByteArrayOutputStream bytes = new ByteArrayOutputStream();
ObjectOutputStream out= new ObjectOutputStream( bytes );

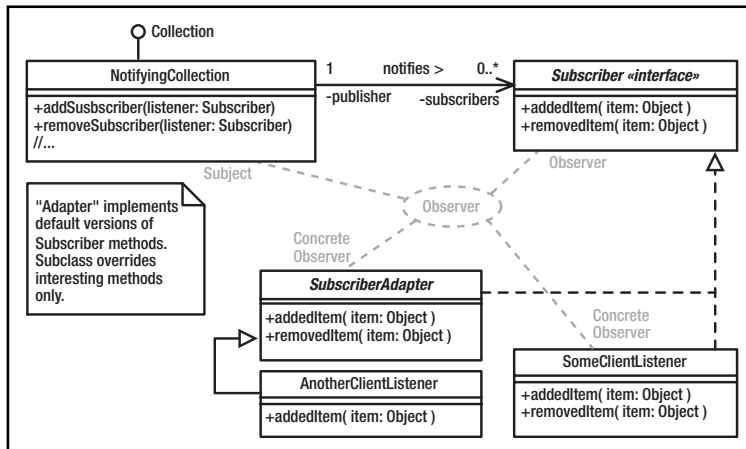
Originator instance = new Originator(); // create
out.writeObject( instance );          // memento
byte[] memento = bytes.toByteArray();

ObjectInputStream in = // restore object
    new ObjectInputStream( // from memento
        new ByteArrayInputStream(memento));
instance= (Originator) in.readObject();
```

A byte array is about as black as a box can be. Decorator is used here to produce a system of streams that manufacture the memento. This example also nicely illustrates a flaw in Decorator—that you sometimes have to access an encapsulated decorator to do work.

Observer (Publish/Subscribe)

When an object changes states, it notifies other objects that have registered their interest at runtime. The notifying object (publisher) sends an event (publication) to all its observers (subscribers).



Subject (The *publisher*). Notifies Observers that some event has occurred. Keeps a *subscription* list and a means for modifying the list. Sometimes Subject is an interface implemented by a Concrete Subject.

Observer (The *subscriber*). Defines an interface for notifying Observers.

Concrete Observer: Implements the Observer interface to do something when notified.

What Problem Does It Solve?

In Chain of Responsibility, a button notifies a parent of a press event like this:

```
class Window
{ void buttonPressed() { /*...*/ }
  //...
}
```

```
class Button implements Window
{ private Window parent;
  public Button(Window parent)
  { this.parent = parent; }
  public void onMouseClick()
  { parent.buttonPressed(); }
}
```

An abstraction-layer (business) object must learn about presses through a Mediator called a *controller*—a Window derivative that overrides `buttonPressed()` to send a message to the business object. The coupling relationships between the controllers, the abstraction layer, and the presentation (the button) are too tight. Too much code is affected if anything changes.

The Observer pattern addresses the problem by adding an interface between the *publisher* of an event (the button) and a *subscriber* (the business object that's actually interested in the button press). This interface decouples the publisher and makes it reusable in the sense that it's a stand-alone component, with no dependencies on the rest of the system. A Subject can notify any class that implements the Observer interface, as compared to the earlier example, where a Button could notify only a Window.

Pros (✓) and Cons (✗)

- ✓ Observer nicely isolates subsystems, since the classes in the subsystems don't need to know anything about each other except that they implement certain “listener” interfaces. This isolation makes the code much more reusable.
- ✗ You have no guarantee that a subscriber won't be notified of an event after the subscriber cancels its subscription—a side effect of a thread-safe implementation. (AWT and Swing both have this problem.)
- ✗ Publication events can propagate alarmingly when observers are themselves publishers. It's difficult to predict that this will happen.
- ✗ Memory leaks are easily created by “dangling” references to subscribers. (When the only reference to an Observer is the one held by a Subject, a dangling Concrete Observer may not be garbage collected.) It's difficult in Java, where there are no “destructor” methods, to guarantee that publishers are notified when a subscriber becomes irrelevant, and it's easy to forget to explicitly cancel the subscription.

Often Confused With

Command: Command objects are very generic. Observers are used solely for notification.

Strategy: Strategy objects define a strategy for performing some work. Observers do implement a notification strategy but, unlike Strategy objects, are not called from within methods to do work.

See Also

Chain of Responsibility

Implementation Notes and Example

```
public final class NotifyingCollection
    implements Collection
{
    private final Collection c;
    public NotifyingCollection(Collection wrap)
    { c = wrap; }
    private final Collection subscribers
        = new LinkedList();
    public interface Subscriber
    { void addItem ( Object item );
      void removedItem ( Object item );
    }
    synchronized public void addSubscriber(
        Subscriber subscriber)
    { subscribers.add( subscriber ); }
    synchronized public void removeSubscriber(
        Subscriber subscriber)
    { subscribers.remove( subscriber );
    }

    private void notify(boolean add, Object o)
    { Object[] copy;
      synchronized(this)
      { copy = subscribers.toArray();
      }
      for( int i = 0; i < copy.length; ++i )
      { if( add )
        { ((Subscriber)copy[i]).addItem(o);
        }
        else
        { ((Subscriber)copy[i]).removeItem(o);
        }
      }
    }

    public boolean add(Object o)
    { notify(true,o); return c.add(o); }
    public boolean remove(Object o)
    { notify(false,o); return c.remove(o); }
    public boolean addAll(Collection items)
    { Iterator i = items.iterator()
      while( i.hasNext() )
        notify( true, i.next() );
      return c.addAll(items);
    }
    // pass-through implementations of other
    // Collection methods go here...
}
```

The example at left is a Decorator that wraps a collection to add a notification feature. Objects that are interested in finding out when the collection is modified register themselves with the collection.

In the following example, I create an “adapter” (in the Java/AWT sense, this is *not* the Adapter pattern) that simplifies subscriber creation. By extending the adapter rather than implementing the interface, I’m saved from having to implement uninteresting methods. I then add a subscriber, like so:

```
class SubscriberAdapter implements
    NotifyingCollection.Subscriber
{
    public void addItem(Object item){}
    public void removedItem(Object item){}
}

NotifyingCollection c =
    new NotifyingCollection(new LinkedList());
c.addSubscriber
( new SubscriberAdapter()
  { public void added( Object item )
    { System.out.println("Added " + item);
    }
  }
)
```

This implementation of Observer is simplistic—copy is a very inefficient strategy for solving the problem of one thread adding or removing a subscriber while notifications are in progress. A more realistic implementation was presented in Chapter 3.

Observer encompasses both one-to-many and many-to-one implementations. For example, one button could notify several observers when it’s pressed, but by the same token, several buttons could all notify the same subscriber, which would use some mechanism (perhaps an event object passed as an argument) to determine the publisher.

Usage

```
JButton b = new JButton("Hello");
b.addActionListener(
    new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { System.out.println("World");
      }
    }
);
```

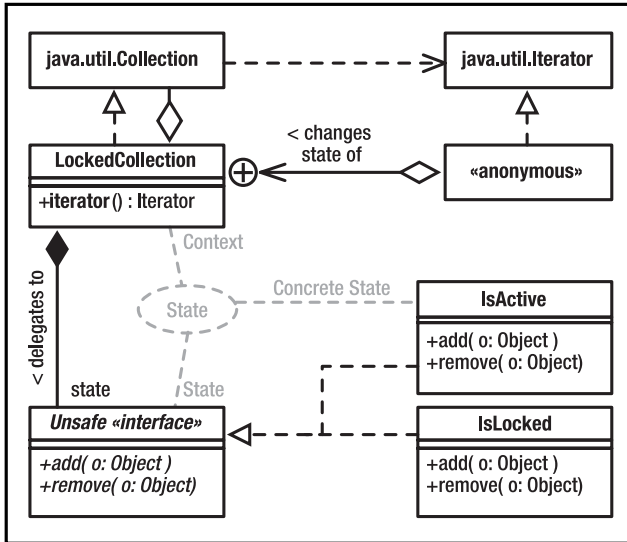
Print *World* when the button is pressed. The entire AWT event model is based on Observer. This model supersedes a Chain-of-Responsibility-based design that proved unworkable in an OO environment.

```
Timer t = new java.util.Timer();
t.scheduleAtFixedRate( new TimerTask()
    { public void run()
      { System.out.println(new Date().toString());
      }
    }
, 0, 1000 );
```

Print the time once a second. The `Timer` object notifies all its observers when the time interval requested in the schedule method elapses.

State

Objects often need to change behavior when they are in certain states. Define an interface comprising all methods that change behavior; interface implementations define behavior for each state.



Context: Defines a public interface to the outside world, methods of which change behavior with object state. Maintains an instance of a Concrete State class.

State: Defines an interface that comprises all the behavior that changes with state.

Concrete State: Implements State to define behavior for a particular state.

What Problem Does It Solve?

Objects often need to change behavior with state. The “obvious” way to implement this change is for each method to contain a large switch statement or equivalent, with a case for each possible state, and the selector is an instance variable. This structure is difficult to maintain at best, and changing the state table or introducing new states is difficult, requiring many changes to many methods.

In the State pattern, each state is represented by a State object that implements the behavior of the Context object when the Context is in a given state. An instance variable references an object that implements the current state’s behavior. A public method that changes behavior with state just delegates to the current state object. To change state, modify the current state reference to reference an object that implements behavior for the new state.

Pros (✓) and Cons (✗)

- ✓ State machines are easier to maintain since all the behavior for a given state is in one place.
- ✓ Eliminates long, hard-to-maintain switch statements in the methods.
- ✗ State tables (indexed by current state and stimulus, holding the next state) are difficult to implement.
- ✗ Increases the number of classes in the system along with concomitant maintenance problems.
- ✗ If only a few methods change behavior with state, this solution may be unnecessarily complex.

Often Confused With

Strategy: The state objects do implement a strategy for implementing a single state, but that strategy is not provided by an outside entity.

See Also

Singleton

Implementation Notes and Example

```

public final class LockedCollection
    implements Collection
{ private final Collection c;
  private int activeIterators = 0;

  private Unsafe active = new IsActive();
  private Unsafe locked = new IsLocked();
  private Unsafe state = active;

  public LockedCollection(Collection c)
  { this.c = c;
  }
  public Iterator iterator()
  { final Iterator wrapped = c.iterator();
    ++activeIterators;
    state = locked;

    return new Iterator()
    { private boolean valid = true;
      //...
      public boolean hasNext()
      { return wrapped.hasNext();
      }
      public Object next()
      { Object next = wrapped.next();
        if( !hasNext() )
        { if( --activeIterators == 0 )
          state = active;
          valid = false;
        }
        return next;
      }
    };
  }
  public int size()
  { return c.size(); }
  public boolean isEmpty()
  { return c.isEmpty(); }
  // ...
  // Collection methods that don't
  // change behavior are defined here.

  public boolean add(Object o)
  {return state.add(o);}
  public boolean remove(Object o)
  {return state.remove(o);}

  private interface Unsafe
  { public boolean add(Object o);
    public boolean remove(Object o);
    //...
  }
  private final class IsActive
    implements Unsafe
  { public boolean add(Object o)
    {return c.add(o);}
    public boolean remove(Object o)
    {return c.remove(o);}
  }

```

```

//...
}
private final class IsLocked
    implements Unsafe
{ public boolean add(Object o)
  { throw new Exception("locked"); }
  public boolean remove(Object o)
  { throw new Exception("locked"); }
  //...
}
}

```

This code combines Decorator, Abstract Factory, and State. It implements a Collection that changes behavior when iterators are active. *Active* means that an iterator has been created, but the last element of the Collection has not been examined through that iterator. (Java's Collection implementations do just that, but it makes a good example.) The class tosses an exception if you attempt to modify a collection while iterators are active.

The Unsafe interface defines those Collection methods that are unsafe to call during iteration. This interface is implemented by two classes: IsActive implements normal collection behavior. IsLocked implements the iterators-are-active behavior. The classes are Singletons whose instances are referenced by active and locked. The variable state defines the current state and points to one or the other of the Singletons.

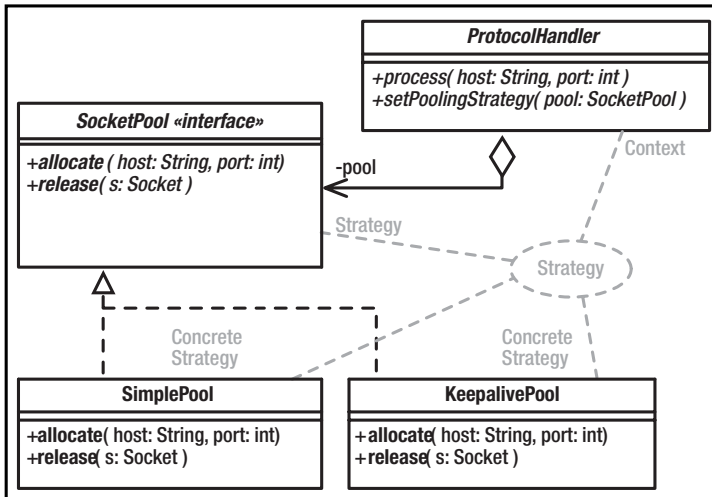
Public methods that don't change state with behavior (such as size()) delegate to the contained Collection, c. Public methods that do change state (such as add(Object)) delegate to whichever state object is referenced by state. The iterator() method forces a change of state to locked when it issues an iterator. It also increments an active-iterator count. This count is decremented by the Iterator's next() method when it reaches the last element, and when the count goes to zero, the active state is activated.

The iterator also changes behavior with state, but only one method is affected, so the State pattern isn't used.

No reason exists why you can't create new objects each time a state transition is made. This way the individual state object can itself keep local state information.

Strategy

Define an interface that defines a strategy for executing some algorithm. A family of interchangeable classes, one for each algorithm, implements the interface.



Strategy: An interface that allows access to an algorithm.

Concrete Strategy: Implements a particular algorithm to conform to the Strategy interface.

Context: Uses the algorithm through the Strategy interface.

What Problem Does It Solve?

Sometimes, the only difference between subclasses is the strategy that's used to perform some common operation. For example, a frame window may lay out its components in various ways, or a protocol handler may manage sockets in various ways. You can solve this problem with derivation—several frame derivatives would each lay out subcomponents in different ways, for example. This derivation-based solution creates a proliferation of classes, however. In Strategy, you define an interface that encapsulates the strategy for performing some operation (such as layout). Rather than deriving classes, you pass the Context class the strategy it uses to perform that operation.

Pros (✓) and Cons (✗)

- ✓ Strategy is a good alternative to subclassing. Rather than deriving a class and overriding a method called from the superclass, you implement a simple interface.

- ✓ The Strategy object concentrates algorithm-specific data that's not needed by the *Context* class in a class of its own.
- ✓ It's easy to add new strategies to a system, with no need to recompile existing classes.
- ✗ Communication overhead is small. Some of the arguments passed to the Strategy objects may not be used.

Often Confused With

Command: Command objects are very generic. The invoker of the command doesn't have a clue what the Command object does. A Strategy object performs a specific action.

See Also

Command

Implementation Notes and Example

```
interface SocketPool
{ Socket allocate( String host, int port )
  void release ( Socket s )
}
class SimplePool implements SocketPool
{ public Socket allocate(String host,int port)
  { return new Socket(host, port);
  }
  public void release(Socket s)
  { s.close();
  }
};
class KeepalivePool implements SocketPool
{ private Map connections = new HashMap();
  public Socket allocate(String host,int port)
  { Socket connection =
    (Socket)connections.get(host+":"+port);
    if(connection == null)
      connection = new Socket(host,port);
    return connection;
  }
  public void release(Socket s)
  { String host =
    s.getInetAddress().getHostName();
    connections.put( host+":"+s.getPort(),s );
  }
  //...
}
class ProtocolHandler
{ SocketPool pool = new SimplePool();
  public void process( String host, int port )
  { Socket in = pool.allocate(host,port);
    //...
    pool.release(in);
  }
  public void setPoolingStrategy( SocketPool p)
  { pool = p;
  }
}
```

The previous code implements a skeleton protocol handler. Some of the hosts that the handler talks to require that sockets used for communication are closed after every message is processed. Other hosts require that the same socket be used repeatedly. Other hosts may have other requirements. Because these requirements are hard to predict, the handler is passed a socket-pooling strategy.

The default strategy (SimplePool) simply opens a socket when asked and closes the socket when the ProtocolHandler releases it.

The KeepalivePool implements a different management strategy. If a socket has never been requested, this second strategy object creates it. When this new socket is released, instead of closing it, the strategy object stores it in a Map keyed by combined host name and port number. The next time a socket is requested with the same port name and host, the previously created socket is used. A more realistic example of this second strategy would probably implement notions such as *aging*, where a socket would be closed if it hadn't been used within a certain time frame.

In the interest of clarity, I've left out the exception handling.

Usage

```
JFrame frame = new JFrame();
frame.getContentPane().setLayout( new FlowLayout() );
frame.add( new JLabel("Hello World");
```

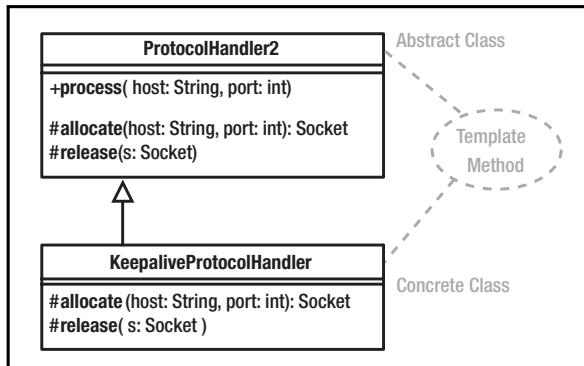
The `LayoutManager` (`FlowLayout`) defines a strategy for laying out the components in a container (`JFrame`, the `Context`).

```
String[] array = new String[]{ ... };
Arrays.sort
( array,
  new Comparator
  { int Compare( Object o1, Object o2 )
    { return ((String)o1).compareTo((String)o2);
    }
  }
);
```

The `Arrays.sort(...)` method is passed an array to sort and a `Comparator` that defines a strategy for comparing two array elements. This use of Strategy makes `sort(...)` completely generic—it can sort arrays of anything.

Template Method

Define an algorithm at the superclass level. Within the algorithm, call an abstract method to perform operations that can't be generalized in the superclass. This way you can change the behavior of an algorithm without changing its structure.



Abstract Class: Defines an algorithm that uses “primitive” operations that are supplied by a subclass.

Concrete Class: Implements the “primitive” operations.

What Problem Does It Solve?

Template Method is typically used in derivation-based application frameworks. The framework provides a set of superclasses that do 90 percent of the work, deferring application-specific operations to abstract methods. That is, superclass methods call abstract template methods. You use the framework by deriving classes that implement this application-specific behavior by providing template-method overrides.

Pros (✓) and Cons (✖)

✖ Template method has little to recommend it in most situations. Strategy, for example, typically provides a better alternative. Well-done class libraries work “out of the box.” You should be able to instantiate a framework class, and it should do something useful. Generally, the 90/10 rule applies (10 percent of the functionality is used 90 percent of the time, so the 10 percent should define the default behavior of the class). In template method, however, the framework often defines *no* default behavior, but rather you are required to provide subclasses for the superclass to do anything useful. Given the 90/10 rule, this means you have to do unnecessary work 90 percent of the time.

Template method does not prohibit the class designer from providing useful default functionality at the superclass level, expecting that the programmer will modify the behavior of the superclass through derived-class overrides if necessary. In an OO system, though, using derivation to modify superclass behavior is just run-of-the-mill programming that’s hardly worth glorifying as an official pattern.

- ✓ One reasonable application of Template Method is to provide empty “hooks” at the superclass level solely so that a programmer can insert functionality into the superclass via derivation.

Often Confused With

Factory Method: Factory Method is nothing but a Template Method that creates objects. To be a template method, you must intend for a subclass to override the template method in order to change the behavior of a superclass method that calls the template method.

See Also

Factory Method

Implementation Notes and Example

```

class ProtocolHandler2
{ protected Socket allocate(String host,int port)
  { return new Socket(host, port);
  }
  protected void release(Socket s)
  { s.close();
  }

  public void process( String host, int port )
  { Socket in =
    socketPool.allocate(host,port);
    //...
    socketPool.release(in);
  }
}

class KeepaliveProtocolHandler extends
ProtocolHandler2
{
  private Map connections = new HashMap();

  public Socket allocate(String host,int port)
  { Socket connection =
    (Socket)connections.get(host+":"+port);

    if(connection == null)
      connection = new Socket(host,port);
    return connection;
  }
  public void release(Socket s)
  { String host=
    s.getInetAddress().getHostName();
    connections.put( host+":"+s.getPort(),s);
  }
}

```

This example comes from the reference page for Strategy, rewritten to use Template Method. Rather than provide a Strategy object, you derive a class that modifies the superclass behavior. Put differently, you modify the behavior of the protocol-processing algorithm with respect to socket management by overriding a method that implements that algorithm.

Note that the class in the Abstract Class role (ProtocolHandler) is not actually abstract. In this reification, the superclass provides reasonable default behavior that a subclass can modify. Also note that the template methods are protected, almost always the case for template methods because they are not intended for outside access. (Ignore the fact that Java grants package access to protected fields—that is really a design flaw in the language.)

Usage

```

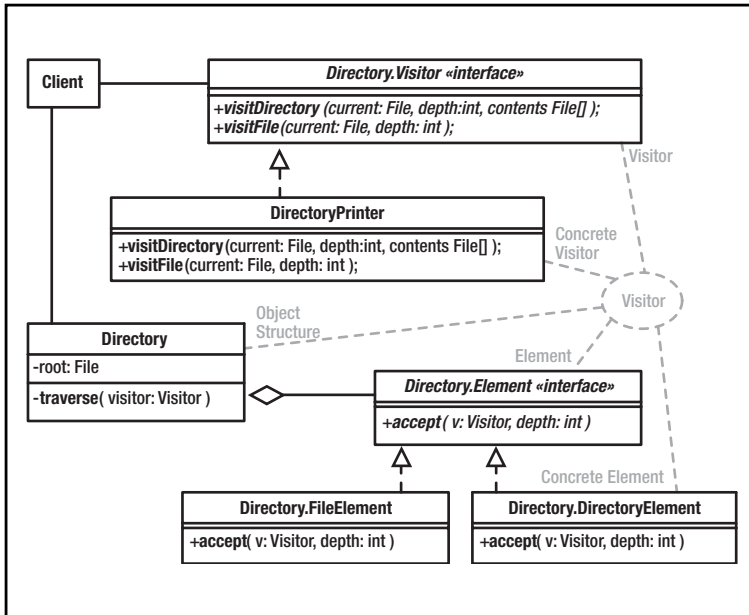
class MyPanel extends JPanel
{ public void paint( Graphics g )
  { g.drawString("Hello World", 10, 10);
  }
}

```

Define painting behavior by overriding the `paint(...)` method. You could easily do the same thing by passing a Panel a “paint” strategy.

Visitor

Add operations to a “host” object by providing a way for a visitor—an object that encapsulates an algorithm—to access the interior state of the host object. Typically, this pattern is used to interact with elements of an aggregate structure. The visitor moves from object to object within the aggregate.



Visitor: Defines an interface that allows access to a Concrete Element of an Object Structure. Various methods can access elements of different types.

Concrete Visitor: Implements an operation to be performed on the Elements. This object can store an algorithm’s local state.

Element: Defines an “accept” operation that permits a Visitor to access it.

Concrete Element: Implements the “accept” operation.

Object Structure: A composite object that can enumerate its elements.

What Problem Does It Solve?

Visitor’s primary use is to effectively add methods to a class without needing to derive classes. Visitors can also collect information or perform operations on all the elements of some aggregation. For example, a Visitor could test all the elements of an aggregation for consistency.

Pros (✓) and Cons (✗)

- ✓ It’s easy to add operations that haven’t thought of yet.
- ✓ Allows the class to be smaller since rarely used operations can be defined externally.
- ✓ Visitors can accumulate state as the visit elements. A “mobile agent” can visit remote objects (database servers, for example) and accumulate a composite result from a distributed database.
- ✗ The internal structure of the composite object is sometimes opened to the visitor, violating encapsulation. For example, an evil visitor could be passed to elements of a tree and change their “key” values, thereby turning the tree to garbage. The visitors are tightly coupled to the object they are visiting.

Often Confused With

Iterator: A Visitor is a lot like an internal (passive) iterator. The main difference is that the visitor object is passed from node to node. In Iterator, the nodes are passed, one at a time, to the iterator. The Visitor structure is different than the Iterator structure in that it gives the element control over whether to accept the visitor.

Strategy: A Visitor is, in a way, a “visiting strategy.” The focus of visitor is to visit every node of a data structure and do something. Strategy is much more general and has no connection to a data structure.

See Also

Strategy, Iterator

Implementation Notes and Example

```

class Directory
{
    public interface Visitor
    { void visitFile(File current, int depth);
      void visitDirectory( File current,
                          int depth, File[] contents );
    }
    public interface Element
    { void accept( Visitor v, int depth );
    }
    public class DirectoryElement
        implements Element
    { private File f;
      public DirectoryElement(File f){this.f=f;}
      public void accept( Visitor v, int depth )
      { v.visitDirectory(f,depth,f.listFiles());
      }
    }
    public class FileElement implements Element
    { private File f;
      public FileElement(File f){this.f = f;}
      public void accept( Visitor v, int depth )
      { v.visitFile( f, depth );
      }
    }
    //=====
    private File root;
    public Directory(String root)
    { this.root = new File(root);
    }

    public void traverse( Visitor visitor )
    { topDown( root, visitor, 0 );
    }

    private void topDown( File root,
                        Visitor visitor, int depth )
    { Element e =
      root.isFile()
      ? (Element)(new FileElement(root))
      : (Element)(new DirectoryElement(root))
      ;

      e.accept( visitor, depth );

      if( !root.isFile() )
      {
          File[] children = root.listFiles();
          for(int i = 0; i < children.length; ++i)
              topDown(children[i],visitor,depth+1);
      }
    }
}

```

Print a directory tree like this:

```

class DirectoryPrinter
    implements Directory.Visitor
{ public void visitFile(File f, int depth)
  {}
  public void visitDirectory( File f,
                              int depth, File[] children)
  { while( --depth >= 0 )
    System.out.print("..");
    System.out.println( f.getName() );
  }
}
Directory d = new Directory("c:/");
d.traverse( new Printer() );

```

The implementation at the left is a bit more complex than it needs to be so that I could demonstrate the general structure of traversing a heterogeneous composite object.

The key feature of Visitor is that it provides a way to add methods to an existing class without having to recompile that class. To my mind, that means that the “composite” could legitimately contain only one element. Consider this class:

```

class Money
{ long value; // value, scaled by 100
  Money increment( Money addend )
  { value += addend.value;
    return value;
  }
  //...
  public interface Modifier // visitor
  { long modify(long current);
  }
  operate( Modifier v )
  { value = v.modify(value);
  }
}

```

It’s impractical to define every possible operation on money, but you can effectively add an operation by implementing a Visitor (Modifier). Compute the future value of money like this:

```

class FutureValue implements Money.Modifier
{ FutureValue(float interest,int period)
  { /*...*/
  public long modify( long currentValue )
  { // return future value of currentValue
  }
}
Money presentValue = new Money(100000.00);
money.operate( new FutureValue(.05,24) );

```

