Decorator Utilities v.0.6

Project Documentation

Table of Contents

1.1 Introduction

Decorator Utilities

A set of utilities for the use of decorators in a Java web application. The goal of this project is to offer an easy way to "decorate" data for presentation purposes. This is partially done by using concepts introduced by the Decorator Design Pattern.

1.2 Dependencies

Dependencies

The following is a list of dependencies for this project.

These dependencies are required to compile and **run** the application:

Artifact ID	Type	Version	URL
commons-beanutils	jar	1.7.0	http://jakarta.apache.org/commons/beanutils/
commons-logging	jar	1.0.4	http://jakarta.apache.org/commons/logging/

These dependencies are required to compile the application:

Artifact ID	Type	Version	URL
cactus	jar	13-1.7	http://jakarta.apache.org/cactus/
cactus-maven	plugin	1.7	http://jakarta.apache.org/cactus/integration/maven/index.ht
httpunit	jar	1.6	http://httpunit.sourceforge.net/
junit	jar	3.8.1	http://junit.sf.net/
maven-findbugs-plugin	plugin	0.8.4	http://maven-plugins.sourceforge.net/maven-findbugs-plugi
maven-taglib-plugin	plugin	1.2.2	http://maven-taglib.sourceforge.net
servletapi	jar	2.3	http://jakarta.apache.org/tomcat/
maven-statcvs-plugin	plugin	2.5	http://statcvs-xml.berlios.de/maven-plugin/

1.3 Mailing Lists

Mailing Lists

These are the mailing lists that have been established for this project. For each list, there is a subscribe, unsubscribe, and an archive link.

List Name	Subscribe	Unsubscribe	Archive
Decorator Utilities User Mailing list	Subscribe	Unsubscribe	Archive
Decorator Utilities Developer Mailing list	Subscribe	Unsubscribe	Archive
Decorator Utilities Cvs Mailing list	Subscribe	Unsubscribe	Archive

1.4 FAQ

Frequently Asked Questions

General

Installation

General

General

Hello?

Stop playing around!

Installation

Installation

How do I install DecorUtils?

Add the Jar file to your app's lib directory and the tlds to your tld directory. Then all you have to do is use it. Check the Cookbook to get some hints on how to do that.

Don't forget to include the dependencies also!

2.1 Decorator

Decorator Utils: Decorator Tag Library

This Tag Library is usefull for decorating objects, transforming the data containers in presentable text data. This is done by aplying a decorator to the object we wish to decorate. This is version 0.6 .

- decorate This tag is usefull for decorating objects, transforming the data containers in presentable text data .
- decorateAndStore This tag is usefull for decorating objects, transforming the data containers in presentable text data .
- iterateAndDecorate This tag is usefull for decorating objects, transforming the data containers in presentable text data .

*Required attributes are marked with a * .*

decorate

This tag is usefull for decorating objects, transforming the data containers in presentable text data. This is done by aplying a decorator to the object we wish to decorate.

Can contain: JSP

Attributes

Name	Description	Type
attributes	A set of extra attributes that you may wish to add to your decorator through setter methods. The set of attributes should be seperated by commas (','). The attributes may be static or dynamic. For static attributes you must indicate the name of the property and it's value in the following manner: property=value. Dynamic attributes only require the name of the bean to retrieve, which will also be the name of the property to set.	`String`
*decorator	The fully qualified class name of a class that should be used to "decorate" the underlying object being displayed.	`String`
iterate	When set to true, the decorator expects a Collection of values to decorate and iterates through the Collection decorating each object seperatly. Default value is false.	`boolean`
*name	The name of the bean to be retrieved.	`String`

Name	Description	Type
property	The name of the property to be retrieved.	`String`
scope	The scope within which to search for the specified bean. Default is page scope.	`String`
setPageContext	If set to true, the decorator will be populated with the PageContext instance. Setter methods for pageContext must be exist in the decorator class (setPageContext). Default value is false.	`boolean`

decorateAndStore

This tag is usefull for decorating objects, transforming the data containers in presentable text data. This is done by aplying a decorator to the object we wish to decorate.

Can contain: JSP

Attributes

Name	Description	Type
attributes	A set of extra attributes that you may wish to add to your decorator through setter methods. The set of attributes should be seperated by commas (','). The attributes may be static or dynamic. For static attributes you must indicate the name of the property and it's value in the following manner: property=value. Dynamic attributes only require the name of the bean to retrieve, which will also be the name of the property to set.	`String`
*decorator	The fully qualified class name of a class that should be used to "decorate" the underlying object being displayed.	`String`
*id	Stores the result of the decoration in the specified scope instead of printing it. If the scope is not specified, the default scope is the page scope.	`String`
iterate	When set to true, the decorator expects a Collection of values to decorate and iterates through the Collection decorating each object seperatly. Default value is false.	`boolean`
*name	The name of the bean to be retrieved.	`String`
property	The name of the property to be retrieved.	`String`
scope	The scope within which to search for the specified bean. Default is page scope.	`String`
setPageContext	If set to true, the decorator will be populated with the PageContext instance. Setter methods for pageContext must be exist in the decorator class (setPageContext). Default value is false.	`boolean`

Name	Description	Type
toScope	The scope in which the result will be stored.	`String`

iterateAndDecorate

This tag is usefull for decorating objects, transforming the data containers in presentable text data. This is done by aplying a decorator to the object we wish to decorate.

Can contain: JSP

Attributes

Name	Description	Type
attributes	A set of extra attributes that you may wish to add to your decorator through setter methods. The set of attributes should be seperated by commas (','). The attributes may be static or dynamic. For static attributes you must indicate the name of the property and it's value in the following manner: property=value. Dynamic attributes only require the name of the bean to retrieve, which will also be the name of the property to set.	`String`
*decorator	The fully qualified class name of a class that should be used to "decorate" the underlying object being displayed.	`String`
*id	Stores the result of the decoration in the specified scope instead of printing it. If the scope is not specified, the default scope is the page scope.	`String`
iterate	When set to true, the decorator expects a Collection of values to decorate and iterates through the Collection decorating each object seperatly. Default value is false.	`boolean`
*name	The name of the bean to be retrieved.	`String`
property	The name of the property to be retrieved.	`String`
scope	The scope within which to search for the specified bean. Default is page scope.	`String`
setPageContext	If set to true, the decorator will be populated with the PageContext instance. Setter methods for pageContext must be exist in the decorator class (setPageContext). Default value is false.	`boolean`
toScope	The scope in which the result will be stored.	`String`

2.2 XML-Decorator

Decorator Utils: XML Tag Library

This Tag Library is meant to transform simple information stored on a XML document. There is no decorator involved for this is only meant to provide simple iteration through a document's child nodes. You may select the attibutes you wish to manipulate and use them for whatever purpose you want within the body of this tag. This is version 0.6 .

- xattribute Outputs the value of an attribute from the parent "X-Tag" .
- xdecorate This tag is meant to transform simple information stored on a XML document .
- xnested Each XNestedTag nested tag refers to the corresponding nested nodes .
- xrule This tag enables you to specify rules for the evaluation or not of it's body .

*Required attributes are marked with a * .*

xdecorate

This tag is meant to transform simple information stored on a XML document. There is no decorator involved for this is only meant to provide simple iteration through a document's child nodes. You may select the attibutes you wish to manipulate and use them for whatever purpose you want within the body of this tag.

Can contain: JSP

Attributes

Name	Description	Type
attributes	The attributes you wish to manipulate. They will be stored in the scope with the same name.	`String`
*document	XML document you wish to decorate.	`String`
xpath	Xpath for the nodes to be selected	`String`

xnested

Each XNestedTag nested tag refers to the corresponding nested nodes.

Can contain: JSP

Attributes

Name	Description	Type
attributes	The attributes you wish to manipulate. They will be stored in the scope with the same name.	`String`
id	This id can be used to avoid a clash between the attributes names in the scope. The attributes will be set as [id]_[attribute].	`String`

xrule

This tag enables you to specify rules for the evaluation or not of it's body.

Can contain: JSP

Attributes

Name	Description	Type
*rules	These rules should be separated by semi-colons (';') which will equivelent to an OR operation. These rules should be written in the form: [attribute]=[value] or [attribute]!=[value].	`String`

xattribute

Outputs the value of an attribute from the parent "X-Tag".

Can contain: JSP

Attributes

Name	Description	Type
*attribute	The name of the attribute to fetch.	`String`

3.1 Introduction

Chapter: Introduction

1. Java's Inheritance
2. The Decorator Design Pattern
3. The Display Logic Dilemma
4. Decorators and Wrappers
5. References

The Decorator Design Pattern describes an alternative to Java's subclassing.

Java's Inheritance

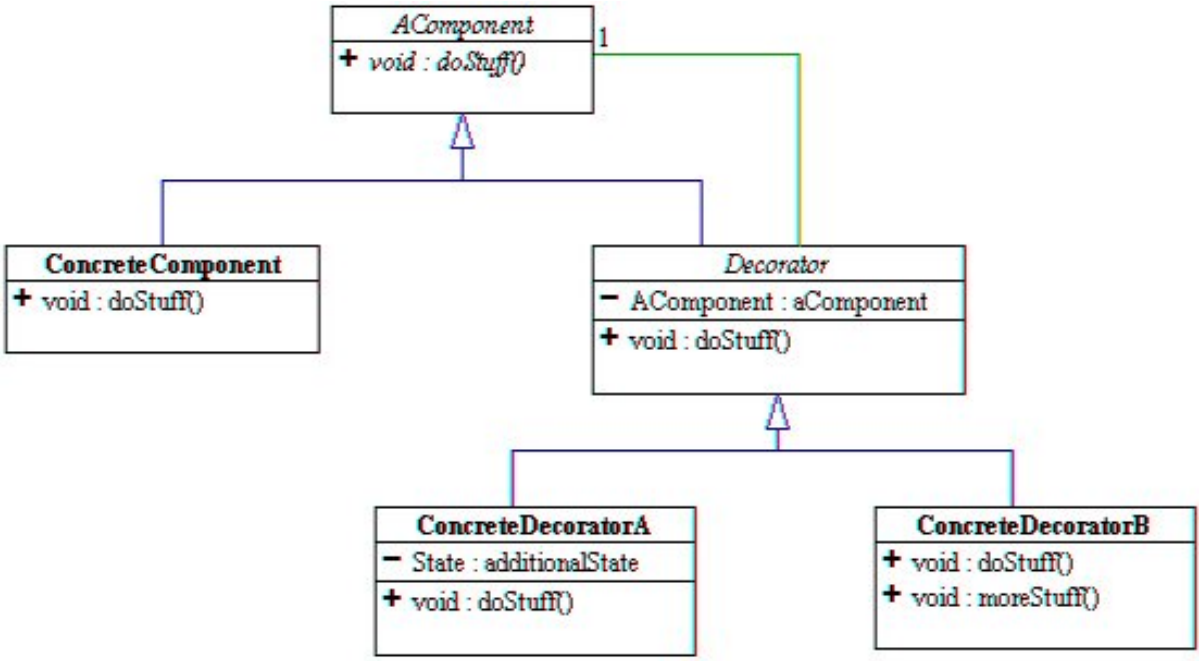
One of the main concepts associated with Object-Oriented Programming is inheritance. By extending a class, you inherit it's state and behaviors and are given the possibility to add new ones. This way, you are able to add functionality to an entire class of objects.

Extending classes is great, but what about extending objects?

Adding functionality to a single object and leaving others like it unmodified would be very useful in numerous situations. The Decorator Tag Library uses the Decorator Design Pattern to achieve this.

The Decorator Design Pattern

A Decorator, also known as a Wrapper, is an object that has an interface identical to an object that it contains. Any calls that the decorator gets, it relays to the object that it contains, and adds its own functionality along the way, either before or after the call. This gives you a lot of flexibility, since you can change what the decorator does at runtime, as opposed to having the change be static and determined at compile time by subclassing. Since a Decorator complies with the interface that the object that it contains, the Decorator is indistinguishable from the object that it contains. That is, a Decorator is a concrete instance of the abstract class, and thus is indistinguishable from any other concrete instance, including other decorators. This can be used to great advantage, as you can recursively nest decorators without any other objects being able to tell the difference, allowing a near infinite amount of customization[1].



The Decorator Design Pattern architecture as seen here [1].

Decorators add the ability to dynamically alter the behavior of an object because a decorator can be added or removed from an object without the client realizing that anything changed. It is a good idea to use a Decorator in a situation where you want to change the behaviour of an object repeatedly (by adding and subtracting functionality) during runtime[1].

The dynamic behavior modification capability also means that decorators are useful for adapting objects to new situations without re-writing the original object's code[1].

The Display Logic Dilemma

Today, a simple task as displaying information has become more and more complex. Rendering dynamic information on the Web usually involves retrieving it from data-sources, which may vary from databases to XML documents. Advanced technology enables you to establish a direct mapping between the data source and easy-to-use containers like Java beans. Displaying these might turn out to be a bit of drag. This raw data is not what you wish to display and subclassing is out of the question.

An easy aproach to this dilemma is the Decorator Design Pattern. To make it even easier, you can use the Decorator Tag Library, which applies a decorator to an object for you.

You can have all of your display logic contained by a set of decorators and easily apply it to any object.

Decorators and Wrappers

Usualy, the *decorator* and *wrapper* concepts are used for the same thing. DecorUtils chose to differentiate them for the purpose of this tool:

- A **wrapper** is considered to be a class that *wraps* another to give it some extra funcionality as described in the Decorator Pattern.

- A **decorator** is a simple class whose sole purpose is to *decorate* a certain object and not it's characteristics. In a way it can be seen as a parser or a transformation.

A decorator is a class that only requires a `decorate()` method that receives an `Object` and returns another one:

```
public class MyDecorator {

        public Object decorate(Object object) {
            // decorate
            return decoratedObject;
        }

}
```

The `Decorator` interface that is provided is nothing more than a facility that is meant to help you implement your decorator. You don't need to implement it.

A wrapper is a class that should wrap the behaviours/states that you wish to override:

```
public class MyWrapper extends Wrapper {

        public String getName() {
            MyObject myObject = (MyObject) getObject();
            return myObject.getParent().getName()
                    + " -> "
                    + myObject.getName();
        }

        public String getDate() {
                // format date
                return dateString;
        }

}
```

References

[1] The Decorator Design Pattern, Antonio Garc□ a and Stephen Wong - http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/DecoratorPattern.htm

3.2 Decorator

Chapter: Decorator Tag Library

1. Simple Decorator
2. Decorator that Uses the Page Scope
3. Decorator With Attributes
4. Store Instead of Printing
5. Decorate a Collection of Objects
6. Decorator Tag Library vs. Display Tag Library

Recipe: Simple Decorator

The simplest way to use the Decorator Tag Library is to simply apply a decorator to an object stored in some scope.

The example bellow shows a `Date` instance being decorated by a `Date` decorator that will transform it into a presentable `String`.

```
<% request.setAttribute("date", new java.util.Date()); %>

<deco:decorate
        name="date"
        decorator="org.devyant.decorutils.decorators.DateDecorator" />
```

This code outputs: `Wednesday, March 23, 2005`

Recipe: Decorator that Uses the Page Scope

In the last example, we used a `Date` wrapper which tranformed the date object and returned a String representing the date. This date was printed in english because the default locale used is `Locale.ENGLISH`.

If you need to access a scope on your decorator, you can use set the attribute *setPageContext* to `true` and a method named `setPageContext` in your decorator will be called.

In this case, the `DateDecorator` has a method setter method for the PageContext instance:

```
    public void setPageContext(PageContext pageContext) {
        this.locale = pageContext.getRequest().getLocale();
```

```
    }
```

This means that we can use that to present the current date in the viewers language:

```
<% request.setAttribute("date", new java.util.Date()); %>

<deco:decorate
        name="date"
        setPageContext="true"
        decorator="org.devyant.decorutils.decorators.DateDecorator" />
```

This code results in: `Wednesday, March 23, 2005`

Recipe: Decorator With Attributes

Sometimes a method that takes only one argument is not quite enough. You might want to make your decorator customizable or it just requires some extra information for its decoration. This is what the *attributes* attribute is for.

The example that we've been using has been transforming the `Date` object always with the same format. We may want to show the date in a more compact format in some places and in a more complete format in others. We could create another decorator, but that doesn't sound that good. We can acomplish it by having an attribute named *format* and specify it at run-time.

```
<% request.setAttribute("date", new java.util.Date()); %>

<deco:decorate
        name="date"
        attributes="format=short"
        setPageContext="true"
        decorator="org.devyant.decorutils.decorators.DateDecorator" />
```

Which results in: `3/23/05`

Inside the decorator:

```
    public void setFormat(String format) {
        // stuff
    }
```

What we did was insert static data into the decorator. This data may also be dynamic. The operator **=** implies static information. When there's no operator, the value for the attibute will be fetched from the scope.

```
<%
    request.setAttribute("date", new java.util.Date());
    request.setAttribute("format", "short");
%>

<deco:decorate
        name="date"
        attributes="format"
        setPageContext="true"
        decorator="org.devyant.decorutils.decorators.DateDecorator" />
```

You can specify more than one attribute by seperating them with comas (','):

```
<deco:decorate
        name="date"
        attributes="xyz=things, scopeStuff, zyx=some more stuff"

        setPageContext="true"
        decorator="org.devyant.decorutils.decorators.DateDecorator" />
```

Recipe: Store Instead of Printing

To take complete advantage of the Decorator Design Pattern, we can't just stick to printing `String` 's, we want to store the decorated objects, not necesseraly `String` 's, for more complex tasks.

```
<% request.setAttribute("date", new java.util.Date()); %>

<deco:decorateAndStore
        id="decoratedDate"
        name="date"
        attributes="format=short"
        setPageContext="true"
        decorator="org.devyant.decorutils.decorators.DateDecorator" />
```

The `decorateAndStore` tag works exactly the same way has the `decorate` tag, but instead of printing the result, it stores it in the scope.

Recipe: Decorate a Collection of Objects

All the tags have an attribute named *iterate* which defaults to `false`. If set to `true`, the tag will act as if the object is a `Collection` and iterate through it and decorate each object.

```
<%
    java.util.Collection c = new java.util.ArrayList();
    c.add(new java.util.Date());
    c.add(new java.util.Date());
```

```
    c.add(new java.util.Date());
    request.setAttribute("collection", c);
%>

<deco:decorate
        name="collection"
        attributes="format=short"
        setPageContext="true"
        iterate="true"
        decorator="org.devyant.decorutils.decorators.DateDecorator" />
```

Which outputs: `3/23/053/23/053/23/05`

In this case, it actually doesn't look very nice. This brings us to the `iterateAndDecorate` tag. This tag expects a `Collection` and iterates through it. Each iteration results in a decorated object being stored in the scope for you to use in any way on the tag's body.

```
<%
    java.util.Collection c = new java.util.ArrayList();
    c.add(new java.util.Date());
    c.add(new java.util.Date());
    c.add(new java.util.Date());
    request.setAttribute("collection", c);
%>

<deco:iterateAndDecorate
    id="date"
    name="collection"
    attributes="format=short"
    setPageContext="true"
    decorator="org.devyant.decorutils.decorators.DateDecorator">

    <p>
        <strong><%= date %></strong>
    </p>

</deco:iterateAndDecorate>
```

Which outputs:

3/23/05

3/23/05

3/23/05

Recipe: Decorator Tag Library vs. Display Tag Library

So you've made a couple of TableDecorator's and wondered: "Wouldn't it be cool if I could you these on the Decorator taglib?". Well, you can. I'm not going to tell you about the TableDecorator, you can check it out at the Display Tag Library site, more precisely: the Decorators Tutorial . DisplayTag also has a

ColumnDecorator which works exactly like the decorators I have presented until now, so you can use those with the Decorator Tag Library as well.

So, has you can see, there's no "vs.", just plain beautiful compatibility :).

DisplayTag's TableDecorators are wrappers just like Decorutils's Wrapper, so the use of TableDecorator's has just a special detail to attend to: the *property* attribute must be set with the property you want to retrieve from the decorator, so the *name* attribute will be the sole responsible for the Java bean (just like with a Wrapper).

```
<%
    SomeObject o = new SomeObject("I'm a fake");
    o.setDate(new java.util.Date());
    request.setAttribute("someObject", o);
%>

<deco:decorate
        name="someObject"
        property="date"
        decorator="org.devyant.decorutils.SomeTableDecorator" />
```

Display *: Tag Library