Sun™ ONE Application Framework Component Author's Guide

Sun™ ONE Studio 5 update 1

Sun Microsystems, Inc.
www.sun.com

Part No. 817-4362-10
October 2003, Revision A

Submit comments about this document at: http://www.sun.com/hwdocs/feedback

Contents

Preface

This Sun™ ONE Application Framework *Component Author's Guide* describes the Sun ONE Application Framework component architecture and the process whereby component authors can design, create, and distribute new components. This book is intended for prospective Sun ONE Application Framework component authors, and assumes that these component authors are already familiar with the Sun ONE Application Framework architecture.

How This Book Is Organized

Chapter 1, Overview and Component Architecture provides an overview of Component-Based Development (CBD), Sun ONE Application Framework Component Library, the Component Class, and the ComponentInfo Class.

Chapter 2, Developing Components provides a description of the fundamental steps involved in creating, distributing, and using a Sun ONE Application Framework component.

Chapter 3, Developing View Components provides a description of the fundamental steps involved in developing view components.

Chapter 4, Developing Model Components provides a description of the fundamental steps involved in developing model components.

Chapter 5, Developing Command Components provides a description of the fundamental steps involved in developing command components.

Chapter 6, ConfigurableBeans (Non-Visual Components) introduces how the IDE toolset makes use of the ConfigurableBean, the role it plays, and the relationship between Sun ONE Application Framework and the ConfigurableBean types.

Chapter 7, Developing and Distributing Non-Extensible Model, Command and ContainerView Components introduces the steps to develop and distribute non-extensible Model, Command, and ContainerView components.

Chapter 8, Design Actions describes developing extensible components which have component design actions, defines a component design action, and shows how to expose design action in `ComponentInfo`.

Chapter A, Component Library Structure offers an overview of the component library and the component library structure, and details the component manifest, with a description of automated unpacking of component tag libraries (TLD) files, and automated unpacking of "Additional Files".

Using UNIX Commands

This document might not contain information on basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices. See the following for this information:

- Software documentation that you received with your system
- Solaris™ operating environment documentation, which is at

  `http://docs.sun.com`

Related Documentation

Application	**Title**	**Part Number**
Sun ONE Application Framework 2.1	*Sun ONE Application Framework Overview, Sun™ ONE Studio 5 update 1*	817-4360-10
Sun ONE Application Framework 2.1	*Sun ONE Application Framework Tutorial, Sun ONE Studio 5 update 1*	817-4358-10
Sun ONE Application Framework 2.1	*Sun ONE Application Framework IDE Guide, Sun ONE Studio 5 update 1*	817-4104-10

Application	Title	Part Number
Sun ONE Application Framework 2.1	*Sun ONE Application Framework Developer's Guide, Sun ONE Studio 5 update 1*	817-4359-10
Sun ONE Application Framework 2.1	*Sun ONE Application Framework Component Reference Guide, Sun ONE Studio 5 update 1*	817-4661-10
Sun ONE Application Framework 2.1	*Sun ONE Application Framework Tag Library Reference, Sun ONE Studio 5 update 1*	817-4361-10

Accessing Sun Documentation

You can view, print, or purchase a broad selection of Sun documentation, including localized versions, at:

`http://www.sun.com/documentation`

Contacting Sun Technical Support

If you have technical questions about this product that are not answered in this document, go to:

`http://www.sun.com/service/contacting`

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can submit your comments by going to:

`http://www.sun.com/hwdocs/feedback`

Please include the title and part number of your document with your feedback:

Sun ONE Application Framework Component Author's Guide, part number 817-4362-10

CHAPTER 1

Overview and Component Architecture

Component-Based Development (CBD)

Component-Based Development (CBD) is a highly regarded engineering strategy whereby the production, distribution, and consumption of components contribute to more efficient and reliable application development. Mature CBD combines a robust component model with a component-aware IDE.

The producers of components (component authors) are responsible for developing components according to the specification of a particular component model (component architecture). A component model formalizes component structure and specifies a means of component distribution. A mature component model also allows each component to be self-describing, so that it can advertise its features to component consumers. Components are typically distributed as collections known as component libraries.

Components can come in a variety of flavors intended for use in different development scenarios. For example, components can be designed in a very generic, or horizontal fashion to cut across a range of specific development needs. These components tend to be the broadest components available, with their strength being flexibility and customizability. These types of components are usable by many different application developer populations, across projects and companies, and in the Web application space, and generally are not biased toward any particular look and feel. Alternatively, other components can be designed to satisfy a narrower, vertical set of development needs. These components are tailored to a particular usage scenario, allowing them to provide high-level features and high ease-of-use. These types of components are less broadly usable, but because their scope is more narrowly defined, they can keep parameterization to a minimum and use a particular look and feel.

The consumers of components are typically application developers. In CBD, application development consists primarily of aggregating or assembling a particular application from a collection of reusable components. The greater the coverage provided by the components, the smaller the amount of application-specific code.

A component-aware IDE is necessary to expose components to the component consumers. The IDE leverages the self-describing nature of components to dynamically present components for instantiation and configuration. The IDE is the final piece of the puzzle, but it is very significant. Without a component-aware IDE, the component model exists only on paper. Without a component-aware IDE, developers can only use a component as they would any other Java class, through its public API. A component-aware IDE, on the other hand, allows developers to browse through collections of components, visually assemble components into application entities, and configure components by declaratively filling in component-specific property sheets.

The rest of this document describes the Sun™ ONE Application Framework component model, and the manner in which component authors can leverage that component model to create powerful component libraries.

What is a Sun ONE Application Framework Component?

Since its inception, the Sun ONE Application Framework application has supported a component model for certain types of objects. However, the prior component model relied on developers to learn each component's API and write code to use that component in their applications. Although this level of functionality was sufficient and provided a significant productivity advantage over contemporary competitors, the Sun ONE Application Framework version 2.1 has significantly extended its component model to encompass all types of primary Sun ONE Application Framework application objects (Views, Models, and Commands). Furthermore, the Sun ONE Application Framework module for the Sun™ ONE Studio now provides a full featured, component-aware IDE that creates a visual development environment for Sun ONE Application Framework applications.

In Sun ONE Application Framework 2.1 terms, a component is one of the various types of supported component classes (Views, Models, and Commands) in conjunction with metadata information. This metadata is encapsulated in a Sun ONE Application Framework-specific class called a `ComponentInfo` class. At design-time, the Sun ONE Studio can inspect the ComponentInfo and present the component in an easy-to-use visual fashion.

The metadata stored in `ComponentInfo` classes is intended to enable automated use of the component in a development environment, such as the Sun ONE Studio. Developers can still manually create and use various types of components in their applications without defining a `ComponentInfo` class.

Sun ONE Application Framework Component Libraries

The Sun ONE Application Framework component model requires that components intended for discovery by the IDE toolset must be packaged into a specific component library format. A Sun ONE Application Framework component library consists of a standard JAR file containing the component classes, ComponentInfo metadata classes, and a single component library manifest file. The component library manifest file is described in detail later in this document.

Note – A component library JAR can contain any number of non-component related classes. It is just a standard JAR file with some component-model-specific additions.

Application developers make use of the Sun ONE Application Framework component libraries by placing them in their Web application's `WEB-INF/lib` directory. The IDE toolset automatically recognizes and mounts any component libraries placed in that directory. After the IDE toolset has discovered and inspected the libraries (it might take a minute or two due to background thread latency), the library components are available for use within the application. The components are then said to be registered with the IDE toolset.

Hint: The library manifest inspection and component registration process recurs every time a Sun ONE Application Framework application is remounted within the IDE toolset. This is natural and should be expected, because the component model is entirely dynamic. However, both component authors and application developers should be aware of this process and understand that the accidental or intentional removal of a component library from the application’s `WEB-INF/lib` directory results in the expected omission of those components the next time the application is mounted.

Sun ONE Studio troubleshooting hint: A common mistake of newcomers to the Sun ONE Studio is to improperly mount a Web application. The Sun ONE Studio's Web application module (upon which the Sun ONE Application Framework toolset module is built) only recognizes a mounted file system as a Web application if the mount point corresponds with the root of the Web application structure. If you do not mount the Web application at its root directory, the Sun ONE Studio treats it as a conventional file system, and fails to provide the Sun ONE Application Framework application view that you expect. Keep this in mind as you build and test your first components. The easiest way to avoid any confusion in this regard is to use the Mount Sun ONE Web Application action of the IDE instead of the Mount File System action.

The Sun ONE Application Framework Component Library

The Sun ONE Application Framework Component Library contains the core interfaces, run-time classes, and many basic components that you use to create a Sun ONE Application Framework application. The standard Sun ONE Application Framework Component Library is packaged as a single JAR file, and should appear in your application's `WEB-INF/lib` directory.

When creating a Sun ONE Application Framework application using the IDE toolset, the current version of the standard Sun ONE Application Framework Component Library is automatically added to the application's `WEB-INF/lib` directory. If you open an application created in a previous version of the IDE toolset, you might be prompted to upgrade the application, including the Sun ONE Application Framework run-time library.

The Component Class

A Sun ONE Application Framework component class is the class which defines a Sun ONE Application Framework run-time type, a View, a Command, or a Model.

The author of the component class is only concerned with design-time considerations to the extent that a JavaBean developer would do so. That is to say, as a component author, you must anticipate the properties which you would like to expose to design time configuration and define appropriate `get` and `set` methods. However, unlike the JavaBean model, the Sun ONE Application Framework component model does not eagerly expose all `get` and `set` methods as properties. That is because the Sun ONE Application Framework recognizes that there are many `get` and `set` methods in the Sun ONE Application Framework core from which the components derive which are not appropriate for design time configuration. Therefore, the Sun ONE Application Framework component model limits the exposed properties to those which are explicitly specified in the companion `ComponentInfo` class.

The ComponentInfo Class

The `ComponentInfo` class is the heart and soul of the Sun ONE Application Framework 2.1 component model. Logically speaking, a Sun ONE Application Framework component can be referred to as a tuple comprised of a component class and a `ComponentInfo` class. The `ComponentInfo` class provides the metadata that is introspected by the IDE toolset to provide the component's design-time presence. When you author a `ComponentInfo` class, you can focus exclusively on design-time considerations. The `ComponentInfo` class plays no run-time role in the Sun ONE Application Framework.

CHAPTER 2

Developing Components

Develop Your First Component

This chapter provides a description of the fundamental steps involved in creating, distributing, and using a Sun ONE Application Framework component.

Approach this as an exercise, and actually build and test drive the component. After completing this section, you should have a good understanding of the process. Do not worry about trying to understand every detail at this point. The rest of this document delves into details concerning the various types of components, the details of the metadata formats, and the extra optional features available to component authors.

This section assumes basic familiarity with the Sun ONE Application Framework application.

Decide the Type of the Component

An ultra-simple example is contrived to focus on technique. You will create a new DisplayField component called "MyTextField". The objective is to have this component expose a new property called "Foo" that will take a boolean value. Application developers will be able to visually select `MyTextField` and add it to their Sun ONE Application Framework pages. It is expected that the component will have all of the properties of the standard Sun ONE Application Framework `TextField` component, plus the new *Foo* property.

Create the Component Class

A new component class is not always needed in Sun ONE Application Framework. This subtlety is discussed later in this document. This example, however, does require a new component class, so you will begin with that.

1. **In any Java editor create the package** `mycomponents`**.**
2. **Create the** `mycomponents.MyTextField` **class.**
3. **Make** `MyTextField` extend `com.iplanet.jato.view.BasicDisplayField`**.**
4. **Implement the appropriate constructor for the component type.**

   All `DisplayField` components must implement a two-arg constructor that takes a View "parent" and a String "name". The IDE toolset assumes that all `DisplayField` components will implement this constructor.
5. **Add a** `get` **and** `set` **method for the new boolean property named "Foo".**

   After these steps, `mycomponents/MyTextField.java` should look as follows:

```
package mycomponents;

import com.iplanet.jato.view.*;

/**
 *
 * @author component author
 */
public class MyTextField extends BasicDisplayField {

     /** Creates a new instance of MyTextField */
     public MyTextField(View parent, String name) {
          super(parent, name);
     }

     public boolean getFoo() {
          return foo;
     }

     public void setFoo(boolean value) {
          foo = value;
     }

     boolean foo;
}
```

Although you are creating a new property on this component, how this property actually interacts with the component at run-time is not defined. That is up to you as the component author and is beyond the scope of this part of the document.

Create the ComponentInfo Class

The `ComponentInfo` class defines the design-time metadata that the IDE toolset requires to incorporate the component. In this example, you extend an existing `ComponentInfo` and, in true OO style, simply augment it. You could, of course, choose to implement the `ComponentInfo` interface from scratch, but that would be unproductive in this case.

1. **Create the class** `mycomponents.MyTextFieldComponentInfo`**.**
2. **Make** `MyTextFieldComponentInfo` **extend** `com.iplanet.jato.view.html2.TextFieldComponentInfo`**.**
3. **Implement the no-arg constructor.**
4. **Implement the** `getComponentDescriptor()` **method to provide the basic design-time description of the component.**
5. **Implement the** `getConfigPropertyDescriptors()` **method to identify which properties you want to expose in the IDE.**

   Utilize inheritance to add the new *Foo* property to those properties already defined in `TextFieldComponentInfo`.

   After these steps, `mycomponents/MyTextFieldComponentInfo.java` should look like the code that follows:

Note – In the following sample code, for demonstration purposes, String values have been embedded directly. If you anticipate the need to localize your display strings, utilize resource bundles.

```
package mycomponents;

import java.util.*;
import com.iplanet.jato.view.*;
import com.iplanet.jato.component.*;
import com.iplanet.jato.view.html2.*;


public class MyTextFieldComponentInfo extends TextFieldComponentInfo {

     public MyTextFieldComponentInfo()
     {
```

```
        super();
    }

    public ComponentDescriptor getComponentDescriptor() {

        // identify the component class
        ComponentDescriptor result=new ComponentDescriptor(
            "mycomponents.MyTextField");

        // The name will be used to determine a name for the component instance
        result.setName("MyTextField");

        // The display name will be used to show the component in a chooser
        result.setDisplayName("MyTextField Component");

        // The description will be the tool tip text for the component
        result.setShortDescription("A simple demonstration of a new component");

        return result;
    }

    public ConfigPropertyDescriptor[] getConfigPropertyDescriptors() {

        if (configPropertyDescriptors!=null)
            return configPropertyDescriptors;

        // Get any properties defined in the super class
        configPropertyDescriptors=super.getConfigPropertyDescriptors();
        List descriptors=new LinkedList(
            Arrays.asList(configPropertyDescriptors));

        ConfigPropertyDescriptor descriptor = null;

        // Add the "foo" property
        descriptor=new ConfigPropertyDescriptor("foo",Boolean.TYPE);
        descriptor.setDisplayName("Foo Property");
        descriptor.setHidden(false);
        descriptor.setExpert(false);
        descriptor.setDefaultValue(new Boolean(false));
        descriptors.add(descriptor);

        // Create/return the array
        configPropertyDescriptors = (ConfigPropertyDescriptor[])
            descriptors.toArray(
                new ConfigPropertyDescriptor[descriptors.size()]);
        return configPropertyDescriptors;
    }

    private ConfigPropertyDescriptor[] configPropertyDescriptors;
}
```

Create the Component Library Manifest

Sun ONE Application Framework components are packaged and distributed in ordinary JAR files. Any classes (component, ComponentInfo, and any other ancillary files) should be placed in the JAR in accordance with standard Java convention. Additionally, the Sun ONE Application Framework requires that a component library JAR contains a special Sun ONE Application Framework library manifest file. This is a simple XML document that describes the collection of components in the library. Library manifests might declare any number of components. In this case, just declare the one component that you have just authored.

The Sun ONE Application Framework library manifest must be named `complib.xml`. Within the JAR file, the Sun ONE Application Framework library manifest must be placed in the `/COMP-INF` directory.

1. **Create the file called `complib.xml`.**

2. **Add the minimum information to satisfy the Sun ONE Application Framework library manifest requirements.**

3. **Add a component declaration for the `MyTextField` component.**

   After these steps, the `COMP-INF/complib.xml` file should look like the code that follows:

Note – If you use a tool to create the XML file, be sure that it looks like this. Some XML tools automatically insert a root element when you create the file. Make sure the root element is <*component-library*> as indicated below. An improper XML file will cause the IDE toolset to fail to discover your component library.

```
<?xml version="1.0" encoding="UTF-8"?>
<component-library>
        <tool-info>
                <tool-version>2.1.0</tool-version>
        </tool-info>
        <library-name>mycomponents</library-name>
        <display-name>My First Component Library</display-name>
        <!-- Your icon here
        <icon>
                <small-icon>/com/iplanet/jato/resources/complib.gif</small-icon>
        </icon>
        -->
        <interface-version>1.0.0</interface-version>
        <implementation-version>20030221</implementation-version>

        <component>
                <component-class>mycomponents.MyTextField</component-class>
                <component-info-class>mycomponents.MyTextFieldComponentInfo</component-
        info-class>
        </component>

</component-library>
```

Create the Component Library JAR File

JAR up the component classes so they can be ready for distribution as a library.

The name of the JAR file is arbitrary.

1. **In this case, name the JAR file** `mycomponents.jar`**.**

   You can omit the Java source files from the JAR.

2. **You should include in the JAR any necessary ancillary resources, such as icon images or resource bundles. In this case there are none.**

   The `mycomponents.jar` internal structure should look like the code that follows:

```
mycomponents/MyTextField.class
mycomponents/MyTextFieldComponentInfo.class
COMP-INF/complib.xml
```

Test the Component

Your library is now ready for testing and distribution. You should test it in a sample project. This stage requires the use of the Sun ONE Studio with the Sun ONE Application Framework module installed and enabled. If you have never built a Sun ONE Application Framework application in the Sun ONE Studio, before continuing, you should first complete the *Sun ONE Application Framework Tutorial* that is included with the Sun ONE Application Framework document set.

Caution – You are free to test your component(s) in any existing Sun ONE Application Framework application, however, you should create a new Sun ONE Application Framework application to serve as the test application for all of the example components that you will build in the course of completing the exercises within this guide. The instructions that follow generally assume that the names for your test objects were generated according to Sun ONE Application Framework defaults (for example, Page1, and so on) and you will have an easier time following the instructions if your test application's object names match those in the instructions.

---

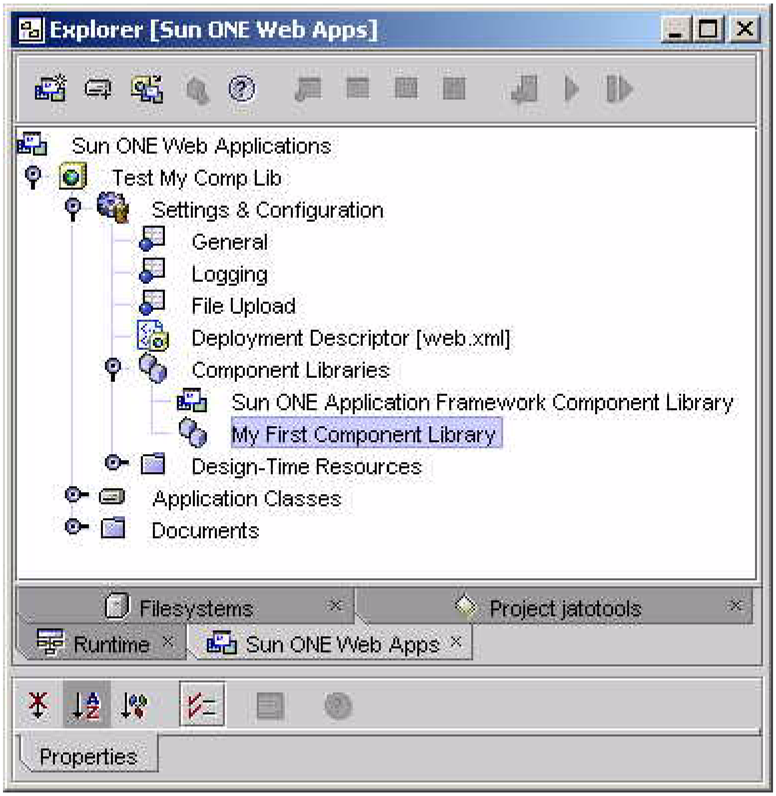
1. **Create a new Sun ONE Application Framework application in the Sun ONE Studio.**

   The name of the application is up to you.

2. **From the filesystem, copy the new** `mycomponents.jar` **file into the** `WEB-INF/lib` **directory within your test application.**

3. **Wait for the Sun ONE Studio background thread to discover that a new JAR has been deployed in the application's** `WEB-INF/lib` **directory.**

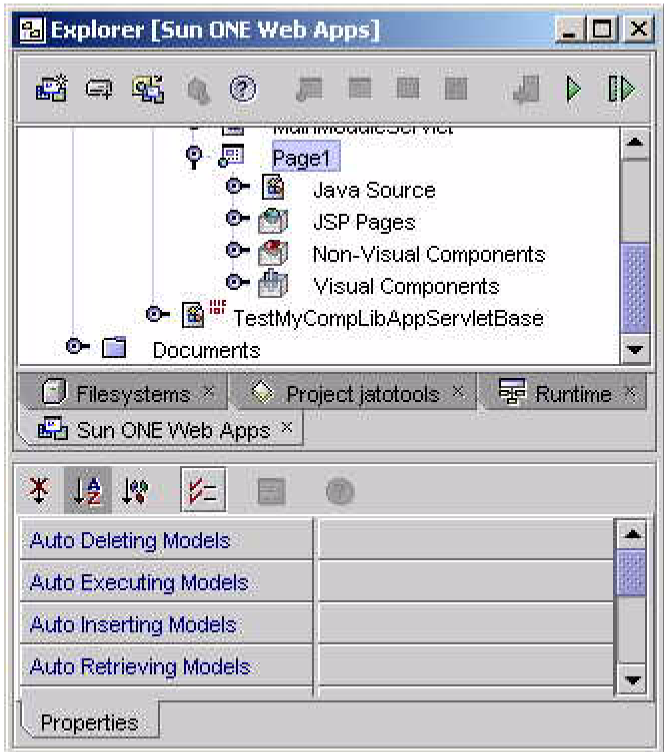
   This takes several seconds, depending upon the value Sun ONE Studio background thread Refresh Interval. The library is fully recognized and functional when a new library node appears under the Sun ONE Application Framework application's Settings and Configuration -> Component Libraries node as shown below.



4. **Create a new Page (ViewBean) object.**

   Take the wizard defaults and the IDE names it "Page1".

5. **Select and expand the newly created Page1 node.**



6. **Add an instance of "MyTextField Component" to Page1.**

   This can be accomplished in either of two equally valid user interface actions.

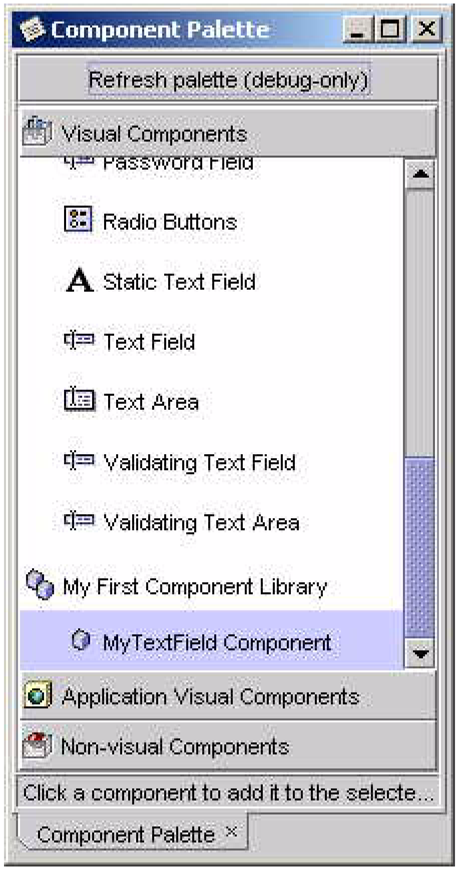
   - First you can utilize the Component Palette (below).
     - Expand the "Visual Components" section.
     - Click the "MyTextField Component" item.

     This adds an instance to whatever page node has focus at that moment.

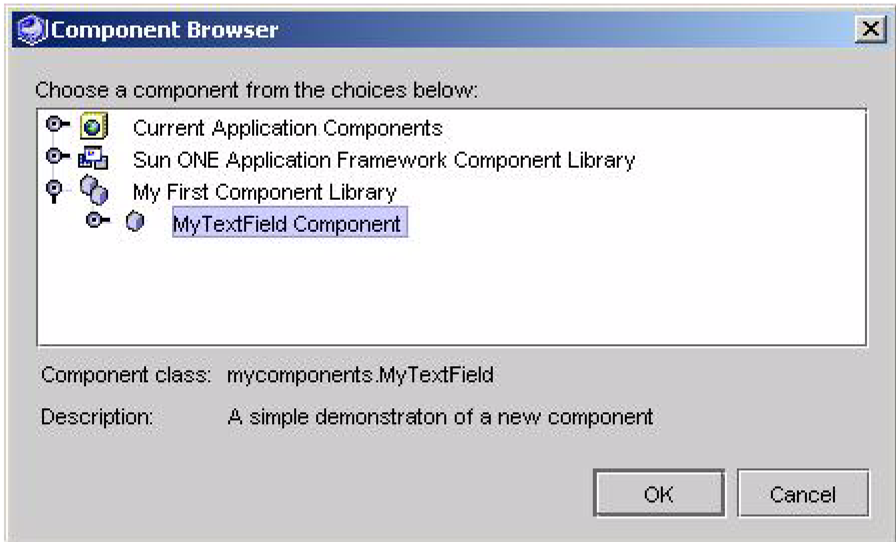
   - Alternatively, you can select Page1's Visual Components sub-node.

     Right-click, and select the *Add Visual Component* action from the pop-up menu.

     Note the generic icons for both the library "My First Component Library" and the component "MyTextField Component". This occurs because, in this example, you did not specify any specific icons. That is just one of the features that you learn about in the rest of this document.



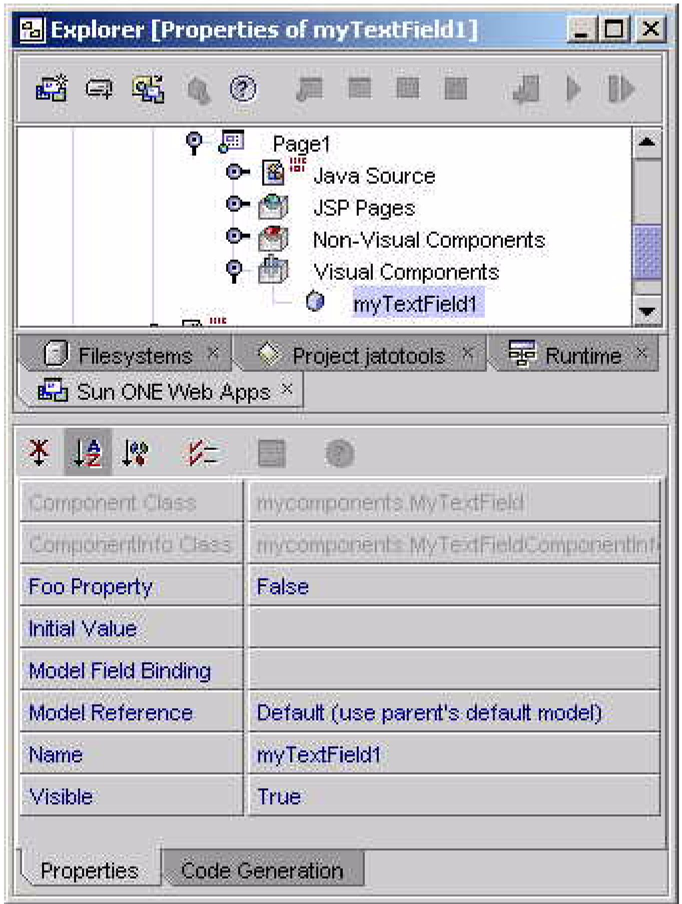
The Component Browser (below) is an alternative to the Component Palette (above). In the rest of this document, any instruction that involves adding a visual component can be fulfilled by using either the Component Palette or the Component Browser. They are functionally interchangeable, and users are free to use either, or both, at all times.



After selecting the `MyTextField` Component from either the Component Palette or the Component Browser, observe how a child View named "myTextField1" is added to the page.

7. **Select the child node `myTextField1`.**

   Observe how the IDE's property sheet has added *Foo* Property, the new custom property, in addition to the inherited `TextField` component properties.



Test the behavior of *Foo* Property to make sure it behaves the way you, as component author, expect.

You should be able to assign *Foo* Property the value `True` or `False`.

8. **In this example, set *Foo* Property to** `True`**.**

9. **Observe the code generation inside the** `Page1` **java file.**

   You should see a block of code inside the `createChildReserved` method that looks like the following code (the indenting in your code might differ from what you see below):

```
...
else if (name.equals(CHILD_MY_TEXT_FIELD1)) {
     mycomponents.MyTextField child =
          new mycomponents.MyTextField(this, CHILD_MY_TEXT_FIELD1);
     child.setFoo(true);
     return child;
}
...
```

Ship It!

Once you are done testing and refining your component, you can distribute the component library JAR file to your developer community. It is up to application developers to add the component JAR file to each application in which they want to utilize the components.

Sun ONE Application Framework Components in More Detail

Sun ONE Application Framework components are designed to enable application developers to more rapidly define Sun ONE Application Framework run-time types (Views, Commands, and Models). However, the manner in which Sun ONE Application Framework components are integrated into the application developer’s design-time experience varies in accordance with the range of Java's object oriented opportunities (for example, class sub-typing vs. object instantiation).

As an experienced Java programmer, a Sun ONE Application Framework component author should easily anticipate the manner in which Sun ONE Application Framework application developers will integrate a new component into their development processes. The component author will know that the application developer expects to subclass one type of component, and instantiate another type of component. Component authors understand that in some circumstances they can distribute a component as a fully enabled, fully configured black box, and in other cases they require the application developer to configure each usage of the component.

The Sun ONE Application Framework component model and the IDE toolset combine to empower component authors and application developers to exploit the full range of Java object orientation. This section details the specific terminology that the Sun ONE Application Framework component model uses to differentiate each component's role as an object oriented building block. The discussion of components is often filled with highly overloaded terms. To provide the grounds for a more precise discussion of Sun ONE Application Framework components, some terminology has been developed to avoid reliance on confusingly overloaded terms.

Distributable vs. Application-Specific (Non-Distributable) Components

Technically speaking, every Sun ONE Application Framework object (Model, View or Command) is a component. However, not all Sun ONE Application Framework components are destined for distribution in a component library. Some components are simply built as part of the standard process of building the application within the Sun ONE Studio, in which every Model, View, and Command is, technically speaking, a component. This distinction is acknowledged by referring to components which are included in libraries as distributable components, and components which are simply built within applications, as application-specific components, or non-distributable components. This is purely a distinction of terminology, not a hard formal distinction. Distributable components and application-specific component do not differ by type. The distinction is merely a soft categorization, meant to help distinguish the component author's role from the application developer's role. Application developers develop application-specific components. Component authors develop distributable components.

The first term, application-specific components, or non-distributable components refers to components which are only reusable within the application in which they are defined. They are not packaged into a component library. They generally do not have an explicit `ComponentInfo` associated with them. As an example, when application developers build a `ContainerView` or Model in their applications, they are implicitly building application-specific components. This is akin to a `javax.swing` application developer building an application specific panel or frame. Because the IDE toolset knows how to manipulate these application-specific components directly, they are usable within the same application without any additional work by the developer. For instance, after creating a new application specific Model, the application developer can then visually connect that new Model to Views within the current application. Development of an application-specific component is transparent and implicit and requires no component authoring knowledge per se.

Application-specific (non-distributable) components are:

- Implicitly developed by the casual application developer.
- Designed for use only within the current application.
- Not accompanied by any explicit `ComponentInfo`.

By contrast, distributable components refers to components which are reusable across many applications. Component authors package distributable components into component libraries. Component authors typically develop an explicit `ComponentInfo` class for each distributable component. Usually, the creation of a distributable component requires more foresight in design due to its greater ambition for reuse. To use the `javax.swing` analogy again, a distributable component would be a new sub-type of `javax.swing.JPanel` which is distributed for use in many new applications. In the Sun ONE Application Framework application, a distributable component might be a new type of `DisplayField`, or a specialized, but highly reusable type of `ContainerView`.

Distributable components are:

- Explicitly developed by someone with an understanding of the component model (a component author).
- Designed for reuse across applications.
- Accompanied by an explicit `ComponentInfo` class.
- Packaged into a library for distribution.

Of course, in accordance with common bottom-up design practices, it is not uncommon for an application-specific component to be explicitly "promoted" to distributable status. This happens when a development team identifies it as a valid candidate for reuse across applications. This is normal, expected, and encouraged. The promotion of a application-specific component to distributed status merely entails fulfilling the tasks that will be identified as standard for distributable components.

Therefore, in deference to the simplicity/transparency of creating application-specific components versus the relative complexity of authoring distributable components, the bulk of this document is dedicated to describing the process of authoring distributable components.

Extensible vs. Non-Extensible Components

In Sun ONE Application Framework component libraries, there is a formal distinction between extensible and non-extensible components. Component authors are responsible for designating a component as either extensible or non-extensible. This distinction allows component authors to control the manner in which the component-aware IDE toolset will expose a given component for usage by application developers. The IDE toolset will expose both extensible and non-extensible components in well-defined, but distinct fashions.

It is worth noting that while the distinction between extensible and non-extensible is important to a component author, practically speaking, component consumers are totally unaware of the distinction. That it to say, the IDE toolset will never present the application developer with either of these terms. Rather, the IDE toolset will automatically manage these subtleties so that application developers can just concentrate on building their applications. Application developers will generally never need to worry about whether a component is extensible or not, or even whether it has a `ComponentInfo` class.

Extensible Components

Extensible components are appropriate in those cases where the application assembly calls for the declaration of a new Sun ONE Application Framework sub-type (for example, a new type of Model, a new type of ContainerView, a new type of Command).

The IDE toolset will present extensible components for direct sub-classing by application developers. When an application developer selects an extensible component from the list of available components, the net result will be that the IDE toolset will create a new Java class that extends the selected component's class. A component author should designate a component as an extensible component if it is envisioned that the proper usage of a given component is through application specific sub-typing. Effectively, the Sun ONE Application Framework dictates where extensible components fit in. Wherever the Sun ONE Application Framework framework designates that an application entity must be a sub-type of a framework entity, that is where extensible components come into play.

Extensible components are designated by an <*extensible-component*> element within the component library manifest, as shown in the following example:

```
<extensible-component>
     <component-class>com.iplanet.jato.view.BasicViewBean</component-class>
     <component-info-class>com.iplanet.jato.view.BasicBeanComponentInfo</component-info-
class>
</extensible-component>
```

Extensible components:

- Allow application developers to create new types which extend the extensible component.
- Might be abstract.
- Can specify a component-specific Java file to serve as the template for the new type.

Examples are: Extensible `ViewBean`, `ContainerViews`, `Model` or `Command` components.

Non-Extensible Components

Non-extensible components are appropriate for those cases where the application assembly calls for the simple declaration and configuration of instances.

The net result of an application developer selecting a non-extensible component is that a new instance of the non-extensible component is declared in the application-specific class. For example, whenever a developer adds a text field or a button to a `ContainerView`, the IDE toolset will turn that design decision into a declaration of an instance of the text field or button in `ContainerView` class. In this manner, application developers populate the application-specific classes with instances of non-extensible components. This is the classic "assembly" model of component based development.

Again, the Sun ONE Application Framework dictates where this is appropriate. For instance, in developing an application-level Page (ViewBean) or Pagelet (ContainerView) component, the application developer expects to be able to add child view objects (such as DisplayFields) to that component. Consequently, the IDE toolset will present the application developer with a list of non-extensible components for direct addition to the page or pagelet.

Non-extensible components are designated by a <*component*> element within the component library manifest, as shown in the following example:

```
<component>
     <component-class>com.iplanet.jato.view.BasicChoiceDisplayField</component-class>
     <component-info-class>com.iplanet.jato.view.html2.ListBoxComponentInfo</component-
info-class>
</component>
```

Non-extensible components:

- Allow application developers to easily declare and configure new instances of the component.
- Cannot be abstract.
  - Fine grained component example: DisplayField components.
  - Coarse grained example: pre-packaged, fully configured non-extensible ContainerViews, Models and Commands.

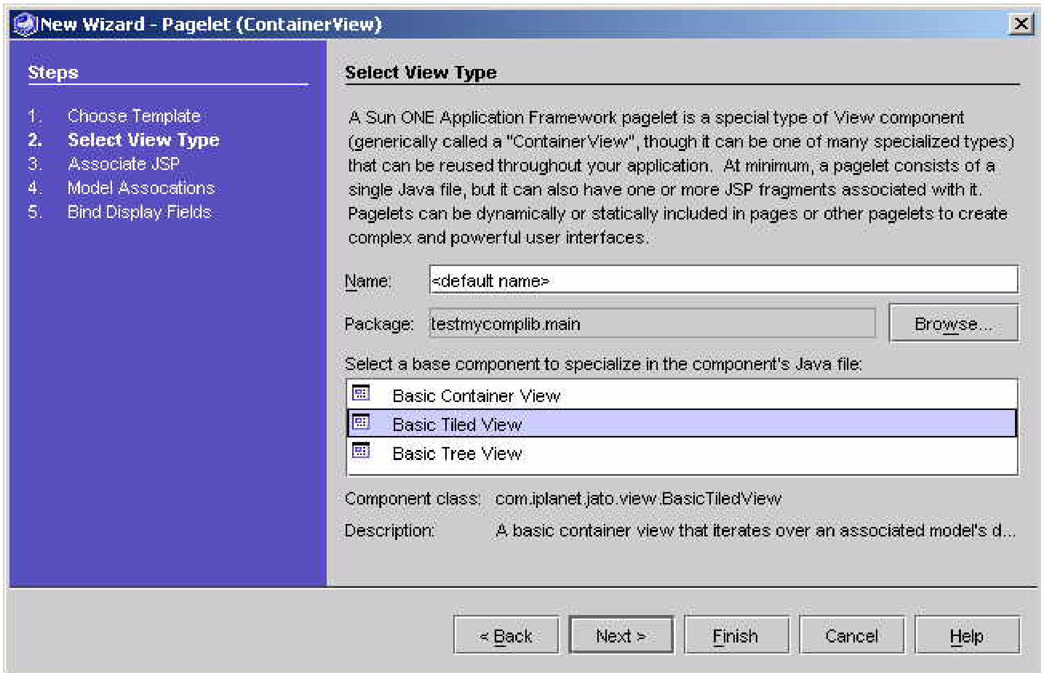
Extensible & Non-Extensible Components in the IDE

If you still find it confusing to distinguish extensible and non-extensible components, it might help at this point to refer to the Sun ONE Studio to see how the IDE toolset transparently exposes extensible and non-extensible components.

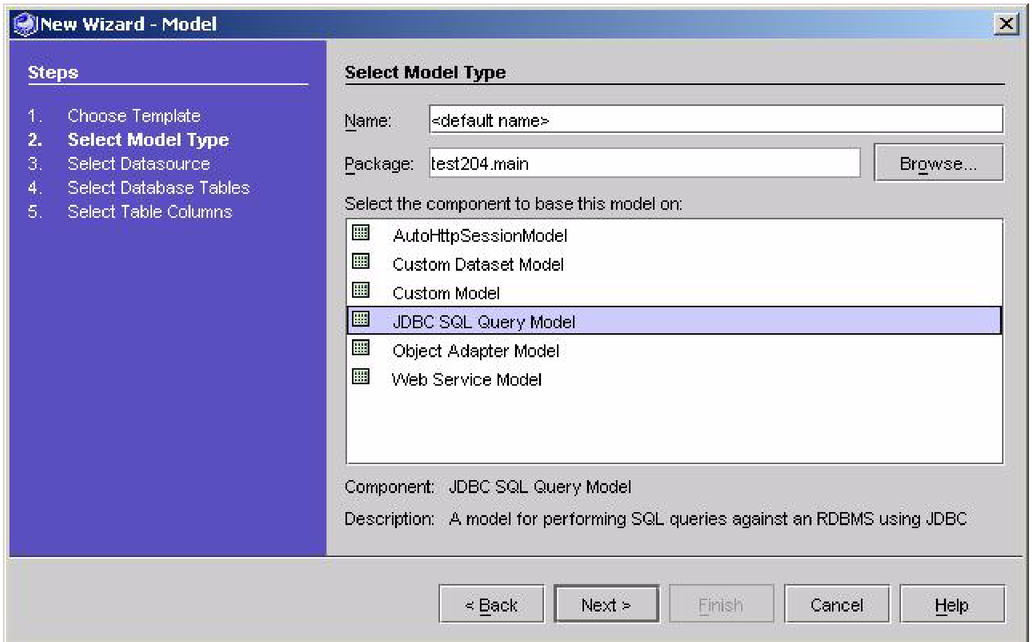
1. **Open a Sun ONE Application Framework project and select a "module" folder.**

2. **Right-click, and choose Add->Model or Add->Page (ViewBean) or Add->Pagelet (ContainerView).**

   These actions invoke wizards which contain an embedded extensible component browser pictured below.



3. **Complete either of the wizards, and you will see that the IDE toolset creates a new class that extends the extensible component's class.**



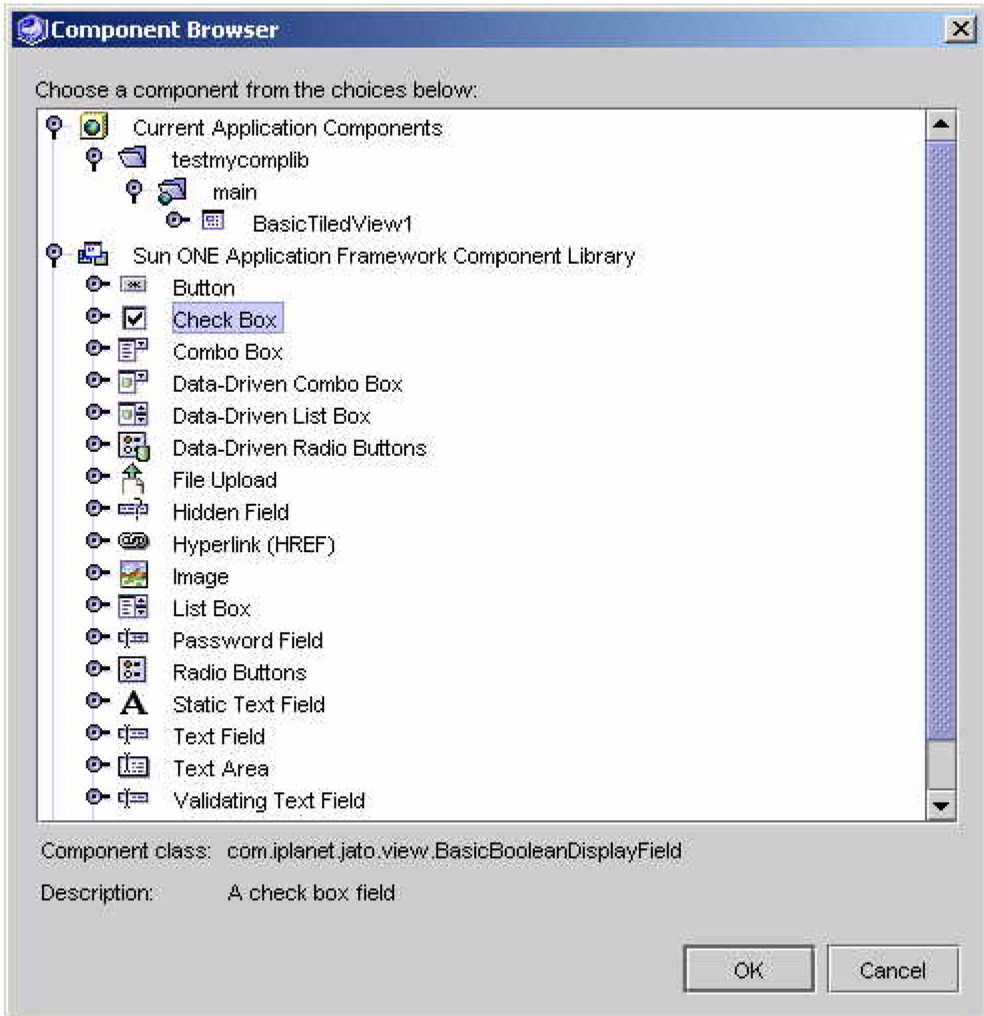
4. **Select an existing page or pagelet node.**

   Expand the top node so you can see its inner Visual Components node.

5. **Select the Visual Components sub-node, right-click, and select the Add Visual Component action.**

   This invokes the non-extensible component browser shown below.

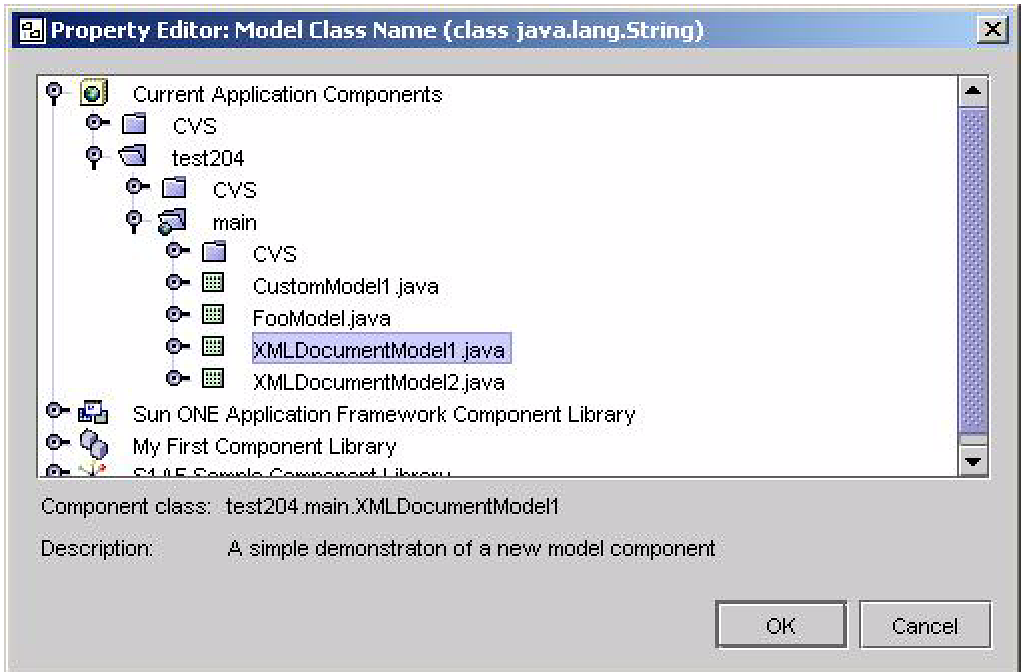
6. **Complete the selection of a child view and notice that this does not result in the creation of a new class, but rather adds a child element to the currently selected class.**



The figure above shows the Non-Extensible component browser employed in the context of "Add Visual Component" action. This figure shows the browser fully expanded to show two libraries and the current application's non-extensible components.

In other areas of the IDE, the non-extensible component browser is used to select Page/Pagelets, or Models, or Commands for assignment to certain property values. For instance, wherever a Sun ONE Application Framework use relationship is expressed in a property (for example, a View uses a Model), the property editor can leverage the non-extensible component browser to enable the application developer to select a valid target object.

For instance, properties of type `Model Class Name` are edited using a non-extensible Component Browser which shows non-extensible Model components in the mounted component libraries (if any), and also any Models which have been added to the current application. Similar behavior applies to editing the "Command Class Name" property, but in that case, Command components are selected instead of Models.



The figure above shows the Non-Extensible component browser employed in context of a Model Class Name property editor. This figure shows the browser fully expanded to show two libraries and the current application's non-extensible Model components.

ComponentInfo in More Detail

The `ComponentInfo` class is the heart and soul of the Sun ONE Application Framework 2.1 component model. Logically speaking, a Sun ONE Application Framework component can be defined as a tuple comprised of a component class and a `ComponentInfo` class. The `ComponentInfo` class provides the metadata that is introspected by the IDE toolset in order to provide the component's design-time presence. When you author a ComponentInfo class, you can focus exclusively on design-time considerations. The ComponentInfo class plays no run-time role in the Sun ONE Application Framework.

Specific ComponentInfo classes must implement the com.iplanet.jato.component.ComponentInfo interface, or one of its sub-interfaces. ComponentInfo class names must end with the "ComponentInfo" suffix. Whenever practical, the ComponentInfo class should share the same base name as the component class (for example, Foo and FooComponentInfo).

Here is an early glimpse into the Sun ONE Application Framework Component Library manifest. In the following snippet, you can see the simple declaration of a component as a component class and ComponentInfo tuple. Note in this example the extra designation of the *<extensible-component>* tag (for complete details of the Sun ONE Application Framework component manifest, see The Component Manifest, found in Chapter A, Component Library Structure.

```
<extensible-component>
     <component-class>com.iplanet.jato.view.BasicViewBean</component-class>
     <component-info-class>com.iplanet.jato.view.BasicViewBeanComponentInfo</component-
info-class>
</extensible-component>
```

However, the Sun ONE Application Framework allows ComponentInfo classes to differ in base name from their associated component class. In fact, the Sun ONE Application Framework allows more than one ComponentInfo class to be associated with the same component class. As stated earlier, logically speaking, a component is a tuple comprised of a component class and a ComponentInfo. The surprise is that the same component class might participate in more than one of these tuples.

This might not be immediately intuitive to most component authors, but it is a very effective and powerful feature of the Sun ONE Application Framework component model. For instance, in the com.iplanet.jato.view.html2 package, there are several ComponentInfo classes which are actually associated with the same component class. For example, the ListBoxComponentInfo, RadioButtonsComponentInfo and ComboBoxComponentInfo classes all specify com.iplanet.jato.view.BasicChoiceDisplayField as their component class.

Following is another actual snippet from the Sun ONE Application Framework Component Library manifest where you can see the component tuples described above:

```
<component>
     <component-class>com.iplanet.jato.view.BasicChoiceDisplayField</component-class>
     <component-info-class>com.iplanet.jato.view.html2.ListBoxComponentInfo</component-
info-class>
</component>
<component>
     <component-class>com.iplanet.jato.view.BasicChoiceDisplayField</component-class>
     <component-info-
class>com.iplanet.jato.view.html2.RadioButtonsComponentInfo</component-info-class>
</component>
<component>
     <component-class>com.iplanet.jato.view.BasicChoiceDisplayField</component-class>
     <component-info-class>com.iplanet.jato.view.html2.ComboBoxComponentInfo</component-
info-class>
</component>
```

These pairs form three distinct tuples, and therefore, three distinct logical components. The value presented by this freedom is that new component variations can be created by simply defining new `ComponentInfo` classes.

Note – To be anything more than just equivalent to other components that use the same component class, the new components must either expose existing component properties not exposed by other components (for example, only `ListBoxComponentInfo` exposes the "Allow Multiple Choices" property), or change other meaningful component metadata. In the examples provided above, the components primarily differ in the JSP tags that they declare, thereby drastically changing the way these components look and feel when added to an HTML page. However, the component functionality itself is essentially the same among all of them. The ability to declare different tags, and thus different rendering mechanisms for a component, is the most compelling reason to define components that use the same component underlying component class.

Unlike declarative metadata, a `ComponentInfo` is specified as a Java class. Therefore, new `ComponentInfo` classes can derive from existing `ComponentInfo` classes and benefit from standard inheritance of superclass functionality. The `com.iplanet.jato.component.SimpleComponentInfo` class can serve as a reliable starting point for any new `ComponentInfo` class, if there is not a more specific and more appropriate subtype already available.

Specialized ComponentInfo Interfaces

The Sun ONE Application Framework provides several specialized sub-types of the `ComponentInfo` which allow component authors to specify additional metadata that is appropriate for certain components. The IDE toolset leverages the additional metadata to provide special visual development support congruent with the additional metadata.

ExtensibleComponentInfo

The `com.iplanet.jato.component.ExtensibleComponentInfo` interface allows developers to provide additional metadata that is specifically appropriate for extensible components. In the IDE toolset, extensible components serve as the base classes when developers create new Sun ONE Application Framework types (Models, Pages/Pagelets, and Commands). To this end, the extra metadata defined in the `ExtensibleComponentInfo` interface allows the component author to influence the construction of the new type. Specifically, component authors might specify a Java class template to serve as the starting point for every new type derived from the extensible component.

Other Types of Specialized ComponentInfo

There are several other specialized types of `ComponentInfo`:

- `com.iplanet.jato.component.ExtensibleComponentInfo`
- `com.iplanet.jato.view.ViewComponentInfo`
- `com.iplanet.jato.view.ContainerViewComponentInfo`
- `com.iplanet.jato.command.CommandComponentInfo`
- `com.iplanet.jato.model.ModelComponentInfo`
- `com.iplanet.jato.model.ExecutingModelComponentInfo`

Details of these interfaces are discussed later in sections describing the steps required to create components of the various types to which these specialized `ComponentInfo` interfaces pertain.

Standard Implementations of ComponentInfo

Since the Sun ONE Application Framework component model is based on well-defined interfaces, component authors are free to implement these interfaces from scratch for any new component. However, the Sun ONE Application Framework generally provides ready-made implementations of all of the various specialized `ComponentInfo` interfaces, and component authors are encouraged to extend one of the existing implementations when writing their own components. This saves you labor and speeds your authoring process.

CHAPTER 3

Developing View Components

This section assumes that you have already read Develop Your First Component found in Chapter 2, "Developing Components" on page 17.

View Components

For background on Sun ONE Application Framework Views, see the *Sun ONE Application Framework Developer's Guide*.

View components are also referred to as visual components. The View term comes from the Model-View-Controller design pattern. Most of the types in the com.sun.iplanet library use the term view for this reason. The IDE however, caters to corporate developer expectations by using the term visual component more frequently than view, and page more frequently than ViewBean. For the purposes of this document, view component and visual component are synonymous. Child view component and child visual component are synonymous. Page component and ViewBean component are synonymous. Pagelet component and ContainerView component are synonymous.

Broadly speaking, there are two types of view components, extensible and non-extensible view components.

Extensible view components are custom implementations of Sun ONE Application Framework ContainerViews which are intended for further specialization by application developers. For instance, in the Sun ONE Application Framework Component Library, the Basic Container View, Basic Tiled View, and Basic ViewBean are all examples of extensible view components.

Do not read too much into the statement "specialization by application developers" above. Frequently, the only specialization an application developer will make is the addition of child view components (which is done via the IDE), and the logic associated with them.

The most recognizable, and easily comprehended non-extensible view components are custom implementations of the DisplayField interface. The Sun ONE Application Framework Component Library contains over a dozen DisplayField components. These fall easily into the classic widget or visual control category, and component developers and application developer alike intuitively relate to these components. As we shall see, the Sun ONE Application Framework goes well beyond this minimal component story and offers more potential in the component domain than many component authors and application developers might have seen before.

For instance, a less recognizable non-extensible view component would be ANY concrete ContainerView implementation created by the IDE toolset. Put bluntly, every ContainerView an application developer creates is a non-extensible component. This is a subtlety of the Sun ONE Application Framework approach where nearly everything is a component. Where these various type of components differ is in the way in which they are packaged for distribution and reuse.

ViewComponentInfo

The ViewComponentInfo interface allows component authors to specify additional metadata that is applicable to all view components. This interface is applicable to both extensible and non-extensible view components, and contains metadata such as which JSP tags should be associated with the view component.

We have indicated above that it is possible and expected that multiple `ComponentInfo` classes can be paired with a single component class to produce a variety of components. For example, the ListBoxComponentInfo, RadioButtonsComponentInfo and ComboBoxComponentInfo all specify `BasicChoiceDisplayField` classes as their component class. These form three distinct tuples, and hence three distinct logical components. One of the key ways in which these three components differ from each other is that they each implement the ViewComponentInfo's getJspTagDescriptors() method to return a different JspTagDescriptor. In summary, these components are nearly identical to each other except for the different JSP tags which the IDE toolset will generate when an instance of the component is added to the application. The opportunity this presents to component authors is quite liberating. A component author could create a whole new library of JSP tags that generate different markup and pair them with existing component classes simply by implementing additional `ComponentInfo` classes.

ContainerViewComponentInfo

The ContainerViewComponentInfo interface allows component authors to additional metadata that is applicable to all ContainerView components. This interface is only applicable to extensible view components.

Develop a Non-Extensible View Component

In this section we shall describe how to create a new TextField component that supports a rudimentary input validation feature. In the interest of simplicity, we shall keep the validation design and implementation to a minimum. This exercise is intended to focus on the mechanics of non-extensible view component design and as such, will only scratch the surface of validation support possibilities.

Note – The Sun ONE Application Framework Component Library 2.1.0 already contains a fully productized ValidatingTextField component. This educational exercise will result in your creating a validating text field component that approximates the Sun ONE Application Framework Component library's functionality. But the resulting component from this exercise will not be equivalent to the one in Sun ONE Application Framework Component library because this exercise will not attempt to implement all of the features of that productized component.

This example will allow us to cover several additional Sun ONE Application Framework component model topics, leveraging ViewComponentInfo, developing a new JSP tag and developing a ConfigurableBean.

Our validating text component should support the following design-time functionality:

- Expose a property called "Validator". This property will take a reference to a Validator object. The DisplayField will delegate validation to the Validator object.
- Expose a property called "Validation Failure Message". This property will take a simple String.

Our validating text component should support the following run-time functionality:

- Upon input, the component will delegate its input value for validation by the Validator object.
- If the value is invalid and the application redisplays the page, the validating component should display the invalid value followed by the application developer supplied validation failure message.

To meet these requirements, we will design and implement the following classes:

- Component class - `mycomponents.ValidatingDisplayField`
- ComponentInfo class - `mycomponents.ValidatingTextFieldComponentInfo`

- JSP `TagHandler` class - `mycomponents.ValidatingTextFieldTag`
- A Validator interface - `mycomponents.Validator`
- An implementation of the Validator interface - `mycomponents.TypeValidator`

We shall also define a new JSP tag library - mycomponents.tld

Finally we shall edit the mycomponents complib.xml to add the new component, taglibrary and ConfigurableBean to the Sun ONE Application Framework component library.

Create the Validator Interface

1. **In any Java editor create the class** `mycomponents.Validator`
2. **Define a very simple validation API.**

   Design principle hint: In designing the Validator as an interface we are setting the stage to leverage the power of the Sun ONE Application Framework component model's ConfigurableBean story. Specifically, we will subsequently define a ValidatingTextField property to be of type "Validator". And as we shall see, the IDE toolset will allow the application developer to choose from a dynamically list of ConfigurableBean types which implement that interface. Furthermore, third party component authors can augment this component story by authoring an distributing additional ConfigurableBean implementations of the same interface.

   After these steps, mycomponents/Validator.java should look like this

```
package mycomponents;


/**
 *
 * @author component-author
 */
public interface Validator {


        /**
         *
         *
         */
        public abstract boolean validate(Object value);
}
```

Create at least one implementation of the Validator interface

1. **In any Java editor create the class** `mycomponents.TypeValidator`

2. **Add a String property called ValidationRule**
3. **Implement the Validator interface**

   After these steps, mycomponents/Validator.java should look like this:

```
package mycomponents;
import com.iplanet.jato.model.*;
import com.iplanet.jato.util.*;


public class TypeValidator implements Validator
{

     public TypeValidator()
     {
          super();
     }

     public String getValidationRule()
     {
          return rule;
     }

     public void setValidationRule(String value)
     {
          rule=value;
     }

     public boolean validate(Object value)
     {
          if (getValidationRule()==null)
               throw new ValidationException("No validation rule has been set");

          try
          {
               value=TypeConverter.asType(getValidationRule(),value);
          }
          catch (Exception e)
          {
               return false;
          }

          return true;
     }
     ////////////////////////////////////////////////////////////////////////////
     // Instance variables
     ////////////////////////////////////////////////////////////////////////////

     private String rule;
}
```

Design hint: The rudimentary implementation of TypeValidator above exposes the ValidationRule as a simple String property. In the absence of any further work, the IDE toolset will expose this property for editing with the default String editor. This will require application developers to explicitly set the value of the property to "java.lang.String" or "java.lang.Integer" or "java.lang.Float". That is not a very user friendly user interface. Since this ValidationRule falls into the ConfigurableBean category, the component author can make use of the full JavaBean component model to improve the user experience. Ideally, a component author would also design and deploy a custom property editor for this property. In this case, a simple drop down list property editor would be a big improvement over the default String editor. Then the component author can create a TypeValidatorBeanInfo which would specify the custom property editor of his choice. For more on this topic see Component Design Guidelines.

Create the Sun ONE Application Framework Component Class

1. **In any Java editor create the class** `mycomponents.ValidatingDisplayField`
2. **Make ValidatingDisplayField extend** `com.iplanet.jato.view.BasicDisplayField`
3. **Implement the appropriate constructor for the component type.**

   All DisplayField components must implement a two-arg constructor that takes a View parent and a String name. The IDE toolset assumes that all DisplayField components will implement this constructor.
4. **Add a get and set method for the property Validator**
5. **Add a get and set method for the property ValidationFailureMessage**
6. **Implement the remaining methods that are required to fulfill our requirements.**
   - A flag to indicate the valid/invalid state
   - A buffer to hold the invalid value(s) for redisplay.
   - Overridden implementations of setValue which will invoke the Validator
   - Overridden implementations of getValue which will conditionally return the buffered invalid value

   After these steps, mycomponents/ValidatingDisplayField.java should look like this:

```
package mycomponents;
import com.iplanet.jato.view.*;
import com.iplanet.jato.model.*;
import com.iplanet.jato.util.*;


public class ValidatingDisplayField extends BasicDisplayField {
```

```
    public ValidatingDisplayField(View parent, String name) {
         super(parent, name);
    }

    public Validator getValidator()
    {
         return validator;
    }


    public void setValidator(Validator value)
    {
         validator=value;
    }

    public String getValidationFailureMessage()
    {
         return validationFailureMessage;
    }

    public void setValidationFailureMessage(String value)
    {
         validationFailureMessage=value;
    }

    public boolean isValid()
    {
              return isValid;
    }

    public void setValid(boolean value)
    {
              isValid = value;
    }

    ////////////////////////////////////////////////////////////////////////////
    // Value methods
    ////////////////////////////////////////////////////////////////////////////

    public Object getValue()
    {
         if (!isValid())
              return getInvalidValue();
         else
              return super.getValue();
    }

    public Object getInvalidValue()
    {
         if (invalidValue !=null)
              return invalidValue;
```

```
        else
            return null;
    }

    public void setValue(Object value)
    {
        if (value!=null && getValidator()!=null)
        {
            if (getValidator().validate(value))
            {
                try
                {
                    super.setValue(value);
                    setValid(true);
                }
                catch (ValidationException e)
                {
                    setValid(false);
                    invalidValue=value;
                    setValidationFailureMessage("Exception setting value \""+
                        "on model: "+ e.getMessage());
                }
            }
            else
            {
                setValid(false);
                invalidValue=value;
            }
        }
        else
            super.setValue(value);
    }


    ////////////////////////////////////////////////////////////////////////////
    // Instance variables
    ////////////////////////////////////////////////////////////////////////////

    private Validator validator;
    private String validationFailureMessage;
    private boolean isValid = true;

    private Object invalidValue;
}
```

Create a Custom JSP TagHandler Class

Our requirements call for the `ValidatingComponent` class to display its validation error message. One way to achieve this, and the approach we shall pursue here, is to pair our new component with a custom JSP `TagHandler` class. This will allow us to fully control the rendering of the component.

1. **In any Java editor create the class** `mycomponents.ValidatingTextFieldTag`
2. **Extend this class from** `com.iplanet.jato.taglib.html.TextFieldTag`
3. **Override the doEndTag method to conditionally append the validation error message whenever the component is not valid.**

   After these steps, mycomponents/ValidatingTextFieldTag.java should look like this:

```
package mycomponents;
import com.iplanet.jato.util.*;
import javax.servlet.jsp.*;
import com.iplanet.jato.taglib.html.*;
import com.iplanet.jato.util.*;
import com.iplanet.jato.view.*;
public class ValidatingTextFieldTag extends TextFieldTag
{

     public ValidatingTextFieldTag()
     {
          super();
     }


     public int doEndTag()
          throws JspException
     {
          int result=super.doEndTag();

          ContainerView parentContainer=getParentContainerView();
          View child=parentContainer.getChild(getName());
          checkChildType(child,ValidatingDisplayField.class);

          ValidatingDisplayField field=(ValidatingDisplayField)child;
          // If the field is valid, do nothing.
          if (field.isValid())
               return result;

          // Append the validation error message in Red
          NonSyncStringBuffer buffer=new NonSyncStringBuffer(
               " <font color=\"#FF0000\">");
          buffer.append(field.getValidationFailureMessage());
          buffer.append("</font>");
          writeOutput(buffer);
```

```
            return result;
        }
    }
```

Note – The Sun ONE Application Framework component model allows component authors to specify multiple JSP TagHandlers for a given component. For more on that subject see the JspTagDescriptor API.

Create the ComponentInfo Class

The `ComponentInfo` class defines the design-time metadata that the IDE toolset requires to incorporate the component. In this example we will extend an existing `ComponentInfo` class and in true OO style, simply augment it. We could, of course, choose to implement the ComponentInfo interface from scratch, but that would be unproductive in this case.

Note that in this example we are going beyond the functionality revealed in our first component example. Below, we are going to take advantage of the key metadata opportunity provided by the ViewComponentInfo interface, the ability to describe JSP tag(s) for a given component.

1. **Create the class** `mycomponents.ValidatingTextFieldComponentInfo`**.**
2. **Make the** `ValidatingTextFieldComponentInfo` **class extend** `com.iplanet.jato.view.html2.TextFieldComponentInfo`
3. **Implement the no-arg constructor.**
4. **Implement the** `getComponentDescriptor()` **method to provide the basic design-time description of the component.**
5. **Implement the** `getConfigPropertyDescriptors()` **method to identify which properties you wish to expose in the IDE.**
   - Add a `ConfigPropertyDescriptor` for the `Validator` property.
   - Add a `ConfigPropertyDescriptor` for the `ValidationFailureMessage` property.

6. **Implement the `getJspTagDescriptors()` method to specify the JSP tag which you wish the IDE toolset to automatically add to associated JSP(s) whenever an instance of this component is added to a ViewBeans/ContainerViews.**

   After these steps, `mycomponents/ValidatingTextFieldComponentInfo.java` should look like this:

   In this sample code we have embedded String values directly for ease of demonstration. If you anticipate the need to localize your display strings, we encourage you to utilize resource bundles.

```
package mycomponents;
import java.beans.*;
import java.util.*;
import com.iplanet.jato.component.*;
import com.iplanet.jato.taglib.*;
import com.iplanet.jato.view.*;
import com.iplanet.jato.view.html2.*;


public class ValidatingTextFieldComponentInfo extends TextFieldComponentInfo {

    public ValidatingTextFieldComponentInfo() {
        super();
    }

    public ComponentDescriptor getComponentDescriptor()
    {
        // identify the component class
        ComponentDescriptor result=new ComponentDescriptor(
            "mycomponents.ValidatingDisplayField");

        // The name will be used to determine a name for the component instance
        result.setName("ValidatingTextField");

        // The display name will be used to show the component in a chooser
        result.setDisplayName("ValidatingTextField Component");

        // The description will be the tool tip text for the component
        result.setShortDescription("A simple validating text field component");

        return result;
    }

    public ConfigPropertyDescriptor[] getConfigPropertyDescriptors()
    {
        if (configPropertyDescriptors!=null)
            return configPropertyDescriptors;

        // get any properties defined in the super class
        configPropertyDescriptors=super.getConfigPropertyDescriptors();
```

```
        List descriptors=new LinkedList(Arrays.asList(configPropertyDescriptors));

        ConfigPropertyDescriptor descriptor = null;

        descriptor=new ConfigPropertyDescriptor(
            "validator",Validator.class);
        descriptor.setDisplayName("Validator");
        descriptor.setHidden(false);
        descriptor.setExpert(false);
        descriptors.add(descriptor);

        descriptor=new ConfigPropertyDescriptor(
            "validationFailureMessage",String.class);
        descriptor.setDisplayName("Validation Failure Message");
        descriptor.setHidden(false);
        descriptor.setExpert(false);
        descriptors.add(descriptor);

        // Create/return the array
        configPropertyDescriptors = (ConfigPropertyDescriptor[])
            descriptors.toArray(
                new ConfigPropertyDescriptor[descriptors.size()]);
        return configPropertyDescriptors;
    }

    public JspTagDescriptor[] getJspTagDescriptors()
    {
        JspTagAttributeDescriptor[] attrs=new JspTagAttributeDescriptor[1];
        attrs[0]=new JspTagAttributeDescriptor(
            TagBase.ATTR_NAME,JspTagDescriptor.ASSUMED_PROPERTY_NAME,null);

        JspTagDescriptor htmlTagDescriptor=new JspTagDescriptor(
            JspTagDescriptor.ENCODING_HTML,"validatingTextField",
            "/WEB-INF/mycomplib.tld",attrs);

        return new JspTagDescriptor[] {htmlTagDescriptor};
    }

    private ConfigPropertyDescriptor[] configPropertyDescriptors;
}
```

Create a New Tag Library TLD File

Since we have defined a new JSP TagHandler. We must create a JSP library TLD file for our component library.

Please note that there is a 'soft' restriction on your custom JSP library. During IDE operations a logical object model is created for the working JSP files. This JSP object model is used by the page and pagelet view component mechanisms to manage the placement of tags in the JSP will the views are mutated. While parsing the JSP file to create the JSP object model, tags for Sun ONE Application Framework component tag libraries have special treatment. If your custom JSP tag library has additional tags which are not related to a Sun ONE Application Framework view component, these tags might be categorized incorrectly in the JSP object model. You should isolate your Sun ONE Application Framework related tags in their own tag library. Internally in the JSP object model, tags from tag libraries specified in the component library manifest will be categorized as "JATO" tags while all other tags in the JSP file are categorized as "OTHER" tags. The reason why this is a 'soft' restriction is that there is only an edge case where a non Sun ONE Application Framework tag would interfere with view component tag management. If a non Sun ONE Application Framework tag remains in your component tag library, and if that tag has an attribute "name" who's value collides with a "name" attribute of a true Sun ONE Application Framework tag, the JSP object model might not operate properly. In other words, if you have non Sun ONE Application Framework tags which have a "name" attribute, you should try and isolate these tags in separate tag library to avoid edge case problems.

The library TLD file name is arbitrary. Its location within the library is also arbitrary. In a later step we shall declare the new TLD file in our component manifest. A full discussion of JSP tld files is beyond the scope of this document. Suffice to say, for this example we only need to declare a new library (mycomlib) containing a single tag element (validatingTextField). All of the tag attributes can be copied verbatim from the declaration of the TextField tag in the Sun ONE Application Framework Component Library's jato.tld file. You can find the jato.tld file located in the WEB-INF\tld\com_iplanet_jato directory of any Sun ONE Application Framework application created by the Sun ONE Studio.

1. **Create the file mycomponents/mycomplib.tld**
2. **Add the basic tld information to declare a new tag library.**
3. **Add a tag element for our new tag validatingTextField and its corresponding tag-class** `mycomponents.ValidatingTextFieldTag`

4. **Complete the tag element declaration by adding all desired tag attributes. Our recommendation in this case is to simply copy those already defined in `jato.tld` for the Sun ONE Application Framework Component Library's TextField tag**

   After these steps, the `mycomponents/mycomplib.tld` file should look like this

```
 <?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE taglib
         PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
         "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<!-- template test -->

<taglib>
     <tlib-version>1.0</tlib-version>
     <jsp-version>1.2</jsp-version>
     <short-name>mycomponents.mycomplib</short-name>
     <display-name>mycomponents.mycomplib</display-name>
     <tag>
          <name>validatingTextField</name>
          <tag-class>mycomponents.ValidatingTextFieldTag</tag-class>
          <body-content>empty</body-content>
          <display-name>Validating Text Field</display-name>
          <description></description>
          <attribute>
               <name>name</name>
               <required>true</required>
               <rtexprvalue>false</rtexprvalue>
                    <type>String</type>
          </attribute>
...
          <!-- more attribute definitions follow -->
...

     </tag>
</taglib>
```

Augment the Component Library Manifest

We have already created the component manifest in the earlier example. So now we will simple add additional information. Note that we will add additional types of information not seen in the prior example.

The Sun ONE Application Framework library manifest must be named complib.xml. Within the JAR file, the Sun ONE Application Framework library manifest must be placed in the /COMP-INF directory.

1. **Create/Open the file COMP-INF/complib.xml**

2. **Add a component element to declare the ValidatingTextField component.**
3. **Add a ConfigurableBean element to declare the mycomponents.TypeValidator**
4. **Add a taglib element to declare the mycomplib.tld.**

   After these steps, the COMP-INF/complib.xml file should look like this. Make sure that the tld is a well formed XML document. Even something as minor as an inappropriate leading spaces before the first XML tag can create a malformed document. If your tld file is not well formed XML, certain servlet containers will fail to load your entire Web application. Such errors might be difficult to track down.

```
<?xml version="1.0" encoding="UTF-8"?>
<component-library>
<tool-info>
<tool-version>2.1.0</tool-version>
</tool-info>
<library-name>mycomponents</library-name>
<display-name>My First Component Library</display-name>
<!-- Your icon here
<icon>
<small-icon>/com/iplanet/jato/resources/complib.gif</small-icon>
</icon>
-->
<interface-version>1.0.0</interface-version>
<implementation-version>20030221</implementation-version>

<component>
<component-class>mycomponents.MyTextField</component-class>
<component-info-class>mycomponents.MyTextFieldComponentInfo</component-info-class>
</component>
<component>
<component-class>mycomponents.ValidatingDisplayField</component-class>
<component-info-class>mycomponents.ValidatingTextFieldComponentInfo</component-info-class>
</component>

<configurable-bean>
<bean-class>mycomponents.TypeValidator</bean-class>
</configurable-bean>

<taglib>
<taglib-uri>/WEB-INF/mycomplib.tld</taglib-uri>
<taglib-resource>/mycomponents/mycomplib.tld</taglib-resource>
<taglib-default-prefix>mycomp</taglib-default-prefix>
</taglib>

</component-library>
```

Re-create the Component Library JAR File

Once again Jar up the component classes as we did in the first example so that they can be ready for distribution as a library.

1. **The name of the JAR file is arbitrary. In this case, name it "mycomponents.jar"**
2. **You can omit the Java source files from the JAR**
3. **You should include in the JAR any necessary ancillary resources, like icon images, or resource bundles. In this case there are none.**

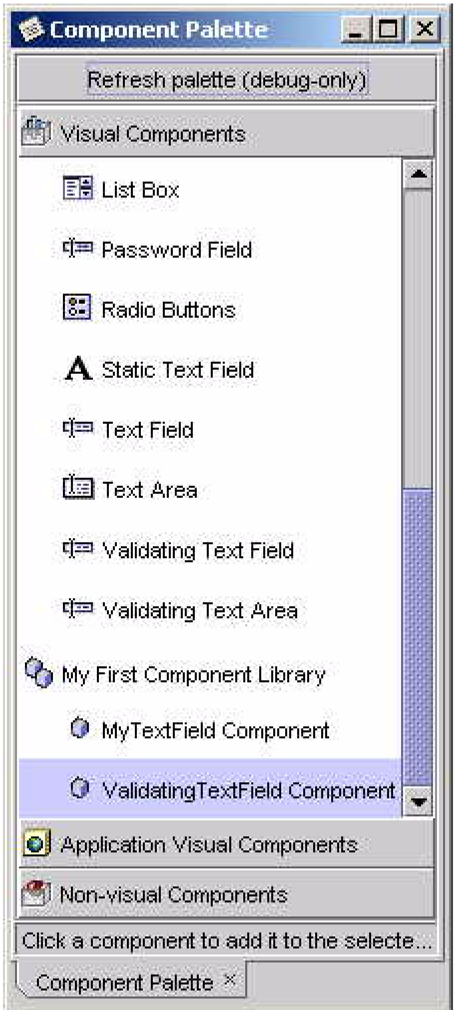
   In this case we are now including several new classes and a new JSP tag library.
4. **The mycomponents.jar internal structure should look like this:**

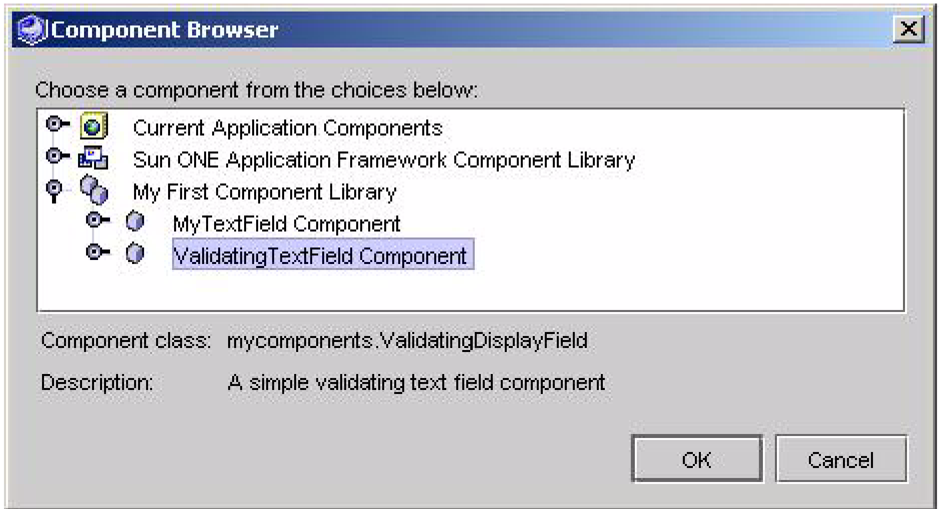
```
mycomponents/MyTextField.class
mycomponents/MyTextFieldComponentInfo.class
mycomponents/TypeValidator.class
mycomponents/ValidatingDisplayField.class
mycomponents/ValidatingTextFieldComponentInfo.class
mycomponents/ValidatingTextFieldTag.class
mycomponents/Validator.class
mycomponents/mycomplib.tld
COMP-INF/complib.xml
```

Test the New Component

Your library is now ready for testing and distribution. We recommend testing it in a sample project. This stage requires the use of the Sun ONE Studio with the Sun ONE Application Framework module installed and enabled. If you have never built a Sun ONE Application Framework application in the Sun ONE Studio we suggest that you stop now, and before continuing, complete the *Sun ONE Application Framework Tutorial*.

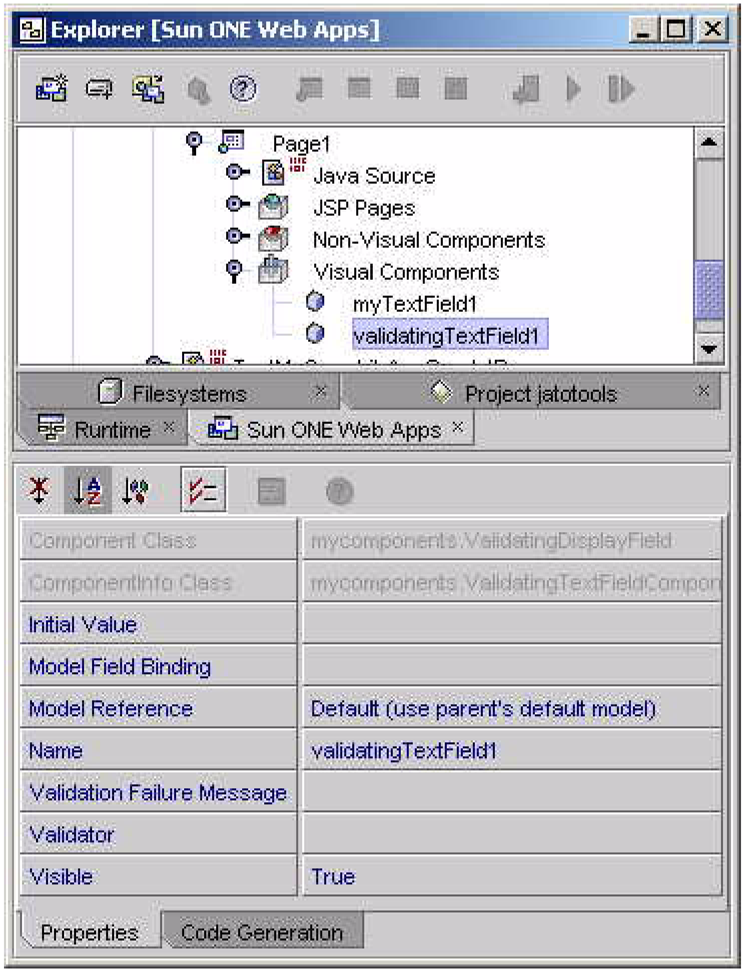
1. **Deploy the new version of the library into your previously created test application**
   - Important Sun ONE Studio note - the Studio will not let you delete or copy over a jar file that is currently mounted. This presents a bit of a challenge when iteratively developing a component library and testing that library in a test application. We recommend the following for repeatedly testing new versions of the same library jar file within a test application.
   - Unmount the test application.
   - After the unmount is complete, go to your operating system file system and copy the new library jar file over the old library jar file in the unmounted test application's `WEB-INF/lib` directory.
   - Remount the test application, the test application should now pick up the new library version.
   - Normally, those steps work fine. If you encounter a spurious failure that either prevents you from copying the new jar over the old jar, or failure to remount the test application properly, the fallback strategy is to restart the Studio.
2. **Select the previously created Page1 object**
3. **Add an instance of the ValidatingTextFieldComponent to Page1. You can either select the component from the Component Palette, or select the Page1's Visual Components sub-node, right-click, and select the Add Visual Component... action from the pop-up menu.**


Component Palette
Refresh palette (debug-only)
Visual Components
List Box
Password Field
Radio Buttons
Static Text Field
Text Field
Text Area
Validating Text Field
Validating Text Area
My First Component Library
MyTextField Component
ValidatingTextField Component
Application Visual Components
Non-visual Components
Click a component to add it to the selecte...
Component Palette



4. **Select the "ValidatingTextField Component" from the list and observe how a child view named "validatingTextField1" is added to the page.**

5. **Now select the validatingTextField1 visual component node. Observe how the IDE's property sheet now displays our custom Validator and Validation Failure Message properties in addition to the inherited TextField component properties.**

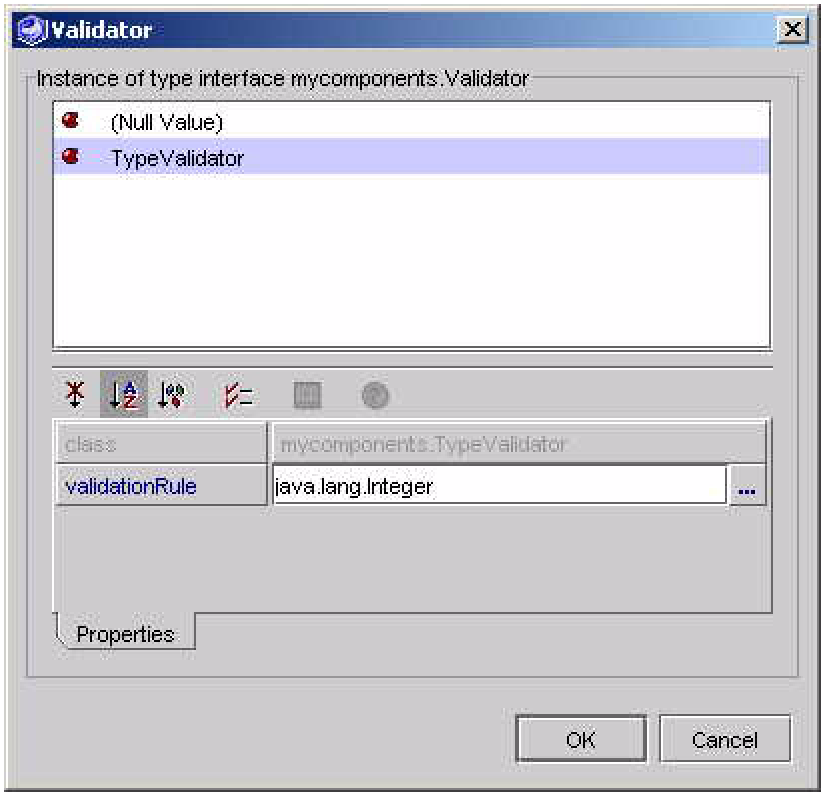


6. **Edit the Validation Failure Message property. Set it to "This is a test failure message" (or anything you like).**

7. **Edit the Validator property.**

   This should bring up the following dedicated ConfigurableBean editor. For test purposes, set the validationRule property to "java.lang.Integer".

   Make sure you specify the fully qualified class name for the validationRule property. Just "Integer" will not evaluate properly at run-time, it must be fully qualified, as in "java.lang.Integer".



8. **Now observe the code generation inside the page's java file. You should see a block of code inside the createChildReserved method that looks like the following (the indenting in your code might differ):**

```
...
else if (name.equals(CHILD_VALIDATING_TEXT_FIELD1)) {
mycomponents.ValidatingDisplayField child =
new mycomponents.ValidatingDisplayField(this, CHILD_VALIDATING_TEXT_FIELD1);
mycomponents.TypeValidator validatorVar =
new mycomponents.TypeValidator();

{ // begin local variable scope
validatorVar.setValidationRule("java.lang.Integer");
} // end local variable scope
child.setValidator(validatorVar);
child.setValidationFailureMessage( "This is a test failure message");
return child;
}
...
```

9. **Open the associated JSP file to observe the inclusion of the validatingTextField tag.**

   Note also the automatic inclusion of a mycomplib.tld directive. (The overall look of your test application's JSP may differ from the one below, depending upon whether you have added other child views you to your test page in addition to validatingTextField1)

```
<%@page contentType="text/html; charset=ISO-8859-1" info="Page1" language="java"%>
<%@taglib uri="/WEB-INF/jato.tld" prefix="jato"%>
<%@taglib uri="/WEB-INF/mycomplib.tld" prefix="mycomp"%>

<jato:useViewBean className="testmycomplib.main.Page1">

<html>
<head>
<title>Page1</title>
</head>
<body>
<jato:form name="Page1" method="post">
<jato:textField name="myTextField1"/>
<mycomp:validatingTextField name="validatingTextField1"/>
</jato:form>
</body>
</html>

</jato:useViewBean>
```

10. **Before you can effectively test run Page1, you will need to add a button and request handling code. For your test purposes, we suggest that you take the following steps to add a button and some request handling code, which will redisplay the page following a submit. This will allow us to see if the ValidatingTextComponent is behaving as designed. If you have not done so already add a button and some request handling code, follow the steps below.**

Note – The steps below, represent conventional Sun ONE Application Framework application development practice, the details of which are beyond the scope of this document. These steps, or similar ones, are required to create an effective test page).

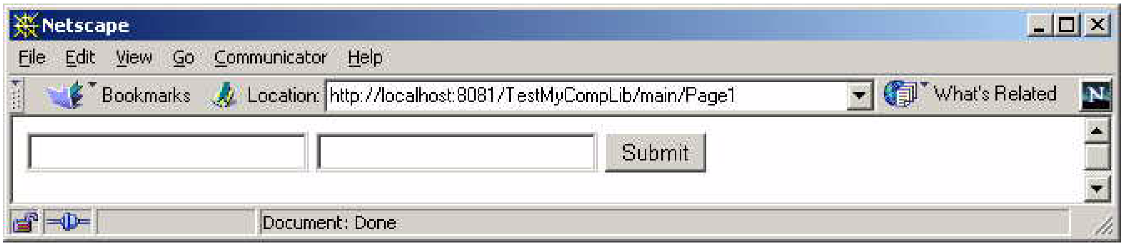
- Add an instance of the Sun ONE Application Framework Library's Basic Button to Page1. You may either select the component from the Component Palette or select the Page1's Visual Components sub-node, right mouse and select the Add Visual Component... action from the pop-up menu.
- This will add a "button1" child to your test ViewBean
- Now, select the button1 visual component node.
- Right mouse and select the pop up menu's Events->handleRequest action.
- This will add an event handler method named handleButton1Request to your ViewBean's Java file.
- For this test, you will not need to modify the body of handleButton1Request since it is auto-generated to redisplay the current page, which is precisely the test we are looking for.
- Make sure your request handler looks like this:

```
public void handleButton1Request(RequestInvocationEvent event) throws Exception {
getParentViewBean().forwardTo(getRequestContext());
}
```

11. **Test run Page1.**

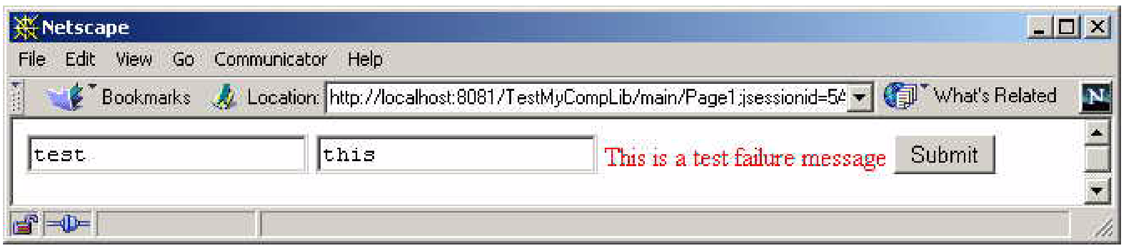
    See the *Sun ONE Application Framework Tutorial* if you do not already know how to test run a Sun ONE Application Framework ViewBean.

12. **The Page1 output should appear in a browser looking like this (it now contains two text fields, one instance of MyTextField and one instance of ValidatingTextField.**



13. **Enter an invalid value (for example, any value other than an integer) in the ValidatingTextField's text input and submit the page.**

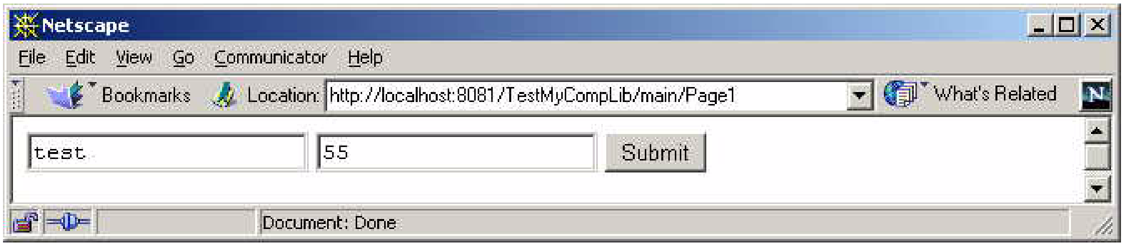
    The page should immediately be redisplayed with the text of the Validation Error Message property immediately following the ValidatingTextField.



14. **Enter a valid value (for example, 55 or any other valid integer) and submit the page.**

    The page should be redisplayed without the Validation Error Message text.

    If you continue to get the Validation Error Message, go back and verify that you set the value of validatingTextField1's Validator->ValidationRule property to "java.lang.Integer", not just "Integer".



Ship It!

Now that your component is functioning properly you may ship it. However, you may also go back and enhance it. For instance, you may decide that requiring the end user to type "java.lang.Integer" into the Validator's ValidationRule property is unacceptably error prone. If so, then you should spend a little time and develop a custom property editor. The details of that are beyond the scope of this document, but can be found in any basic JavaBean reference.

Develop an Extensible View Component

In this section we shall describe how to create a new ViewBean component that supports a rudimentary page level security feature. In the interest of simplicity, we shall keep the security model and implementation to a minimum. This exercise is intended to focus on the mechanics of extensible View component design and as such, will only scratch the surface of security model possibilities. Upon completion of this section you should have a good understanding of the role of extensible components within the Sun ONE Application Framework. Please bear in mind that this example will implement several optional features and goes beyond the bare minimum required to author an extensible View component.

This example will allow us to introduce several additional Sun ONE Application Framework component model topics

- ExtensibleComponentInfo
- Component supplied Java templates
- IndexedConfigPropertyDescriptor
- EventHandlerDescriptor

As a basic design principle, the Sun ONE Application Framework prefers to be enabling rather than prescriptive when it comes to application and page level security, since developer preferences in this domain vary widely. This example demonstrates that the Sun ONE Application Framework can easily enable an arbitrary page level security model. It is not meant to suggest that this example is the ultimate or recommended implementation.

Our secure ViewBean component should support the following design-time functionality:

- Each secure ViewBean will expose an indexed property called “RequiredTokens”. Application developers will configure this property to specify an arbitrary list of “required” String tokens (i.e. the tokens that are required to gain access to the current page)
- Each secure ViewBean will expose an indexed property called "GrantTokens". Application developers will configure this property to specify an arbitrary list of "grant" String tokens (i.e. the tokens that will be granted to the user after they successfully access the current page)
- Each secure ViewBean will expose the "handleMissingTokens" event handler for custom implementation. This means Sun ONE Studio developers can select the "handleMissingTokens" from the "Events" pop up menu, and the IDE toolset will automatically insert the event handler into the current secure ViewBean's Java file. This is an advanced and optional feature of the Sun ONE Application Framework component model.

Our secure ViewBean component should support the following run-time functionality:

- Each secure ViewBean can "grant" tokens to users who successfully access the current secure ViewBean. Thus, an application user will "accumulate" tokens as they proceed through the application.
- Also, each secure ViewBean will limit run-time access to itself through a simple comparison of required tokens to user accumulated tokens. If the application user has not accumulated all of the required tokens, a MissingTokensEvent will be fired. A specific event handler method called handleMissingTokens will be invoked. The secure ViewBean base class implementation of handleMissingTokens will throw a SecurityCheckException. The Sun ONE Application Framework will automatically process an uncaught SecurityCheckException as it does any uncaught exception (i.e. it will return the standard Sun ONE Application Framework error page to the user). Individual secure ViewBeans can override the implementation of the handleMissingTokens in order to perform arbitrary context specific behavior.

This run-time model assumes that both the grant tokens and the required tokens will be specified on a per secure ViewBean basis by the application developers.

The implementation of the secure ViewBean component is responsible for tracking the accumulated tokens at run-time and enforcing the security model described above. The implementation shall store the accumulated tokens per user in a special HttpSession attribute.

Please note, the choice to implement the base class version of handleMissingTokens to throw a SecurityCheckException is purely arbitrary. Alternatively, we could implement that method to transfer control to a more user friendly error page, or anything else that the component author prefers. Strictly in the interest of brevity and simplicity, we choose to throw a SecurityCheckException.

To meet these requirements, we will design and implement the following classes:

- Component class - mycomponents.SecureViewBean
- ComponentInfo class - mycomponents.SecureViewBeanComponentInfo
- A simple event class - mycomponents.MissingTokensEvent

Additionally, we will implement a custom Java template which the IDE toolset will use as the basis for application specific sub-types of our SecureViewBean.

Finally we shall edit the mycomponents complib.xml to add the new component to the Sun ONE Application Framework component library.

Create the MissingTokensEvent Class

1. **In any Java editor create the class mycomponents.MissingTokensEvent**
2. **Define a very simple event API, that will allow the event handler to discover which tokens were missing.**

   After these steps, mycomponents/MissingTokensEvent.java should look like this

```
package mycomponents;
  import java.util.*;

  public class MissingTokensEvent extends Object {

    public MissingTokensEvent(List tokens) {
        missingTokens = new ArrayList(tokens);
    }

    public String toString() {
        Iterator iter = missingTokens.iterator();
        StringBuffer buff = new StringBuffer();
        buff.append("MissingToken count[" + missingTokens.size() + "] ");
        while(iter.hasNext()) {
            buff.append("Token[" + (String)iter.next() + "] ");
        }
        return buff.toString();
    }

    public ArrayList getMissingTokens() {
        return missingTokens;
    }

    private ArrayList missingTokens = null;

}
```

Create the Sun ONE Application Framework Component Class

1. **In any Java editor create the class mycomponents.SecureViewBean**
2. **Make SecureViewBean extend com.iplanet.jato.view.BasicViewBean**
3. **Implement the appropriate constructor for the component type. All ViewBean components must implement a no-arg constructor.**
4. **Add a get and set method for the property named "RequiredTokens"**
5. **Add a get and set method for the property named "GrantTokens"**
6. **Implement the remaining methods that are required to fulfill our component specific requirements.**
   - Overridden implementation of the securityCheck method which will enforce the component's page security model

- Default implementation of the component's handleMissingTokens method.

After these steps, mycomponents/SecureViewBean.java should look like this:

```
package mycomponents;
import java.util.*;
import com.iplanet.jato.view.*;
import com.iplanet.jato.*;

public class SecureViewBean extends BasicViewBean {

    public SecureViewBean()
    {
        super();
    }

    public String[] getRequiredTokens()
    {
        return requiredTokens;
    }

    public void setRequiredTokens(String[] value)
    {
         requiredTokens = value;
    }

    public String[] getGrantTokens()
    {
        return grantTokens;
    }

    public void setGrantTokens(String[] value)
    {
         grantTokens = value;
    }

    public void securityCheck() throws SecurityCheckException
    {
        super.securityCheck();

        // Get the accumulated tokens from session.
        HashSet accumulated = (HashSet)
            getSession().getAttribute("AccumulatedTokens");
        // Defensively prepare the accumulated collection
        if(accumulated == null)
            accumulated = new HashSet();

        // Check to see if required tokens are present
        if(requiredTokens.length > 0) {
            // Check for presence of required tokens
            List missingTokens = new ArrayList();
```

```
            for(int i=0; i<requiredTokens.length; i++)
            {
                if(! accumulated.contains(requiredTokens[i]))
                    missingTokens.add(requiredTokens[i]);
            }

            if(missingTokens.size() > 0)
                handleMissingTokens(new MissingTokensEvent(missingTokens));
        }

        // Now add the current grant tokens to the accumulated.
        // Note, as expected, we will not reach this point if the
        // handleMissingTokens throws an Exception.
        for(int i=0; i<grantTokens.length; i++)
        {
                accumulated.add(grantTokens[i]);
        }
        getSession().setAttribute("AccumulatedTokens", accumulated);
    }

    public void handleMissingTokens(MissingTokensEvent e)
        throws SecurityCheckException
    {
        // This default implementation will just trigger conventional
        // Sun ONE Application Framework SecurityCheckException handling
        throw new SecurityCheckException(e.toString());
    }

    private String[] requiredTokens = new String[0];
    private String[] grantTokens = new String[0];

}
```

Create the Extensible Component's Java Template

Extensible components serve as base classes for application defined entities. Therefore, the Sun ONE Application Framework component model provides extensible component authors the opportunity to provide a custom Java template. The IDE toolset will, subsequently, use the component supplied template to create the application specific sub-type. Component authors can utilize the custom template to enhance the application developer's experience. Component authors may prepare the component specific Java template with a set of template tokens defined in com.iplanet.jato.component.ExtensibleComponentInfo. For token details see ExtensibleComponent API.

Component authors may also utilize any arbitrary Java constructs within the Java template (for example, import statements, methods, variables, interface declarations, and so on). Minimally, the custom template will ensure that the new Java class extends from the extensible component class.

In this example we will keep the template utterly minimal.

- Create a new directory mycomponents.resources
- In any text editor create the template mycomponents.resources.SecureViewBean_java.template
- The template contents should look like this. Note the tokens follow a __TOKEN__ pattern.

```
package __PACKAGE__;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.iplanet.jato.*;
import com.iplanet.jato.view.*;
import com.iplanet.jato.view.event.*;
import com.iplanet.jato.model.*;
import mycomponents.*;


/**
 *
 *
 */
public class __CLASS_NAME__ extends SecureViewBean
{
    /**
     * Default constructor
     *
     */
    public __CLASS_NAME__()
    {
        super();
    }


}
```

Create the ComponentInfo Class

The ComponentInfo class defines the design-time metadata that the IDE toolset requires to incorporate the component. In this example we will extend an existing ComponentInfo and in true OO style, simply augment it. We could, of course, choose to implement the ComponentInfo interface from scratch, but that would be unproductive in this case.

Note that in this example we are going beyond the functionality revealed in our earlier component examples. Below, we are going to take advantage of two new metadata opportunities provided by the ExtensibleComponentInfo interface, the opportunity to specify a Java template, and the opportunity to describe event handler methods for the extensible component.

1. **Create the class mycomponents.SecureViewBeanComponentInfo.**
2. **Make SecureViewBeanComponentInfo extend com.iplanet.jato.view.BasicViewBeanComponentInfo**
3. **Implement the no-arg constructor.**
4. **Implement the getComponentDescriptor() method to provide the basic design-time description of the component.**
5. **Implement the getConfigPropertyDescriptors() method to identify which properties you wish to expose in the IDE.**
   - Add an IndexedConfigPropertyDescriptor for the RequiredTokens property.
   - Add an IndexedConfigPropertyDescriptor for the GrantTokens property.
6. **Implement the getPrimaryTemplateAsStream() method to return a Java template file which you wish the IDE toolset to use as the starting point for new classes derived from this extensible component.**
7. **Implement the getEventHandlerDescriptors() method to provide a design-time description of any event handler methods which you wish the IDE toolset to expose for automated insertion into new classes derived from this extensible component.**

   After these steps, mycomponents/SecureViewBeanComponentInfo.java should look like this:

   In this sample code we have embedded String values directly for ease of demonstration. If you anticipate the need to localize your display strings, we encourage you to utilize resource bundles.

```
package mycomponents;
import java.util.*;
import java.io.*;
import com.iplanet.jato.*;
import com.iplanet.jato.component.*;
```

```
import com.iplanet.jato.view.*;


public class SecureViewBeanComponentInfo extends BasicViewBeanComponentInfo
{

    public SecureViewBeanComponentInfo()
    {
        super();
    }

    public ComponentDescriptor getComponentDescriptor()
    {
        final String CLASS_NAME="mycomponents.SecureViewBean";

        ComponentDescriptor descriptor=new ComponentDescriptor(
            CLASS_NAME);
        descriptor.setName("SecurePage");
        descriptor.setDisplayName("Secure ViewBean");
        descriptor.setShortDescription(
            "A Page with a token based security model");
        return descriptor;
    }

    public ConfigPropertyDescriptor[] getConfigPropertyDescriptors()
    {
        if (configPropertyDescriptors!=null)
            return configPropertyDescriptors;

        configPropertyDescriptors=super.getConfigPropertyDescriptors();
        List descriptors=new LinkedList(Arrays.asList(configPropertyDescriptors));

        ConfigPropertyDescriptor descriptor = null;

        descriptor=new IndexedConfigPropertyDescriptor(
            "grantTokens",String.class); // NOI18N
        descriptor.setDisplayName("Grant Tokens"); // NOI18N
        descriptor.setHidden(false);
        descriptor.setExpert(false);
        descriptors.add(descriptor);

        descriptor=new IndexedConfigPropertyDescriptor(
            "requiredTokens",String.class); // NOI18N
        descriptor.setDisplayName("Required Tokens"); // NOI18N
        descriptor.setHidden(false);
        descriptor.setExpert(false);
        descriptors.add(descriptor);

        // Create/return the array
        configPropertyDescriptors = (ConfigPropertyDescriptor[])
            descriptors.toArray(
```

```
                  new ConfigPropertyDescriptor[descriptors.size()]);
         return configPropertyDescriptors;
     }

  public String getPrimaryTemplateEncoding()
     {
/* Production version would be resource bundle driven, like this:
return getResourceString(getClass(),
"PROP_SecureViewBean_SOURCE_TEMPLATE_ENCODING", "ascii");
*/

         return "ascii";
     }

     public InputStream getPrimaryTemplateAsStream()
     {
/* Production version would be resource bundle driven, like this:

return SecureViewBeanComponentInfo.class.getClassLoader().
getResourceAsStream(
getResourceString(getClass(),
"RES_SecureViewBeanComponentInfo_SOURCE_TEMPLATE",""));
*/

         return SecureViewBeanComponentInfo.class.getResourceAsStream(
              "/mycomponents/resources/SecureViewBean_java.template");
     }


     public EventHandlerDescriptor[] getEventHandlerDescriptors()
     {
         if (eventHandlerDescriptors!=null)
              return eventHandlerDescriptors;

         eventHandlerDescriptors=super.getEventHandlerDescriptors();
         List descriptors=new LinkedList(
              Arrays.asList(eventHandlerDescriptors));

         EventHandlerDescriptor descriptor =new EventHandlerDescriptor(
              "handleMissingTokens",
              "handleMissingTokens",
              "public void handleMissingTokens(MissingTokensEvent e)" +
              "throws SecurityCheckException",
              "throw new SecurityCheckException(e.toString());",
              "");

         descriptors.add(descriptor);

         // Create/return the array
         eventHandlerDescriptors = (EventHandlerDescriptor[])
              descriptors.toArray(
```

```
                   new EventHandlerDescriptor[descriptors.size()]);
         return eventHandlerDescriptors;
    }

    private ConfigPropertyDescriptor[] configPropertyDescriptors;
    private EventHandlerDescriptor[] eventHandlerDescriptors;
}
```

Augment the Component Library Manifest

We have already created the component manifest in the earlier example. So now we will simple add additional information. Note that we will add additional types of information not seen in the prior example.

The Sun ONE Application Framework library manifest must be named complib.xml. Within the JAR file, the Sun ONE Application Framework library manifest must be placed in the /COMP-INF directory.

1. **Create/Open the file COMP-INF/complib.xml**
2. **Add an extensible-component element to declare the SecureViewField component.**

   After these steps, the COMP-INF/complib.xml file should look like this. In the interest of clarity, we are only showing the significant delta to the prior version of this file shown earlier.

```
<?xml version="1.0" encoding="UTF-8"?>
<component-library>
<tool-info>
<tool-version>2.1.0</tool-version>
</tool-info>
<library-name>mycomponents</library-name>
<display-name>My First Component Library</display-name>

 ...

    <extensible-component>
         <component-class>mycomponents.SecureViewBean</component-class>
         <component-info-class>mycomponents.SecureViewBeanComponentInfo</component-info-
class>
    </extensible-component>

...

</component-library>
```

Re-create the Component Library JAR File

Once again, Jar up the component classes as we did in the first example so that they can be ready for distribution as a library.

1. **The name of the JAR file is arbitrary. In this case, name it "mycomponents.jar"**
2. **You may omit the Java source files from the JAR**
3. **You should include in the JAR any necessary ancillary resources, like icon images, or resource bundles.**

   In this case we are including several new classes and a Java template file
4. **The mycomponents.jar internal structure should look like this:**

```
mycomponents/resources/SecureViewBean_java.template
mycomponents/MissingTokensEvent.class
mycomponents/MyTextField.class
mycomponents/MyTextFieldComponentInfo.class
mycomponents/SecureViewBean.class
mycomponents/SecureViewBeanComponentInfo.class
mycomponents/TypeValidator.class
mycomponents/ValidatingDisplayField.class
mycomponents/ValidatingTextFieldComponentInfo.class
mycomponents/ValidatingTextFieldTag.class
mycomponents/Validator.class
mycomponents/mycomplib.tld
COMP-INF/complib.xml
```

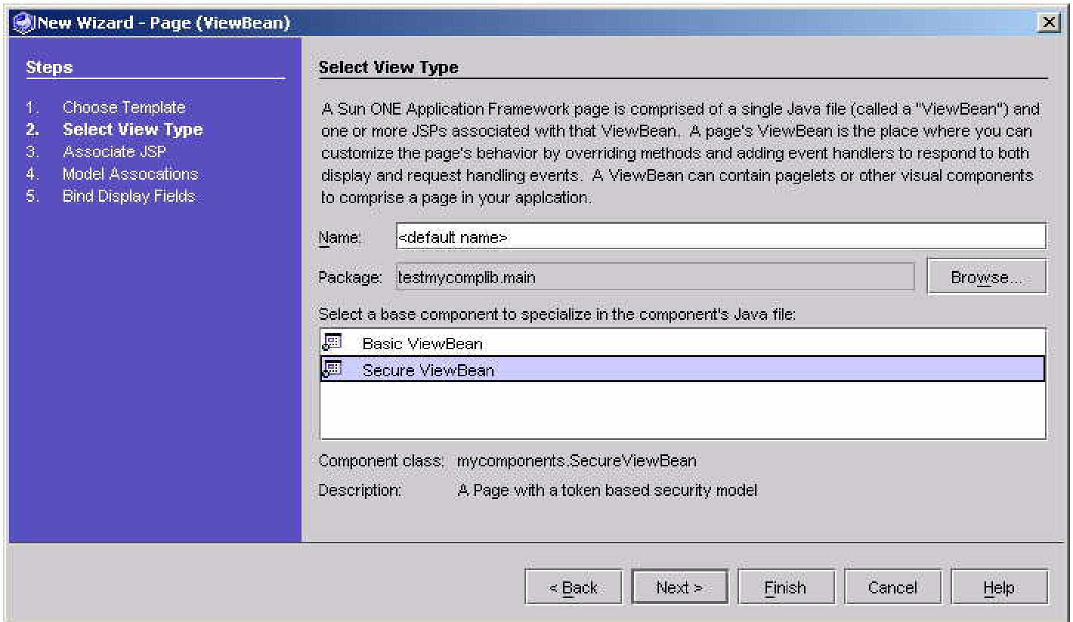
Test the New Component

1. **Deploy the new version of the library into your previously created test application**
   - **Important Sun ONE Studio note:** The Studio will not let you delete or copy over a jar file that is currently mounted. This presents a bit of a challenge when iteratively developing a component library and testing that library in a test application. We recommend the following for repeatedly testing new versions of the same library jar file within a test application.
   - Unmount the test application.
   - After the unmount is complete, go to your operating system file system and copy the new library jar file over the old library jar file in the unmounted test application's `WEB-INF/lib` directory.
   - Remount the test application, the test application should now pick up the new library version.

- Normally, those steps work fine. If you encounter a spurious failure that either prevents you from copying the new jar over the old jar, or failure to remount the test application properly, the fallback strategy is to restart the Studio.

2. **Create a new ViewBean object.**

   The new ViewBean wizard should now look like this:



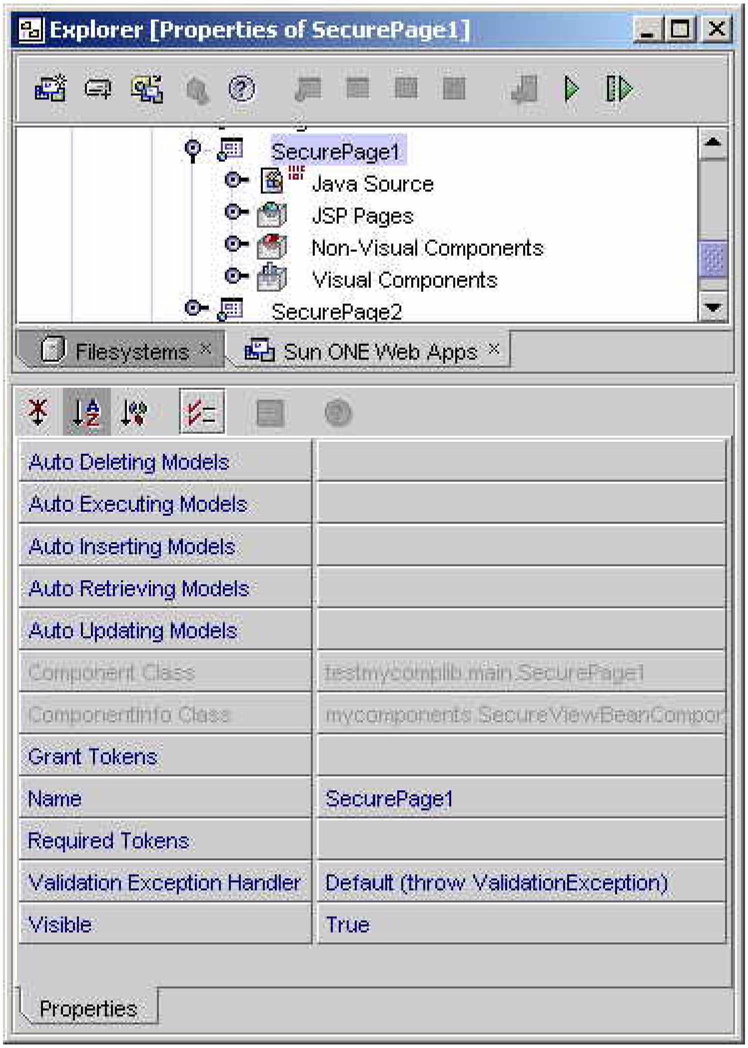
3. **Select the "Secure ViewBean" from the component list and complete the wizard. Take the default settings and let the wizard create SecurePage1 for you. (You may select Finish in the wizard stage above)**

4. **After the wizard completes you can see that the IDE toolset has created a new class based on the component supplied template.**

5. **In order to test our security model, create a second SecureViewBean.**

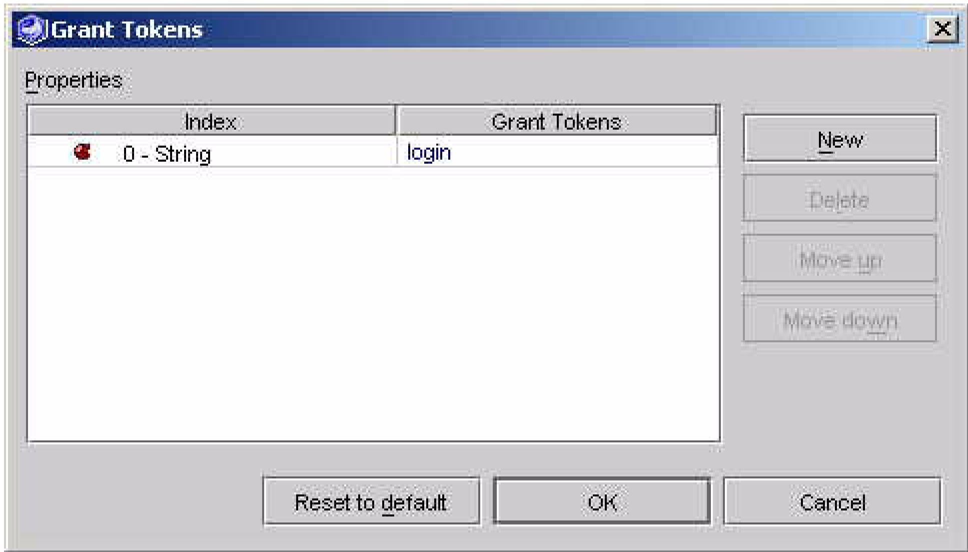
6. **You application should now contain two SecureViewBeans (SecurePage1 and SecurePage2).**

   The new SecureViewBeans contain the Grant Tokens and Required Tokens properties.



7. **Now test the security model by introducing some values into the token properties. Select SecurePage1's Grant Tokens property.**

   If you select the ellipsis in the property sheet it will bring up the indexed String property editor. Add the value "login" to that property. Feel free to add additional tokens.



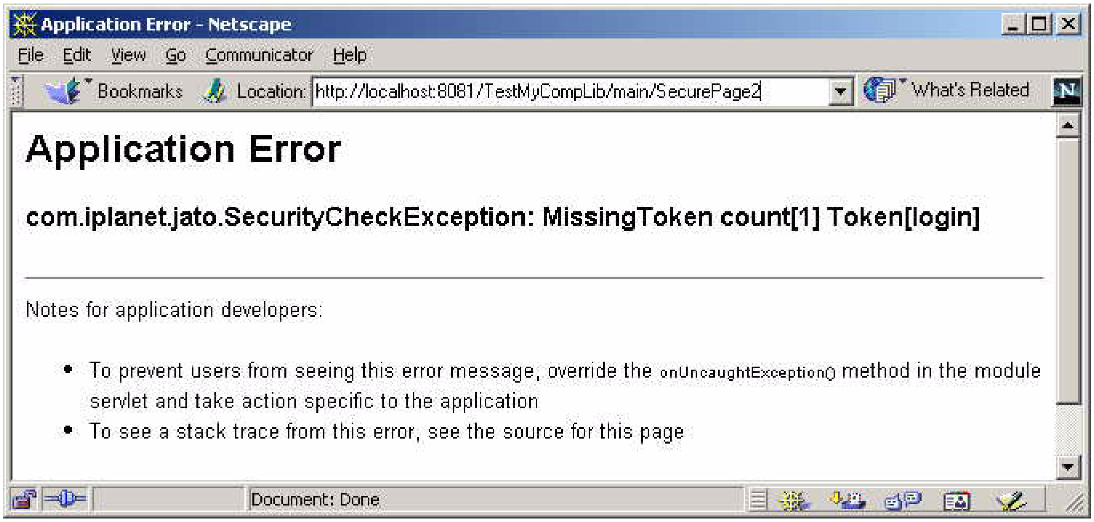
8. **Select the other Secure ViewBean, SecurePage2. Select its Required Tokens property, and add the value "login".**

9. **You have now established a page security rule in your application. SecurePage2 requires the "login" token, and SecurePage1 grants the "login" token. Therefore, an end user who does not visit SecurePage1 BEFORE SecurePage2 should trigger a security exception.**

10. **Add some static content to the SecurePage2's associated JSP, SecurePage2.jsp, since this is currently a blank page. For example, put the text "Welcome to Secure2" into SecurePage2.jsp so you will recognize it in the browser**

11. **Test run SecurePage2.**

    Instead of seeing SecurePage2.jsp's content you should see this message in the browser: If you see this message, it means that the SecureViewBean security model has worked as intended. At least the access prevention has worked.

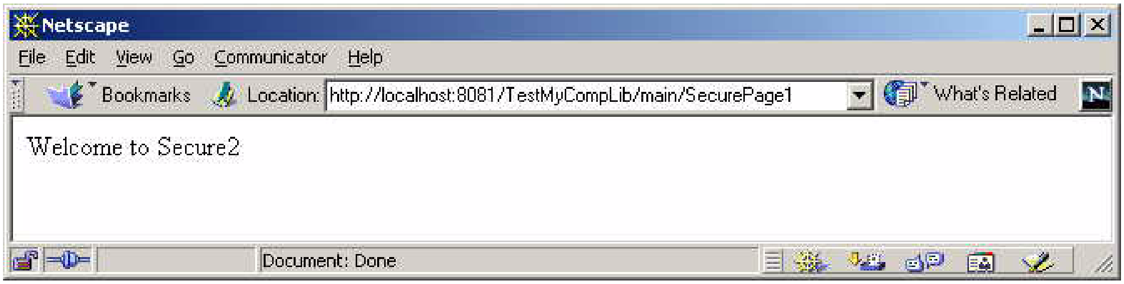


12. **Now, create a link between SecurePage1 and SecurePage2 so that you can test the positive path. There are several ways to do this, feel free to implement your own link. The instructions which follow are just one approach.**
    - Add an instance of the Sun ONE Application Framework Library's Basic Button to SecurePage1. You may either select the component from the Component Palette or select the SecurePage1's Visual Components sub-node, right mouse and select the Add Visual Component... action from the pop-up menu.
    - This will add a "button1" child to your test ViewBean
    - Now, select the button1 visual component node.
    - Right mouse and select the pop up menu's Events->handleRequest action.
    - This will add an event handler method named handleButton1Request to your SecurePage1's Java file.
    - Now rework the body of the handleButton1Request to look like this

```
public void handleButton1Request(RequestInvocationEvent event) throws Exception {
getViewBean(SecurePage2.class).forwardTo(getRequestContext());
}
```

13. **Test run the page flow from SecurePage1 to SecurePage2**
    - Test run SecurePage1
    - Secure1 should appear in the browser as a blank page with a single button labeled "Submit". (the user should now have been granted the "login" token.
    - Press the Submit button. This will trigger the handleButton1Request logic which will forward the request to SecurePage2.

- The contents of SecurePage2.jsp should show up in the browser. (because the user had accumulated the required tokens).



Ship It? Not yet, first test the EventHandlerDescriptor feature (handleMissingTokens)

Recall that the SecureViewBeanComponentInfo declares an EventHandlerDescriptor which described an event handler called handleMissingTokens. Now you need to test this feature.

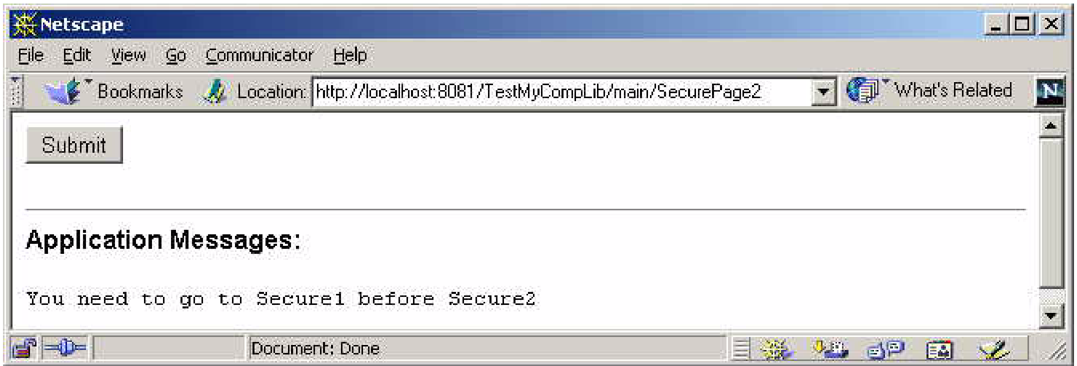
1. **Select the SecurePage2 node.**
2. **Right mouse and select the pop up menu's Events->handleMissingTokens option. This should insert the handleMissingTokens method skeleton into SecurePage2.java and automatically position the Java editor at that method.**
3. **Edit that method to automatically route users back to SecurePage1 when this event is triggered.**

   This is just an arbitrary means of testing the event handler. Application developers can implement this handler any way they want.

```
public void handleMissingTokens(MissingTokensEvent e)throws SecurityCheckException {
        // Route invalid access users to SecurePage1
        appMessage("You need to go to Secure1 before Secure2");
        getViewBean(SecurePage1.class).forwardTo(getRequestContext());
        // Stop further processing of the original request.
        throw new CompleteRequestException();
}
```

4. **Test run SecurePage2 again.**

   This time the browser should return SecurePage1 because the event handler took control.


Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost:8081/TestMyCompLib/main/SecurePage2
What's Related
Submit
Application Messages:
You need to go to Secure1 before Secure2
Document: Done

CHAPTER 4

Developing Model Components

This section assumes that you have already read Develop Your First Component found in Chapter 2, "Developing Components" on page 17.

Model Components

The obvious Model components are the extensible Model components. Extensible Model components are custom implementations of the Model class which are intended for specialization by application developers. The specialization by application developers will usually consist of application developers adding schema information to their application specific Models. The Sun ONE Application Framework Component Library contains a number of extensible Model components, such as QueryModelBase, WebServiceModel, SessionModel, ObjectAdapterModel and CustomModel.

ModelComponentInfo

The ModelComponentInfo interface allows component authors to define additional metadata that is applicable to all Model components.

ExecutingModelComponentInfo

The ExecutingModelComponentInfo interface allows component authors to define additional metadata that is applicable to all Model components whose component class implements the com.iplanet.jato.model.ExecutingModel interface.

Is it possible to create a non-extensible Model component? The answer is yes. In fact, whenever an application developer creates a new Model via the Model wizard, they are in fact extending an extensible Model component and creating an application-specific Model. This new application specific Model is by definition a non-extensible component. Whenever an application developer attempts to fill out a property of type ModelReference, the IDE toolset will invoke a component browser that allows the application developer to choose from a set of existing non-extensible Models. For instance when an application developer specifies a DisplayField's Model Reference property, the IDE toolset presents them with a browser that allows them to select a Model.

Is it possible to create one of these non-extensible Models and add it to a library so that it can be distributed? Again, the answer is yes. See the section Developing and Distributing Non-Extensible Model, Command and ContainerView Components below.

Developing a Non-Extensible Model Component

A non-extensible Model component is a concrete Model that has been created within the IDE from an extensible Model component. It is no different from an application specific Model, except that is distributed in a jar file and can be incorporated into multiple applications. The distribution technique is common for non-extensible Models, ContainerViews and Commands. See the section Developing and Distributing Non-Extensible Model, Command and ContainerView Components

Developing an Extensible Model Component

In this section we shall describe how to create a new extensible Model component that acts as an adapter to an arbitrary XML document. The adapter pattern is one of the patterns which Sun ONE Application Framework Models are well suited to implement. In this example, the Model component will allow Sun ONE Application Framework Views to access arbitrary XML document data in an Sun ONE Application Framework consistent way. View developers will not need to know anything about the XML internals, or any XML specific APIs. Instead the View developers will interact with the XML document Model as they would any other Sun ONE Application Framework Model. This hightlights one of the key aspects of Sun ONE Application Framework Model design. Sun ONE Application Framework

Models are intended primarily to serve as application resources which are used by Views. For more on the relationship between Sun ONE Application Framework Views and Models see *Sun ONE Application Framework Developer's Guide*.

Designing a new extensible Model is generally a non-trivial undertaking. The following example is sophisticated, yet concise enough for this guide. As with any Model, alternative designs are possible. As with any example, further refinement is encouraged for a production quality version. Remember, the objective of this section, is to familiarize yourself with the mechanics of Sun ONE Application Framework extensible Model implementation.

This example will allow us to introduce several additional Sun ONE Application Framework component model topics

- ExtensibleModelComponentInfo
- ModelFieldGroupDescriptor
- ModelFieldDescriptor

Key XML Document Model Design points

1. **This Model will not be a business delegate. Some models ARE both adapters and business delegates (e.g. the Sun ONE Application Framework standard component library's JDBC SQL Query Model is both an adapter and a business delegate because it is responsible for communicating with the enterprise tier). The XML Document Model will not be responsible for the lifecycle of the XML Document. It will assume that the application has managed to acquire the XML document. The Model does not care how the application acquires the XML document. The Model will rely on the application to place the XML document within a well defined location. The Model will access the XML document from that location, as needed. There are several benefits to this design decision beyond just making the Model's job simpler. For one, this approach will allow > 1 XML Document Model access to the same XML Document. During the testing of the component we shall see how this Document-Model cardinality will benefit application developers. Another benefit is that it allows application developers to seamlessly leverage non-Sun ONE Application Framework infrastructure code that they may already have written to manage the document lifecycle.**

2. **This Model will limit its ambition to serving as a read-only Model. This means that the Model will support retrieval and display of XML document data, but it will not facilitate modification of document data. The implementation of full XML document update support is beyond the scope of this document. Furthermore, it is perfectly justifiable for a Model to limit its ambition to a well defined feature set, as long as the Model documentation makes it clear what is and is not supported. Application developers will then limit the use of the Model according to its documented usage.**

Our XML Document Model component should support the following design-time functionality:

- Each XML Document Model will expose a property called "Document Scope". Application developers will configure this property to specify one of three standard servlet container scopes, request scope, session scope or application scope. By setting this property the application developer commits to placing the XML document in the specified scope at run-time. The Model will then fetch the document from the specified scope at run-time. The default value for this property will be request scope.
- Each XML Document Model will expose a property called "Document Scope Attribute Name". This is a companion property to "Document Scope". Application developers will configure this property to specify a scoped attribute name. By setting this property the application developer commits to placing the XML document in the specified scope and attribute name.
- Each XML Document Model will expose a property called "Base Dataset Name". Application developers will configure this property to specify an offset into the XML Document. The dataset name will be specified as an XPath expression. Within a given Model all ModelField specific XPath expressions will be relative to the "Base Dataset Name". This property may be left blank, in which case, Model Field specific XPath expressions will be assumed to be absolute.
- Each XML Document Model will allow application developers to add an arbitrary number of Model Fields to the Model at design-time.
  - Application developers can configure each Model Field to have an arbitrary field name.
  - Application developers can configure each Model Field to access a value within the XML Document. This access must be configured as an XPath expression (either relative to the "Base Dataset Name", or absolute in the absence of any "Base Dataset Name").
  - View developers will be able to bind to the Model Fields in the Sun ONE Application Framework conventional manner.

Our XML Document Model component should support the following run-time functionality:

- The XML Document Model will defensively access the XML Document by retrieving it from the named attribute within the specified scope.
- The XML Document Model will implement the key com.iplanet.jato.model.Model method "getValue(String fieldName)" to resolve a field name to an XPath expression, and an XPath expression to a value within the document.
- The XML Document Model will implement the com.iplanet.jato.model.DatasetModel interface. All DatasetModels provide consistent access to multiple discreet sets of data. In this case, a dataset would be a section of the XML Document for which an XPath expression would return > 1 nodes. The implementation of the DatasetModel interface will allow application

developers to use the DatasetModel API to iterate across the multiple values within the dataset. The conventional, but not only, means for achieving this is to associate a TiledView with a DatasetModel.

Please note, the implementation shown below will take shortcuts in the interest of brevity. The sample code contains some comments which point out areas where run time optimizations are possible but would require more complex code beyond the scope of this exercise.

In order to meet these requirements, we will design and implement the following classes:

- Component class - mycomponents.XMLDocumentModel\
- ComponentInfo class - mycomponents.XMLDocumentModelComponentInfo
- A ModelFieldDescriptor class - mycomponents.XMLDocumentModelFieldDescriptor

Additionally, we will implement a custom Java template which the IDE toolset will use as the basis for application specific sub-types of our XMLDocumentModel.

Finally we shall edit the mycomponents complib.xml to add the new component to the Sun ONE Application Framework component library.

Create the ModelFieldDescriptor class

The Sun ONE Application Framework component model provides extensible Model component authors with the opportunity to specify an arbitrary implementation of the com.iplanet.jato.model.ModelFieldDescriptor interface. This is a very minimal interface. Each implementation of ModelFieldDescriptor must also be a JavaBean. Model component authors should design a ModelFieldDescriptor as a bean that can be configured by application developers in order to define a model field at design-time. Component authors, thus have tremendous freedom to design model fields which can expose all the design-time configuration opportunity they want, as long as it can be expressed as a JavaBean.

In our example, our model field design-time configuration needs are trivial. We need to allow the application developer to configure each model field with an XPath expression.

1. **In any Java editor create the class mycomponents.XMLDocumentModelFieldDescriptor**
2. **Implement the basic com.iplanet.jato.model.ModelFieldDescriptor interface**
3. **Add a get and set method for the property XPath.**

4. **Add a get and set method for the property FieldClass. This is an optional property. If populated, at run-time the Model will coerce the raw value retrieved with the XPath expression into the type specified by the FieldClass property.**

   After these steps, mycomponents/XMLDocumentModelFieldDescriptor.java should look like this

```
package mycomponents;

import java.io.*;
import java.util.*;
import com.iplanet.jato.model.*;

/**
 *
 *
 *
 */
public class XMLDocumentModelFieldDescriptor extends Object
     implements ModelFieldDescriptor, Serializable
{

     public XMLDocumentModelFieldDescriptor()
     {
          super();
     }

     public String getName()
     {
          return name;
     }

     public void setName(String name)
     {
          this.name = name;
     }

     public String getXPath()
     {
          return xpath;
     }

     public void setXPath(String xpath)
     {
          this.xpath = xpath;
     }

     public Class getFieldClass()
     {
          return fieldClass;
```

```
    }

    public void setFieldClass(Class fieldClass)
    {
        this.fieldClass = fieldClass;
    }

    private String xpath;
    private String name;
    private Class fieldClass; // DO NOT change this init to a default value

}
```

Create the Sun ONE Application Framework Component Class

1. **In any Java editor create the class mycomponents.XMLDocumentModel**
2. **Make XMLDocumentModel extend com.iplanet.jato.view.DatasetModelBase**
3. **Make XMLDocumentModel implement com.iplanet.jato.view.MultiDatasetModel**
4. **Implement the appropriate constructor for the component type. All Model components must implement a no-arg constructor.**
5. **Add a get and set method for the property named "DocumentScope"**
6. **Add a get and set method for the property named "DocumentScopeAttributeName"**
7. **Add a get and set method for the property named "CurrentDatasetName". Note we shall give this property a more user friendly display name "Base Dataset Path", but that work will be done in the XMLDocumentModelComponentInfo.**
8. **Implement the remaining methods that are required to fulfill our component specific requirements.**
   - Implement the methods which DatasetModelBase left abstract.
   - Implement the methods required by MultiDatasetModel interface.
   - Implement any helper methods which are needed to fulfill the XML Document adaptation.

After these steps, mycomponents/XMLDocumentModel.java should look like this

```
package mycomponents;
import java.util.*;
import com.iplanet.jato.*;
import com.iplanet.jato.model.*;
```

```
import com.iplanet.jato.model.custom.*;
import com.iplanet.jato.util.*;
import org.w3c.dom.*;
import org.w3c.dom.traversal.*;
import org.apache.xpath.XPathAPI;
import javax.xml.transform.*;
import javax.servlet.jsp.PageContext;


/**
 *
 * @author   component-author
 */
public class XMLDocumentModel extends DatasetModelBase implements MultiDatasetModel
{

     public XMLDocumentModel()
     {
          super();
     }


     ////////////////////////////////////////////////////////////////////////////
     // Properties
     ////////////////////////////////////////////////////////////////////////////

     public String getCurrentDatasetName()
     {
          // Add some defensive logic to ensure a valid currentDatasetName
          if(currentDatasetName == null || currentDatasetName.trim().equals(""))
               currentDatasetName = "/";
          return currentDatasetName;
     }

     public void setCurrentDatasetName(String datasetName)
     {
          this.currentDatasetName = datasetName;
     }

     public int getDocumentScope()
     {
          return documentScope;
     }


     public void setDocumentScope(int documentScope)
     {
          this.documentScope = documentScope;
     }

     public String getDocumentScopeAttributeName()
```

```
    {
        return documentScopeAttr;
    }


    public void setDocumentScopeAttributeName(String name)
    {
        this.documentScopeAttr = name;
    }

    public void setDocument(Document value)
    {
        doc = value;
    }

    public Document getDocument()
    {
        if(doc == null) {
            // Use the scope and attribute name to find the document
            // The assumption is that the application logic has placed doc
            // in the appropriate scope.
            RequestContext rc = RequestManager.getRequestContext();
            String attr = getDocumentScopeAttributeName();
            switch (getDocumentScope())
            {
                case PageContext.REQUEST_SCOPE:
                    doc = (Document)
                    rc.getRequest().getAttribute(attr);
                    break;
                case PageContext.APPLICATION_SCOPE:
                    doc = (Document)
                    rc.getServletContext().getAttribute(attr);
                    break;
                case PageContext.SESSION_SCOPE:
                    doc = (Document)
                    rc.getRequest().getSession().getAttribute(attr);
                    break;
                default:
                    throw new IllegalArgumentException(
                    "DocumentScope is set to an invalid value " +
                        getDocumentScope());
            }

            if(DEBUG)
                System.out.println("XMLDocumentModel.getModel doc is " +
                (doc==null?"null":"not null"));

        }
        return doc;
    }
```

```
    ////////////////////////////////////////////////////////////////////////////
    // Model Interface Methods
    ////////////////////////////////////////////////////////////////////////////

    public Object getValue(String name)
    {
        Node node=null;
        try
        {
            node=getNode(name);
        }
        catch (Exception e)
        {
            throw new ModelValueException("Exception getting value for "+
                "field \""+name+"\"",e);
        }

        if (node==null)
            return null;

        Object result=null;
        if (isTextNode(node) || isAttributeNode(node))
        {
            result=node.getNodeValue();

            XMLDocumentModelFieldDescriptor descriptor=(XMLDocumentModelFieldDescriptor)
                getFieldGroup().getFieldDescriptor(name);
            if (descriptor.getFieldClass()!=null)
                result=TypeConverter.asType(descriptor.getFieldClass(),result);
        }
        else
        {
            // Return the node as is and let the caller figure out what to
            // do with it--this could've been what they actually wanted
            result=node;
        }

        return result;
    }


    public Object[] getValues(String name)
    {
        NodeList nodes=null;
        try
        {
            nodes=getNodes(name);
        }
        catch (Exception e)
        {
            throw new ModelValueException("Exception getting values for "+
```

```
                "field \""+name+"\"",e);
        }

        if (nodes==null)
            return new Object[0];

        Object[] result=null;
        try
        {
            List resultList=new LinkedList();
            for (int i=0; i<nodes.getLength(); i++)
            {
                Node node=nodes.item(i);
                if (isTextNode(node) || isAttributeNode(node))
                {
                    Object data=node.getNodeValue();

                    XMLDocumentModelFieldDescriptor descriptor=
(XMLDocumentModelFieldDescriptor)
                        getFieldGroup().getFieldDescriptor(name);
                    if (descriptor.getFieldClass()!=null)
                    {
                        data=TypeConverter.asType(descriptor.getFieldClass(),
                            data);
                    }

                    resultList.add(data);
                }
                else
                {
                    // Return the node as is and let the caller figure out what
                    // to do with it--this could've been what they actually
                    // wanted
                    resultList.add(node);
                }
            }

            result=resultList.toArray();
        }
        catch (Exception e)
        {
            throw new ModelValueException("Exception getting values "+
                "for field \""+name+"\"",e);
        }

        return result;
    }



    public void setValue(String name, Object value)
```

```
    {
        // Ignore
    }

    public void setValues(String name, Object[] value)
    {
        // Ignore
    }

    ////////////////////////////////////////////////////////////////////////////
    // DatasetModel Interface Methods
    ////////////////////////////////////////////////////////////////////////////

    protected NodeList getCurrentDatasetNodeList()
        throws ModelControlException
    {
        if (nodeList!=null)
            return nodeList;

        if (getDocument()==null)
        {
            throw new ModelControlException(
                "No XML document has been provided");
        }

        try
        {
            // Note: instead of XPathAPI, we can use CachedXPathAPI to improve
            // the efficiency of this call.  This requires some additional
            // complexity not useful in this example, however.
            // Also, we could potentially move away from use Apache-specific
            // code by using the org.w3c.dom.xpath package, as long as the
            // XML parser supported DOM Level 3.
            nodeList=XPathAPI.selectNodeList(getDocument(),
                getCurrentDatasetName());
        }
        catch (TransformerException e)
        {
            throw new ModelControlException("Exception getting NodeList for "+
                "dataset \""+getCurrentDatasetName()+"\"");
        }

        return nodeList;
    }


    public int getLocationOffset()
    {
        return 0;
    }
```

```
public int getLocation()
     throws ModelControlException
{
     Integer index=(Integer)datasetContexts.get(getCurrentDatasetName());
     if (index==null)
     {
          // Call just to check for NodeList validity
          getCurrentDatasetNodeList();
          return -1;
     }

     return index.intValue();
}


public void setLocation(int value)
     throws ModelControlException
{
     int maxLength=getCurrentDatasetNodeList().getLength();
     if (value>=maxLength || value<-1)
     {
          throw new ModelControlException("Location index out of "+
               "range (max value = "+(maxLength-1)+")");
     }

     datasetContexts.put(getCurrentDatasetName(),new Integer(value));
}


public int getSize()
     throws ModelControlException
{
     return getCurrentDatasetNodeList().getLength();
}


public void setSize(int value)
     throws ModelControlException
{
     throw new ModelControlException("Unsupported operation; "+
          "model size cannot be set");
}

protected boolean ensureValidDataPosition()
     throws ModelControlException
{
     if (getSize()==0)
          return false; // No data to retrieve
     else
     if (getLocation()==-1)
```

```
        {
            // If we're currently before the first item, we need to move
            // to the first item to retrieve some data
            if (!first())
                throw new ModelControlException("Could not move to first item");
        }

        return true;
    }

    ////////////////////////////////////////////////////////
    // XML Node methods
    //////////////////////////////////////////////////////////////////////////

    public Node getNode(String fieldName)
        throws ModelControlException, TransformerException
    {
        if (!ensureValidDataPosition())
            return null;

        Node contextNode=getCurrentDatasetNodeList().item(getLocation());

        // Note: instead of XPathAPI, we can use CachedXPathAPI to improve
        // the efficiency of this call.  This requires some additional
        // complexity not useful in this example, however.
        // Also, we could potentially move away from use Apache-specific
        // code by using the org.w3c.dom.xpath package, as long as the
        // XML parser supported DOM Level 3.
        Node n = XPathAPI.selectSingleNode(contextNode,getFieldXPath(fieldName));
        if(DEBUG) {
            if(n == null)
                System.out.println("Warning: getNode found no node at[" +
                    getFieldXPath(fieldName) + "]");
        }
        return n;
    }

    public NodeList getNodes(String fieldName)
        throws ModelControlException, TransformerException
    {
        if (!ensureValidDataPosition())
            return null;

        Node contextNode=getCurrentDatasetNodeList().item(getLocation());

        // Note: instead of XPathAPI, we can use CachedXPathAPI to improve
        // the efficiency of this call.  This requires some additional
        // complexity not useful in this example, however.
        // Also, we could potentially move away from use Apache-specific
        // code by using the org.w3c.dom.xpath package, as long as the
        // XML parser supported DOM Level 3.
```

```
        NodeList nl = XPathAPI.selectNodeList(contextNode,getFieldXPath(fieldName));
        if(DEBUG) {
            if(nl == null)
                System.out.println("Warning: getNodes found no nodes at[" +
                    getFieldXPath(fieldName) + "]");
        }
        return nl;
    }


    public static boolean isTextNode(Node node)
    {
        if (node==null)
            return false;
        return (node instanceof CharacterData);
    }

    public static boolean isAttributeNode(Node node)
    {
        if (node==null)
            return false;
        return node.getNodeType()==Node.ATTRIBUTE_NODE;
    }

    ////////////////////////////////////////////////////////////////////////////
    // Helper method
    ////////////////////////////////////////////////////////////////////////////
    public String getFieldXPath(String fieldName)
    {
        XMLDocumentModelFieldDescriptor descriptor=(XMLDocumentModelFieldDescriptor)
            getFieldGroup().getFieldDescriptor(fieldName);
        return descriptor.getXPath();
    }
    ////////////////////////////////////////////////////////////////////////////
    // Instance variables
    ////////////////////////////////////////////////////////////////////////////

    private int documentScope = PageContext.REQUEST_SCOPE; // request scope by default
    private String documentScopeAttr = "testDoc";
    private String currentDatasetName;
    private Document doc;

    private NodeList nodeList;
    private Map datasetContexts=new HashMap();
    private String datasetName;

    private static final boolean DEBUG = true;
}
```

Create the Extensible Component's Java template

Extensible components serve as base classes for application defined entities. Therefore, the Sun ONE Application Framework component model provides extensible component authors the opportunity to provide a custom Java template. The IDE toolset will, subsequently, use the component supplied template to create the application specific sub-type. Component authors can utilize the custom template to enhance the application developer's experience. Component authors may prepare the component specific Java template with a set of template tokens defined in com.iplanet.jato.component.ExtensibleComponentInfo. For token details see ExtensibleComponent API.

Component authors may also utilize any arbitrary Java constructs within the Java template (for example, import statements, methods, variables, interface declarations, and so on). Minimally, the custom template will ensure that the new Java class extends from the extensible component class. Component authors may also use the template as a means of communicating to the developer documentation inline in the source so as to provide "recommended steps" or conditions or boundaries to keep in mind while specializing.

In this example we will keep the template utterly minimal.

- In any text editor create the template mycomponents.resources.XMLDocumentModel_java.template
- The template contents should look like this. Note the tokens follow a __TOKEN__ pattern.

```
package __PACKAGE__;

import java.io.*;

import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.iplanet.jato.*;
import com.iplanet.jato.model.*;
import com.iplanet.jato.util.*;
import mycomponents.*;


/**
 *
 *
 * @author
 */
public class __CLASS_NAME__ extends XMLDocumentModel
{
     /**
      * Default constructor
```

```
     *
     */
    public __CLASS_NAME__()
    {
        super();
    }

}
```

Create the ComponentInfo Class

The ComponentInfo class defines the design-time metadata that the IDE toolset requires to incorporate the component. In this example we will extend an existing ComponentInfo and in true OO style, simply augment it. We could, of course, choose to implement the ComponentInfo interface from scratch, but that would be unproductive in this case.

Note that in this example we are going beyond the functionality revealed in our earlier component examples. Below, we are going to take advantage of a new metadata opportunity provided by the ExtensibleModelComponentInfo interface, the opportunity to describe an arbitrary Model Field type.

1. **Create the class mycomponents.XMLDocumentModelComponentInfo.**

2. **Make XMLDocumentModelComponentInfo extend com.iplanet.jato.model.ExtensibleModelComponentInfo**

3. **Implement the no-arg constructor.**

4. **Implement the getComponentDescriptor() method to provide the basic design-time description of the component.**

5. **Implement the getConfigPropertyDescriptors() method to identify which properties you wish to expose in the IDE. Note the use of default values within the ConfigPropertyDescriptor declarations.**
   - Add a ConfigPropertyDescriptor for the DocumentScope property.
   - Add a ConfigPropertyDescriptor for the DocumentScopeAttributeName property.
   - Add a ConfigPropertyDescriptor for the CurrentDatasetName property.

6. **Implement the getPrimaryTemplateAsStream() method to return a Java template file which you wish the IDE toolset to use as the starting point for new classes derived from this extensible component.**

7. **Implement the getModelFieldGroupDescriptors() method to provide a design-time description of the model field type required by the Model.**

Do not get confused by the extra level of indirectness suggested by ModelFieldGroupDescriptor on top of ModelFieldDescriptor. The ModelFieldDescriptor is the vital feature we want you to focus on. The ModelFieldGroupDescriptor is an advanced optional feature. Suffice to say that most Sun ONE Application Framework Model components can simply make use of the standard com.iplanet.jato.model.ModelFieldGroup.

After these steps, mycomponents/XMLDocumentModelComponentInfo.java should look like this:

In this sample code we have embedded String values directly for ease of demonstration. If you anticipate the need to localize your display strings, we encourage you to utilize resource bundles.

```
package mycomponents;

import java.util.*;
import java.io.*;
import com.iplanet.jato.component.*;
import com.iplanet.jato.model.*;

/**
 *
 *
 */
public class XMLDocumentModelComponentInfo extends ExtensibleModelComponentInfo
{

     public XMLDocumentModelComponentInfo()
     {
          super();
     }

     public ComponentDescriptor getComponentDescriptor()
     {
          // identify the component class
          ComponentDescriptor result=new ComponentDescriptor(
               "mycomponents.XMLDocumentModel");

          // The name will be used to determine a name for the component instance
          result.setName("XMLDocumentModel");

          // The display name will be used to show the component in a chooser
          result.setDisplayName("XML Document Model");

          // The description will be the tool tip text for the component
          result.setShortDescription("A simple demonstration of a new model component");

          return result;
```

```
    }


    public String getPrimaryTemplateEncoding()
    {
/* Production version would be resource bundle driven, like this:
return getResourceString(getClass(),
"PROP_XMLDocumentModel_SOURCE_TEMPLATE_ENCODING", "ascii");
*/

        return "ascii";
    }

    public InputStream getPrimaryTemplateAsStream()
    {
/* Production version would be resource bundle driven, like this:
return XMLDocumentModelComponentInfo.class.getClassLoader().
getResourceAsStream(
getResourceString(getClass(),
"RES_XMLDocumentModelComponentInfo_SOURCE_TEMPLATE",""));
*/

        return XMLDocumentModelComponentInfo.class.getResourceAsStream(
            "/mycomponents/resources/XMLDocumentModel_java.template");
    }

    public ConfigPropertyDescriptor[] getConfigPropertyDescriptors()
    {
        if (configPropertyDescriptors!=null)
            return configPropertyDescriptors;

        configPropertyDescriptors=super.getConfigPropertyDescriptors();
        List descriptors=new LinkedList(Arrays.asList(configPropertyDescriptors));

        ConfigPropertyDescriptor descriptor = null;

        descriptor=new ConfigPropertyDescriptor(
            "documentScope",Integer.TYPE);
        descriptor.setDisplayName("Document Scope");
        descriptor.setHidden(false);
        descriptor.setExpert(false);
        descriptor.setDefaultValue(new Integer(
            javax.servlet.jsp.PageContext.REQUEST_SCOPE));
        descriptors.add(descriptor);

        descriptor=new ConfigPropertyDescriptor(
            "documentScopeAttributeName",String.class);
        descriptor.setDisplayName("Document Scope Attribute Name");
        descriptor.setHidden(false);
        descriptor.setExpert(false);
        descriptor.setDefaultValue("");
```

```
        descriptors.add(descriptor);

        descriptor=new ConfigPropertyDescriptor(
            "currentDatasetName",String.class);
        descriptor.setDisplayName("Base Dataset Path");
        descriptor.setHidden(false);
        descriptor.setExpert(false);
        descriptor.setDefaultValue("");
        descriptors.add(descriptor);

        // Create/return the array
        configPropertyDescriptors = (ConfigPropertyDescriptor[])
            descriptors.toArray(
                new ConfigPropertyDescriptor[descriptors.size()]);
        return configPropertyDescriptors;
    }

    public ModelFieldGroupDescriptor[] getModelFieldGroupDescriptors()
    {
        if(null != modelFieldGroupDescriptors)
            return modelFieldGroupDescriptors;

        List descriptors=new ArrayList();
        ModelFieldGroupDescriptor descriptor=null;

        descriptor = new ModelFieldGroupDescriptor(
            "Fields",
            ModelFieldGroup.class,
            new ConfigPropertyDescriptor[0],
            XMLDocumentModelFieldDescriptor.class,
            "addFieldDescriptor",
            "setFieldGroup");

        descriptor.setFieldBaseName("field");
        descriptor.setFieldTypeDisplayName("Field");
        descriptor.setGroupDisplayName("Fields");
        descriptor.setFieldPropertyEditorClass(null);
        descriptors.add(descriptor);

        modelFieldGroupDescriptors = (ModelFieldGroupDescriptor[])
            descriptors.toArray(
                new ModelFieldGroupDescriptor[descriptors.size()]);
        return modelFieldGroupDescriptors;
    }

    private ModelFieldGroupDescriptor[] modelFieldGroupDescriptors;
    private ConfigPropertyDescriptor[] configPropertyDescriptors;

}
```

Augment the Component Library Manifest

We have already created the component manifest in the earlier example. So now we will simple add additional information. Note that we will add additional types of information not seen in the prior example.

The Sun ONE Application Framework library manifest must be named complib.xml. Within the JAR file, the Sun ONE Application Framework library manifest must be placed in the /COMP-INF directory.

1. **Create/Open the file COMP-INF/complib.xml**
2. **Add an extensible component element to declare the XMLDocumentModel component.**

   After these steps, the COMP-INF/complib.xml file should look like this. In the interest of clarity, we are only showing the significant delta to the prior version of this file shown earlier.

```
<?xml version="1.0" encoding="UTF-8"?>
<component-library>
<tool-info>
<tool-version>2.1.0</tool-version>
</tool-info>
<library-name>mycomponents</library-name>
<display-name>My First Component Library</display-name>

 ...

     <extensible-component>
          <component-class>mycomponents.XMLDocumentModel</component-class>
         <component-info-class>mycomponents.XMLDocumentModelComponentInfo</component-info-
class>
     </extensible-component>

...

</component-library>
```

Re-create the Component Library JAR File

Once again Jar up the component classes as we did in the first example so that they can be ready for distribution as a library.

1. **The name of the JAR file is arbitrary. In this case, name it "mycomponents.jar"**
2. **You may omit the Java source files from the JAR**

3. **You should include in the JAR any necessary ancillary resources, like icon images, or resource bundles. In this case there are none.**

   In this case we are now including several new classes and a Java template file

4. **The mycomponents.jar internal structure should look like this:**

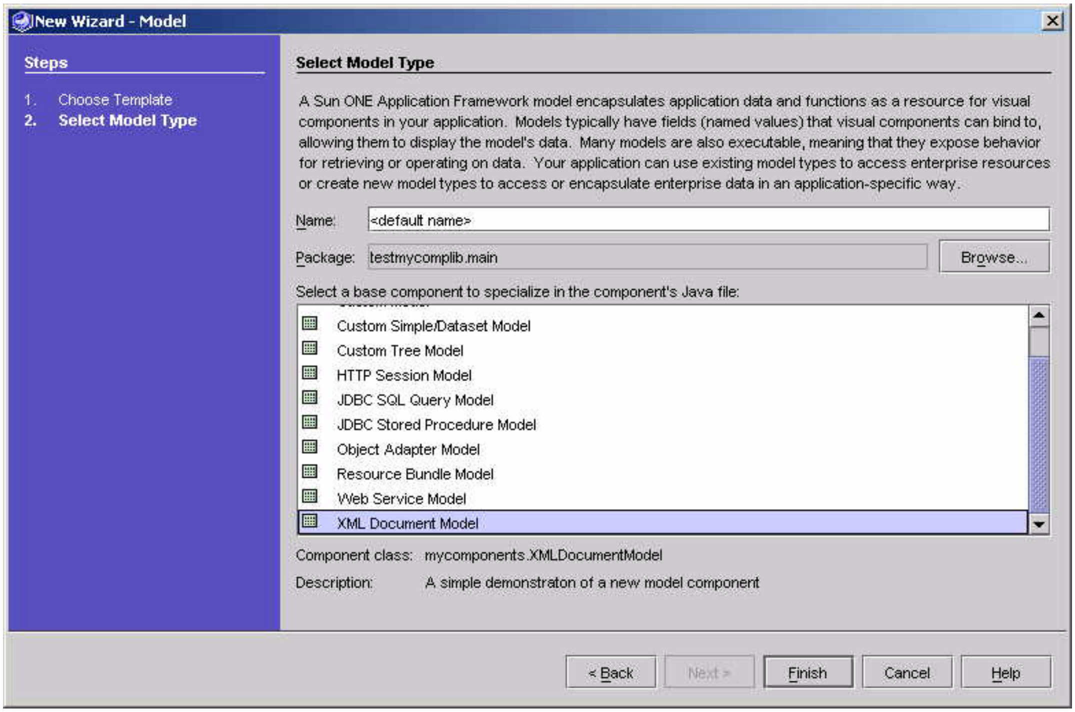
```
mycomponents/resources/SecureViewBean_java.template
mycomponents/resources/XMLDocumentModel_java.template
mycomponents/MissingTokensEvent.class
mycomponents/MyTextField.class
mycomponents/MyTextFieldComponentInfo.class
mycomponents/SecureViewBean.class
mycomponents/SecureViewBeanComponentInfo.class
mycomponents/TypeValidator.class
mycomponents/ValidatingDisplayField.class
mycomponents/ValidatingTextFieldComponentInfo.class
mycomponents/ValidatingTextFieldTag.class
mycomponents/Validator.class
mycomponents/XMLDocumentModel.class
mycomponents/XMLDocumentModelComponentInfo.class
mycomponents/XMLDocumentModelFieldDescriptor.class
mycomponents/mycomplib.tld
COMP-INF/complib.xml
```

Test the New Component

1. **Deploy the new version of the library into your previously created test application.**

   Important Sun ONE Studio note - the Studio will not let you delete or copy over a JAR file that is currently mounted unless this is done via the IDE using ANT tasks which share the same VM as the IDE and share the file locks. Actually, we recommend shutting down the Sun ONE Studio whenever you need to replace one of the JAR files that are currently mounted. So if you are trying to test the new version of component library in a project that is already opened inside the Sun ONE Studio, we recommend that your first shut down the Sun ONE Studio. Once the Sun ONE Studio has released its hold on the old copy of the library JAR file, you can copy the new version of the JAR file over the old version. After successfully deploying the new version of the library, you may re-open the application in Sun ONE Studio.

2. **Create a new Model object. (If you have not done this before, please complete the *Sun ONE Application Framework Tutorial*. The new Model wizard should now look like this. (Note, depending on the version of Sun ONE Application Framework, you may not see all of the models that are pictured below. The important point is that you see the entry for "XML Document Model").**

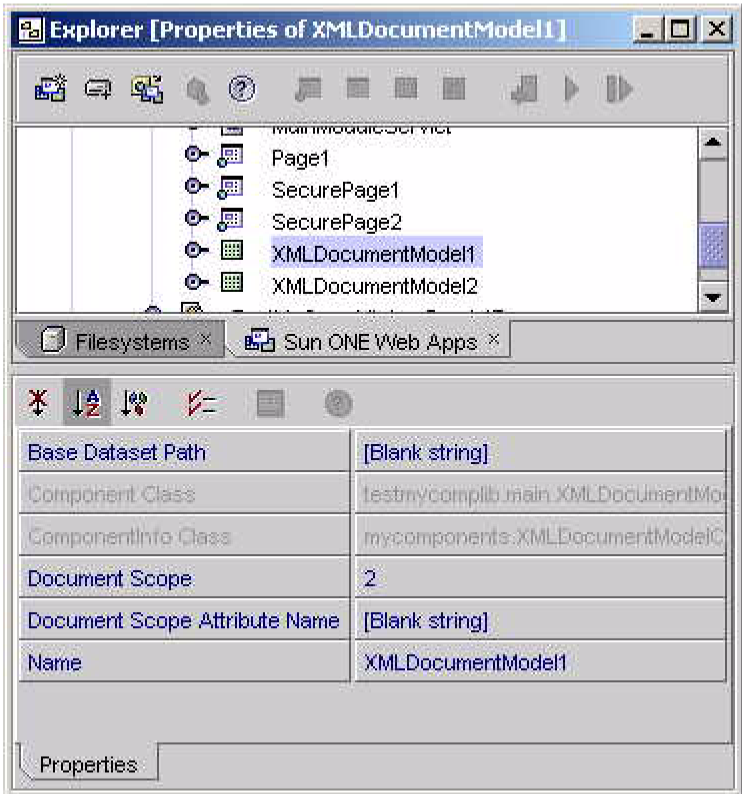


3. **Select the "XML Document Model" from the component list and complete the wizard. Take the default settings and let the wizard create "XMLDocumentModel1" for you.**

4. **After the wizard completes you can see that the IDE toolset has created a new class based on the component supplied template.**

5. **In order to test our mode fully, create a second XML Document Model.**

   Your application should now contain two XML Document Models (XMLDocumentModel1 and XMLDocumentModel2). Note the Base Dataset Name, Document Scope, and Document Scope Attribute Name properties.



- Also note above that the Document Scope property value is the raw integer 2. That is because XMLDocumentModelComponentInfo declared the DocumentScope property thus. The type is Integer.Type, and the default value is javax.servlet.jsp.PageContext.REQUEST_SCOPE. The net effect in the IDE will use the default Integer property editor which will express the raw integer value, in this case 2.

```
descriptor=new ConfigPropertyDescriptor(
     "documentScope",Integer.TYPE);
descriptor.setDisplayName("Document Scope");
descriptor.setHidden(false);
descriptor.setExpert(false);
descriptor.setDefaultValue(new Integer(
     javax.servlet.jsp.PageContext.REQUEST_SCOPE));
descriptors.add(descriptor);
```

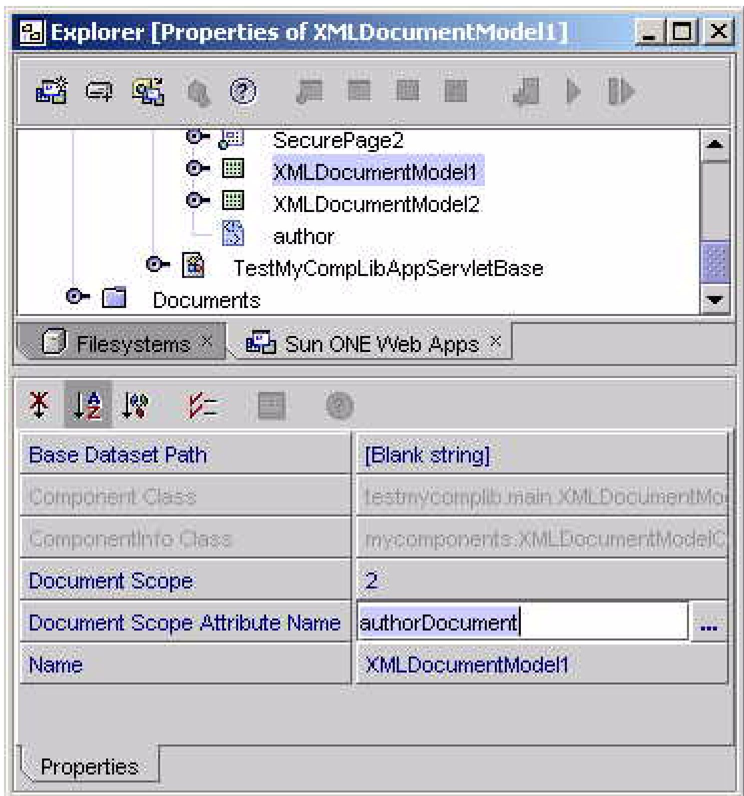
- You would be correct in thinking that this is a poor user interface since most developers will not know that 2 corresponds to request scope. Therefore, as a follow up exercise we shall later show you how to substitute a more user friendly property editor in place of the default Integer property editor.

6. **In order to test our new Model component we need a suitable XML document. We shall contrive a test case by placing an arbitrary XML file on disk, and at run time the test application will read the document from disk and place it into the request scope. Please remember our XML Document Model component does not care where the XML document comes from. In the real world, the XML document will probably be dynamically fetched by the application from the enterprise tier. That is of no concern to our XML Document Model.**
   - In any text editor, copy the following XML into a file named "author.xml".
   - Place author.xml in the same application module directory as XMLDocumentModel1 and XMLDocumentModel2. The code we will enter below will assume that it is in the same directory as our test Models. This is purely a convention of this exercise.

```
<?xml version="1.0"?>
<author>
     <name first="Charles" last="Dickens"/>
     <details birth="1812" death="1870"/>
     <works>
          <book title="Great Expectations" publisher="Penguin USA " pages="544"/>
          <book title="Nicholas Nickleby" publisher="Penguin USA " pages="816"/>
          <book title="A Tale of Two Cities" publisher="Signet Classic" pages="371"/>
          <book title="Hard Times" publisher="Bantam Classic" pages="280"/>
          <book title="Oliver Twist" publisher="Tor Books" pages="496"/>
          <book title="David Copperfield " publisher="Penguin USA " pages="912"/>
          <book title="A Christmas Carol" publisher="Bantam Classics" pages="102"/>
          <book title="Our Mutual Friend" publisher="Indypublish.Com" pages="472"/>
          <book title="Bleak House" publisher="Penguin USA " pages="1036"/>
          <book title="The Pickwick Papers " publisher="Penguin USA " pages="848"/>
          <book title="The Haunted House" publisher="Hesperus Press" pages="128"/>
          <book title="Little Dorrit" publisher="Indypublish.Com" pages="460"/>
          <book title="Barnaby Rudge" publisher="Viking Press" pages="766"/>
          <book title="The Mystery of Edwin Drood" publisher="Penguin USA" pages="432"/>
          <book title="Sketches by Boz" publisher="Penguin USA" pages="635"/>
     </works>
</author>
```

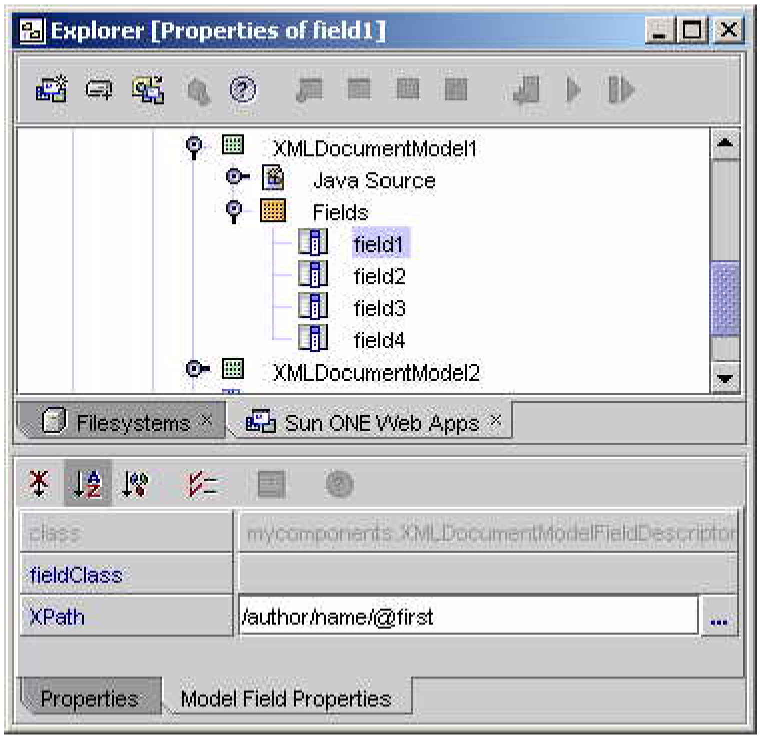
   - Review author.xml for a moment. Notice that for a single author, there are many book entries. Now is the time to point out, that we will actually utilize the two models (XMLDocumentModel1 and XMLDocumentModel2) to access different parts of the same XML document. We shall configure XMLDocumentModel1 to access the scalar author information, name and details. We shall configure XMLDocumentModel2 to access the non-scalar collection of books. We chose this approach when we designed the XMLDocumentModel because it simplifies both the implementation of the Model component and simplifies the usage of the component within an application.

7. **Configure XMLDocumentModel1 to access the scalar author information**
   - Select the XMLDocumentModel1 node.
   - Edit its Document Scope Attribute Name property. Set the value to "authorDocument".
   - Leave the Document Scope and the Base Dataset Path properties unchanged.



   - Expand the XMLDocumentModel1 node so that you can see its Fields sub-node.
   - Select the Fields sub-node.
   - Right mouse and select the pop up menu's Add Field ... action. This will automatically add a field with a default name. In this case the default name will be "field1".

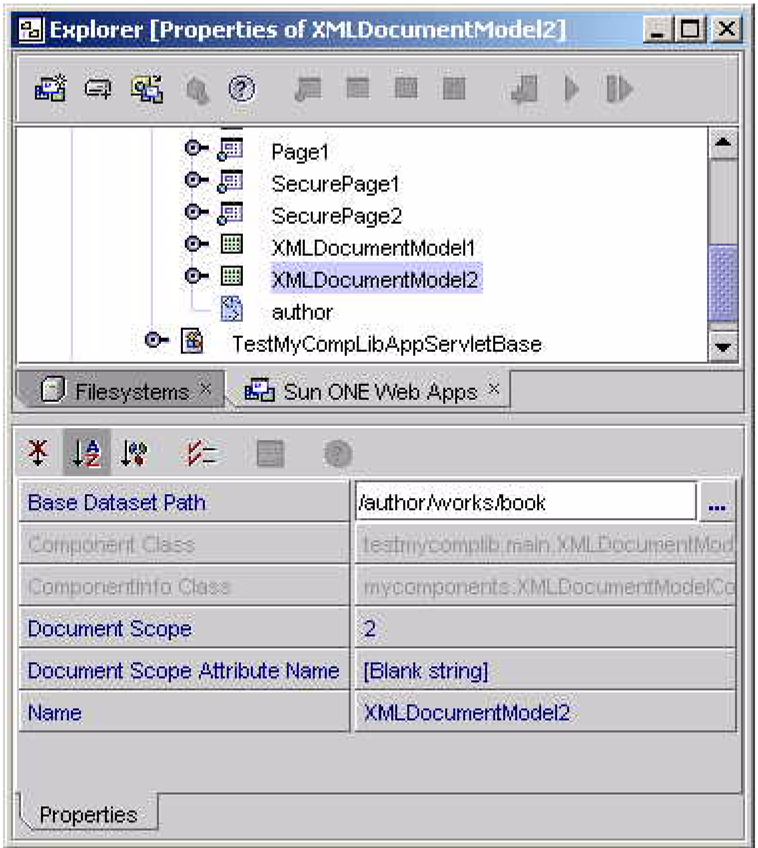
- Repeat the previous step to create additional fields "field2", "field3", and "field4". For the purposes of this exercise we will leave the names unchanged. In a real application, the Model developer would probably change the field names to make them more descriptive of their role.
- Select the field1 node. Select the Model Field Properties tab on its property sheet.
- Edit field1's XPath property. Set the value to the XPath expression "/author/name/@first"



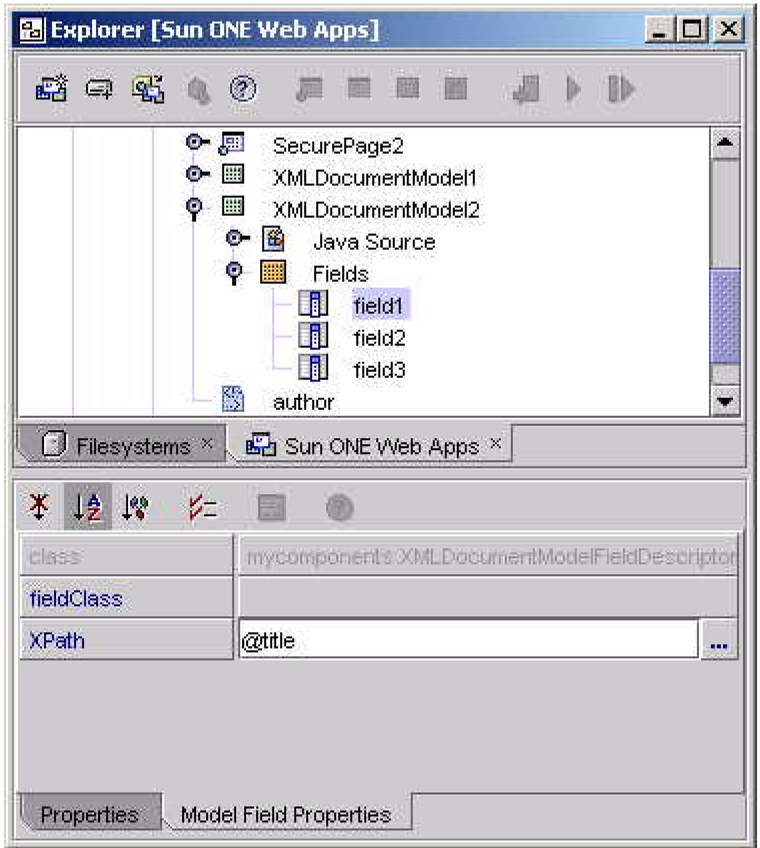
- Repeat the previous step and adjust the XPath property for the remaining three fields as follows
- Set field2's XPath value to "/author/name/@last"
- Set field3's XPath value to "/author/details/@birth"
- Set field4's XPath value to "/author/details/@death"

8. **Configure XMLDocumentModel2 to access the "books" dataset.**
   - Select the XMLDocumentModel2 node.
   - Edit its Document Scope Attribute Name property. Set the value to "authorDocument".

- Edit its Base Dataset Path property. Set the value to the XPath expression "/author/works/book". That is an XPath expression that will address the collection of book entries (i.e. a Sun ONE Application Framework dataset).
- Leave the Document Scope property unchanged.



- Expand the XMLDocumentModel2 node so that you can see its Fields sub-node.
- Select the Fields sub-node.
- Right mouse and select the pop up menu's Add Field ... action to add "field1", "field2", and "field3".
- Edit each field's XPath property to XPath expressions relative to the value of the Base Dataset Path property set above.
- Set field2's XPath value to "@title"
- Set field2's XPath value to "@publisher"
- Set field2's XPath value to "@pages"

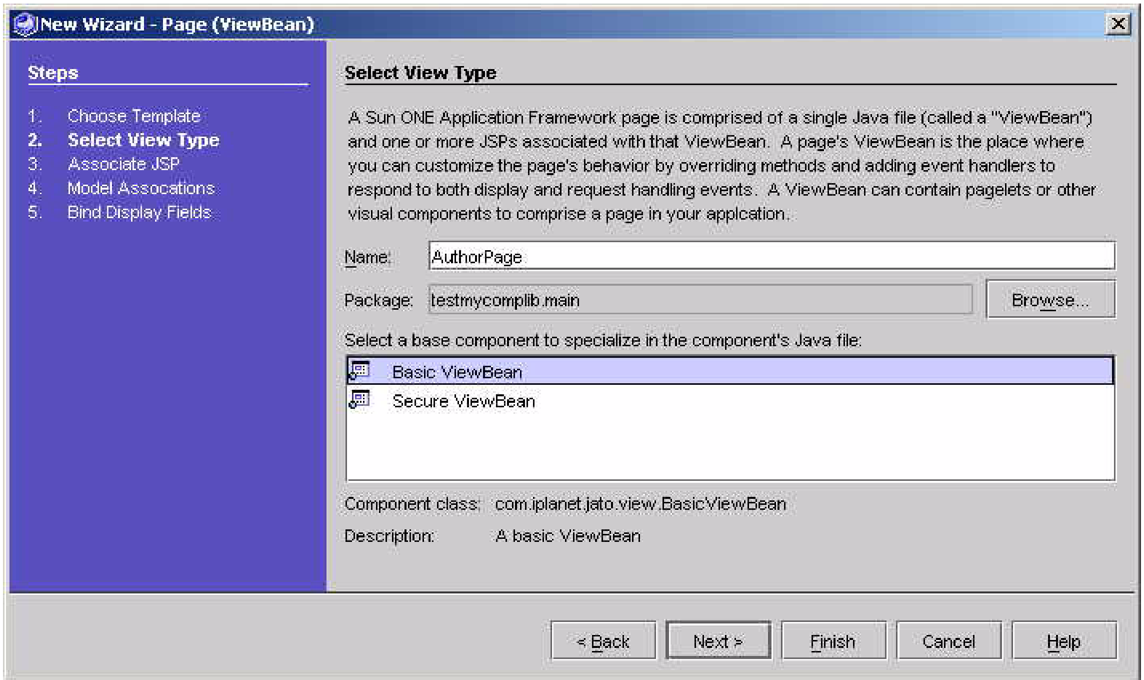


The models have now been configured. Now you need to create some Views to use the Models, and also provide some application logic to read the author.xml document from disk and store it in the request scope attribute "authorDocument". First, create the Views.

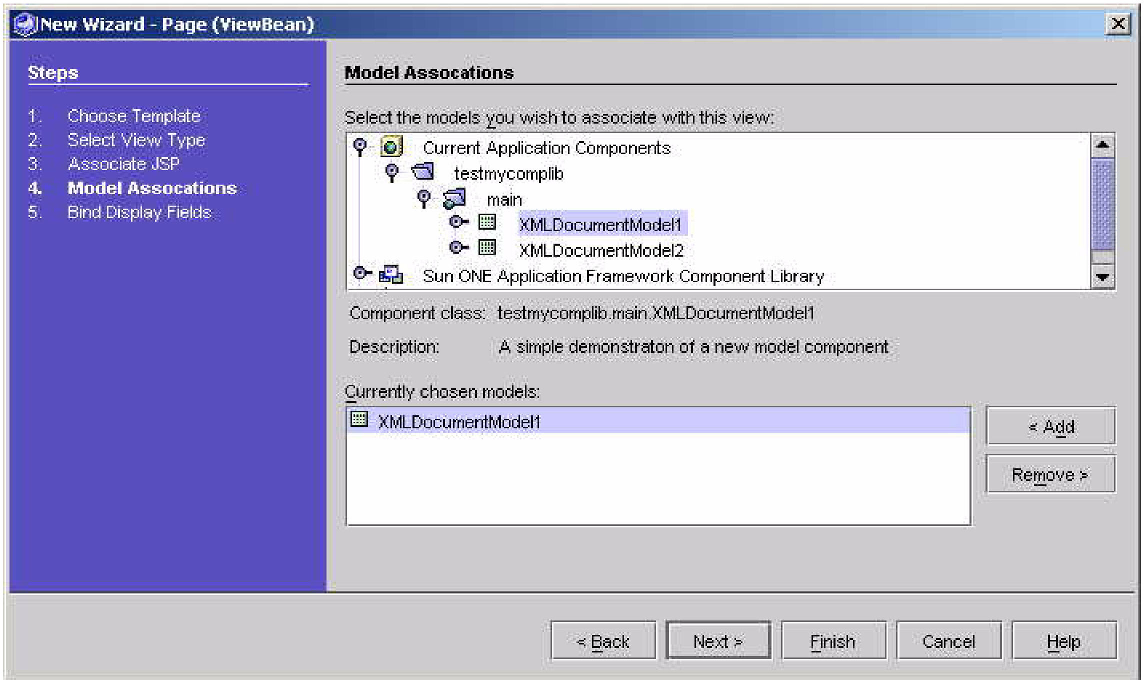
Advisory: Read this next step fully before attempting, since if you follow the instructions correctly you can save some time and effort by utilizing the full capability of the "New View" wizard. It is perfectly acceptable to create a ViewBean that will exercise the XMLDocumentModel1 without following the steps detailed below and you are free to create the ViewBean according to your preferred style. But for newcomers to the Sun ONE Application Framework, the following steps are, hopefully, the most concise.

9. **Create the AuthorPage**

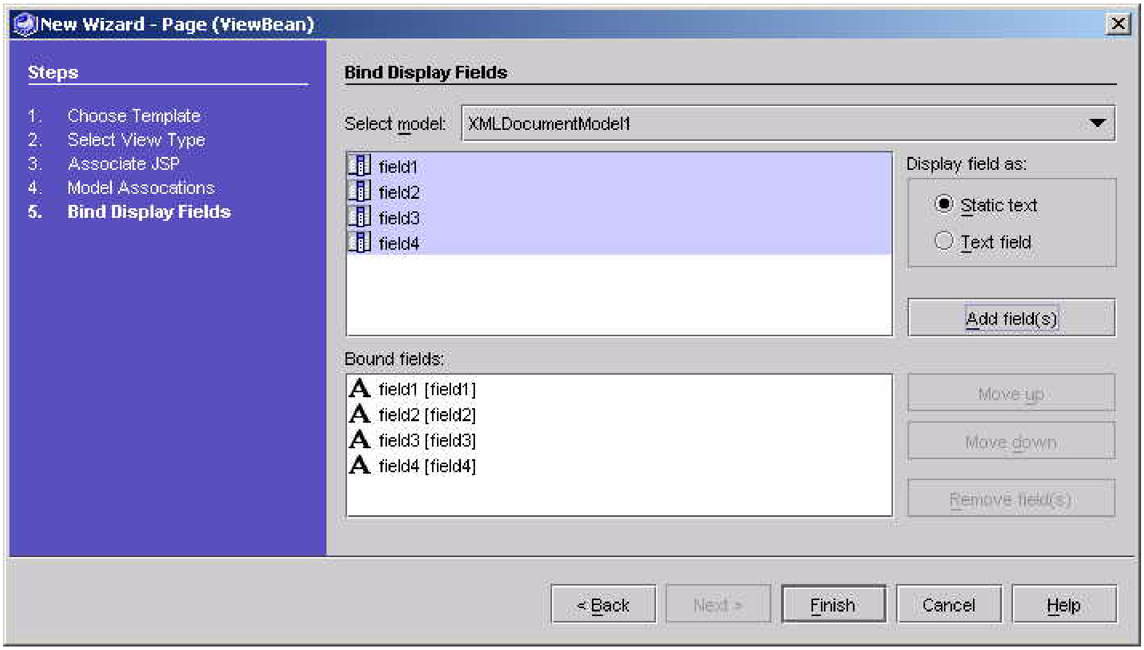
- Invoke the "New View" wizard. In the Select View Type panel (below) select the "Basic ViewBean" component. Explicitly set the Name to be "AuthorPage". Then press Next.



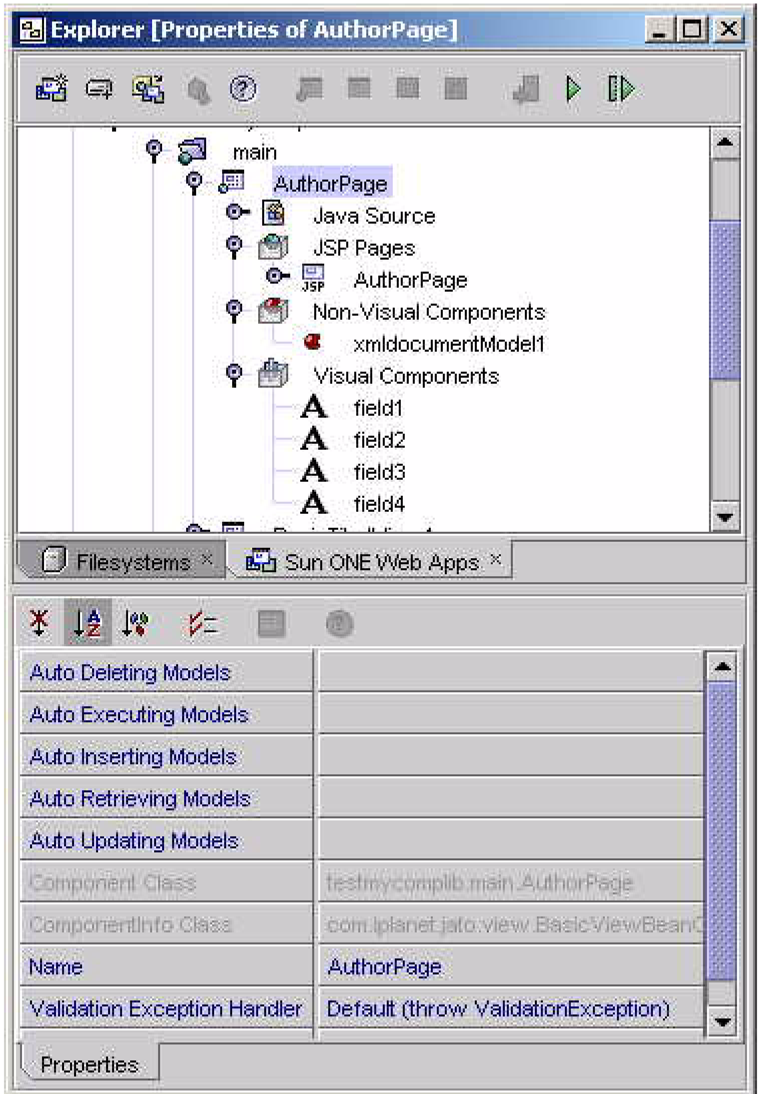
- Take the default values in the Associate JSP panel and press Next.
- In the Model Associations panel (below) expand the Current Application Components node until you find "XMLDocumentModel1". Select "XMLDocumentModel1" and press the Add button to create the association between the ViewBean and the Model. That will cause the "XMLDocumentModel1' to appear in the Currently chosen models section of the panel. Then press Next.



- In the Bind Display Fields panel (below) select all four of the Model fields that are available and press the Add Fields button. That will cause 4 entries to appear in the Bound Fields section of the panel. Then press Finish.



- After completing the wizard in the manner described above, you should find that the AuthorPage node looks like this. The individual child display fields should be properly bound to the corresponding XMLDocumentModel1 fields.



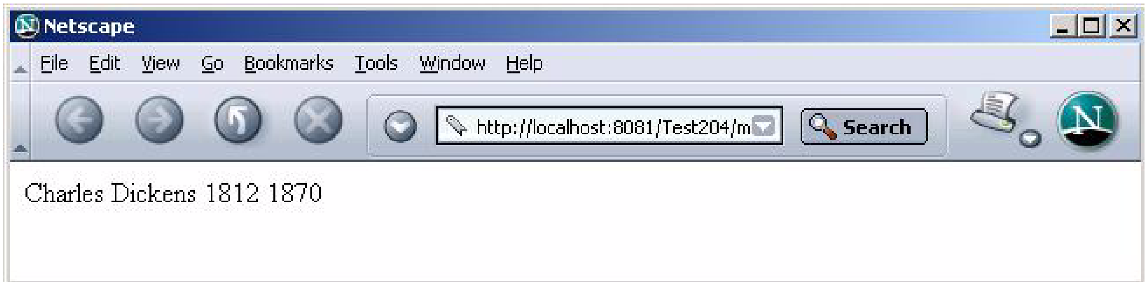
10. **Open AuthorPage.java and add the following code to the constructor. This code will read the "author.xml" document from disk and store it in the request scope attribute named "authorDocument", which is where the XMLDocumentModel1 expects to find it. The choice of placing this code here in the AuthorPage constructor is simply an arbitrary test stratagem. As stated before, the XMLDocumentModel does not care how or when the XML document is placed into the agreed upon attribute as long as it is there when the Model is accessed. Please note the extra import statements at the top. Also, please note the**

getResourceAsStream method's parameter must take a parameter which reflects the name of your test application (e.g. getResourceAsStream("/testmycomplib/main/author.xml")

```
import org.w3c.dom.*;
import org.xml.sax.InputSource;
import javax.xml.parsers.DocumentBuilderFactory;

/**
 *
 *
 */
public class AuthorPage extends BasicViewBean
{
     /**
      * Default constructor
      *
      */
     public AuthorPage()
     {
          super();
          try {

               InputSource in = new InputSource(AuthorPage.class.
                    getResourceAsStream("/testmycomplib/main/author.xml"));
               DocumentBuilderFactory dfactory = DocumentBuilderFactory.newInstance();
               dfactory.setNamespaceAware(true);
               Document doc = dfactory.newDocumentBuilder().parse(in);
               doc.normalize(); // Make sure text in the document is in normal form
               RequestManager.getRequest().setAttribute("authorDocument", doc);
               System.out.println("Author XML Document has been put into request");
          }
          catch(Exception e) {
               System.out.println("Exception trying to load author.xml" + e);
          }
     }
```

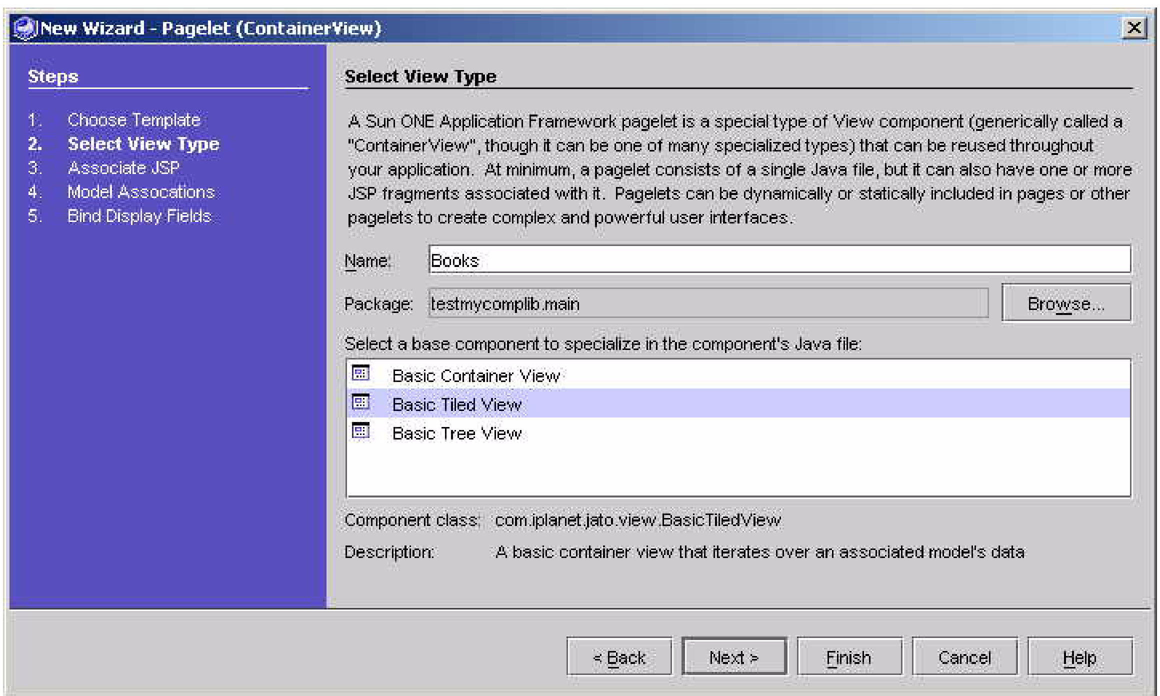
11. **Compile all the classes in the application and test run the AuthorPage**

    Author should appear in the browser as follows:



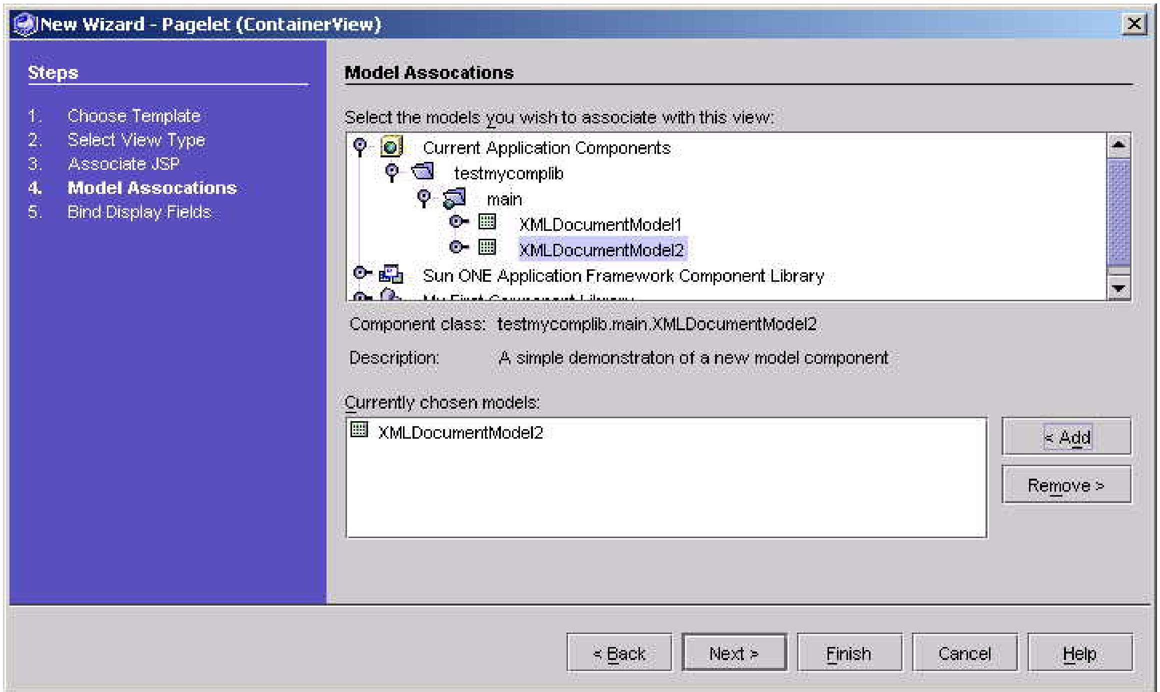
12. **Create a TiledView. Now you want to test XMLDocumentModel2 and its dataset capability. For this you will need to create a TiledView. Essentially, duplicate the steps taken in creating AuthorPage, but select a "Basic TiledView" instead of a "Basic ViewBean" and associate it with XMLDocumentModel2 instead of XMLDocumentModel1. Here are the detailed steps.**

    Invoke the "New View" wizard. In the Select View Type panel (below) select the "Basic Tiled View" component. Explicitly set the Name to be "Books". Then press Next.

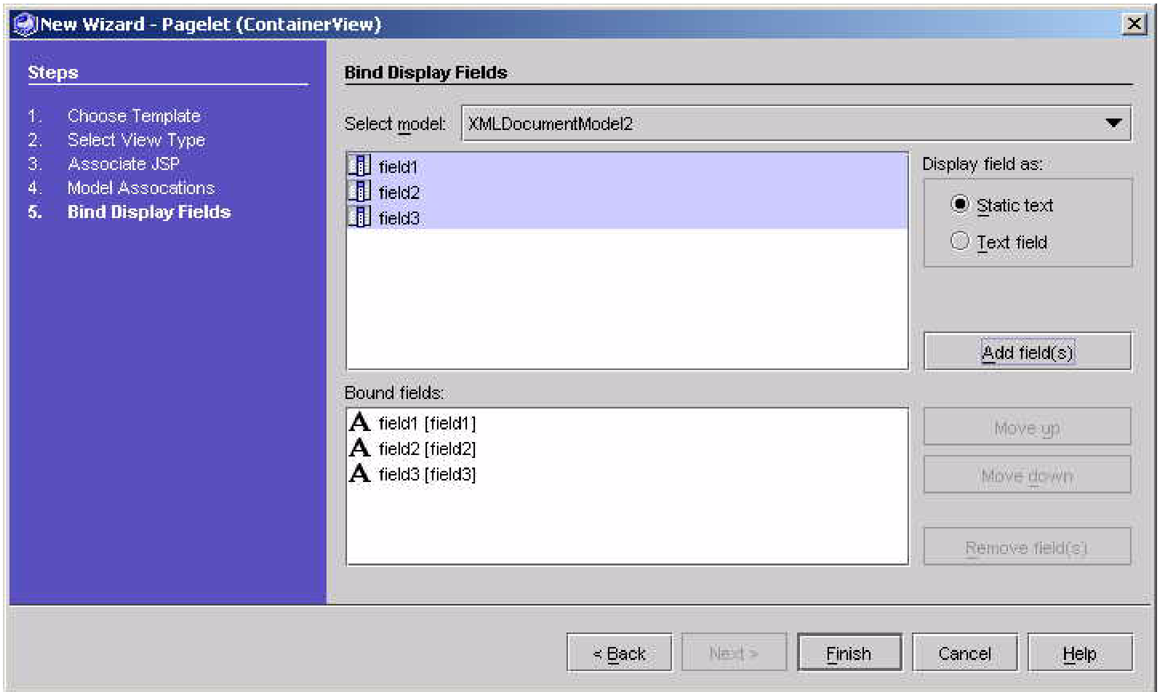


- Take the default values in the Associate JSP panel and press Next.

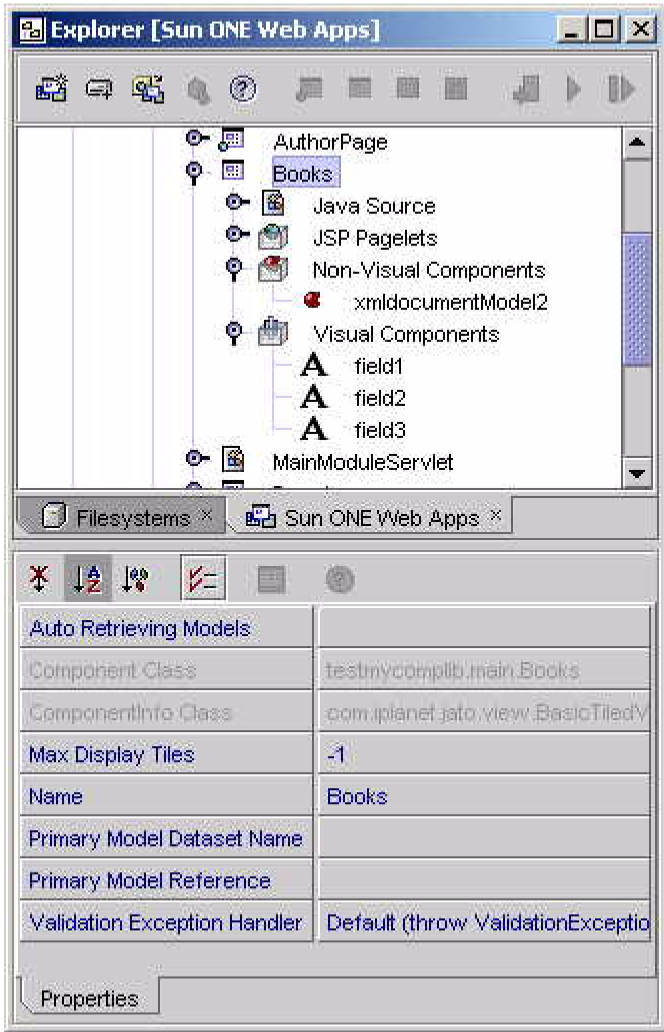
■ In the Model Associations panel (below) expand the Current Application Components node until you find "XMLDocumentModel2". Select "XMLDocumentModel2" and press the Add button to create the association between the TiledView and the Model. That will cause the "XMLDocumentModel2' to appear in the Currently chosen models section of the panel. Then press Next.



■ In the Bind Display Fields panel (below) select all three of the Model fields that are available and press the Add Fields button. That will cause 3 entries to appear in the Bound Fields section of the panel. Then press Finish.

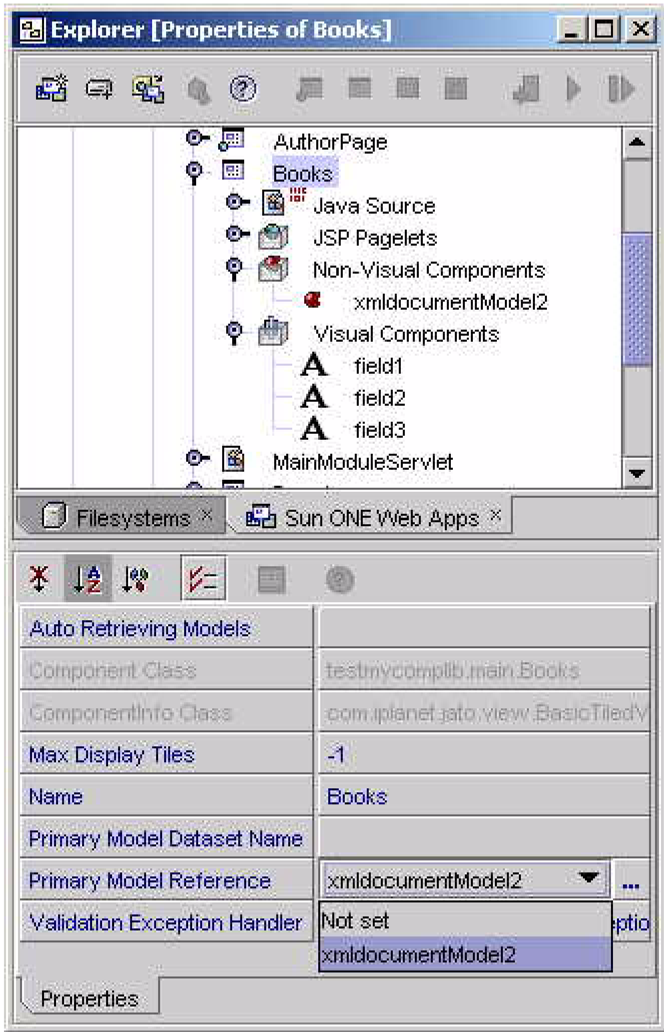


- After completing the wizard in the manner described above, you should find that the Books TiledView node looks like this. The individual child display fields should be properly bound to the corresponding XMLDocumentModel2 fields.



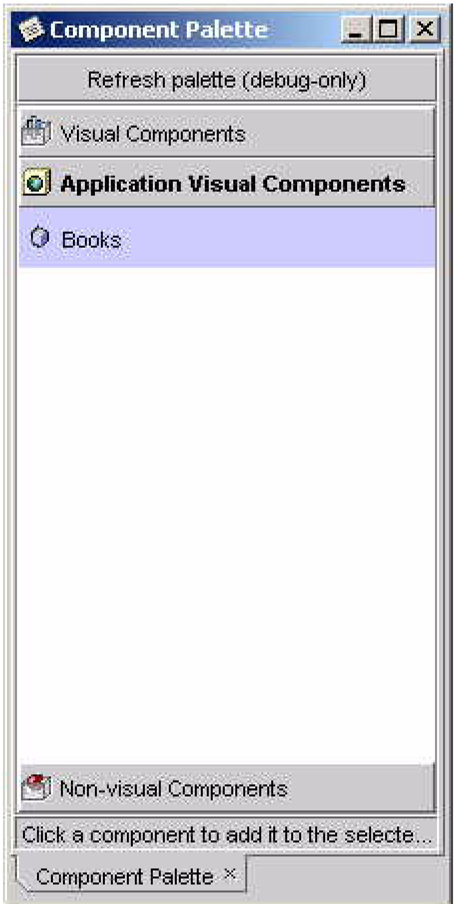
13. **The TiledView requires one additional configuration step that you have not seen before in this guide. You need to configure the :Book TiledView's Primary Model Reference property to reference "XMLDocumentModel2". If you do not configure this property, the test run of this TiledView will show no data. That is a common Sun ONE Application Framework application developer error.**

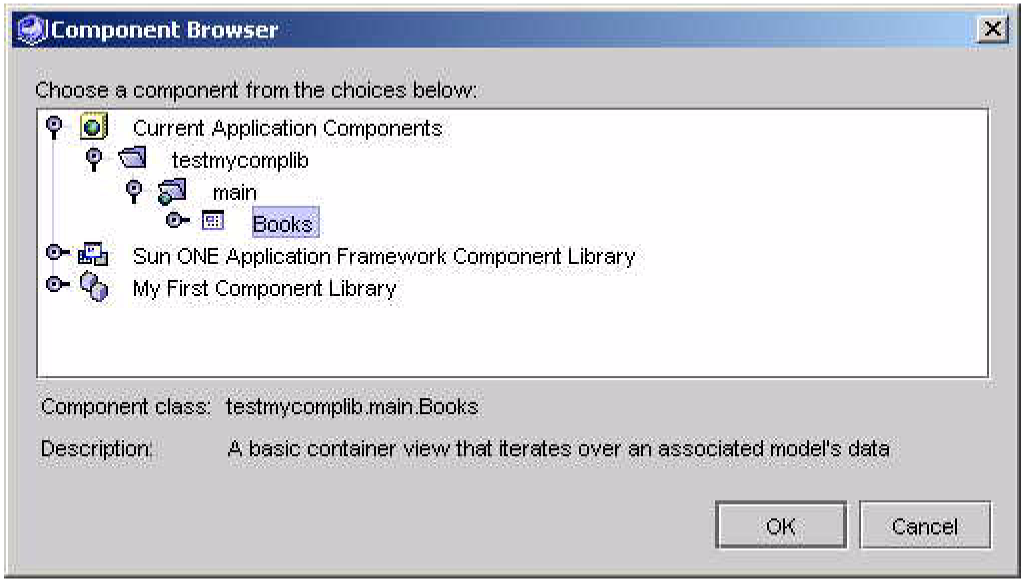
    Edit the Primary Model Reference property (below), and from its drop down list select the "xmldocumentModel2" value.



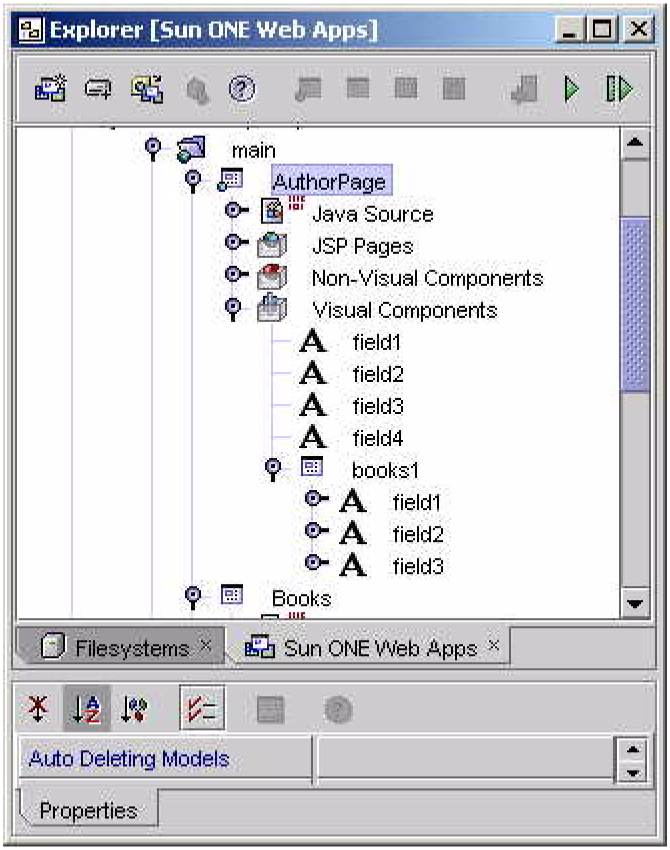
14. **Before you can test run the Books TiledView, you must add it to a ViewBean. To make our example more interesting, we want you to add the Books TiledView as a child of the AuthorPage. This is a good example of Sun ONE Application Framework's hierarchical view support.**

    Add an instance of Books TiledView to AuthorPage. Again, you can achieve this by selecting Books TiledView from either the Component Palette or the Component Browser, just as you did earlier with the various text fields and button components. Except this time, the component will be located in the Current Application Components section of either the Component Palette or Component Browser.


Component Palette
Refresh palette (debug-only)
Visual Components
Application Visual Components
Books
Non-visual Components
Click a component to add it to the selecte...
Component Palette



- After adding Books as a child view, the AuthorPage node should look like this;



15. Now, you can format the JSP associated with the AuthorPage to suit your taste. By default when you add a container view child to a ViewBean, the IDE toolset will add the container view child and its children's tags to the ViewBean's JSP(s). The application developer may use the Synchronize to View action on the JSP node to batch remove tags for any child container view children. Lets take this opportunity to also add some basic formatting to the JSP so it renders more neatly.

    a. Select and expand the AuthorPage's JSPs node.

    b. Select the actual AuthorPage.jsp node.

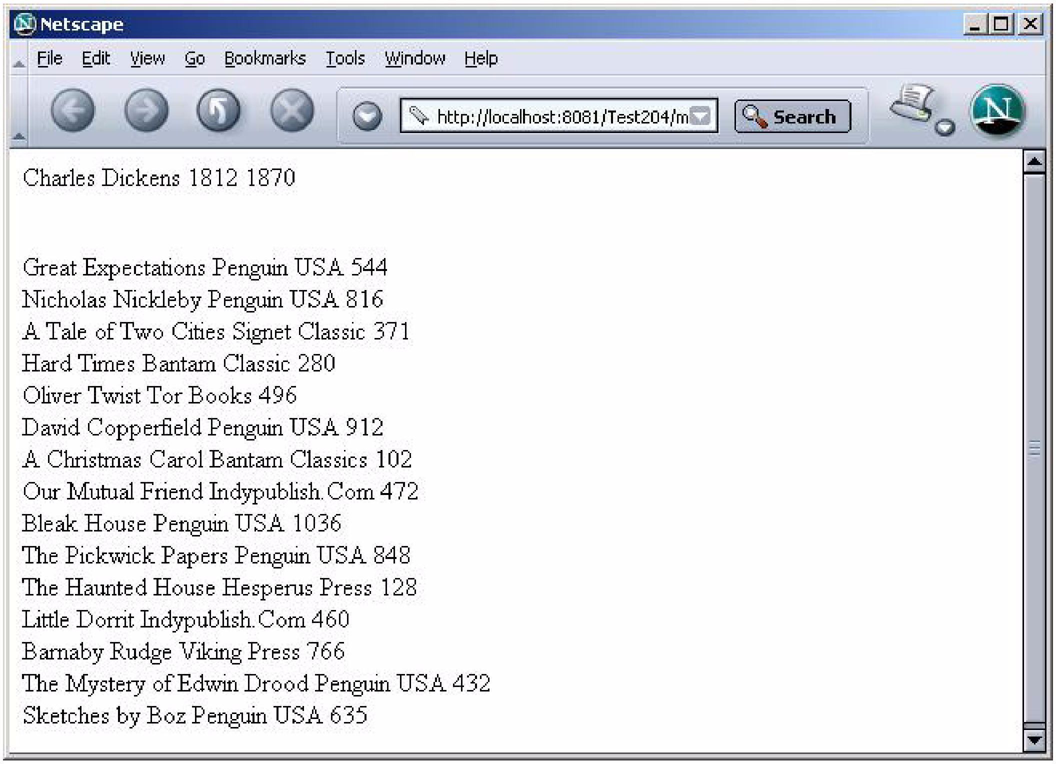
    c. Double click the AuthorPage.jsp node to open the JSP file in the editor so you can edit it.

**d. Now add some very basic formatting (e.g. <p> and
) to the AuthorPage.jsp so that we will get some separation between rows of Book data. When done, the AuthorPage.jsp should look something like this:**

```
<%@page contentType="text/html; charset=ISO-8859-1" info="AuthorPage" language="java"%>
<%@taglib uri="/WEB-INF/jato.tld" prefix="jato"%>

<jato:useViewBean className="testmycomplib.main.AuthorPage">

<html>
<head>
<title></title>
</head>
<body>

<jato:form name="Author" method="post">

<jato:text name="field1"/>
<jato:text name="field2"/>
<jato:text name="field3"/>
<jato:text name="field4"/>
<p>
<jato:tiledView name="books1">
<br>
<jato:text name="field1"/>
<jato:text name="field2"/>
<jato:text name="field3"/>
</jato:tiledView></jato:form>

</body>
</html>

</jato:useViewBean>
```

16. Finally, test run AuthorPage again. Make sure you select the ExecutePage (Redeploy) action, not the Execute Page action, otherwise you will not see the effects of the recent changes.

The contents of AuthorPage.jsp should show up in the browser. (because the user had accumulated the required tokens).



Ship It? Not yet, provide a custom editor for that "Document Scope" property

Recall that the XMLDocumentModel's Document Scope property is currently vulnerable to user error, because it exposes a raw int value for direct editing. What you really need is a custom editor that presents the application developer with a fool proof drop down list containing only valid choices. You would normally spend a little time and develop a custom property editor. The details of that are beyond the scope of this document, but can be found in any basic JavaBean reference.

CHAPTER 5

Developing Command Components

This section assumes that you have already read Develop Your First Component found in Chapter 2, "Developing Components" on page 17.

Developing an Extensible Command Component

In this section we shall describe how to create a new Command component that adds value on top of the ValidatingDisplayField we created in the Developing a Non-Extensible View Component section found in Chapter 3, "Developing View Components" on page 43. We shall call this the ValidatingCommand component and it will encapsulate some reusable logic related to the processing of pages which contain instances of the ValidatingDisplayField.

This exercise is intended to focus on the mechanics of extensible Command component design and as such, will only scratch the surface of extensible Command possibilities.

As you will see, the mechanics of creating extensible Commands are very straightforward. If you have completed the previous sections you know the mechanics by now. The reality is that Commands are structurally simple. This section will reinforce your familiarity with the command design pattern, introduce you to its role in the Sun ONE Application Framework, and hopefully encourage you to become creative in leveraging this simple but powerful pattern.

This example will allow us to introduce several additional Sun ONE Application Framework component model topics

- CommandComponentInfo
- CommandDescriptor

Our validating Command component should support the following design-time functionality:

- Allow developers to subclass our ValidatingCommand and add some application specific behavior on top of the behavior encapsulated in the base class.

  Specifically, subclass developers will focus on implementing two component specific methods handleInvalid and handleValid instead of the conventional Command execute method. The application command developer will rely on the component base class to determine if the page on which the component has been activated is valid or invalid, and invoke the appropriate handler. Therefore, application command developers can focus on responding to the valid or invalid state and not need to worry about detecting the state.

- The extensible command allows component authors to specify configuration properties, like we did with the View and Model components. However, in this example we will not need to define any.

Our validating Command component should support the following run-time functionality:

The ValidatingCommand base class implementation of the execute method will perform a deep search on the submitted ViewBean, detect any and all instances of ValidatingTextField and check to see if any of those fields are invalid.

  - If any field is found to be invalid, the ValidatingCommand base class will invoke the handleInvalid method. It is assumed that ValidatingCommand will override the handleInvalid method if they wish to perform some command specific behavior. The base class implementation of handleInvalid will simply redisplay the invalid page. This behavior may be deemed sufficient in some cases.
  - If no fields are found to be invalid, the ValidatingCommand base class will invoke the handleValid method. It is assumed that ValidatingCommand will override the handleValid method to perform some command specific behavior. The base class implementation of handleValid is an abstract method. This command assumes that the application specific command developer will implement the handleValid method.

The choice to implement this Command component as outlined above is purely a matter of style. As pointed out the command pattern is elementary. Therefore, personal OO style will factor largely into component authors designs.

To meet these requirements, we will design and implement the following classes:

- Component class - mycomponents.ValidatingCommand
- ComponentInfo class - mycomponents.ValidatingCommandComponentInfo

Additionally, we will implement a custom Java template which the IDE toolset will use as the basis for application specific sub-types of our ValidatingCommand.

Finally we shall edit the mycomponents complib.xml to add the new component to the Sun ONE Application Framework component library.

Note this example assumes the co-existence of the mycomponents.ValidatingDisplayField. If you have not already completed the Developing a Non-Extensible View Component section, please do so before continuing.

Create the Sun ONE Application Framework Component Class

1. **In any Java editor create the class mycomponents.ValidatingCommand**
2. **Make ValidatingCommand extend com.iplanet.jato.comand.BasicCommand**
3. **Implement the appropriate constructor for the component type. All Command components must implement a no-arg constructor.**
4. **Implement the remaining methods that are required to fulfill our component specific requirements.**
   - Implementation of execute method which will enforce the component's validation logic.
   - Default implementation of the component's handleInvalid method.
   - Abstract declaration of the component's handleValid method.

After these steps, mycomponents/ValidatingCommand.java should look like this

```
package mycomponents;
import java.util.*;
import com.iplanet.jato.*;
import com.iplanet.jato.command.*;
import com.iplanet.jato.model.*;
import com.iplanet.jato.view.*;
import com.iplanet.jato.view.event.*;


public abstract class ValidatingCommand extends Object implements Command {


    public ValidatingCommand() {
        super();
    }


    public void execute(CommandEvent event) throws CommandException {
        Map map=event.getParameters();

        try {
            boolean isValid = true;
            ViewBean viewBean =  ViewBase.getRootView((View)event.getSource());
            List vFields = getValidatingTextChildren(viewBean);
            Iterator iter = vFields.iterator();
```

```
            while(iter.hasNext()) {
                ValidatingDisplayField vText = (ValidatingDisplayField)iter.next();
                if(! vText.isValid()) {
                    isValid = false;
                    break;
                }
            }

            if( isValid ) {
                handleValid(event);
            }
            else {
                handleInvalid(event, viewBean);
            }

        }
        catch (Exception e) {
            if (e instanceof CommandException)
                throw (CommandException)e;
            else {
                throw new CommandException(
                    "Error executing ValidatingCommand",e);
            }
        }
    }


    public List getValidatingTextChildren(ContainerView container) {
        List result=new LinkedList();

        String[] childNames=container.getChildNames();
        for (int i=0; i<childNames.length; i++) {
            Class childType=container.getChildType(childNames[i]);
            if (ValidatingDisplayField.class.isAssignableFrom(childType)) {
                ValidatingDisplayField child=(ValidatingDisplayField)
                container.getChild(childNames[i]);
                result.add(child);
            }
            else if (ContainerView.class.isAssignableFrom(childType)) {
                ContainerView child=
                    (ContainerView) container.getChild(childNames[i]);
                result.addAll(getValidatingTextChildren(child));
            }
        }
        return result;
    }


    public abstract void handleValid(CommandEvent event) throws CommandException;
```

```
    public void handleInvalid(CommandEvent event, ViewBean invalidVB)
        throws CommandException {
        // default implementation is to just redisplay the invalid page
        invalidVB.forwardTo(event.getRequestContext());
    }

}
```

Create the Extensible Component's Java template

Extensible components serve as base classes for application defined entities. Therefore, the Sun ONE Application Framework component model provides extensible component authors the opportunity to provide a custom Java template. The IDE toolset will, subsequently, use the component supplied template to create the application specific sub-type. Component authors can utilize the custom template to enhance the application developer's experience. Component authors may prepare the component specific Java template with a set of template tokens defined in com.iplanet.jato.component.ExtensibleComponentInfo. For token details see ExtensibleComponent API.

Component authors may also utilize any arbitrary Java constructs within the Java template (for example, import statements, methods, variables, interface declarations, and so on). Minimally, the custom template will ensure that the new Java class extends from the extensible component class.

In this example we will use the template to assist the developer in their implementation of methods which are declared abstract in the base class.

- In any text editor create the template mycomponents.resources.ValidatingCommand_java.template
- The template contents should look like this. Note the tokens follow a __TOKEN__ pattern.

```
package __PACKAGE__;

import com.iplanet.jato.*;
import com.iplanet.jato.command.*;
import com.iplanet.jato.model.*;
import com.iplanet.jato.view.*;
import com.iplanet.jato.view.event.*;
import mycomponents.*;

/**
 *
 *
 */
public class __CLASS_NAME__ extends ValidatingCommand
{
    /**
     * Default constructor
     *
     */
    public __CLASS_NAME__()
    {
        super();
    }

    /**
     *
     *
     */
    public void handleValid(CommandEvent event) throws CommandException
    {
        // TODO - Developers must implement this method.
    }

}
```

Create the ComponentInfo Class

The ComponentInfo class defines the design-time metadata that the IDE toolset requires to incorporate the component. In this example we will extend an existing ComponentInfo and in true OO style, simply augment it. We could, of course, choose to implement the ComponentInfo interface from scratch, but that would be unproductive in this case.

In this example, we are not going beyond the functionality revealed in our earlier component examples.

1. **Create the class mycomponents.ValidatingCommandComponentInfo.**
2. **Make ValidatingCommandComponentInfo extend com.iplanet.jato.command.BasicCommandComponentInfo**
3. **Implement the no-arg constructor.**
4. **No need to Implement the getComponentDescriptor() method since we do not need to define any new properties.**
5. **Implement the getPrimaryTemplateAsStream() method to return a Java template file which you wish the IDE toolset to use as the starting point for new classes derived from this extensible component.**

   After these steps, mycomponents/ValidatingCommandComponentInfo.java should look like this:

Note – In this sample code we have embedded String values directly for ease of demonstration. If you anticipate the need to localize your display strings, we encourage you to utilize resource bundles.

```
package mycomponents;
import java.util.*;
import java.awt.Image;
import java.io.*;
import com.iplanet.jato.component.*;
import com.iplanet.jato.command.*;


public class ValidatingCommandComponentInfo extends BasicCommandComponentInfo   {


    public ValidatingCommandComponentInfo()
    {
        super();
    }


    public ComponentDescriptor getComponentDescriptor()
    {
        // identify the component class
        ComponentDescriptor result=new ComponentDescriptor(
            "mycomponents.ValidatingCommand");

        // The name will be used to determine a name for the component instance
        result.setName("ValidatingCommand");

        // The display name will be used to show the component in a chooser
        result.setDisplayName("Validating Command");
```

```
        // The description will be the tool tip text for the component
        result.setShortDescription("A validating command component");

        return result;
    }


    public String getPrimaryTemplateEncoding()
    {
/* Production version would be resource bundle driven, like this:
return getResourceString(getClass(),
"PROP_ValidatingCommand_SOURCE_TEMPLATE_ENCODING", "ascii");
*/

        return "ascii"; // NOI18N
    }


    public InputStream getPrimaryTemplateAsStream()
    {
/* Production version would be resource bundle driven, like this:

return ValidatingCommandComponentInfo.class.getClassLoader().
getResourceAsStream(
getResourceString(getClass(),
"RES_ValidatingCommandComponentInfo_SOURCE_TEMPLATE",""));
*/

        return mycomponents.ValidatingCommandComponentInfo.class.getResourceAsStream(
        "/mycomponents/resources/ValidatingCommand_java.template"); // NOI18N
    }

}
```

Augment the Component Library Manifest

We have already created the component manifest in the earlier example. So now we will simple add additional information. Note that we will add additional types of information not seen in the prior example.

The Sun ONE Application Framework library manifest must be named complib.xml. Within the JAR file, the Sun ONE Application Framework library manifest must be placed in the /COMP-INF directory.

1. **Create/Open the file COMP-INF/complib.xml**

2. **Add an extensible component element to declare the ValidatingCommand component.**

After these steps, the COMP-INF/complib.xml file should look like this. In the interest of clarity, we are only showing the significant delta to the prior version of this file shown earlier.

```
<?xml version="1.0" encoding="UTF-8"?>
<component-library>
        <tool-info>
                <tool-version>2.1.0</tool-version>
        </tool-info>
        <library-name>mycomponents</library-name>
        <display-name>My First Component Library</display-name>

        ...

    <extensible-component>
        <component-class>mycomponents.ValidatingCommand</component-class>
        <component-info-class>mycomponents.ValidatingCommandComponentInfo</component-info-
class>
    </extensible-component>

        ...

</component-library>
```

Re-create the Component Library JAR File

Once again, JAR up the component classes as we did in the first example so that they can be ready for distribution as a library.

1. **The name of the JAR file is arbitrary. In this case, name it "mycomponents.jar"**
2. **You may omit the Java source files from the JAR**
3. **You should include in the JAR any necessary ancillary resources, like icon images, or resource bundles. In this case there are none.**

   In this case we are now including several new classes and a Java template file

4. **The mycomponents.jar internal structure should look like this:**

```
mycomponents/resources/SecureViewBean_java.template
mycomponents/resources/ValidatingCommand_java.template
mycomponents/resources/XMLDocumentModel_java.template
mycomponents/MissingTokensEvent.class
mycomponents/MyTextField.class
mycomponents/MyTextFieldComponentInfo.class
mycomponents/SecureViewBean.class
mycomponents/SecureViewBeanComponentInfo.class
mycomponents/TypeValidator.class
mycomponents/ValidatingCommand.class
mycomponents/ValidatingCommandComponentInfo.class
mycomponents/ValidatingDisplayField.class
mycomponents/ValidatingTextFieldComponentInfo.class
mycomponents/ValidatingTextFieldTag.class
mycomponents/Validator.class
mycomponents/XMLDocumentModel.class
mycomponents/XMLDocumentModelComponentInfo.class
mycomponents/XMLDocumentModelFieldDescriptor.class
mycomponents/mycomplib.tld
COMP-INF/complib.xml
```

Test the New Component

Additional Sun ONE Application Framework IDE toolset features introduced in this section:

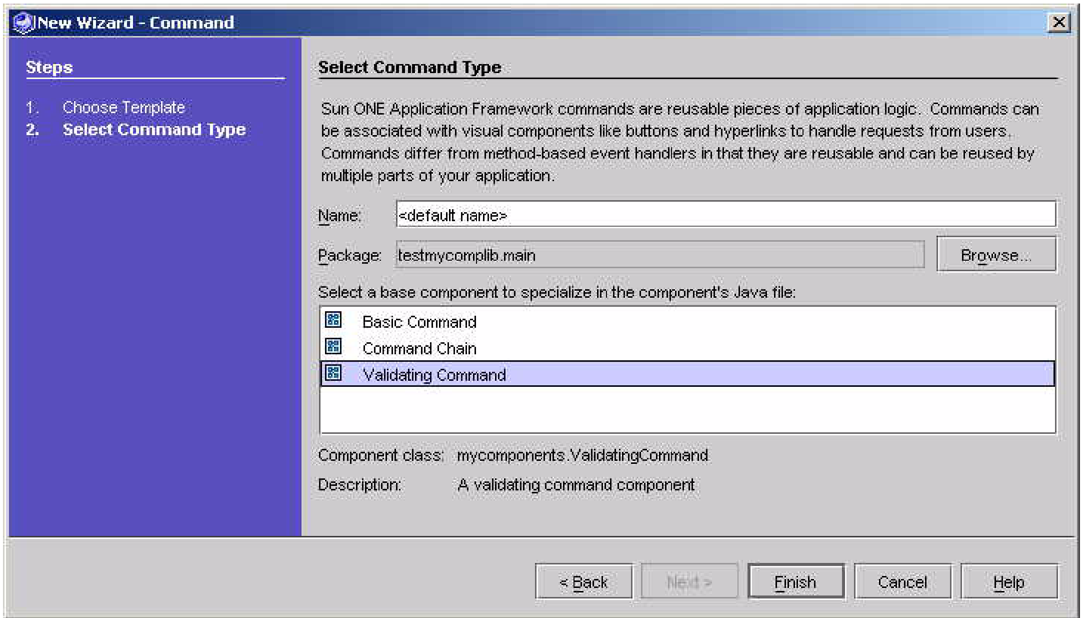
- The new Command Wizard
- The Command Descriptor property editor

1. **Deploy the new version of the library into your previously created test application**

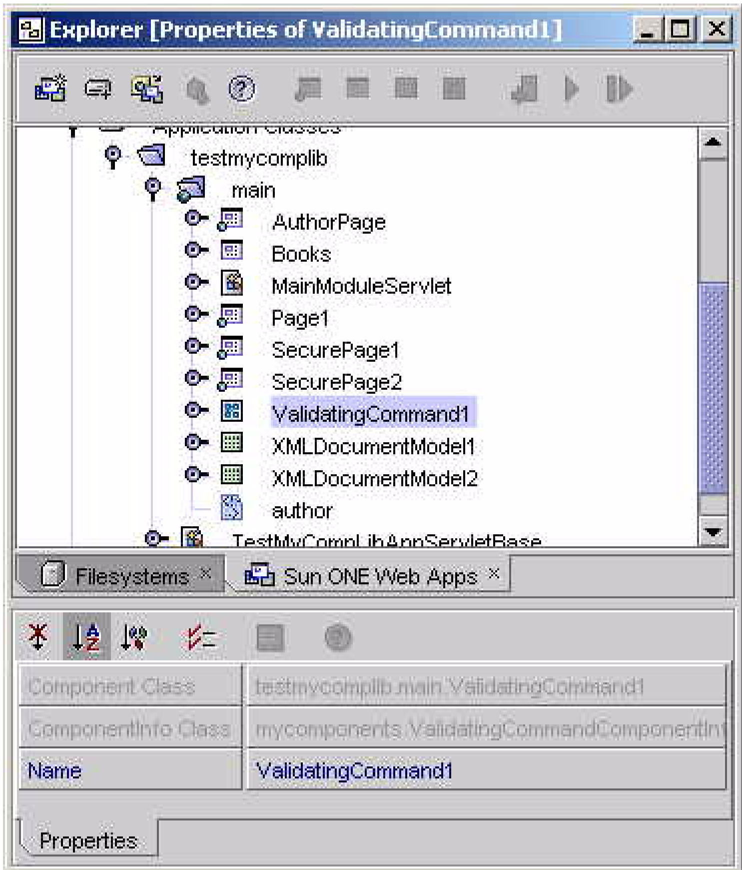
   Important Sun ONE Studio note: The Studio will not let you delete or copy over a JAR file that is currently mounted. Actually, we recommend shutting down the Sun ONE Studio whenever you need to replace one of the JAR files that are currently mounted. So if you are trying to test the new version of component library in a project that is already opened inside the Sun ONE Studio, we recommend that your first shut down the Sun ONE Studio. Once the Sun ONE Studio has released its hold on the old copy of the library JAR file, you can copy the new version of the JAR file over the old version. After successfully deploying the new version of the library, you may re-open the application in Sun ONE Studio.

2. **Create a new Command object. (If you have not done this before, complete the *Sun ONE Application Framework Tutorial*).**

   The "New Command" wizard should now look like this:



3. **Select "Validating Command" from the component list and complete the wizard. Take the default settings and let the wizard create ValidatingCommand1 for you.**
4. **After the wizard completes you can see that the IDE toolset has created a new class based on the component supplied template.**
5. **You application should now contain a ValidatingCommand1 object.**



6. **The remaining steps assume that your test application already contains two pages that we can leverage in testing your new ValidatingCommand.**

   The two pages you will need are Page1 from the Develop a Non-Extensible View Component section and SecurePage1 from the Develop an Extensible View Component section. You can see those ViewBeans in the test application explorer graphic just above.

7. **First, you need to complete the coding of ValidatingCommand1.**

   As designed, the superclass will handle validation state detection, while the application specific class (e.g ValidatingCommand1) is responsible for determining what to do when the submitted page is valid. This is a very application specific determination. So in the interest of simply testing the component we will code its handleValid method to just display SecurePage1.

   a. **Open the java source for ValidatingCommand1**

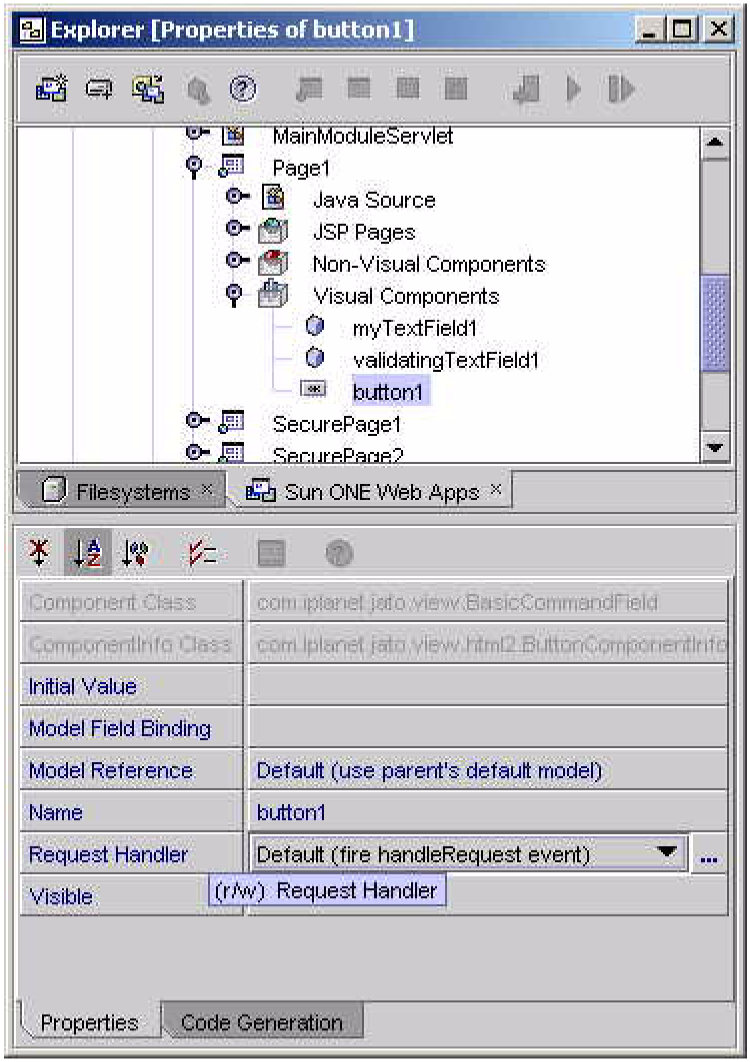
b. **Implement its handleValid method**

```
public void handleValid(CommandEvent event) throws CommandException
    {
        ViewBean next = event.getRequestContext().getViewBeanManager().getViewBean(
            SecurePage1.class);
        next.forwardTo(event.getRequestContext());
    }
```

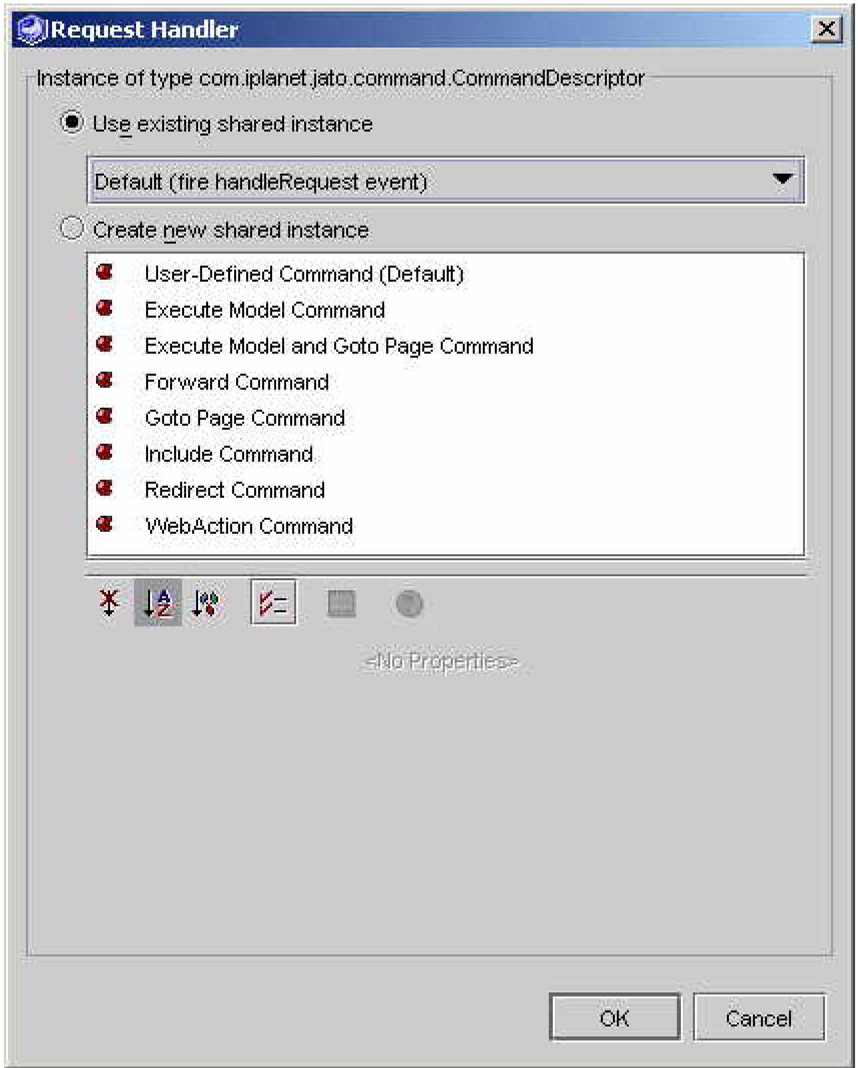
8. **In order to test a ValidatingCommand component, in particular, you need a ViewBean that contains some ValidatingDisplayFields. Fortunately, we created just such a ViewBean earlier, Page1**
   - Select the Page1 node
9. **To test any Command component, you need to set up a uses relationship between a command client (for example, a CommandField like a Button or HREF) and your command object. At run-time, the CommandField will use (activate) the command object. At design-time one establishes this uses relationship by configuring a CommandDescriptor (to declaratively describe a Command instance) and associating this CommandDescriptor with a CommandField. The IDE toolset actually makes this relatively easy by allowing developers to initiate this multi-step configuration process by selecting the CommandField first and it will automatically walk you through the configuration of the CommandDescriptor as part of the CommandField configuration. Learning is doing, so follow the steps below and see.**

   a. **Select its Visual Components sub-node.**

   b. **Select the node for button1 (below)**



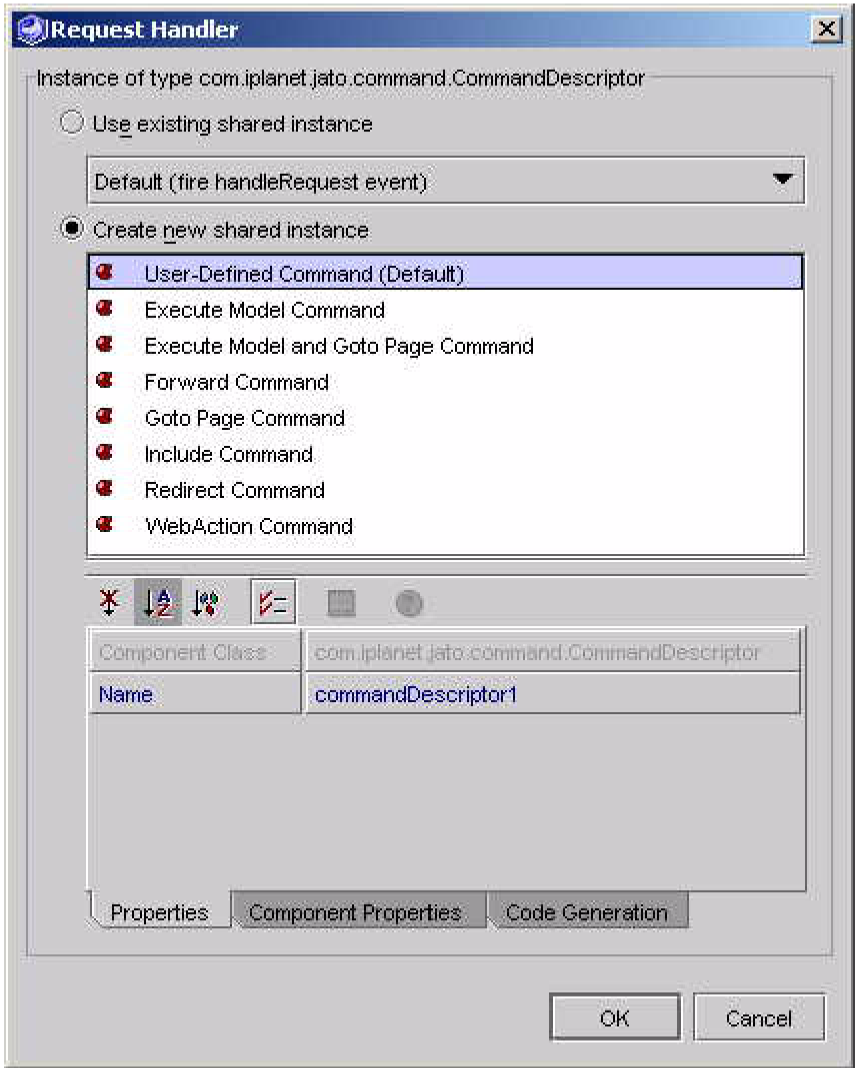
10. **Edit button1's Request Handler property. Click on the property's ellipsis to bring up the full blown Command Descriptor Property Editor. This is a very sophisticated editor and takes some effort to get familiar with it, but the pay off is substantial, as it offers exciting additional opportunities to component authors which we will discuss later. First you must become comfortable using the editor. The Command Descriptor Property Editor contains a dynamic list of available CommandDescriptor types and it also contains an embedded property sheet (at bottom of editor) which will dynamically display the properties for the type of CommandDescriptor that is selected (this will become more clear in subsequent steps).**



11. **Select the Create new shared instance radio button.**

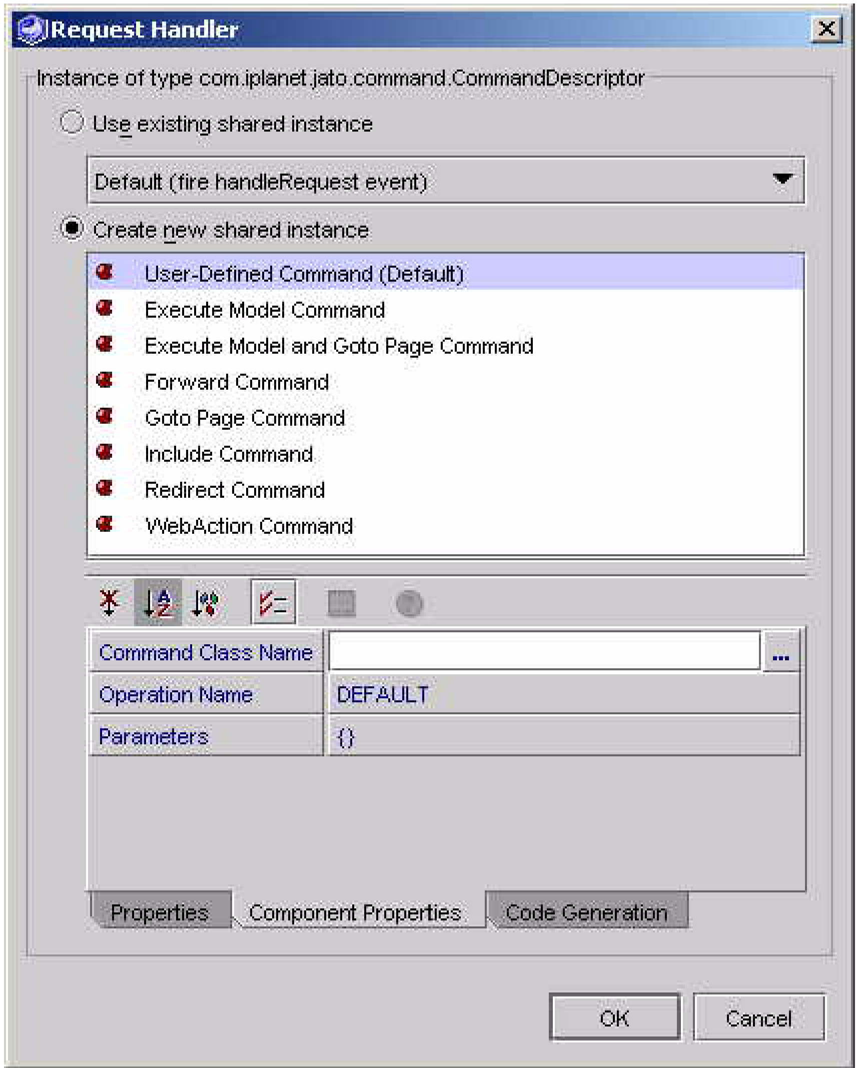
    We will explain the meaning of "shared instance" in just a few more steps when it will be easier to clarify.

12. **Select the "User-Defined Command (Default)" item from the list of available descriptor types (see graphic above). As you select a given command descriptor type, the bottom section of the editor will display the properties which are particular to the type of descriptor you selected. Your property editor should look like this:**

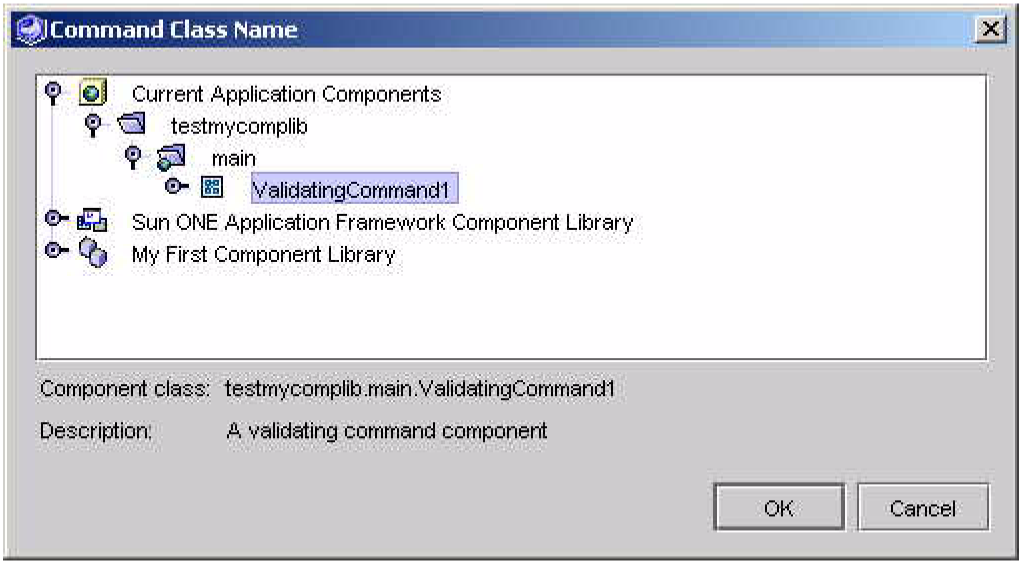


13. Note that the embedded property sheet contains three tabs Properties, Component Properties and Code Generation. Select the Component Properties tab.

- Within the Component Properties tab select the Command Class Name property (below)

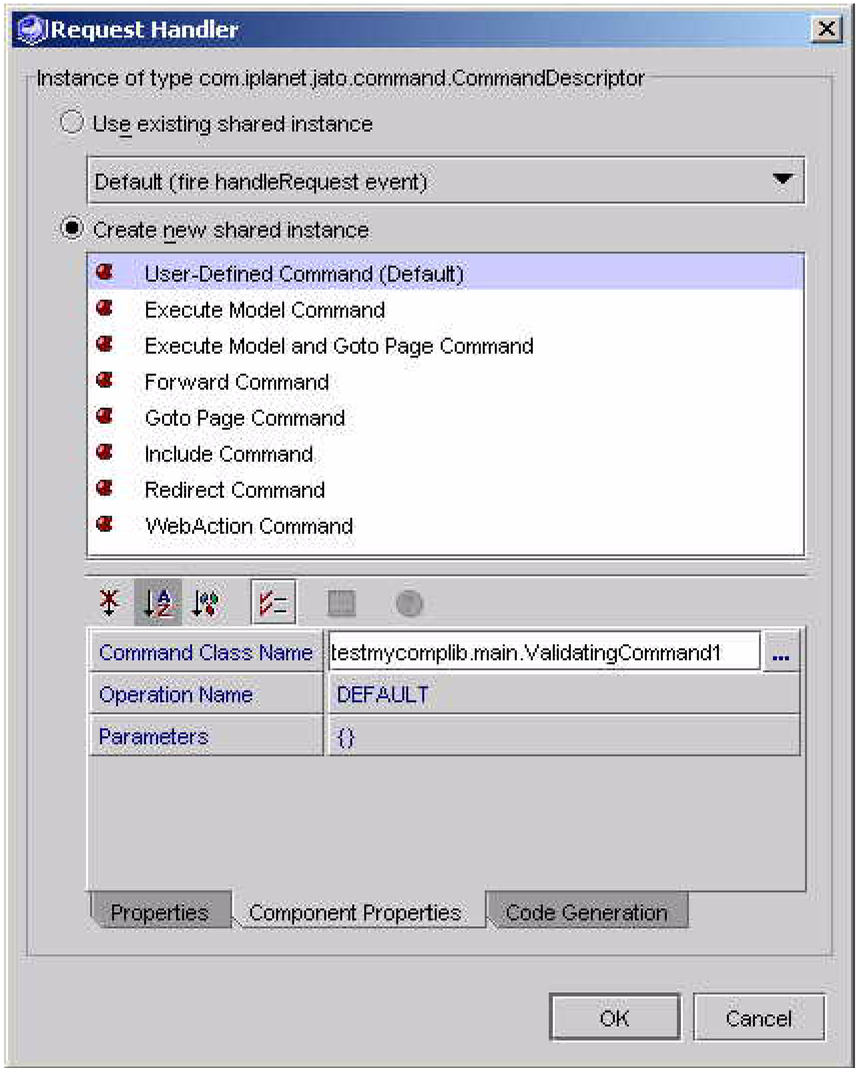


14. **The Command Class Name property is of type java.lang.String, but at this time, instead of directly typing into the exposed property field, select the ellipsis to bring up the full blown editor. You should now see that the Command Class Name property editor is actually the same non-extensible Component browser that we have seen in several other contexts. In this context, though, it intentionally filters the list of components to those which are appropriate for the context (i.e. Command components). Fully expand the Current Application Components node and you should see "ValidatingCommand1" available (below).**



15. Select the "ValidatingCommand1" node from the property editor and click OK (above).

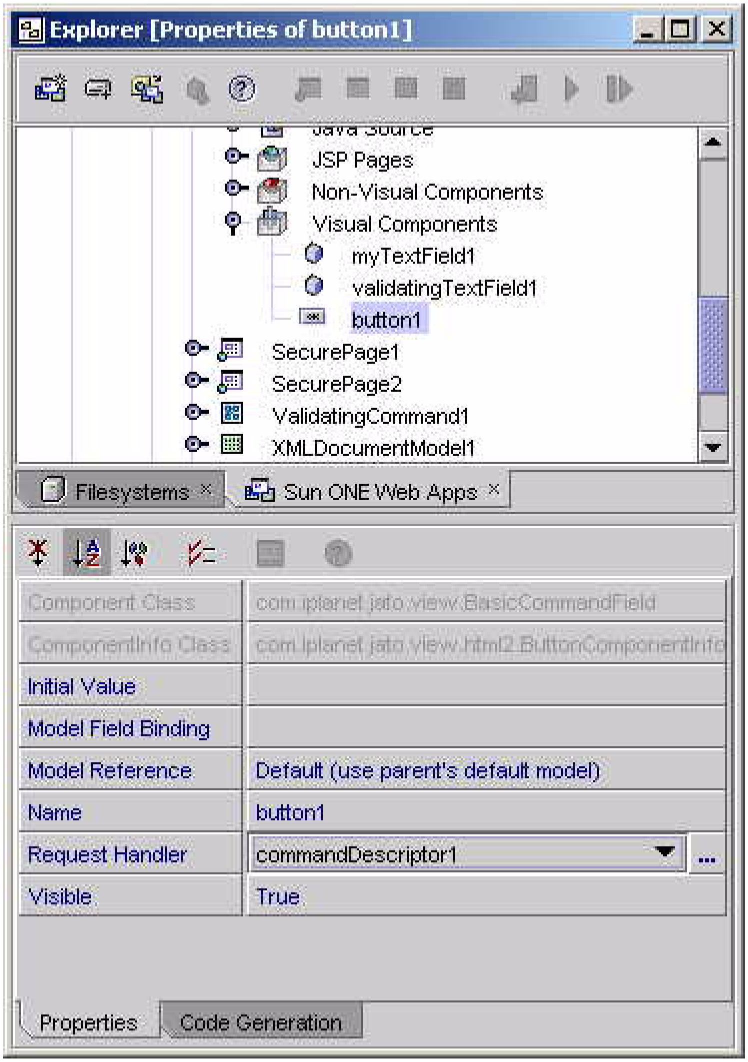
Notice that the Command Class Name property in the CommandDescriptor's embedded property sheet now contains the fully qualified class name for ValidatingCommand1 (below).



You could have directly typed in the fully qualified name of the Command class (e.g. <yourTestApplication>.main.ValidatingTest1) into the Command Class Name property. But direct editing of the Command Class Name property is really only recommended for special cases (e.g. where you need to refer to a Command Class that exists only as a .class file and is therefore not visible for direct selection in the Command Component browser we saw above.

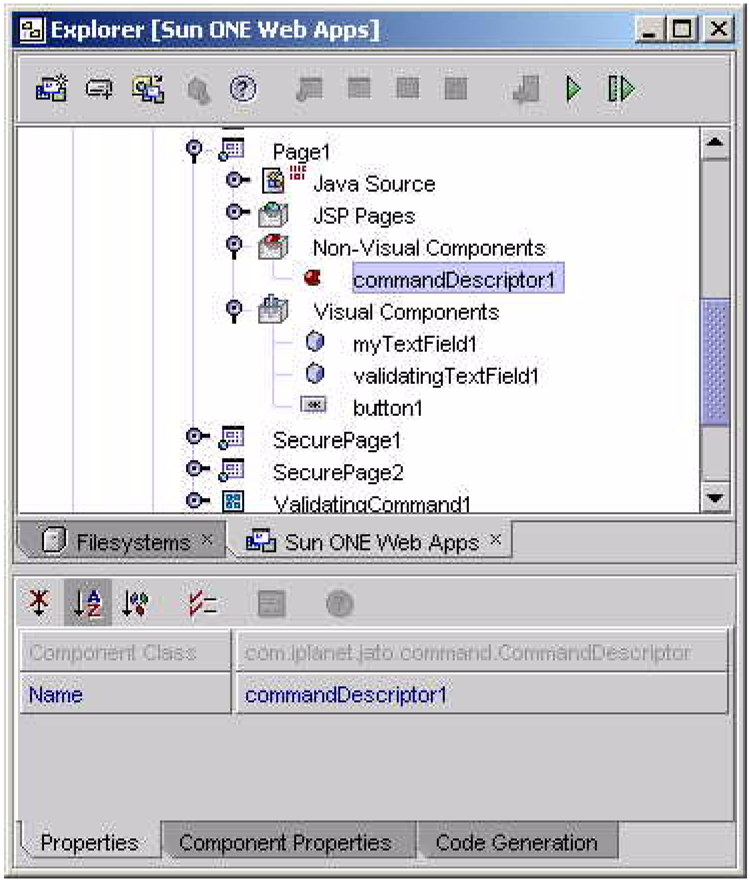
16. **Finally, click the OK button to complete the configuration of the CommandDescriptor.**

17. Before further configuration, spend a moment to fully understand the impact of the previous configuration on the Page1. First, note that the value of button1's Request Handler property now reads "commandDescriptor1" (below).



18. Naturally, you should ask yourself, "where is commandDescriptor1?". Also you are probably still wondering about that "Create new shared instance" radio button in the CommandDescriptor property editor that you selected (above). The answer to both of those mysteries is revealed by expanding the Page1's Non-Visual Components node (below). There you will find a new non-visual component node "commandDescriptor1". It is the CommandDescriptor object you configured just

moments ago. It is a CommandDescriptor that you configured as a "shared instance". In order to visually express its "shared" nature, the IDE toolset provides the "Non-visual Components" node to house all of these shared instances.

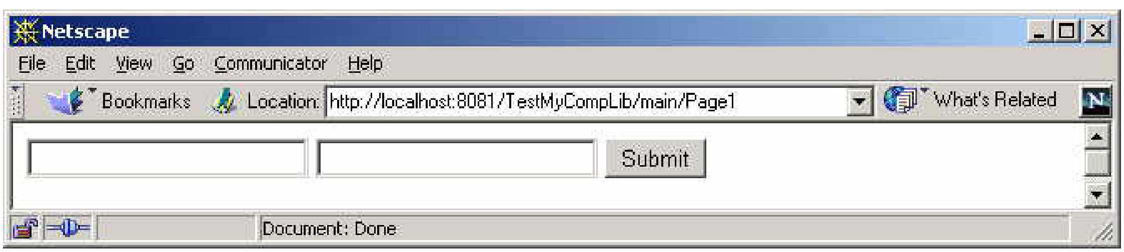


The "Non-Visual Components" node is an Sun ONE Application Framework component model construct. It provides a ContainerView scoped space for the configuration of JavaBean objects (i.e. Configured Beans) which are referenceable by properties elsewhere in the current ContainerView. In this example, "commandDescriptor1" is a configured CommandDescriptor (which is a JavaBean), which is referenced by button1's Request Handler property. The key to this component model feature, is that the same configured non-visual component may be referenced by more than one property within the current ContainerView scope (e.g. more than one button or HREF could have its CommandDescriptor property also set to refer to commandDescriptor1. A quick glance at the IDE toolset generated Java code for Page1 would reveal how this is expressed in Java terms. The benefits to both component authors and application developers are substantial. As a further clarification, not only may multiple CommandFields (for example, Buttons, HREFs, and so on) share commandDescriptor1 by each referring to it in their specific "Command Descriptor" property, but any property within the current ContainerView whose type is assignable from com.iplanet.jato.command.CommandDescriptor may be set to refer to commandDescriptor1. In essence, the non-visual components are class scoped, IDE configurable JavaBean objects which may be referenced in a type safe manner by any number of other visual components within the class. At this time the IDE toolset does not support the ability to have one non-visual component refer to another non-visual component.

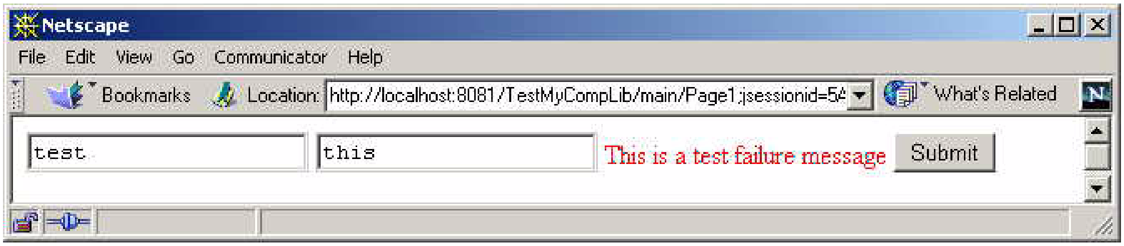
Also note, application developers must understand and respect the shared nature of the non-visual components. Modifications to the configuration of an existing non-visual component will indirectly affect all circumstances in which that instance of the non-visual component is referenced at run-time.This is precisely why the CommandDescriptor editor (or more generally, the non-visual component editor) always allows one to "Create a new shared instance" of a non-visual component. More often than not, multiple CommandFields within a given ContainerView will not share the same instance of CommandDescriptor, but rather, refer to different and distinctively configured instances of CommandDescriptor.

19. **Now you have configured Page1 to instantiate ValidatingCommand1 and invoke its execute method whenever button1 is indicated in a form submit. Test run the page flow from Page1 to SecurePage1 which is now controlled by ValidatingCommand1.**

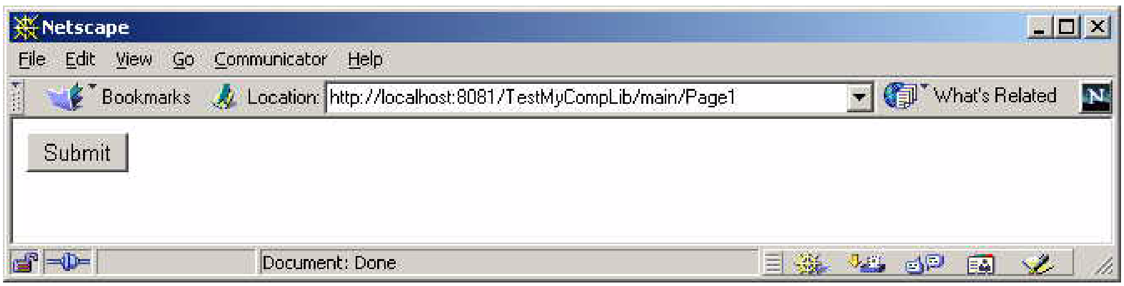
 - Test run Page1
 - The contents of Page1.jsp should show up in the browser



20. **Enter an invalid value (any non-integer) in the ValidatingTextField text input and submit the page. The page should immediately be redisplayed with the text of the "Validation Error Message" property immediately following the ValidatingTextField.**



21. Enter a valid value (e.g. 55 or any other valid Integer) and submit the page. Now instead of Page1 being redisplayed as it was earlier in this guide, the logic within ValidatingCommand1 will display SecurePage1.



CHAPTER 6

ConfigurableBeans (Non-Visual Components)

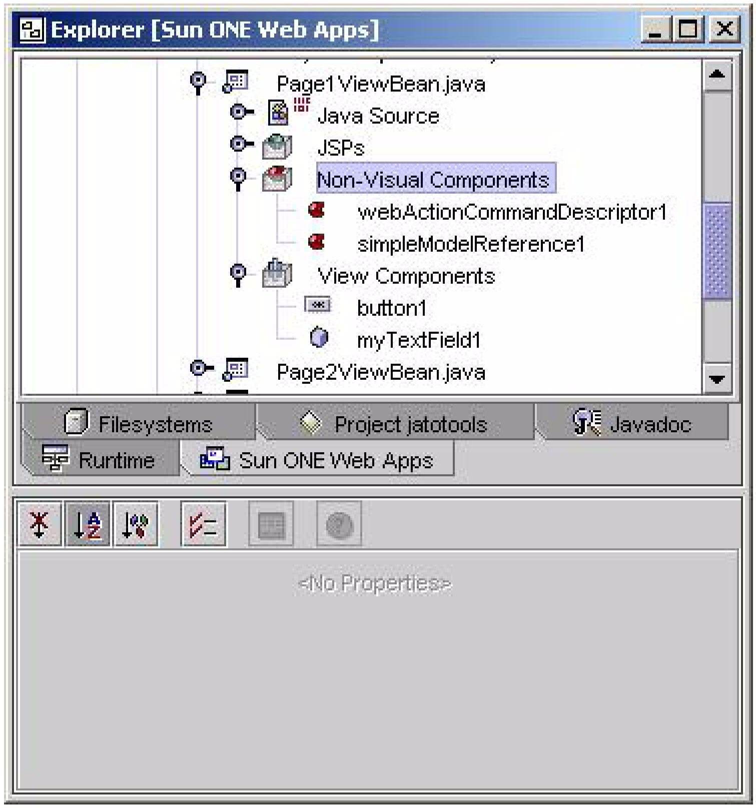
ConfigurableBeans (Non-Visual Components)

ConfigurableBeans are JavaBean types which have been explicitly designated as ConfigurableBeans in a component library manifest. The IDE toolset automatically inspects all component library manifests and builds a dynamic list of ConfigurableBean types in memory. After the component manifest has been inspected these types are said to be registered with the IDE toolset.

Here is a snippet of the Sun ONE Application Framework Component Library manifest that declares some ConfigurableBeans. As you can see, the designation is very straightforward.

```
<configurable-bean>
       <bean-class>com.iplanet.jato.model.SimpleModelReference</bean-class>
</configurable-bean>
<configurable-bean>
       <bean-class>com.iplanet.jato.command.CommandDescriptor</bean-class>
</configurable-bean>
<configurable-bean>
       <bean-class>com.iplanet.jato.view.command.WebActionCommandDescriptor</bean-class>
</configurable-bean>
```

First, we must note that the technical name for these components is ConfigurableBeans. That is the name by which these entities are declared within the component library manifest. However, within the IDE toolset, application developers see the more developer friendly term Non-Visual Components.



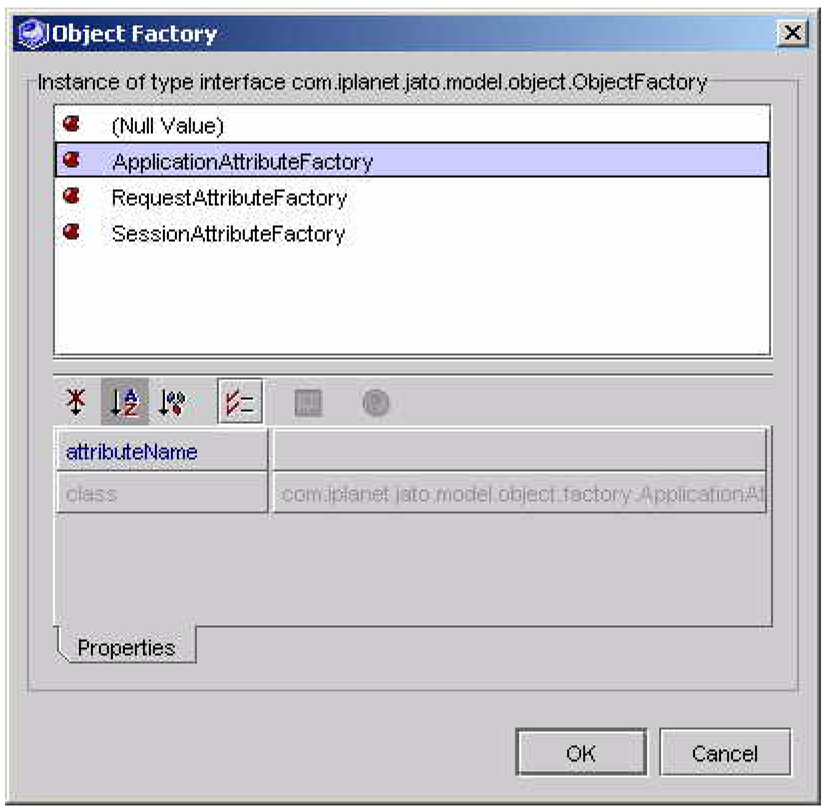
Non-Visual Components node

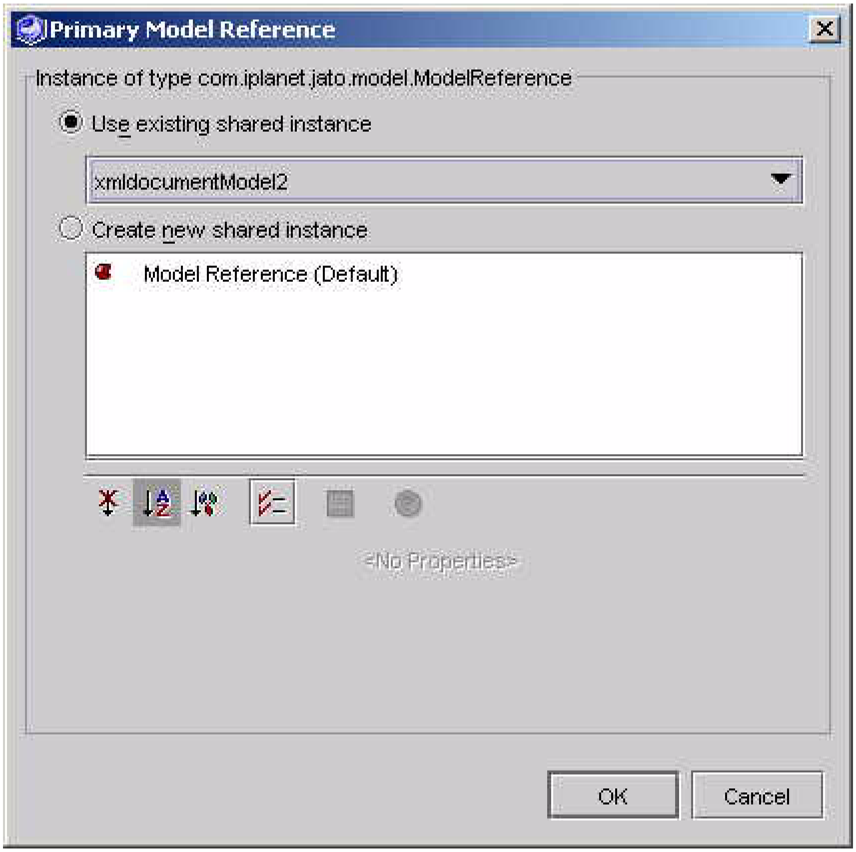
Only component authors need understand that ConfigurableBeans and Non-Visual Components are essentially the same thing. Technically speaking, the Non-Visual Components which are visible as a sub-node of a ContainerView are really just a special case of the IDE toolset exposing ConfigurableBeans as nodes. So formally speaking, all Non-Visual Components are ConfigurableBeans, but not all ConfigurableBeans are Non-Visual Components. There are in fact other cases of ConfigurableBeans being used within the IDE toolset which do not appear as explicit nodes. See ConfigPropDescriptors API - Value Policy.

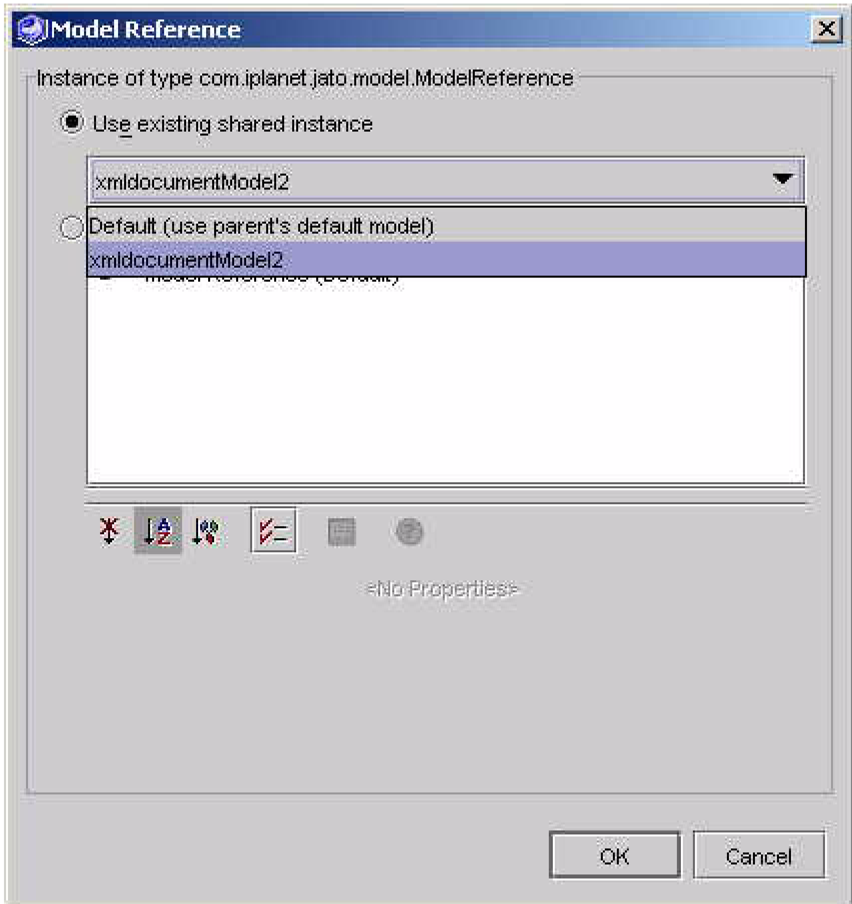
How does the IDE toolset make use of the ConfigurableBean? What role do they play?

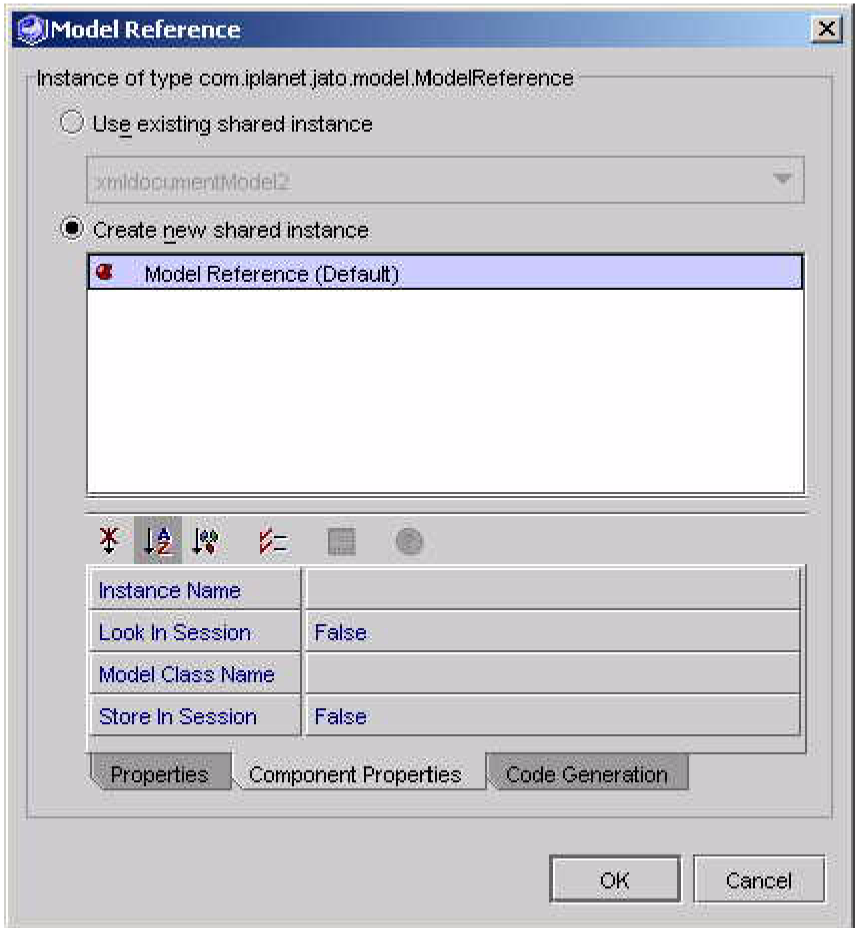
It is fair to say that ConfigurableBeans are just ordinary JavaBean types which play a well defined but subtle role within the Sun ONE Application Framework IDE toolset. The Sun ONE Application Framework component model relies on ConfigurableBeans to complement the standard Sun ONE Application Framework components (Model, View, Commands). Specifically, ConfigurableBeans complete the story begun by Sun ONE Application Framework ConfigPropertyDescriptors. A component author adds ConfigPropertyDescriptors to ComponentInfo whenever he needs to specify a configuration property that he wishes to expose for design time configuration. Each ConfigPropertyDescriptor specifies a property "type". Application developers must edit/configure these properties within the IDE. Since the properties are typed, the IDE toolset can leverage this formalism, and provide a type specific editor. For example, if the configuration property type is Boolean.TYPE then the IDE will invoke the standard Boolean editor. This behavior is typical of any JavaBean aware IDE.

However, the Sun ONE Application Framework IDE toolset offers functionality above and beyond that of the standard JavaBean editor. This extra functionality involves the special treatment that the IDE toolset provides for Sun ONE Application Framework configuration properties whose property types correspond to ConfigurableBean designated types. The IDE toolset uses a lookup algorithm to determine if the configuration property type corresponds to a registered ConfigurableBean type and if so it automatically invokes one of two special ConfigurableBean editors. These special Sun ONE Application Framework ConfigurableBean editors are shown below.




Primary Model Reference
Instance of type com.iplanet.jato.model.ModelReference
Use existing shared instance
xmldocumentModel2
Create new shared instance
Model Reference (Default)
<No Properties>
OK
Cancel


Model Reference
Instance of type com.iplanet.jato.model.ModelReference
Use existing shared instance
xmldocumentModel2
Default (use parent's default model)
xmldocumentModel2
<No Properties>
OK
Cancel



The IDE toolset will invoke one of the two editors above based on a further subtlety in the ConfigPropertyDescriptor, known as the value policy. The details of value policy are beyond the scope of this section, for more information see ConfigPropDescriptors API - Value Policy. For this section, it is sufficient to observe that while their layout is radically different both of the ConfigurableBean editors provide a common core functionality. Both of these editors provide the application developer with a dynamic list of ConfigurableBean types which are assignable from the configuration property type. That is the key value add of the ConfigurableBean component. It is this mechanism which allows the IDE to seamlessly and dynamically incorporate new choices into properties that otherwise would normally be severely restricted. For instance, the IDE will enable the editing of a configuration property of type CommandDescriptor with the ConfigurableBean editor that displays a dynamic list of CommandDescriptor sub-types. The application developer first selects the type of CommandDescriptor from the list, and then configures an instance of that type. The properties of the selected sub-type, of course, are dynamically exposed via conventional JavaBean logic.

Instead of being limited to a very plain vanilla CommandDescriptor editor which would be the case if left to the standard JavaBean handling, the Sun ONE Application Framework IDE toolset provides an unlimited opportunity for component authors to introduce custom ConfigurableBean types with their own sets of properties and potentially custom property editors. The IDE then transparently leverages these type/property sheet/editor combinations into the IDE as new offerings for simply defined properties. Effectively, the ConfigurableBean editor introduces an extra level of indirection that is extremely powerful and somewhat unprecedented. It is so unprecedented that it may take component authors some time to actually fully appreciate the opportunity that this offers them.

What is the relationship between Sun ONE Application Framework and the ConfigurableBean types?

ConfigurableBeans are really a value added feature of the Sun ONE Application Framework component model. Note that component authors are not required to utilize the ConfigurableBean feature. There is no formal notion of ConfigurableBean in the Sun ONE Application Framework run-time environment, or framework API. Rather, ConfigurableBeans are a feature offered by the component model to empower component authors and make the IDE experience richer for developers. Component authors are encouraged to come up with new ConfigurableBean types to either augment existing components, or enhance entirely new components.

The Sun ONE Application Framework Component Library does define quite a few ConfigurableBean types. Component authors should familiarize themselves with the usage of these ConfigurableBean types, as they provide the best illustration of the feature. Component authors should understand the manner in which the Sun ONE Application Framework standard components declare configuration properties which are satisfied by ConfigurableBean types. In addition to writing new components, component authors should understand that they can also immediately augment the existing Sun ONE Application Framework components by providing additional ConfigurableBean types that are appropriate for the already defined Sun ONE Application Framework configuration properties identified below.

Following is a table to help guide your review.

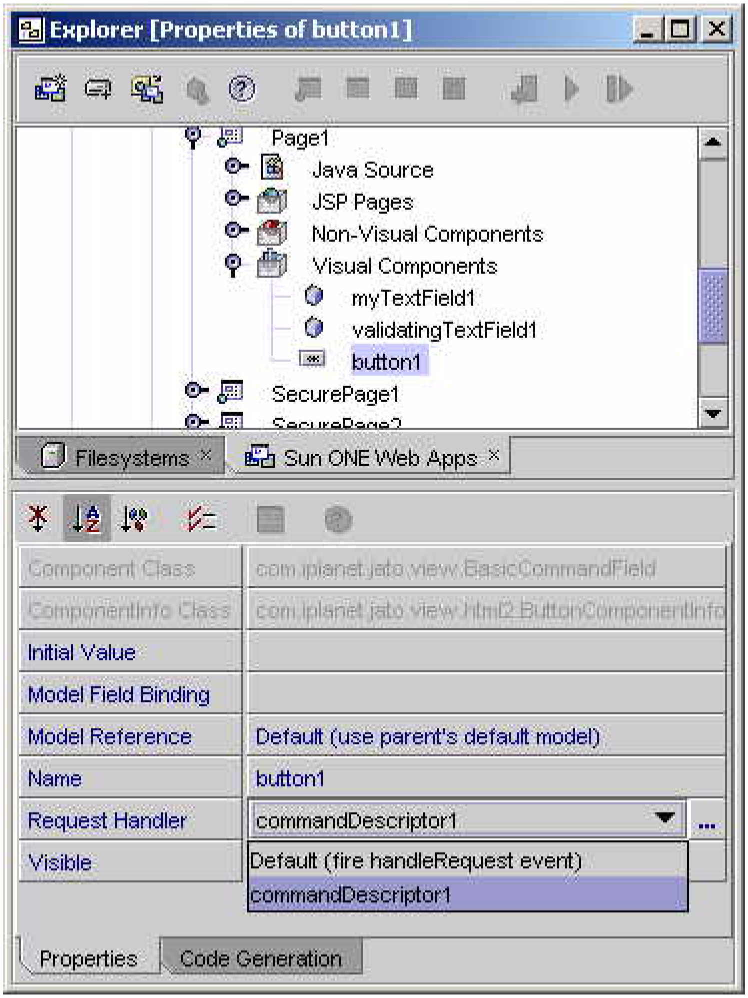
ComponentInfo	ConfigPropertyDescriptor	ConfigBeans Assignable From Property Type
BasicDisplayFieldComponentInfo	ConfigPropertyDescriptor("modelReference", com.iplanet.jato.model.ModelReference.class)	com.iplanet.jato.model.SimpleModelReference
BasicCommandFieldComponentInfo	ConfigPropertyDescriptor("commandDescriptor", com.iplanet.jato.command.CommandDescriptor.class)	com.iplanet.jato.command.CommandDescriptor com.iplanet.jato.view.command.WebActionCommandDescriptor com.iplanet.jato.view.command.ExecuteModelCommandDescriptor com.iplanet.jato.view.command.GotoViewBeanCommandDescriptor
ObjectAdapterModelComponentInfo	ConfigPropertyDescriptor("objectFactory", com.iplanet.jato.model.object.ObjectFactory.class)	com.iplanet.jato.model.object.factory.SessionAttributeFactory com.iplanet.jato.model.object.factory.ApplicationAttributeFactory com.iplanet.jato.model.object.factory.RequestAttributeFactory
BasicChoiceDisplayFieldComponentInfo	IndexedConfigPropertyDescriptior("choices",com.iplanet.view.Choice.class)	com.iplanet.jato.view.SimpleChoice

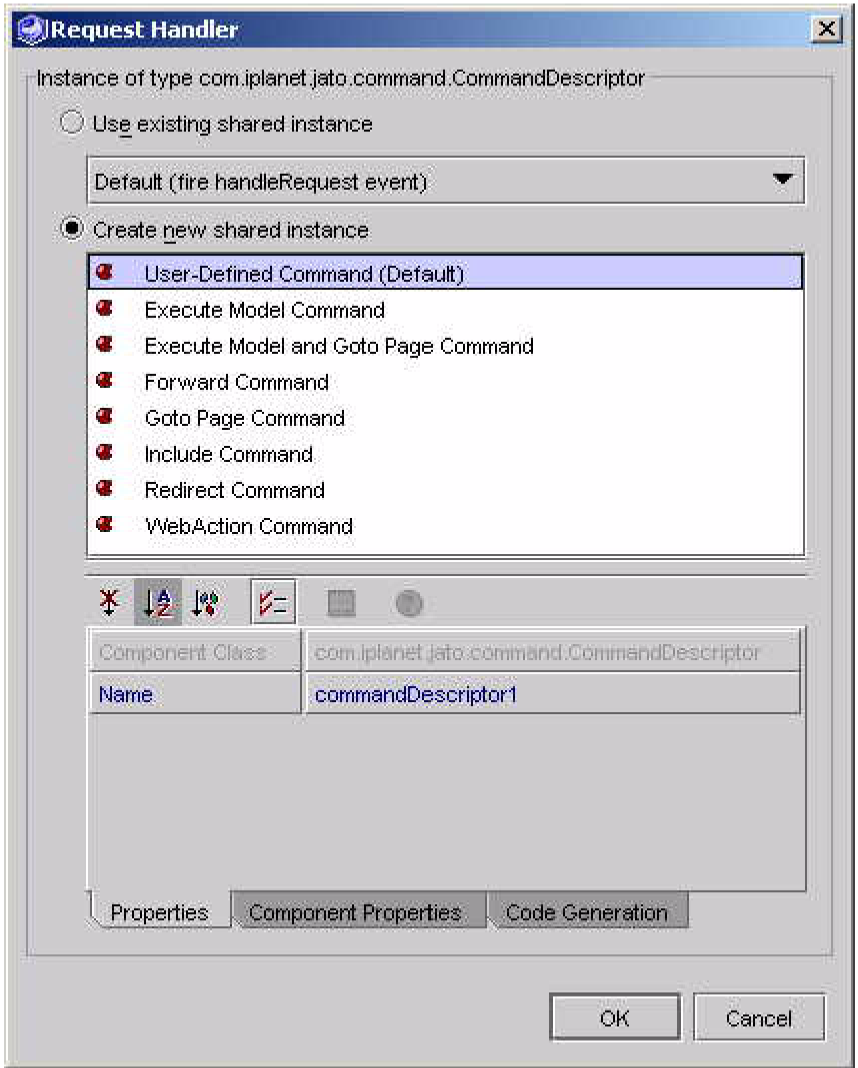
Configurable Bean Example: CommandDescriptor

The obvious Command components are the extensible Command components. Extensible Command components are custom implementations of the com.iplanet.jato.command.Command interface, which are intended for specialization by application developers. The specialization by application developers will usually consist of application developers adding custom logic to their application specific Commands. Command objects have minimal formal structure, being arbitrary implementations of a very simple interface, com.iplanet.jato.command.Command. Therefore, there is not as much of an opportunity to formalize the construction of new Command types within the IDE beyond the specification of properties.

Additionally, the Sun ONE Application Framework offers other Command component opportunities. To understand this opportunity, it is necessary for the component author to fully understand the formal role of CommandDescriptors. Effectively, CommandDescriptors are configurable beans that allow for the design-time configuration of Command object instances.

The Sun ONE Application Framework and the IDE toolset utilize CommandDescriptors to allow the application developer to configure the usage of Command objects. That is to say, there is a formal uses relationship between CommandFields and Command objects and this relationship is mediated by CommandDescriptors. CommandFields are Views (e.g. Buttons and HREFs) which invoke Command objects when activated. The application developer specifies which Command object will be invoked when the CommandField is activated via the field's Command Descriptor property. A CommandDescriptor is a Sun ONE Application Framework object that encodes the information needed at run-time to construct a particular instance of a Command class and invoke it. Minimally, the CommandDescriptor specifies which Command class should be instantiated at run-time. The CommandDescriptor also allows developers to specify Command specific parameterized values. For instance, it is very common for a single Command object to be associated with multiple CommandFields. Each CommandField would employ a distinctly configured CommandDescriptor to direct and influence the execution of the Command. For more detailed information on this subject, see the *Sun ONE Application Framework Developer's Guide*.


Explorer [Properties of button1]
Page1
Java Source
JSP Pages
Non-Visual Components
Visual Components
myTextField1
validatingTextField1
button1
SecurePage1
Filesystems
Sun ONE Web Apps
Component Class
com.iplanet.jato.view.BasicCommandField
ComponentInfo Class
com.iplanet.jato.view.html2.ButtonComponentInfo
Initial Value
Model Field Binding
Model Reference
Default (use parent's default model)
Name
button1
Request Handler
commandDescriptor1
Visible
Default (fire handleRequest event)
commandDescriptor1
Properties
Code Generation



Given the role of the CommandDescriptor, the opportunity exists for component authors to create a very rich Command component story through the combination of non-extensible Command components and component specific CommandDescriptor classes. For instance, a component author can create and distribute a non-extensible Command component plus a custom CommandDesciptor class designed to allow developers to visually configure the invocation of the non-extensible Command component. The custom CommandDescriptor, itself, can be distributed as a ConfigurableBean component. ConfigurableBeans are visually exposed by the IDE toolset as Non-Visual Components.

CHAPTER 7

Developing and Distributing Non-Extensible Model, Command and ContainerView Components

Recall that there is a fundamental difference between extensible and non-extensible components, see the earlier section Extensible vs. Non-Extensible Components. In the previous exercises we demonstrated the development, distribution and test cycle for five components. All but two of those were extensible components.

- MyTextField (non-extensible DisplayField component)
- ValidatingDisplayField (non-extensible DisplayField component)
- SecureViewBean (extensible ViewBean component)
- XMLDocumentModel (extensible Model component)
- ValidatingCommand (extensible Command component)

What about the possibility of developing and distributing non-extensible Model, ContainerView and Command components?

The short answer is that such components are possible, easy to develop and easy to distribute. A non-extensible Model, ContainerView or Command component is a concrete Model, ContainerView or Command that has been created within the IDE from an extensible Model, ContainerView or Command component. It is no different from an application specific Model, ContainerView or Command, except that is subsequently distributed in a component library JAR file with the express purpose of being incorporated into multiple applications, like any other distributed component. The distribution technique is common for non-extensible Models, ContainerViews and Commands.

Why would someone develop and distribute non-extensible Model, ContainerView and Command components?

Non-extensible Model, ContainerView and Command components provide several opportunities for component authors to deliver highly leveraged components to their component consumers. The discussion which follows, only scratches the surface of this topic. The most obvious opportunity provided by non-extensible Model, ContainerView and Command components is the opportunity for component authors to move beyond delivering small building blocks to large reusable application and organization sized components.Non-extensible Model, ContainerView and Command components typically form the top end of the component food chain. They allow component authors to deliver arbitrarily complex, very coarse grained components. If you consider DisplayField components to be the most fine grained components, the non-extensible Model, ContainerView and Command components are at the opposite end of the component spectrum. Companies or organizations can create very sophisticated horizontal or vertical libraries of non-extensible Model, ContainerView and Command components from which application developers can assemble applications out of very large, very reusable, very powerful building blocks.

For instance, non-extensible Model components can provide pre-packaged ready to use access to specific organizational data. Application developers can then simply define new Views and visually bind these applications specific Views to the pre-packaged Model. Non-extensible ContainerView components can deliver pre-configured visual building blocks comprised of arbitrarily complex aggregations of smaller Views. Non-extensible Command components can provide plug and play behavior. Non-extensible components can be preconfigured to use other non-extensible components within the library. For example, a ShoppingCart ContainerView component can be pre-configured to use a companion non-extensible Model component. Together, such preconfigured ContainerView and Model components provide ready to use already integrated visual presentation and data access. On top of that, the component author could preconfigure said ContainerView component to use one or more non-extensible Command components, thereby adding already integrated command behavior to the composite. We hope you can get the picture. Organizations can create toolboxes comprised of collections of integrated non-extensible Model, ContainerView and Command components. These toolboxes can be used internally to facilitate the rapid development of applications or even delivered to partners as part of a broader business to business architecture. The opportunity is really boundless.

Develop a Non-Extensible Model, ContainerView or Command Component

1. **Develop the component within the Sun ONE Application Framework IDE as you would an ordinary Sun ONE Application Framework application object.**
   - Use the appropriate Model, ContainerView or Command wizard to construct the component.
   - Configure the component's properties

- Add arbitrary behavior to the component's Java class
- (ContainerView components only) Optionally add zero or more child view components to the ContainerView.
- (ContainerView components only) Optionally associate zero or more JSP pagelets, to provide the rendering specification for the ContainerView component.
- (Model components only) Optionally add zero or more model fields or model operations to the Model.

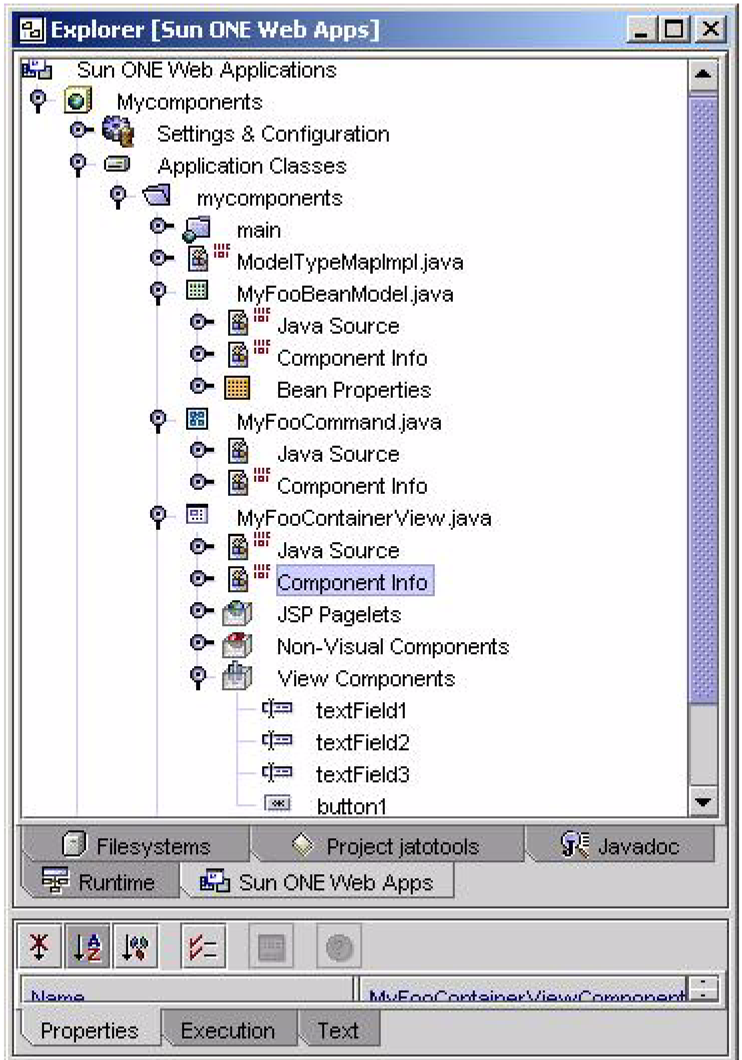
2. **Create a ComponentInfo class for the new component.**

   This is highly recommended, though strictly speaking not required. A component specific ComponentInfo is recommended to minimally provide a component specific ComponentDescriptor. Optionally, the ComponentInfo can be used to specify any and all of the advanced component model features which are appropriate for non-extensible components.

   Special Note - In order to visually create any non-extensible Model, ContainerView or Command components within the Sun ONE Studio, you must construct the components within the context of an Sun ONE Application Framework application. That is to say, you must, first create an Sun ONE Application Framework application before you can leverage the IDE toolset to create any new non-extensible Model, ContainerView or Command components. This is because the IDE toolset does not currently support a "library only design" mode. Future versions of the Sun ONE Application Framework IDE toolset may allow developers to choose between a "new library" or a "new application", but currently, you can only leverage the Sun ONE Application Framework new object wizards within an Sun ONE Application Framework application. The fact that a component author will design these new components within an "application" has no bearing on the future independence of the components. The application merely provide the IDE toolset recognized context that allows the component author to leverage the complete visual IDE feature set during the authoring process. The ultimate end product of the component authoring will be the component's Java resources which are totally independent of the application in which they might have been originally visually designed.

   Assuming that you intend to create a new Sun ONE Application Framework application merely for the purposes of designing some new non-extensible Model, ContainerView or Command components, the name of the application itself does not matter. One recommended approach is to consider the application as a convenient "test application" for your new non-extensible Model, ContainerView or Command components. Then after you are satisfied with their performance within the test application you add the new components to a component library JAR file.

   The IDE Toolset will treat a Sun ONE Application Framework component's ComponentInfo Java source as "part" of the component. This means that the ComponentInfo Java source node will appear as a child of the component's primary node, just as the component's Java source appears as a child of the component's primary node.


Explorer [Sun ONE Web Apps]
Sun ONE Web Applications
Mycomponents
Settings & Configuration
Application Classes
mycomponents
main
ModelTypeMapImpl.java
MyFooBeanModel.java
Java Source
Component Info
Bean Properties
MyFooCommand.java
Java Source
Component Info
MyFooContainerView.java
Java Source
Component Info
JSP Pagelets
Non-Visual Components
View Components
textField1
textField2
textField3
button1
Filesystems
Project jatotools
Javadoc
Runtime
Sun ONE Web Apps
Properties
Execution
Text

Distributing a Non-extensible Model, ContainerView or Command Component

1. **Add a component element to the component library's complib.xml with one additional sub-element not discussed previously**
   - The component element must include a design-reference-resource sub-element
   - The design-reference-resource sub-element must specify the location of the component's object definition resource
2. **Add the non-extensible component to a Sun ONE Application Framework component library JAR**
   - Include any classes and other component specific resources as you would for any Sun ONE Application Framework component
   - Include the component's object definition file. This is the key distinction as this is not required for the other components discussed n this guide
   - Include any special "Additional Files" resources. See below Unpacking of "Additional Files"

     Common additional file resources will include ContainerView components' associated JSP pagelet files

   Here is an example of a non-extensible component entry in a complib.xml

```
<component>
         <component-class>mycomponents.MyFooCommand</component-class>
         <component-info-
class>com.iplanet.jato.command.BasicCommandComponentInfo</component-info-class>
         <design-reference-resource>/mycomponents/MyFooCommand.command</design-reference-
resource>
</component>
```

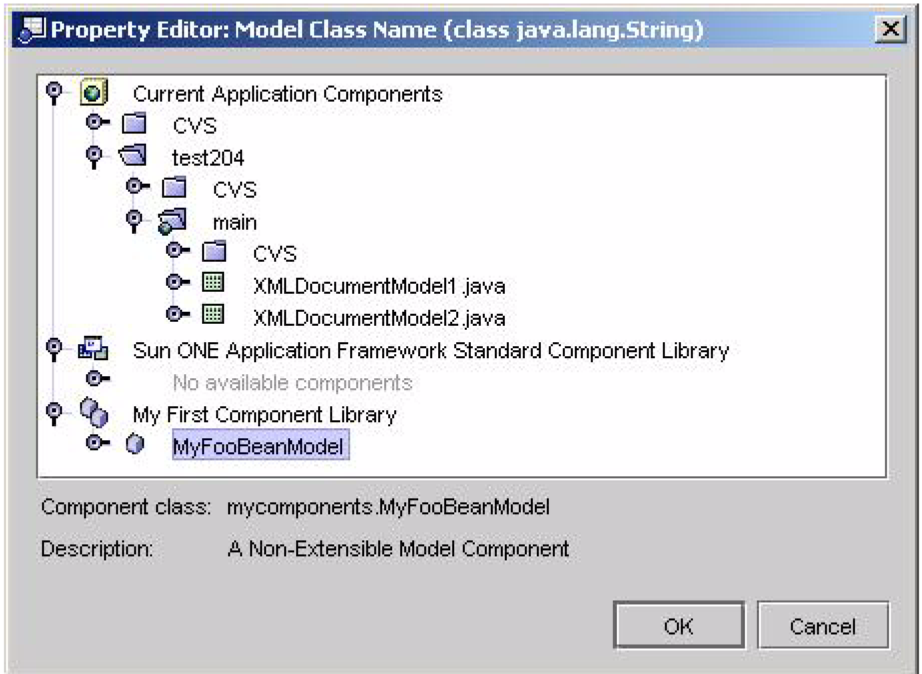
The emphasis on non-extensible Models, ContainerViews and Commands being created within the IDE is intentional and important. All preceding component examples in this guide did not assume or require that the component types themselves be developed inside of the Sun ONE Studio. The visual use of the components within the IDE did require the use of the Sun ONE Application Framework enabled Sun ONE Studio, but the authoring of the component classes, ComponentInfo, complib.xml, and the preparation of the component library JAR files did not assume or require the use of the Sun ONE Studio. However, with non-extensible Models, ContainerViews, and Commands that is not the case. They must be developed within the IDE because it is only the IDE which can generate the non-extensible component metadata, called the object definition file. In short, the key to distributing a non-extensible Model, ContainerView, or Command is to distribute its object definition file along with the class and ComponentInfo.

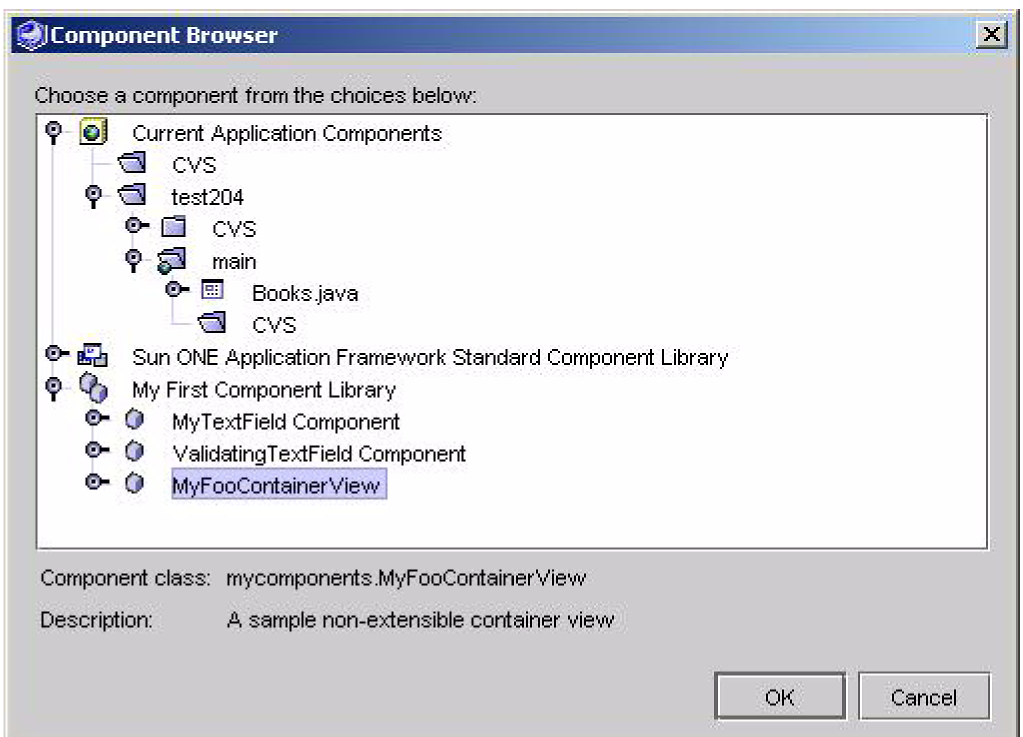
Can a component author mix non-extensible Model, Command and ContainerView components in the same library with extensible Model, Command and ContainerView components?

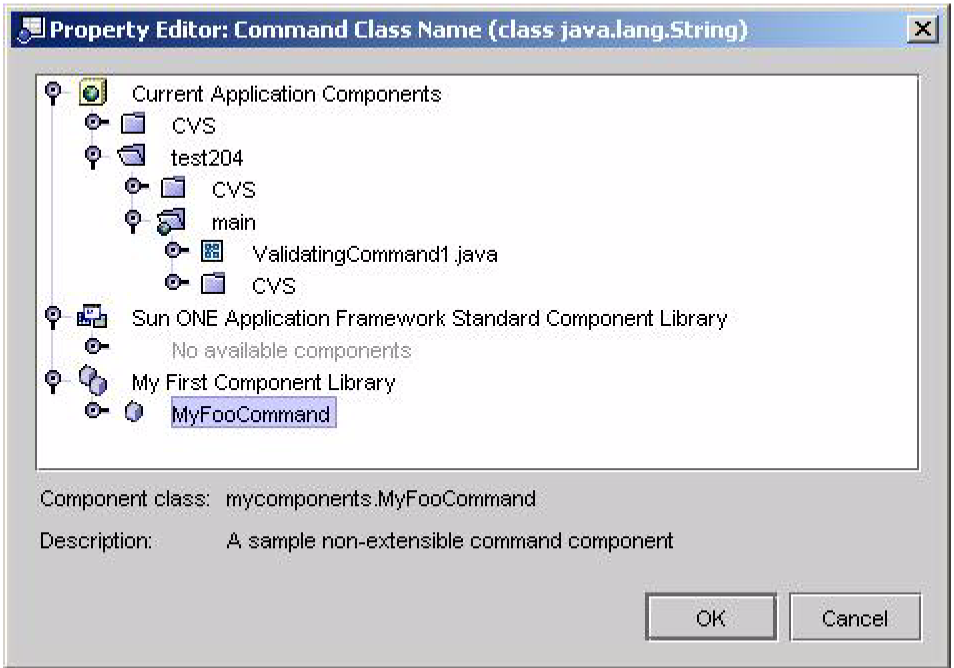
Absolutely yes. This is expected and encourage.

After non-extensible Model, Command and ContainerView components have been distributed in a Sun ONE Application Framework component library, how do application developer's make use of the components?

The non-extensible Model, Command and ContainerView components within a given Sun ONE Application Framework component library will appear in the IDE toolset in exactly the same IDE contexts that the current application's non-extensible components appear. If this sounds tautological, it is intentionally so. Its points out that from the Sun ONE Application Framework IDE toolset's perspective all Model, View and Command objects are components. As far as the IDE toolset is concerned it does not matter whether it discovers the components in a component library JAR or within the current application space. The images below will demonstrate this point by showing that the various component choosers within the IDE toolset allow the application developer to choose freely and transparently between library supplied non-extensible components and application defined non-extensible components.


Property Editor: Model Class Name (class java.lang.String)
Current Application Components
CVS
test204
CVS
main
CVS
XMLDocumentModel1.java
XMLDocumentModel2.java
Sun ONE Application Framework Standard Component Library
No available components
My First Component Library
MyFooBeanModel
Component class: mycomponents.MyFooBeanModel
Description: A Non-Extensible Model Component
OK
Cancel


Component Browser
Choose a component from the choices below:
Current Application Components
CVS
test204
CVS
main
Books.java
CVS
Sun ONE Application Framework Standard Component Library
My First Component Library
MyTextField Component
ValidatingTextField Component
MyFooContainerView
Component class: mycomponents.MyFooContainerView
Description: A sample non-extensible container view
OK
Cancel



The Object Definition File (non-extensible component metadata)

Suffice to say that both component authors and application developers should understand that the object definition files are produced by the IDE toolset and should be treated as first class application resources.

- They should be preserved in source code control systems along with the conventional application resources.
- They should not be edited by hand.
- They have zero run-time value and need not be deployed to the servlet container
- They do need to be distributed with non-extensible Model, ContainerView and Command components

The following details of the object definition files are provided for information purposes only. These are implementation details of the Sun ONE Application Framework component model that neither component authors nor application developers are required to know

The Sun ONE Application Framework IDE toolset stores design-time state in XML format within its object definition files. The term object definition file is an arbitrary designation for these files. The object definition files:

- Are XML files that conform to the Sun ONE Application Framework Model, ContainerView and Command DTDs.
- Are the authoritative representation of Sun ONE Application Framework design-time state
  - They are produced by the IDE toolset to capture the application developer design-time decisions (i.e. object hierarchy declarations property configurations)
  - They are read by the IDE toolset to restore the design-time state across Sun ONE Studio sessions
  - The IDE toolset generates Java code within the component's Java source file which is a Java equivalent of the design-time state stored in the object definition file. This code is generated into Sun ONE Studio Java source code editor protected blocks (i.e. non-editable blocks).
- Have file suffixes recognized by the Sun ONE Application Framework IDE toolset .model, .viewbean, .cview, and .command
- Have no run-time value at all

  The IDE toolset generates Java code within the application class which is a Java equivalent of the design-time state stored in the object definition file. Therefore, the object definition files have absolutely no run-time role or presence.

CHAPTER 8

Design Actions

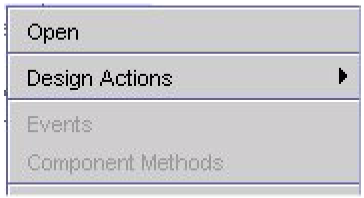
This section assumes that you have already read Develop Your First Component found in Chapter 2, "Developing Components" on page 17, and "Developing Model Components" on page 85.

Developing Extensible Components which have Component Design Actions

This section of the document provides a description of the basic steps involved in adding design actions to your extensible component. Component design actions encapsulate arbitrary design time behavior for a component. The design action may post an "About" or "Credits" dialog, perform validation, autoconfigure the component, synchronize support files, display messages or warnings, open complex editors and wizards and even read and edit the finest details of the component object model. In order to support component design actions we introduce a new interface to Sun ONE Application Framework component architecture called DesignableComponentInfo. DesignableComponentInfo is an optional specialization of ExtensibleComponentInfo that allows the component author to define special design time behavior for extensible components. The practical use of DesignableComponentInfo is to expose design actions to the developer in the IDE. Any extensible command, model and view ComponentInfo classes may optionally implement DesignableComponentInfo to expose ComponentDesignActionDescriptors.

The IDE module only exposes design actions for extensible components and therefore the DesignableComponentInfo specializes ExtensibleComponentInfo to impose this rule.

When available, the IDE uses ComponentDesignActionDescriptors to present a submenu list of the defined design actions in the component node's contextual menu "Design Actions".



Because DesignableComponentInfo is a recent addition to the Sun ONE Application Framework component architecture only the Bean Adapter Model uses DesignableComponentInfo to expose the Update Properties design action.

What is a Component Design Action?

Although a component design action may be used to perform practically anything, here are some guidelines to consider. The standard mechanism a developer expects to use when configuring an extensible component during design time is the property sheet of the primary component node and/or the contained property sheets of subnodes. The component author should first turn to automatic support for config properties and optionally any custom editors which may be applicable for complex config properties. If the component author is using design actions to take the place of config properties or custom editors then he or she may be going in the wrong direction.

Discrete component config property editors alone are often not enough for managing the design aspects of certain components. An editor for a config property will not have scope to other config properties or other parts of the component object model. Therefore, it is not possible to edit a subset of related config properties as a whole. In the case of models and container views there is no way for a collection of model fields or child views to be edit together. Also, there is currently no support in the component architecture to specify wizards or initialization mechanisms to be used by the developer when creating a new component. A solution to these issues is the component design action. Finally, we will see later in this section that while performing a component design action the component author is provided a context in which very powerful design changes (even continuous and re-entrant design changes) may be implemented.

A good example of a component design action may be found in Bean Adapter Model in the Sun ONE Application Framework component library. Although the Bean Adapter Model has a config property "Bean Type" which allows the developer to specify the type/class of the adapted Java Bean, there is no easy way for the developer to automatically generate model fields for the properties of the Bean. The Update Properties design action of the Bean Adapter Model validates the Bean Type config property and ensures that at least the full set of bean properties have representative model fields. We can say that this design action is supports continuous design because the action may be used as the adapted Bean changes.

An opportunity is provided with component design actions for component authors to publish "black box" components which may even be configured by a mechanism which does not even use the component object model. For instance, an existing view or model component or even the foundation of a new component may already exists which uses a custom runtime XML descriptor or properties file. In this case the component author needs a way to provide editors for this custom configuration. Advanced APIs of the ComponentDesignContext support such situations.

The rest of this section assumes that the reader has reviewed the JavaDocs for the com.iplanet.jato.component.design and com.iplanet.jato.component.design.objmodel packages. We will present a simple example of how to specify a ComponentDesignAction for an extensible component using the DesignableComponentInfo and ComponentDesignActionDescriptor APIs.

Exposing Design Action in ComponentInfo

To expose component design actions for your extensible component the first step is to implement DesignableComponentInfo. In the example below we build onto the XMLDocumentModel example. We will create a simple design action called "About" which presents an informational dialog to the developer dumping some component details including name, logical name, and config properties.

1. **have your component info implement DesignableComponentInfo.**

2. **implement method getComponentDesignActionDescriptors() (see code example below)**

3. **because the ComponentDesignActionDescriptor bean requires the ComponentDesignAction class to be assigned, you will also need to create the action class; the minimum that you need is a class implementing ComponentDesignAction including the performAction(ComponentDesignContext) method; an simple technique is to have an inner class of the component info define the action class as is shown in the code example below where we have the static inner class AboutDialog**

   The component design action mechanism will only call the default (no arg) constructor of the design action class. Therefore, avoid using alternate constructors for they simply will have no use.

4. **the performAction() method is not required to do anything other than return promptly; in our minimal example below we will post a modal dialog and call a helper method aboutDisplayMessage()**

   In this sample code we have embedded String values directly for ease of demonstration. If you anticipate the need to localize your display strings, we encourage you to utilize resource bundles. The following code represents what needs to be added to the XMLDocumentModelComponentInfo.java. Non-relevant code has been snipped as represented by the ellipsis " . . ."

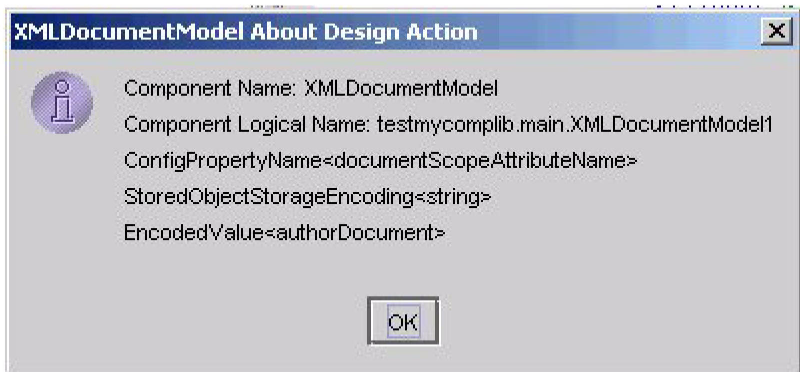
```
. . .
import com.iplanet.jato.component.design.*;
import com.iplanet.jato.component.design.objmodel.*;

public class XMLDocumentModelComponentInfo extends ExtensibleModelComponentInfo
        implements DesignableComponentInfo
{
        . . .

        public ComponentDesignActionDescriptor[] getComponentDesignActionDescriptors()
       {
                if(null != designActionDescriptors)
                        return designActionDescriptors;

                List descriptors=new ArrayList();

                ComponentDesignActionDescriptor descriptor = new
ComponentDesignActionDescriptor(AboutDialog.class);
                descriptor.setName("About");

                descriptor.setDisplayName("About");
                descriptor.setShortDescription(
                        "Displays a small list of component details");
                descriptors.add(descriptor);

                designActionDescriptors = (ComponentDesignActionDescriptor[])
                        descriptors.toArray(
                                new ComponentDesignActionDescriptor[descriptors.size()]);
                        return designActionDescriptors;
                }

                public static class AboutDialog implements ComponentDesignAction
{
                        public void performAction(ComponentDesignContext context)
                                throws DesignActionException
                        {
                                javax.swing.JOptionPane.showMessageDialog(
                                        context.getMainWindow(),
                                        aboutDisplayMessage(context),
                                        "XMLDocumentModel About Design Action",
```

```
                                        javax.swing.JOptionPane.INFORMATION_MESSAGE);
                        }

                        private String aboutDisplayMessage(ComponentDesignContext context)
                        {
                                StringBuffer msg = new StringBuffer(
                                        "Component Name: " +
                                        context.getComponentInfo().getComponentDescriptor(
).getName() +
                                        "\nComponent Logical Name: " +
                                        context.getComponentLogicalName() + "\n");
                                ConfigPropertyNode[] props =
((ConfigPropertyNodeContainer)
                                        context.getPrimaryObjectModel()).getConfigProperty
Node();
                                for(int i=0;i<props.length;i++)
                                {
                                        props[i].dump(msg, "\t");
                                        msg.append("\n");
                                }
                                return msg.toString();
                        }
        }

        . . .

        private ComponentDesignActionDescriptor[] designActionDescriptors;
}
```

In our AboutDialog design action we present a Swing informational modal dialog box to the developer. We use the "MainWindow" property of the ComponentDesignContext to place the Swing visual component. The message we present comes from the helper method aboutDisplayMessage(). Again, we use the various properties of the ComponentDesignContext to acquire information about the component including its name, logical name, and we loop through the config properties and dump their values. In order to access the config properties we take advantage of object model interfaces ConfigPropertyNodeContainer and ConfigPropertyNode. Note that this AboutDialog example may be used on Command, ContainerView, ViewBean, and Model extensible components. This example is not Model specific.

After compiling, packaging and using this component in an application, instances of XMLDocumentModel will provide the design action "About"

The result of performing the design action is:



Component Library Structure

Component Library Overview

Sun ONE Application Framework components are packaged and distributed in ordinary JAR files. The JAR file must contain

- A component library manifest (/COMP-INF/complib.xml)
- Component library specific Java resources (component classes, ComponentInfo classes, resource bundles, component icon images, and any other ancillary files). Any classes (component, ComponentInfo, and any other ancillary files) should be placed in the JAR in accordance with standard Java convention.

Optionally, a Sun ONE Application Framework component library JAR may contain

- A special directory named /webapp

  The contents of the /webapp directory are called the "Additional Files". This is a Sun ONE Application Framework IDE toolset value add feature that allows developers to distribute arbitrary additional files inside their component library jar. The Sun ONE Application Framework Sun ONE Application Framework IDE toolset will "unpack" these additional files into the Web application development environment. See Automated Unpacking of "Additional Files" below

Component Library Structure

The contents of a Sun ONE Application Framework component library JAR must be structured as follows:

```
/COMP-INF/complib.xml
/[Java classes and resources]
/webapp/[additional files intended for IDE toolset design-time auto-extraction]
```

Note: The webapp directory is optional

The Component Manifest

Sun ONE Application Framework requires that each component library JAR contain a special Sun ONE Application Framework component library manifest file. The component library manifest file is a simple XML document that describes the collection of components within the library. A component library manifest may declare any number of components and associated Sun ONE Application Framework component model resources.

The IDE toolset automatically introspects each JAR file mounted in an Sun ONE Application Framework application's `WEB-INF/lib` directory. It specifically looks inside the JAR for the component library manifest file. If the IDE toolset finds a valid component library manifest file in the prescribed location within the JAR file, the IDE toolset will expose any properly declared components for design time utilization within the IDE toolset. If it does not find the component library manifest file in the expected location within the JAR file, or if the component library manifest is invalid, the IDE toolset will not recognize the JAR as a component library.

The component library manifest must comply with the following strict rules

- The component library manifest file must be named complib.xml
- The complib.xml file must be a well formed XML file
- The complib.xml file must comply with the jato-component-library_1_0.dtd (shown below)
- The complib.xml file must be located in the component library JAR's /COMP-INF directory

jato-component-library_1_0.dtd

```
<!--

The component-library element is the root element of the component
manifest

-->

<!ELEMENT component-library (tool-info, library-name, display-name,

                 description?, legal-notice?, icon?, interface-
version, implementation-version,
```

```
                 author-info?, taglib*, component*, extensible-
component*,
                 configurable-bean*)>

<!--
The tool-info elements contains information about the tool
environment this
library was written against
-->
<!ELEMENT tool-info (tool-version)>

<!--
The tool-version element contains the interface version of Sun ONE
Application
Framework/JATO this library targets.  Should be a dot-separated
version number,
for example "2.1.0".
-->
<!ELEMENT tool-version (#PCDATA)>

<!--
The library-name element contains the internal name of the component
library.
This name is expected to be globally unique, and should follow the
standard
Java package naming convention. For example, the library name of the
Sun ONE
Application Framework/JATO component library is "com.iplanet.jato",
the root of
its package structure.
-->
<!ELEMENT library-name (#PCDATA)>

<!--
The display-name element contains a short display name of the library
which
will be presented in GUI tools.
-->
```

```
<!ELEMENT display-name (#PCDATA)>


<!--
The description element is used to contain descriptive text about its
parent
element.
-->
<!ELEMENT description (#PCDATA)>


<!--
The legal-notice element contains legal or copyright text that should
accompany
this library.  This element is meant to provide an additional
opportunity to
keep this information in proximity to the library itself; however it
should not
be considered a sufficient means of conveying licensing terms or other
legally
binding terms to users of the library.
-->
<!ELEMENT legal-notice (#PCDATA)>


<!--
The icon element contains a small-icon and a large-icon element
which specify the location within the Web application for a small and
large image used to represent the Web application in a GUI tool. At a
minimum, tools must accept GIF format images.
-->
<!ELEMENT icon (large-icon?, small-icon?)>


<!--
The large-icon element contains the resource name within the library
of a file containing a large (32x32 pixel) icon image. The resource
name must
follow standard Java resource name syntax, with individual path
elements
separated by forward slashes ("/").
-->
```

```
<!ELEMENT large-icon (#PCDATA)>

<!--
The small-icon element contains the resource location within the
library
of a file containing a small (16x16 pixel) icon image.  The resource
name must
follow standard Java resource name syntax, with individual path
elements
separated by forward slashes ("/").
-->
<!ELEMENT small-icon (#PCDATA)>

<!--
The author-info element contains information on the author(s) of this
library.
-->
<!ELEMENT author-info (author*, info-resource*) >

<!--
The author element contains information about a particular author of
the
library.
-->
<!ELEMENT author (author-name, description?, author-contact?) >

<!--
The author-name element contains the author's full name.
-->
<!ELEMENT author-name (#PCDATA)>

<!--
The author-contact element contains the author's contact information,
usually
an email address.
-->
<!ELEMENT author-contact (#PCDATA)>
```

```
<!--
The info-resource element contains information describing external
informational resources relevant to this library, such as a link to a
publisher's homepage, a public link to API documentation, or a support
email
address.
-->
<!ELEMENT info-resource (info-resource-name, description?, info-
resource-contact?) >

<!--
The info-resource-name element contains an arbitrary descriptive name
of the
resource that can be presented to the user of the library.
-->
<!ELEMENT info-resource-name (#PCDATA)>

<!--
The info-resource-contact elements contains the actual contact
information
for the resource, such as an email address or HTTP link.
-->
<!ELEMENT info-resource-contact (#PCDATA)>

<!--
The interface-version element contains the interface version of this
library,
used to determine interface compatibility of the contained code.  The
version
should be a dot-separated numeric version number, such as "1.0.0".
The version
number can contain as many dot-separated elements as desired.
-->
<!ELEMENT interface-version (#PCDATA)>

<!--
The implementation-version element contains the interface version of
this
```

```
library, used to determine the implementation version of the contained
code.
This version number usually takes the form of a timestamp or build
number.
The version should be a dot-separated numeric version number, such as
"2003.1.31".  The version number can contain as many dot-separated
elements as
desired.
-->
<!ELEMENT implementation-version (#PCDATA)>


<!--
The taglib element declares any JSP tag libraries included in this
library.
Declared tag libraries will be automatically unpacked from the library
and
registered in the web.xml of the application under this URI.
-->
<!ELEMENT taglib (taglib-uri, taglib-resource, taglib-default-
prefix)>


<!--
The taglib-uri element contains a logical URI that will be used to
identify the
tag library within the application.  This URI will be registered to
the declared
tab library descriptor in the application's web.xml file.  Note that
this URI
is purely logical and need not have any relation to the physical
location of
the tag library descriptor file (which will be unpacked into a
physical location
determined solely by the GUI tool).  This URI must match the TaglibURI
property
value in a component's JspTagDescriptor.
-->
<!ELEMENT taglib-uri (#PCDATA)>


<!--
```

```
The taglib-resource element contains the resource name of the tag
library's
taglib descriptor (.tld) file.  The resource name must follow standard
Java
resource name syntax, with individual path elements separated by
forward
slashes ("/").  This file will automatically be extracted and
registered with
the application.
-->
<!ELEMENT taglib-resource (#PCDATA)>


<!--
The taglib-default-prefix element specifies the tag prefix that
should be used
for this tag library in JSP pages that use the tag library.  For
example, the
default prefix for the Sun ONE Application Framework/JATO tag library
is "jato".
This prefix may be changed by the JSP author on any given page; this
element simply
gives the default name of the prefix when the tag library declaration
is automatically
added to a page.
-->
<!ELEMENT taglib-default-prefix (#PCDATA)>


<!--
The component element declares a non-extensible component within this
library.
All components must be declared in the component manifest in order to
be
recognized at design-time.
-->
<!ELEMENT component (component-class, component-info-class, design-
reference-resource?)>



<!--
```

```
The extensible-component element declares an extensible component
within this
library.  All extensible components must be declared in the component
manifest
in order to be recognized at design-time.
-->
<!ELEMENT extensible-component (component-class, component-info-
class)>

<!--
The component-class element specifies the fully-qualified class name
of the
component.
-->
<!ELEMENT component-class (#PCDATA)>

<!--
The component-info-class element specifies the fully-qualified name
of the
component's ComponentInfo class.
-->
<!ELEMENT component-info-class (#PCDATA)>

<!--
The design-reference-resource specifies the metadata file resource
that will
be used at design-time to inspect the component.  Components without
this
declaration will generally not be inspectable at design-time other
than through
ComponentInfo. The resource name must follow standard Java resource
name
syntax, with individual path elements separated by forward slashes
("/").

-->
<!ELEMENT design-reference-resource (#PCDATA)>

<!--
```

```
The configurable-bean element declares a non-visual bean component
contained
within this library.
-->
<!ELEMENT configurable-bean (bean-class)>

<!--
The configurable-bean element specifies the fully-qualified class
name of the
non-visual component bean.
-->
<!ELEMENT bean-class (#PCDATA)>
```

Automated Unpacking of Component Tag Libraries (TLD) files

As part of a Sun ONE Application Framework component library, a library developer may provide one or more tag libraries to support rendering of the library's View components. Tag libraries are declared in the component library's component manifest file, and when the IDE toolset recognizes the component library, its tag library descriptors (.tld files) are automatically unpacked from the library JAR file for use by the application. In addition, the IDE toolset automatically adds tag library entries to the web.xml file.

Tag library descriptor files are unpacked to a special location under the application's WEB-INF/tld directory based on the name of the library to ensure that same-named files from different libraries do not conflict. In this scheme, library names are converted to directory names by replacing dots (".") with underscores ("_"). For example, the Sun ONE Application Framework Component Library's internal library name is "com.iplanet.jato", which is translated to "com_iplanet_jato" when unpacking the tag library descriptor. The SCL's tag descriptor file ultimately appears under the WEB-INF/tld/com_iplanet_jato directory in your application.

The tag descriptor's derived physical directory name is automatically registered to a logical resource name in the web.xml file for use by the application. This logical name is chosen by the component library author and specified in the component library manifest. In the SCL's case, the descriptor is registered as the resource /WEB-INF/jato.tld.

The tag descriptor unpacking mechanism makes use of timestamps to determine if an existing file should be overwritten when a new version of the library is added to an application. This feature ensures that upgrading of an application's component libraries is just a single step for a developer.

Referring to the example "mycomponents" library described in this guide. The library author has created a tag library tld file called mycomplib.tld and arbitrarily placed it in the mycomponents package. Therefore, looking into the mycomponents.jar file the tld appears physically located as follows:

```
/COMP-INF/complib.xml
/mycomponents/*.class
/mycomponents/mycomplib.tld
/mycomponents/...
```

Inside of the complib.xml the component author has declared a taglib element as follows

```
<taglib>
      <taglib-uri>/WEB-INF/mycomplib.tld</taglib-uri>
      <taglib-resource>/mycomponents/mycomplib.tld</taglib-resource>
      <taglib-default-prefix>mycomp</taglib-default-prefix>
</taglib>
```

Based on the configuration described by the taglib element, whenever the mycomponens.jar is deployed into an Sun ONE Application Framework Web application's `WEB-INF/lib` directory the IDE toolset will automatically perform the following steps. These steps will allow the run-time JSP engine to properly locate the tag library. This frees the application developer from having to perform any configuration.

- Automatically adds the following entry to the Web application's web.xml file, which sets up a conventional servlet container run-time mapping between the logical resource and its physical location.

```
<taglib>
      <taglib-uri>/WEB-INF/mycomplib.tld</taglib-uri>
      <taglib-location>/WEB-INF/tld/mycomponents/mycomplib.tld</taglib-location>
</taglib>
```

- Automatically extracts the mycomplib.tld file from the mycomponents.jar and places it into the following location

```
[current app]/WEB-INF/tld/mycomponents/mycomplib.tld
```

Also, at design-time as the developer builds application Views the IDE toolset will perform the following

- Automatically ensure that the appropriate tag library declaration is present in any associated JSP files. Note that this declaration contains the "prefix" as specified by the component author in the complib.xml's taglib element.

```
<%@taglib uri="/WEB-INF/mycomplib.tld" prefix="mycomp"%>
```

- Automatically ensures that as the IDE toolset adds additional component library specific tags to the JSP, it also utilize the prefix declared in the taglib directive. For example,

```
<mycomp:validatingTextField name="validatingTextField1"/>
```

Sun ONE Application Framework recognizes that taglib prefix is a JSP page specific directive. J2EE allows each page to declare arbitrary prefixes for the included tag libraries via he tablib directive. The IDE toolset will always utilize the current taglib directive declared prefix as it parses the JSP looking for tags or whenever it automatically inserts additional tags into the JSP in conjunction with the developer's View design decisions. The IDE toolset merely utilizes the complib.xml specified taglib "prefix" in order to insert an initial taglib directive into the application JSP files. Application developers may subsequently manually change the prefix declared in the page specific taglib directive. The IDE toolset thenceforth utilize the newly declared prefix for any additional tags, but it will not automatically change any already declared tags to coincide with the adjusted prefix. This is an application developer issue. We only mention it here so that component authors fully understand the design-time usage of complib.xml's taglib element.

Automated Unpacking of "Additional Files"

Optionally, a Sun ONE Application Framework component library JAR may contain arbitrary "Additional Files" arranged hierarchically underneath the /webapp directory. Do not confuse a Sun ONE Application Framework component library JAR's internal /webapp directory with the common servlet container directory called "webapps". There is absolutely no relationship between the two.

The hierarchical arrangement of the files within the Sun ONE Application Framework component library JAR's /webapp root is totally up to the discretion of the component library author. As a value added feature, the Sun ONE Application Framework IDE tools will "unpack" these additional files into the Web application development environment in direct correspondence to the additional files location relative to the /webapp root.

Please not that this is a pure value added, totally optional, "resource distribution" opportunity provided to component authors by the Sun ONE Application Framework IDE toolset. The assumption is that the extracted files will provide some arbitrary design time or run time value as determined by the component author. It is further assumed that in order to provide this arbitrary value, the extracted files must be extracted to the Web application's file system, otherwise they need not be placed in the "Additional Files" (i.e. /webapp) section of the jar and should be placed in the conventional location within the JAR where they will be picked up by the Java runtime.

For example, consider a mycomponents.jar that contains the following /webapp structure

```
/mycomponents/...
/mycomponents/mycomplib.tld
/webapp/mycomponents/foo.jsp
/webapp/mycomponents/bar.jsp
/webapp/mycomponents/images/banner.gif
/webapp/WEB-INF/jato/templates/jsp/MyViewBeanJSP.jsp
/webapp/WEB-INF/jato/templates/jsp/FooContainerViewJSP.jsp
/webapp/WEB-INF/lib/helper.jar
/webapp/WEB-INF/mycomponents/config-files/configA.xml
/webapp/WEB-INF/mycomponents/config-files/configB.xml
```

When the mycomponents.jar, with the above content, is deployed into a Sun ONE Application Framework application, the IDE toolset will extract the /webapp content into the particular Sun ONE Application Framework application's structure.

For example, consider a Sun ONE Application Framework application called "AppOne" which has the following initial structure created by the Sun ONE Application Framework IDE toolset.

```
AppOne/index.html
AppOne/WEB-INF/classes/...
AppOne/WEB-INF/jato/templates/jsp/DefaultViewBeanJSP.jsp
AppOne/WEB-INF/jato/templates/jsp/DefaultContainerViewJSP.jsp
AppOne/WEB-INF/lib/jato-2_1_0.jar
AppOne/WEB-INF/tld/com_iplanet_jato/jato.tld
```

After "deploying" the mycomponents.jar (i.e. dropping it into AppOne/`Web-INF/lib`), the IDE toolset will discover that it is a Sun ONE Application Framework component library and extract the "additional files" which will result in the creation of the following integrated structure

```
AppOne/index.html
AppOne/mycomponents/foo.jsp
AppOne/mycomponents/bar.jsp
AppOne/mycomponents/images/banner.gif
AppOne/WEB-INF/classes/...
AppOne/WEB-INF/jato/templates/jsp/DefaultViewBeanJSP.jsp
AppOne/WEB-INF/jato/templates/jsp/DefaultContainerViewJSP.jsp
AppOne/WEB-INF/jato/templates/jsp/MyViewBeanJSP.jsp
AppOne/WEB-INF/jato/templates/jsp/FooContainerViewJSP.jsp
AppOne/WEB-INF/lib/jato-2_1_0.jar
AppOne/WEB-INF/lib/mycomponents.jar
AppOne/WEB-INF/lib/helper.jar
AppOne/WEB-INF/mycomponents/config-files/configA.xml
AppOne/WEB-INF/mycomponents/config-files/configB.xml
AppOne/WEB-INF/tld/com_iplanet_jato/jato.tld
AppOne/WEB-INF/tld/mycomponents/mycomplib.tld
```

Component library authors can leverage the Additional Files Feature in order to provide any arbitrary resources which they consider appropriate for extraction. Examples of common Additional Files are (but not limited to)

- Component specific JSP files (for example, component pagelets, and so on)
- Arbitrary Web application document resources (for example, images, static HTML pages, style sheets, JavaScript, and so on)
- Additional arbitrary JAR files. For example, assume that the component library relies on a custom XML parsing library. The component author can "bundle" that JAR inside the component library JAR. This is potentially a more convenient distribution model than requiring the application developer to deploy these extra libraries manually.
- Arbitrary Web application WEB-INF resources. For example suppose the component author designs a set of components which support extra configuration via arbitrary configuration file(s). These configuration files can be "bundled" with the component library and extracted into the appropriate location via the Additional Files mechanism.

Note, the automated extraction of the component library tld file(s) is handled via a different mechanism. The component library tld file(s) should not be located under the /webapp root but rather placed in their normal "resource" appropriate location within the component library jar (e.g. mycomplib.tld above). See Unpacking of Component Tag Libraries - TLD file(s).

Index