

Metawidget User Guide and Reference Documentation



V 0.95

Table of Contents

Preface	viii
Supported Technologies	ix
1. Introduction to Metawidget	1
1.1. Part 1 - The First Metawidget Application	1
1.1.1. The Object	1
1.1.2. The Interface	1
1.1.3. The Output	2
1.1.4. Ordering The Fields	3
1.1.5. Inspectors	4
1.1.6. Combining Multiple Inspection Results	6
1.1.7. Controlling The Layout	7
1.1.8. Controlling Widget Creation	10
1.1.9. Configuring Metawidget Externally	13
1.1.10. Inspecting Different Sources	14
1.2. Part 2 - The Address Book Application	16
1.2.1. Desktop Address Book	16
Read-Only Mode	18
Binding	19
Localization	20
1.2.2. Web Address Book	20
Mixing Metawidgets	23
Expression Based Lookups	24
Alternate Widget Libraries	24
1.2.3. Mobile Address Book	26
1.2.4. Conclusion	29
1.3. Part 3 - Other Examples	29
1.3.1. Swing Applet Address Book Example	30
1.3.2. Seam Example	30
1.3.3. Groovy Example	31
1.3.4. jBPM Example	32
1.3.5. ICEfaces Example	33
1.3.6. Swing AppFramework Example	33
1.3.7. Scala Example	34
1.3.8. GWT Client Side Example	36
1.3.9. GWT Hosted Mode Examples	37
1.3.10. SWT Address Book Examples	39
2. Architecture	41
2.1. Metawidgets	42
2.1.1. Interface	42
2.1.2. Customizing Look and Feel	43
2.1.3. Overriding Widget Creation	43
2.1.4. Implementing Your Own Metawidget	43
BasePipeline	44
Android	44
Java Server Faces	44
Java Server Pages	44

Swing	44
2.2. Inspectors	44
2.2.1. Interface	44
2.2.2. Usage	45
2.2.3. CompositeInspector	46
2.2.4. Defaults	47
2.2.5. Immutability	48
2.2.6. inspection-result	49
2.2.7. Implementing Your Own Inspector	49
2.3. Inspection Result Processors	51
2.3.1. Interface	51
2.3.2. Defaults	52
2.3.3. Immutability	53
2.3.4. Implementing Your Own InspectionResultProcessor	53
2.4. Widget Builders	54
2.4.1. Interface	54
2.4.2. Usage	55
2.4.3. CompositeWidgetBuilder	55
2.4.4. OverriddenWidgetBuilder	56
2.4.5. ReadOnlyWidgetBuilder	57
2.4.6. Defaults	58
2.4.7. Immutability	60
2.4.8. Implementing Your Own WidgetBuilder	60
Special considerations for Java Server Faces	61
2.5. Widget Processors	62
2.5.1. Interface	62
2.5.2. Advanced Interface	63
2.5.3. Defaults	64
2.5.4. Immutability	65
2.5.5. Implementing Your Own WidgetProcessor	65
2.6. Layouts	67
2.6.1. Interface	67
2.6.2. Advanced Interface	67
2.6.3. LayoutDecorator	68
2.6.4. Defaults	70
2.6.5. Immutability	71
2.6.6. Implementing Your Own Layout	72
2.7. metawidget.xml and ConfigReader	74
2.7.1. Constructing New Objects	74
2.7.2. Calling Setter Methods	75
2.7.3. Constructing Primitive Types	75
2.7.4. Resolving Resources	76
2.7.5. Understanding Immutability	76
3. Metawidgets	78
3.1. Desktop Metawidgets	78
3.1.1. SwingMetawidget	78
Installation	78
Customizing Look and Feel	78

3.1.2. SwtMetawidget	78
Installation	78
3.2. Web Metawidgets	79
3.2.1. GwtMetawidget	79
Installation	79
Reflection and Annotations	79
Client-Side Inspection	80
3.2.2. HtmlMetawidgetTag (JSP)	81
Hidden Fields	81
3.2.3. UIMetawidget (JSF)	81
Installation	81
Customizing Look and Feel	81
3.2.4. SpringMetawidgetTag	82
Installation	82
Customizing Look and Feel	82
Overriding Widget Creation	83
3.2.5. StrutsMetawidgetTag	83
Installation	83
Customizing Look and Feel	83
Overriding Widget Creation	83
Troubleshooting	83
I get "Cannot find bean org.apache.struts.taglib.html.BEAN in any scope"	83
I see "MultipartRequestHandler", "ServletWrapper" and other weird names	84
3.3. Mobile Metawidgets	84
3.3.1. AndroidMetawidget	84
Installation	84
Internationalization	84
4. Inspectors	86
4.1. Property Inspectors	86
4.1.1. BaseObjectInspector	86
PropertyStyle	86
JavaBeanPropertyStyle	87
GroovyPropertyStyle	87
JavassistPropertyStyle	87
ScalaPropertyStyle	88
Implementing Your Own PropertyStyle	88
ActionStyle	89
MetawidgetActionStyle	89
SwingAppFrameworkActionStyle	89
4.1.2. PropertyTypeInspector	90
4.1.3. Java5Inspector	90
4.2. Annotation Inspectors	90
4.2.1. BeanValidationInspector	90
4.2.2. FacesInspector	91
4.2.3. HibernateValidatorInspector	92
4.2.4. JexlInspector	92
4.2.5. JpaInspector	93
4.2.6. MetawidgetAnnotationInspector	93

4.2.7. OvalInspector	94
4.2.8. Troubleshooting	95
I get "java.lang.TypeNotPresentException"	95
My inspector is not finding my annotations	95
4.3. XML Inspectors	95
4.3.1. BaseXmlInspector	95
4.3.2. CommonsValidatorInspector	96
4.3.3. HibernateInspector	96
4.3.4. JexlXmlInspector	97
4.3.5. PageflowInspector	97
4.3.6. SeamInspector	97
4.3.7. XmlInspector	97
5. InspectionResultProcessors	98
5.1. ComesAfterInspectionResultProcessor	98
6. Widget Builders	99
6.1. Desktop Widget Builders	99
6.1.1. Swing Widget Builders	99
SwingWidgetBuilder	99
SwingXWidgetBuilder	100
6.1.2. SWT Widget Builders	100
SwtWidgetBuilder	100
6.2. Web Widget Builders	101
6.2.1. JSP Widget Builders	101
DisplayTagWidgetBuilder	101
HtmlWidgetBuilder	101
SpringWidgetBuilder	102
StrutsWidgetBuilder	103
6.2.2. GWT Widget Builders	103
ExtGwtWidgetBuilder	103
GwtWidgetBuilder	104
6.2.3. JSF Widget Builders	104
HtmlWidgetBuilder	104
IceFacesWidgetBuilder	105
RichFacesWidgetBuilder	106
TomahawkWidgetBuilder	106
6.3. Mobile Widget Builders	107
6.3.1. Android Widget Builders	107
AndroidWidgetBuilder	107
7. WidgetProcessors	108
7.1. Desktop Widget Processors	108
7.1.1. Swing Widget Processors	108
Property Binding	108
BeansBindingProcessor	108
BeanUtilsProcessor	108
Action Binding	109
7.1.2. SWT Widget Processors	109
DataBindingProcessor	109
ReflectionBindingProcessor	109

7.2. Web Widget Processors	109
7.2.1. GWT Widget Processors	109
Property Binding	109
Action Binding	110
7.2.2. JSF Widget Processors	110
HiddenFieldProcessor	110
StandardValidationProcessor	110
RichFacesProcessor	110
8. Layouts	111
8.1. Desktop Layouts	111
8.1.1. Swing Layouts	111
BoxLayout	111
FlowLayout	111
GridBagLayout	111
GroupLayout	111
MigLayout	112
SeparatorLayoutDecorator	112
TabbedPaneLayoutDecorator	112
TitledPanelLayoutDecorator	112
8.1.2. SWT Layouts	113
FillLayout	113
GridLayout	113
MigLayout	113
RowLayout	113
SeparatorLayoutDecorator	113
TabFolderLayoutDecorator	114
8.2. Web Layouts	114
8.2.1. GWT Layouts	114
FlexTableLayout	114
FlowLayout	114
LabelLayoutDecorator	114
TabPanelLayoutDecorator	115
8.2.2. JSF Layouts	115
HtmlDivLayoutRenderer	115
HtmlTableLayoutRenderer	115
OutputTextLayoutDecorator	116
PanelLayoutDecorator	116
SimpleTogglePanelLayoutDecorator	117
SimpleLayout	117
TabPanelLayoutDecorator	117
8.2.3. JSP Layouts	117
HeadingTagLayoutDecorator	117
HtmlTableLayout	117
SimpleLayout	118
8.3. Mobile Layouts	118
8.3.1. Android Layouts	118
LinearLayout	118
TableLayout	118

TabHostLayoutDecorator	119
TextViewLayoutDecorator	119
9. How To's	120
9.1. Order Fields	120
9.2. Remote Inspection	120
9.3. Combine Remote Inspections	121
10. Performance	122
10.1. JAR Size	122
10.2. Memory Usage	122
10.3. Rebinding	122
11. Epilogue	123

Preface

Metawidget is an object/user interface mapping tool for Java environments. The term object/user interface mapping (OIM) refers to the technique of inspecting objects, at runtime, and creating User Interface (UI) widgets.

As much as possible, Metawidget does this without introducing new technologies. As shown in [Figure 1](#), Metawidget inspects an application's *existing* back-end architecture (such as JavaBeans, XML configuration files, annotations) and creates widgets native to its *existing* front-end framework (such as Swing, Java Server Faces, Struts or Android).

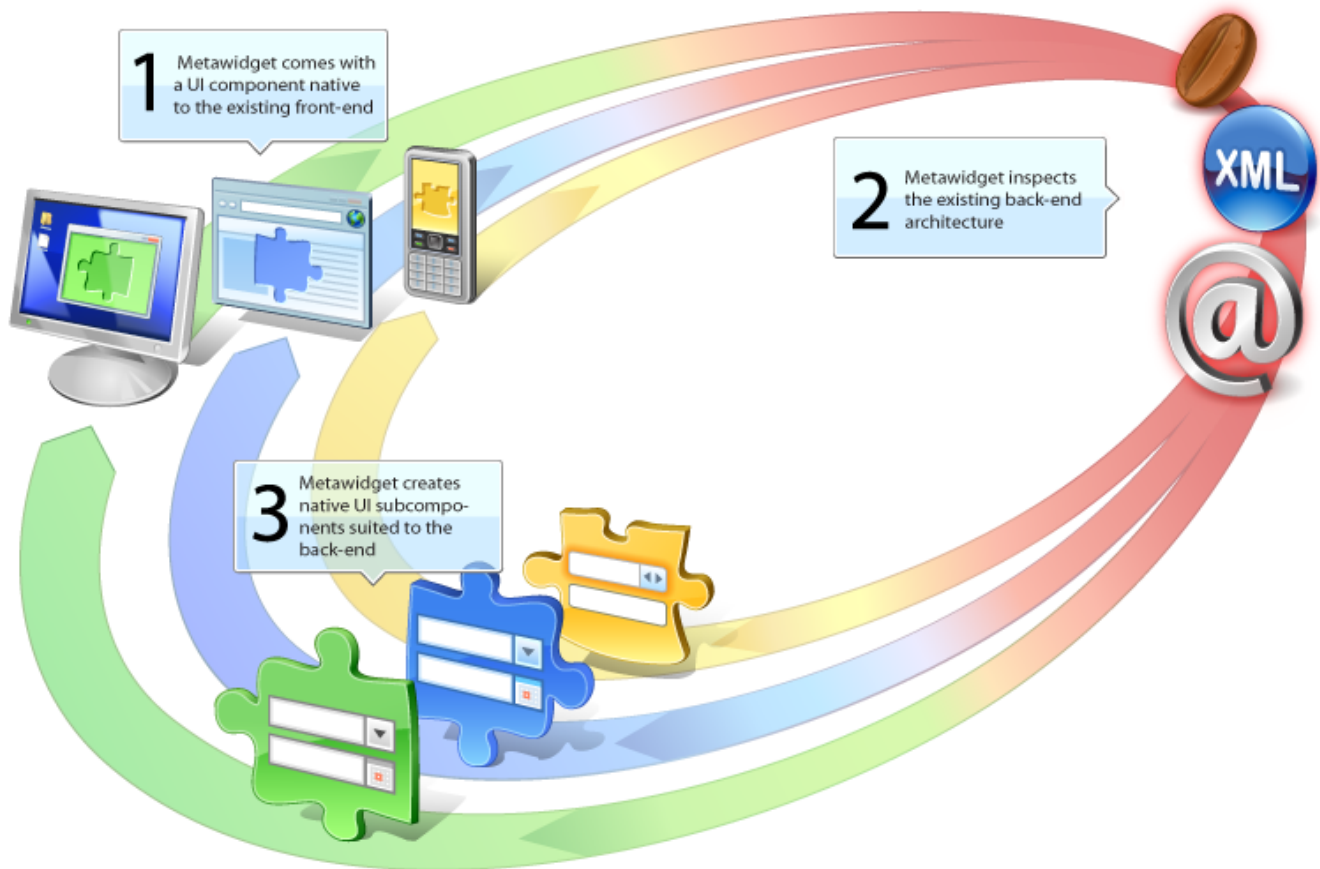


Figure 1. Metawidget inspects existing back-ends and creates widgets native to existing front-ends

Building great UIs is both art and science. Metawidget does not attempt to address the *art*, it only automates the *science*. That is to say, it does not overlap with those areas of UI design involving creativity and subjectivity - its goal is only to ease the creation of areas that are already rigidly defined. Typically, this means those areas that display data and those that collect data - these tend to be both commonplace and consistent (indeed, consistency is a desirable trait) so there is good opportunity for automation.



Note

This User Guide and Reference Documentation is included in the distribution as a PDF, a single HTML page or as multiple HTML pages, depending on your reading preference.

Supported Technologies

A primary goal of Metawidget is to work with your *existing* front-end and back-end architecture. Out of the box, Metawidget supports a broad range of both front-end and back-end technologies, and makes it easy to add your own.

Metawidget comes with a native UI component for each supported front-end. This support includes: Android, Google Web Toolkit (including extensions such as ExtGWT), Java Server Faces (including extensions such as Facelets, ICEfaces, RichFaces and Tomahawk), 'plain' Java Server Pages (including extensions such as DisplayTag), Spring Web MVC, Struts, Swing (including extensions such as Beans Binding, JGoodies, MigLayout and SwingX) and SWT.

Metawidget can read business object information from any combination of supported back-end technologies. This support includes: annotations, Bean Validation (JSR 303), Commons JEXL, Commons Validator, Groovy, Hibernate, Hibernate Validator, JavaBeans, Java Persistence Architecture (JPA), Javassist, JBoss jBPM, OVal, Scala, Seam and the Swing AppFramework.



Note

It is *not* a goal of Metawidget that all widgets look the same on every front-end framework, or that all back-end technologies conform to some 'lowest common denominator': every technology has different features, and Metawidget takes full advantage of this.

The next chapter presents a tutorial covering using Metawidget with a variety of front-ends and back-ends. Chapter 2 then follows with a more in-depth architectural overview. Chapters 3-8 explore each supported front-end and back-end technology in detail. Finally, chapters 9 and 10 offer general advice and performance tips.

1. Introduction to Metawidget

This chapter is an introductory tutorial for new users of Metawidget. Before you begin, you need to download at least the binary distribution, and preferably the source code distribution as well, from <http://www.metawidget.org/download.html>.

1.1 Part 1 - The First Metawidget Application

Part 1 starts with a simple Swing application and develops it in easy to understand steps. Metawidget supports many UI frameworks, not just Swing, but we start with Swing because it ships with Java SE and requires minimal setup.

This tutorial should take around 20 minutes. We recommend you use your preferred Java development environment. If you use an Integrated Development Environment (IDE), you will need to start a new Java project and add `metawidget.jar` to it. Otherwise, you just need to ensure `metawidget.jar` is on your `CLASSPATH`.

1.1.1 The Object

Metawidget is an object/user interface mapping tool (OIM), so first we need an object to map from - the *O* in OIM. Create a class in your project called `Person` with the following code:

```
package com.myapp;

public class Person {
    public String name; ❶
    public int age;
    public boolean retired;
}
```

- ❶ This tutorial uses public member variables for brevity. The recommended design is to use JavaBean property getter and setter methods. Metawidget supports both approaches.

1.1.2 The Interface

Next we need a User Interface framework - the *I* in OIM. Create a class in your project called `Main` with the following code:

```
package com.myapp;

import javax.swing.*;
import org.metawidget.swing.*;

public class Main {

    public static void main( String[] args ) {
        Person person = new Person();

        SwingMetawidget metawidget = new SwingMetawidget();
        metawidget.setToInspect( person );
    }
}
```

```

JFrame frame = new JFrame( "Metawidget Tutorial" );
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.getContentPane().add( metawidget );
frame.setSize( 400, 250 );
frame.setVisible( true );
}
}

```

**Note**

Many IDEs include visual UI builders for dragging and dropping widgets. Metawidget integrates with these tools and Metawidget widgets can be dragged and dropped like any other. As we shall see, however, Metawidget widgets automatically fill themselves with child widgets at runtime, saving significant development time.

1.1.3 The Output

If you're using an IDE, such as [NetBeans](#), your project should look something like pictured in [Figure 1.1](#).

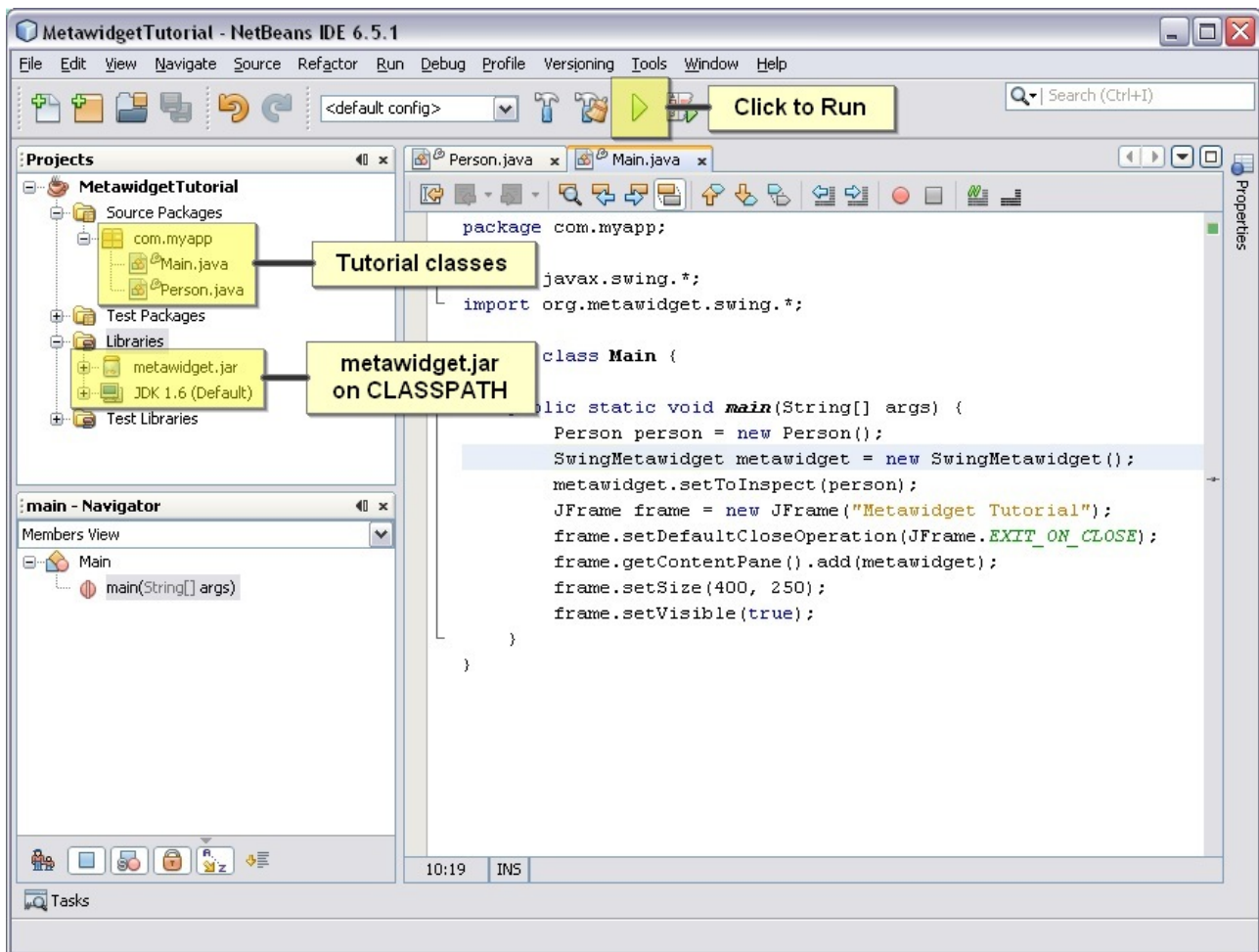


Figure 1.1. Metawidget tutorial in NetBeans IDE

Run the code. You should see the screen in [Figure 1.2](#).

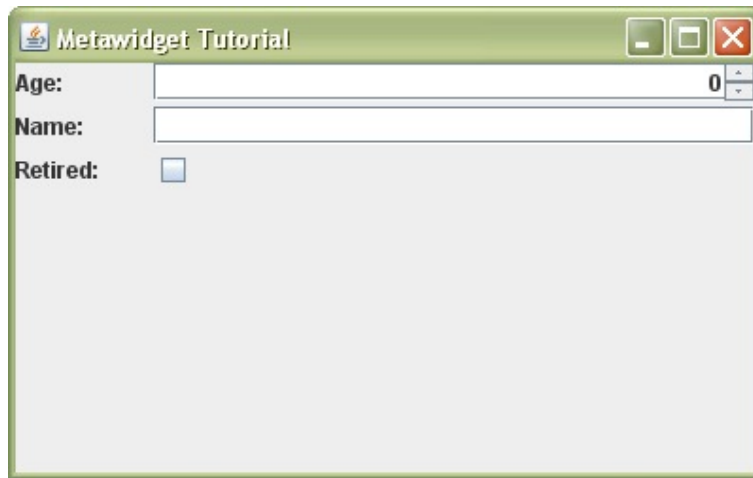


Figure 1.2. *SwingMetawidget* rendering of *Person* class

The `SwingMetawidget` has automatically populated itself with child widgets at runtime. It has chosen `JSpinner`, `JTextField` and `JCheckBox` widgets based on the types of the properties of the `Person` class. This is the First Goal Of Metawidget:



First Goal Of Metawidget

Metawidget creates UI widgets by inspecting existing back-end architectures

By default, `SwingMetawidget` has laid out the `JComponents` using `java.awt.GridBagLayout`. Try resizing the window, and the `JComponents` will resize with it. If you've ever tried using `java.awt.GridBagLayout` yourself, either through code or a visual UI builder, you'll know how fiddly it can be. Having Metawidget do it for you is a real time-saver.

Clearly this is not a complete UI. There are no Save or Cancel buttons, for example, and the `JComponents` appear uncomfortably tight to the left, top and right edges of the `JFrame`. This is explained by the Second Goal Of Metawidget:



Second Goal Of Metawidget

Metawidget does not try to 'own' the entire UI - it focusses on creating native sub-widgets for slotting into existing UIs

You slot Metawidget alongside your standard UI components, often combining several Metawidgets on the same screen. We'll see how this works later.

1.1.4 Ordering The Fields

Currently the *name*, *age* and *retired* fields are arranged alphabetically in the UI - their order does not match the way they are defined in the `Person` class. This is because field ordering information is not retained within Java class files (as per the Java Language Specification).

To correct this, Metawidget needs to gather additional information. There are several ways to do this (ie. you don't have to use annotations), but the simplest for now is to use the built-in Metawidget annotation `@UiComesAfter`.



Note

The following code uses annotations, so you'll need Java SE 5 or higher. Metawidget itself can run on J2SE 1.4, but you'll need to gather the ordering information from a different source (see [Section 1.1.10, "Inspecting Different Sources"](#))

Annotate the `Person` class as shown below (lines to add are highlighted):

```
package com.myapp;

import org.metawidget.inspector.annotation.*;

public class Person {
    public String name;

    @UiComesAfter( "name" ) ❶
    public int age;

    @UiComesAfter( "age" )
    public boolean retired;
}
```

- ❶ Such annotations can (and should) be applied to getter methods, but for brevity here they are applied directly to the member variable.

Run the code again. This time the fields appear in the correct order:

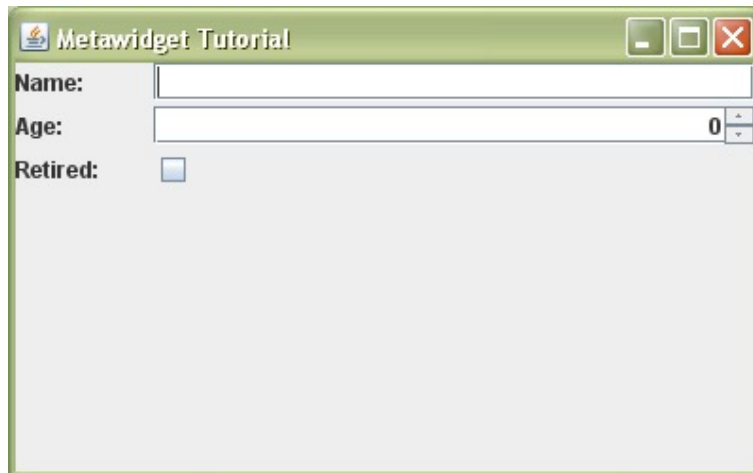


Figure 1.3. Fields in correct order

1.1.5 Inspectors

Introducing new annotations to improve the UI is not really in the spirit of the First Goal Of Metawidget. We'd much rather improve it by gathering information from *existing* sources. To demonstrate how Metawidget can adapt to different sources of metadata, add the following lines to `Person` (lines to add are highlighted):

```
package com.myapp;

import org.metawidget.inspector.annotation.*;

public class Person {
    public String name;

    @UiComesAfter( "name" )
    public int age;

    @UiComesAfter( "age" )
    public boolean retired;
}
```

```

@UiComesAfter( "retired" )
public Gender gender;

public enum Gender { Male, Female }
}

```

Run the code again:

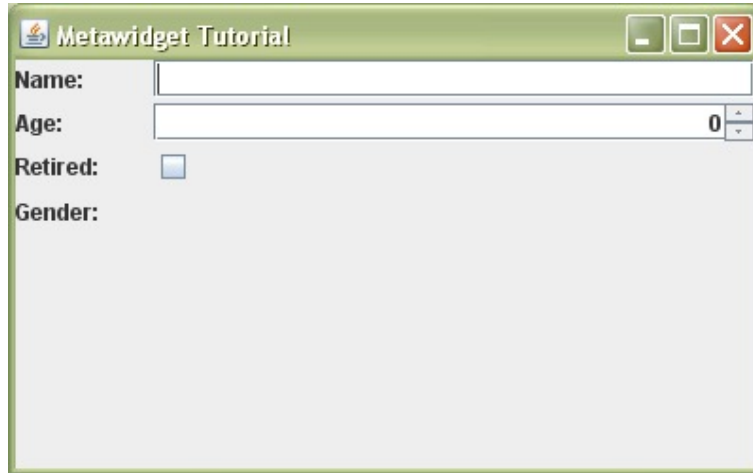


Figure 1.4. New Gender field, but missing a component

Metawidget finds the new gender property, and renders a `JLabel` for it on the left, but doesn't know what `JComponent` to put on the right. This is because, by default, Metawidget does not inspect Java 5 language features such as enums and generics.

To recognise the enum, Metawidget needs to use a different *Inspector*. Metawidget comes with multiple *Inspectors*, each targeting different sources of information. Change the Main class to use a `Java5Inspector` (lines to add are highlighted):

```

package com.myapp;

import javax.swing.*;
import org.metawidget.inspector.java5.*;
import org.metawidget.swing.*;

public class Main {

    public static void main( String[] args ) {
        Person person = new Person();

        SwingMetawidget metawidget = new SwingMetawidget();
        metawidget.setInspector( new Java5Inspector() );
        metawidget.setToInspect( person );

        JFrame frame = new JFrame( "Metawidget Tutorial" );
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.getContentPane().add( metawidget );
        frame.setSize( 400, 250 );
        frame.setVisible( true );
    }
}

```

Run the code again. It does not yield the correct result - the gender enum appears correctly as a `JComboBox` but all the other fields have disappeared! What happened?

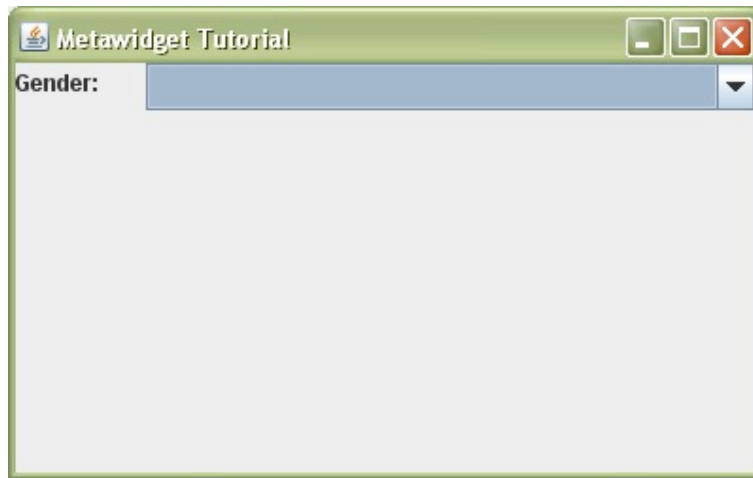


Figure 1.5. Correct Gender field, but missing other fields

Metawidget Inspectors are very targeted in what they inspect. `Java5Inspector` looks for Java 5 language features - such as enums - but it does *not* look for anything else - such as JavaBean properties. Before we explicitly specified a `Java5Inspector`, Metawidget had been implicitly using two Inspectors for us, called `PropertyTypeInspector` and `MetawidgetAnnotationInspector`.

1.1.6 Combining Multiple Inspection Results

What we need is to *combine* the results of `PropertyTypeInspector`, `MetawidgetAnnotationInspector` and `Java5Inspector` before returning them to `SwingMetawidget`. We do this using `CompositeInspector`:

```
package com.myapp;

import javax.swing.*;
import org.metawidget.inspector.annotation.*;
import org.metawidget.inspector.composite.*;
import org.metawidget.inspector.java5.*;
import org.metawidget.inspector.propertytype.*;
import org.metawidget.swing.*;

public class Main {

    public static void main( String[] args ) {
        Person person = new Person();

        SwingMetawidget metawidget = new SwingMetawidget();
        CompositeInspectorConfig inspectorConfig = new CompositeInspectorConfig().setInspectors(
            new PropertyTypeInspector(),
            new MetawidgetAnnotationInspector(),
            new Java5Inspector() );
        metawidget.setInspector( new CompositeInspector( inspectorConfig ) );
        metawidget.setToInspect( person );

        JFrame frame = new JFrame( "Metawidget Tutorial" );
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.getContentPane().add( metawidget );
        frame.setSize( 400, 250 );
        frame.setVisible( true );
    }
}
```

Run the code again. This time both the original fields and the new gender JComboBox appear:

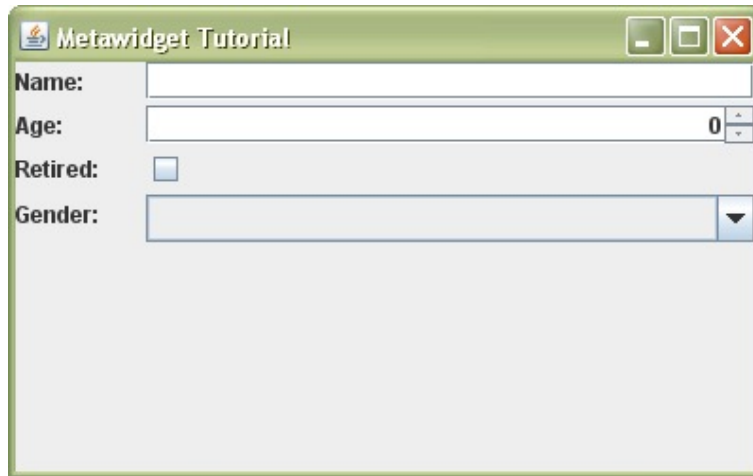


Figure 1.6. Using multiple inspectors

This idea of combining multiple Inspectors to inspect different characteristics of your existing architecture is very powerful. Metawidget comes with many pre-written Inspectors for many different architectures - from JPA and Hibernate Validator annotations, to struts-config.xml configuration files, to Groovy and Scala properties - Metawidget will gather and combine UI information from wherever it can find it.

1.1.7 Controlling The Layout

There are several ways to control the layout of the components. To demonstrate, Try adding the following fields to the Person class:

```
package com.myapp;

import org.metawidget.inspector.annotation.*;

public class Person {
    public String name;

    @UiComesAfter( "name" )
    public int age;

    @UiComesAfter( "age" )
    public boolean retired;

    @UiComesAfter( "retired" )
    public Gender gender;

    public enum Gender { Male, Female }

    @UiComesAfter( "gender" )
    @UiLarge
    public String notes;

    @UiComesAfter( "notes" )
    @UiSection( "Work" )
    public String employer;

    @UiComesAfter( "employer" )
```



```
public String department;
}
```

This code produces the screen in [Figure 1.7](#). Annotations have been used to define section headings and 'large' fields (ie. a `JTextArea`).

Figure 1.7. Additional fields and a section heading



Note

For a list of all the annotations `MetawidgetAnnotationInspector` recognises, see [Section 4.2.6](#), “`MetawidgetAnnotationInspector`”.

By default, `SwingMetawidget` lays out `JComponents` using `org.metawidget.swing.layout.GridBagLayout`. You can configure this layout, or swap it for a different layout, using `SwingMetawidget.setMetawidgetLayout`. Modify the code to use a `GridBagLayout` with 2 columns.

```
package com.myapp;

import javax.swing.*;
import org.metawidget.inspector.annotation.*;
import org.metawidget.inspector.composite.*;
import org.metawidget.inspector.java5.*;
import org.metawidget.inspector.propertytype.*;
import org.metawidget.swing.*;
import org.metawidget.swing.layout.*;

public class Main {

    public static void main( String[] args ) {
        Person person = new Person();

        SwingMetawidget metawidget = new SwingMetawidget();
        CompositeInspectorConfig inspectorConfig = new CompositeInspectorConfig().setInspectors(
            new PropertyTypeInspector(),
            new MetawidgetAnnotationInspector(),
            new Java5Inspector() );
        metawidget.setInspector( new CompositeInspector( inspectorConfig ) );
        GridBagLayoutConfig nestedLayoutConfig = new GridBagLayoutConfig().setNumberOfColumns( 2 );
        SeparatorLayoutDecoratorConfig layoutConfig = new SeparatorLayoutDecoratorConfig().setLayout(
            new org.metawidget.swing.layout.GridBagLayout( nestedLayoutConfig ) );
        metawidget.setMetawidgetLayout( new SeparatorLayoutDecorator( layoutConfig ) );
        metawidget.setToInspect( person );
    }
}
```

```

JFrame frame = new JFrame( "Metawidget Tutorial" );
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.getContentPane().add( metawidget );
frame.setSize( 400, 250 );
frame.setVisible( true );
}
}

```

Run the code. The `JComponents` are now arranged across two columns as in [Figure 1.8](#).

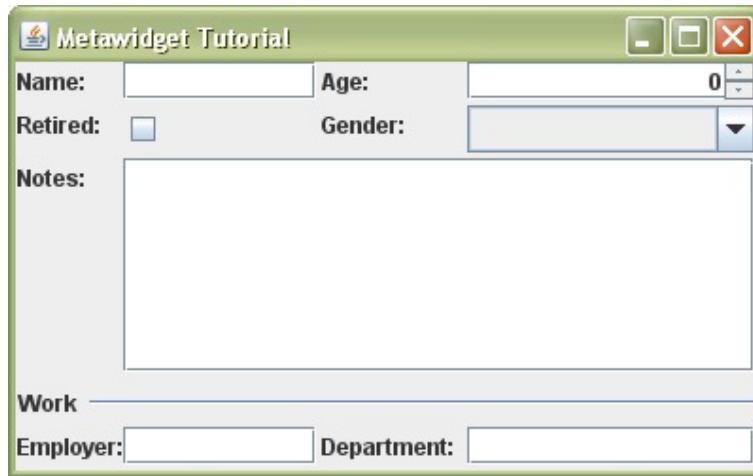


Figure 1.8. A two-column layout

You may have noticed the `GridBagLayout` is nested inside a `SeparatorLayoutDecorator`. This is responsible for separating widgets in different sections using `JSeparators`. However there are other choices for separating widgets. Modify the code to use `TabbedPaneLayoutDecorator` instead:

```

package com.myapp;

import javax.swing.*;
import org.metawidget.inspector.annotation.*;
import org.metawidget.inspector.composite.*;
import org.metawidget.inspector.java5.*;
import org.metawidget.inspector.propertytype.*;
import org.metawidget.swing.*;
import org.metawidget.swing.layout.*;

public class Main {

    public static void main( String[] args ) {
        Person person = new Person();

        SwingMetawidget metawidget = new SwingMetawidget();
        CompositeInspectorConfig inspectorConfig = new CompositeInspectorConfig().setInspectors(
            new PropertyTypeInspector(),
            new MetawidgetAnnotationInspector(),
            new Java5Inspector() );
        metawidget.setInspector( new CompositeInspector( inspectorConfig ) );
        GridBagLayoutConfig nestedLayoutConfig = new GridBagLayoutConfig().setNumberOfColumns( 2 );
        TabbedPaneLayoutDecoratorConfig layoutConfig = new TabbedPaneLayoutDecoratorConfig().setLayout(
            new org.metawidget.swing.layout.GridBagLayout( nestedLayoutConfig ) );
        metawidget.setMetawidgetLayout( new TabbedPaneLayoutDecorator( layoutConfig ) );
        metawidget.setToInspect( person );
    }
}

```

```

JFrame frame = new JFrame( "Metawidget Tutorial" );
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.getContentPane().add( metawidget );
frame.setSize( 400, 250 );
frame.setVisible( true );
}
}

```

Run the code. The section heading is now a `JTabbedPane` as in [Figure 1.9](#).

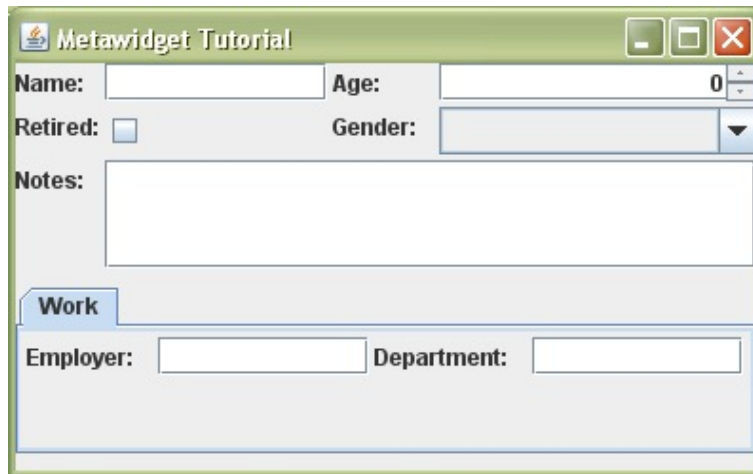


Figure 1.9. Two-column layout with a `JTabbedPane`

Again, if you've ever used `java.awt.GridBagLayout` by hand, you'll appreciate how much easier Metawidget makes all this.

1.1.8 Controlling Widget Creation

There are several ways to control widget creation. One way is to drop child controls inside the `SwingMetawidget`. This approach works well both within code and within visual UI builders.

Modify the code to add a `JComboBox` to the Metawidget:

```

package com.myapp;

import javax.swing.*;
import org.metawidget.inspector.annotation.*;
import org.metawidget.inspector.composite.*;
import org.metawidget.inspector.java5.*;
import org.metawidget.inspector.propertytype.*;
import org.metawidget.swing.*;
import org.metawidget.swing.layout.*;

public class Main {

    public static void main( String[] args ) {
        Person person = new Person();

        SwingMetawidget metawidget = new SwingMetawidget();
        CompositeInspectorConfig inspectorConfig = new CompositeInspectorConfig().setInspectors(
            new PropertyTypeInspector(),
            new MetawidgetAnnotationInspector(),

```

```

new Java5Inspector() );
metawidget.setInspector( new CompositeInspector( inspectorConfig ) );
GridBagLayoutConfig nestedLayoutConfig = new GridBagLayoutConfig().setNumberOfColumns( 2 );
TabbedPaneLayoutDecoratorConfig layoutConfig = new TabbedPaneLayoutDecoratorConfig().setLayout(
    new org.metawidget.swing.layout.GridBagLayout( nestedLayoutConfig ) );
metawidget.setMetawidgetLayout( new TabbedPaneLayoutDecorator( layoutConfig ) );
metawidget.setToInspect( person );
JComboBox combo = new JComboBox();
combo.setName( "retired" );
metawidget.add( combo );

JFrame frame = new JFrame( "Metawidget Tutorial" );
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.getContentPane().add( metawidget );
frame.setSize( 400, 250 );
frame.setVisible( true );
}
}

```

Run the code. The `JComboBox` appears in place of the *retired* `JCheckBox`, because it has the same name (ie. 'retired') as Metawidget would have given the `JCheckBox`:

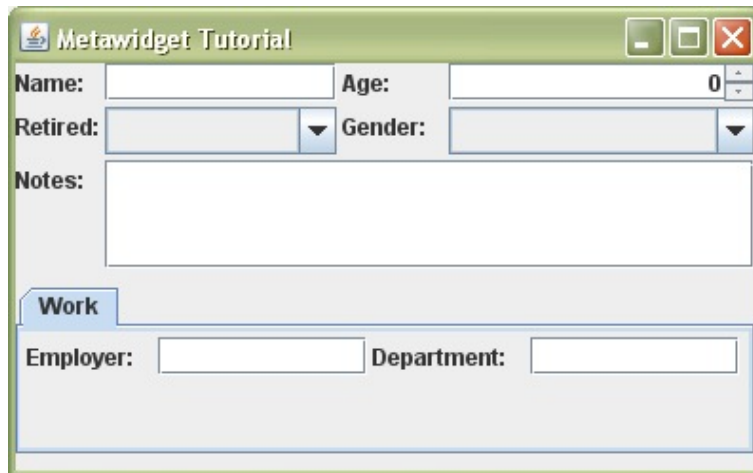


Figure 1.10. The 'retired' field has been overridden



Note

The default algorithm looks for child controls with the same name, but you can plug in your own implementation if you need to. See [Section 2.4.4, “OverriddenWidgetBuilder”](#).

To suppress a widget's creation entirely, simply supplying an empty `JPanel` named 'retired' will not work as Metawidget will still create an accompanying label in the left hand column. Instead, Metawidget includes special `Stub` widgets for this purpose:

```

package com.myapp;

import javax.swing.*;
import org.metawidget.inspector.annotation.*;
import org.metawidget.inspector.composite.*;
import org.metawidget.inspector.java5.*;
import org.metawidget.inspector.propertytype.*;
import org.metawidget.swing.*;

public class Main {

```

```

public static void main( String[] args ) {
    Person person = new Person();

    SwingMetawidget metawidget = new SwingMetawidget();
    CompositeInspectorConfig inspectorConfig = new CompositeInspectorConfig().setInspectors(
        new PropertyTypeInspector(),
        new MetawidgetAnnotationInspector(),
        new Java5Inspector() );
    metawidget.setInspector( new CompositeInspector( inspectorConfig ) );
    GridBagLayoutConfig nestedLayoutConfig = new GridBagLayoutConfig().setNumberOfColumns( 2 );
    TabbedPaneLayoutDecoratorConfig layoutConfig = new TabbedPaneLayoutDecoratorConfig().setLayout(
        new org.metawidget.swing.layout.GridBagLayout( nestedLayoutConfig ) );
    metawidget.setMetawidgetLayout( new TabbedPaneLayoutDecorator( layoutConfig ) );
    metawidget.setToInspect( person );
    metawidget.add( new Stub( "retired" ) );

    JFrame frame = new JFrame( "Metawidget Tutorial" );
    frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    frame.getContentPane().add( metawidget );
    frame.setSize( 400, 250 );
    frame.setVisible( true );
}
}

```

Run the code. The *retired* field and its label will not appear, as in [Figure 1.11](#).

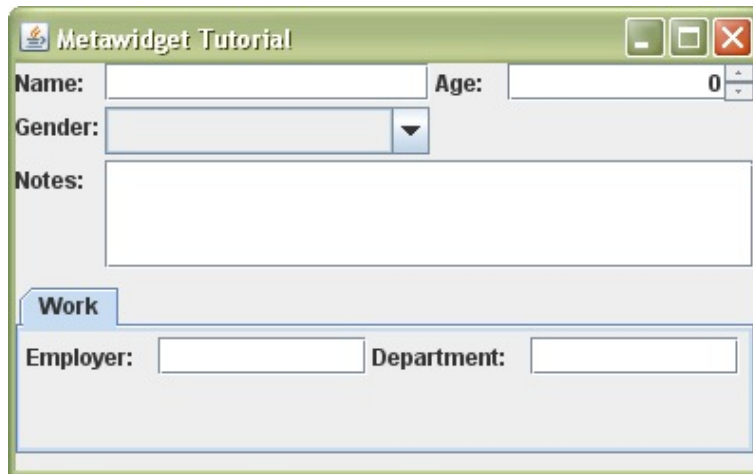


Figure 1.11. The 'retired' field has been suppressed

Another way is to use a `@UiHidden` annotation on the business class:

```

package com.myapp;

import org.metawidget.inspector.annotation.*;

public class Person {
    public String name;

    @UiComesAfter( "name" )
    public int age;

    @UiComesAfter( "age" )
    @UiHidden

```

```

public boolean retired;

@UiComesAfter( "retired" )
public Gender gender;

public enum Gender { Male, Female }

@UiComesAfter( "gender" )
@UiLarge
public String notes;

@UiComesAfter( "notes" )
@UiSection( "Work" )
public String employer;

@UiComesAfter( "employer" )
public String department;
}

```

In both cases, `org.metawidget.swing.layout.GridBagLayout` is smart enough to always give large `JComponents` like `notes` the full width of the `JFrame`.

1.1.9 Configuring Metawidget Externally

So far we have been instantiating our `Inspectors` and `Layouts` in Java code. Whilst this approach is possible for all `Inspectors` and `Layouts`, many UI frameworks employ visual UI builders or intermediate languages (such as JSP) that make getting to the Java code cumbersome (ie. you have to derive custom widgets).

As an alternative, Metawidget supports external XML configuration. Create a file called `metawidget.xml` in the same folder as your `Main` class:

```

<metawidget xmlns="http://metawidget.org"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="1.0"
  xsi:schemaLocation="http://metawidget.org http://metawidget.org/xsd/metawidget-1.0.xsd">

  <swingMetawidget xmlns="java:org.metawidget.swing">
    <inspector>
      <compositeInspector xmlns="java:org.metawidget.inspector.composite"
        config="CompositeInspectorConfig">
        <inspectors>
          <array>
            <propertyTypeInspector xmlns="java:org.metawidget.inspector.propertytype" />
            <metawidgetAnnotationInspector xmlns="java:org.metawidget.inspector.annotation" />
            <java5Inspector xmlns="java:org.metawidget.inspector.java5" />
          </array>
        </inspectors>
      </compositeInspector>
    </inspector>
    <metawidgetLayout>
      <tabbedPaneLayoutDecorator xmlns="java:org.metawidget.swing.layout"
        config="TabbedPaneLayoutDecoratorConfig">
        <layout>
          <gridBagLayout config="GridBagLayoutConfig">
            <numberOfColumns>
              <int>2</int>
            </numberOfColumns>
          </gridBagLayout>

```

```
</layout>
</tabbedPaneLayoutDecorator>
</metawidgetLayout>
</swingMetawidget>

</metawidget>
```

Now update your Main class to use this file:

```
package com.myapp;

import javax.swing.*;
import org.metawidget.swing.*;

public class Main {

    public static void main( String[] args ) {
        Person person = new Person();

        SwingMetawidget metawidget = new SwingMetawidget();
        metawidget.setConfig( "com/myapp/metawidget.xml" );
        metawidget.setToInspect( person );

        JFrame frame = new JFrame( "Metawidget Tutorial" );
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.getContentPane().add( metawidget );
        frame.setSize( 400, 250 );
        frame.setVisible( true );
    }
}
```

Run the code. The output is the same as before, but this time we are configuring our Metawidget via external XML.

Visual UI builders can call `SwingMetawidget.setConfig` from the builder, with no coding required. Other UI frameworks (eg. JSPs, Android) have similar 'code free' approaches (eg. setting an attribute on a JSP tag, setting an attribute in an Android layout file) to setting the XML file.

1.1.10 Inspecting Different Sources

It could be argued UI-oriented annotations such as `@UiComesAfter` sit uncomfortably on a business class from a 'separation of concerns' perspective. It has advantages in that it keeps the metadata close to the data it refers to. It is also sufficiently abstract that it does not tie the code to any particular UI framework.

However, for those needing a different approach Metawidget can use different `Inspectors` to gather information from almost any source. One example is to use `XmlInspector`. Create a file called `metawidget-metadata.xml` in the same folder as your Main class:

```
<inspection-result xmlns="http://metawidget.org/inspection-result"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="1.0"
  xsi:schemaLocation="http://metawidget.org/inspection-result
    http://metawidget.org/xsd/inspection-result-1.0.xsd">

  <entity type="com.myapp.Person">
    <property name="name" />
    <property name="age" />
    <property name="retired" hidden="true" />
    <property name="gender" />
  </entity>
</inspection-result>
```

```

<property name="notes" large="true" />
<property name="employer" section="Work" />
<property name="department" />
</entity>

</inspection-result>

```

**Note**

In XML, the *comes-after* attribute is optional: XML nodes are inherently ordered, and CompositeInspector combines inspection results so that later results respect the ordering of earlier results.

**Note**

The XML does not need to specify a *type* attribute: we will still be using PropertyTypeInspector to look up the type. It also does not need to specify a *lookup* attribute: we will still be using Java5Inspector to determine possible enum values.

Update your `metawidget.xml` to use `XmlInspector` instead of `MetawidgetAnnotationInspector`:

```

<metawidget xmlns="http://metawidget.org"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="1.0"
  xsi:schemaLocation="http://metawidget.org http://metawidget.org/xsd/metawidget-1.0.xsd">

  <swingMetawidget xmlns="java:org.metawidget.swing">
    <inspector>
      <compositeInspector xmlns="java:org.metawidget.inspector.composite"
        config="CompositeInspectorConfig">
        <inspectors>
          <array>
            <xmlInspector xmlns="java:org.metawidget.inspector.xml"
              config="XmlInspectorConfig">
              <inputStream>
                <resource>com/myapp/metawidget-metadata.xml</resource>
              </inputStream>
            </xmlInspector>
            <propertyTypeInspector xmlns="java:org.metawidget.inspector.propertytype" />
            <java5Inspector xmlns="java:org.metawidget.inspector.java5" />
          </array>
        </inspectors>
      </compositeInspector>
    </inspector>
    <metawidgetLayout>
      <tabbedPaneLayoutDecorator xmlns="java:org.metawidget.swing.layout"
        config="TabbedPaneLayoutDecoratorConfig">
        <layout>
          <gridBagLayout config="GridBagLayoutConfig">
            <numberOfColumns>
              <int>2</int>
            </numberOfColumns>
          </gridBagLayout>
        </layout>
      </tabbedPaneLayoutDecorator>
    </metawidgetLayout>
  </swingMetawidget>

</metawidget>

```


Remove all the annotations from the `Person` class and run the code again. The ordering is still correct, and there is still a section heading, but this time it is being dictated by `XmlInspector`.

This idea of UI characteristics being derivable from different back-end sources is fundamental to Metawidget. There is a lot of metadata already lurking in back-end systems - it just needs extracting. For example, `JpaInspector` understands this...

```
import javax.persistence.Column;

public class Person {
    @Column( nullable = false )
    public String name;
}
```

...denotes *name* is a required field (could be rendered with a star after it in the UI). Equally, `PropertyTypeInspector` understands that...

```
public class Person {
    private String name;

    public String getName() {
        return this.name;
    }

    // No setter
}
```

...signifies *name* is a read-only field (could be rendered as a label in the UI).

Metawidget comes with a range of `Inspectors`, and it is straightforward to write your own to inspect anything from XML configuration files to database schemas to annotations. The inspection process is decoupled from the widget creation process, so the same inspector can supply information to multiple UI frameworks.

1.2 Part 2 - The Address Book Application

Part 2 explores a more substantial application, and shows how Metawidget can be used to map the *same* back-end to *multiple* front-ends. We will develop an Address Book application with desktop-based, Web-based and mobile-based UIs.

This tutorial should take around 45 minutes. To save time, we use the pre-built example applications located in the `examples` folder. Also to save time, we will not focus on any one front-end framework in detail. For detailed framework-specific instructions, please see [Chapter 3, Metawidgets](#).

1.2.1 Desktop Address Book

The Desktop Address Book is essentially a larger version of the Swing application developed in Part 1 - it just has more business objects and more Swing widgets.

The application is pre-built for you in `examples\swing\addressbook-swing.jar` or, if you've downloaded the source code distribution, you can build it yourself by changing to the `examples` folder of the source distribution and typing:

```
ant example-swing-addressbook
```

This is a self-executing JAR. For convenience, it has `MANIFEST.MF` dependencies hard-wired into it to `..\..\metawidget.jar` and `lib\beansbinding.jar` among others, so it's best not to move it to a different folder (if you do, you'll need to manually put `metawidget.jar` and `beansbinding.jar` on your classpath).

**Note**

There is also an SWT version of the Address Book sample in `examples/swt/addressbook-swt.jar`. You can mostly replace references to 'Swing' in this section with 'SWT' and follow along using SWT if preferred.

Run the code by navigating to the `examples\swing` folder and typing:

```
java -jar addressbook-swing.jar
```

The opening screen displays a search filter (at the top) and lists existing Address Book entries (at the bottom) as in [Figure 1.12](#).



Figure 1.12. Desktop Address Book opening screen

The three search filter fields (*Firstname*, *Surname* and *Type*) are created by `SwingMetawidget` based on the `ContactSearch` business class. This includes populating the *Type* dropdown based on the `ContactType` enum. The `Search`, `Add Personal Contact` and `Add Business Contact` buttons are created by `SwingMetawidget` based on annotated methods in the `ContactDialog` class.

**Note**

To view the source code for the examples, such as the code for the `ContactSearch`, `ContactType` and `ContactDialog` classes, download the Metawidget source code distribution or browse it online at <http://metawidget.svn.sourceforge.net/viewvc/metawidget/trunk/examples/src>.

Click `Add Personal Contact`. The screen displays a form for filling out Personal Contact information as in [Figure 1.13](#).

Figure 1.13. Desktop Address Book 'Add Personal Contact' screen

All the form fields are created by `SwingMetawidget` based on the `PersonalContact` business class. This class is itself derived from the `Contact` business class. It includes some Metawidget annotations for dropdown values and section headings.

Note the code only has one `JDialog` class (`ContactDialog`), but is capable of supporting both `PersonalContact` and `BusinessContact` UIs. The fields in the UI change depending on the object passed to `ContactDialog` at runtime. This is the Third Goal Of Metawidget:



Third Goal Of Metawidget

Metawidget performs inspection *at runtime*, detecting types and subtypes dynamically

The *Address* field is created as a nested `SwingMetawidget`. This is the default behaviour when Metawidget encounters datatypes it does not know how to represent with any other UI widget. The *Communications* field has been overridden with a manually specified `JTable`.

In addition, `JTable.setCellEditor` uses `SwingMetawidget` to render single `JComponents` as `CellEditors`. This includes automatically populating dropdown values.

Read-Only Mode

The Desktop Address Book uses Metawidget's `setReadOnly(true)` method to display read-only screens. Return to the main screen, and double-click on an existing contact (such as Homer Simpson). The same `ContactDialog` is used, but this time all the widgets are read-only labels as in [Figure 1.14](#).

Mr Homer Simpson - Personal Contact

Title: Mr
 Firstnames: Homer
 Surname: Simpson
 Date of Birth: 12/05/56
 Gender: Male

Contact Details

Address: Street: 742 Evergreen Terrace
 City: Springfield
 State: Anytown
 Postcode: 90701

Communications:

Type	Value
Telephone	(939) 555-0113

Other

Notes:

Edit Cancel

Figure 1.14. Desktop Address Book read-only mode

Click `Edit`. The labels are transformed into editable widgets by using Metawidget's `setReadOnly(false)`, as in [Figure 1.15](#).

Mr Homer Simpson - Personal Contact

Title: Mr
 Firstnames: Homer
 Surname: Simpson
 Gender: Male
 Date of Birth: 12/05/56

Contact Details

Address: Street: 742 Evergreen Terrace
 City: Springfield
 State: Anytown
 Postcode: 90701

Communications:

Type	Value
Telephone	(939) 555-0113

Other

Notes:

Save Delete Cancel

Figure 1.15. Desktop Address Book edit mode

Binding

The data from the `PersonalContact` object is automatically inserted into the `JComponents`. It is also automatically saved back when clicking `Save`.

Swing does not define a `JComponent` to Object mapping mechanism, so by default `SwingMetawidget` only supplies `setValue` and `getValue` methods for manually fetching values. This situation is no worse than a normal Swing application, but Metawidget can do better.

`SwingMetawidget` directly supports third-party binding alternatives such as [Apache BeanUtils](#) and [Beans Binding \(JSR 295\)](#) via `SwingMetawidget.addWidgetProcessor`. These binding implementations automatically map `JComponent`

values to `Object` values, including performing the necessary conversions, further reducing the amount of boilerplate code required.

Localization

All text within the application has been localized to the `org.metawidget.example.shared.addressbook.resource.Resources` resource bundle. Text created manually (such as the buttons) uses typical Swing localization code (eg. `bundle.getString`). Text created by `SwingMetawidget` uses `SwingMetawidget.setBundle`, which internally defers to `bundle.getString`.

Localization is very easy with Metawidget. For field names, if no resource bundle is supplied, Metawidget uses an 'uncamel-cased' version of the name. If a bundle *is* supplied, Metawidget uses the field name as the bundle key. For section labels, if a bundle is supplied, Metawidget uses a 'camel-cased' version of the label as the key.

This means developers can initially build their UIs without worrying about resource bundles, then turn on localization support later.

1.2.2 Web Address Book

As there are a large number of Java Web application frameworks to choose from, this example comes written in five of the most popular: Google Web Toolkit (GWT), Java Server Faces (JSF), Java Server Pages (JSP), Spring Web MVC and Struts. We recommend you follow along using the one most relevant to you.

Web-based applications are inherently more difficult to setup and run than desktop-based applications because they require a container application. For this tutorial, we will use Apache Tomcat 6 (Tomcat), as it is one of the easier containers to get running. Tomcat can be downloaded from <http://tomcat.apache.org>.

Take a fresh install of Tomcat. The Address Book WAR is pre-built for you in either `examples\faces\addressbook-faces.war`, `examples\gwt\addressbook-gwt.war`, `examples\jsp\addressbook-jsp.war`, `examples\spring\addressbook-spring.war` or `examples\struts\addressbook-struts.war`. If you've downloaded the source code distribution, you can build it yourself by changing to the `examples` folder of the source distribution and typing:

```
ant example-faces-addressbook
```

(replacing *faces* with *gwt*, *jsp*, *spring* or *struts* as appropriate).



Note

For most web environments, deploying Metawidget is as simple as adding `metawidget.jar` to `WEB-INF\lib`. For GWT, you'll also need to include `metawidget.jar` and `examples\gwt\metawidget-gwt.jar` in the `CLASSPATH` during your `GWTCompiler` step.

Copy the WAR into Tomcat's `webapps` folder, start Tomcat, and open a Web browser to <http://localhost:8080/addressbook-faces>. The home page displays a search filter (at the top) and lists existing Address Book entries (at the bottom) as in [Figure 1.16](#).

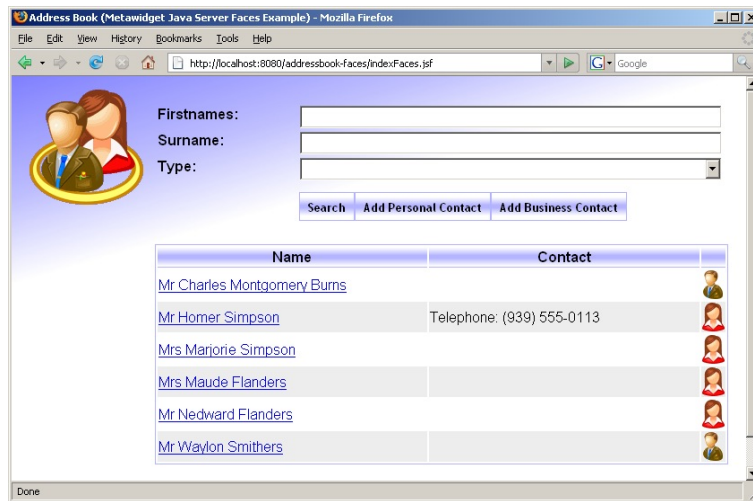


Figure 1.16. Web Address Book opening screen

As with the Desktop Address Book, the three search filter fields are created by Metawidget (this time `UIMetawidget`, `GwtMetawidget`, `HtmlMetawidgetTag` or `StrutsMetawidgetTag`) based on the `ContactSearch` business class:

```
<m:metawidget value="#{contact.search}">
...
</m:metawidget>
```

Again, this includes populating the `Type` dropdown and localizing the text. The `Search`, `Add Personal Contact` and `Add Business Contact` buttons are either manually specified in the JSP page (for GWT, Spring and Struts) or created by `UIMetawidget` based on annotated methods in the `ContactBean` (for JSF).



Note

As with the Desktop Address Book, all source code for the examples can be found in the source code distribution under `examples/src/java/org/metawidget/example`. All Web-specific resources (such as JSP files) can be found under `examples/src/web`.

The look of the Web page relies entirely on HTML and CSS technologies. These are configured in `metawidget.xml`:

```
<htmlMetawidget>
  <parameter>
    <string>tableStyleClass</string>
    <string>table-form</string>
  </parameter>
  <parameter>
    <string>columnClasses</string>
    <string>table-label-column,table-component-column,table-required-column</string>
  </parameter>
  ...
</htmlMetawidget>
```

Only the layout of 'one column for the label, one column for the widget' is dictated by Metawidget, and that is again pluggable and configurable.

Click `Add Personal Contact`. The page displays a form for filling out Personal Contact information as in [Figure 1.17](#).

Figure 1.17. Web Address Book 'Add Personal Contact' screen

All the form fields are created by Metawidget based on the `PersonalContact` business class. The section headings are the same, but have this time been rendered as HTML.

The `Address` field is a nested Metawidget. The `Communications` field has been overridden in the JSP page with a manually specified table. `UIMetawidget` understands a manually-specified widget to override an automatic one if it has the same `value` binding as the automatic widget would have (`GwtMetawidget`, `SpringMetawidgetTag` and `StrutsMetawidgetTag` do something similar):

```
<m:metawidget value="#{contact.current}">
...
<h:dataTable value="#{contact.current.communications}">
...
</h:dataTable>
...
</m:metawidget>
```

JSF has built-in support for executing actions on table rows. In order to use it, however, the `Set` returned by `Contact.getCommunications` must be wrapped into a `DataModel`. This is handled by `ContactController.getCurrentCommunications`, but this presents a problem: the mapping for the `HtmlDataTable` must be `#{contact.currentCommunications}`, but the mapping required to override `UIMetawidget`'s automatic widget creation is `#{contact.current.communications}`.

`UIMetawidget` supplies `UIStub` for these situations. Stubs have a binding, but do nothing with it and render nothing. They can be used either to suppress widget creation entirely (a stub with an empty body) or to replace the automatic widget creation with one or more other widgets with different bindings:

```
<m:metawidget value="#{contact.current}">
...
<m:stub value="#{contact.current.communications}">
  <h:dataTable value="#{contact.currentCommunications}">
```

```
...
</h:dataTable>
</m:stub>
...
<m:metawidget>
```

JSP, Spring, Struts lack some component-based features found in Swing and JSF. Specifically, whilst it is possible for tags to reference their *parent* (using `TagSupport.findAncestorWithClass`), they have no way to interrogate their *children*. Therefore, it is not possible to directly support arbitrary child tags within `HtmlMetawidget`, `SpringMetawidgetTag` and `StrutsMetawidgetTag`.

Instead, we wrap the overridden `Communications` field in Metawidget's `Stub` tag. Metawidget and its `Stub` tags have explicit support for co-ordinating the overriding of widget creation:

```
<m:metawidget property="contactForm">
...
<m:stub property="communications">
  <table class="data-table">
    ...
  </table>
</m:stub>
...
<m:metawidget>
```

`GwtMetawidget` uses stubs around GWT widgets like `FlexTable`, but can use the overriding widget directly if it supports the `HasName` interface (eg. `TextBox`, `CheckBox`, etc).

Mixing Metawidgets

The section is specific to Spring/Struts.

Within the `Communications` table, implementing `Add Communication` calls for a design decision. Struts does not support multiple `ActionForms` per `Action`, so we are unable to combine `PersonalContactForm` with a `CommunicationForm` (as we did in the JSF). Spring has a similar limitation of not supporting multiple `commandNames` per form. Instead, we need to either:

- add fields from `Communication` to `PersonalContactForm`, and ignore them when saving the `PersonalContact`; or
- output plain HTML tags (ie. independent of Spring and Struts) and handle them manually

Both approaches would be valid. For this tutorial, we choose the latter as it allows us to introduce `HtmlMetawidget` (a Metawidget for plain HTML/JSP webapps that don't use Struts or Spring) and demonstrate *mixing* two Metawidgets on the same page:

```
<m:metawidget property="contactForm">
...
<m:stub property="communications">
  <table class="data-table">
    ...
    <tr>
      <jsp:useBean id="communication"
        class="org.metawidget.example.shared.addressbook.model.Communication"/>
      <td><mh:metawidget value="communication.type" style="width: 100%" /></td>
      <td><mh:metawidget value="communication.value" style="width: 100%" /></td>
    </tr>
    ...
  </table>
```



```

</m:stub>
...
</m:metawidget>

```

The two different tag prefixes *m:* and *mh:* denote different tag libraries. `HtmlMetawidget` is very similar to `StrutsMetawidgetTag`, but has to use `jsp:useBean` to manually instantiate the bean (rather than letting Struts do it). Within `metawidget.xml`, the default layout for `HtmlMetawidget` has been set to `org.metawidget.jsp.tagext.layout.SimpleLayout` (ie. a plain layout, without a label column)

Expression Based Lookups

This section does not apply to GWT.

In the Desktop Address Book, the *title* dropdown was populated by a static *lookup* attribute in `metawidget-metadata.xml`. JSP and JSF-based technologies can do better, because they have a built-in scope-based component model and Expression Language.

`Contact.getTitle` is annotated using `@UiFacesLookup` and `@UiSpringLookup` (and `ContactForm.getTitle` is annotated using `@UiStrutsLookup`). These are used at runtime to create dynamic lookups.

These annotations, unlike the ones we have used so far, *are* UI-framework specific so you may prefer to declare them in `metawidget-metadata.xml`. Before doing so, however, you should understand we are still not introducing runtime dependencies into our business classes: an important feature of annotations is they 'fall away gracefully' if their implementing class is not found. Annotations never throw `ClassDefNotFoundError`.

Alternate Widget Libraries

This section is specific to JSF.

Metawidget factors all widget creation into `WidgetBuilders`. Like `Inspectors`, multiple `WidgetBuilders` can be combined using a `CompositeWidgetBuilder` to support third-party component libraries. In this section we will override Metawidget's default and introduce a `RichFacesWidgetBuilder` alongside the standard JSF `HtmlWidgetBuilder`.

Go into Tomcat's `webapps\addressbook-faces` folder (the exploded WAR) and edit `WEB-INF/metawidget.xml`:

```

<metawidget xmlns="http://metawidget.org"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="1.0"
  xsi:schemaLocation="http://metawidget.org http://metawidget.org/xsd/metawidget-1.0.xsd">

  <htmlMetawidget xmlns="java:org.metawidget.faces.component.html">
    ...
    <inspector>
      <compositeInspector xmlns="java:org.metawidget.inspector.composite"
        config="CompositeInspectorConfig">
        <inspectors>
          <array>
            <propertyTypeInspector xmlns="java:org.metawidget.inspector.propertytype"/>
            <metawidgetAnnotationInspector xmlns="java:org.metawidget.inspector.annotation"/>
            <java5Inspector xmlns="java:org.metawidget.inspector.java5"/>
            <facesInspector xmlns="java:org.metawidget.inspector.faces"/>
            <xmlInspector xmlns="java:org.metawidget.inspector.xml" config="XmlInspectorConfig"/>
          </array>
        </inspectors>
      </compositeInspector>
    </inspector>
  </widgetBuilder>

```

```

<compositeWidgetBuilder xmlns="java:org.metawidget.widgetbuilder.composite"
  config="CompositeWidgetBuilderConfig">
  <widgetBuilders>
    <array>
      <overriddenWidgetBuilder xmlns="java:org.metawidget.faces.component.widgetbuilder"/>
      <readOnlyWidgetBuilder xmlns="java:org.metawidget.faces.component.html.widgetbuilder"/>
      <richFacesWidgetBuilder xmlns="java:org.metawidget.faces.component.html.widgetbuilder.richfaces"/>
      <htmlWidgetBuilder xmlns="java:org.metawidget.faces.component.html.widgetbuilder"/>
    </array>
  </widgetBuilders>
</compositeWidgetBuilder>
</widgetBuilder>
<layout>
  <outputTextLayoutDecorator xmlns="java:org.metawidget.faces.component.html.layout"
    config="OutputTextLayoutDecoratorConfig">
    <layout>
      <simpleLayout xmlns="java:org.metawidget.faces.component.layout"/>
    </layout>
    <styleClass>
      <string>section-heading</string>
    </styleClass>
  </outputTextLayoutDecorator>
</layout>
</htmlMetawidget>
</metawidget>

```

Now restart Tomcat, refresh your Web browser and click on Homer Simpson. Notice how the *Date of Birth* field for Personal Contacts is now a RichFaces date picker widget, and the *Number of Staff* field for Business Contacts is a RichFaces slider widget.

Going further, Metawidget's pluggable layouts make it easy to support third-party layout components. Edit WEB-INF/metawidget.xml again:

```

<metawidget xmlns="http://metawidget.org"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="1.0"
  xsi:schemaLocation="http://metawidget.org http://metawidget.org/xsd/metawidget-1.0.xsd">
  <htmlMetawidget xmlns="java:org.metawidget.faces.component.html">
    ...
    <inspector>
      <compositeInspector xmlns="java:org.metawidget.inspector.composite"
        config="CompositeInspectorConfig">
        <inspectors>
          <array>
            <propertyTypeInspector xmlns="java:org.metawidget.inspector.propertytype"/>
            <metawidgetAnnotationInspector xmlns="java:org.metawidget.inspector.annotation"/>
            <java5Inspector xmlns="java:org.metawidget.inspector.java5"/>
            <facesInspector xmlns="java:org.metawidget.inspector.faces"/>
            <xmlInspector xmlns="java:org.metawidget.inspector.xml" config="XmlInspectorConfig"/>
          </array>
        </inspectors>
      </compositeInspector>
    </inspector>
    <widgetBuilder>
      <compositeWidgetBuilder xmlns="java:org.metawidget.widgetbuilder.composite"
        config="CompositeWidgetBuilderConfig">
        <widgetBuilders>

```

```

<array>
  <overriddenWidgetBuilder xmlns="java:org.metawidget.faces.component.html.widgetbuilder"/>
  <readOnlyWidgetBuilder xmlns="java:org.metawidget.faces.component.html.widgetbuilder"/>
  <richFacesWidgetBuilder xmlns="java:org.metawidget.faces.component.html.widgetbuilder.richfaces"/>
  <htmlWidgetBuilder xmlns="java:org.metawidget.faces.component.html.widgetbuilder"/>
</array>
</widgetBuilders>
</compositeWidgetBuilder>
</widgetBuilder>
<layout>
  <tabPanelLayoutDecorator xmlns="java:org.metawidget.faces.component.html.layout.richfaces"
    config="TabPanelLayoutDecoratorConfig">
    <layout>
      <simpleLayout xmlns="java:org.metawidget.faces.component.layout"/>
    </layout>
  </tabPanelLayoutDecorator>
</layout>
</htmlMetawidget>

</metawidget>

```

Restart Tomcat, refresh your Web browser and click on Homer Simpson again. Notice how the *Contact Details* and *Other* sections are laid out as tabs within a RichFaces TabPanel as in [Figure 1.18](#).

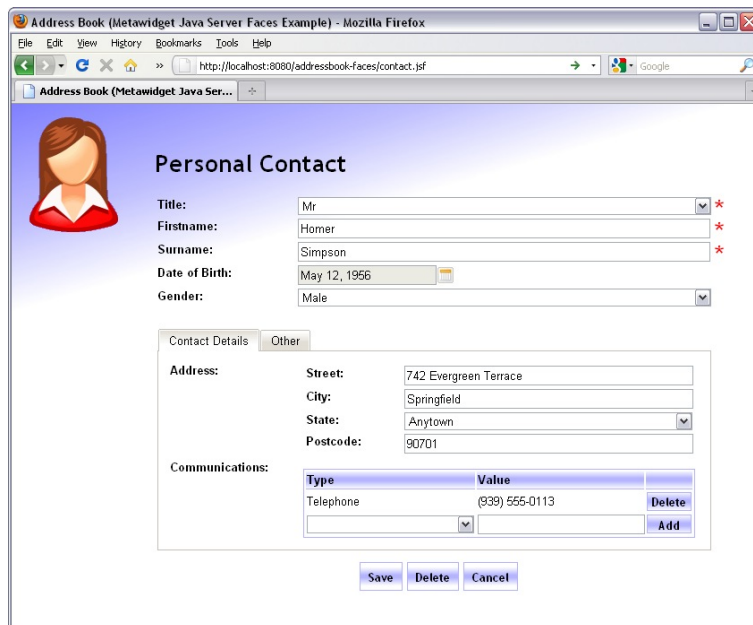


Figure 1.18. Web Address Book using JBoss RichFaces

This demonstrates how easy it is to leverage widget libraries with Metawidget (this example cheats a bit, as we've pre-added the RichFaces JARs into `WEB-INF\lib` and some lines into `web.xml`, but you get the idea).

1.2.3 Mobile Address Book

For the Mobile Address Book we use the Android platform. Android has uniquely strong support for reflection and annotations, and guarantees the availability of key packages such as `org.w3c.dom`. This affords Metawidget excellent runtime access to inspect the *O* in OIM.

Like Web-based applications, Mobile applications require a container to run. Download the Android SDK (Metawidget has been tested against versions 1.1, 1.5, 1.6 and 2.0) from <http://code.google.com/android/download.html>. Then change to the installation directory (usually `android-sdk-windows`) and run the emulator by opening a command prompt and typing:

```
tools\android create avd -n my_avd -t 1
```

Replacing `-t 1` with the Android version you're using (eg. `-t 2` for 1.5, `-t 3` for 1.6, `-t 4` for 2.0). Android 1.1 doesn't use AVDs, so you can skip this step. Next type:

```
tools\emulator -avd my_avd
```

The emulator may take a little while to start. Once finished, it will display the phone's desktop. The Address Book APK is pre-built for you in `examples\android\addressbook-android.apk`. If you've downloaded the source code distribution, you can build it yourself by changing to the `examples` folder of the source distribution and typing:

```
ant example-android-addressbook
```

(ensuring your `build.properties` is set correctly to point to the Android SDK).

Next, open a *second* command prompt, change to the Android installation directory and type:

```
tools\adb install <metawidget folder>\examples\android\addressbook-android.apk
```

This deploys the APK into the emulator. To run it, click the emulator's *Menu* button and then choose the Address Book application. The emulator displays a search filter (at the top) and lists existing Address Book entries (at the bottom) as in [Figure 1.19](#).

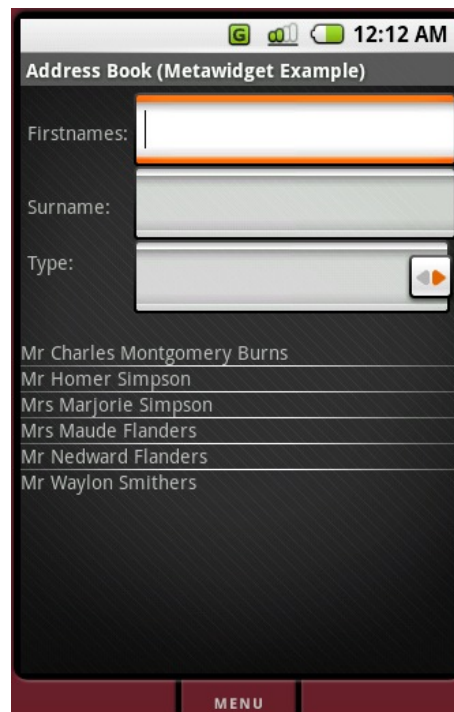


Figure 1.19. Mobile Address Book opening screen

As with the Desktop and Web Address Books, the three search filter fields are created by Metawidget (this time `AndroidMetawidget`) based on the `ContactSearch` business class. Again, this includes populating the `Type` dropdown.

**Note**

As with the Desktop Address Book, all source code for the examples can be found in the source code distribution under `examples/src/java/org/metawidget/example`. All Android-specific resources (such as XML files) can be found under `examples/src/android`.

The look of the screen relies entirely on Android XML layout files, styles and themes. Only the 'one column for the label, one column for the widget' layout is dictated by Metawidget, and that is pluggable and configurable.

Choose `Add Personal` from the Android menu. The page displays a form for filling out Personal Contact information as in [Figure 1.20](#).

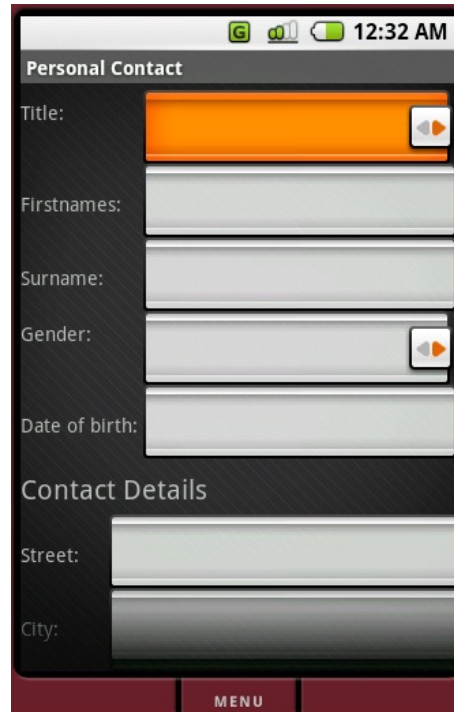


Figure 1.20. Mobile Address Book 'Add Personal Contact' screen

UIs in Android are typically defined in XML layout files, though they can also be built programmatically. AndroidMetawidget supports both approaches. For example, the Personal Contact screen is defined in `contact.xml`, and contains a Metawidget defined in much the same way as in JSP (including configuring section style and overriding widget creation):

```
<view class="org.metawidget.android.widget.AndroidMetawidget" android:id="@+id/metawidget"
    config="@raw/metawidget">

    <view class="org.metawidget.android.widget.Stub" tag="communications">

        <ListView android:id="@id/communications" ... />

        <Button android:id="@+id/buttonAddCommunication"
            android:text="@string/addCommunication" ... />

    </view>
</view>
```

Within `CommunicationDialog`, a Metawidget is defined programmatically in much the same way as in Swing:

```
mMetawidget = new AndroidMetawidget( activity );  
mMetawidget.setConfig( R.raw.config );  
...  
mMetawidget.setToInspect( mCommunication );
```

This produces the dialog box in [Figure 1.21](#).

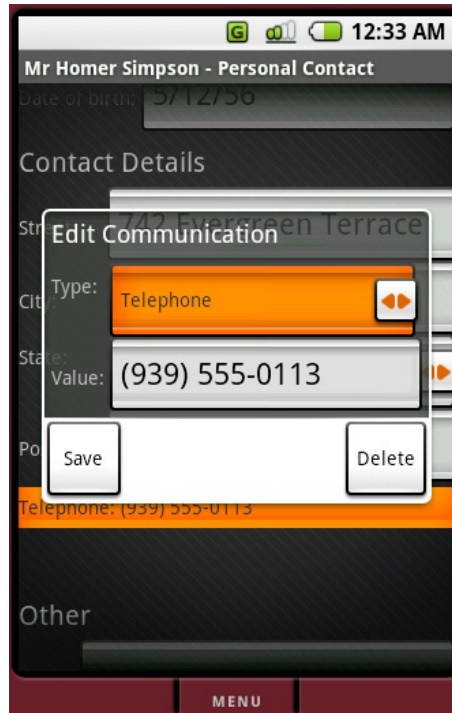


Figure 1.21. Mobile Address Book Communications Dialog

1.2.4 Conclusion

That concludes the introductory tutorial. In summary, we now:

- have seen how to build an application whose UI is largely dictated by its business classes, not by hand-written UI code
- significantly reduced the amount of UI code needed in our applications
- have seen how to build an application that targets multiple platforms. If we were to add a new field to one of the business classes (say, *numberOfChildren* to *PersonalContact*), it would automatically appear and be functional on every platform.

1.3 Part 3 - Other Examples

The Metawidget distribution includes other examples showcasing particular features on particular platforms. These additional examples are not a required part of the tutorial, but you may find them useful depending on which platform you use.

1.3.1 Swing Applet Address Book Example

The Swing Applet Address Book Example demonstrates using Metawidget in applets. The example is pre-built for you in `examples\swing\applet\addressbook`. If you've downloaded the source code distribution, you can build it yourself by changing to the `examples` folder of the source distribution and typing:

```
ant example-swing-addressbook-applet
```

To run the applet, open the `index.html` file in a Web browser. The code is identical to the Swing Address Book covered in Part 2 of this tutorial, except it uses `org.metawidget.example.swing.applet.AddressBookApplet` instead of `org.metawidget.example.swing.addressbook.MainFrame`.

The notable feature of the example is how the applet is packaged. Metawidget is highly modular and has no mandatory third-party JAR dependencies. The `example-swing-addressbook-applet` Ant task builds only those inspectors necessary for the Address Book application. The resulting `metawidget-applet.jar` is then further compressed using `pack200`. The small download size makes Metawidget very viable for Applet-based environments.

1.3.2 Seam Example

The Seam Booking Example demonstrates updating an existing Seam application to use Metawidget, reducing boilerplate code. The example requires you to have previously downloaded [Seam 2.2.0.GA](#) and [JBoss 5.1.0.GA](#), and you should be [familiar with the existing Seam Booking application](#).

The example is located in `examples\faces\seam\booking`. It is not pre-built. To build it, change to the `examples\faces\seam\booking` folder in the Metawidget binary distribution and type:

```
ant
```

To run it, type:

```
cd \Applications\jboss-5.1.0.GA  
bin\run
```

Open a Web browser to <http://localhost:8080/seam-booking>. The updated Metawidget Seam Booking Example looks very similar to the original, as in [Figure 1.22](#), but uses significantly less boilerplate code.

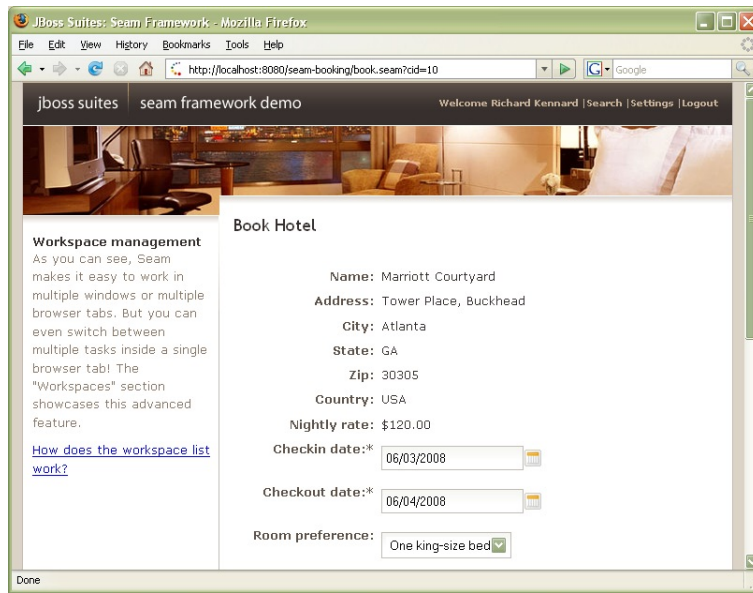


Figure 1.22. Seam Booking with Metawidget

The files modified for adding Metawidget support are in `examples\faces\seam\booking`. Most of the UI code in `view\book.xhtml`, `view\confirm.xhtml` and `view\hotelview.xhtml` has been replaced with a single Metawidget tag. Some annotations have been added to `Hotel.java` and `Booking.java`, though Metawidget also leverages the existing JPA and Hibernate Validator ones.

The example further demonstrates packaging Metawidget for multi-tier environments. The `pack-as-frontend-backend` Ant task in Metawidget's `build.xml` creates two JAR files: `metawidget-frontend.jar` for deployment in the WAR, and `metawidget-backend.jar` for deployment in the EJB layer. This ensures a clean separation between frontend and backend code, and is important for avoiding WAR/EJB classloading problems.

1.3.3 Groovy Example

The Seam Groovy Booking Example demonstrates updating an existing Seam Groovy application to use Metawidget, reducing boilerplate code. The example is a more advanced version of the previous Seam section, so you should work through that first.

The example is located in `examples\faces\seam\groovybooking`. It is not pre-built. To build it, change to the `examples\faces\seam\groovybooking` folder in the Metawidget binary distribution and type:

```
ant
```

To run it, type:

```
cd \Applications\jboss-5.1.0.GA
bin\run
```

Open a Web browser to <http://localhost:8080/jboss-seam-groovybooking>. As with the previous section, the updated Metawidget Seam Groovy Booking Example looks very similar to the original, but uses significantly less boilerplate code. Also, this time we are using Groovy to define our business classes. The biggest impact this has is in `metawidget.xml`, where the inspectors have been configured to use a Groovy property style instead of a JavaBean property style.

Metawidget supports pluggable 'property styles' for JavaBean, Groovy and other property styles. Groovy properties differ from JavaBean properties in that their annotations are tied to the private member variable, rather than the getters and setters.

The use of Groovy is configured per Inspector, as in `examples\faces\seam\groovybooking\resources\WEB-INF\metawidget.xml`:

```
<propertyTypeInspector config="org.metawidget.inspector.impl.BaseObjectInspectorConfig">
  <propertyStyle>
    <groovyPropertyStyle xmlns="java:org.metawidget.inspector.impl.propertystyle.groovy"/>
  </propertyStyle>
</propertyTypeInspector>
```

1.3.4 jBPM Example

The Seam DVD Store Example demonstrates updating an existing Seam jBPM application to use Metawidget, reducing boilerplate code. The example requires you to have previously downloaded Seam 2.2.0.GA and JBoss 5.1.0.GA, and you should be familiar with the existing Seam DVD Store application.

The example is located in `examples\faces\seam\dvdstore`. It is not pre-built. To build it, change to the `examples\faces\seam\dvdstore` folder in the Metawidget binary distribution and type:

```
ant
```

To run it, type:

```
cd \Applications\jboss-5.1.0.GA
bin\run
```

Open a Web browser to <http://localhost:8080/seam-dvdstore>. As with the previous two sections, the updated Metawidget Seam DVD Store example looks very similar to the original, as in [Figure 1.23](#), but uses significantly less boilerplate code.

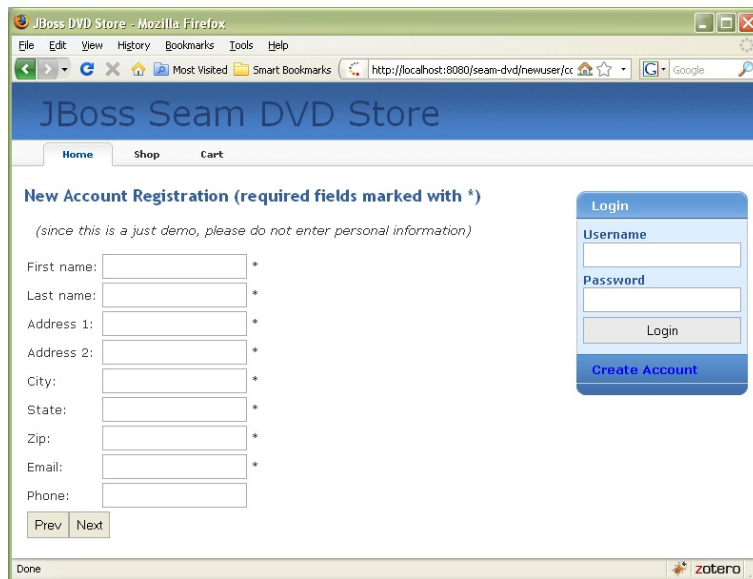


Figure 1.23. Seam DVD Store with Metawidget

This time, as well as generating `UIComponents` for business objects such as `com.jboss.dvd.seam.Customer`, Metawidget inspects jBPM pageflow files like `newuser.jpdl.xml` and `checkout.jpdl.xml` to generate the correct `UICommand` buttons for each screen.

1.3.5 ICEfaces Example

[ICEfaces](#) is an AJAX component library for Java Server Faces. This example showcases how Metawidget can work with ICEfaces (1.8.2 and above) to deliver rich AJAX applications.

The example is pre-built for you in `examples\faces\penguincolony-faces.war`. Copy the WAR into Tomcat's `webapps` folder, start Tomcat, and open a Web browser to <http://localhost:8080/penguincolony-faces/>:

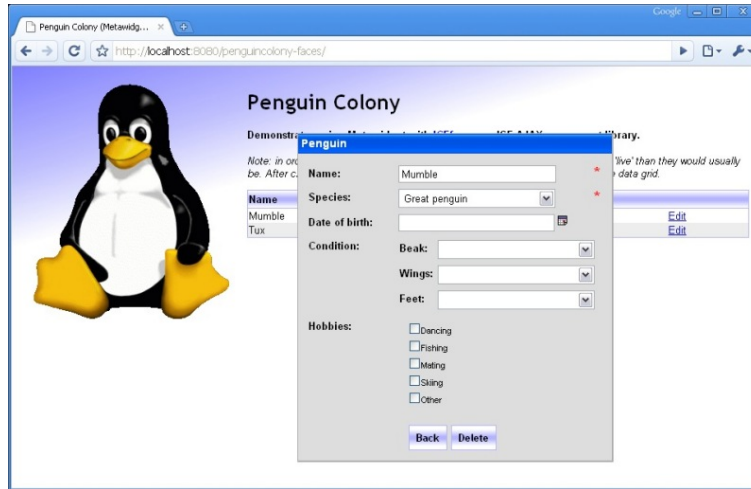


Figure 1.24. ICEfaces with Metawidget

The application manages details of penguins in a colony. To begin, click the `Edit` link in the first row of the table: ICEfaces and Metawidget work together to pop up an AJAX form without refreshing the page. Next, clear the form's `Name` box and tab to the next field: an AJAX call is made and a validation error appears. Try entering a new name and tabbing again: the validation error disappears and the new name is immediately reflected in the table behind the popup box.

The example also makes use of the `@UiFacesAttribute` annotation. For example, the Java code is annotated:

```
@UiAction
@UiFacesAttribute( name = InspectionResultConstants.HIDDEN,
    expression = "#{!empty this.condition}" )
public void addCondition() { ... }

@UiFacesAttribute( name = InspectionResultConstants.HIDDEN,
    expression = "#{empty this.condition}" )
public PenguinCondition getCondition() { ... }
```

Clicking the `Add Condition` button, or checking one of the `Hobbies` checkboxes, triggers an AJAX call that re-evaluates the `@UiFacesAttribute` annotations and dynamically reconstructs the form without requiring a page refresh. This includes removing existing buttons, creating new dropdown boxes and creating new labels.

For more details on ICEfaces support, see [the section called “IceFacesWidgetBuilder”](#).

1.3.6 Swing AppFramework Example

The Swing AppFramework Car Demo demonstrates using Metawidget with the [Swing AppFramework](#). Metawidget can use Swing AppFramework's `@Action` annotation to identify actions, both amongst a business object's properties and in external controllers, and automatically generate `JButtons` for them.

The application is pre-built for you in `examples\swing\appframework-swing.jar` or, if you've downloaded the source code distribution, you can build it yourself by changing to the `examples` folder of the source distribution and typing:

```
ant example-swing-appframework
```

**Note**

This example uses annotations, so you'll need Java SE 5 or higher

This is a self-executing JAR. For convenience, it has `MANIFEST.MF` dependencies hard-wired into it to `..\..\metawidget.jar` and `lib\AppFramework.jar` among others, so it's best not to move it to a different folder (if you do, you'll need to manually put those JARs on your classpath).

Run the code by navigating to the `examples\swing` folder and typing:

```
java -jar appframework-swing.jar
```

The opening screen displays two fields to allow you to enter the make and type of a car. You can also optionally add an owner by clicking the `Add an Owner` button, or save the car using the `Save` button.

The `Add an Owner` button is generated by Metawidget based on the `addOwner` method in the `Car` class (which has been annotated `@org.jdesktop.application.Action`). The `Save` button is generated based on the `save` method in the `CarApplication` class (also annotated `@Action`). Two different Metawidgets are used in the example: one pointed at the `Car` class, the other at the `CarApplication` class.

Metawidget supports pluggable 'action styles'. The use of Swing `AppFramework` is configured per `Inspector`, as in `examples/src/java/org/metawidget/example/swing/appframework/metawidget.xml` from the source distribution:

```
<metawidgetAnnotationInspector
  config="org.metawidget.inspector.impl.BaseObjectInspectorConfig">
  <actionStyle>
    <swingAppFrameworkActionStyle xmlns="java:org.metawidget.inspector.impl.actionstyle.swing">
  </actionStyle>
</metawidgetAnnotationInspector>
```

As a further feature, after the `Add an Owner` button is clicked it disappears. This is achieved by using `JexlInspector` to introduce an expression language for Swing similar to JSP's EL. The method is annotated...

```
@Action( name = "add" )
@UiJexlAttribute( name = HIDDEN, value = "this.owner != null" )
public void addOwner() {
    mOwner = new Owner();
    fireActionEvent( "addOwner" );
}
```

...such that the button gets hidden when the car has an owner.

1.3.7 Scala Example

The Scala Animal Races Example demonstrates using `SwingMetawidget` together with [Scala](#) and [MigLayout](#).

The application is pre-built for you in `examples\swing\animalraces-swing.jar` or, if you've downloaded the source code distribution, you can build it yourself by changing to the `examples` folder of the source distribution and typing:

```
ant example-swing-animalraces
```

**Note**

This example uses annotations, so you'll need Java SE 5 or higher

This is a self-executing JAR. For convenience, it has `MANIFEST.MF` dependencies hard-wired into it to `..\..\metawidget.jar` and `lib\scala-library.jar` among others, so it's best not to move it to a different folder (if you do, you'll need to manually put those JARs on your classpath).

Run the code by navigating to the `examples\swing` folder and typing:

```
java -jar animalraces-swing.jar
```

The screen displays fields to allow you to change the name, speed and type of each animal as well buttons to start and stop the race.

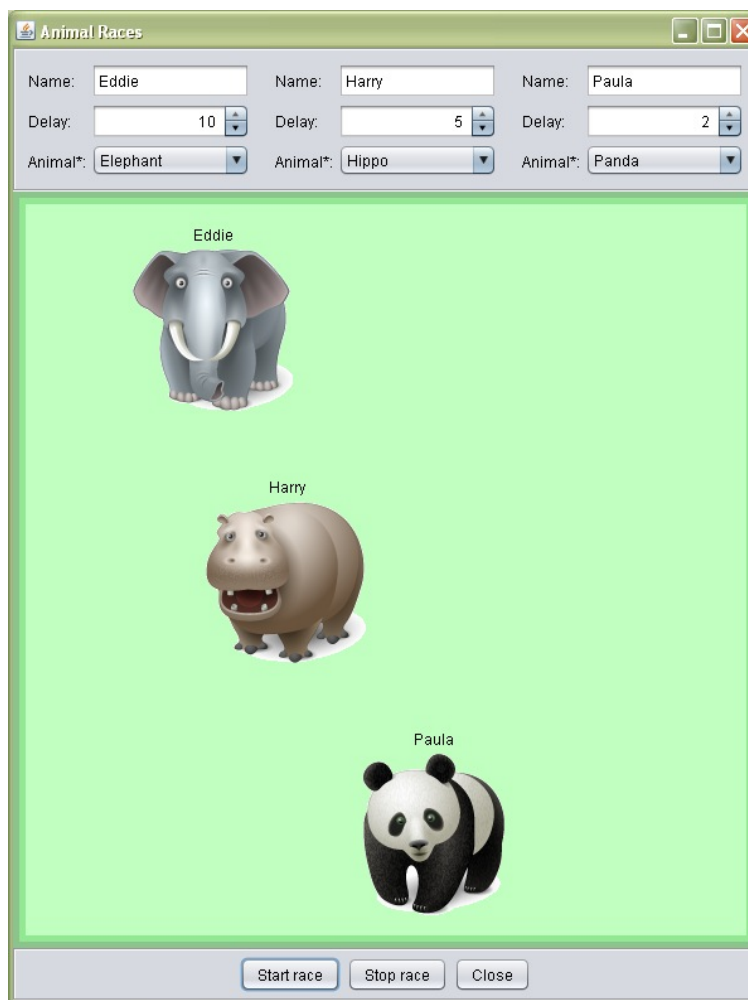


Figure 1.25. Scala and MigLayout with Metawidget

Animal Races' whimsical User Interface demonstrates how Metawidget's goal of not 'owning' the UI allows multiple Metawidgets to be combined for unconventional UIs. There are three Metawidgets across the top (one for each animal in the race), and a fourth Metawidget for the buttons at the bottom.

The top three Metawidgets all use MigLayout. Because Metawidget does not hide the underlying UI framework, using MigLayout allows the Animal Races code to easily pad the Metawidget:

```
metawidget.setMetawidgetLayout( new MigLayout() );
metawidget.setToInspect( animal );
((MigLayout) metawidget.getLayout()).setLayoutConstraints( new LC().insets( "10" ) );
```

The Animal Races code is written purely in Scala, located at `examples/src/scala/org/metawidget/example/swing/animalraces/AnimalRaces.scala`. It uses `ScalaPropertyStyle` to allow Metawidget to inspect Scala-based business objects:

```
<metawidgetAnnotationInspector xmlns="java:org.metawidget.inspector.annotation"
  config="org.metawidget.inspector.impl.BaseObjectInspectorConfig">
  <propertyStyle>
    <scalaPropertyStyle xmlns="java:org.metawidget.inspector.impl.propertystyle.scala"/>
  </propertyStyle>
</metawidgetAnnotationInspector>
```

In addition, it uses `BeanUtilsBindingProcessorConfig.PROPERTYSTYLE_SCALA` to bind JComponents to Scala-based business objects:

```
val metawidget = new SwingMetawidget()
metawidget.addWidgetProcessor( new BeanUtilsBindingProcessor()
  .setPropertyStyle( BeanUtilsBindingProcessorConfig.PROPERTYSTYLE_SCALA ) )
metawidget.setToInspect( animal )
```

1.3.8 GWT Client Side Example

By default, `GwtMetawidget` inspects business objects server-side using `GwtRemoteInspector`. This is because client-side JavaScript does not support reflections or annotations. However if you don't need reflections or annotations, and have your own way of retrieving inspection results, you can plug in your own inspector and keep everything client-side. This example implements a `TextAreaInspector` that retrieves inspection results from a textarea and generates the UI.

The example is pre-built for you in `examples\gwt\clientside-gwt`. Because everything is purely client side, there is no need for Tomcat or any other container: simply navigate to the example folder and open `index.html` in your Web browser:

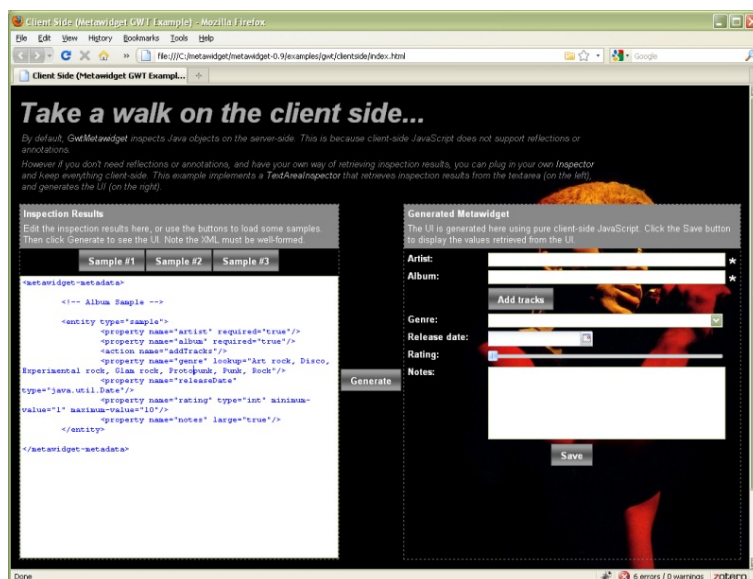


Figure 1.26. Client-Side GwtMetawidget

The example shows a textbox on the left containing inspection results in `inspection-result-1.0.xsd` format. The result of generating this XML is shown on the right. Click the `Sample #2` and `Sample #3` buttons to preload different XML samples, and the `Generate` button to generate their UI. Alternatively, you can edit the XML by hand to add new properties and actions and click `Generate`.

Data binding and event binding are also implemented client side. Click the `Save` to save the data from the generated UI using a `MapPropertyBinding` (in a real application, this `Map` could be passed back to the server for persisting). Click the `Add Tracks` button to trigger an event binding.

Finally, this example showcases using third-party GWT component libraries. `ExtGwtWidgetBuilder` is used to render a date picker widget. For more details on [ExtGWT](#) support, see the section called “`ExtGwtWidgetBuilder`”.

1.3.9 GWT Hosted Mode Examples

The examples\gwt\addressbook-gwt.war (discussed in Part 2 of this tutorial) and the examples\gwt\clientside (discussed in Part 3 of this tutorial) demonstrates GWT running in *GWT Web mode*. Developers may prefer instead to run the examples in *GWT hosted mode* as in [Figure 1.27](#).

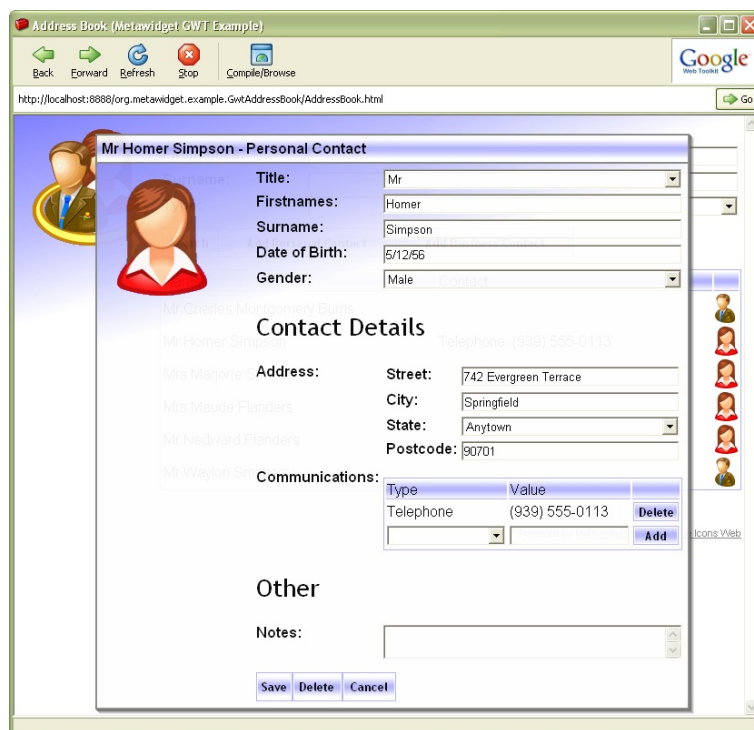


Figure 1.27. Address Book Example running in GWT Hosted Mode

There are two ways to set up the hosted mode projects. For Windows developers, there is an automated Ant build included in the source distribution (not the binary distribution). To run it, type:

```
cd examples\src\web\gwt
ant
```



Note

You will need to either configure `GWT_HOME`, `METAWIDGET_HOME` and `METAWIDGET_SRC_HOME` environment variables or pass `-Dgwt.home`, `-Dmetawidget.home` and `-Dmetawidget.src.home` arguments to Ant. `GWT_HOME` should point to a GWT 1.7 installation. `METAWIDGET_HOME` should point to the binary Metawidget distribution. `METAWIDGET_SRC_HOME` should point to the source Metawidget distribution.

The build will create Eclipse projects which can be imported by using Eclipse's *File->Import* menu and choosing *Existing Projects into Workspace* (as described in the [GWT Getting Started Guide](#)). You should now be able to run the sample applications in hosted mode, including running their unit tests.

Alternatively, you can set up the hosted mode projects manually. Here are the steps for the Address Book example (for other examples, replace the phrase `AddressBook` as appropriate):

- Run the GWT `webAppCreator` (as described in the [GWT Getting Started Guide](#)). Here we assume GWT is installed at `\gwt-windows-1.7.0` and you want to create a hosted mode project at `\gwt-addressbook`:

```
\gwt-windows-1.7.0\webAppCreator -out
    \gwt-addressbook org.metawidget.example.gwt.addressbook.client.AddressBook
```

- Delete the default generated Java source files:

```
del \gwt-addressbook\src\org\metawidget\example
```

- Copy over the Address Book example's Java source files. Here we assume the Metawidget source distribution is installed at `\metawidget-src` and you are using the Windows `xcopy` command:

```
xcopy /s /i \metawidget-src\examples\src\java\org\metawidget\example\gwt\addressbook
    \gwt-addressbook\src\org\metawidget\example\gwt\addressbook
xcopy /s /i \metawidget-src\examples\src\java\org\metawidget\example\shared\addressbook
    \gwt-addressbook\src\org\metawidget\example\shared\addressbook
copy \metawidget-src\examples\src\java\org\metawidget\example\GwtAddressBook.gwt.xml
    \gwt-addressbook\src\org\metawidget\example
```

- Delete the default generated Web source files:

```
del \gwt-addressbook\war
```

- Copy over the Address Book example's Web source files:

```
xcopy /s /i \metawidget-src\examples\src\web\gwt\addressbook \gwt-addressbook\war
xcopy /s /i \metawidget-src\examples\src\web\shared\addressbook \gwt-addressbook\war
copy \metawidget-src\examples\src\java\org\metawidget\example\GwtAddressBook.gwt.xml
    \gwt-addressbook\src\org\metawidget\example
```

- Copy over Metawidget itself. Here we assume the Metawidget binary distribution is installed at `\metawidget-bin`:

```
copy \metawidget-bin\metawidget.jar \gwt-addressbook\war\WEB-INF\lib
mkdir \gwt-addressbook\lib
copy \metawidget-bin\examples\gwt\metawidget-gwt.jar
    \gwt-addressbook\lib\metawidget-gwt.jar
```

- Import `\gwt-addressbook` into Eclipse by using *File->Import* and choosing *Existing Projects into Workspace* (as described in the [GWT Getting Started Guide](#)).
- Use Eclipse's *Project->Properties->Java Build Path* dialog and its *Libraries* tab to add the JARs `lib/metawidget-gwt.jar` and `WEB-INF/lib/metawidget.jar` into the project.
- Finally, edit the `AddressBook.launch` file to replace...

```
-startupUrl AddressBook.html org.metawidget.example.gwt.addressbook.client.AddressBook
```


...with...

```
-startupUrl index-hosted.jsp org.metawidget.example.GwtAddressBook
```

You should now be able to run the sample applications in hosted mode.

Once you have the samples running, you can quickly make changes and play around. For example, find the line in `ContactDialog` that configures a `LabelLayoutDecorator` (which decorates sections using `Labels`) and change it to a `TabPanelLayoutDecorator`. This will decorate sections using a GWT `TabPanel`, as shown in [Figure 1.28](#).

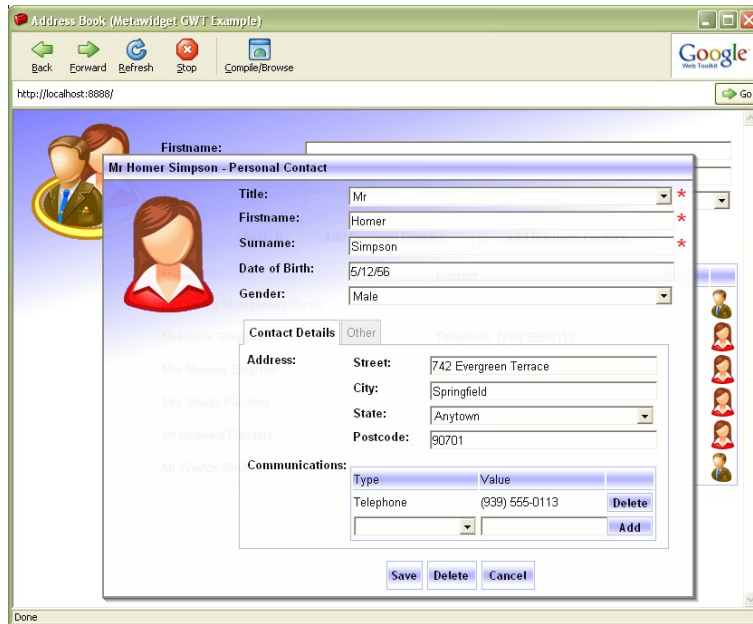


Figure 1.28. Address Book Example using GWT TabPanel

1.3.10 SWT Address Book Examples

There is an SWT version of the Address Book sample in `examples/swt/addressbook-swt.jar`. If you prefer SWT to Swing, you may prefer this version as in [Figure 1.29](#).

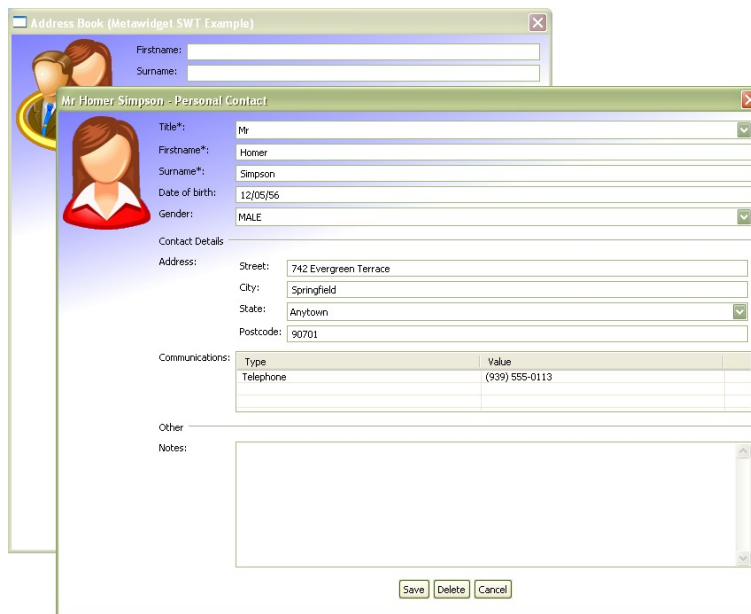


Figure 1.29. SWT Address Book Example

Key differences to the Swing version (see [Section 1.2.1, “Desktop Address Book”](#)) include:

- Data binding is implemented using `org.eclipse.core.databinding.Binding`.

2. Architecture

There are a large variety of system architectures, and many different types of UI. Metawidget's approach to managing this diversity is not to define lots of 'flags' to tweak lots of little variables, but to establish a pipeline with plug-in points along the way for your own custom classes. This five-stage pipeline is shown on the right of [Figure 2.1](#). It is co-ordinated by a platform-specific Metawidget class (eg. `SwingMetawidget`, `StrutsMetawidgetTag` etc) as shown on the left.

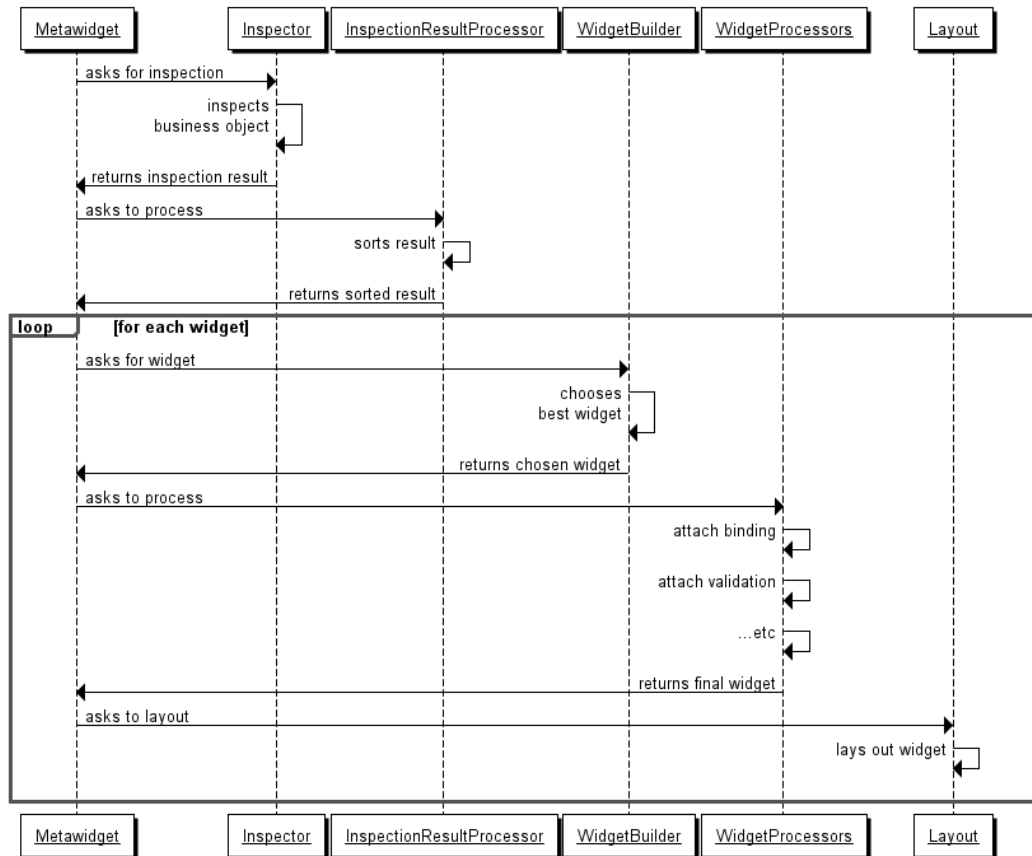


Figure 2.1. Metawidget uses a five-stage pipeline

The five stages of the pipeline are:

1. an `Inspector` that inspects specific back-end architectures. This can be a single `Inspector` or a list of multiple `Inspectors` (eg. `PropertyTypeInspector`, `JpaInspector` etc) by using `CompositeInspector` (see [Section 1.1.6](#), “[Combining Multiple Inspection Results](#)”). In the latter case, the multiple inspection results are all combined into a single result.
2. a list of `InspectionResultProcessors` that can modify the inspection result. These can be used to sort the result, exclude certain fields, and so on.
3. a `WidgetBuilder` that creates widgets for specific front-end frameworks. This can be a single `WidgetBuilder` or a list of multiple `WidgetBuilders` (eg. `HtmlWidgetBuilder`, `RichFacesWidgetBuilder` etc) by using `CompositeWidgetBuilder`.

In the latter case, the first `WidgetBuilder` to return a match is used. In this way `WidgetBuilders` for third party UI component libraries (that provide specialized components) can be listed first, and `WidgetBuilders` for the platform's standard components can be listed last (as a 'fallback' choice).

4. a list of `WidgetProcessors` that can modify each widget. These can be used to add data binding, event handlers, validation, tooltips and so on.
5. a `Layout` that arranges each widget on the screen, possibly organising them into columns and decorating them with labels (eg. `HtmlTableLayout`).



Immutable

All `Inspectors`, `WidgetBuilders`, `WidgetProcessors` and `Layouts` are required to be immutable. This means you only need a single instance of them for your entire application. If you are using `metawidget.xml` (see later) then `ConfigReader` takes care of this for you, but if you are instantiating them in Java code you should reuse instances. Keeping everything immutable allows `Metawidget` to be very performant at the same time as being very pluggable.

The following sections discuss each of the five stages of the pipeline in detail.

2.1 Metawidgets

`Metawidget` comes with a native component for each popular UI framework. This section discusses characteristics common to all `Metawidgets`. For in-depth documentation of a `Metawidget` for a specific UI framework, see [Chapter 3, *Metawidgets*](#).

2.1.1 Interface

`Metawidgets` are not required to extend any Java base class or implement any Java interface. This is because most UI frameworks require widgets inherit one of *their* base classes, such as `javax.swing.JComponent` or `javax.faces.UIComponent`, and Java does not support multiple inheritance.

In addition, while all `Metawidgets` support roughly the same functionality, different UI frameworks have different in-built capabilities. For example, JSF has `UIComponent.setRenderer` for choosing different layouts for the same widget, whereas `SwingMetawidget` has to roll its own `setMetawidgetLayout` method. This diversity of capabilities means there cannot be a common 'Metawidget Java interface' either.

However, despite not extending any common base class or interface, all `Metawidgets` follow roughly the same design, with roughly the same method names:

1. `setToInspect` is called to set the Object for inspection. The user typically calls this method, either directly or through some intermediate language (eg. using a JSP attribute).
2. Internally, `buildWidgets` is called to begin the process. It first calls `Inspector.inspect` to return an XML string of inspection results, then `InspectionResultProcessor.processInspectionResult` to sort them.
3. `buildWidgets` calls `WidgetBuilder.buildWidget` to choose a suitable widget for the top-level element of the XML (based on its `@type` attribute). If `WidgetBuilder.buildWidget` returns such a widget, skip to 6.
4. if `WidgetBuilder.buildWidget` returns null for the top-level element, call `buildCompoundWidget` to iterate over each child of the top-level element.

5. for each child, call `WidgetBuilder.buildWidget` and add the returned widget to the Metawidget. If `WidgetBuilder.buildWidget` returns null for a child, create a nested Metawidget.
6. the created widgets are passed through a series of `WidgetProcessors`. These can apply binding mechanisms, validators and so on.
7. as a final step, the created widgets are passed to a `Layout`. They can further be adorned with facet widgets.

For those looking to write their own Metawidget (say, for a currently unsupported platform) there is a `BasePipeline` class that implements the above steps 2-7 for you, see [the section called “BasePipeline”](#). All of the supplied Metawidgets are implemented using this class.

2.1.2 Customizing Look and Feel

As much as possible, Metawidgets defer to the existing Look and Feel technology of their native UI framework. For example, `HtmlMetawidget` uses HTML/CSS, `SwingMetawidget` uses Swing Look-and-Feels, and `AndroidMetawidget` uses Android styles and themes.

The one visual area Metawidget *does* control is how the widgets it creates are laid out. Typically this is in a tabular 'one column for the label, one column for the widget' format, but this is pluggable.

Metawidgets come with different `Layout` classes that can arrange the widgets in different ways, and these are set on the Metawidget in a framework-specific way. For example, JSF uses `<m:metawidget rendererType="">` whereas `SwingMetawidget` uses `setMetawidgetLayout`. Where possible, the layout classes defer back to the capabilities of the native framework. For example, Swing's `GridBagLayout` or Android's `TableLayout`.

Some layouts will add localized labels and section headings to the widgets, whereas other layouts may leave them unadorned. Different Layouts may support different parameters (for example, a `TableLayout` may support `numberOfColumns`). These are initialized on the Layout at construction time using a `xxxLayoutConfig`.

2.1.3 Overriding Widget Creation

Metawidget tries to automate much of the widget creation, but provides many hooks to customize the process:

- **stub** child widgets can be used to suppress widget creation entirely or to replace automatic widget creation with one or more other widgets with different bindings.
- **facet** child widgets pass through to the chosen layout as decorations (such as button bars).
- **WidgetBuilders** can be plugged in to the pipeline to control widget creation. They can be configured through `xxxWidgetBuilderConfig` classes.
- **WidgetProcessors** can be plugged in to the pipeline to fine-tune widget properties. They can be configured through `xxxWidgetProcessorConfig` classes.
- **Layouts** can be plugged in to the pipeline to control layout. They can be configured through `xxxLayoutConfig` classes.

2.1.4 Implementing Your Own Metawidget

Metawidget creates widgets native to a particular UI framework. Having to implement your own Metawidget should be far less common than having to implement your own `Inspector`, `WidgetBuilder` or `WidgetProcessor`, but if your chosen UI framework is not supported 'out of the box' you may need to implement your own.

Metawidgets are not required to extend any base class or implement any interface. However, it is recommended developers familiarize themselves with existing Metawidgets (such as `UIMetawidget`) to make their API similar. Whilst there is no one Metawidget base class, a number of convenience classes are provided:

BasePipeline

All the built-in Metawidgets use `BasePipeline` to ease their implementation. It provides pre-built functionality such as co-ordinating the 5 stages of the pipeline, deciding when to use single versus compound widgets, support for overriding widgets, incorporating stubs, and changing between read-only and active modes.

Android

Android already defines a separation between `Views` and `ViewGroups`.

`org.metawidget.android.widget.layout.Layout` and its subclasses automate the use of existing Android `ViewGroups`.

Java Server Faces

For frameworks based on JSF, `org.metawidget.faces.component.UIMetawidget` provides base widget functionality. See `org.metawidget.faces.component.html.HtmlMetawidget` for example usage.

JSF already defines a clean separation between widgets and their renderers.

`org.metawidget.faces.renderkit.LayoutRenderer` and its subclasses leverage this to support different layouts.

Java Server Pages

For frameworks based on JSP, `org.metawidget.jsp.tagext.MetawidgetTag` and the more commonly used `org.metawidget.jsp.tagext.html.BaseHtmlMetawidgetTag` provide base taglib functionality. See `StrutsMetawidgetTag` for example usage.

`MetawidgetTag` also defines a clean separation between choosing widgets and laying them out.

`org.metawidget.jsp.tagext.Layout` and its subclasses can perform the layout for *all* JSP-based frameworks.

Swing

Swing already defines a clean separation between widgets and layout managers.

`org.metawidget.swing.layout.Layout` and its subclasses automate the use of existing Swing `LayoutManagers`.

2.2 Inspectors

Inspectors decouple the process of examining back-end metadata and generating inspection results. This section covers inspectors in general. For in-depth documentation of individual inspectors see [Chapter 4, *Inspectors*](#).

2.2.1 Interface

All inspectors must implement the `Inspector` interface. This is a simple interface that defines only one method:

```
String inspect( Object toInspect, String type, String... names );
```

Each inspector must look to the *type* parameter and the *names* array. These form a path into the business object domain model. For example the *type* may be *com.myapp.Person* and the *names* may be *address* and *street*. This would form a path into the domain model of *com.myapp.Person/address/street* (ie. return information on the *street* property within the *address* property of the *Person* type).

Depending on the type of inspector, it may use the given *toInspect* to access the runtime object for the given *type*. Or it may ignore the *toInspect* and look up information for the given *type* from an XML file or a database schema. This allows Metawidget to inspect types that have no corresponding Java object. For example:

```
metawidget.setToInspect( null ); // No setToInspect
metawidget.setPath( "Login Form" );
```

This could be combined with, say, an *XmlInspector* and a *metawidget-metadata.xml*:

```
<entity type="Login Form">
  <property name="username"/>
  <property name="password"/>
</entity>
```

This approach also allows Metawidget to inspect abstract classes:

```
metawidget.setToInspect( null ); // No setToInspect
metawidget.setPath( MyAbstractClass.class.getName() );
```



Note

In general, a non-null *setToInspect* is a preferable, as many binding and validation technologies (see later) will be expecting a concrete object.

2.2.2 Usage

Unless explicitly specified, each Metawidget will instantiate a default inspector. Typically this will be a *CompositeInspector* composed of a *PropertyTypeInspector* and a *MetawidgetAnnotationInspector*.

This default behaviour can be overridden either in code:

```
metawidget.setInspector( new MyInspector() );
```

Or via *metawidget.xml*:

```
<swingMetawidget xmlns="java:org.metawidget.swing">
  <inspector>
    <myInspector xmlns="java:com.myapp"/>
  </inspector>
</swingMetawidget>
```

This allows easy plugging in of alternate inspectors. Note that overriding the default means the default is no longer instantiated. In the example above, this would mean *MyInspector* is used but the default inspectors are not. This is usually not what you want, because *MyInspector* will be focussed on a particular type of back-end metadata and will want to leave other metadata to other inspectors.

To achieve this, use *CompositeInspector*.

2.2.3 CompositeInspector

`CompositeInspector` composes the results of several inspectors into one and returns a single, combined inspection result. As shown in [Figure 2.2](#) `CompositeInspector` works by calling each inspector in turn, combining the inspection result as it goes.

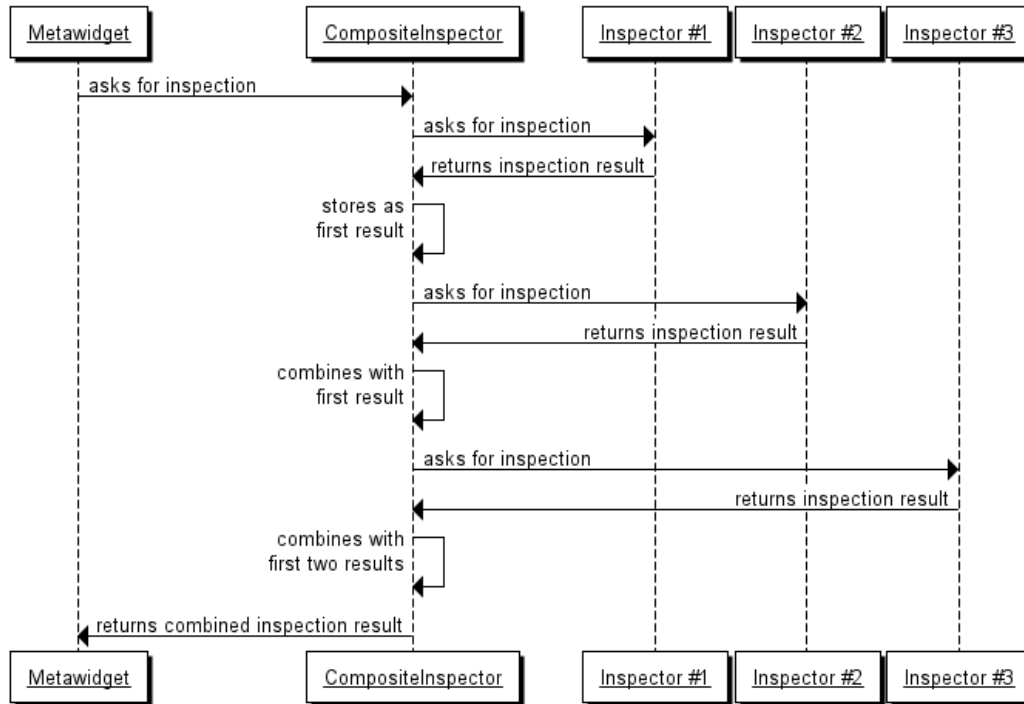


Figure 2.2. `CompositeInspector` composes multiple inspectors into one

All inspectors are required to be immutable (see later). Therefore, although `CompositeInspector` maintains a list of inspectors this must not be changeable. To enforce this, the list is set at instantiation time using `CompositeInspectorConfig`. This can either be set in code:

```
metawidget.setInspector( new CompositeInspector( new CompositeInspectorConfig()
    .setInspectors(
        new PropertyTypeInspector(),
        new MetawidgetAnnotationInspector(),
        new MyInspector()
    )
));
```

Or via `metawidget.xml`

```
<swingMetawidget xmlns="java:org.metawidget.swing">
<inspector>
<compositeInspector
  xmlns="java:org.metawidget.inspector.composite"
  config="CompositeInspectorConfig">
<inspectors>
  <array>
    <propertyTypeInspector xmlns="java:org.metawidget.inspector.propertytype"/>
    <metawidgetAnnotationInspector xmlns="java:org.metawidget.inspector.annotation" />
    <myInspector xmlns="java:com.myapp" />
  </array>
</inspectors>
</compositeInspector>
</inspector>
</swingMetawidget>
```

```

</compositeInspector>
</inspector>
</swingMetawidget>

```

2.2.4 Defaults

All Metawidgets have default Inspectors. Overriding the default means the default is no longer instantiated. Usually this is not what you want, so you should consider instantiating the default along with your new Inspector (ie. use CompositeInspector). You can see the default by looking in the Metawidget JAR for the file metawidget-xxx-default.xml (where xxx is your target platform, such as swing or struts).

For reference, the defaults are:

Platform	Default
Android	<pre> <compositeInspector xmlns="java:org.metawidget.inspector.composite" config="CompositeInspectorConfig"> <inspectors> <array> <propertyTypeInspector /> <metawidgetAnnotationInspector /> </array> </inspectors> </compositeInspector> </pre>
GWT	<pre> <compositeInspector xmlns="java:org.metawidget.inspector.composite" config="CompositeInspectorConfig"> <inspectors> <array> <propertyTypeInspector /> <metawidgetAnnotationInspector /> </array> </inspectors> </compositeInspector> </pre>
JSF	<pre> <compositeInspector xmlns="java:org.metawidget.inspector.composite" config="CompositeInspectorConfig"> <inspectors> <array> <propertyTypeInspector /> <metawidgetAnnotationInspector /> <facesInspector /> </array> </inspectors> </compositeInspector> </pre>
JSP	<pre> <compositeInspector xmlns="java:org.metawidget.inspector.composite" config="CompositeInspectorConfig"> <inspectors> <array> <propertyTypeInspector /> <metawidgetAnnotationInspector /> <jspAnnotationInspector /> </array> </inspectors> </compositeInspector> </pre>

Platform	Default
Spring	<pre> <compositeInspector xmlns="java:org.metawidget.inspector.composite" config="CompositeInspectorConfig"> <inspectors> <array> <propertyTypeInspector /> <metawidgetAnnotationInspector /> <springAnnotationInspector /> </array> </inspectors> </compositeInspector> </pre>
Struts	<pre> <compositeInspector xmlns="java:org.metawidget.inspector.composite" config="CompositeInspectorConfig"> <inspectors> <array> <propertyTypeInspector /> <metawidgetAnnotationInspector /> <strutsAnnotationInspector /> <commonsValidatorInspector config="CommonsValidatorInspectorConfig"/> </array> </inspectors> </compositeInspector> </pre>
Swing	<pre> <compositeInspector xmlns="java:org.metawidget.inspector.composite" config="CompositeInspectorConfig"> <inspectors> <array> <propertyTypeInspector /> <metawidgetAnnotationInspector /> </array> </inspectors> </compositeInspector> </pre>

2.2.5 Immutability

All inspectors are required to be immutable. This means you only need a single instance of an `Inspector` for your entire application. If you are using `metawidget.xml` (see later) then `ConfigReader` takes care of this for you, but if you are instantiating `Inspectors` in Java code you should reuse instances.

Note that immutable only means `Inspectors` cannot be changed once instantiated - it does not mean they cannot be configured. Many `Inspectors` have corresponding `xxxConfig` classes that allow them to be configured prior to instantiation in a type-safe way. For example, a `JpaInspector` can be configured in code:

```
metawidget.setInspector( new JpaInspector( new JpaInspectorConfig().setHideIds( false ) ));
```

Or in `metawidget.xml`:

```

<jpaInspector xmlns="java:org.metawidget.inspector.jpa" config="JpaInspectorConfig">
  <hideIds>
    <boolean>true</boolean>
  </hideIds>
</jpaInspector>

```

2.2.6 inspection-result

The *inspection-result* XML format is the 'glue' that holds everything together: the Metawidgets request it, the Inspectors provide it, and the WidgetBuilders base their choice of widgets on it.

It is a very simple format. As an example:

```
<inspection-result version="1.0">
  <entity type="com.myapp.Person">
    <property name="name" required="true"/>
    <property name="age" minimum-value="0"/>
  </entity>
</inspection-result>
```

Only a handful of XML attributes are mandatory (see *inspection-result-1.0.xsd*). Most, such as *retired* and *minimum-value*, are provided at the discretion of the Inspector and recognised at the discretion of the WidgetBuilders, WidgetProcessors and Layouts. This loose coupling allows Inspectors to evolve independently for new types of metadata, WidgetBuilders to evolve independently with new types of widgets, and so on.

2.2.7 Implementing Your Own Inspector

Metawidget inspects a wide variety of back-end architectures. If your chosen back-end architecture is not supported 'out of the box', you may need to implement your own Inspector.

All Inspectors must implement the `org.metawidget.inspector.Inspector` interface:

```
public interface Inspector {
    String inspect( Object toInspect, String type, String... names );
}
```

The interface has only one method: `inspect`. Its parameters are:

- an `Object` to inspect. This may be `null`, or can be ignored for inspectors inspecting static metadata (such as XML files)
- a *type*. This must match the given `Object`, or some attribute in the inspected config file
- a list of *names* to be traversed beneath the type

The returned `String` must be an XML document conforming to *inspection-result-1.0.xsd*. To assist development, deploy your inspector within `ValidatingCompositeInspector` to automatically validate the returned DOM during testing.

A number of convenience base classes are provided for different inspectors:

- `BaseObjectInspector` assists in inspecting annotations and properties (including support for different property styles, such as `JavaBean` properties or `Groovy` properties). Here is an example of a custom `Inspector` to inspect tooltip metadata from a custom annotation. It extends the code from the tutorial (see [Section 1.1, "Part 1 - The First Metawidget Application"](#)).

```
package com.myapp;

import java.lang.annotation.*;
import java.util.*;
import javax.swing.JFrame;
import org.metawidget.inspector.composite.*;
import org.metawidget.inspector.impl.*;
import org.metawidget.inspector.impl.propertystyle.*;
```

```

import org.metawidget.inspector.propertytype.*;
import org.metawidget.swing.*;
import org.metawidget.util.*;

public class Main {

    public static void main( String[] args ) {
        Person person = new Person();

        SwingMetawidget metawidget = new SwingMetawidget();
        CompositeInspectorConfig inspectorConfig = new CompositeInspectorConfig().setInspectors(
            new PropertyTypeInspector(),
            new TooltipInspector() );
        metawidget.setInspector( new CompositeInspector( inspectorConfig ) );
        metawidget.setToInspect( person );

        JFrame frame = new JFrame( "Metawidget Tutorial" );
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.getContentPane().add( metawidget );
        frame.setSize( 400, 250 );
        frame.setVisible( true );
    }

    @Retention( RetentionPolicy.RUNTIME )
    @Target( { ElementType.FIELD, ElementType.METHOD } )
    static @interface Tooltip {
        String value();
    }

    static class TooltipInspector
        extends BaseObjectInspector {

        protected Map<String, String> inspectProperty( Property property )
            throws Exception {

            Map<String, String> attributes = CollectionUtils.newHashMap();

            Tooltip tooltip = property.getAnnotation( Tooltip.class );

            if ( tooltip != null )
                attributes.put( "tooltip", tooltip.value() );

            return attributes;
        }
    }
}

```

You could then annotate the Person class...

```

package com.myapp;

import com.myapp.Main.Tooltip;

public class Person {
    @Tooltip("Person's full name")
    public String name;
    @Tooltip("Age in years")
    public int age;
    @Tooltip("Whether person is retired")
    public boolean retired;
}

```

```
}
```

...and `TooltipInspector` would pick up the custom `@Tooltip` annotation and feed it into the Metawidget pipeline.



Note

Because Metawidget decouples inspection from widget creation, by default `SwingMetawidget` will not be expecting this new `tooltip` attribute and will ignore it. You will need to further combine this example with a `TooltipProcessor`, see [Section 2.5.5, “Implementing Your Own WidgetProcessor”](#).

- For inspecting XML files, `BaseXmlInspector` assists in opening and traversing through the XML, as well as merging multiple XML files into one (eg. merging multiple Hibernate mapping files). Here is an example of a custom `Inspector` to inspect XML:

TODO: for now, see `StrutsInspector` for example usage.

When implementing your own inspector, try to avoid technology-specific XML attribute names. For example, `FacesInspector` has an annotation `@UiFacesNumberConverter`. This annotation certainly has technology-specific parts to it, as it names a JSF Converter that only applies in JSF environments, so it is reasonable to name the XML attribute `faces-converter-class`. However, `NumberConverters` also use other properties about the field, such as the maximum number of integer digits. Such properties are not JSF-specific (eg. we can source the same property from Hibernate Validator's `@Digits` annotation), so are better named 'neutrally' (eg. `maximum-integer-digits`).



Config classes must override equals and hashCode

If your `Inspector` has a `xxxInspectorConfig` class, and you want it to be cacheable and reusable by `ConfigReader` and `metawidget.xml`, the `xxxInspectorConfig` class *must* override `equals` and `hashCode`.



Generate an XML Schema

If your `Inspector` has an `xxxInspectorConfig` class, consider defining an XML Schema for it. This is optional, but allows users to validate their use of your `Inspector` in their `metawidget.xml` at development time. There is an Ant task, `org.metawidget.config.XmlSchemaGeneratorTask`, provided in the source distribution that can help with this by auto-generating the schema. All the existing Metawidget schemas are generated using this Ant task.

2.3 Inspection Result Processors

`InspectionResultProcessors` allow arbitrary processing of the inspection result returned by the `Inspector`, before it is passed to the `WidgetBuilder`. This section covers `InspectionResultProcessors` in general. For in-depth documentation of individual `InspectionResultProcessors` see [Chapter 5, *InspectionResultProcessors*](#).

2.3.1 Interface

All `InspectionResultProcessors` must implement the `InspectionResultProcessor` interface. This is a simple interface that defines only one method:

```
E processInspectionResult( E inspectionResult, M metawidget );
```

Where *E* is a DOM Element (typically `org.w3c.dom.Element`) containing the inspection result, and *M* is a Metawidget class (ie. `SwingMetawidget`, `UIMetawidget` etc).

The `InspectionResultProcessor` must return the processed inspection result's DOM Element. This is typically the same as the given `inspectionResult`. The parent `Metawidget` then passes this to the next `InspectionResultProcessor` in the list as show in [Figure 2.3](#).

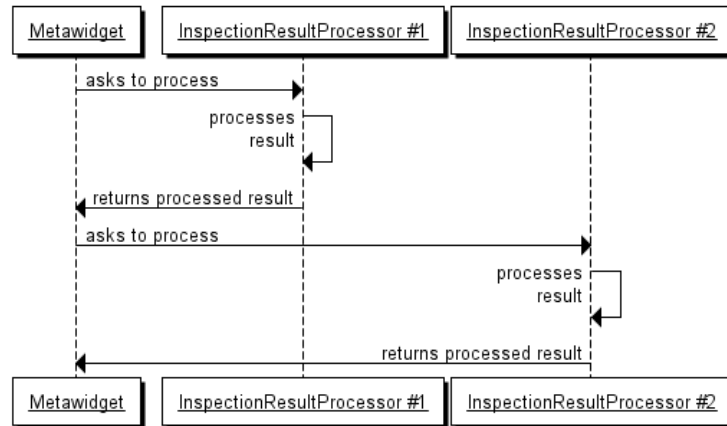


Figure 2.3. Typical `InspectionResultProcessor` list

In most cases the `InspectionResultProcessor` will simply be modifying the given `inspectionResult`. However it can decide to swap it out by returning a different DOM. This new DOM will then be passed down the list. Alternatively, the `InspectionResultProcessor` can return `null` to cancel inspection entirely. No further `InspectionResultProcessors` will be called, as shown in [Figure 2.4](#).

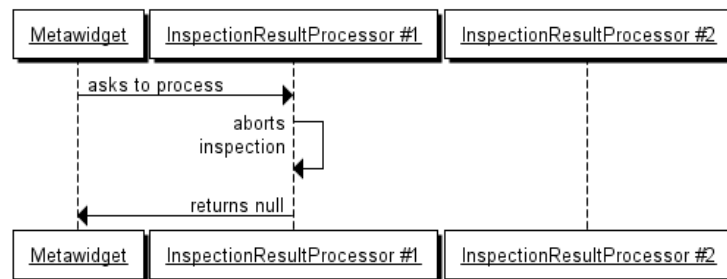


Figure 2.4. An `InspectionResultProcessor` can abort the inspection

2.3.2 Defaults

Most `Metawidgets` have default `InspectionResultProcessors`. You can see the default by looking in the `Metawidget` JAR for the file `metawidget-xxx-default.xml` (where `xxx` is your target platform, such as `swing` or `struts`).

For reference, the defaults are:

Platform	Default
Android	<pre> <array> <comesAfterInspectionResultProcessor /> </array> </pre>
GWT	<pre> <array> <comesAfterInspectionResultProcessor /> </array> </pre>
JSF	<pre> <array> </pre>

Platform	Default
	<pre><comesAfterInspectionResultProcessor /> </array></pre>
JSP	<pre><array> <comesAfterInspectionResultProcessor /> </array></pre>
Spring	<pre><array> <comesAfterInspectionResultProcessor /> </array></pre>
Struts	<pre><array> <comesAfterInspectionResultProcessor /> </array></pre>
Swing	<pre><array> <comesAfterInspectionResultProcessor /> </array></pre>

2.3.3 Immutability

All `InspectionResultProcessors` are required to be immutable. This means you only need a single instance of an `InspectionResultProcessor` for your entire application. If you are using `metawidget.xml` then `ConfigReader` takes care of this for you, but if you are instantiating `InspectionResultProcessors` in Java code you should reuse instances.

2.3.4 Implementing Your Own `InspectionResultProcessor`

Here is an example of a custom `InspectionResultProcessor` that chooses, and sorts, business object fields based on a `JComponent` client property. It extends the code from the tutorial (see [Section 1.1, “Part 1 - The First Metawidget Application”](#)).

```
package com.myapp;

import static org.metawidget.inspector.InspectionResultConstants.*;

import javax.swing.*;
import org.metawidget.swing.*;
import org.metawidget.inspectionresultprocessor.iface.*;
import org.metawidget.util.*;
import org.w3c.dom.*;

public class Main {

    public static void main( String[] args ) {
        Person person = new Person();

        SwingMetawidget metawidget = new SwingMetawidget();
        metawidget.addInspectionResultProcessor( new IncludingInspectionResultProcessor() );
        metawidget.putClientProperty( "include", new String[]{ "age", "retired" } );
        metawidget.setToInspect( person );

        JFrame frame = new JFrame( "Metawidget Tutorial" );
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
```

```

frame.getContentPane().add( metawidget );
frame.setSize( 400, 250 );
frame.setVisible( true );
}

static class IncludingInspectionResultProcessor
    implements InspectionResultProcessor<Element, SwingMetawidget> {

    public Element processInspectionResult( Element inspectionResult, SwingMetawidget metawidget ) {

        String[] includes = (String[]) metawidget.getClientProperty( "include" );
        Element entity = (Element) inspectionResult.getFirstChild();
        int propertiesToCleanup = entity.getChildNodes().getLength();

        // Pull out the names in order

        for( String include : includes ) {

            Element property = XmlUtils.getChildWithAttributeValue( entity, NAME, include );

            if ( property == null )
                continue;

            entity.appendChild( property );
            propertiesToCleanup--;
        }

        // Remove the rest

        for( int loop = 0; loop < propertiesToCleanup; loop++ ) {
            entity.removeChild( entity.getFirstChild() );
        }

        return inspectionResult;
    }
}

```

**Note**

We don't necessarily recommend this approach, as it requires hard-coding business property names and won't refactor well.

2.4 Widget Builders

WidgetBuilders decouple the process of choosing widgets based on inspection results. This section covers WidgetBuilders in general. For in-depth documentation of individual WidgetBuilders see [Chapter 6, *Widget Builders*](#).

2.4.1 Interface

All WidgetBuilders must implement the `WidgetBuilder` interface. This is a simple interface that defines only one method:

```
W buildWidget( String elementName, Map<String, String> attributes, M metawidget )
```

Where *W* is a widget type (such as `JComponent` or `UIComponent`) and *M* is a Metawidget type (such as `SwingMetawidget` or `UIMetawidget`).

Each `WidgetBuilder` must look to the `elementName`, which is typically just 'property' or 'action' from the `inspection-result`, and to the various `attributes` and instantiate an appropriate widget. `WidgetBuilders` can use the given `metawidget` to help them if needed (for example, to access a UI context with which to instantiate widgets). Typically the `WidgetBuilders` do not need to configure the widget beyond simply instantiating it: the job of setting `ids`, attaching validators, configuring bindings and so forth is done by the `WidgetProcessors` (see [???](#)).

2.4.2 Usage

Unless explicitly specified, each `Metawidget` will instantiate default `WidgetBuilders` to match the target platform. For example, `SwingMetawidget` will by default instantiate an `OverriddenWidgetBuilder`, `ReadOnlyWidgetBuilder` and `SwingWidgetBuilder`.

This default behaviour can be overridden either in code:

```
metawidget.setWidgetBuilder( new MyWidgetBuilder() );
```

Or via `metawidget.xml`

```
<swingMetawidget xmlns="java:org.metawidget.swing">
  <widgetBuilder>
    <myWidgetBuilder xmlns="java:com.myapp" />
  </widgetBuilder>
</swingMetawidget>
```

This allows easy plugging in of third-party widget libraries. Note that overriding the default means the default is no longer instantiated. In the example above, this would mean `MyWidgetBuilder` is used but `OverriddenWidgetBuilder`, `ReadOnlyWidgetBuilder` and `SwingWidgetBuilder` are not. This is usually not what you want, because `MyWidgetBuilder` will be focussed on a particular third party library and will want to leave widget overriding to `OverriddenWidgetBuilder`, read-only widgets (ie. labels) to `ReadOnlyWidgetBuilder` and standard widgets to `SwingWidgetBuilder`.

To achieve this, use `CompositeWidgetBuilder`.

2.4.3 CompositeWidgetBuilder

`CompositeWidgetBuilder` combines the widget libraries of several `WidgetBuilders`. It defers widget building to an internal list of `WidgetBuilders`, in order, and goes with the first one that returns non-null (see [figure Figure 2.5](#)).

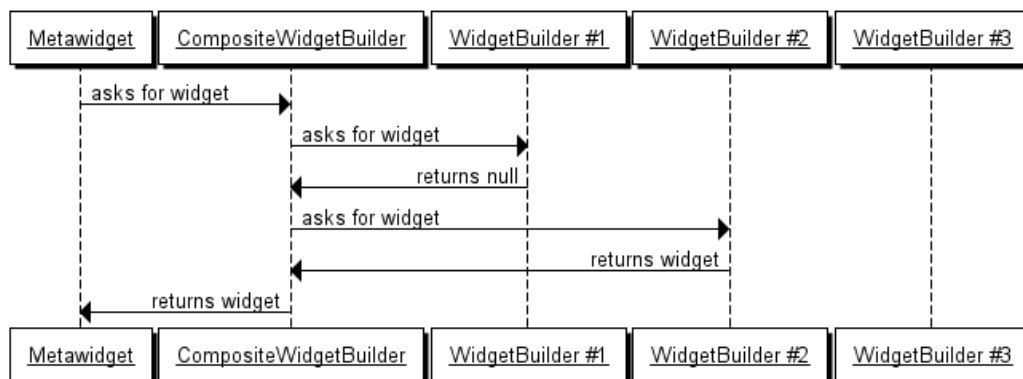


Figure 2.5. `CompositeWidgetBuilder` composes multiple `WidgetBuilders` into one

In this way `WidgetBuilders` for third party UI component libraries (that provide specialized components) can be listed first, and `WidgetBuilders` for the platform's standard components can be listed last (as a 'fallback' choice).

`CompositeWidgetBuilder` can be instantiated either in code:

```
metawidget.setWidgetBuilder( new CompositeWidgetBuilder( new CompositeWidgetBuilderConfig()
    .setWidgetBuilders(
        new MyWidgetBuilder(), new SwingWidgetBuilder()
    )
));
```

Or via `metawidget.xml`

```
<swingMetawidget xmlns="java:org.metawidget.swing">
  <widgetBuilder>
    <compositeWidgetBuilder
      xmlns="java:org.metawidget.widgetbuilder.composite"
      config="CompositeWidgetBuilderConfig">
      <widgetBuilders>
        <array>
          <overriddenWidgetBuilder xmlns="java:org.metawidget.swing.widgetbuilder"/>
          <readOnlyWidgetBuilder xmlns="java:org.metawidget.swing.widgetbuilder"/>
          <myWidgetBuilder xmlns="java:com.myapp"/>
          <swingWidgetBuilder xmlns="java:org.metawidget.swing.widgetbuilder"/>
        </array>
      </widgetBuilders>
    </compositeWidgetBuilder>
  </widgetBuilder>
</swingMetawidget>
```

2.4.4 OverriddenWidgetBuilder

The first `WidgetBuilder` in the `CompositeWidgetBuilder` chain should generally be an `OverriddenWidgetBuilder`. This looks for existing child widgets that override default generation. What constitutes an 'overridden widget' varies from platform to platform. For example, for Swing any child widget with the same *name* will be taken as the override (see [Section 1.1.8, “Controlling Widget Creation”](#)). Android uses the *tag* attribute, JSF uses the value binding, and so on. For details on the `OverriddenWidgetBuilder` for your platform see [Chapter 6, *Widget Builders*](#).

You can also choose to plug-in your own `WidgetBuilder` that detects 'overridden widgets' based on your own criteria. Here is an example of a custom `WidgetBuilder` that excludes widgets based on a client property. It extends the code from the tutorial (see [Section 1.1, “Part 1 - The First Metawidget Application”](#)).

```
package com.myapp;

import static org.metawidget.inspector.InspectionResultConstants.*;

import java.util.*;

import javax.swing.*;
import org.metawidget.swing.*;
import org.metawidget.swing.widgetbuilder.*;
import org.metawidget.widgetbuilder.composite.*;
import org.metawidget.widgetbuilder.iface.*;
import org.metawidget.util.*;

public class Main {

    public static void main( String[] args ) {
```

```

Person person = new Person();

SwingMetawidget metawidget = new SwingMetawidget();
metawidget.setWidgetBuilder( new CompositeWidgetBuilder<JComponent, SwingMetawidget>(
    new CompositeWidgetBuilderConfig<JComponent, SwingMetawidget>().setWidgetBuilders(
        new ExcludingWidgetBuilder(),
        new SwingWidgetBuilder() ) ) );
metawidget.putClientProperty( "exclude", new String[]{ "age", "retired" } );
metawidget.setToInspect( person );

JFrame frame = new JFrame( "Metawidget Tutorial" );
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.getContentPane().add( metawidget );
frame.setSize( 400, 250 );
frame.setVisible( true );
}

static class ExcludingWidgetBuilder
    implements WidgetBuilder<JComponent, SwingMetawidget> {
    public JComponent buildWidget( String elementName, Map<String, String> attributes,
        SwingMetawidget metawidget ) {
        String[] exclude = (String[]) metawidget.getClientProperty( "exclude" );

        if ( ArrayUtils.contains( exclude, attributes.get( NAME ) ))
            return new Stub();

        return null;
    }
}

```

**Note**

We don't necessarily recommend this approach, as it requires hard-coding business property names and won't refactor well.

**Note**

Although you could adapt this approach to only include (instead of exclude) certain fields, you could not adapt it to include fields *in the order given in the client property*. This is because `WidgetBuilders` only operate on single widgets at a time. Instead, see [Section 2.3, “Inspection Result Processors”](#)

2.4.5 ReadOnlyWidgetBuilder

The second `WidgetBuilder` in the `CompositeWidgetBuilder` chain should generally be a `ReadOnlyWidgetBuilder`. This builds standard platform widgets (ie. labels) for fields with `read-only="true"`.

The exception to this rule would be if you wanted to add a custom `WidgetBuilder` for a widget library that had its own read-only components, or if you wanted to customise the read-only handling. Here is an example of a custom `WidgetBuilder` that returns non-editable `JTextFields` (instead of `JLabels`) for read-only fields. It extends the code from the tutorial (see [Section 1.1, “Part 1 - The First Metawidget Application”](#)).

```

package com.myapp;

import static org.metawidget.inspector.InspectionResultConstants.*;

import java.util.*;

```

```

import javax.swing.*;
import org.metawidget.swing.*;
import org.metawidget.swing.widgetbuilder.*;
import org.metawidget.widgetbuilder.composite.*;
import org.metawidget.widgetbuilder.iface.*;
import org.metawidget.util.*;

public class Main {

    public static void main( String[] args ) {
        Person person = new Person();

        SwingMetawidget metawidget = new SwingMetawidget();
        metawidget.setWidgetBuilder( new CompositeWidgetBuilder<JComponent, SwingMetawidget>(
            new CompositeWidgetBuilderConfig<JComponent, SwingMetawidget>().setWidgetBuilders(
                new ReadOnlyTextFieldWidgetBuilder(),
                new SwingWidgetBuilder() ) ) );
        metawidget.setToInspect( person );

        JFrame frame = new JFrame( "Metawidget Tutorial" );
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.getContentPane().add( metawidget );
        frame.setSize( 400, 250 );
        frame.setVisible( true );
    }

    static class ReadOnlyTextFieldWidgetBuilder
        implements WidgetBuilder<JComponent, SwingMetawidget> {
        public JComponent buildWidget( String elementName, Map<String, String> attributes,
            SwingMetawidget metawidget ) {
            if ( !WidgetBuilderUtils.isReadOnly( attributes ) )
                return null;

            if ( TRUE.equals( attributes.get( HIDDEN ) ) )
                return null;

            Class<?> clazz = ClassUtils.niceForName( attributes.get( TYPE ) );

            if ( String.class.equals( clazz ) || clazz.isPrimitive() ) {
                JTextField textField = new JTextField();
                textField.setEditable( false );

                return textField;
            }

            return null;
        }
    }
}

```

2.4.6 Defaults

All Metawidgets have default `WidgetBuilders`. Overriding the default means the default is no longer instantiated. Usually this is not what you want, so you should consider instantiating the default along with your new `WidgetBuilder` (ie. use `CompositeWidgetBuilder`). You can see the default by looking in the Metawidget JAR for the file `metawidget-xxx-default.xml` (where `xxx` is your target platform, such as `swing` or `struts`).

For reference, the defaults are:

Platform	Default
Android	...TODO...
GWT	...TODO...
JSF	<pre> <compositeWidgetBuilder config="CompositeWidgetBuilderConfig"> <widgetBuilders> <array> <overriddenWidgetBuilder /> <readOnlyWidgetBuilder /> <htmlWidgetBuilder /> </array> </widgetBuilders> </compositeWidgetBuilder> </pre>
JSP	<pre> <compositeWidgetBuilder config="CompositeWidgetBuilderConfig"> <widgetBuilders> <array> <overriddenWidgetBuilder /> <readOnlyWidgetBuilder /> <htmlWidgetBuilder /> </array> </widgetBuilders> </compositeWidgetBuilder> </pre>
Spring	<pre> <compositeWidgetBuilder config="CompositeWidgetBuilderConfig"> <widgetBuilders> <array> <overriddenWidgetBuilder /> <springWidgetBuilder /> <readOnlyWidgetBuilder /> <htmlWidgetBuilder /> </array> </widgetBuilders> </compositeWidgetBuilder> </pre>
Struts	<pre> <compositeWidgetBuilder config="CompositeWidgetBuilderConfig"> <widgetBuilders> <array> <overriddenWidgetBuilder /> <strutsWidgetBuilder /> <readOnlyWidgetBuilder /> <htmlWidgetBuilder /> </array> </widgetBuilders> </compositeWidgetBuilder> </pre>
Swing	<pre> <compositeWidgetBuilder config="CompositeWidgetBuilderConfig"> <widgetBuilders> <array> <overriddenWidgetBuilder /> <readOnlyWidgetBuilder /> <swingWidgetBuilder /> </array> </widgetBuilders> </compositeWidgetBuilder> </pre>

2.4.7 Immutability

All WidgetBuilders are required to be immutable. This means you only need a single instance of a WidgetBuilder for your entire application. If you are using `metawidget.xml` then `ConfigReader` takes care of this for you, but if you are instantiating WidgetBuilders in Java code you should reuse instances.

Note that immutable only means WidgetBuilders cannot be changed once instantiated - it does not mean they cannot be configured. Many WidgetBuilders have corresponding `xxxConfig` classes that allow them to be configured prior to instantiation in a type-safe way. For example, a `HtmlWidgetBuilder` can be configured in code:

```
metawidget.setWidgetBuilder( new HtmlWidgetBuilder( new HtmlWidgetBuilderConfig()
    .setDataTableStyleClass( "data-table" ) ));
```

Or in `metawidget.xml`:

```
<htmlWidgetBuilder xmlns="java:org.metawidget.faces.component.html.widgetbuilder"
  config="HtmlWidgetBuilderConfig">
  <dataTableStyleClass>
    <boolean>data-table</boolean>
  </dataTableStyleClass>
</htmlWidgetBuilder>
```

2.4.8 Implementing Your Own WidgetBuilder

The pluggable nature of WidgetBuilders makes it easy to add your own. Because `CompositeWidgetBuilder` can be used to chain WidgetBuilders together, you only need worry about supporting your particular component library's widgets. You can simply return `null` for all other types of fields and rely on another WidgetBuilder further down the chain to instantiate one of the usual widgets.

For example, `RichFacesWidgetBuilder` only instantiates the RichFaces components, and returns `null` for everything else.

Here is an example of a custom WidgetBuilder that uses two `JRadioButtons`, instead of the usual `JCheckBox`, to represent boolean properties. It extends the code from the tutorial (see [Section 1.1, “Part 1 - The First Metawidget Application”](#)).

```
package com.myapp;

import static org.metawidget.inspector.InspectionResultConstants.*;

import java.awt.*;
import java.util.*;
import javax.swing.*;
import org.metawidget.swing.*;
import org.metawidget.swing.widgetbuilder.*;
import org.metawidget.widgetbuilder.composite.*;
import org.metawidget.widgetbuilder.impl.*;

public class Main {

    public static void main( String[] args ) {
        Person person = new Person();

        SwingMetawidget metawidget = new SwingMetawidget();
        metawidget.setWidgetBuilder( new CompositeWidgetBuilder<JComponent, SwingMetawidget>(
            new CompositeWidgetBuilderConfig<JComponent, SwingMetawidget>().setWidgetBuilders(
```

```

    new JRadioButtonWidgetBuilder(),
    new SwingWidgetBuilder() ) );
metawidget.setToInspect( person );

JFrame frame = new JFrame( "Metawidget Tutorial" );
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.getContentPane().add( metawidget );
frame.setSize( 400, 250 );
frame.setVisible( true );
}

static class JRadioButtonWidgetBuilder
implements WidgetBuilder<JComponent, SwingMetawidget> {

    public JComponent buildWidget( String elementName, Map<String, String> attributes,
    SwingMetawidget metawidget ) {

        if ( !"boolean".equals( attributes.get( TYPE ) ) )
            return null;

        JRadioButton trueButton = new JRadioButton( "True" );
        JRadioButton falseButton = new JRadioButton( "False" );
        JPanel panel = new JPanel();
        panel.setLayout( new GridLayout( 2, 1 ) );
        panel.add( trueButton );
        panel.add( falseButton );

        ButtonGroup buttonGroup = new ButtonGroup();
        buttonGroup.add( trueButton );
        buttonGroup.add( falseButton );

        return panel;
    }
}

```



Generate an XML Schema

If your WidgetBuilder has an xxxWidgetBuilderConfig class, consider defining an XML Schema for it. This is optional, but allows users to validate their use of your WidgetBuilder in their metawidget.xml at development time. There is an Ant task, `org.metawidget.config.XmlSchemaGeneratorTask`, provided in the source distribution that can help with this by auto-generating the schema. All the existing Metawidget schemas are generated using this Ant task.

Special considerations for Java Server Faces

When developing WidgetBuilders for JSF component libraries, be aware that Metawidget integrates with the JSF lifecycle in a slightly unorthodox way. Upon POSTback, Metawidget first decodes, processes validators and updates the business model as usual. Upon `encodeBegin`, however, Metawidget *destroys and recreates* all previously generated UIComponents. This is so the UIComponents can adapt to updates to the business model. For example, they might need to be changed from a `UIOutputText` to a `UIInputText` if the user clicks an *Edit* button.

In most cases such recreation works well, but on occasion a component may not be expecting to be recreated, and may not function properly. For example, the ICEfaces `SelectInputDate` component stores the state of its date popup internally. If it is recreated, this state is lost and the popup never appears. For such components, WidgetBuilder authors can set the attribute `UIMetawidget.COMPONENT_ATTRIBUTE_NOT_RECREATABLE` on the component to prevent its destruction and

recreation. Of course, this somewhat impacts its flexibility. For example, a `SelectInputDate` would not be able to change its date format in response to another component on the form.

2.5 Widget Processors

`WidgetProcessors` allow arbitrary processing of a widget following its building by a `WidgetBuilder` and before its inclusion in the `Layout`. This section covers `WidgetProcessors` in general. For in-depth documentation of individual `WidgetProcessors` see [Chapter 7, *WidgetProcessors*](#).

2.5.1 Interface

All `WidgetProcessors` must implement the `WidgetProcessor` interface. This is a simple interface that defines only one method:

```
W processWidget( W widget, String elementName, Map<String, String> attributes, M metawidget );
```

Where *W* is a widget class (ie. `JComponent`, `UIComponent` etc) and *M* is a `Metawidget` class (ie. `SwingMetawidget`, `UIMetawidget` etc).

`processWidget` is called for each widget built by the `WidgetBuilders`. `WidgetProcessors` can modify the given *widget* according to the given *elementName* and various *attributes*. They can use the given *metawidget* to help them if needed (for example, to access a UI context with which to create validators).

The `processWidget` method must return the processed widget. This is typically the same as the given *widget*. The parent `Metawidget` then passes this to the next `WidgetProcessor` in the list as show in [Figure 2.6](#).

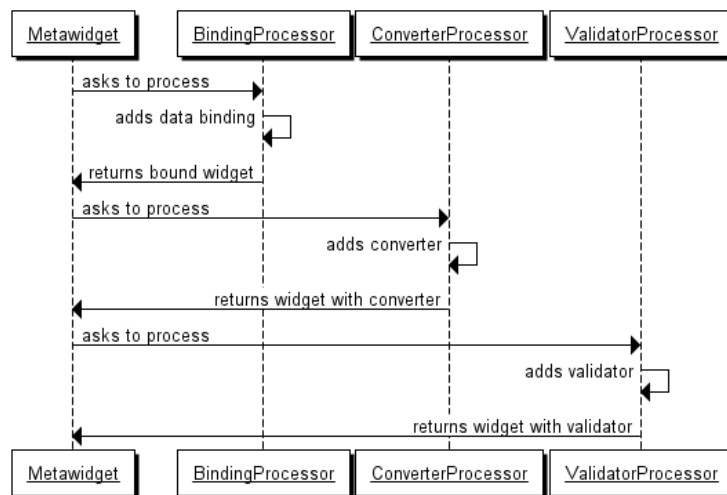


Figure 2.6. Typical `WidgetProcessor` list

In most cases the `WidgetProcessor` will simply be modifying the given *widget* (adding validators, changing styles and so on). However it can decide to swap the widget out by returning a different widget. This new widget will then be passed down the list as shown in [Figure 2.7](#). For an example use of this capability, see `HiddenFieldProcessor`.

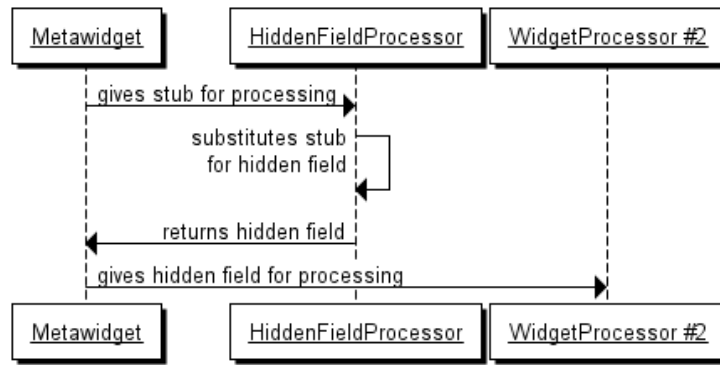


Figure 2.7. WidgetProcessors can substitute widgets

Alternatively, the WidgetProcessor can decide to exclude the widget entirely by returning null. Subsequent WidgetProcessors will not be called, as shown in Figure 2.8, and no widget will be added to the Layout.

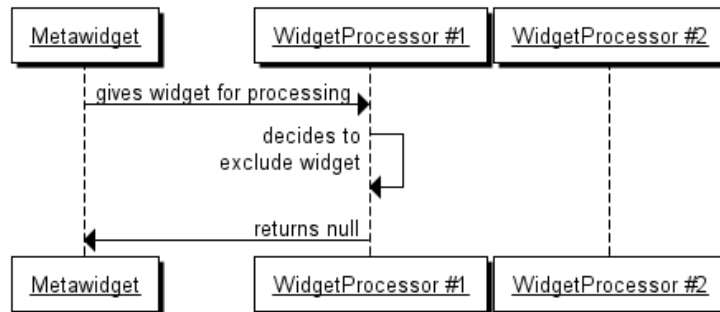


Figure 2.8. WidgetProcessors can exclude widgets

The list of WidgetProcessors is maintained by the parent Metawidget, and is changeable (this is different to say, CompositeInspector or CompositeWidgetBuilder whose lists are immutable). This capability is designed to allow easy attaching of event handlers, and scenarios such as inner classes that have connections to their parent class:

```

final Object someObject = ...;

metawidget.addWidgetProcessor( new WidgetProcessor<JComponent, SwingMetawidget>() {
    JComponent processWidget( JComponent widget, String elementName, Map<String, String> attributes,
        SwingMetawidget metawidget ) {
        ...decide whether to attach event handler...

        widget.add( new AbstractAction() {
            public void actionPerformed( ActionEvent e ) {
                someObject.doSomething();
            }
        } )
    }
} )

```

2.5.2 Advanced Interface

The WidgetProcessor interface only has a single method. This allows it to take advantage of Java 7 language features such as:

```

metawidget.addWidgetProcessor(
    #(JComponent w, String name, Map<String, String> attr, SwingMetawidget m)

```



```
{ w( #(ActionEvent e) { someObject.doSomething } ) } );
```

However for those needing more control over the WidgetProcessor lifecycle there is an extended interface AdvancedWidgetProcessor. This interface defines two additional methods:

```
void onStartBuild( M metawidget );

void onEndBuild( M metawidget );
```

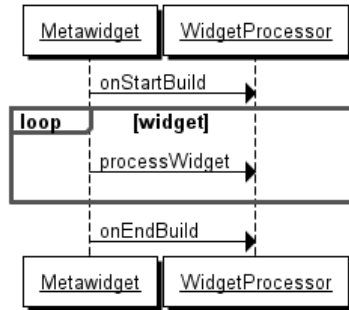


Figure 2.9. *onStartBuild* and *onEndBuild* are called once, *processWidget* is called for each widget

The first method, *onStartBuild*, is called at the start of the widget building process, before the *WidgetBuilder* is called. *WidgetProcessors* may wish to act on this event to initialize themselves ready for processing. However it is acceptable to do nothing.

The last method, *onEndBuild*, is called at the end of the widget building process, after all widgets have been built and added to the *Layout*. *WidgetProcessors* may wish to act on this event to clean themselves up following processing. However it is acceptable to do nothing.

2.5.3 Defaults

Most *Metawidgets* have default *WidgetProcessors*. You can see the default by looking in the *Metawidget* JAR for the file *metawidget-xxx-default.xml* (where *xxx* is your target platform, such as *swing* or *struts*).

For reference, the defaults are:

Platform	Default
Android	(none)
GWT	(none)
JSF	<pre><array> <requiredAttributeProcessor /> <immediateAttributeProcessor /> <standardBindingProcessor /> <readableIdProcessor /> <labelProcessor /> <standardValidatorProcessor /> <standardConverterProcessor /> <cssStyleProcessor /> </array></pre>
JSP	(none)
Spring	(none)

Platform	Default
Struts	(none)
Swing	<pre><array> <reflectionBindingProcessor /> </array></pre>

2.5.4 Immutability

All `WidgetProcessors` are required to be immutable. This means you only need a single instance of a `WidgetProcessor` for your entire application. If you are using `metawidget.xml` then `ConfigReader` takes care of this for you, but if you are instantiating `WidgetProcessors` in Java code you should reuse instances.

Although individual `WidgetProcessors` are immutable, the `List` they are contained in can be changed. Methods such as `addWidgetProcessor` allows clients to dynamically add `WidgetProcessors` at runtime. This is useful for adding event handlers (see [Section 2.5.1, “Interface”](#)).

Note that immutable only means `WidgetProcessors` cannot be changed once instantiated - it does not mean they cannot be configured. Many `WidgetProcessors` have corresponding `xxxConfig` classes that allow them to be configured prior to instantiation in a type-safe way. For example, a `BeansBindingProcessor` can be configured in code:

```
metawidget.addWidgetProcessor( new BeansBindingProcessor( new BeansBindingProcessorConfig()
    .setUpdateStrategy( UpdateStrategy.READ_WRITE ) ) );
```

Or in `metawidget.xml`:

```
<beansBindingProcessor xmlns="java:org.metawidget.swing.widgetprocessor.binding.beansbinding"
  config="BeansBindingProcessorConfig">
  <updateStrategy>
    <enum>READ_WRITE</enum>
  </updateStrategy>
</beansBindingProcessor>
```

2.5.5 Implementing Your Own WidgetProcessor

Here is an example of a custom `WidgetProcessor` to add tooltips to add `JComponents`. It extends the code from the tutorial (see [Section 1.1, “Part 1 - The First Metawidget Application”](#)).

```
package com.myapp;

import static org.metawidget.inspector.InspectionResultConstants.*;
import java.util.*;
import javax.swing.*;
import org.metawidget.swing.*;
import org.metawidget.widgetprocessor.impl.*;

public class Main {

    public static void main( String[] args ) {
        Person person = new Person();

        SwingMetawidget metawidget = new SwingMetawidget();
        metawidget.addWidgetProcessor( new TooltipProcessor() );
        metawidget.setToInspect( person );
    }
}
```

```

JFrame frame = new JFrame( "Metawidget Tutorial" );
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.getContentPane().add( metawidget );
frame.setSize( 400, 250 );
frame.setVisible( true );
}

static class TooltipProcessor
    implements WidgetProcessor<JComponent, SwingMetawidget> {

    public JComponent processWidget( JComponent widget, String elementName,
        Map<String, String> attributes, SwingMetawidget metawidget ) {
        widget.setToolTipText( attributes.get( NAME ) );
        return widget;
    }
}

```

Like Inspectors, InspectionResultProcessors, WidgetBuilders and Layouts, WidgetProcessors are required to be immutable. However you can still make them configurable by using xxxWidgetProcessorConfig classes. For example:

```

public class TooltipProcessorConfig {
    private String mPrefix;

    public TooltipProcessorConfig setPrefix( String prefix ) {
        mPrefix = prefix;
        return this;
    }

    public String getPrefix() {
        return mPrefix;
    }

    // ...must override equals and hashCode too...
}

```

These xxxWidgetProcessorConfig classes are then passed to the WidgetProcessor at construction time, and stored internally:

```

public class TooltipProcessor {
    private String mPrefix;

    public TooltipProcessor( TooltipProcessorConfig config ) {
        mPrefix = config.getPrefix();
    }
}

```

This mechanism can then be controlled either programmatically:

```

metawidget.addWidgetProcessor( new TooltipProcessor( new TooltipProcessorConfig().setPrefix("Tip:") ));

```

Or through metawidget.xml:

```

<tooltipProcessor xmlns="java:com.foo" config="TooltipProcessorConfig">
  <prefix>
    <string>Tip:</string>
  </prefix>
</tooltipProcessor>

```

```
</prefix>
</tooltipProcessor>
```



Config classes must override equals and hashCode

If you want your configurable `WidgetProcessor` to be cacheable and reusable by `ConfigReader` and `metawidget.xml`, the `xxxWidgetProcessorConfig` class *must* override `equals` and `hashCode`.



Generate an XML Schema

If you intend your `WidgetProcessor` to be configurable via `metawidget.xml`, consider defining an XML Schema for it. This is optional, but allows users to validate their use of your `WidgetProcessor` in their `metawidget.xml` at development time. There is an Ant task, `org.metawidget.config.XmlSchemaGeneratorTask`, provided in the source distribution that can help with this by auto-generating the schema. All the existing Metawidget schemas are generated using this Ant task.

2.6 Layouts

Layouts arrange widgets on the screen, following their building by a `WidgetBuilder` and processing by any `WidgetProcessors`. This section covers Layouts in general. For in-depth documentation of individual Layouts see [Chapter 8, Layouts](#).

2.6.1 Interface

All Layouts must implement the `Layout` interface. This is a simple interface that defines only one method:

```
void layoutWidget(W widget,String elementName,Map<String,String> attributes,C container,M metawidget);
```

Where *W* is a widget class (ie. `Control`, `JComponent` etc), *C* is widget container class (ie. `Composite`, `JComponent` etc) and *M* is a Metawidget class (ie. `SwtMetawidget`, `SwingMetawidget` etc).

`layoutWidget` is called for each widget. Layouts should add the given *widget* as a child of the given *container*, according to the given *elementName* and *attributes*. They can use the given *metawidget* to access additional services if needed (such as state saving).

2.6.2 Advanced Interface

The `Layout` interface only has a single method. However for those needing more control over the `Layout` lifecycle there is an extended interface `AdvancedLayout`. This interface defines four additional methods:

```
void onStartBuild(M metawidget);

void startContainerLayout(W container,M metawidget);

void endContainerLayout(W container,M metawidget);

void onEndBuild(M metawidget);
```

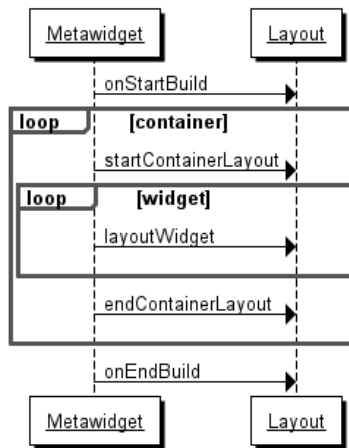


Figure 2.10. *startContainerLayout* and *endContainerLayout* are called for each container, *layoutWidget* is called for each widget

The first method, *onStartBuild*, is called at the start of the widget building process, before the *WidgetBuilder* is called. Layouts may wish to act on this event to initialize themselves ready for processing, or to perform 'outermost-container-only' processing, such as adding facets. However it is acceptable to do nothing.

The second method, *startContainerLayout*, is called to initialize the given *container*. It is acceptable to do nothing.

The third method, *endContainerLayout*, is called to finish the given *container*. It is acceptable to do nothing.

The last method, *onEndBuild*, is called at the end of the widget building process, after all widgets have been built and added to the *Layout*. Layouts may wish to act on this event to clean themselves up following processing, or to perform 'outermost-container-only' processing, such as adding facets. However it is acceptable to do nothing.

2.6.3 LayoutDecorator

LayoutDecorator allows you to combine multiple *Layouts* together in a hierarchy. Conceptually, this is similar to *CompositeInspector* or *CompositeWidgetBuilder*, but *Layouts* are fundamentally different in that most are 'end points' that cannot sensibly be composed into sequential lists (eg. what should happen if you try to combine a *GridBagLayout* with a *FlowLayout*?).

Rather, *Layouts* must be combined *hierarchically*, with an 'outer' *Layout* delegating to a single 'inner' *Layout*. *LayoutDecorator* is an abstract class that can be extended in order to decorate other *Layouts*. For example, *GridBagLayout* can be decorated using *TabbedPaneLayoutDecorator* to add tabbed section functionality.

```

<metawidgetLayout>
  <tabbedPaneLayoutDecorator xmlns="java:org.metawidget.swing.layout"
    config="TabbedPaneLayoutDecoratorConfig">
    <layout>
      <gridBagLayout />
    </layout>
  </tabbedPaneLayoutDecorator>
</metawidgetLayout>
  
```

A *LayoutDecorator* can also decorate another *LayoutDecorator* to provide fine-grained control over nested sections. For example, the business object...

```

public class Person {
  @UiSection( { "Person", "Name" } )
  public String firstname;
}
  
```

```

public String surname;

@UiSection( { "Person", "Contact Detail" } )
public String telephone;
}

```

...could be rendered using nested `TabbedPaneLayoutDecorators`...

```

<metawidgetLayout>
  <tabbedPaneLayoutDecorator xmlns="java:org.metawidget.swing.layout"
    config="TabbedPaneLayoutDecoratorConfig">
    <layout>
      <tabbedPaneLayoutDecorator config="TabbedPaneLayoutDecoratorConfig">
        <layout>
          <gridBagLayout />
        </layout>
      </tabbedPaneLayoutDecorator>
    </layout>
  </tabbedPaneLayoutDecorator>
</metawidgetLayout>

```

...as shown in [Figure 2.11](#).

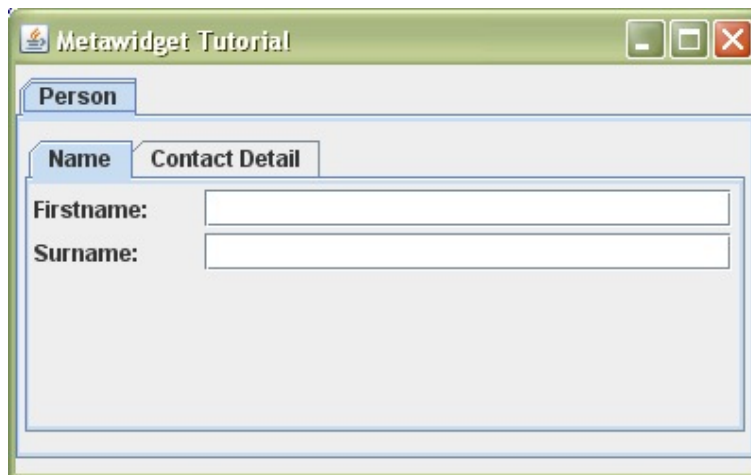


Figure 2.11. Nested `TabbedPaneLayoutDecorators`

Alternatively, it could use a `TabbedPaneLayoutDecorator` nested within a `SeparatorLayoutDecorator`...

```

<metawidgetLayout>
  <separatorLayoutDecorator xmlns="java:org.metawidget.swing.layout"
    config="SeparatorLayoutDecoratorConfig">
    <layout>
      <tabbedPaneLayoutDecorator config="TabbedPaneLayoutDecoratorConfig">
        <layout>
          <gridBagLayout />
        </layout>
      </tabbedPaneLayoutDecorator>
    </layout>
  </separatorLayoutDecorator>
</metawidgetLayout>

```

...as shown in [Figure 2.12](#).

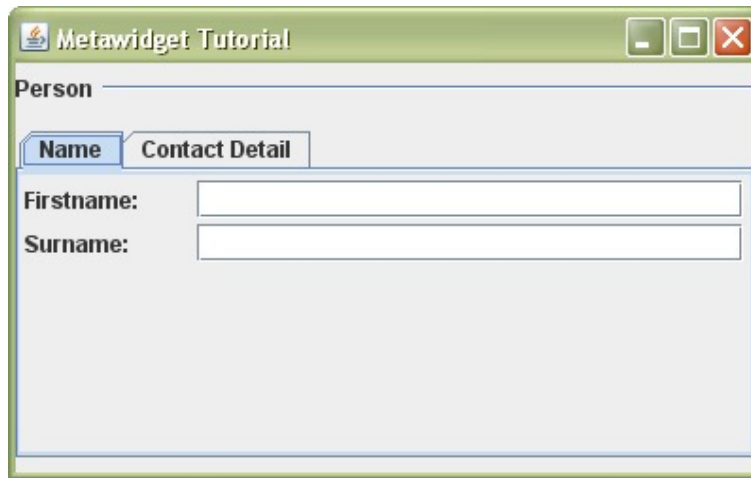


Figure 2.12. *TabbedPaneLayoutDecorator* within a *SeparatorLayoutDecorator*

Or the opposite - a *SeparatorLayoutDecorator* nested within a *TabbedPaneLayoutDecorator*...

```
<metawidgetLayout>
  <tabbedPaneLayoutDecorator xmlns="java:org.metawidget.swing.layout"
    config="TabbedPaneLayoutDecoratorConfig">
    <layout>
      <separatorLayoutDecorator config="SeparatorLayoutDecoratorConfig">
        <layout>
          <gridBagLayout />
        </layout>
      </separatorLayoutDecorator>
    </layout>
  </tabbedPaneLayoutDecorator>
</metawidgetLayout>
```

...as shown in [Figure 2.13](#).

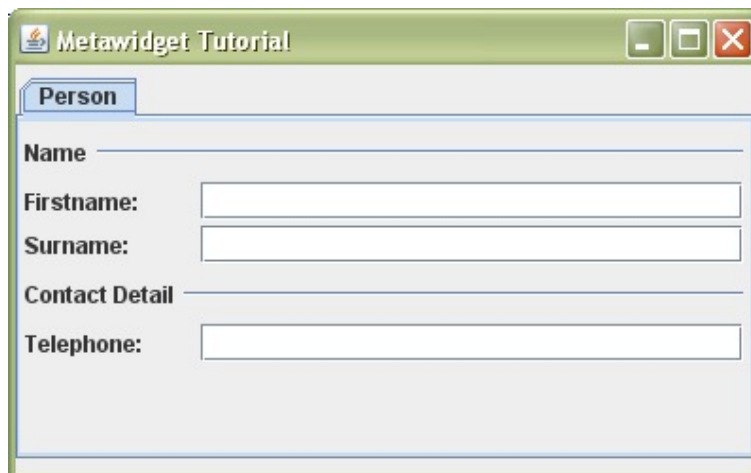


Figure 2.13. *SeparatorLayoutDecorator* within a *TabbedPaneLayoutDecorator*

2.6.4 Defaults

All Metawidgets have default Layouts. You can see the default by looking in the Metawidget JAR for the file `metawidget-xxx-default.xml` (where xxx is your target platform, such as swing or struts).

For reference, the defaults are:

Platform	Default
Android	<pre><textViewLayoutDecorator config="TextViewLayoutDecoratorConfig"> <layout> <tableLayout /> </layout> </textViewLayoutDecorator></pre>
GWT	LabelLayoutDecorator around a FlexTableLayout
JSF	<pre><outputTextLayoutDecorator config="OutputTextLayoutDecoratorConfig"> <layout> <simpleLayout/> </layout> </outputTextLayoutDecorator></pre>
JSP	<pre><headingTagLayoutDecorator config="HeadingTagLayoutDecoratorConfig"> <layout> <htmlTableLayout/> </layout> </headingTagLayoutDecorator></pre>
Spring	<pre><headingTagLayoutDecorator config="HeadingTagLayoutDecoratorConfig"> <layout> <htmlTableLayout/> </layout> </headingTagLayoutDecorator></pre>
Struts	<pre><headingTagLayoutDecorator config="HeadingTagLayoutDecoratorConfig"> <layout> <htmlTableLayout/> </layout> </headingTagLayoutDecorator></pre>
Swing	<pre><separatorLayoutDecorator config="SeparatorLayoutDecoratorConfig"> <layout> <gridBagLayout/> </layout> </separatorLayoutDecorator></pre>

2.6.5 Immutability

All Layouts are required to be immutable. This means you only need a single instance of a Layout for your entire application. If you are using `metawidget.xml` then `ConfigReader` takes care of this for you, but if you are instantiating Layouts in Java code you should reuse instances.

Note that immutable only means Layouts cannot be changed once instantiated - it does not mean they cannot be configured. Many Layouts have corresponding `xxxConfig` classes that allow them to be configured prior to instantiation in a type-safe way. For example, an `HtmlTableLayout` can be configured in code:

```
metawidget.setLayout( new HtmlTableLayout( new HtmlTableLayoutConfig().setNumberOfColumns( 2 ) );
```

Or in `metawidget.xml`:


```
<htmlTableLayout xmlns="java:org.metawidget.jsp.tagext.html.layout"
  config="HtmlTableLayoutConfig">
  <numberOfColumns>
    <int>2</int>
  </numberOfColumns>
</htmlTableLayout>
```

2.6.6 Implementing Your Own Layout

Here is an example of a custom Layout that arranges components in a bulleted HTML list. It could be useful for, say, arranging *action* elements that were represented by HTML anchor tags:

```
package com.myapp;

import java.util.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import org.metawidget.jsp.*;
import org.metawidget.jsp.tagext.*;
import org.metawidget.layout.iface.*;

public class HtmlListLayout
  implements AdvancedLayout<Tag, MetawidgetTag> {

  public void onStartBuild( MetawidgetTag metawidgetTag ) {}

  public void startContainerLayout( Tag containerTag, MetawidgetTag metawidgetTag ) {
    try {
      JspWriter writer = metawidgetTag.getPageContext().getOut();
      writer.write( "<ul>" );
    } catch ( Exception e ) {
      throw LayoutException.newException( e );
    }
  }

  public void layoutChild( Tag tag, String elementName, Map<String, String> attributes,
    Tag containerTag, MetawidgetTag metawidgetTag ) {
    try {
      JspWriter writer = metawidgetTag.getPageContext().getOut();
      writer.write( "<li>" );
      writer.write( JspUtils.writeTag( metawidgetTag.getPageContext(), tag, containerTag, null ) );
      writer.write( "</li>" );
    } catch ( Exception e ) {
      throw LayoutException.newException( e );
    }
  }

  public void endContainerLayout( Tag containerTag, MetawidgetTag metawidgetTag ) {
    try {
      JspWriter writer = metawidgetTag.getPageContext().getOut();
      writer.write( "</ul>" );
    } catch ( Exception e ) {
      throw LayoutException.newException( e );
    }
  }

  public void onEndBuild( MetawidgetTag metawidgetTag ) {}
}
```

Like `Inspectors`, `WidgetBuilders` and `WidgetProcessors`, `Layouts` are required to be immutable. However they will generally need to use some internal state, such as tracking the current row in a table layout. This can be achieved in two ways:

1. For state that will remain constant throughout the life of the `Layout`, such as a CSS class to put on a generated HTML table, use `xxxLayoutConfig` classes. For example:

```
public class HtmlTableLayoutConfig {
    private String mTableStyle;

    public HtmlTableLayoutConfig setTableStyle( String tableStyle ) {
        mTableStyle = tableStyle;
        return this;
    }

    public String getTableStyleClass() {
        return mTableStyleClass;
    }

    // ...must override equals and hashCode too...
}
```

These `xxxLayoutConfig` classes are then passed to the `Layout` at construction time, and stored internally:

```
public class HtmlTableLayout {
    private String mTableStyle;

    public HtmlTableLayout() {
        this( new HtmlTableLayoutConfig() );
    }

    public HtmlTableLayout( HtmlTableLayoutConfig config ) {
        mTableStyle = config.getTableStyle();
    }
}
```

This mechanism can then be controlled either programmatically:

```
metawidget.setLayout( new HtmlTableLayout( new HtmlTableLayoutConfig().setTableStyleClass("foo")) );
```

Or through `metawidget.xml`:

```
<htmlTableLayout xmlns="java:org.metawidget.jsp.tagext.html.layout" config="HtmlTableLayoutConfig">
    <tableStyleClass>
        <string>foo</string>
    </tableStyleClass>
</htmlTableLayout>
```



Config classes must override equals and hashCode

If you want your configurable `Layout` to be cacheable and reusable by `ConfigReader` and `metawidget.xml`, the `xxxLayoutConfig` class *must* override `equals` and `hashCode`.



Generate an XML Schema

If you intend your `Layout` to be configurable via `metawidget.xml`, consider defining an XML Schema for it. This is optional, but allows users to validate their use of your `Layout` in their `metawidget.xml`

at development time. There is an Ant task, `org.metawidget.config.XmlSchemaGeneratorTask`, provided in the source distribution that can help with this by auto-generating the schema. All the existing Metawidget schemas are generated using this Ant task.

- For state that will change during laying out, such as tracking the current row, store it in the Metawidget that is passed in to `startContainerLayout`, `layoutChild` and `endContainerLayout`. You may want to further wrap the state in a small helper class, for example:

```
public void layoutChild( JComponent component, String elementName, Map<String, String> attributes,
                        JComponent container, SwingMetawidget metawidget ) {
    getState( container ).currentRow++;
}

private State getState( SwingMetawidget metawidget ) {
    State state = (State) container.getClientProperty( getClass() );

    if ( state == null ) {
        state = new State();
        metawidget.putClientProperty( getClass(), state );
    }

    return state;
}

static class State {
    int currentRow;
}
```

2.7 metawidget.xml and ConfigReader

`metawidget.xml` is an alternate (and optional) way to configure Metawidget. It allows you to configure a Metawidget without writing any Java code. This can be useful in environments with intermediate languages that shield the developer from the raw Java, such as JSPs or Facelets. It can also be useful as a single place for configuring multiple Metawidgets, such as across multiple dialogs of a desktop application.

The `metawidget.xml` format, as parsed by `org.metawidget.config.ConfigReader`, is specialised for configuring Metawidget instances. The following sections explore some of the features of the XML format and `ConfigReader`.

2.7.1 Constructing New Objects

`ConfigReader` can construct new instances of objects. The XML element name is the Java class name and the XML namespace is the Java package. The following example constructs an `org.metawidget.swing.SwingMetawidget`.

```
<metawidget xmlns="http://metawidget.org"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://metawidget.org http://metawidget.org/xsd/metawidget-1.0.xsd">

  <swingMetawidget xmlns="java:org.metawidget.swing"/>

</metawidget>
```

Using the XML namespace to denote the Java package allows the (optional) plugging in of XML Schema validation on a per-package basis. For example:

```
<swingMetawidget xmlns="java:org.metawidget.swing"
xsi:schemaLocation="java:org.metawidget.swing http://metawidget.org/xsd/org.metawidget.swing-1.0.xsd"/>
```

2.7.2 Calling Setter Methods

Within an object, ConfigReader can call setXXX methods. The following example calls the setOpaque method of SwingMetawidget:

```
<metawidget xmlns="http://metawidget.org"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://metawidget.org http://metawidget.org/xsd/metawidget-1.0.xsd">

  <swingMetawidget xmlns="java:org.metawidget.swing">
    <opaque>
      <boolean>true</boolean>
    </opaque>
  </swingMetawidget>

</metawidget>
```

Multi-parameter methods are also supported. The following example calls the setParameter method of HtmlMetawidget (it takes two arguments):

```
<metawidget xmlns="http://metawidget.org"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://metawidget.org http://metawidget.org/xsd/metawidget-1.0.xsd">

  <htmlMetawidget xmlns="java:org.metawidget.faces.component.html">
    <parameter>
      <string>numberOfColumns</string>
      <int>2</int>
    </parameter>
  </htmlMetawidget>

</metawidget>
```

2.7.3 Constructing Primitive Types

As alluded to in the previous section, when calling setXXX methods the XML format can specify simple types. The previous example used boolean, string and int. Also supported are:

Element name	Java type
<array>	constructs a Java array. The array's component type (ie. String[], int[] etc) is based on the signature of the method being invoked (ie. setInspectors(Inspector...))
<boolean>	Java boolean primitive
<bundle>	uses ResourceBundle.getBundle to construct a ResourceBundle
<class>	Java Class
<constant>	Static field. This can either be fully qualified (eg. javax.swing.SwingConstants.LEFT) or just the field name, in

Element name	Java type
	which case the field must be defined by the class of the parent XML node
<code><enum></code>	Java enum primitive. The enum type is based on the signature of the method being invoked
<code><file></code>	uses <code>FileInputStream</code> to open a file as an <code>InputStream</code>
<code><int></code>	Java int primitive
<code><list></code>	constructs a <code>java.util.ArrayList</code>
<code><null></code>	Java null value
<code><pattern></code>	Java Pattern
<code><resource></code>	uses <code>Class.getResourceAsStream</code> to open a resource as an <code>InputStream</code>
<code><set></code>	constructs a <code>java.util.HashSet</code>
<code><string></code>	constructs a <code>java.lang.String</code>
<code><url></code>	uses <code>URL.openStream</code> to open a URL as an <code>InputStream</code>

2.7.4 Resolving Resources

Some environments store their resources in specialized locations that are inaccessible by normal means (ie. `ClassLoader.getResource`). For example, Web environments use a `WEB-INF` folder that must be accessed through `ServletContext.getResource`. Similarly, Android environments must resolve resources using `Context.getResources`.

`ConfigReader` and its specialized subclasses, such as `ServletConfigReader` and `AndroidConfigReader` understand this distinction and provide resource resolving capability to all the objects they create. Specifically, `ConfigReader` implements `ResourceResolver` and passes itself to any `xxxConfig` classes that implement `NeedsResourceResolver`.

2.7.5 Understanding Immutability

Metawidget dictates all `Inspectors`, `WidgetBuilders`, `WidgetProcessors` and `Layouts` be immutable. This is an important design decision as it means a single instance can be reused across an entire application. Immutability is enforced by not having any `setXXX` methods on the objects themselves. Rather, the `setXXX` methods are called on `Config` objects, which are then passed to the object's constructor. Once constructed, the object cannot be changed.

`ConfigReader` understands this distinction by way of a `config` attribute. The following example configures an immutable `Inspector`. The `setInspectors` method is called on `org.metawidget.inspector.composite.CompositeInspectorConfig` and then passed to `CompositeInspector`:

```
<compositeInspector xmlns="java:org.metawidget.inspector.composite" config="CompositeInspectorConfig">
  <inspectors>
    <array>
      <propertyTypeInspector xmlns="java:org.metawidget.inspector.propertytype"/>
      <metawidgetAnnotationInspector xmlns="java:org.metawidget.inspector.annotation" />
    </array>
  </inspectors>
</compositeInspector>
```

```
</inspectors>  
</compositeInspector>
```

Having constructed an immutable object, `ConfigReader` will cache the instance and reuse it. The `config` attribute defaults to using the same package as the `xmlns` (ie. `org.metawidget.inspector.composite` in the example above). This can be overridden if a fully qualified classname is provided.

**Config classes must override equals and hashCode**

In order to reliably cache and reuse an immutable object that uses a `config` attribute, the `xxxConfig` class *must* override `equals` and `hashCode`. This is important to bear in mind when implementing your own custom objects.

3. Metawidgets

Metawidget ships with native widgets for different UI frameworks. Whilst all Metawidget widgets are broadly similar, they are tailored to take advantage of their native environment. This chapter examines each Metawidget widget in detail.

3.1 Desktop Metawidgets

3.1.1 SwingMetawidget

SwingMetawidget is a Swing component. For an introduction to SwingMetawidget, see [Section 1.1, “Part 1 - The First Metawidget Application”](#) and [Section 1.2.1, “Desktop Address Book”](#).

Installation

There are two steps to installing SwingMetawidget within a Swing application:

1. Add `metawidget.jar` to your `CLASSPATH`.
2. You can (optionally) configure the Metawidget, either programmatically (as detailed in [Section 1.1.5, “Inspectors”](#)) or using a `metawidget.xml` file (as detailed in [Section 1.1.9, “Configuring Metawidget Externally”](#)).

Customizing Look and Feel

Since inception, Swing has had built-in, and extensive, Look and Feel support. Metawidget does not overlap this. For layouts, Swing supports a multitude of `LayoutManagers`. Metawidget leverages these, and automates them to construct UIs automatically.

3.1.2 SwtMetawidget

SwtMetawidget is an SWT composite. For an introduction to SwtMetawidget, see [Section 1.2.1, “Desktop Address Book”](#).

Installation

There are two steps to installing SwtMetawidget within an SWT application:

1. Add `metawidget.jar` to your `CLASSPATH`.
2. You can (optionally) configure the Metawidget, either programmatically (as detailed in [Section 1.1.5, “Inspectors”](#)) or using a `metawidget.xml` file (as detailed in [Section 1.1.9, “Configuring Metawidget Externally”](#)).

3.2 Web Metawidgets

3.2.1 GwtMetawidget

GwtMetawidget is a client-side, JavaScript widget for GWT. Despite the limitations of the JavaScript environment, GwtMetawidget supports reflection, annotations, pluggable layouts and data binding. For an introduction to GwtMetawidget, see [Section 1.2.2, “Web Address Book”](#) and [Section 1.3.9, “GWT Hosted Mode Examples”](#).

Installation

There are five steps to installing Metawidget within a GWT application:

1. Update the application's `.gwt.xml` module to include Metawidget:

```
<module>
  <inherits name="org.metawidget.GwtMetawidget" />
  ...
</module>
```

2. Include both `metawidget.jar` *and* `examples\gwt\metawidget-gwt.jar` in the `CLASSPATH` during the GWTCompiler phase. This provides the GwtMetawidget component.
3. Add `metawidget.jar` into `WEB-INF/lib`. This provides the GwtRemoteInspectorImpl servlet.
4. Update the application's `web.xml` to include GwtRemoteInspectorImpl:

```
<web-app>
  ...
  <servlet>
    <servlet-name>metawidget-inspector</servlet-name>
    <servlet-class>org.metawidget.inspector.gwt.remote.server.GwtRemoteInspectorImpl</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>metawidget-inspector</servlet-name>
    <url-pattern>/metawidget-inspector</url-pattern>
  </servlet-mapping>
</web-app>
```

5. You can (optionally) configure the default Inspector. To do this, add a `metawidget.xml` file into `WEB-INF`, and an `init-param` to your `web.xml`:

```
<init-param>
  <param-name>config</param-name>
  <param-value>metawidget.xml</param-value>
</init-param>
```

A working example of all five steps can be found in `addressbook-gwt.war` included in the binary distribution. You may also find the `example-gwt-addressbook` Ant task in the source distribution's `build.xml` useful.

Reflection and Annotations

GwtMetawidget leverages Metawidget's separate Inspector/renderer architecture and AJAX to perform server-side inspection as in [Figure 3.1](#). This allows GwtMetawidget to reflect properties and inspect annotations of business objects, even though JavaScript supports neither.

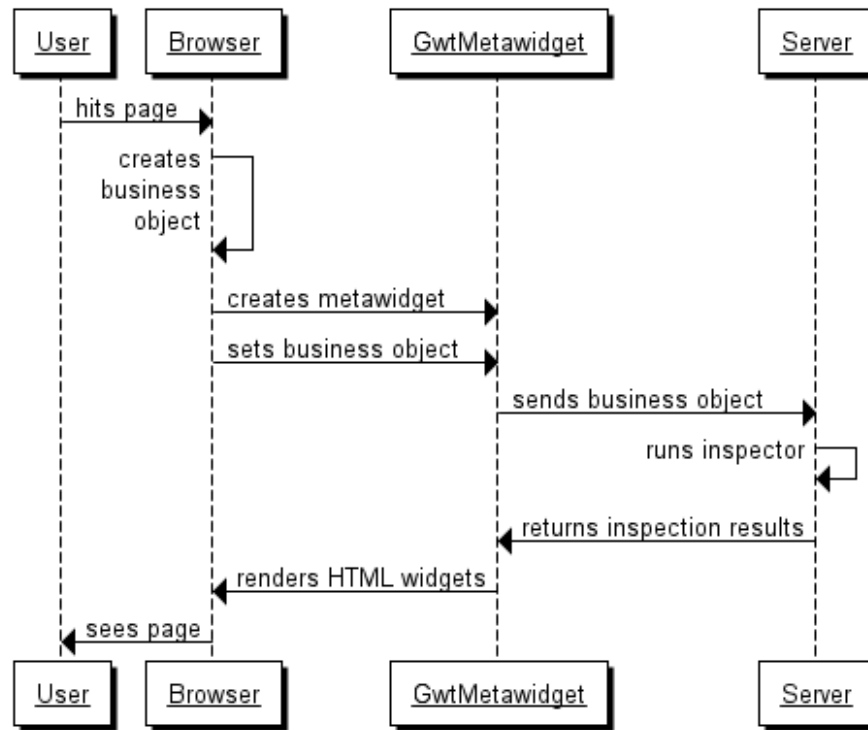


Figure 3.1. *GwtMetawidget uses AJAX to perform server-side inspection*

The process is:

1. instantiate the business object on the client-side as normal (ie. as JavaScript)
2. give the business object to `GwtMetawidget` (a client-side, JavaScript GWT Widget)
3. `GwtMetawidget` uses AJAX to pass the business object to the server
4. the server, using Java, runs all the `Inspectors` (including reflection and annotations)
5. the server returns the inspection results as an XML document
6. `GwtMetawidget` uses JavaScript to render the HTML widgets

Note that steps 3 and 5 (the AJAX call to and from the server) are the most costly in terms of performance. Techniques to improve GWT performance are discussed in [Section 10.3, “Rebinding”](#).

Client-Side Inspection

As noted in the section called “[Reflection and Annotations](#)” by default `GwtMetawidget` uses server-side inspectors. This allows the full power of Java-based reflection but carries the performance cost of an AJAX call. This cost can be mitigated by using rebinding (see [Section 10.3, “Rebinding”](#)), but there is another way: inspection can be performed *client-side*, with no AJAX calls.

Setting up a client-side `Inspector` is very easy. The default `GwtMetawidget` `Inspector` is `GwtRemoteInspectorProxy`, which is itself a client-side `Inspector` (one that makes a remote call to `GwtRemoteInspectorImpl`). To replace this default, simply implement your own `Inspector`:

```

public class MyClientSideInspector
    implements Inspector {
    public String inspect( Object toInspect, String type, String... names ) {

```

```

    return ...some XML string...
  }
}

```

Make sure this `Inspector` is located under the `client` folder of your GWT application so that it is compiled by the `GWTCompiler` into JavaScript. Use this `Inspector` by doing...

```
myGWTMetawidget.setInspector( new MyClientSideInspector() )
```

...which overrides the default `GwtRemoteInspectorProxy`. For an example of this technique see [Section 1.3.8, “GWT Client Side Example”](#).

3.2.2 HtmlMetawidgetTag (JSP)

Hidden Fields

Many Web applications store their data at the `HttpServletRequest` level, not at the `HttpSession` level. Using session-level state (or, ideally, a UI framework that supports some kind of 'conversation'-level state) is safer than passing variables to and from the client in hidden HTML fields. However, Web Metawidgets support that approach for those that need it through `setCreateHiddenFields(true)`.

3.2.3 UIMetawidget (JSF)

`UIMetawidget` is a Java Server Faces component. For an introduction to `UIMetawidget`, see [Section 1.2.2, “Web Address Book”](#) and [Section 1.3.2, “Seam Example”](#).

Installation

There are three steps to installing `UIMetawidget` within a JSF application:

1. Add `metawidget.jar` into `WEB-INF/lib`.
2. Add a tag library descriptor to the top of your JSP page...

```

<%@ taglib uri="http://metawidget.org/faces" prefix="m" %>
...
<m:metawidget value="#{foo}" />
...

```

...or Facelets page...

```

<ui:composition xmlns="http://www.w3.org/1999/xhtml "
...
xmlns:m="http://metawidget.org/faces">
...
<m:metawidget value="#{foo}" />
...

```

3. You can (optionally) configure the Metawidget by adding a `metawidget.xml` file into `WEB-INF`.

Customizing Look and Feel

One of JSF's most important Look and Feel technologies is CSS. Metawidget supports several approaches to suit different needs.

By convention, JSF's HTML widgets (`HtmlInputText`, `HtmlSelectBooleanCheckbox`, etc) define *style* and *styleClass* attributes for applying CSS styles and classes to their output. `HtmlMetawidget` follows this convention. When expanding to a single widget (such as an `HtmlInputText`) the CSS styles are applied to it. When expanding to multiple widgets, *all* widgets have the same CSS styles applied to them.

Another important JSF Look and Feel technology is Renderers. Whilst often Renderers are discussed in the context of rendering the same widget to different platforms (eg. HTML or WML), they can equally be used to render the same widget to the same platform but in different layouts.

`HtmlTableLayoutRenderer` is the default `LayoutRenderer`. It further defines parameters such as *tableStyle*, *labelStyle* and *columnStyleClasses* parameters (see the JavaDoc for a complete list). The latter is a comma separated list of CSS style classes to be applied to table columns. The first style class is the label column, the second the widget column, and the third the 'required' column. Further style classes may be used for multi-column layouts. You can get quite far using, for example:

```
.table-component-column input { width: 100%; }
```

..this approach has the advantage of automatically applying to every widget, so overridden widgets do not have to explicitly set *styleClass* information. However, not all Web browsers support fine-grained CSS selectors such as...

```
.table-component-column input[type="button"] { width: auto; }
```

...in which case it may be better to switch to using *styleClass* on `HtmlMetawidget` itself.

Other supplied `LayoutRenderers` include *div* and *simple* (see the JavaDoc, and the `META-INF/faces-config.xml` in `metawidget.jar` for a complete list).

3.2.4 SpringMetawidgetTag

`SpringMetawidgetTag` is a Spring taglib. For an introduction to `SpringMetawidgetTag`, see [Section 1.2.2, “Web Address Book”](#).

Installation

There are three steps to installing `SpringMetawidgetTag` within a Spring application:

1. Add `metawidget.jar` into `WEB-INF/lib`.
2. Add a tag library descriptor to the top of your JSP page:

```
<%@ taglib uri="http://metawidget.org/spring" prefix="m" %>
...
<m:metawidget value="#{foo}" />
...
```

3. You can (optionally) configure the Metawidget by adding a `metawidget.xml` file into `WEB-INF`.

Customizing Look and Feel

JSP-based technologies do not distinguish between what a widget is versus how it is rendered. Instead, `SpringMetawidgetTag` mimics this by providing loosely coupled `Layout` classes. Each layout can further be configured by using specific *param* tags.

Overriding Widget Creation

With regard to overriding widget creation, JSP-based technologies such as Spring lack some component-based features. Specifically, whilst it is possible for JSP tags to reference their *parent* (using `TagSupport.findAncestorWithClass`), they have no way to enumerate their *children*. Therefore, it is not possible to directly support arbitrary child tags within `SpringMetawidgetTag`. As a next best thing, Metawidget includes `StubTag` for wrapping arbitrary tags. It also supports wrapping arbitrary HTML.

3.2.5 StrutsMetawidgetTag

`StrutsMetawidgetTag` is a Struts taglib. For an introduction to `StrutsMetawidgetTag`, see [Section 1.2.2, “Web Address Book”](#).

Installation

There are three steps to installing `StrutsMetawidgetTag` within a Struts application:

1. Add `metawidget.jar` into `WEB-INF/lib`.
2. Add a tag library descriptor to the top of your JSP page:

```
<%@ taglib uri="http://metawidget.org/struts" prefix="m" %>
...
<m:metawidget value="#{foo}" />
...
```

3. You can (optionally) configure the Metawidget by adding a `metawidget.xml` file into `WEB-INF`.

Customizing Look and Feel

JSP-based technologies do not distinguish between what a widget is versus how it is rendered. Instead, `StrutsMetawidgetTag` mimics this by providing loosely coupled `Layout` classes. Each layout can further be configured by using specific *param* tags.

Overriding Widget Creation

With regard to overriding widget creation, JSP-based technologies such as Struts lack some component-based features. Specifically, whilst it is possible for JSP tags to reference their *parent* (using `TagSupport.findAncestorWithClass`), they have no way to enumerate their *children*. Therefore, it is not possible to directly support arbitrary child tags within `SpringMetawidgetTag`. As a next best thing, Metawidget includes `StubTag` for wrapping arbitrary tags. It also supports wrapping arbitrary HTML.

Troubleshooting

I get "Cannot find bean org.apache.struts.taglib.html.BEAN in any scope"

`StrutsMetawidgetTag` creates native Struts widgets, such as `<html:text>`, but does not create the surrounding Struts form. Make sure your Metawidget tag is enclosed in a `<html:form>` tag and the Struts HTML taglib is included at the top of the page.

I see "MultipartRequestHandler", "ServletWrapper" and other weird names

If you use `PropertyTypeInspector` to inspect your `ActionForm`-based classes, by default it will discover properties from the `org.apache.struts.action.ActionForm` base class, such as `getMultipartRequestHandler`. To prevent this, configure `metawidget.xml`:

```
<propertyTypeInspector xmlns="java:org.metawidget.inspector.propertytype"
  config="org.metawidget.inspector.impl.BaseObjectInspectorConfig">
  <propertyStyle>
    <javabeanPropertyStyle xmlns="java:org.metawidget.inspector.impl.propertystyle.javabean"
      config="org.metawidget.inspector.impl.propertystyle.BasePropertyStyleConfig">
      <excludeBaseType>
        <pattern>^( java| javax| org\.apache\.struts)\.*$</pattern>
      </excludeBaseType>
    </javabeanPropertyStyle>
  </propertyStyle>
</propertyTypeInspector>
```

3.3 Mobile Metawidgets

3.3.1 AndroidMetawidget

`AndroidMetawidget` is an Android widget. For an introduction to `AndroidMetawidget`, see [Section 1.2.3, “Mobile Address Book”](#).

Installation

There are two steps to installing `AndroidMetawidget` within an Android application:

1. Add `metawidget.jar` to your `CLASSPATH` (ie. under your project's *Libraries* tab in Eclipse)
2. You can (optionally) configure the Metawidget, either programmatically (as detailed in [Section 1.1.5, “Inspectors”](#)) or using a `metawidget.xml` file (as detailed in [Section 1.1.9, “Configuring Metawidget Externally”](#)) in your `res/raw` folder.



Note

Given the resource constraints of a mobile device, consider creating a custom `metawidget.jar` that only includes the classes you need (as detailed in [Section 10.1, “JAR Size”](#)).

Internationalization

`AndroidMetawidget` supports localization through the `setBundle` method. This method takes your `R.string` class, which is generated by Android from your `res/values/strings.xml` file. For example, if your `strings.xml` file was:

```
<resources>
  <string name="dob">Date of Birth</string>
</resources>
```

And your business object had a property:

```
class Person {
  ...
```

```
public Date getDob() {  
    return mDob;  
}  
}
```

Then you can set Metawidget to use your `R.string` class either programmatically or using `metawidget.xml`:

```
<androidMetawidget>  
  <bundle>  
    <class>org.metawidget.example.android.addressbook.R$string</class>  
  </bundle>  
</androidMetawidget>
```

And at runtime Metawidget will translate the `Person.getDob` method into a property called *dob*, resolve the integer `R.string.dob` and use that to look up the localized text in `strings.xml`. For different locales, put the `strings.xml` file in adjacent `res/values-xx` folders - for example `res/values-en`.

4. Inspectors

This chapter covers each `Inspector` in detail. For an explanation of how `Inspectors` fit into the overall architecture of `Metawidget`, see [Chapter 2, *Architecture*](#)

Throughout this chapter when we say 'returns the following attributes' this is a shorthand way of saying 'returns the following `Metawidget` attributes (via a piece of XML conforming to `inspection-result.xsd`). These are passed to the `Widget Builder` to assist with choosing appropriate widgets'. Quite which widget will be chosen is covered in [Chapter 6, *Widget Builders*](#): it could be a JSF `HtmlInputTextarea`, or a Swing `JTextArea`, or some other framework.

4.1 Property Inspectors

4.1.1 BaseObjectInspector

`BaseObjectInspector` underlies many of the `Inspectors` that inspect objects (as opposed to, say, XML files). It provides easy-to-override methods such as...

```
protected Map<String, String> inspectProperty( Property property )
```

...for inspecting properties, and...

```
protected Map<String, String> inspectAction( Action action )
```

...for inspecting actions, and finally...

```
protected Map<String, String> inspectTrait( Trait trait )
```

...for inspecting things that apply to both properties and actions (eg. `@UILabel`). Quite what constitutes a 'property' or an 'action' is decoupled into pluggable `PropertyStyles` and `ActionStyles`.

PropertyStyle

The `PropertyStyle` interface allows pluggable, fine-grained control over what is considered a 'property'. Different environments may have different approaches to defining what constitutes a property. For example, `JavaBean`-properties are convention-based, whereas `Groovy` has explicit property support. Equally, some environments may have framework-specific, base class properties that should be filtered out and excluded from the list of 'real' business model properties.

The default property style is `JavaBeanPropertyStyle`. To change it within `metawidget.xml`:

```
<propertyTypeInspector xmlns="java:org.metawidget.inspector.propertytype"
  config="org.metawidget.inspector.impl.BaseObjectInspectorConfig">
  <propertyStyle>
    <groovyPropertyStyle xmlns="java:org.metawidget.inspector.impl.propertystyle.groovy"/>
  </propertyStyle>
</propertyTypeInspector>
```

To change it programmatically:

```
BaseObjectInspectorConfig config = new BaseObjectInspectorConfig();
config.setPropertyStyle( new GroovyPropertyStyle() );
metawidget.setInspector( new PropertyTypeInspector( config ) );
```

JavaBeanPropertyStyle

The `JavaBeanPropertyStyle` is the default property style used by all `BaseObjectInspector` subclasses (which includes all annotation inspectors).

This property style recognizes JavaBean-convention `getXXX`, `setXXX` and `isXXX` methods, as well as public member variables. In addition, it maintains a cache of reflected classes for performance.



Note

When using getter methods with private members, make sure you annotate the getter *not the private member*. `JavaBeanPropertyStyle` cannot find annotations on private members, because the JavaBean specification does not define a way to determine which private members belong to which getters.

GroovyPropertyStyle

The `GroovyPropertyStyle` recognizes `GroovyBean` properties.

Groovy tries hard to make its `GroovyBean` properties compatible with JavaBean getters/setters, and indeed one can almost use the default `JavaBeanPropertyStyle` to read them. Unfortunately, `GroovyBeans` differ in that:

- annotations defined on properties are only attached to the (generated) private member variable, not the (generated) getter/setter methods.
- `GroovyBeans` define an implicit `getMetaClass` method which, although matching the JavaBean signature, should not be treated as a business model property.

JavassistPropertyStyle

The `JavassistPropertyStyle` extends `JavaBeanPropertyStyle` and makes use of `Javassist` for those environments that have it available.

`Javassist` is used to inspect the debug line numbering information embedded in JVM bytecode to sort getters/setters according to their original declaration order in the source code. This saves business objects having to use `@UiComesAfter` (or an XML file, or some other method) to impose an ordering.

However, a danger of this approach is that if the business objects are ever recompiled *without* debug line numbering information (eg. when moving from development to production) the UI fields will lose their ordering. Such a subtle bug may not be picked up, so as a safeguard `JavassistPropertyStyle` 'fails hard' with an `InspectorException` if line numbers are not available.

`JavassistPropertyStyle` uses the following sorting algorithm:

- superclass public fields come first, sorted by name.
- superclass methods come next, sorted by getter line number (or, if no getter, setter line number).
- public fields come next, sorted by name.
- methods come last, sorted by getter line number (or, if no getter, setter line number).

Note this algorithm is less flexible than `@UiComesAfter`, which can interleave superclass and subclass properties. However, it is possible to use both `@UiComesAfter` and `JavassistPropertyStyle` together to get the best of both worlds.

ScalaPropertyStyle

The `ScalaPropertyStyle` recognizes Scala properties.

Scala can make its properties compatible with JavaBean getters/setters, but only if you put special `@BeanProperty` annotations on them. Instead, `ScalaPropertyStyle` is designed to access Scala properties natively.

Implementing Your Own PropertyStyle

All property styles must implement the `PropertyStyle` interface. `BasePropertyStyle` assists in caching properties per class (looking them up is often expensive, involving reflection or similar techniques) and in excluding properties based on name, type or base class. Here is an example of a custom `PropertyStyle` that identifies fields based on `ResourceBundle` i18n entries. It extends the code from the tutorial (see [Section 1.1, “Part 1 - The First Metawidget Application”](#)).

```
package com.myapp;

import java.util.*;
import javax.swing.*;
import org.metawidget.inspector.iface.*;
import org.metawidget.inspector.impl.*;
import org.metawidget.inspector.impl.propertystyle.*;
import org.metawidget.inspector.impl.propertystyle.javabean.*;
import org.metawidget.inspector.propertytype.*;
import org.metawidget.swing.*;
import org.metawidget.util.*;

public class Main {

    public static void main( String[] args ) {
        Person person = new Person();

        SwingMetawidget metawidget = new SwingMetawidget();
        metawidget.setInspector( new PropertyTypeInspector( new BaseObjectInspectorConfig()
            .setPropertyStyle( new BundlePropertyStyle() ) ) );
        metawidget.setToInspect( person );

        JFrame frame = new JFrame( "Metawidget Tutorial" );
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.getContentPane().add( metawidget );
        frame.setSize( 400, 250 );
        frame.setVisible( true );
    }

    static class BundlePropertyStyle
        extends JavaBeanPropertyStyle {

        protected Map<String, Property> inspectProperties( Class<?> clazz ) {
            try {
                Map<String, Property> properties = CollectionUtils.newHashMap();
                ResourceBundle bundle = ResourceBundle.getBundle( "MyBundle" );

                for ( Enumeration<String> e = bundle.getKeys(); e.hasMoreElements(); ) {
                    String key = e.nextElement();
```

```

        properties.put( key, new FieldProperty( key, clazz.getField( key ) ) );
    }

    return properties;
}
catch ( Exception ex ) {
    throw InspectorException.newException( ex );
}
}
}
}

```

For brevity, this example extends `JavaBeanPropertyStyle`. Normally, you would want to extend `BasePropertyStyle` and, as well as overriding `inspectProperties` to locate the properties, implement the `Property` interface with mechanisms for interrogating the property.



Note

In this particular example, it may be useful to create a `BundlePropertyStyleConfig` class that implements `NeedsResourceResolver` (see [Section 2.7.4, “Resolving Resources”](#)). Then it could use `ResourceResolver.openResource` to locate the bundle in case it was in a specialized location (such as `WEB-INF/`).

ActionStyle

The `ActionStyle` interface allows pluggable, fine-grained control over what is considered an 'action'.

Different environments may have different approaches to defining what constitutes an action. For example, the `SwingAppFramework` uses an `@org.jdesktop.application.Action` annotation.

The default property style is `MetawidgetActionStyle`. To change it within `metawidget.xml`:

```

<metawidgetAnnotationInspector config="org.metawidget.inspector.impl.BaseObjectInspectorConfig">
  <actionStyle>
    <swingAppFrameworkActionStyle xmlns="java:org.metawidget.inspector.impl.actionstyle.swing">
  </actionStyle>
</metawidgetAnnotationInspector>

```

To change it programmatically:

```

BaseObjectInspectorConfig config = new BaseObjectInspectorConfig();
config.setActionStyle( SwingAppFrameworkActionStyle.class );
metawidget.setInspector( new MetawidgetAnnotationInspector( config ) );

```

Note these action styles only apply to `BaseObjectInspector` and its subclasses. This covers most annotation-recognising inspectors (eg. `JpaInspector`, `HibernateValidatorInspector`) but *not* XML-based inspectors. For example, `PageflowInspector` recognizes actions in JBoss jBPM pageflow files without any concept of an 'action style'.

MetawidgetActionStyle

The default `Metawidget` action style recognizes any method annotated with `@UiAction`. Action methods must not accept any parameters in their signature.

SwingAppFrameworkActionStyle

The `SwingAppFrameworkActionStyle` recognises `SwingAppFramework`'s `@Action` annotation as denoting an action.

4.1.2 PropertyTypeInspector

`PropertyTypeInspector` extends `BaseObjectInspector`, and so inherits its features. In addition, it returns the following attributes for the following business properties:

Metawidget Attribute	Property Type
<i>lookup</i>	lookup of 'true, false' if the type is Boolean
<i>lookup-labels</i>	lookup of 'Yes, No' if the type is Boolean. This will generally be localized by the Metawidget
<i>no-setter</i>	if the property has no <code>setXXX</code> method. Note <i>no-setter</i> is distinct from <i>read-only</i> , because it is common to have no setter for a complex type (eg. <code>Person.getAddress</code>) but this shouldn't make all its contents (eg. <code>Address.getStreet</code>) read-only.
<i>no-getter</i>	if the property has no <code>getXXX</code> method
<i>type</i>	declared type of the property
<i>actual-type</i>	if the actual type differs from the declared type (ie. it is a subclass)

4.1.3 Java5Inspector

`Java5Inspector` extends `BaseObjectInspector`, and so inherits its features. In addition, it returns the following attributes for the following business properties:

Metawidget Attribute	Java5 feature
<i>lookup</i>	values of enums, as returned by <code>.name()</code>
<i>lookup-labels</i>	labels of enums, as returned by <code>.toString()</code>
<i>parameterized-type</i>	if the property is using generics

4.2 Annotation Inspectors

Beyond the base issue of inspecting an object and its properties, a number of inspectors are focussed on third-party annotations. These annotation inspectors all extend `BaseObjectInspector`, and so inherit its features, but in addition they inspect the following frameworks.

4.2.1 BeanValidationInspector

`BeanValidationInspector` inspects Bean Validation (JSR 303) annotations. It returns the following attributes for the following business properties:

Metawidget Attribute	Bean Validation Annotation
<i>maximum-fractional-digits</i>	<code>@Digits(fraction=...)</code>
<i>maximum-integer-digits</i>	<code>@Digits(integer=...)</code>

Metawidget Attribute	Bean Validation Annotation
<i>maximum-length</i>	@Size(max=...)
<i>maximum-value</i>	@Max
<i>minimum-length</i>	@Size(min=...)
<i>minimum-value</i>	@Min
<i>required</i>	@NotNull

4.2.2 FacesInspector

FacesInspector inspects Java Server Faces-specific annotations. It returns the following attributes for the following business properties:

Metawidget Attribute	Annotation
(any)	<p>@UiFacesAttributes and @UiFacesAttribute - annotates an arbitrary Metawidget attribute, based on a Java Server Faces EL expression.</p> <p>If FacesInspectorConfig.setInjectThis is true, a special request-level attribute is injected into the FacesContext. This can be useful so that the EL expression can refer to the originating object (ie. #{this.name}).</p> <p>Unlike @UiFacesLookup, which fits into a well-defined place within the JSF framework (ie. <i>f:selectItems</i>), the @UiFacesAttribute expression is evaluated by the Inspector, not by the Metawidget. This means the Inspector must be able access to FacesContext. In practice this usually happens automatically, but in some cases it may be necessary to 'combine remote inspections' (see Section 9.3, "Combine Remote Inspections").</p>
<i>currency-code</i> , <i>currency-symbol</i> , <i>number-uses-grouping-separators</i> , <i>minimum-integer-digits</i> , <i>maximum-integer-digits</i> , <i>minimum-fractional-digits</i> , <i>maximum-fractional-digits</i> , <i>locale</i> , <i>number-pattern</i> and <i>number-type</i>	<p>@UiFacesNumberConverter - annotates the field should use the standard Faces NumberConverter.</p> <p>Note: the NumberConverter property <i>integerOnly</i> is not specified using this annotation, as it can be inferred from the property's type</p>
<i>date-style</i> , <i>locale</i> , <i>datetime-pattern</i> , <i>time-style</i> , <i>time-zone</i> and <i>datetime-type</i>	@UiFacesDateTimeConverter - annotates the field should use the standard Faces DateTimeConverter
<i>faces-ajax-event</i>	@UiFacesAjax - annotates the widget for this field should use AJAX in response to the given event
<i>faces-component</i>	@UiFacesComponent - annotates the field should be rendered using the given Faces UIComponent in the UI.

Metawidget Attribute	Annotation
	Use of this annotation does not bind the business class to the UI quite as tightly as it may appear, because JSF has a loosely coupled relationship between <code><component-name></code> and <code><component-class></code> , and a further loose coupling between <code><component></code> and <code><render-kit></code> - as defined in <code>faces-config.xml</code>
<code>faces-converter-id</code>	<p><code>UiFacesConverter</code> - annotates the field should use the given Faces converter in the UI.</p> <p>This annotation uses the converter <i>Id</i>, not the class. Whilst it is possible to specify the class through the XML (see <code>FacesInspectionResultConstants.FACES_CONVERTER_CLASS</code>), this does not work well for the annotations because they are applied to domain objects, whereas converters are UI-object. Using an annotation value of type <code>Class</code> would introduce dependencies in the wrong direction. Using a value of type <code>String</code> that contains a fully-qualified classname would work, but is brittle to refactoring</p>
<code>faces-lookup</code>	<code>@UiFacesLookup</code> - annotates the value in the field should belong to the set returned by the given EL expression
<code>faces-suggest</code>	<code>@UiFacesSuggest</code> - annotates the value in the field should be 'suggested' (ie. using a pop-up box) using the set returned by the given EL expression

4.2.3 HibernateValidatorInspector

`HibernateValidatorInspector` inspects Hibernate Validator annotations. It returns the following attributes for the following business properties:

Metawidget Attribute	Hibernate Validator Annotation
<code>maximum-fractional-digits</code>	<code>@Digits(fractionalDigits=...)</code>
<code>maximum-integer-digits</code>	<code>@Digits(integerDigits=...)</code>
<code>maximum-length</code>	<code>@Length(max=...)</code>
<code>maximum-value</code>	<code>@Max</code>
<code>minimum-length</code>	<code>@Length(min=...)</code>
<code>minimum-value</code>	<code>@Min</code>
<code>required</code>	<code>@NotNull</code> or <code>@NotEmpty</code>

4.2.4 JexlInspector

`JexlInspector` inspects `@UiJexlAttribute` annotations and sets arbitrary attributes based on the result of evaluating an Apache Commons JEXL expression. It can be used to introduce declarative UI scripting into environments that lack their own expression language (ie. JSP has an EL, Swing does not). For example:

```
import org.metawidget.inspector.common.jexl.*;

public class Person {
    public boolean retired;
    @UiJexlAttribute( name = "hidden", value = "!this.retired" )
    public BigDecimal pension;
}
```

This code returns a *hidden* attribute based on evaluating the JEXL expression *!this.retired* (where *this* refers to the runtime instance of the *Person* being inspected). It could be used to show/hide the *pension* field in response to the *retired* checkbox being checked.

The JEXL expression language also supports branching statements. For example:

```
import org.metawidget.inspector.common.jexl.*;

public class PersonController {

    @UiJexlAttribute( name = "label", value = "if ( this.readOnly ) 'Back'" )
    public void cancel() { ... }
}
```

This code overrides the *label* of an action to be either 'Back' or 'Cancel', depending on whether the *Person* was being edited. It is taken from the Swing Address Book sample (see [Section 1.2.1, “Desktop Address Book”](#)).

4.2.5 JpaInspector

JpaInspector inspects Java Persistence Architecture annotations. It returns the following attributes for the following business properties:

Metawidget Attribute	JPA Annotation
<i>hidden</i>	@Id, unless <i>JpaInspectorConfig.setHideIds</i> is false
<i>large</i>	@Lob
<i>maximum-length</i>	@Column(length=...)
<i>required</i>	@Column(nullable=false) or @ManyToOne(optional=false)

4.2.6 MetawidgetAnnotationInspector

As much as possible, *Metawidget* tries to inspect metadata from existing sources, without introducing new concepts. Where that is not sufficient, *MetawidgetAnnotationInspector* adds a handful of annotations:

Metawidget Attribute	Metawidget Annotation
(any)	@UiAttributes and @UiAttribute - a 'catch all' for denoting arbitrary UI metadata
(order of fields)	@UiComesAfter
<i>action</i>	@UiAction
<i>dont-expand</i>	@UiDontExpand - denotes a value should not be inspected and expanded into sub-widgets. This can be useful if, say, you have a read-

Metawidget Attribute	Metawidget Annotation
	only field and just want to display its top-level <code>toString()</code> rather than all its child properties
<i>hidden</i>	@UiHidden - denotes a value should be hidden in the UI. The value may still be rendered on the client, depending on the Metawidget (ie. for Web clients, may use a HTML hidden field)
<i>label</i>	@UiLabel - denotes the label to used in the UI. Can be a resource key if the UI is using resource bundles, or an EL expression if the UI has an expression language (ie. JSF)
<i>large</i>	@UiLarge - denotes the field should be 'large' in the UI (ie. a multi-line textbox)
<i>lookup</i>	@UiLookup - denotes the value in the field should belong to the given set of Strings
<i>masked</i>	@UiMasked - denotes a value should be masked in the UI (eg. a password field)
<i>read-only</i>	@UiReadOnly - denotes a value should be read-only in the UI
<i>read-only</i>	@UiReadOnly - denotes a value should be read-only in the UI
<i>section</i>	@UiSection - denotes the start of a logical grouping in the UI. Subsequent fields are assumed to belong to the same section until a different section heading is encountered. Sections can be cancelled using a section heading with an empty String. Sections can be nested by specifying multiple section names.
<i>wide</i>	@UiWide - denotes the field should be 'wide' in the UI, spanning all columns in a multi-column layout. 'Wide' is different to 'large', because 'large' implies a data size (ie. BLOB or CLOB) whereas 'wide' refers purely to spanning columns. Generally all 'large' fields are implicitly 'wide', but not all 'wide' fields are 'large'. For example, you may want a normal text field (not a text area) to span all columns.

4.2.7 OvalInspector

`OvalInspector` inspects `OVal` annotations. It returns the following attributes for the following business properties:

Metawidget Attribute	OVal Annotation
<i>maximum-length</i>	@Length(max=...) or @MaxLength
<i>maximum-value</i>	@Max or @Range(max=...)
<i>minimum-length</i>	@Length(min=...) or @MinLength
<i>minimum-value</i>	@Min or @Range(min=...)
<i>required</i>	@NotNull or @NotEmpty or @NotBlank

4.2.8 Troubleshooting

I get "java.lang.TypeNotPresentException"

If you are using Sun's implementation of Java, Metawidget's annotation support requires Java 5.0u6 or later, which includes a fix for this bug (Bug Parade ID: 6322301).

My inspector is not finding my annotations

Annotations are designed to 'silently fall away' in environments that do not support them: they never throw `ClassDefNotFoundError`. For example, if a JPA-annotated class is transferred to an application tier without `ejb3-persistence.jar` (or equivalent) in its classpath, the JPA annotations will disappear.

If this is the cause, either add the appropriate JAR to the tier, or consider implementing a remote inspector (see [Section 9.1](#), "Order Fields").

4.3 XML Inspectors

Whilst we don't necessarily encourage the use of XML-based metadata, if you *already* have XML configuration files in your architecture Metawidget will take advantage of them. Equally, XML can be useful for declaring 'ad hoc' UI entities that do not map to any Java class, as well as for declaring UI-specific attributes for existing Java classes (ie. if you prefer not to use annotations, or if you want to introduce additional 'virtual' properties).

Note when using XML-based metadata you should still try to avoid duplicating metadata that already exists in other parts of your application. For example, if you are also using `PropertyTypeInspector` in your `CompositeInspector` there is no need to duplicate the names and types of properties.

Once nice feature of XML is that ordering of child elements (such as `<property name=" ">`) is explicit, so XML-based inspectors make great 'first inspectors' for use within `CompositeInspector` (eg. you don't need to also use `@UiComesAfter`).

4.3.1 BaseXmlInspector

`BaseXmlInspector`'s config class, `BaseXmlInspectorConfig`, uses a `setInputStream` method to specify the location of the XML. This allows a variety of options for sourcing the XML. For example:

```
<xmlInspector xmlns=" java:org.metawidget.inspector.xml "
  config="XmlInspectorConfig">
  <inputStream>
    <resource>com/myapp/metawidget-metadata.xml</resource>
  </inputStream>
</xmlInspector>
```

And:

```
<xmlInspector xmlns=" java:org.metawidget.inspector.xml "
  config="XmlInspectorConfig">
  <inputStream>
    <url>http://myserver.com/my-xml.xml</url>
  </inputStream>
</xmlInspector>
```


As well as specifying multiple files (which will all be inspected as one):

```
<xmlInspector xmlns=" java:org.metawidget.inspector.xml "
  config="XmlInspectorConfig">
  <inputStreams>
    <array>
      <url>http://myserver.com/my-xml-1.xml</url>
      <url>http://myserver.com/my-xml-2.xml</url>
    </array>
  </inputStreams>
</xmlInspector>
```

This functionality is extended to all XML-based Inspectors.

Another useful piece of functionality applies when mixing XML-based Inspectors (eg. `XmlInspector`) and Object-based Inspectors (eg. `PropertyTypeInspector`) in the same application (ie. via `CompositeInspector`). If your XML-based Inspectors and your Object-based Inspectors are inspecting the same classes, you may encounter a problem in particular scenarios. Specifically, the Object-based Inspectors will always stop at null or recursive Object references, whereas the XML Inspectors (which have no knowledge of Object values) will continue. This can lead to the `WidgetBuilders` constructing a UI for a null Object, which may upset some `WidgetProcessors` (eg. `BeansBindingProcessor`).

If you are encountering this particular scenario, you can set `BaseXmlInspector.restrictAgainstObject`, whereby the XML-based Inspector will do a check for null or recursive Object references, and not return any XML.

4.3.2 CommonsValidatorInspector

`CommonsValidatorInspector` inspects Apache Commons Validator `validation.xml` files. It returns the following attributes for the following business properties:

Metawidget Attribute	Validator XML
<code>maximum-length</code>	<code><field depends="maxlength" /><var><var-name>maxlength</var-name>...</code>
<code>maximum-value</code>	<code><field depends="intRange" /><var><var-name>max</var-name>... (or <code>floatRange</code> or <code>doubleRange</code>)</code>
<code>minimum-length</code>	<code><field depends="minlength" /><var><var-name>minlength</var-name>...</code>
<code>minimum-value</code>	<code><field depends="intRange" /><var><var-name>min</var-name>... (or <code>floatRange</code> or <code>doubleRange</code>)</code>
<code>required</code>	<code><field depends="required" /></code>

4.3.3 HibernateInspector

`HibernateInspector` inspects Hibernate `hibernate.cfg.xml` and `mapping.hbm.xml` files. For the former, it iterates over `<session-factory>`'s `<mapping>` elements and inspects all mapping files. It returns the following attributes for the following business properties:

Metawidget Attribute	Hibernate XML
<i>hidden</i>	<code><id /></code>
<i>large</i>	<code><property type="clob" /></code>
<i>maximum-length</i>	<code><property length="..." /></code>
<i>parameterized-type</i>	<code><bag type="..." /></code> or <code><list type="..." /></code> or <code><set type="..." /></code>
<i>required</i>	<code><property not-null="true" /></code>

4.3.4 JexlXmlInspector

`JexlXmlInspector` inspects files in `inspection-result-1.0.xsd` format looking for XML attributes wrapped in `${...}` notation. It processes these attributes as JEXL expressions before returning them. For example:

```
<inspection-result>

  <entity type="com.myapp.Person">
    <property name="pension" hidden="${!this.retired}"/>
  </entity>

</inspection-result>
```

`JexlXmlInspector` is effectively equivalent to `JexlInspector` but uses XML files instead of annotations.

4.3.5 PageflowInspector

`PageflowInspector` inspects JBoss jBPM pageflow files looking for *page* nodes and their associated *transitions* to be used as actions. For example, this `pageflow.jpdl.xml` file...

```
<page name="confirm">
  <transition name="purchase" to="complete" />
  <transition name="cancel" to="cancel" />
</page>
```

...would return *purchase* and *cancel* as available actions for the *confirm* page. For an example of `PageflowInspector` in use, see [Section 1.3.4, “jBPM Example”](#).

4.3.6 SeamInspector

`SeamInspector` inspects Seam XML files for useful metadata. Specifically:

- Delegates `jbpm:pageflow-definitions` elements from `components.xml` to `PageflowInspector`.

4.3.7 XmlInspector

`XmlInspector` inspects files in `inspection-result-1.0.xsd` format. It can be used when no other inspector is available for the given attribute.

Some attributes accept multiple values, such as *lookup*. These can be supplied as a comma-separated string. The values will be trimmed for whitespace. If the values themselves contain commas, they can be escaped with the `\` character.

5. InspectionResultProcessors

This chapter covers each `InspectionResultProcessor` in detail. For an explanation of how `InspectionResultProcessors` fit into the overall architecture of Metawidget, see [Chapter 2, Architecture](#)

5.1 ComesAfterInspectionResultProcessor

`ComesAfterInspectionResultProcessor` sorts inspection results according to the *comes-after* attribute. This attribute can be created using the `@UiComesAfter` annotation (among other ways). For example, the following inspection result...

```
<entity type="Address Screen">
  <property name="city" comes-after="street">
  <action name="save" comes-after="state">
  <property name="state" comes-after="city">
  <property name="street">
</entity>
```

...would be sorted into...

```
<entity type="Address Screen">
  <property name="street">
  <property name="city" comes-after="street">
  <property name="state" comes-after="city">
  <action name="save" comes-after="state">
</entity>
```

The *comes-after* attribute can contain multiple names (comma separated) in which case the field will be sorted to come after all the named fields. Alternatively the attribute can be empty, in which case the field will come after every other field in the *entity*.

6. Widget Builders

This chapter covers each `WidgetBuilder` in detail. For an explanation of how `WidgetBuilders` fit into the overall architecture of `Metawidget`, see [Chapter 2, Architecture](#)

Throughout this chapter when we refer to 'Metawidget Attributes' we mean the intermediate XML that `Metawidget` passes between `Inspectors` and `WidgetBuilders`. Quite which `Inspector` set the attribute, and based on what, is covered in [Chapter 4, Inspectors](#). For example, the `maximum-value` attribute could be set by `HibernateValidatorInspector` based on the `@Max` annotation, or by `CommonsValidatorInspector` based on an `intRange` element in an XML file, or some other source.

6.1 Desktop Widget Builders

6.1.1 Swing Widget Builders

SwingWidgetBuilder

`SwingWidgetBuilder` is the default `WidgetBuilder` for `SwingMetawidget`. It instantiates the following widgets for the following `Metawidget` attributes:

Widget	Metawidget Attribute
javax.swing	
<code>JButton</code>	<code>action</code> (except when also <code>read-only</code>)
<code>JCheckBox</code>	<code>type="boolean"</code> (and <code>type="java.lang.Boolean"</code> when also <code>required</code>)
<code>JComboBox</code>	<code>lookup</code>
<code>JLabel</code>	<code>read-only</code> (except when also <code>type="java.util.Collection"</code> or <code>masked</code>)
<code>JPanel</code>	<code>read-only</code> and <code>masked</code>
<code>JPasswordField</code>	<code>masked</code> (except when also <code>read-only</code>)
<code>JSlider</code>	<code>type</code> is a primitive (except boolean and char) and has both <code>minimum-value</code> and <code>maximum-value</code>
<code>JSpinner</code>	<code>type</code> is a primitive (except boolean and char) and has only one, or neither <code>minimum-value</code> and <code>maximum-value</code> . For floats and doubles, the step size is governed by <code>maximum-fractional-digits</code>
<code>JTextArea</code>	<code>type="java.lang.String"</code> and <code>large</code> . The <code>JTextArea</code> is automatically wrapped in a <code>JScrollPane</code>

Widget	Metawidget Attribute
JTextField	<i>type</i> is a <code>java.lang.String</code> , <code>java.util.Date</code> , primitive wrapper (eg. <code>Integer</code> or <code>Float</code>) except <code>java.lang.Boolean</code> . Also if property is of unknown type but <i>dont-expand</i>
org.metawidget.swing	
Stub	<i>hidden</i> , <i>action</i> when also <i>read-only</i> , <i>type</i> =" <code>java.util.Collection</code> " when also <i>read-only</i>

SwingXWidgetBuilder

SwingXWidgetBuilder is a pluggable WidgetBuilder for the SwingX library. It is intended to be used in conjunction with the default SwingWidgetBuilder. It instantiates the following widgets for the following Metawidget attributes:

Widget	Metawidget Attribute
org.jdesktop.swingx	
JXDatePicker	<i>type</i> =" <code>java.util.Date</code> "

6.1.2 SWT Widget Builders

SwtWidgetBuilder

SwtWidgetBuilder is the default WidgetBuilder for SwtMetawidget. It instantiates the following widgets for the following Metawidget attributes:

Widget	Metawidget Attribute
org.eclipse.swt.widgets	
Button	<i>action</i> (except when also <i>read-only</i>)
Button(SWT.CHECK)	<i>type</i> =" <code>boolean</code> " (and <i>type</i> =" <code>java.lang.Boolean</code> " when also <i>required</i>)
Combo	<i>lookup</i>
Label	<i>read-only</i> (except when also <i>type</i> =" <code>java.util.Collection</code> " or <i>masked</i>)
Composite	<i>read-only</i> and <i>masked</i>
Text(SWT.MASKED)	<i>masked</i> (except when also <i>read-only</i>)
Scale	<i>type</i> is a primitive (except <code>boolean</code> and <code>char</code>) and has both <i>minimum-value</i> and <i>maximum-value</i>
Spinner	<i>type</i> is a primitive (except <code>boolean</code> and <code>char</code>) and has only one, or neither <i>minimum-value</i> and <i>maximum-value</i> . For floats and doubles, the step size is governed by <i>maximum-fractional-digits</i>

Widget	Metawidget Attribute
<code>Text(SWT.MULTI)</code>	<code>type="java.lang.String"</code> and <code>large</code>
<code>Text</code>	<code>type</code> is a <code>java.lang.String</code> , <code>java.util.Date</code> , primitive wrapper (eg. <code>Integer</code> or <code>Float</code>) except <code>java.lang.Boolean</code> . Also if property is of unknown type but <i>dont-expand</i>
org.metawidget.swt	
<code>Stub</code>	<i>hidden</i> , <i>action</i> when also <i>read-only</i> , <code>type="java.util.Collection"</code> when also <i>read-only</i>

6.2 Web Widget Builders

6.2.1 JSP Widget Builders

DisplayTagWidgetBuilder

`DisplayTagWidgetBuilder` is a pluggable `WidgetBuilder` for the `DisplayTag` library. It is intended to be used in conjunction with the `JSPHtmlWidgetBuilder`, `SpringWidgetBuilder` or `StrutsWidgetBuilder`. It instantiates the following widgets for the following Metawidget attributes:

Widget	Type of field
org.displaytag.tags	
<code>TableTag</code>	<code>type="java.util.Collection"</code> or an array, except when also <i>hidden</i> or <i>lookup</i> . The columns in the table are based on inspecting <i>parameterized-type</i> or the component type of the array. If neither can be determined, the table will only have a single column

HtmlWidgetBuilder

`HtmlWidgetBuilder` is the default `WidgetBuilder` for the `JSPHtmlMetawidgetTag`. Since JSP has only a light component model (ie. HTML tags such as `<input>` and `<select>` are just strings, not modelled as JSP tags), `HtmlWidgetBuilder` uses `org.metawidget.jsp.tagext.LiteralTag` to wrap string-based fragments. It returns the following literals for the following Metawidget attributes:

Widget	Metawidget Attribute
<code><input type="checkbox"></code>	<code>type="boolean"</code> (and <code>type="java.lang.Boolean"</code> when also <i>required</i>)
<code><input type="hidden"></code>	<i>hidden</i> when also <code>HtmlMetawidgetTag.setCreateHiddenFields</code> is set to true
<code><input type="password"></code>	<i>masked</i> (except when also <i>read-only</i>)
<code><input type="submit"></code>	<i>action</i> (except when also <i>read-only</i>)

Widget	Metawidget Attribute
<code><input type="text"></code>	<i>type</i> is a <code>java.lang.String</code> , <code>java.util.Date</code> , primitive wrapper (eg. <code>Integer</code> or <code>Float</code>) except <code>java.lang.Boolean</code> . Also if property is of unknown type but <i>dont-expand</i> . If <i>maximum-length</i> , adds <i>maxlength="..."</i>
<code><select></code>	<i>lookup</i>
raw text	<i>read-only</i> (except when also <i>type="java.util.Collection"</i> or <i>masked</i>). If <code>HtmlMetawidgetTag.setCreateHiddenFields</code> is set to true, further adds a <code><input type="hidden"></code> so that something gets POSTed back
<code><textarea></code>	<i>type="java.lang.String"</i> and <i>large</i>
org.metawidget.jsp.tagext	
StubTag	<i>hidden</i> , <i>action</i> when also <i>read-only</i> , <i>type="java.util.Collection"</i> when also <i>read-only</i>

SpringWidgetBuilder

SpringWidgetBuilder is the default WidgetBuilder for SpringMetawidgetTag, albeit used in conjunction with the JSP HtmlWidgetBuilder. Like HtmlWidgetBuilder, it simply returns fragments of HTML. The JSP component model is too light to support returning tags containing child tags (ie. a SelectTag containing OptionTags). SpringWidgetBuilder returns the following widgets for the following Metawidget attributes:

Widget	Metawidget Attribute
org.springframework.web.servlet.tags.form	
CheckboxTag	<i>type="boolean"</i> (and <i>type="java.lang.Boolean"</i> when also <i>required</i>)
HiddenInputTag	<i>hidden</i> when also <code>HtmlMetawidgetTag.setCreateHiddenFields</code> is set to true
InputTag	<i>type</i> is a <code>java.lang.String</code> , <code>java.util.Date</code> , primitive wrapper (eg. <code>Integer</code> or <code>Float</code>) except <code>java.lang.Boolean</code> . Also if property is of unknown type but <i>dont-expand</i> . If <i>maximum-length</i> , calls <code>setMaxlength</code>
PasswordInputTag	<i>masked</i> (except when also <i>read-only</i>)
raw text	<i>read-only</i> (except when also <i>type="java.util.Collection"</i> or <i>masked</i>). If <code>HtmlMetawidgetTag.setCreateHiddenFields</code> is set to true, further adds a HiddenInputTag so that something gets POSTed back
SelectTag	<i>lookup</i>
TextareaTag	<i>type="java.lang.String"</i> and <i>large</i>

Widget	Metawidget Attribute
org.metawidget.jsp.tagext	
StubTag	<i>action</i> when not <i>read-only</i>

StrutsWidgetBuilder

StrutsWidgetBuilder is the default WidgetBuilder for StrutsMetawidgetTag, albeit used in conjunction with the JSP HtmlWidgetBuilder. Like HtmlWidgetBuilder, it simply returns fragments of HTML. The JSP component model is too light to support returning tags containing child tags (ie. a SelectTag containing OptionTags). StrutsWidgetBuilder returns the following widgets for the following Metawidget attributes:

Widget	Metawidget Attribute
org.apache.struts.taglib.html	
CheckboxTag	<i>type</i> ="boolean" (and <i>type</i> ="java.lang.Boolean" when also <i>required</i>)
SelectTag	<i>lookup</i>
HiddenTag	<i>hidden</i> when also <i>HtmlMetawidgetTag.setCreateHiddenFields</i> is set to true
PasswordTag	<i>masked</i> (except when also <i>read-only</i>)
raw text	<i>read-only</i> (except when also <i>type</i> ="java.util.Collection" or <i>masked</i>). If <i>HtmlMetawidgetTag.setCreateHiddenFields</i> is set to true, further adds a <i>HiddenInputTag</i> so that something gets POSTed back
TextTag	<i>type</i> of <i>java.util.Date</i> , primitive wrapper (eg. <i>Integer</i> or <i>Float</i>) except <i>java.lang.Boolean</i> . Also if property is of unknown type but <i>dont-expand</i> . If <i>maximum-length</i> , calls <i>setMaxlength</i>
TextareaTag	<i>type</i> ="java.lang.String" and <i>large</i>
org.metawidget.jsp.tagext	
StubTag	<i>action</i> when not <i>read-only</i>

6.2.2 GWT Widget Builders

ExtGwtWidgetBuilder

ExtGwtWidgetBuilder is a pluggable WidgetBuilder for the ExtGWT library. It is intended to be used in conjunction with the default GwtWidgetBuilder. It instantiates the following widgets for the following Metawidget attributes:

Widget	Metawidget Attribute
com.extjs.gxt.ui.client.widget.form	
DateField	<i>type</i> ="java.util.Date"

Widget	Metawidget Attribute
Slider	<i>type</i> is a primitive (except boolean and char) and has both <i>minimum-value</i> and <i>maximum-value</i>

To build applications that use `ExtGwtWidgetBuilder`, include `metawidget.jar` *and* `examples\gwt\metawidget-gwt.jar` *and* `examples\gwt\metawidget-gwt-extgwt-client.jar` in the `CLASSPATH` during the GWTCompiler phase.

GwtWidgetBuilder

`GwtWidgetBuilder` is the default `WidgetBuilder` for `GwtMetawidget`. It instantiates the following widgets for the following Metawidget attributes:

Widget	Metawidget Attribute
com.google.gwt.user.client.ui	
Button	<i>action</i> (except when also <i>read-only</i>)
CheckBox	<i>type</i> ="boolean" (and <i>type</i> ="java.lang.Boolean" when also <i>required</i>)
Label	<i>read-only</i> (except when also <i>type</i> ="java.util.Collection" or <i>masked</i>)
ListBox	<i>lookup</i>
PasswordTextBox	<i>masked</i> (except when also <i>read-only</i>)
SimplePanel	<i>masked</i> and <i>read-only</i>
TextArea	<i>type</i> ="java.lang.String" and <i>large</i>
TextBox	<i>type</i> of <code>java.util.Date</code> , primitive wrapper (eg. Integer or Float) except <code>java.lang.Boolean</code> . Also if property is of unknown type but <i>dont-expand</i> . If <i>maximum-length</i> , calls <code>setMaxlength</code>
org.metawidget.gwt.client.ui	
Stub	<i>hidden</i> , <i>action</i> when also <i>read-only</i> , <i>type</i> ="java.util.Collection" when also <i>read-only</i>

6.2.3 JSF Widget Builders

HtmlWidgetBuilder

`HtmlWidgetBuilder` is the default `WidgetBuilder` for the JSF `HtmlMetawidget`. It instantiates the following widgets for the following Metawidget attributes:

Widget	Metawidget Attribute
javax.faces.component.html	

Widget	Metawidget Attribute
HtmlCommandButton	<i>action</i> (except when also <i>read-only</i>)
HtmlDataTable	<i>type</i> of List, DataModel or array, except when also <i>hidden</i> or <i>lookup</i> . The columns in the table are based on inspecting <i>parameterized-type</i> or the component type of the array. If neither can be determined, the table will only have a single column
HtmlInputHidden	<i>hidden</i> when <i>HtmlMetawidget.setCreateHiddenFields</i> is set to true
HtmlInputSecret	<i>masked</i> (except when also <i>read-only</i>)
HtmlInputText	<i>type</i> of <code>java.util.Date</code> , primitive wrapper (eg. Integer or Float) except <code>java.lang.Boolean</code> . Also if property is of unknown type but <i>dont-expand</i> . If <i>maximum-length</i> , calls <code>setMaxlength</code>
HtmlInputTextarea	<i>type</i> =" <code>java.lang.String</code> " and <i>large</i>
HtmlOutputText	<i>read-only</i> (except when also <i>type</i> =" <code>java.util.Collection</code> " or <i>masked</i>)
HtmlSelectBooleanCheckbox	<i>type</i> =" <code>boolean</code> " (and <i>type</i> =" <code>java.lang.Boolean</code> " when also <i>required</i>)
HtmlSelectManyCheckbox	<i>type</i> of List or array, with <i>lookup</i>
HtmlSelectOneListbox	<i>lookup</i>
org.metawidget.faces.component	
HtmlLookupOutputText	<i>read-only</i> with <i>lookup-label</i>
UIStub	<i>hidden</i> , <i>action</i> when also <i>read-only</i> , <i>masked</i> when also <i>read-only</i> , <i>type</i> =" <code>java.util.Collection</code> " when also <i>read-only</i>

IceFacesWidgetBuilder

IceFacesWidgetBuilder is a pluggable WidgetBuilder for the JSF UIMetawidget. It is intended to be used in conjunction with the default JSF HtmlWidgetBuilder. For an example, see [Section 1.3.5, “ICEfaces Example”](#). IceFacesWidgetBuilder instantiates the following widgets for the following Metawidget attributes:

Widget	Metawidget Attribute
com.icesoft.faces.component.ext	
HtmlCommandButton	<i>action</i> (except when also <i>read-only</i>)
HtmlInputSecret	<i>masked</i> (except when also <i>read-only</i>)
HtmlInputText	<i>type</i> of <code>java.util.Date</code> , primitive wrapper (eg. Integer or Float) except <code>java.lang.Boolean</code> . Also if property is of unknown type but <i>dont-expand</i> . If <i>maximum-length</i> , calls <code>setMaxlength</code>
HtmlInputTextarea	<i>type</i> =" <code>java.lang.String</code> " and <i>large</i>

Widget	Metawidget Attribute
HtmlSelectBooleanCheckbox	<i>type="boolean" (and <code>type="java.lang.Boolean"</code> when also required)</i>
HtmlSelectManyCheckbox	<i>type="boolean" (and <code>type="java.lang.Boolean"</code> when also required)</i>
HtmlSelectOneListbox	<i>lookup</i>
com.icesoft.faces.component	
SelectInputDate	<i>type="java.util.Date"</i> <i>Note: the ICEfaces SelectInputDate widget uses internal state to control its popup. Therefore Metawidget will not destroy and recreate it upon POSTback, as it does with most other widgets (in order to update them to reflect changes in business model state).</i>

All components are instantiated with *partialSubmit* set to true.

RichFacesWidgetBuilder

RichFacesWidgetBuilder is a pluggable WidgetBuilder for the JSF UIMetawidget. It is intended to be used in conjunction with the default JSF HtmlWidgetBuilder. For an example, see [the section called “Alternate Widget Libraries”](#).

RichFacesWidgetBuilder instantiates the following widgets for the following Metawidget attributes:

Widget	Metawidget Attribute
org.richfaces.component.html	
HtmlCalendar	<i>type="java.util.Date"</i>
HtmlColorPicker	<i>type="java.awt.Color"</i>
HtmlInputNumberSlider	<i>type is a primitive (except boolean and char) and has both <code>minimum-value</code> and <code>maximum-value</code></i>
HtmlInputNumberSpinner	<i>type is a primitive (except boolean and char) or a Number and has only one, or neither <code>minimum-value</code> and <code>maximum-value</code></i>
HtmlSuggestionBox	<i>faces-suggest</i>

TomahawkWidgetBuilder

TomahawkWidgetBuilder is a pluggable WidgetBuilder for the JSF UIMetawidget. It is intended to be used in conjunction with the default JSF HtmlWidgetBuilder. TomahawkWidgetBuilder instantiates the following widgets for the following Metawidget attributes:

Widget	Metawidget Attribute
org.apache.myfaces.custom	
HtmlInputFileUpload	<i>type="org.apache.myfaces.custom.fileupload.UploadedFile"</i>

6.3 Mobile Widget Builders

6.3.1 Android Widget Builders

AndroidWidgetBuilder

AndroidWidgetBuilder is the default WidgetBuilder for AndroidMetawidget. It instantiates the following widgets for the following Metawidget attributes:

Widget	Metawidget Attribute
android.widget	
CheckBox	<i>type="boolean"</i> (and <i>type="java.lang.Boolean"</i> when also <i>required</i>)
DatePicker	<i>type="java.util.Date"</i> when also <i>required</i>
EditText	<i>type</i> of primitive wrapper (eg. Integer or Float) except <i>java.lang.Boolean</i> . Also if property is of unknown type but <i>dont-expand</i> . If <i>large</i> , calls <i>setMinLines</i> . If <i>masked</i> , sets a <i>PasswordTransformationMethod</i> . If <i>numeric</i> , sets a <i>DigitsKeyListener</i> . If <i>maximum-length</i> , sets a <i>InputFilter.LengthFilter</i> . If <i>type</i> is a <i>Date</i> , sets a <i>DateKeyListener</i>
PasswordTextBox	<i>masked</i> (except when also <i>read-only</i>)
Spinner	<i>lookup</i>
TextView	<i>read-only</i> (except when also <i>type="java.util.Collection"</i>). If <i>masked</i> sets <i>View.INVISIBLE</i>
org.metawidget.gwt.client.ui	
Stub	<i>hidden</i> , <i>action</i> when also <i>read-only</i> , <i>masked</i> when also <i>read-only</i> , <i>type="java.util.Collection"</i> when also <i>read-only</i>

7. WidgetProcessors

This chapter covers each `WidgetProcessor` in detail. For an explanation of how `WidgetProcessors` fit into the overall architecture of `Metawidget`, see [Chapter 2, *Architecture*](#).

7.1 Desktop Widget Processors

7.1.1 Swing Widget Processors

Property Binding

Swing does not include an automatic `JComponent` to `Object` binding mechanism, but `Metawidget` supports third-party alternatives via `addWidgetProcessor`.

BeansBindingProcessor

`BeansBindingProcessor` binds properties using Beans Binding (JSR 295). It supports the various Beans Binding update strategies:

```
myMetawidget.addWidgetProcessor( new BeansBindingProcessor(
    new BeansBindingProcessorConfig()
        .setUpdateStrategy( UpdateStrategy.READ ) ) );
```

If set to `READ` or `READ_WRITE` (the default is `READ_ONCE`), the object being inspected must provide `PropertyChangeSupport`. If set to `READ_WRITE`, updates to the UI are automatically sync'ed back to the `setToInspect`, otherwise the client must manually call `save`:

```
myMetawidget.getWidgetProcessor( BeansBindingProcessor.class ).save( myMetawidget )
```

After `JComponents` have been generated for the initial `setToInspect`, clients can update their values to a new `Object` without a full re-inspection by using `rebind`:

```
myMetawidget.getWidgetProcessor( BeansBindingProcessor.class ).rebind( newObject, myMetawidget )
```

For more details, see [Section 10.3, “Rebinding”](#).

BeanUtilsProcessor

`BeanUtilsProcessor` binds properties using [Apache BeanUtils](#). It supports JavaBean and Scala property styles:

```
myMetawidget.addWidgetProcessor( new BeanUtilsBindingProcessor(
    new BeanUtilsBindingProcessorConfig()
        .setPropertyStyle( BeanUtilsBindingProcessorConfig.PROPERTYSTYLE_SCALA ) ) );
```

Updates to the UI can be saved back to the `setToInspect` by calling `save`:

```
myMetawidget.getWidgetProcessor( BeanUtilsBindingProcessor.class ).save( myMetawidget )
```

After JComponents have been generated for the initial `setToInspect`, clients can update their values to a new Object without a full re-inspection by using `rebind`:

```
myMetawidget.getWidgetProcessor( BeanUtilsBindingProcessor.class ).rebind( newObject, myMetawidget )
```

For more details, see [Section 10.3, “Rebinding”](#).

Action Binding

Swing supplies `javax.swing.Action` for binding JButtons to backing classes, and this is typically combined with Java-based reflection to support runtime binding. This is exactly what the default action binding, `ReflectionBinding`, does.

However, Metawidget makes action bindings pluggable to support other use cases. In particular, use cases where there *is* no backing class, and instead the JButton should invoke, say, an RPC call. Implement your own pluggable binding by implementing `WidgetProcessor` and use it by calling:

```
myMetawidget.addWidgetProcessor( new MyWidgetProcessor() );
```

7.1.2 SWT Widget Processors

DataBindingProcessor

`DataBindingProcessor` binds properties using `org.eclipse.core.databinding`. Once bound, values can be saved back from the UI by calling `save`:

```
myMetawidget.getWidgetProcessor( DataBindingProcessor.class ).save( myMetawidget )
```

ReflectionBindingProcessor

`ReflectionBindingProcessor` binds Buttons to backing classes using Java-based reflection to support runtime binding.

7.2 Web Widget Processors

7.2.1 GWT Widget Processors

Property Binding

Like most other Metawidgets, `GwtMetawidget` supports property binding. Property binding generally requires reflection, and GWT recommends using `Generators` to achieve this. As of the time of writing, however, much of the burden of implementation rests on the developer.

`GwtMetawidget` automates this burden by supplying a `SimpleBindingProcessor` implementation. This implementation is pluggable, so may be swapped out as and when later releases of GWT more fully support data binding.

`SimpleBindingProcessor` expects every domain object to be wrapped with a `SimpleBindingProcessorAdapter`. The supplied `SimpleBindingProcessorAdapterGenerator` automates this process. To configure it, add the following to the `application-name.gwt.xml` file...

```
<generate-with
  class="org.metawidget.gwt.generator.widgetprocessor.binding.simple.SimpleBindingProcessorAdapterGenerator">
  <when-type-assignable class="org.metawidget.example.shared.addressbook.model.Contact" />
</generate-with>
```

...and in the application code...

```
metawidget.addWidgetProcessor( new SimpleBindingProcessor(
  new SimpleBindingProcessorConfig().setAdapter( Contact.class,
    (SimpleBindingProcessorAdapter<Contact>) GWT.create(Contact.class)));
```



ClassCastException

If this line throws a `ClassCastException` casting to `SimpleBindingProcessorAdapter`, it means GWT is not applying the `SimpleBindingProcessorAdapterGenerator`. Check you've specified the correct class in your *generate-with* block.

Updates to the UI can be saved back to the `setToInspect` by calling `save`:

```
myMetawidget.getWidgetProcessor( SimpleBindingProcessor.class ).save( myMetawidget )
```

After Widgets have been generated for the initial `setToInspect`, clients can update their values to a new Object without a full re-inspection by using `rebind`:

```
myMetawidget.getWidgetProcessor( SimpleBindingProcessor.class ).rebind( newObject, myMetawidget )
```

For more details, see [Section 10.3, “Rebinding”](#).

Action Binding

GWT supplies `com.google.gwt.user.client.ui.ClickListener` for binding Buttons to backing classes. It is further possible to combine this with a generator to support runtime binding. This is exactly what the default action binding, `SimpleBindingProcessor`, does.

However, Metawidget makes action bindings pluggable to support other use cases. In particular, use cases where there is no backing class, and instead the Button should invoke, say, an RPC call. Implement your own pluggable binding by implementing `WidgetProcessor` and use it by calling...

```
myMetawidget.addWidgetProcessor( new MyWidgetProcessor() );
```

7.2.2 JSF Widget Processors

HiddenFieldProcessor

StandardValidationProcessor

By default, `UIMetawidget` uses `StandardValidator`, which adds *required*, `RangeValidator` and `LengthValidator` to all widgets.

You can implement your own validators by extending `org.metawidget.faces.component.validator.Validator`, or disable validation by setting `validatorClass=""`.

RichFacesProcessor

8. Layouts

This chapter covers each `Layout` in detail. For an explanation of how `Layouts` fit into the overall architecture of `Metawidget`, see [Chapter 2, *Architecture*](#).

8.1 Desktop Layouts

8.1.1 Swing Layouts

BoxLayout

Layout to simply output components one after another, with no labels and no structure, using `javax.swing.BoxLayout`. This is like `FlowLayout` (below), except it fills width. It can be useful for `JTable` `CellEditors`.

FlowLayout

Layout to simply output components one after another, with no labels and no structure, using `javax.awt.FlowLayout`. This is like `BoxLayout`, except it does not fill width. It can be useful for button bars.

GridBagLayout

Layout to arrange widgets using `javax.awt.GridBagLayout`. Widgets are arranged in a table, with one column for labels and another for the widget. This `Layout` recognizes the following parameters, configured using `GridBagLayoutConfig`:

Property	Description
<i>labelAlignment</i>	Such as <code>SwingConstants.LEFT</code> or <code>SwingConstants.RIGHT</code>
<i>labelFont</i>	Label font
<i>labelForeground</i>	Label foreground color
<i>labelSuffix</i>	Text to display after label text. Defaults to a colon (:
<i>numberOfColumns</i>	number of columns. Each label/component pair is considered one column
<i>requiredAlignment</i>	Such as <code>SwingConstants.LEFT</code> or <code>SwingConstants.RIGHT</code>
<i>requiredText</i>	Text to display for required fields. Defaults to a star (*)

GroupLayout

Layout to arrange widgets using `javax.swing.GroupLayout`. Widgets are arranged in a table, with one column for labels and another for the widget.

MigLayout

Layout to arrange widgets using `net.miginfocom.swing.MigLayout`. Widgets are arranged in a table, with one column for labels and another for the widget. This Layout recognizes the following parameters, configured using `MigLayoutConfig`:

Property	Description
<code>numberOfColumns</code>	number of columns. Each label/component pair is considered one column

SeparatorLayoutDecorator

LayoutDecorator to decorate widgets from different sections using a `JSeparator`. This LayoutDecorator recognizes the following parameters, configured using `SeparatorLayoutDecoratorConfig`:

Property	Description
<code>alignment</code>	One of <code>SwingConstants.LEFT</code> or <code>SwingConstants.RIGHT</code> .
<code>layout</code>	Metawidget Layout to use for laying out the sections, for example <code>org.metawidget.swing.layout.GridBagLayout</code> .

TabbedPaneLayoutDecorator

LayoutDecorator to decorate widgets from different sections using a `JTabbedPane`. This LayoutDecorator recognizes the following parameters, configured using `TabbedPaneLayoutDecoratorConfig`:

Property	Description
<code>layout</code>	Metawidget Layout to use for laying out the sections, for example <code>org.metawidget.swing.layout.GridBagLayout</code> .
<code>tabAlignment</code>	One of <code>SwingConstants.TOP</code> , <code>SwingConstants.BOTTOM</code> , <code>SwingConstants.LEFT</code> or <code>SwingConstants.RIGHT</code> as defined by <code>JTabbedPane.setTabAlignment</code> .

TitledPanelLayoutDecorator

LayoutDecorator to decorate widgets from different sections using a `JPanel` with a `TitledBorder`. This LayoutDecorator recognizes the following parameters, configured using `LayoutDecoratorConfig`:

Property	Description
<code>layout</code>	Metawidget Layout to use for laying out the sections, for example <code>org.metawidget.swing.layout.GridBagLayout</code> .

8.1.2 SWT Layouts

FillLayout

Layout to simply output components one after another, with no labels and no structure, using `org.eclipse.swt.layout.FillLayout`. This is like `RowLayout`, except it fills width. It can be useful for Table Editors.

GridLayout

Layout to arrange widgets using `org.eclipse.swt.layout.GridLayout`. Widgets are arranged in a table, with one column for labels and another for the widget. This Layout recognizes the following parameters, configured using `GridLayoutConfig`:

Property	Description
<i>labelAlignment</i>	Such as <code>SWT.LEFT</code> or <code>SWT.RIGHT</code>
<i>labelFont</i>	Label font
<i>labelForeground</i>	Label foreground color
<i>labelSuffix</i>	Text to display after label text. Defaults to a colon (:
<i>numberOfColumns</i>	number of columns. Each label/component pair is considered one column
<i>requiredAlignment</i>	Such as <code>SWT.LEFT</code> or <code>SWT.RIGHT</code>
<i>requiredText</i>	Text to display for required fields. Defaults to a star (*)

MigLayout

Layout to arrange widgets using `net.miginfocom.swt.MigLayout`. Widgets are arranged in a table, with one column for labels and another for the widget. This Layout recognizes the following parameters, configured using `MigLayoutConfig`:

Property	Description
<i>numberOfColumns</i>	number of columns. Each label/component pair is considered one column

RowLayout

Layout to simply output components one after another, with no labels and no structure, using `org.eclipse.swt.layout.RowLayout`. This is like `FillLayout`, except it does not fill width. It can be useful for button bars.

SeparatorLayoutDecorator

`LayoutDecorator` to decorate widgets from different sections using a `Label(SWT.SEPARATOR)`. This `LayoutDecorator` recognizes the following parameters, configured using `SeparatorLayoutDecoratorConfig`:

Property	Description
<i>alignment</i>	One of <code>SWT.LEFT</code> or <code>SWT.RIGHT</code> .
<i>layout</i>	Metawidget Layout to use for laying out the sections, for example <code>org.metawidget.swt.layout.GridLayout</code> .

TabFolderLayoutDecorator

LayoutDecorator to decorate widgets from different sections using a TabFolder. This LayoutDecorator recognizes the following parameters, configured using TabFolderLayoutDecoratorConfig:

Property	Description
<i>layout</i>	Metawidget Layout to use for laying out the sections, for example <code>org.metawidget.swt.layout.GridLayout</code> .
<i>tabLocation</i>	One of <code>SWT.TOP</code> or <code>SWT.BOTTOM</code> .

8.2 Web Layouts

8.2.1 GWT Layouts

FlexTableLayout

Layout to arrange widgets in a table, with one column for labels and another for the widget. This Layout recognizes the following parameters, configured using FlexTableLayoutConfig:

Property	Description
<i>columnStyleNames</i>	comma delimited string of CSS style classes to apply to table columns in order of: label, component, required
<i>footerStyleName</i>	CSS style class to apply to table footer
<i>numberOfColumns</i>	number of columns. Each label/component pair is considered one column
<i>tableStyleName</i>	CSS style class to apply to outer table tag

FlowLayout

Layout to simply output components one after another, with no labels and no structure. This Layout is suited to rendering single components, or for rendering components whose layout relies entirely on CSS.

LabelLayoutDecorator

LayoutDecorator to decorate widgets from different sections using a Label. This Layout recognizes the following parameters, configured using LabelLayoutDecoratorConfig:

Property	Description
<i>styleName</i>	CSS style class to apply to section label

TabPanelLayoutDecorator

LayoutDecorator to decorate widgets from different sections using a TabPanel.

8.2.2 JSF Layouts

Layouts in JSF behave a little differently to those for other Metawidgets. By and large, JSF already provides a pluggable mechanism (`javax.faces.render.Renderers`) that can be used for displaying a component in different ways. Metawidget leverages this mechanism via its `xxxLayoutRenderer` classes (see below).

However, JSF Renderers are not supposed to *modify* the component tree (say, to wrap a group of components into a tabbed panel). For those cases, Metawidget employs its usual Layout mechanism. Layouts are executed at component-building-time, which is before Renderer-time so can safely modify the component tree.

Separating `xxxLayout` and `xxxLayoutRenderer` classes in this way has the added advantage they can be mix-and-matched. For example, you can combine a `RichFacesLayout` (for wrapping components in tabs) with either a `HtmlTableLayoutRenderer` (to render the inside of the tabs as a table) or a `HtmlDivLayoutRenderer` (to render the inside of the tabs using *divs*).

HtmlDivLayoutRenderer

Layout to arrange widgets in HTML *DIV* tags, with one *DIV* per label and per widget, and an outer *DIV* around both. This Layout recognizes the following parameters (passed either as `<f:param>` tags or set via `<parameter>` in `metawidget.xml`):

Parameter	Description
<i>componentStyle</i>	CSS styles to apply to the component DIV. This is the style applied to the DIV <i>around</i> the component, not to the component itself. The widget component can be styled using the <i>style</i> attribute on the Metawidget tag
<i>divStyleClasses</i>	comma separated list of style classes to apply to the DIVs, in order of outer, label, required, component, errors
<i>inlineMessages</i>	whether to wrap input components with inline <code>h:message</code> tags. True by default
<i>labelStyle</i>	CSS styles to apply to the label DIV
<i>outerStyle</i>	CSS styles to apply to the outer DIV
<i>requiredStyle</i>	CSS styles to apply to the required DIV

HtmlTableLayoutRenderer

Layout to arrange components in a table, with one column for labels and another for the component. This Layout recognizes the following parameters (passed either as `<f:param>` tags or set via `<parameter>` in `metawidget.xml`):

Parameter	Description
<i>columns</i>	number of columns. Each label/component pair is considered one column
<i>columnClasses</i>	comma delimited string of CSS style classes to apply to table columns in order of: label, component, required
<i>componentStyle</i>	CSS styles to apply to required column
<i>footerStyle</i>	CSS styles to apply to table footer
<i>footerStyleClass</i>	CSS style class to apply to table footer
<i>headerStyle</i>	CSS styles to apply to table header
<i>headerStyleClass</i>	CSS style class to apply to table header
<i>inlineMessages</i>	whether to wrap input components with inline <code>h:message</code> tags. True by default
<i>labelStyle</i>	CSS styles to apply to label column
<i>labelSuffix</i>	suffix to put after the label name. Defaults to a colon (ie. 'Name:')
<i>requiredStyle</i>	CSS styles to apply to required column (ie. the star)
<i>rowClasses</i>	comma delimited string of CSS style classes to apply to alternating table rows
<i>tableStyle</i>	CSS styles to apply to outer table tag
<i>tableStyleClass</i>	CSS style class to apply to outer table tag

OutputTextLayoutDecorator

`LayoutDecorator` to decorate widgets from different sections using an `HtmlOutputText`. As dictated by the JSF spec, CSS styles and style classes applied to an `HtmlOutputText` are wrapped in an HTML `span` tag. Therefore you must use CSS...

```
display: block
```

...if you want to use margins or padding around the `HtmlOutputText`.

This `LayoutDecorator` recognizes the following parameters, configured using `OutputTextLayoutDecoratorConfig`:

Property	Description
<i>style</i>	CSS styles to apply to <code>HtmlOutputText</code>
<i>styleClass</i>	CSS style class to apply to <code>HtmlOutputText</code>

PanelLayoutDecorator

`LayoutDecorator` to decorate widgets from different sections using a RichFaces Panel. This `LayoutDecorator` recognizes the following parameters, configured using `PanelLayoutDecoratorConfig`:

Property	Description
<i>style</i>	CSS styles to apply to the panel

Property	Description
<i>styleClass</i>	CSS style class to apply to the panel

SimpleTogglePanelLayoutDecorator

LayoutDecorator to decorate widgets from different sections using a RichFaces SimpleTogglePanel. This LayoutDecorator recognizes the following parameters, configured using SimpleTogglePanelLayoutDecoratorConfig:

Property	Description
<i>style</i>	CSS styles to apply to the panel
<i>styleClass</i>	CSS style class to apply to the panel
<i>switchType</i>	Mechanism to use to open/close panels. Either 'client' or 'ajax'. Default is 'client'
<i>opened</i>	whether the panel is initially opened. Defaults to true

SimpleLayout

SimpleLayout is the default JSF Layout. It simply adds the widget as a child of the Metawidget, leaving everything up to the Renderer.

TabPanelLayoutDecorator

LayoutDecorator to decorate widgets from different sections using a RichFaces TabPanel. This LayoutDecorator recognizes the following parameters, configured using TabPanelLayoutDecoratorConfig:

Property	Description
<i>headerAlignment</i>	Defaults to 'left'

8.2.3 JSP Layouts

All of the supported JSP-based technologies (ie. 'pure' JSP, Spring and Struts) share the same Layouts.

HeadingTagLayoutDecorator

LayoutDecorator to decorate widgets from different sections using an HTML heading tag (ie. *H1*, *H2* etc). This LayoutDecorator recognizes the following parameters, configured using HeadingTagLayoutDecoratorConfig:

Property	Description
<i>style</i>	CSS styles to apply to heading tag
<i>styleClass</i>	CSS style class to apply to heading tag

HtmlTableLayout

Layout to arrange widgets in a table, with one column for labels and another for the widget. This Layout recognizes the following parameters, configured using HtmlTableLayoutConfig:

Property	Description
<i>columnStyleClasses</i>	comma delimited string of CSS style classes to apply to table columns in order of: label, component, required
<i>footerStyle</i>	CSS styles to apply to table footer
<i>footerStyleClass</i>	CSS style class to apply to table footer
<i>numberOfColumns</i>	number of columns. Each label/component pair is considered one column
<i>tableStyle</i>	CSS styles to apply to outer table tag
<i>tableStyleClass</i>	CSS style class to apply to outer table tag

SimpleLayout

Layout to simply output components one after another, with no labels and no structure. This Layout is suited to rendering single components, or for rendering components whose layout relies entirely on CSS.

8.3 Mobile Layouts

8.3.1 Android Layouts

LinearLayout

Layout to arrange widgets vertically, using `android.widget.LinearLayout`. This Layout recognizes the following parameters, configured using `LinearLayoutConfig`:

Property	Description
<i>labelStyle</i>	Android style (ie. <code>@com.myapp:style/label</code>) to apply to labels. The style should be defined within <code>res/values/styles.xml</code> . For example: <div data-bbox="558 1373 1157 1522" data-label="Text"> <pre><style name="section"> <item name="android:textSize">20sp</item> <item name="android:paddingTop">10px</item> <item name="android:paddingBottom">5px</item> </style></pre> </div>

TableLayout

Layout to arrange widgets in a table, with one column for labels and another for the widget, using `android.widget.TableLayout`. This Layout recognizes the following parameters, configured using `TableLayoutConfig`:

Property	Description
<i>labelStyle</i>	Android style (ie. <code>@com.myapp:style/label</code>) to apply to labels. The style should be defined within <code>res/values/styles.xml</code> . For example:

Property	Description
	<pre> <style name="label"> <item name="android:textSize">20sp</item> <item name="android:paddingTop">10px</item> <item name="android:paddingBottom">5px</item> </style> </pre>

TabHostLayoutDecorator

LayoutDecorator to wrap widgets in different sections inside a TabHost, as in [Figure 8.1](#).

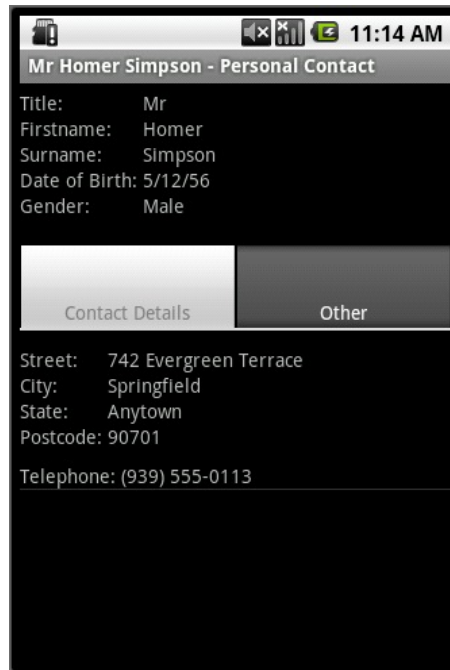


Figure 8.1. TabHostLayoutDecorator

This LayoutDecorator recognizes the following parameters, configured using LayoutDecoratorConfig:

Property	Description
<code>layout</code>	Metawidget Layout to use for laying out the inside of the tabs, for example <code>org.metawidget.android.widget.layout.TableLayout</code> .

TextViewLayoutDecorator

LayoutDecorator to decorate widgets from different sections using a TextView. This LayoutDecorator recognizes the following parameters, configured using TextViewLayoutDecoratorConfig:

Property	Description
<code>layout</code>	Metawidget Layout to use for laying out the sections, for example <code>org.metawidget.android.widget.layout.TableLayout</code> .
<code>style</code>	Android style (ie. <code>@com.myapp:style/section</code>) to apply to section breaks

9. How To's

This section contains 'How To's' (or 'Recipes') for various scenarios you may encounter when using Metawidget.

9.1 Order Fields

Metawidget supports several ways to control the order of fields in the UI, depending on your architecture and your preference:

- Annotate the fields with `@UiComesAfter` and use `MetawidgetAnnotationInspector` as the first inspector in your `CompositeInspector` chain.
- Use one of the XML-based inspectors (such as `XmlInspector` or `HibernateInspector`) as the first inspector in your `CompositeInspector` chain. XML nodes are inherently ordered.
- Compile your business model with debug information turned on, and use `JavassistPropertyStyle`. This approach uses Javassist to extract line numbering information, and order the fields in source file order.
- Write your own `Inspector`.
- Write your own `PropertyStyle`. For example, `JavassistPropertyStyle` extends `JavaBeanPropertyStyle` and reorders the fields using debug information.

9.2 Remote Inspection

Metawidget inspects back-end metadata and creates front-end UI widgets. If your application is split over multiple tiers, however, sometimes the back-end metadata is not accessible from the front-end. For example, annotations (such as JPA ones) are designed to 'fall away' if the class is transferred to a tier without JPA in its classpath. Equally, configuration files may not be accessible across tiers.

Metawidget supports these situations. Because each remoting environment is different, however, you will need to add a little code yourself. Every inspector returns an XML string, which is inherently serializable and safe to pass across tiers. Therefore, to run inspection remotely:

- create a back-end class suited to your environment, such as an EJB Session Bean. Have the back-end class instantiate an inspector - either programmatically:

```
XmlInspectorConfig config = new XmlInspectorConfig();
config.setInputStream( getClass().getResourceAsStream( "metawidget-backend-metadata.xml" ));
Inspector inspector = new XmlInspector( config );
```

Or by using a `metawidget.xml` file and the `ConfigReader` helper class:

```
inspector = ConfigReader.configure( "backend-metawidget.xml", Inspector.class );
```

- have the back-end class declare the `Inspector` interface. Implement the interface by delegating to the inspector you just instantiated.
- depending on your environment, it may further be necessary to create a front-end class (a proxy). Have it declare the `Inspector` interface. Implement the interface by remoting to the back-end class and returning the XML string.
- set the inspector on the `Metawidget` by using the `setInspector` method (rather than the `setConfig` method).

An example of this technique can be seen in `GwtRemoteInspectorProxy` and `GwtRemoteInspectorImpl`.

**Note**

All inspectors are both thread-safe and immutable. Therefore you only need one inspector for your *entire* application. Some remoting architectures support 'singletons' or 'service beans' well suited to this.

9.3 Combine Remote Inspections

If your architecture is strongly separated, some metadata may only be available in one tier (eg. JPA annotations in the back-end) and some only available in another tier (eg. `struts-config.xml` in the front-end).

For this, `CompositeInspector` supplies an overloaded method outside the normal `Inspector` interface. The overloaded `CompositeInspector.inspect` method takes an additional XML string of inspection results, and merges forthcoming inspection results with it.

Therefore, to combine metadata from different tiers:

- create a front-end class that implements the `Inspector` interface
- implement the interface by first remoting to the back-end class and returning the XML string, as before
- next, delegate to a `CompositeInspector` to inspect the front-end, passing it the XML string from the back-end as a starting point

10. Performance

Performance is very important to Metawidget. Whilst generating a UI dynamically - as opposed to hard coding it statically - is always likely to involve a performance tradeoff, Metawidget supports a number of techniques to help minimize this.

10.1 JAR Size

Metawidget has no mandatory third-party JAR dependencies and is highly modular. This allows the standard `metawidget.jar` to be repackaged for different environments to save JAR size.

For example, the `example-swing-addressbook-applet` Ant task builds `examples\swing\applet\addressbook\metawidget-applet.jar`. This JAR includes only those classes (`Inspectors`, `WidgetBuilders` and so on) necessary for the Swing Address Book example applet (it is further compressed using `pack200`). Similarly, the `example-android-addressbook` Ant task includes only those classes necessary for the Android Address Book.

10.2 Memory Usage

All `Inspectors`, `InspectionResultProcessors`, `WidgetBuilders`, `WidgetProcessors` and `Layouts` are immutable. This means you only need a single instance of them for your entire application. If you are using `metawidget.xml` then `ConfigReader` takes care of this for you, but if you are instantiating them yourself in Java code you should reuse instances to save memory.

10.3 Rebinding

For Metawidgets that do not use automatic binding, the general approach is to call `setToInspect` and then `setValue` to populate the generated widgets with values. This technique has an implicit side effect: the values can also be *repopulated* as many times as required from different objects, without re-calling `setToInspect`. This allows the Metawidget to be generated once and reused many times, mitigating the performance cost of generation.

For Metawidgets that *do* use automatic binding, however, `setValue` is never used. Setting new values requires re-calling `setToInspect` (and re-running generation) for every different object.

To avoid this some `WidgetProcessors` support a second, lightweight version of `setToInspect` called `rebind`. Using `rebind`, a `WidgetProcessor` can update the values in the generated widgets *without* re-running generation. This allows the Metawidget to be generated once and reused many times.

The downside of `rebind` is that the rebound object must have exactly the same set of field names as the original object. It becomes the responsibility of the caller to ensure this consistency.

For an example of using rebinding, see the GWT Address Book sample application.

11. Epilogue

That concludes the Metawidget User Guide and Reference Documentation.

For further documentation - including forums, bug reports, a Wiki community area and information on how to contribute - please visit <http://www.metawidget.org>.

Thank you for using Metawidget!