SwingML Custom Event Handlers

Developer's Tutorial

Robert J. Morris
CRC Press LLC

April 2003

oduction

ocument explains how to extend the standard SwingML event handling mechanism using the InvokableEvent interface. The rd method for invoking functionality in the SwingML specification is to use the <ACTION> tag and specify an object, its meth pes of the method's parameters, and then the corresponding values. This is illustrated in Listing 1.

g 1: An example of the ACTION tag

```
L NAME="mainPanel" LAYOUT="BorderLayout">
  <TEXTFIELD NAME="aTextBox" TEXT="" COLS="10" ORIENTATION="North"/>
  <BUTTON NAME="aButton" TEXT="Press Me" ORIENTATION="South">
        <LISTENER EVENT="ActionListener.actionPerformed">
              <ACTION
                    COMPONENT="aTextBox"
                    METHOD="setText"
                    TYPES="String"
                    VALUES="Hello, World"/>
        </LISTENER>
  </BUTTON>
EL>
```

xample outlined in Listing 1 sets the displayed text in the TEXTFIELD object identified as aTextBox to "Hello, World". This anism is very useful for single function calls. However, it becomes readily apparent that this mechanism does not satisfy every amming need. For this reason, the EXTERNAL-ACTION tag was designed to provide a means for programmers to "plug in" th vent handling mechanisms.

emainder of this document explains the EXTERNAL-ACTION tag as well as the means by which a programmer may implemen n event handlers.

EXTERNAL-ACTION Tag

XTERNAL-ACTION tag, like the ACTION tag resides within a LISTENER tag. This tag allows a SwingML coder to specify a onent that provides the custom event handling functionality. Listing 2 provides an example of this kind of functionality.

g 2: An example of the EXTERNAL-ACTION tag

```
L NAME="mainPanel" LAYOUT="BorderLayout">
  <TEXTFIELD NAME="aTextBox" TEXT="" COLS="10" ORIENTATION="North"/>
  <BUTTON NAME="aButton" TEXT="Press Me" ORIENTATION="South">
        <LISTENER EVENT="ActionListener.actionPerformed">
              <EXTERNAL-ACTION
                    COMPONENT="aTextBox"
                    EXTERNAL-CLASS="com.mycompany.event.MyWidget"/>
        </LISTENER>
  </BUTTON>
EL>
```

ustrated in Listing 2, The EXTERNAL-ACTION tag requires a reference to another component within the current document, in he aTextField object. It also requires the fully qualified path name of a Java class. The reference class must implement, at least vingml.event.InvokableEvent interface. This provides instances of the referenced class the necessary methods to be invoked pro SwingML event handler. The details of the InvokableEvent interface will be explained later in this document. For now, suffic at pressing the button invokes the functionality contained within the `com.mycompany.event.MyWidget` class.

so possible to specify parameters for an EXTERNAL-ACTION tag beyond the required COMPONENT and EXTERNAL-CLA
utes. To provide additional parameters to an EXTERNAL-ACTION, you use the ACTION-PARAM tag. The ACTION-PARA
equires a NAME attribute. The NAME of an ACTION-PARAM does not need to be unique within the SwingML document (it'
). The other attribute is VALUE which contains the value of the parameter. Optionally, the ACTION-PARAM tag can also con
ue rather than specify it in the VALUE attribute. This is illustrated in Listing 3.

g 3: An example of the ACTION-PARAM tag

```
L NAME="mainPanel" LAYOUT="BorderLayout">
  <TEXTFIELD NAME="aTextBox" TEXT="" COLS="10" ORIENTATION="North"/>
  <BUTTON NAME="aButton" TEXT="Press Me" ORIENTATION="South">
        <LISTENER EVENT="ActionListener.actionPerformed">
              <EXTERNAL-ACTION
              COMPONENT="aTextBox"
              EXTERNAL-CLASS="com.mycompany.event.MyWidget">
                    <ACTION-PARAM NAME="turn-on" VALUE="0"/>
                    <ACTION-PARAM NAME="display-text">
                    <![CDATA[
                    This is some additional text to display as part
                    of this custom action.
                    ]]>
                    </ACTION-PARAM>
              </EXTERNAL-ACTION>
        </LISTENER>
  </BUTTON>
EL>
```

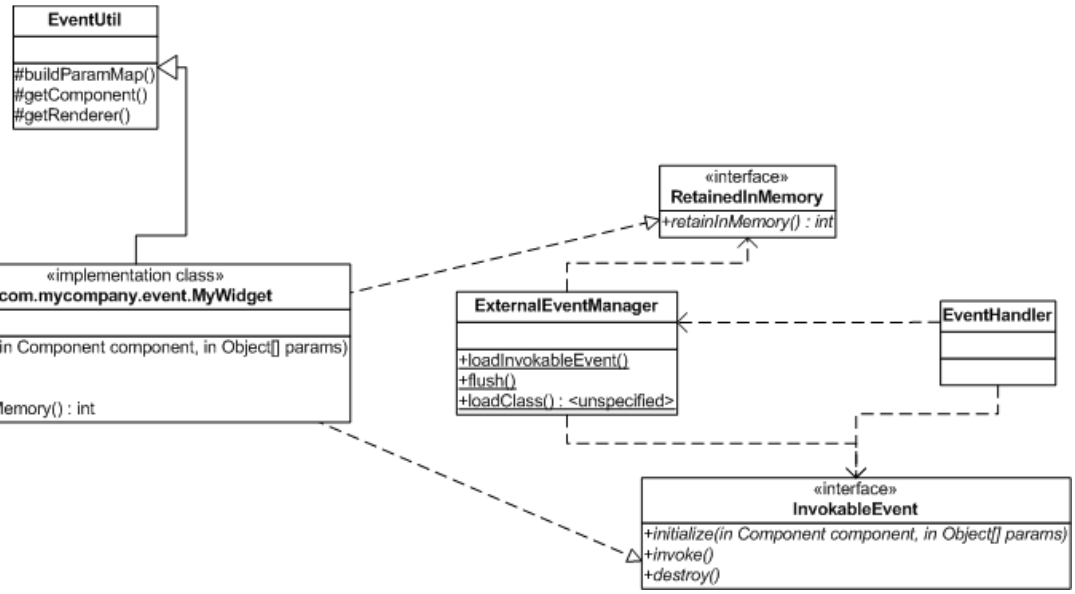
examine listing 3, you'll notice that two ACTION-PARAM tags were added to our SwingML code from the previous example.
CTION-PARAM uses the VALUE attribute to specify the value of the parameter. The second ACTION-PARAM encloses a
TA block. The CDATA block (`<![CDATA … ]]>`) allows the ACTION-PARAM to contain any kind of text including XML rese
cters like < and >.
values are supplied to the `com.mycompany.event.MyWidget` class when it is invoked.

s all that's required to place a reference to a custom event handler from the SwingML perspective. The next section will examin
aces you need to implement to successfully create a custom event handler Java class.

ting a Custom Event Handler

previous section, we placed a reference to the `com.mycompany.event.MyWidget` Java class. Obviously, in order for this SwingMI
we need to create the class that the SwingML EventHandler ultimately invokes. Before jumping into the code, however, we sh
ne the architecture of the SwingML Custom Event Handler.

e 1: Custom Event Architecture



ustrated in figure 1, The `MyWidget` class is a subclass of the `org.swingml.event.EventUtil` class. This utility class provides stand onality that facilitates the creation of new Event handlers. There is no need to derive your class from `EventUtil`. The `MyWidget` mplements the `InvokableEvent` interface and the `RetainedInMemory` interface. The `InvokableEvent` interface is required for Swin a class as an event handler. The `RetainedInMemory` interface tells the SwingML `EventHandler` mechanism how to manage the li of `MyWidget` instances.

ode for the InvokableEvent interface is illustrated below.

g 4: The InvokableEvent interface

```
c interface InvokableEvent

  public void initialize(Component component, Object[] params);
  public void invoke();
  public void destroy();
```

of these methods is explained below:

alize()

nethod provides an initialization routine for an `InvokableEvent` implementation. This method is called upon every invocation, lless of whether or not the implementation is retained in memory (This will be explained in the section regarding the `nedInMemory` interface).

omponent parameter is a reference to the component specified in the EXTERNAL-ACTION tag's COMPONENT attribute. The s parameter provides an array of `ActionParamModel` objects. These are best handled by the `EventUtil.buildParamMap` method wh s the array as a Map object where the name of the ACTION-PARAM is the key into the map.

ke()

s the "body" of the event handler. Once the necessary parameters for the event have been established within the `initialize()` d, the `invoke()` method performs the actual work of the event handler.

roy()

`estroy()` method releases any persistent, allocated resources like opened file streams, network connections, or database connecti

mple Event Handler

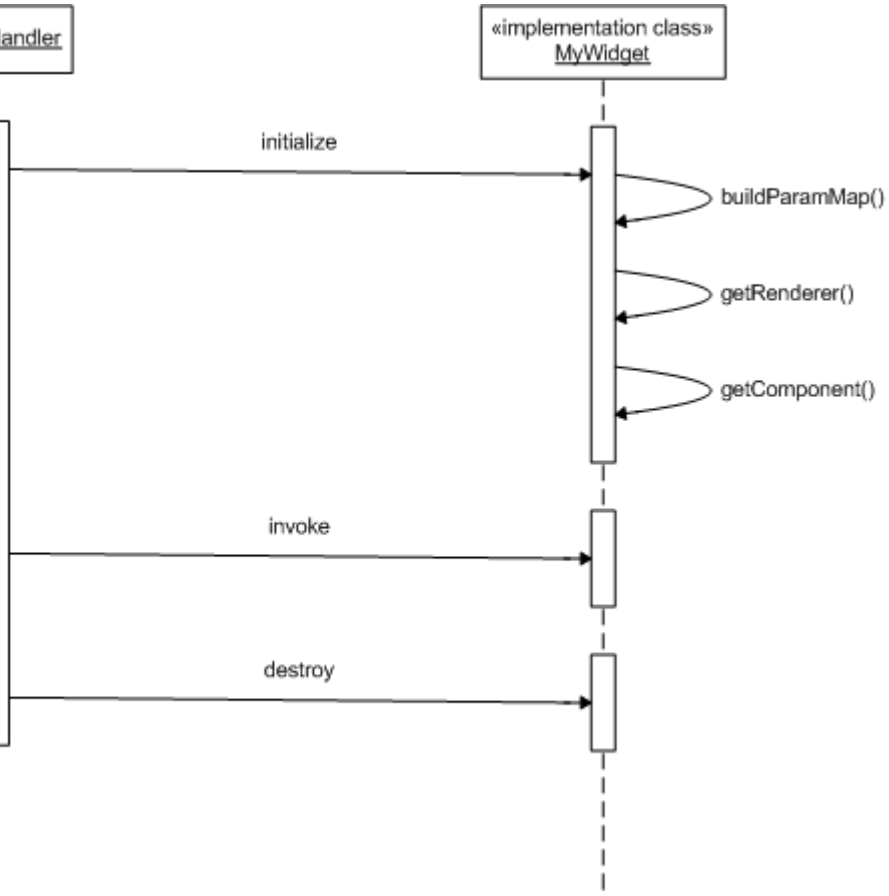
very least, a custom event handler must implement the `InvokableEvent` interface. The code for such a class could look like listi

g 5: MyWidget implementing InvokableEvent

```
ge com.mycompany.event;

t org.swingml.event.*;
t org.swingml.component.*;

c class MyWidget extends EventUtil implements InvokableEvent

 private Map m_params;
 private Component m_component;

 public void initialize(Component component, Object[] params)
 {
        this.m_component = component;
        this.m_params= super.buildParamMap(params);
 }

 public void invoke()
 {
        ((JTextFieldComponent) component).setText(this.m_params.get("MESSAGE"));

        if (((JTextFieldComponent) component).getText().length() % 2 != 0)
        {
               component.setVisible(false);
        }
        else
        {
               component.setVisible(true);
        }
 }

 public void destroy()
 {
        // nothing to do.
 }
```

's not much to the example in listing 5. The `initialize()` method takes the supplied parameters and packages them for future u
voke()` method. The `invoke()` method simply takes the referenced component, and calls its `setText()` method using the value o
AGE" parameter. The `destroy()` method has nothing to do. There is no bullet-proofing in this code and there are a lot of opportu
n-time failures. However, the basic idea is still there. A custom event handler is initialized, invoked, and then destroyed. The
nce of calls between the `EventHandler` class and the `MyWidget` class looks like figure 2.

e 2: Custom Event Handler Interaction



iscussion has involved only the implementation of the `InvokableEvent` interface. The `RetainedInMemory` interface and the probl
lifetime will be addressed in the next section.

RetainedInMemory Interface

the SwingML `EventHandler` class encounters a reference to a custom event handler, it must performs the following actions:

1. Load the class
2. Create an instance
3. Initialize the instance
4. Invoke the instance
5. Destroy the instance

3 through 5 were discussed in the previous section. Steps 1 through 2, however involve the actual task of creating the instance o
m event handler. That's where the `RetainedInMemory` interface and the `ExternalEventManager` figure in. The `EventHandler` class
`ternalEventManager` to create every instance of a custom event handler. The `ExternalEventManager` first tries to find a reference
lly qualified class name in its object cache. If one does not exist, it loads the class and checks to see if it implements the
`nedInMemory` interface. If the object does not implement `RetainedInMemory`, it simply returns the new instance. If the object does
ment the `RetainedInMemory` interface, it queries the `RetainedInMemory.retainInMemory()` method to determine how to cache the
. Once the object has been cached, it returns the instance.

are currently three levels of retention provided by the `ExternalEventManager`. These constants are exposed by the `RetainedInMe`
ace:

MEM_GLOBAL – When the class is first instantiated, the new instance remains cached until the entire SwingML system shu
down.

MEM_DOCUMENT – When the class is first instantiated, it is attached to an instance of the `SwingMLRenderer` class. Whenev
the `SwingMLRenderer` instance redraws a document (`render()` or `submit()`) or if it goes out of scope, the associated object cach
flushed using the `ExternalEventManager.flush(SwingMLRenderer renderer)` static method.

MEM_VOLATILE – This is just like not implementing the `RetaindInMemory` interface at all. This gives the programmer the
opportunity to say emphatically to anyone who may wish to subclass an event handler DO NOT CACHE AN INSTANCE OF
CLASS IN MEMORY.

revious example in listing 5 could be extended so that each instance is retained in document-level memory. This is illustrated in
6.

g 6: MyWidget with RetainedInMemory

```
ge com.mycompany.event;

t org.swingml.event.*;
t org.swingml.component.*;

c class MyWidget extends EventUtil implements InvokableEvent, RetainedInMemory

 private Map m_params;
 private Component m_component;

 public void initialize(Component component, Object[] params)
 {
       this.m_component = component;
       this.m_params = super.buildParamMap(params);
 }

 public void invoke()
 {
       ((JTextFieldComponent) component).setText(this.m_params.get("MESSAGE"));

       if (((JTextFieldComponent) component).getText().length() % 2 != 0)
       {
             component.setVisible(false);
       }
       else
       {
             component.setVisible(true);
       }
 }

 public void destroy()
 {
       // nothing to do.
 }

 public int retainInMemory()
 {
       return RetainedInMemory.MEM_DOCUMENT;
 }
```

old-faced code represents the only new text added to the previous example. The new `retainInMemory()` method only returns the of `RetainedInMemory.MEM_DOCUMENT`. This flag indicates to the `ExternalEventManager` that this instance should be associated wit t `SwingMLRenderer` instance and cached for the lifetime of its current document.

EventUtil Class

ted earlier, it is not necessary to extend the `EventUtil` class when creating your custom event handler. However, the protected u ons it provides can be very useful. Of greatest import are the `getRenderer()` method and the `getComponent()` method. Both of t le convenience methods for retrieving named components within the current SwingML hierarchy.

Renderer()

```
c SwingMLRenderer getRenderer(Component component)
```

lass will take the `component` reference (this is usually the one provided by the component parameter in the `ableEvent.initialize()` method) to "climb" the object tree back to the `SwingMLRenderer` root. The `component` parameter must b f a SwingML document tree or this method will return `null`.

Component()

```
c Component getComponent(Component component, String name)
```

nethod takes the `component` reference (this is usually the one provided by the `component` parameter in the `ableEvent.initialize()` method) to "walk" the object tree to find the object whose name matches the string specified by the `nam` eter. The `component` parameter must be a part of a SwingML document tree or this method will return `null`.