ULC

Extension

Guide

RIASUITE

canoo

Canoo Engineering AG

Kirschgartenstrasse 5

CH-4051 Basel

Switzerland

Tel: +41 61 228 9444

Fax: +41 61 228 9449

ulc-info@canoo.com

http://ulc.canoo.com

DISCLAIMER OF WARRANTIES

This document is provided “as is”. This document could include technical inaccuracies or typographical errors. This document is provided for informational purposes only, and the information herein is subject to change without notice. Please report any errors herein to Canoo Engineering AG. Canoo Engineering AG does not provide any warranties covering and specifically disclaim any liability in connection with this document.

TRADEMARKS

Java and all Java-based trademarks are registered trademarks of Sun Microsystems, Inc.

All other trademarks referenced herein are trademarks or registered trademarks of their respective holders

.

Contents

1 Overview

ULC includes a standard set of widgets, suitable for most applications. In some cases however, it may be necessary to add new widgets to the library, or to customize existing widgets. For instance a developer may want to customize *ULCTable* or add a spreadsheet or a business graphic widget. In other cases, a developer may want to introduce new data types that format and validate text values in a label or a textfield. For all such cases, both the ULC library and the UI Engine need to be extended. This guide explains the process of implementing a new ULC widget and extending the UI Engine to recognize and render the new widget.

This document starts with an explanation of some basic architecture concepts behind ULC. In Section 3 it gives an introductory example on how to extend ULC by writing a custom widget. Section 4 describes the structure and behaviour of all ULC methods and classes, which are relevant in the context of ULC extensions.

2 Half Object Pattern

Each ULC widget consists of a client-side and a server-side implementation. These implementations are called “half objects”. The client-side half object is responsible for display, user interaction, and event forwarding to the server. It adapts the so called basic widget, which provides the graphical representation at the client’s GUI. The server-side half object represents a proxy to the client-side half object and does not have a graphical representation.

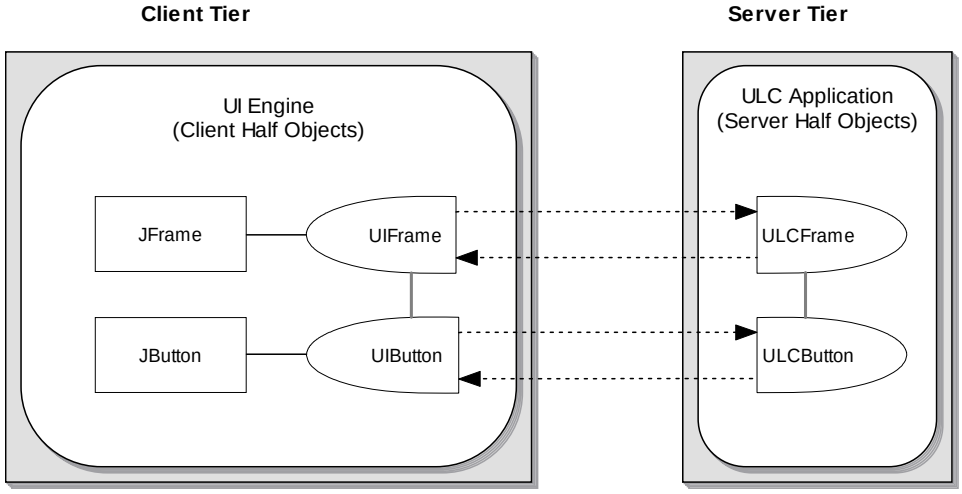

Figure 1: Half Object Pattern

The above diagram illustrates the Half Object pattern. In the example, *JFrame* is the basic widget. *UIFrame* is the client half object that adapts *JFrame* for interaction with the UI Engine. *ULCFrame* is the server-side half object, which has an API similar to the basic widget. In other words, it acts as a server-side proxy to the real widget and provides a server-side API to the basic widget.

Implementing a ULC widget requires the following steps:

- Implement the basic widget class, if the widget class does not already exist.
- Implement the client-side half object class that adapts the basic widget for interaction with the server-side half object via the UI Engine.
- Implement the server-side half object class that mimics the API of the basic widget class.

In general, the half object pattern can be applied to visual objects (i.e. basic widgets) as well as non-visual objects. A non-visual example is a client-side timer object, which sends keep-alive events to the server. Therefore, the rest of this guide often refers to the generalized term “*basic object*” instead of “basic widget”.

3 Introductory Example

This section shows how to implement a new ULC widget. The purpose of a ULC widget is to provide the same (or a similar) API as the basic object (e.g., a third-party or a custom Swing component) and therefore act as a server-side proxy to the basic object.

This section explains the steps required to add a *PieChart* widget to the ULC widget set. The *PieChart* widget can then be used by developers to display pie charts for given data. The *PieChart* widget is shown below. Please note that the *PieChart* widget is a custom implementation created just for the sake of demonstrating how to extend of ULC; developers should not confuse this with the charting package *com.ulcjava.base.server.chart* that provides out-of-the-box support for charting in ULC.

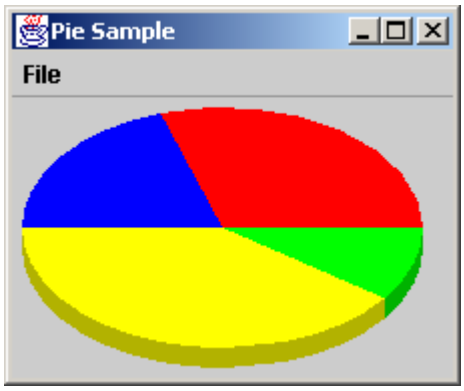

Figure 2: Pie Chart widget

Source code for this demonstration is available in the **`sample/pie/src`** directory of your ULC installation directory. The package structure is as follows:

Package	Description
com.ulcjava.sample.pie.client.widget	The basic PieChart widget.
com.ulcjava.sample.pie.client	Client-side proxy class *UIPieChart*.
com.ulcjava.sample.pie.server	Server-side proxy class *ULCPieChart*.
com.ulcjava.sample.pie	ULC application that demonstrates the *ULCPieChart* widget. The Pie sample can be executed by executing the *Pie* and *PieApplet* classes in your IDE.

3.1 PieChart Widget

The pie chart widget class *PieChart* is implemented as a subclass of *JPanel*. It has an overridden *paintComponent()* method that draws a pie chart on the graphic context as per the given values and colors.

PieChart has the following properties which can be set:

- Data values from which the relative segment size is computed.

- Colors with which the segments are painted.
- Segment labels used to identify each segment.
- Title of the pie chart.

PieChart listens to mouse activity and in response to a mouse click, it generates an *ActionEvent.* The label of the pie segment on which the click occurred is referenced by the *ActionEvent*'s *actionCommand* attribute.

To add this *PieChart* widget to the ULC widget set, create the client- and server-side half objects as described in the following sections. The server-side half object is a ULC widget that offers the same API as the *PieChart* class and thus acts as a server-side proxy to it. In other words, the ULC *PieChart* widget should provide the following:

- Setter/getter methods for data values, colors, labels and title.
- Methods to add and remove event listeners.

3.2 Implementing the Client-Side Classes

The class *UIPieChart* in the package *com.ulcjava.sample.pie.client* is the client-side half object that adapts *PieChart* to the UI Engine. The basic object *PieChart* class is a visual class that indirectly extends *JComponent*. Therefore *UIPieChart* extends *UIComponent* in the client-side half object hierarchy. *UIComponent* itself extends the class *UIProxy*, which forms the root of the client-side half object class hierarchy.

3.2.1 Instantiate the Basic Object

First and foremost the client-side half object instantiates and initializes the basic object. To do this, *UIPieChart* overrides the *createBasicObject(Object[] arguments)* method of *UIProxy*. The latter method creates a *PieChart* object, which is initialized with *width* and *height* values (see 3.3.3 on how these initial values are determined).

```
protected Object createBasicObject(Object[] arguments) {
    int width = ((Integer)arguments[0]).intValue();
    int height = ((Integer)arguments[1]).intValue();
    return new PieChart(width, height);
}
```

UIPieChart implements the following convenience *getter* method to easily access the basic object:

```
public PieChart getBasicPieChart() {
    return (PieChart) getBasicObject();
}
```

3.2.2 Initialize the Proxy and the Basic Object

After creating the basic object – by calling the *createBasicObject(Object[] arguments)* method on the *UIPieChart* – the ULC framework automatically initializes the *UIPieChart* and *PieChart* instances via a sequence of recorded method calls obtained from the server side (see 3.3.3 for details). Note that this is an additional initialization step, which happens after the creation of the basic object via

createBasicObject(Object[] arguments). In many cases, the UI object might require additional initialization before or after the initialization method calls from the server are executed. An example for this is the registration of listeners on the basic object. Such listeners might be necessary in order to propagate corresponding events to the server. To enable this type of initialization, ULC offers suitable callback methods *preInitializeState()* and *postInitializeState()* on *UIProxy*. These methods may be overridden in subclasses of *UIProxy*.

In this example, the *UIPieChart* should add itself as an *ActionListener* to the basic object. This may be done in *preInitializeState()* or in *postInitializeState()*. However, since none of the initialization commands obtained from the server side depend on this listener, it is preferably registered in *postInitializeState()*:

```
public void postInitializeState() {
    super.postInitializeState();

    getBasicPieChart().addActionListener(new PieActionListener());
}
```

It is important that the *super.postInitializeState()* method is called inside the overridden version of *postInitializeState()* because the super class might have to perform further actions in this context. (The same holds for *preInitializeState()*)

3.2.3 Handle Events of the Basic Object

The *PieChart* widget fires *ActionEvents* when a user clicks on a painted sector. *UIPieChart* must capture these events and pass them on to the server-side widget. For this purpose, it creates an instance of *ActionListener* (i.e., *PieActionListener*) and registers it as an event listener to *PieChart* (see 3.2.2).

When the *PieActionListener* receives an *ActionEvent* from the *PieChart* widget, it forwards it to the server-side *ULCPieChart* proxy by calling *fireActionEventULC()*. Therefore, the *PieActionListener* class looks as follows:

```
private class PieActionListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        fireActionPerformedULC(e.getActionCommand(), e.getModifiers());
    }
}
```

Note that *UIProxy* offers *fire...ULC()* methods for all event types, which are directly supported by ULC. If non-standard events types (i.e., other than the common Swing/AWT even types) are required by an extension, then the ULC event mechanism must be extended as well (see Section 5.3 for details).

3.3 Implementing the Server-Side Classes

The server-side widget maintains the state of the basic *PieChart* widget locally on the server and handles events coming from the client. Therefore, it mirrors the basic *PieChart* widget in terms of properties, events, and its API – it acts as a server-side proxy to the basic *PieChart* widget. Since *PieChart* is a visual class, *ULCPieChart* inherits from *ULCComponent* (which in turn extends *ULCProxy*).

ULCPieChart has properties and access methods for *height*, *width*, *data values*, *colors*, *labels* and *title*. In addition, it provides methods to add and remove *IActionListener*s.

3.3.1 Adding Instance Variables and Constructors

The following code presents an initial version of *ULCPieChart* with all properties and constructors added:

```
public class ULCPieChart extends ULCComponent {
    private double[] fValues;
    private String[] fColors
    private String[] fLabels;
    private int fWidth;
    private int fHeight;
    private String fTitle;

    public ULCPieChart() {
        this(100, 100);
    }

    public ULCPieChart(int width, int height) {
        fWidth = width;
        fHeight = height;
        fColors = new String[0];
        fValues = new double[0];
        fLabels = new String[0];
        fTitle = null;
    }

    ... // More to be added here (see text below)
}
```

3.3.2 Specifying the Client-Side Proxy Class

The ULC framework automatically creates the client-side proxy instance based on the type information returned by the callback method *typeString()* defined as abstract in *ULCProxy*. *ULCPieChart* implements this method in order to return the classname of the *UIPieChart* class:

```
public String typeString() {
    return "com.ulcjava.sample.pie.client.UIPieChart";
}
```

Note that the *UIPieChart* class is not available on the server-side class path of ULC and so, the class name must be provided directly as a “hard-coded” textual representation and not by using the *getName()* method of the corresponding class object (e.g. *UIPieChart.class.getName()*).

3.3.3 Uploading State to the Client-Side Proxy

In ULC, a server-side half object may be created and exist a long time before the corresponding client-side half object is eventually created and initialized. E.g., for

graphical widgets, ULC usually delays the creation and initialization of the client-side widget until it becomes visible on the client (i.e., it must be painted). The process behind creating and initializing the client-side widget is referred to as the *upload* of the widget. Due to the mentioned delay between creating the server-side widget and the client-side, the server-side widget might have undergone many state changes before it is uploaded. These state changes must be reconstructed as part of the initialization phase of the *UIPieChart* object and its basic object. This takes place by means of the sequence of remote method calls, which were already mentioned in Section 3.2.1. This sequence is initially generated at the server-side half object and then the remote method calls are sent to client. To generate the correct sequence on the server-side, *ULCPieChart* must override the callback method *uploadStateUI()*. Inside *uploadStateUI()*,certain convenience methods may be called, where each one adds a specific remote method call. Moreover the method *createStateUI()* may also be called. The latter provides the arguments of the *createBasicObject()* method from Section 3.2.1. In the context of *ULCPieChart* the method *uploadStateUI()* is overridden as follows:

```
protected void uploadStateUI() {
   super.uploadStateUI();
   createStateUI(new Object[]{new Integer(fWidth),
                              new Integer(fHeight)});
   setStateUI("title", null, fTitle);
   setCompositeStateUI("data",
                       new Object[]{new String[0], new double[0],
                                    new String[0]},
                       new Object[]{fLabels, fValues, fColors});
}
```

This method performs the following actions:

- It calls *super.uploadStateUI()* to ensure that the super class uploads its state to the client.
- It calls *createStateUI()* in order to pass the required arguments to create the basic object on the client side.
- It calls *setStateUI()* methods to set the corresponding properties on the basic object class *PieChart*. The related command are executed via reflection during the initialization phase of the *UIPieChart* on the client side. (ULC first tries to dispatch a related client-side method call on the *UIProxy* object. If this fails, it tries to dispatch it on the basic object. Since the *UIPieChart* does not provide any corresponding setter methods to dispatch to, all the related method calls are dispatched to the basic object *PieChart*.)
- It calls *setCompositeStateUI()* to set the composite property (labels, values, colors) on the basic object. More specifically, the method *PieChart.setData(String[], int[], Color[])*.

For all *setStateUI()* and *setCompositeStateUI()* methods, a default value can be passed in that is used by the ULC framework to eliminate unnecessary calls to the client side. Section 4.4.2 will describe the details on how remote calls from a *ULCProxy* subclass are dispatched to a *UIProxy* instance and its basic object.

3.3.4 Adding Property Accessors

After the upload of a server-side half object, whenever a property is set on the server-side half object by the ULC application, it also needs to be updated on the client-side half object. When implementing a property accessor, the *setStateUI()* method can be used to transfer the changed state of the server-side proxy to the client-side proxy. As discussed in Section 3.3.3 *setStateUI()* should also be used inside *uploadStateUI()* in order to transfer server state during the upload phase of a widget. In both cases (before and after upload) the server-side *setStateUI()* calls are transformed into setter method calls on the client side. After transferring corresponding requests to the client, ULC dispatches the setter method calls automatically to the *UIProxy* or its basic object.

For instance, *ULCPieChart* implements the *setTitle()* methods as follows:

```
public String getTitle() {
    return fTitle;
}

public void setTitle(String title) {
    fTitle = setStateUI("title", fTitle, title);
}
```

The *ULCPieChart.setData()* method is implemented as follows:

```
public void setData(String[] labels, double[] values, String[] colors) {
    fLabels = labels;
    fValues = values;
    fColors = colors;
    setCompositeStateUI("data", new Object[]{fLabels, fValues, fColors});
}
```

3.3.5 Adding Event Support

ULCPieChart requires *addActionListener()* and *removeActionListener()* methods for registering and unregistering *IActionListener*s, which can react to clicks on the displayed pie chart sectors. *ULCProxy* offers a mechanism for managing listeners and distributing events to them automatically. The method *ULCProxy.addListener()* stores listeners such that events sent from the client-side widget are automatically dispatched to those listeners. *ULCProxy.removeListener()* removes a corresponding listener again. Therefore, the implementations of *addActionListener()* and *removeActionListener()* simply delegate to the *add/removeListener()* methods defined on *ULCProxy*:

```
public void addActionListener(IActionListener actionListener) {
    addListener(UlcEventCategories.ACTION_EVENT_CATEGORY,
                        actionListener);
}

public void removeActionListener(IActionListener actionListener) {
    removeListener(UlcEventCategories.ACTION_EVENT_CATEGORY,
                        actionListener);
}
```

3.4 Using ULCPieChart

The usage of *ULCPieChart* is illustrated by the class *com.ulcjava.sample.pie.Pie*. This class demonstrates how to manipulate properties and handle events of *PieChart.*

First, a *ULCPieChart* widget is instantiated and initialized by the *start()* method of *Pie*:

```
fPieChart = new ULCPieChart(300, 220);
fPieChart.setData(fLabels, fValues, fColors);
fPieChart.setTitle("Pie Chart");
```

To handle an *ActionEvent*, an *IActionListener* is registered on the *ULCPieChart* instance:

```
fPieChart.addActionListener(new ULCPieActionListener());
```

The implementation of *IActionListener* handles *ActionEvents* by incrementing the data value associated with the clicked pie segment:

```
private class ULCPieActionListener implements IActionListener {
    public void actionPerformed(ActionEvent event) {
        int i = findValueIndex(event.getActionCommand());
        fValues[i] = fValues[i] + 1.0;
        fValueFields[i].setValue(new Double(fValues[i]));
        fPieChart.setData(fLabels, fValues, fColors);
    }
}
```

4 Extension Infrastructure Details

After presenting an introductory example in the previous section, this section explains in more details how to add properties and events to a ULC widget.

4.1 Introduction

The root of the server-side half object inheritance hierarchy is the *ULCProxy* class, whereas the root of the client-side half object inheritance hierarchy is the *UIProxy* class. ULC objects such as formatters, enablers, polling timer, models, and model adapters inherit directly from *ULCProxy*. In general, server-side half objects, which do not have a visual representation on the client side extend *ULCProxy*.

The server-side half object class hierarchy for visual components starts with the class *ULCComponent* (a subclass of *ULCProxy*) whereas the corresponding client-side hierarchy for visual components starts with the class *UIComponent* (a subclass of *UIProxy*). E.g., the server-side half object classes *ULCTable*, *ULCButton*, *ULCLabel*, *ULCTree* have a visual representations on the client side and inherit from *ULCComponent*.

These root classes of the proxy hierarchy are abstract classes and provide many auxiliary and convenience methods required inside their subclasses. All *ULCProxy* subclasses have their counterpart *UIProxy* classes on the client side. The *UIProxy* subclasses do not perform the GUI rendering themselves but hold onto and control the basic object classes (usually (sub)classes of the Java Swing API). The basic object classes are instantiated via the *UIProxy.createBasicObject()* method.

Note that there is always a one-to-one relationship between the server-side half object class, the client-side half object class and the basic object class. The same holds for instances of these associated classes. For example, a server-side half object of type *ULCButton* is associated with a client-side half object of type *UIButton* and the latter references an instance of *BasicButton* – a subclass of *JButton*.

4.2 Extension API Methods

The following paragraphs discuss important methods on the class *ULCProxy* and *UIProxy*, which are relevant when writing an extension.

4.2.1 ULCProxy

The following table provides a description of important methods on the *ULCProxy* class:

Method(s)	Description
String typeString()	Returns the fully qualified class name of the client-side half object class associated with the implemented server-side half object class. ULC uses the name of the client-side half object in order to instantiate a corresponding client half object during the upload of a ULC widget object.
void uploadStateUI()	This method should be overridden in order to initialize the state of a newly created client-side half object during the upload of a ULC widget (see also Section 3.3.3
createStateUI(Object[])	This method may be called inside an overriding version of *uploadStateUI()* in order to supply the constructor of the basic object with suitable arguments. The construction of the basic object happens on client side at the beginning of the upload phase. The argument object array from *createStateUI()* will eventually be passed in as an argument to *UIProxy.createBasicObject()* when constructing the basic object.
setStateUI(String, ...) *setCompositeStateUI(String, ...)* *addStateUI(String, ...)* *addCompositeStateUI(String, ...)* *removeStateUI(String, ...)* *removeCompositeStateUI(String, ...)*	These methods should be called in order send state changes of properties from the server-side half object to the associated client-side half object (or the basic object). The first argument of these methods is the name of the property whose state change should be forwarded to the client. Depending on the signature, some of these methods first check, whether a state change really occurred, and only then they forward that change to the server. On the client side, the state change is performed by calling a setter method on the client-side half object or on the associated basic object. E.g., if the property name *"value"* is passed in as the first argument to *setStateUI()*, then ULC will potentially try to invoke a setter method called *setValue()* on the client-side. ULC first tries to find a corresponding matching setter method on the client-side half object. If no matching method is found there, it searches on the basic object. Eventually the setter method is invoked with a

	(potentially converted) value which was provided via *setStateUI()*. Typically these methods are used to send server state inside *uploadStateUI()* or inside setter methods of a server-side half object class.
void invokeUI(String, Object[]) void invokeUI(String)	This method records a method call on the server-side half object, which will eventually be executed on the associated client-side half object (or the related basic object). The first argument is the name of the method to be invoked on the client. The second argument, if existing, is an object array, which provides the arguments for the resulting client-side method invocation. If there is no second argument, the referenced method is assumed to be a no-argument method. The rules for finding a matching method on the client side are the same as for *setStateUI()*. Typically *invokeUI()* is used inside *uploadStateUI()* or in public API methods of a server-side half object class. ULC records invocations of *invokeUI()*, *setStateUI()* etc. during a server round trip and dispatches the corresponding client-side method calls when a resulting response reaches the client. ULC guarantees that the order of the recorded invocations is maintained locally on the associated client-side half object. The recording only happens during the execution of *uploadStateUI()* or after the widget was uploaded. *invokeUI()* calls that happen prior to these occasions are ignored.
void addListeners(String, EventListener) *void removeListeners(String EventListener)*	These methods allow for registering / unregistering listeners associated with a particular event category. The event category is the first argument and the listener to be registered / unregistered is the second argument for these methods. An event category is a name that is associated with a certain listener interface. The category name enables the grouping of listeners which all have to be notified when a particular type of event occurs. All listeners registered for an event category must implement the listener interface that the category is associated with. ULC comes with a standard set of event categories and associated listener interfaces. The respective category names are defined in the class *UlcEventCategories*.

EventListener[] getListeners(String)	Returns an array with all listeners registered at an event category. The event category is the first argument of the method.
void setEventDeliveryMode(String, int)	This method sets the delivery mode (second argument) for an event category (the first argument). The delivery mode determines how the client behaves upon sending an event, which will be delivered to listeners registered at the event category. E.g. “synchronous” delivery mode ensures that the user interface is blocked until the server has processed the event and has sent a response to the client. (The class *ClientContext* defines constants for three different delivery modes.)
IDispatcher createDispatcher()	ULC allows calling methods of the server-side half object from the associated client-side half object. This feature is similar to the one provided by *invokeUI()*. However, *invokeUI()* aims at the server to client communication. To enable the invocation of protected and private methods on a server-side half object class from the associated client-side half object class, the related methods must be made accessible via a special dispatcher class. The dispatcher class comes as a non-static inner class of a server-side half object class and provides a set of public methods that may be invoked from the client. The latter methods usually delegate to protected or private methods of the outer server-side half object class. In short, a dispatcher class opens non-public methods to a client-side half object associated with a server-side half object without allowing other server-side objects to access these non-public methods. The method *createDispatcher()* returns an instance of the dispatcher class associated with the server-side half object class. Subclasses of *ULCProxy* should override this method in order to return an instance of the (unique) dispatcher class that belongs to them.
void upload()	Usually proxies are uploaded when their “parent” is uploaded. Therefore the extension developer does not have to invoke *upload()* in his code. However, for proxies that do not have such a “parent” the extension developer is responsible for uploading the proxy to the client side. (See also

	Section 3.3.3 for information about the upload of widgets.)
boolean isUploaded()	The method tells whether or not the server-side half object has already been uploaded.
void markUncollectable()	In ULC, a newly created server-side half object gets registered with the associated session right before its upload. However, the registered server-side half object may still be garbage collected by the Java virtual machine (ULC uses Java's weak reference concept for this). Unfortunately a server-side half object might still be needed, although it can be garbage collected, because the associated client-side half object might still be in use. To avoid this, this method prevents a server-side half object from ever being garbage collected. (Note that in most cases server side half object do not need to marked as uncollectable because they are transitively referenced by uncollectable parent or ancestor widgets. E.g. uploaded *ULCWindow* objects are by default marked as uncollectable.)

4.2.2 UIProxy

The following table provides a description of important methods on the *UIProxy* class:

Method(s)	**Description**
Object createBasicObject(Object[])	This method creates the basic object that is associated with a client-side half object (see also Section 3.3.3). Extension writers should override this method in case they want to integrate their custom basic object in ULC.
void preInitializeState() *void postInitializeState()*	As explained in Section 3.3.3 these two methods partially initialize the state of a client-side half object during the upload of the widget. *preInitializeState()* is called after the basic object associated with the client-side half object has been created. ULC performs this callback right after invoking *createBasicObject()*. After calling *preInitializeState()* ULC dispatches method calls, which have been recorded on the server side inside *uploadStateUI()*. Finally it calls *postInitializeState()* which completes the widget upload.

	Extension writers may override *preInitializeState()* and *postInitializeState()* in order to perform initializations of the client-side half object, which go beyond creating the basic object and dispatching method calls recorded on the server side via *uploadStateUI()*.
boolean isInitializingState()	This method tells whether or not the client side half object is in the phase of the upload when recorded method calls are being dispatched.
Object getBasicObject()	Returns the basic object.
void fireActionPerformedULC(String, int) void fireSelectionChangedULC(String, int) ...	These methods fire events, which will be propagated to the associated server-side half object and dispatched to listeners registered for a certain event category. E.g., *fireActionPerformedULC()* sends an event to server-side listeners, which implement the *IActionListener* interface and which are registered for the event category named *"action"*. However, if no listeners are registered on the server-side, then the event will not be sent at all. The first argument of *fireActionPerformedULC()* represents the *command* property of the resulting server-side action event and the second argument forms the *modifiers* property. The other *fire...ULC()* method follow a similar pattern.
void fireEventULC(String, ...)	This method allows for firing custom events to custom event categories and listener interfaces (see Section 5.3 for details). An event is only sent to the server if there are listeners registered at the event category on the server-side half object.
void fireMandatoryEventULC(String, ...)	This method behaves similarly to *fireEventULC()* but it *always* sends the resulting event (regardless of whether listeners are registered on the server side).
updateStateULC(String, ...)	The purpose of *updateStateULC()* methods is similar to the one of the *setStateUI()* methods on *ULCProxy*. However, *updateStateULC()* forwards state changes of properties from the client-side half object to the associated server-side half object. The first argument of the *updateStateULC()* methods is the name of the property to be updated on the server-side half object. The second argument is the

	update value. To perform the update on the server-side half object, an extension writer must provide a public update method on the server-side half object class or on the associated dispatcher class. (See also the description of the *createDispatcher()* method.) The name of the method is *update<PropertyName>()* (e.g. *updateValue()* if the property name is *"value"*) and it should have one parameter for the update value. Every *updateStateULC()* invocation triggers a separate client server roundtrip given that it is called outside the context of a *flushDirtyData()* method call (see sub-section Dirty Data Owner in section 3.4.3 for *flushDirtyData()*). When called inside *flushDirtyData(),* a sequence of *updateStateULC()* calls is sent via a single client-server roundtrip. If *updateStateULC()* triggers a separate client server roundtrip then the delivery mode is asynchronous, which means that the GUI is not blocked during the roundtrip.
invokeULC(String, Object[]) *invokeULC(String)*	The purpose of *invokeULC()* is similar to *ULCProxy.invokeUI() but in the direction client to server*. This method records a method call on the client-side half object, which will eventually be executed on the associated server-side half object (or the related basic object). The first argument is the name of the method to be invoked on the server. The second argument, if existing, is an object array, which provides the arguments for the resulting server-side method invocation. If there is no second argument, the referenced method is assumed to be a no-argument method. The rules for finding a matching method on the server side are the same as those for *updateStateULC()* i.e., there should be a public method by that name in the server-side half object or its associated dispatcher class. *updateStateULCinvokeULC()* causes a separate server roundtrip in the same cases as *updateStateULC()*.
invokeULC(int, String, Object[])	The method behaves similarly to *invokeULC(String, Object[])* but via the first argument one can determine the event delivery

	mode in case the method is called outside a *flushDirtyData()* context. The event delivery mode can be one of *UlcEventConstants.SYNCHRONOUS*, *UlcEventConstants.ASYNCHRONOUS*, or *UlcEventConstants.DEFERRED*. The default mode (when the first argument is not specified) is *UlcEventConstants.ASYNCHRONOUS*.
sendMarkCollectable()	This method is related to *ULCProxy.markUncollectable()*. It allows for indicating that a widget which was formerly marked as uncollectable is now collectable due to changes on the client-side.

4.3 Upload and Half Objects

This section details the upload process of a server-side half object. In addition, it gives general hints on how to implement the major classes of an extension.

As explained in Section 3.3.3, a server-side half object may be created and exist a long time before the corresponding client-side half object is eventually created and initialized. Due to the delay between creating the server-side half object and the client-side half object, the server-side half object might have undergone many state changes before it is uploaded. These state changes must be reconstructed as part of the initialization phase of the client-side half object and its basic object. The information on how to perform the reconstruction is given on the server side by an overridden version *uploadStateUI()*.

Figure 3 presents a UML sequence diagram to illustrate the order of method calls in the context of an upload:

1. A server-side half object (here an instance of *ULCWidget*) is constructed by the server-side application code.
2. There may be a sequence of state changing method calls on the server-side object until the upload phase of the object begins. The upload may be triggered explicitly via *ULCProxy.upload()* or implicitly because ULC requires the associated client-side half object to perform operations on it.
3. To prepare the upload, ULC calls *ULCProxy.uploadStateUI()*. An overridden version of this method may invoke *invokeUI()*, *setStateUI()* and related methods to record a sequence of remote method calls (see also Section 4.4 for details on remote method calls in ULC). Besides, *createStateUI()* may be used to provide construction arguments for creating the basic object on the client-side.
4. When the upload happens, the information collected via *uploadStateUI()* is sent to the client. On the client-side ULC performs a set of operations to create and initialize the client-side half object and the basic object (see following steps).
5. The client-side half object is created via Java reflection. To do so, ULC instantiates an instance of the client-side class, whose class name is given by (an overriden version of) *ULCProxy.typeString()*. For this purpose, the related class

must be accessible via the client-side classpath and the class must have a public no-argument constructor.

6. ULC calls *UIProxy.createBasicObject()* and supplies it with the arguments, which have been passed in to the *createStateUI()* call from Step 3. The related arguments are subject to the marshalling conversion process, which will be explained Section 4.4.4.
7. ULC invokes *UIProxy.preIntializeState()* and then, it dispatches the remote method calls, which were recorded in Step 3. Afterwards it invokes *UIProxy.postIntializeState()*. This completes the upload phase of the server-side half object.
8. After the upload, all invocations of *ULCProxy.invokeUI()* (e.g., from within public API methods of the server-side half object) are recorded in the order, in which they happen on the server during the server round trip. The recorded remote method calls get dispatched in the same order, when the corresponding response arrives on the client side.

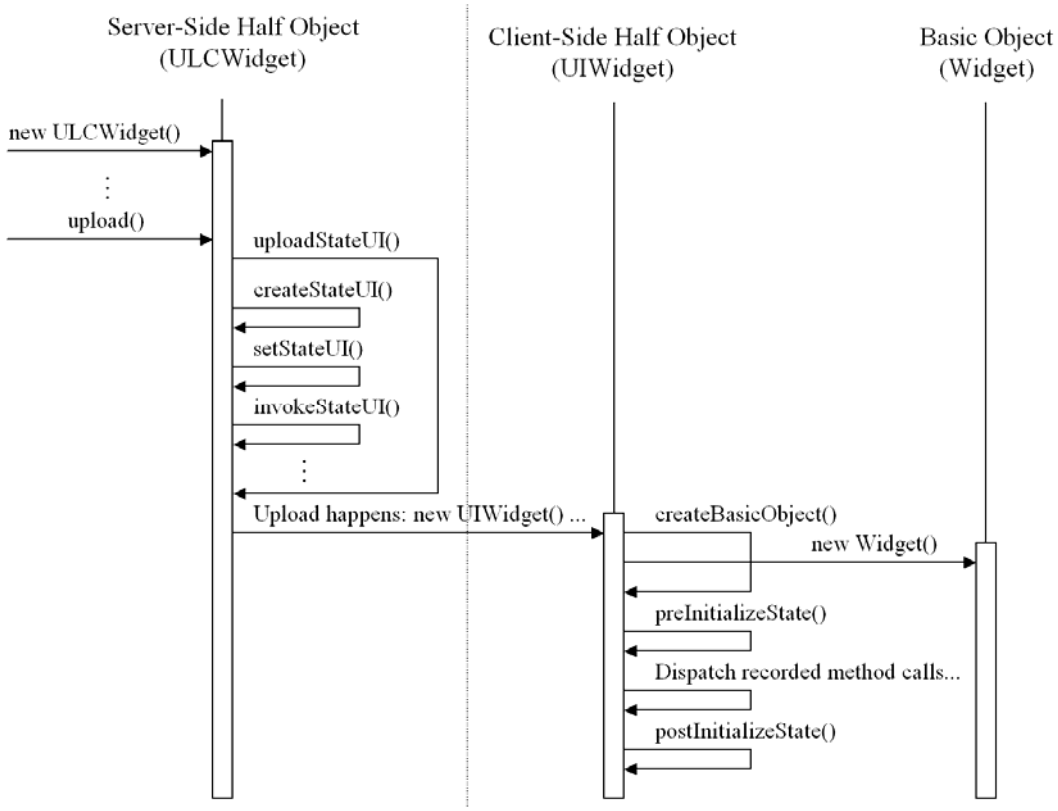

Figure 3: UML Sequence Diagram of the Upload Process

The following code serves as template for overriding *ULCProxy.uploadStateUI()*:

```
protected void uploadStateUI() {
      // Call super method here so that the super class of your
      // extension can add its own initialization calls.
      super.uploadStateUI();
      // Use the following method call to pass in the arguments used for
      // the client-side createBasicObject() call. Calling createStateUI()
      // is optional. If it is missing, the arguments provided by
      // super.uploadStateUI() are used in createBasicObject().
      // (Otherwise, the ones from this createStateUI() call are used.)
      createStateUI(new Object[]{...});
      // Record the remote method calls for the client-side
      // half object initialization:
      setStateUI(...);
      invokeUI(...);
      setCompositeStateUI(...);
      ...
}
```

Extensions should stick to the following rules regarding the naming and placing of classes involved in the half object pattern:

- Extension classes may be placed in one or more separate packages but should not be placed in any of the existing ULC Java packages.
- Client-side half object classes should be prefixed with “UI” and should be public. As mentioned in Step 5, they must have a publicly accessible default constructor. The client-side half object classes must be deployed on the client side and must be accessible via the classpath of the UI Engine.
- Server-side half object classes should be prefixed with “ULC” and must be deployed on the server side.

4.4 Communication

This section covers all aspects of client server communication via remote method calls. Using the methods *ULCProxy.invokeUI()* and *UIProxy.invokeULC()* an extension writer obtains a way to perform remote method calls on a corresponding associated half object. Section 4.4.1 discusses in detail, how ULC tries to dispatch such method calls in the context of the half object pattern. At first, we explain how methods of a remote class are uniquely resolved when addressing them via the *invoke...()* methods. These resolving rules are identical for client server as well as for server client communication.

Section 4.4.2 explains, on what remote classes ULC searches for dispatchable methods. In this context, we also detail on the concept of server-side dispatcher classes.

4.4.1 Addressing a Remote Method

The methods *ULCProxy.invokeUI()* and *UIProxy.invokeULC()* offer a way to perform remote method calls on a corresponding associated half object. For convenience, a remote method is simply referenced via its name and by the invocation arguments, which may be passed in as an object array to *invokeUI()* and *invokeULC()* respectively. Unfortunately, in the context of overloading, addressing a remote method this way might be ambiguous. As a general rule for ULC, an extension writer has to ensure that such ambiguity does not occur for methods, which are called remotely. When trying to

find a method, to which a remote call should be dispatched, ULC checks for such ambigous cases and throws an exception in case the addressed method is not unique.

To understand the problem, consider the following client-side class, to which method calls should be dispatched via *ULCProxy.invokeUI()*:

```
class MethodAddressingSample {
   public int m1(int i) {}
   public boolean m1(Double d) {}
   public int m2(String s) {}
   public boolean m2(Long l) {}
}
```

The server-side call *invokeUI("m1", new Object[] {new Integer(1)})* should address the method *m1(int)* because the argument's type *Integer* wraps *int*, and the *int* value *1* is assignable to the (only) parameter of *m1(int)*. Moreover, *1* is not assignable without conversion to the parameter type *Double* of *m1(Double)*.

The server-side call *invokeUI("m1", new Object[]{null})* addresses *m1(Double)*, because *null* is only assignable to *Double* but not to *int*.

Now consider *invokeUI("m2", new Object[]{null})*: In this case, it is not clear, whether ULC should prefer dispatching to *m2(String)* or rather to *m2(Object)* because *null* is assignable to both parameter types. Due to this ambiguity, ULC throws an exception when trying to dispatch a related method call on the client side.

ULC is even more restrictive and prohibits any remote invocations of *m2(String)* and *m2(Object)*. E.g., ULC does not dispatch the remote method call incurred by *invokeUI("m2", new Object[]{"hello"})*! The rationale behinds this restriction is that if the dispatch to a certain method works for *some* argument combination it should work for all possible argument combinations which are assignable to that method's parameters.

Since *null* is always assignable to parameters with non-primitive types, the *null* value often causes ambiguity when trying to dispatch a remote method calls. E.g., if there are two methods on a remote class with the same name and the same number of parameters, and if all associated parameter types are non-primitive, then remote calls addressing these methods are never dispatchable.

The following paragraph specifies in detail, when a method addressed via *invoke...()* is considered unique by ULC, so that a related remote method call is dispatchable.

Let m be the name of the referenced method and $a_1, \ldots, a_n$ be the arguments of the remote method call. Further, let C be the class on which the method call should be dispatched. Let M be the set of all public methods on class C with the same name m and with n parameters.

ULC can dispatch the related call, if the following conditions hold:

1. M contains one or more elements.
2. Let m_1 and m_2 be any two different methods in M with the signatures $m_1(p_{1,1}, ..., p_{n,1})$ and $m_2(p_{1,2}, ..., p_{n,2})$. Then, there must be at least one position k in the parameter lists of m_1 and m_2 such that either

a. $p_{k,1}$ and $p_{k,2}$ are different primitive types or

b. $p_{k,1}$ is a primitive type and $p_{k,2}$ is not or

c. $p_{k,2}$ is a primitive type and $p_{k,1}$ is not.

3. The arguments $a_1, \ldots, a_n$ are assignable to the parameter types $p_{k,1}, \ldots, p_{k,n}$ with respect to (exactly) one method m_k in M. Hereby, a_i is assignable to $p_{k,i}$ if and only if

   a. $p_{k,i}$*.class.isAssignableFrom(*a_i*.getClass())* holds or

   b. $p_{k,i}$ is not primitive and a_i == *null* or

   c. a_i*.getClass()* is a primitive wrapper type (e.g. *java.lang.Integer()*) and $p_{k,i}$ is the matching primitive type (e.g. *int*)

   If (and only if) these conditions hold, then the remote method call is dispatchable to m_k.

Note that in addition to these rules arguments sent from the client to the server or vice versa may undergo a conversion before the dispatch. The details of this conversion process will be explained in Section 4.4.4.

4.4.2 Dispatching on Remote Classes

From Server to Client

In the server client direction, a remote method call is carried out by *invokeUI ()* on the server side half object. When trying to dispatch the related remote method call on the client side, ULC first searches for a dispatch method on the class of the associated client side half object according to the rules from Section 4.4.1.

- If a unique dispatch method is found, then ULC calls it with the given arguments via Java reflection and afterwards, the dispatch for this remote method call ends. (A potential return value of the invocation is ignored.)
- If a dispatch method is found, but it is not unique, then ULC throws a runtime exception.
- If no dispatch method is found, then ULC applies the same dispatch procedure on the class of the associated basic object. If no dispatch method is found there, then ULC cannot dispatch the remote method call and throws a runtime exception.

  As will be detailed in Section 4.4.4, ULC is capable of marshalling server-side half object references to the client. In this process, the server-side half object reference is converted into a client-side half object reference of the associated client-side half object. Server-side half object references may also be contained as an argument in the argument list array, which is passed in to *invokeUI()*. In this case, ULC first converts a related argument into the reference of a basic object before it tries to dispatch a remote method call on the basic object. This behaviour will detailed at the end of Section 4.4.4.

From Client to Server

In the client server direction, a remote method call is carried out by *invokeULC()* on the server side half object. In this direction it might be desirable to invoke protected or

private methods on the class of the server side half object. Unfortunately this is not possible without potentially corrupting some security restrictions of Java. To bypass these restrictions in a clean manner, ULC offers the concept of dispatcher classes.

A dispatcher class is a non-static inner class of a server side half object class. A dispatcher class should have a set of public methods that may be invoked from the client. It is recommended that these public methods delegate to private or protected methods from the outer class (which is the server side half object class) in order to make these non-public methods accessible for remote calls from the client side.

The method *ULCProxy.createDispatcher()* returns an instance of the dispatcher class associated with the server-side half object class. Subclasses of *ULCProxy* should override this method in order to return an instance of the (unique) dispatcher class that belongs to them.

When trying to dispatch a remote method call on the server side, ULC first creates a local instance of the dispatcher class using *createDispatcher()*. Then, it searches for a dispatch method on the class of the created dispatcher object according the rules from Section 4.4.1.

- If a unique dispatch method is found, then ULC calls it with the given arguments via Java reflection and dispatching for this remote method call ends. (A potential return value of the invocation is ignored.)
- If a dispatch method is found, but it is not unique, then ULC throws a runtime exception.
- If no dispatch method is found, then ULC applies the same dispatch procedure on the server-side half object. If no dispatch method is found there, then ULC cannot dispatch the remote method call and throws a runtime exception.

Implementation Rules for Dispatcher Classes

Custom dispatcher classes should adhere to the following implementation rules:

- A dispatcher class should be a protected non-static inner class of your server-side half object class.
- If *ULCXyz* is the name of your server-side half object class, then, the name of a dispatcher class should ideally be *ULCXyzDispatcher*.
- To associate a dispatcher class with the outer server-side class you must override the method *ULCProxy.createDispatcher()* such that it returns an instance of the custom dispatcher class.
- A dispatcher class should always inherit from the dispatcher class of the super class of the outer server-side half object class. This ensures that methods, which are dispatchable in the super class of server side half object class, remain dispatchable in the subclass. E.g., if you write a (faceless) extension by extending *ULCProxy*, and then provide a dispatcher class for your extension, it should inherit from *ULCProxy.ULCProxyDispatcher*.
- Ideally, a dispatcher consists of a set of public final methods that merely delegate to private or protected methods of the outer class. Thus, the signatures of the delegating methods and the referenced outer class methods should be identical.

Example of a Dispatcher Class

The following code extract demonstrates, how to best implement a dispatcher class for a server-side half object class which extends *ULCProxy*:

```
public class ULCMyExtension extends ULCProxy {
    ...
    public IDispatcher createDispatcher() {
        return new ULCMyExtensionDispatcher();
    }

    private void myMethod(int something) {
        ...
    }
    ...
    protected class ULCMyExtensionDispatcher() extends ULCProxyDispatcher {
        public final void myMethod(int something) {
            ULCMyExtension.this.myMethod(something);
        }
    }
}
```

4.4.3 Convenience Methods Based on the *invoke…()* Methods

ULC offers a set of convenience methods on the classes *ULCProxy* and *UIProxy*. These methods assist in synchronizing state between client and server, and in essence, they are based on *invokeULC()* and *invokeUI()*. The following paragraphs characterize the behaviour of some of these convenience methods.

From Server to Client

The purpose of the server-side convenience methods is mostly to propagate property changes from a server-side half object to the associated client-side half object.

Method(s)	Description
<type> setStateUI(String propertyName, <type> value)	This type of methods exists for all primitive types, for *String* and for *Object* (so *<type> = int*, *short*, …, *Object*). The methods return the value that was passed in as *value*. Other than that they are (essentially) equivalent to *invokeUI("set<PropertyName>", new Object[] {value})*. If *value* has a primitive type, then it is first converted into the corresponding Java wrapper type. *<PropertyName>* stands for the capitalized version of the string, which is passed in a *propertyName*. E.g., if the first argument of *setStateUI()* is *"myProperty"*, then *<ProperytName>* stands for *MyProperty*.
<type> setStateUI(String propertyName, <type> oldValue, <type> newValue)	These methods are very similar to *<type> setStateUI(String propertyName, <type> value)*. The only (major) difference is that they only delegate to *invokeUI()* if *oldValue* and *newValue* are not equal. They always return *newValue*.

There exist further convenience methods on *ULCProxy* such as *addStateUI()* and *removeStateUI()*, which are also based on *invokeUI()*. Please refer to the ULC API Documentation for details on them.

Examples of applying *setStateUI()* have already been presented in Section 3.3.3 and 3.3.4. The following steps describe, how to add a property to a server-side half object class *ULCWidget*, where changes on the property should always be propagated to the associated client-side half object. The example assumes that the same property is also defined on the client-side half object class *UIWidget* or on the associated basic object class.

1. Add the property as an instance variable in *ULCWidget* with a getter method. Initialize the property to a default value, which matches the default of the property on the client side half object.

   ```
   private float fWeight = 0.5f;
   ```

2. In *ULCWidget*'s *uploadStateUI()* method, use the *setStateUI()* convenience methods to synchronize the property to the client side. Pass in the default and the current value:

   ```
   protected void updateStateUI() {
       ...
       setStateUI("weight", 0.5f, fWeight);
   ```

3. Add a setter method that updates the server-side instance variable and uses the *setStateUI()* convenience methods to synchronize the property to the client side. Pass in the old and the new value.

```
public void setWeight(float weight) {
    fWeight = setStateUI("weight", fWeight, weight);
}
```

4. Either on *UIWidget* or *Widget*, the following public method must exist since it will be invoked by the ULC framework whenever the server side property changes:

```
public void setWeight(float weight) {
    ...
}
```

From Client to Server

The client-side convenience methods assist in propagating property changes from the client-side half object to the associated server-side half object. The related *updateStateULC()* methods have already been mentioned in Section 4.2.2.

Method(s)	**Description**
<type> updateStateULC(String propertyName, <type> value)	This type of methods exists for all primitive types, for *String* and for *Object* (so *<type> = int*, *short*, …, *Object*). The methods return the value that was passed in as *value*. Other than that, they are (essentially) equivalent to *invokeULC(update<PropertyName>, new Object[] {value})*. If *value* has a primitive type, then it is first converted into the corresponding Java wrapper type. *<PropertyName>* stands for the capitalized version of the string, which is passed in a *propertyName*. E.g., if the first argument of *updateStateULC()* is *"myProperty"*, then *<PropertyName>* stands for *MyProperty*.

Dirty Data Owner - Updating the server-side half object

When a client-side proxy needs to update the state of its corresponding server-side half object in the next round-trip, it can do so using an implementation of *IDirtyDataOwner* interface. Such a situation arises when there is state change on the client component due to a user interaction like window resizing, button selection state, enabled state, etc. The client-side proxy needs to register an object of the class implementing *IDirtyDataOwner* interface with the *UISession* using the method *getSession().addDirtyDataOwner().* During the next round-trip, before sending other requests, *flushDirtyData()* is called on

all *IDirtyDataOwners*, wherein they can update the state of the server-side widget using *updateStateULC("<attributeName>", getBasicComponent().get<AttributeName>())*.

The *updateStateULC()* calls are handled and dispatched on the server-side component by its *ULCProxyDispatcher*. After a round-trip all *IDirtyDataOwner*s are removed from the *UISession*. Therefore, the client-side proxy must register an *IDirtyDataOwner* with the *UISession* every time its state changes, i.e. every time it becomes "dirty".

It should be noted that the client-side proxy should not mark itself as dirty, i.e. it should not add a *IDirtyDataOwner* to *UISession*, when its state is being changed as a result of a state change on the server-side proxy. A client-side proxy's state gets synchronized with its server-side proxy while the client is processing requests (responses) from the server, which can be identified by the condition: *getSession().isHandlingRequest()*. In such a situation the client-side state will be the same as the server-side state and hence no synchronization is needed from client to server.

The following code shows how *ULCSlide*r's client-side proxy synchronizes its value to the server-side proxy.

```
// Client side class
public class UISlider extends UIComponent
     implements ChangeListener, IDirtyDataOwner, IUserInteractionListener {
   ...
   // flush the value of the slider to the server side proxy
   public void flushDirtyData() {
      updateStateULC("value", getBasicSlider().getValue());
   }
   // method fired when value on the slider changes
   public void stateChanged(ChangeEvent e) {
      // not a state change on server
      if (!getSession().isHandlingRequest()) {
          // Add self as dirty data owner
         getSession().addDirtyDataOwner(this);

         // send the value changed event to server
         fireValueChangedULC();
      }
   }
 }

 // Server side class
 public class ULCSlider extends ULCComponent {
   ...
   protected void updateValue(int value) {
       fValue = value;
   }
   protected class ULCSliderDispatcher extends ULCComponentDispatcher {
      public final void updateValue(int value) {
          ULCSlider.this.updateValue(value);
       }
   }
}
```

4.4.4 Marshalling

When performing remote method calls via *invokeUI()* and *invokeULC()*, the related method arguments must be transferred from the server to the client or vice versa. To do this, ULC provides a special serialization infrastructure, which is capable of writing the state of objects to an output stream and reading them back in from an input stream.[1] The mechanism is called the *ULC serialization intrastructure*.

The following paragraphs describe, how the ULC serialization infrastructure marshalls data objects and half objects. Besides, they explain consequences for remote method calls performed via *ULCProxy.invokeUI()* and *UIProxy.invokeULC()*.

Serializable Classes and Type Conversions

By default ULC comes with a broad but also extendable set of classes whose instances can be marshalled. If you want to make additional, custom classes to be serializable by ULC, you will have to implement and add a few classes to the infrastructure (see Section 5.4 for details).

The ULC serialization infrastructure is capable of changing the class type of an object, which is serialized and afterwards deserialized. This feature is necessary because ULC provides its own set of server-side data classes, which correspond to Swing or AWT classes on the client side. E.g., the client-side AWT class *java.awt.Dimension* corresponds to the server-side ULC class *com.ulcjava.base.application.util.Dimension*. Therefore, when a server side dimension object of type *com.ulcjava.base.application.util.Dimension* is serialized and marshalled to the client, the ULC serialization infrastructure deserializes it on the client by creating and returning a corresponding *java.awt.Dimension* instance. From client to server, the resulting type conversion of a dimension object happens just the other way around.

[1] Note that ULC does not rely on Java object serialization when transferring objects, because Java object serialization consumes too much network band width and potentially causes security problems in ULC's runtime environment. Java object serialization does not support type changes well for serialized objects, which is an important feature for ULC.

The following table shows all important classes, whose objects can be serialized by default. It also states the classes, whose objects undergo a conversion when being marshalled such as in the case of dimension objects.

Server-side Class	**Client-side class**
All primitive Java types	same
Array types with component types java.lang.Object	same
Array types with component types which implement com.ulcjava.base.server.IProxy *(i.e. server-side half objects)*	Array types with component type *java.lang.Object*
Array types with component types which are serializable with ULC	Array type of potentially converted component type
java.lang.Short java.lang.Integer java.lang.Long java.lang.Float java.lang.Double java.lang.Boolean java.lang.Character java.lang.Byte java.lang.String	Same
java.util.Date java.util.Locale java.util.HashMap java.util.LinkedList java.util.ArrayList java.sql.Date java.sql.Timestamp java.sql.Time	same
java.math.BigDecimal	same
com.ulcjava.base.shared.internal.Anything	same
com.ulcjava.base.application.util.AffineTransform com.ulcjava.base.application.util.Color	*java.awt.geom.AffineTransform* *java.awt.Color*

com.ulcjava.base.application.uti.Dimension	*java.awt.Dimension*
com.ulcjava.base.application.util.Font	*java.awt.Font*
com.ulcjava.base.application.uti.Insets	*java.awt.Insets*
com.ulcjava.base.application.uti.KeyStroke	*javax.swing.KeyStroke*
com.ulcjava.base.application.uti.Point	*java.awt.Point*
com.ulcjava.base.application.uti.Rectangle	*java.awt.Rectangle*

It is important to note that the ULC serialization infrastructure converts objects even if they are contained in other objects, which will actually not be converted when marshalled. For example, if an instance of *java.util.HashMap* exists on the server side and it contains a key value pair of type *String* and *com.ulcjava.base.application.util.Color*, then the marshalled client-side version of the hash map instance has still got the type *java.util.HashMap*, but the contained key value pair has the types *String* and *java.awt.Color*, respectively.

Note that the ULC serialization infrastructure cannot deal with cyclic references inside data objects that should be marshalled. E.g. consider the object *cyclicObject* which contains a cyclic reference, created by the following lines of code:

```
Object[] cyclicObject = new Object[1];
cyclicObject[0] = cyclicObject;
```

When ULC tries to serialize this *cyclicObject* it will throw an error. However ULC *can* deal with cyclic reference with regard to half objects.

Marshalling enums

Since *enums* are *Serializable*, you can use the inbuilt *SerializationCoder* for marshalling *enum* types:

1. Ensure that the *enum* type is present on both the client and ther server.
2. Register the *SerializationCoder* for your *enum* type with the *ICoderRegistryProvider* on the client and server in the *ULCApplicationConfiguration.xml* file:

```
<ulc:coders>
       <ulc:symmetricCoder>
          <ulc:class>
             EnumType
          </ulc:class>
          <ulc:coderClassName>
             com.ulcjava.base.shared.streamcoder.SerializationCoder
          </ulc:coderClassName>

       </ulc:symmetricCoder>

       …
```

Marshalling Half Objects

The ULC serialization infrastructure is also capable of marshalling references to half objects from the server to the client side. When a server-side half object reference is marshalled to the client for the first time, the marshalling also triggers the upload of the server-side half object such as described in Section 3.3.3. The marshalling process converts the server-side half object reference into a client-side half object reference of the associated client-side half object. This implies a type change of the reference during

marshalling. E.g., when marshalling a server-side reference of a *ULCButton* instance, it becomes a client-side reference of the associated *UIButton* half object instance.

The marshalling process converts a server-side half object references into a client-side half object references, even if the related server-side reference is nested in another marshalled container data object such as a map, a list or an array. However, you must ensure that your container class is one of the following: *ArrayList*, *LinkedList*, *Vector*, *HashSet*, *TreeSet*, *HashMap* or *TreeMap*. It should be possible to instantiate your container class by reflection, i.e., it should be *public* and should have a *public* default constructor. In case you have a custom container class, you have to provide a *IStreamCoder* for it.

Note that no such conversion takes place when client-side half object references should be marshalled to the server. In this case, ULC does not resolve the corresponding reference but throws a client-side exception instead.

Effects on Remote Method Calls

The above discussed conversion of data objects and half object references changes the list of arguments, which is used on the remote side, when dispatching a remote method call. E.g., when calling

```
ULCProxy.invokeUI("m",
    new Object[] {new com.ulcjava.base.application.util.Point(0,0)});
```

on the server-side, then the client-side argument list for invoking *m()* contains a single instance of *java.awt.Point*. Therefore, ULC can only dispatch the remote method call, if *java.awt.Point* is assignable to the (only) parameter type of *m()*. E.g. *public m(java.awt.Point)* is a suitable signature for a method, to which the given remote method call should be dispatched on the client side. However, *public m(com.ulcjava.base.application.util.Point)* is not a suitable signature (because it references the server-side point class).

Similarly a suitable remote (i.e. client-side) method signature for the server-side statement

```
ULCProxy.invokeUI("m2", new Object[] {new ULCButton("OK")});
```

is *public m2(UIButton)* but not *public m2(ULCButton)*.

As explained in Section 4.4.2, ULC first tries to dispatch a server-initiated remote method calls on the client-side half object and in case this fails, it tries on the associated basic object. In the second case, ULC performs a second conversion for client-side half object references, which are contained in the client-side argument list for the method dispatch: During the second conversion, a client-side half object reference is replaced by a reference of the associated basic object. This step assures that a basic object must not deal with half objects, when it receives a remote method call. The following example illustrates this behaviour:

In the case of *ULCButton* the associated client-side half object class is *UIButton* and the basic object class is *BasicButton* (a subclass of *JButton*). *ULCButton* offers the method *public setIcon(ULCIcon icon)*.

The implementation of this method triggers a remote method call via *invokeUI()*, which corresponds to *invokeUI("setIcon", new Object[]{ icon })*. On the client-side, ULC first tries to dispatch the related call to a method on the associated *UIButton* instance.The argument list of the dispatch contains a single argument of type *UIIcon*. The latter is the

client-side representation of the *icon*, which was passed in on the server side. Since no matching *setIcon()* method exists for the dispatch on *UIButton*, ULC tries to dispatch the call on the associated basic object of type *BasicButton*. The argument list for the second dispatch try contains a single argument of type *javax.swing.Icon*. The latter is the basic object associated with the *UIIcon* instance of the first dispatch try. Since there is a method *BasicButton.setIcon(javax.swing.Icon)* the call can eventually be dispatched.

The explained replacement of a client-side half object reference by its associated basic object reference only happens for references on the level of the argument list used for the dispatch. It does not happen for client-side half object references, which are nested inside argument objects.

4.5 Events

This section explains how an extension makes use of ULC's mechanism to deliver client-initiated events at the server. In this context, ULC offers a set of support methods on *UIProxy* and *ULCProxy*, which have already been discussed in Section 4.2.

Suppose a basic widget class *Widget* throws a standard event supported by ULC (e.g., *ActionEvent)* and the corresponding client-side half object class is *UIWidget* and the server-side half object class is *ULCWidget*. The following steps show, how to make *UIWidget* and *ULCWidget* handle the action event and propagate it to the registered server-side listeners.

To add support for the action event:

1. Implement an inner class inside *UIWidget* that implements the *java.awt.event.ActionListener* interface and registers it with the *Widget* in the *postInitializeState()* method. The *actionPerformed()* method of *WidgetActionListener* invokes *UIProxy.fireActionPerformedULC()* in order to forward the action event to the server. *fireActionPerformedULC()* obtains important event-related arguments such as the action command and the modifiers. These arguments are used on the server side in order to create a corresponding *com.ulcjava.base.application.event.ActionEvent*, which will be delivered to ULC action listeners. The latter must implement the interface *com.ulcjava.base.application.event.IActionListener*.

```
protected void postInitializeState() {
   super.postInitializeState();
   getBasicPieChart().addActionListener(new WidgetActionListener());
}

private class WidgetActionListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      fireActionPerformedULC(e.getActionCommand(), e.getModifiers());
   }
}
```

2. Add *addActionListener()* and *removeActionListener()* methods to the *ULCWidget* class. These methods forward the registration and unregistration to the *ULCProxy.addListener()* and *ULCProxy.removeListener()* methods:

```
public void addActionListener(IActionListener actionListener) {
    addListener(UlcEventCategories.ACTION_EVENT_CATEGORY,
                actionListener);
}

public void removeActionListener(IActionListener actionListener) {
    removeListener(UlcEventCategories.ACTION_EVENT_CATEGORY,
                   actionListener);
}
```

Given these provisions, ULC forwards a client-side action event from the basic widget to the associated server-side half object and delivers it to registered action listeners. Note that ULC implicitly maps the client-side event type *java.awt.event.ActionEvent* to the server-side event type *com.ulcjava.base.application.event.ActionEvent* when forwarding the event. ULC only sends the event, if one or more action listeners are registered at the server-side half object.

As mentioned in Section 4.2.2 the delivery mode for sending the event can be set via *setEventDeliveryMode()*. If the mode is not set, ULC uses the default event delivery as returned by the method *UIProxy.getEventDeliveryMode()*. The method can be overridden and the default implementation *UIProxy* returns the mode *UlcEventConstants.SYNCHRONOUS_MODE*.

4.6 Extended Visual Effects

This section explains how the extended visual effects such as translucency, rounded corners as well as gradient and image paints are achieved. All of these effects make use of the universal Swing decorator *JXLayer*, which influences the painting of its decorated view as well as all of that view's children. Please note, that the *JXLayer* component within the basic component tree has no counterpart in ULC, i.e. there is neither a client-side *UIXLayer* nor a server-side *ULCXLayer*.

The decoration of a basic component, which must be of the type *JComponent*, is performed on an "as needed" basis, i.e. a basic component is only decorated if any extended visual effect is applied. As a consequence, the decoration may take place before or after adding the basic component to some *Container*.

In case the basic component is first added to some container and then decorated, the basic component has to be replaced by the *JXLayer* while using the same layout constraints. For this purpose *UIContainer* keeps the constraints of added components and applies them in its *decorateAndReplaceBasicComponent* method when replacing the original basic component with the *JXLayer*.

In case the basic component is first decorated and then added to some container, actually the *JXLayer* has to be added instead. For this purpose the new method *UIComponent.getBasicComponentForContainer* was introduced. This method provides the basic component or alternatively its *JXLayer* decorator, depending on whether some extended visual effect was set on the component or not.

There is one exception to the above described mechanism and that is with *JInternalFrames*. As *JDesktopPane* and *DesktopManager* are not capable of handling decorated *JInternalFrames*, the same extended visual effects are achieved by overriding the internal frame's *paint()* method instead.

The way these extended visual effects are achieved is very convenient for application programmers as they do not need to decorate their server-side components just to make use of these extended visual effects. On the other hand, there are some rules that need to be adhered to when integrating a new container with an ULC extension. For details, please see the corresponding chapter 5.8.

5 How Tos

5.1 Extending an Existing Visual ULC Component

This section demonstrates how to implement a simple extension, which adds a property to an existing ULC widget. The example provides a strike-through label – a label which might be painted with a strike-through line, if the corresponding property *strikeThrough* is set. Figure 4 illustrates the appearance of the strike-through label, with the *strikeThrough* property set to *false* and set to *true*, respectively.

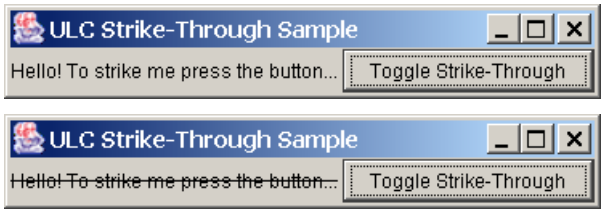

Figure 4: The Appearance of the Strike-Through Label

The strike-through widget is based on the standard label widget from ULC. To add the new property and the related functionality, one must extend the three ULC classes of the label widget: *ULCLabel* is the server-side half object class of the standard ULC label widget, *UILabel* is the client-side half object class of the standard ULC label widget and *UILabel.BasicLabel* is the basic widget which extends *javax.swing.JLabel*.

The client-side extension classes for the strike-through label look as follows (see the comments in the code for details):

```
package sample.client;
import com.ulcjava.base.client.UILabel;
import java.awt.Graphics;

// The UI class inherits from UILabel.
public class UIStrikeThroughLabel extends UILabel {
    // Override to create the basic strike-through label.
    protected Object createBasicObject(Object[] a) {
        return new BasicStrikeThroughLabel();
    }
    // The basic widget class extends BasicLabel, whereby BasicLabel is
    // an extension of javax.swing.JLabel.
    // This class must be public so that ULC's remote method dispatch on
    // setStrikeThrough() works.
    public class BasicStrikeThroughLabel extends BasicLabel {
        // The strike-through property is maintained on the basic widget
        // because it is needed for painting. The default is false.
        boolean fStrikeThrough = false;

        public boolean isStrikeThrough() {
            return fStrikeThrough;
        }

        // This setter method is called from the ULC remote method call
        // dispatcher to set or unset the strike-through behaviour.
```

```
        public void setStrikeThrough(boolean strikeThrough) {
            this.fStrikeThrough = strikeThrough;
        }

        // This override of paint, provides the strike-through
        // appearance of the label.
        protected void paintComponent(Graphics g) {
          super.paintComponent(g);
          // Paint the strike-through line,
          // if the strikeThrough property is set.
          if (isStrikeThrough()) {
             g.setColor(getForeground());
             g.drawLine(0, getHeight() / 2, getWidth(), getHeight() / 2);
          }
       }
    }
}
```

The server-side class for the strike-through label looks as follows (see the comments in the code for details):

```
package sample.server;
import com.ulcjava.base.application.ULCLabel;

// The ULC class inherits from ULCLabel.
public class ULCStrikeThroughLabel extends ULCLabel {
    // The server-side representation of the strikeThrough property.
    // The default value for the property is false. This value should
    // be and is identical to the corresponding client-side default.
    boolean fStrikeThrough = false;

    public ULCStrikeThroughLabel(String text) {
        super(text);
    }

    // Called during upload...
    protected void uploadStateUI() {
        super.uploadStateUI();
        // Propagate the current value of the fStrikeThrough field
        // to the client, if it differs from the default value.
        // On the client-side, the related remote method call will
        // be dispatched to BasicStrikeThroughLabel.setStrikeThrough().
        setStateUI("strikeThrough", false, fStrikeThrough);
    }

    public boolean isStrikeThrough() {
        return fStrikeThrough;
    }
    // To set the strike-through value.
    public void setStrikeThrough(boolean inStrikeThrough) {
        // The invocation of setStateUI() ensures that a change
        // of the server-side strikeThrough property
```

```
        // is propagated to the client even after the widget has been
        // uploaded.
        fStrikeThrough =
            setStateUI("strikeThrough", fStrikeThrough, inStrikeThrough);
    }
    // Names the client-side half object class associated with this class.
    protected String typeString() {
        return "sample.client.UIStrikeThroughLabel";
    }
}
```

The following code shows, how to use the strike-through label in order to obtain the application from Figure 4:

```
package sample.server;
import com.ulcjava.base.application.AbstractApplication;
import com.ulcjava.base.application.ULCBoxPane;
import com.ulcjava.base.application.ULCButton;
import com.ulcjava.base.application.ULCFrame;
import com.ulcjava.base.application.event.IActionListener;
import com.ulcjava.base.application.event.ActionEvent;

public class Sample extends AbstractApplication {
    public void start() {
        final ULCStrikeThroughLabel label =
            new ULCStrikeThroughLabel("Hello! " +
                    "To strike me press the button...");
        ULCBoxPane boxPane = new ULCBoxPane();
        boxPane.add(label);
        ULCButton button = new ULCButton("Toggle Strike-Through");
        button.addActionListener(new IActionListener() {
           public void actionPerformed(ActionEvent arg0) {
               label.setStrikeThrough(!label.isStrikeThrough());
               // Triggers a repaint on the client side in order to
               // paint or wipe out the strike-through line.
               label.repaint();
           };
        });
        boxPane.add(button);
        ULCFrame frame = new ULCFrame("ULC Strike-Through Sample");
        frame.setDefaultCloseOperation(ULCFrame.TERMINATE_ON_CLOSE);
        frame.add(boxPane);
        frame.setVisible(true);
    }
}
```

5.2 Integrating a Non-Visual Client-Side Service

This section demonstrates how to implement a simple, non-visual extension, which is not based on an existing ULC component. The example provides a service for triggering the execution of arbitrary processes on the client.[2] It allows the server side ULC application to query the client-side free Java heap memory.

The service extension consists of two classes: *ULCRuntime* is the server-side half object class and *UIRuntime* is the associated client-side half object class. The example uses the existing class *java.lang.Runtime* as its basic object class.

To execute a system operation on the client, the server-side half object method *execOnClient()* simply performs a remote method call which triggers an invocation of *java.lang.Runtime.exec()* on the client.

Requesting the client-side free memory at the server is a little bit more complex: To trigger the request, the *ULCRuntime* instance first informs the *UIRuntime* instance that it needs the latest update on free client memory. This happens via a remote method call of *UIRuntime.updateFreeClientMemory()*. In response, *UIRuntime* performs a remote method invocation on *ULCRuntimeDispatcher.updateFreeClientMemory()*. The latter method updates the server-side property *freeClientMemory* with the latest free memory and notifies a listener about the update.

The listener approach is useful because the response from the client happens asynchronously at the next server round trip. In this context, the listener approach ensures that a server-side object, which is interested in the latest free client memory update, will not be "too early" when accessing the *freeClientMemory* property. In other words, the following server-side code would generally *fail* to get the *latest* client memory update:

```
ULCRuntime runtime = new ULCRuntime();
runtime.upload(); // Trigger an  explicit upload.
runtime.updateFreeClientMemory();
System.out.println("Current free client memory in bytes: " +
                   runtime.getFreeClientMemory());
```

Instead, the correct solution is as follows:

```
ULCRuntime runtime = new ULCRuntime();
runtime.upload();
runtime.setFreeClientMemoryUpdateListener(
   new FreeClientMemoryUpdateListener() {
   public void memoryUpdated(long memory) {
      System.out.println("Current free client memory in bytes: " + memory);
   }
});
runtime.updateFreeClientMemory();
```

[2] The behaviour of this extension is arguably dangerous, because it enables the server to potentially perform arbitrary system operations on the client. Therefore, please note that the example is not meant to be applied in practice. It has been chosen, because it is simple and also highlights important characteristics of non-visual ULC extensions.

The client-side extension class *UIRuntime* looks as follows (see the comments in the code for details):

```
package sample.client;
import com.ulcjava.base.client.UIProxy;
import java.io.IOException;

// The UI class inherits from UIProxy because it is not based on an existing
// ULC component.
public class UIRuntime extends UIProxy {
    // The basic object is the Java instance of Runtime.
    protected Object createBasicObject(Object[] args) {
        return Runtime.getRuntime();
    }

    // A convenience method to return the down-casted basic object.
    protected Runtime getBasicRuntime() {
        return (Runtime) getBasicObject();
    }

    // The method to be called from the ULC remote method call dispatcher
    // for starting a process on the client. It simply delegates to the basic
    // object. This method is neseccary for the following reason:
    // The method Runtime.exec() is overloaded in an ambigous way
    // such that ULC cannot dispatch a remote method call right on Runtime
    // (see Section 4.4.1 for details on dispatchable method calls).
    // Since the exec() method here (on UIRuntime) is not overloaded,
    // the dispatch works.
    public void exec(String[] commandArray) throws IOException {
        getBasicRuntime().exec(commandArray);
    }

    // This method sends the current free memory to the server via
    // updateStateULC(). updateStateULC() causes a remote method call on the
    // server, which is dispatched to updateFreeClientMemory().
    // The latter method must defined as public on ULCRuntime or
    // on the associated dispatcher class ULCRuntime.ULCRuntimeDispatcher.
    // The ULCRuntime class invokes UIRuntime.updateFreeClientMemory() via
    // invokeUI() and gets the result in return.
    public void updateFreeClientMemory() {
        updateStateULC("freeClientMemory", getBasicRuntime().freeMemory());
    }
}
```

The server-side class *ULCRuntime* looks as follows (see the comments in the code for details):

```
package sample.server;
```

```
import com.ulcjava.base.application.ULCProxy;
import com.ulcjava.base.server.IDispatcher;

// The ULC class inherits from ULCProxy because it is not based
// on an existing ULC component.
// Note that in the constructor you need to mark this ULCProxy
// to be un-collectable by the server side garbage collector.
// In addition you may also choose to upload() the ULCProxy to the
// client in the constructor itself.

public class ULCRuntime extends ULCProxy {
    // A property to store the last update of the client-side free memory.
    // -1 indicates that the client-side free memory is initially unknown.
    private long fFreeClientMemory = -1;
    // The listener to be informed about data updates.
    private FreeClientMemoryUpdateListener fFreeClientMemoryUpdateListener;

    public ULCRuntime() {
        // make sure that this proxy is not garbage collected on the server
        markUncollectable();
        // Do implicit upload or alternatively call upload() explicitly when
        // needed
        //upload();
    }

    protected String typeString() {
        return "sample.client.UIRuntime";
    }

    public void execOnClient(String[] commandArray) {
        invokeUI("exec", new Object[] { commandArray });
    }

    // Trigger an update request from server to client.
    public void updateFreeClientMemory() {
        invokeUI("updateFreeClientMemory");
    }

    public long getFreeClientMemory() {
        return fFreeClientMemory;
    }

    public void setFreeClientMemoryUpdateListener(
        FreeClientMemoryUpdateListener listener) {
        fFreeClientMemoryUpdateListener = listener;
    }

    // Updates the freeClientMemory property and
    // notifies an associated listener.
    private void updateFreeClientMemory(long freeClientMemory) {
        fFreeClientMemory = freeClientMemory;
        if (fFreeClientMemoryUpdateListener != null) {
            fFreeClientMemoryUpdateListener.memoryUpdated(freeClientMemory);
```

```
        }
    }

    // To create an instance of the local dispatcher class
    // (see Section 4.4.2).
    protected IDispatcher createDispatcher() {
        return new ULCRuntimeDispatcher();
    }

    // The dispatcher class extension to dispatch the free memory update call
    // (See Section 4.4.2 for dispatcher classes.)
    protected class ULCRuntimeDispatcher extends ULCProxyDispatcher {
        public final void updateFreeClientMemory(long freeClientMemory) {
            ULCRuntime.this.updateFreeClientMemory(freeClientMemory);
        }
    }

    // The listener interface for notifying a
    // listener about the memory update.
    public interface FreeClientMemoryUpdateListener {
        public void memoryUpdated(long memory);
    }
}
```

5.3 Adding a Custom Event Type

This section improves the example from the last section and shows how to make use of ULC's event support in order to deliver updates on free client memory to a group of server-side listeners. In this respect, the example from Section 5.3 is flawed because it only allows a single listener to register for a respective update (via *ULCRuntime.setFreeClientMemoryUpdateListener()*).

The general goal of this section is to demonstrate, how an extension can support a custom server-side event class, which is not shipped with ULC.

For this purpose, ULC leverages the standard Java Beans event model, where listener interfaces extend *java.util.EventListener* and event classes inherit from *java.util.EventObject*. To improve the example from Section 5.3, the following code defines an event class to send client-side memory updates and provides a corresponding listener interface. For convenience, the class and the interface are defined inside *ULCRuntime*:

```
package sample.server;
import com.ulcjava.base.application.ULCProxy;
import com.ulcjava.base.server.IDispatcher;

import java.util.EventListener;
import java.util.EventObject;

public class ULCRuntime extends ULCProxy {

   ... // The other changes of ULCRuntime will be presented below.
```

```
   public interface FreeClientMemoryUpdateListener extends EventListener {
      public void memoryUpdated(FreeClientMemoryUpdateEvent event);
   }

   public static class FreeClientMemoryUpdateEvent extends EventObject {
       private long fFreeClientMemory;

       public FreeClientMemoryUpdateEvent(Object source,
                                          long freeClientMemory) {
            super(source);
            fFreeClientMemory = freeClientMemory;
        }

        public long getFreeClientMemory() {
            return fFreeClientMemory;
        }
    }
}
```

The next step is to ensure that listeners of the type *FreeClientMemoryUpdateListener* can register on *ULCRuntime*. To do this, we define a name, the so-called event category, which groups all memory update listeners on a *ULCRuntime* instance. For the given example, the related event category name is *"freeClientMemoryUpdate"*.

By implementing the method *ULCProxy.getFreeClientMemoryUpdateListenerClass()* appropriately, the new event category becomes associated with the interface *FreeClientMemoryUpdateListener*. This enables ULC to deliver memory update events to the corresponding listeners. ULC searches for this method based on the name of the event category. In general, for an event category name *xyz*, the related get-method must have the signature *java.lang.Class getXyzListenerClass()* and must return the associated listener interface class object. If the method is missing or if it does not return the class object, then ULC will throw a runtime exception when it tries to deliver the event.

```
protected Class getFreeClientMemoryUpdateListenerClass() {
   return FreeClientMemoryUpdateListener.class;
}
```

In addition, *ULCRuntime* should offer the corresponding add and remove listener methods:

```
public void addFreeClientMemoryUpdateListener(
        FreeClientMemoryUpdateListener listener) {
   // Use ULCProxy.addListener() to associate the listener with the event
   // category "freeClientMemoryUpdate".
   addListener("freeClientMemoryUpdate", listener);
}

public void removeFreeClientMemoryUpdateListener(
        FreeClientMemoryUpdateListener listener) {
   // Use ULCProxy.removeListener() to remove the listener from the listener
   // list associated with the event category "freeClientMemoryUpdate".
```

```
   removeListener("freeClientMemoryUpdate", listener);
}
```

When the client forwards an event to the server, ULC does not actually send an event object. Instead, it sends a list of arguments, which describe the event and supplies a special create event method with these arguments on the server side. The create event method must be defined on the server-side half object class or on the associated dispatcher class. The prefix of the related method name is "*create"*, the middle part is the name of the associated event category with the first letter in upper case and the suffix is "*Event*". Thus, for the example the name of the create event method is "*createFreeClientMemoryUpdateEvent*". This method must have a parameter list so that the arguments provided on the client side are assignable to the given parameters. For the example, the client sends a single *long* argument – the free client memory in bytes.

The purpose of the create event method is to create an event object from the given arguments on the server side and so, it should return a corresponding event object – namely instance of *java.util.EventObject*. Therefore, the create event method looks as follows for the provided example:

```
protected FreeClientMemoryUpdateEvent
   createFreeClientMemoryUpdateEvent(long freeClientMemory) {
   return new FreeClientMemoryUpdateEvent(this, freeClientMemory);
}
```

After the event has been created, ULC tries to process the event. To do so, it looks for a a special event process method on the server-side half object class or its associated dispatcher class. The prefix of the related method must be *process* and the rest of the name is the same as for the create event method.

Usually the process event method delivers the event to listeners of the event category, but sometimes it might be useful, if the implemention performs other or additional operations. The process event method has got three parameters: the event category name (type *String*), the listener method name, to which the event is to delivered (type *String*) and the event object (type *java.util.EventObject* or an appropriate subtype). To deliver the event to registered listeners, add a call of *ULCProxy.dispatchEvent()* and supply it with the parameters from the process event method. (However, calling *dispatchEvent()* is optional.)

For the example the process event method looks as follows:

```
protected void processFreeClientMemoryUpdateEvent(
   String eventCategory, String listenerMethodName,
   FreeClientMemoryUpdateEvent event) {
   // Update the server-side freeClientMemory property.
   fFreeClientMemory = event.getFreeClientMemory();
   // Deliver the event to registered listeners.
   dispatchEvent(eventCategory, listenerMethodName, event);
}
```

In order to make the process event method and the create event method accessible to ULC's event dispatching mechanism, they must be added to a new version of the dispatcher class:

```
protected class ULCRuntimeDispatcher extends ULCProxyDispatcher {
  public final FreeClientMemoryUpdateEvent
              createFreeClientMemoryUpdateEvent(long freeClientMemory) {
     return
        ULCRuntime.this.createFreeClientMemoryUpdateEvent(freeClientMemory);
  }

  public final void processFreeClientMemoryUpdateEvent(
                        String eventCategory, String listenerMethodName,
                        FreeClientMemoryUpdateEvent event) {
     ULCRuntime.this.
        processFreeClientMemoryUpdateEvent(eventCategory,
                                           listenerMethodName, event);
  }

  public final Class getFreeClientMemoryUpdateListenerClass() {
     return ULCRuntime.this.getFreeClientMemoryUpdateListenerClass();
  }
}
```

After removing the methods *ULCRuntime.setFreeClientMemoryUpdateListener()* and *ULCRuntime.updateFreeClientMemory()* (see Section 5.2), the improved version of *ULCRuntime* is complete.

The only thing that remains open is to adjust *UIRuntime*. In the latter class, we replace the old version of *updateFreeClientMemory()* by the following one:

```
public void updateFreeClientMemory() {
    fireEventULC("freeClientMemoryUpdate",
                 "memoryUpdated",
                 new Object[]{ new Long(getBasicRuntime().freeMemory()) });
}
```

The *fireEventULC()* method sends the long argument to create and process the server-side event to the associated server-side half object (of type *ULCRuntime*). ULC uses the passed in event category name "freeClientMemoryUpdate" to determine the name of the appropriate create and process event methods. The elements of the given object array (in this case only a long value) are eventually passed in to the server-side create event method. The string "memoryUpdated" is passed in as the second argument (the listener method name) to the server-side process event method.

5.4 Writing a Custom Coder

As explained in Section 4.4.4 the ULC serialization infrastructure is in charge of marshalling objects from the client to the server side and vice versa. ULC supports a set of standard types, whose instances it can marshall right away. However, sometimes it might be useful to marshal instances of additional custom classes.

In this section we briefly demonstrate how to extend ULC serialization infrastructure such that instances of custom classes can be marshalled.

The ULC serialization infrastructure is capable of changing the class type of an object, which is serialized and afterwards deserialized. This feature is necessary for some ULC classes, but it is unlikely to be used for custom classes. Therefore, our example focuses on the case where the type of a serialized object does *not change* during marshalling.

Before proceeding further, it may be pointed out that it is not always necessary to write a coder for a custom class, i.e., there are alternate ways of marshalling objects of a custom class between client and server, and vice-versa. For example, you could use a *Map* to send across individual attributes (that are of standard types) of a custom class. So, for a *ComplexNumber* class you could send the *real* and the *imaginary* parts, which are of type *double*, as two key-value pairs within a *Map*. ULC provides coders for types *double* and *Map* out of the box. Another way of sending across custom objects is to use the *invokeUI* and *invokeULC* APIs that take mulitple arguments. For instance, to pass an object of class *ComplexNumber* while calling a method on a client-side proxy from a server-side proxy you could use:

```
invokeUI("clientSideMethod", new Object[] {
       new Double(complexNumber.getRealPart()),
       new Double(complexNumber.getImaginaryPart())});
```

Consequently, in the client-side method, you can construct an instance of *ComplexNumber* from the arguments:

```
public void clientSideMethod(double realPart, double imaginaryPart) {
   ComplexNumber complexNumber = new ComplexNumber(realPart, imaginaryPart);
   …
}
```

Now we come back to the theme of this section, i.e., writing a coder for a custom class. Suppose you want to marshall complex numbers, i.e., corresponding numbers should be instances of the following class *sample.ComplexNumber*:

```
package sample;
import java.io.Serializable;

public class ComplexNumber implements Serializable {
    private final double fRealPart;
    private final double fImaginaryPart;

    public ComplexNumber(double realPart, double imaginaryPart) {
        fRealPart = realPart;
        fImaginaryPart = imaginaryPart;
    }

    public double getRealPart() {
        return fRealPart;
    }
```

```
    public double getImaginaryPart() {
        return fImaginaryPart;
    }

    // Add complex number arithmetic operations here...
}
```

ComplexNumber implements the standard Java serialization interface *java.io.Serializable*. This is not strictly necessary because, as mentioned in Section 4.4.4, ULC does not rely on Java's standard object serialization, since the latter approach might incur performance and security problems. However, it is still recommended to implement *java.io.Serializable*.

The next step is to provide an additional class, a so-called *coder class*, which tells the ULC serialization infrastructure, how it must serialize and deserialize a complex number. An instance of a coder class is simply called a *coder*. A coder class must implement the interface *com.ulcjava.base.shared.IStreamCoder* with the following three methods:

- *String getPeerClassName()* returns the name of the class, of which ULC creates an instance, when a marshalled object is deserialized. If the object should not change its type during marshalling then this name is simply the fully qualified class name of the serialized object itself.

  If the marshalled object should change its type, then the returned name is the fully qualified class name of the object after deserialization.

  ULC writes the class name returned by *getPeerClassName()* as a string to the stream before serializing the actual data of the respective object. The written class name is used at deserialization time in order to determine the coder for reading the data from the stream and therefore correctly deserializing the object.

- *writeObject(IObjectOutputStream out, Object object)* is in charge of serializing the object, which is passed in as *object*. The implementation of *writeObject()* must write all information from *object*, which is relevant for marshalling, to the stream *out*. For this purpose, the interface *IObjectOutputStream* offers methods to write primitive elements of *object* to the stream, e.g. *writeBoolean(boolean)*, *writeInt(int)*, *writeDouble(double)* and so on.

  Moreover, *IObjectOutputStream* offers the method *IObjectOutputStream .writeObject(Object)*, which allows for serializing objects contained in *object* to the stream. By standard, *IObjectOutputStream.writeObject()* is capable of writing array structures, strings and the *null* value as contained objects to the stream. If the object passed in to *IObjectOutputStream.writeObject()* is not an array or a string or *null*, then the ULC serialization infrastructure tries to find a coder for that object and applies the coder's *writeObject()* method to it. Therefore, it is necessary that a related coder is implemented and registered at the ULC serialization infrastructure. (The registration of a coder will be discussed below.)

  According to the previous paragraph, writing an object with contained objects is a recursive process which involves coders associated with each of the participating objects, which are being serialized. If there is cyclic reference of

objects to be serialized, then this recursive process will cause an infinite loop, which results in a runtime exception. Besides, if ULC cannot find a coder for a participating object, then it also throws a runtime exception.

- *Object readObject(IObjectInputStream in)* is in charge of deserializing an object from the input stream *in*. The method *readObject()* must read the serialized data in the same order from the *in*, as the data has been written to the corresponding output stream before via *writeObject(IObjectOutputStream, Object)*. ULC invokes *readObject()* on a coder, which is associated with the class name as given by *getPeerClassName()* at serialization time of the marshalled object (see description of *getPeerClassName()* from above). By contract, *readObject()* must return an instance of the class, whose name was given by *getPeerClassName()* at serialization time.

  To obtain data from the stream *in*, *IObjectInputStream* offers methods to read primitive data, e.g. *readBoolean(boolean)*, *readInt(int)*, *readDouble(int)* and so on. In addition, the method *readObject()* allows for reading of contained objects, which were written to the stream via *IObjectOutput.writeObject()* at serialization time. In short, the process of reading data from the input stream is complementary to the above explained write process.

Given this background knowledge, the implementation of the coder class for *ComplexNumber*, is straight-forward:

```
package sample;
import com.ulcjava.base.shared.IObjectInputStream;
import com.ulcjava.base.shared.IObjectOutputStream;
import com.ulcjava.base.shared.IStreamCoder;

import java.io.IOException;

public class ComplexNumberCoder implements IStreamCoder {
    // Since the type of a complex number does not change during serialization
    // the peer class name is simply sample.ComplexNumber.
    public String getPeerClassName() {
        return ComplexNumber.class.getName();
    }

    public void writeObject(IObjectOutputStream out, Object data) {
        ComplexNumber c = (ComplexNumber)data;
        // Write the contained data: the real part and the imaginary part.
        out.writeDouble(c.getRealPart());
        out.writeDouble(c.getImaginaryPart());
    }

    public Object readObject(IObjectInputStream in) throws IOException {
        // Read the data from the stream: first the real part, then the
        // imaginary part, and return a corresponding deserialized complex
        // number object of type sample.ComplexNumber.
        return new ComplexNumber(in.readDouble(), in.readDouble());
    }
}
```

To make use of the new coder class, related coders must be registered at the ULC serialization infrastructure. This must be done at both ends of the communication line, namely at the client and the server side. This is achieved by adding a <coders> element in the *ULCApplicationConfiguration.xml*.

```
<ulc:coders>
    <ulc:symmetricCoder>
        <ulc:class>
            sample.ComplexNumber
        </ulc:class>
        <ulc:coderClassName>
            sample.ComplexNumberCoder
        </ulc:coderClassName>
    </ulc:symmetricCoder>
</ulc:coders>
```

5.5 Configuring Carrier Streams for Communication

When the ULC serialization infrastructure (see Section 4.4.4) reads and writes data for communication between client and server, it essentially relies on the functionality of the Java classes *java.io.InputStream* and *java.io.OutputStream*. However, in many cases it might be useful to additionally encode/decode the output/input of the ULC serialization infrastructure before the data is eventually written to/read from the raw Java stream. E.g., useful encodings at this point are zipping and base 64 encoding.

To enable this kind of encoding ULC allows the configuration of related intermediate input or output streams – the so-called *carrier streams*. For the communication to work, the encoding and decoding carrier streams must match on the client and on the server side. Therefore, carrier streams must always be configured for both, the client-*and* the server-side deployment. The configuration of a carrier stream is done via a carrier stream provider – a class, which implements the interface *com.ulcjava.base.shared.ICarrierStreamProvider*.

ULC already ships with three implementions of *ICarrierStreamProvider*:

- *com.ulcjava.base.shared.ZipCarrierStreamProvider* provides carrier streams, which perform a zip encoding/decoding on communication data. This is the default encoding used by ULC.
- *com.ulcjava.base.shared.TrivialCarrierStreamProvider* provides carrier streams, which perform *no* further encoding/decoding of communication data. If this provider is used, then requests exchanged between client and server are larger and consume more bandwidth than in the case of zipping carrier streams. The advantage of the trivial carrier streams is that they require less CPU resources than zipping carrier streams.
- *com.ulcjava.base.shared.Base64CarrierStreamProvider* provides carrier streams for a standard base 64 encoding/decoding of communication data.

To configure a carrierStreamProvider add a <carrierStreamProvider> element in the ULCApplicationConfiguration.xml. The carrierStremProvider is specified with one of the two subelements:

<standardCarrierStreamProvider> to choose one of the standard implementations (TRIVIAL, BASE64 , ZIP):

```
<ulc:carrierStreamProvider>
   <ulc:standardCarrierStreamProvider>
      ZIP
   </ulc:standardCarrierStreamProvider>
</ulc:carrierStreamProvider>
```

Or <carrierStreamProviderClassName> to specifiy your own implementation class:

```
<ulc:carrierStreamProvider>
   <ulc:carrierStreamProviderClassName>
      sample.CarrierStreamProvider
   </ulc:carrierStreamProviderClassName>
</ulc:carrierStreamProvider>
```

5.6 Configuring Customized Model Adapter

Sometimes it is useful to use customized model adapters to control and modify the behavior of server- and client side model synchronization e.g. to use differing lazy loading strategies.

Model adapter providers create model adapters. A model adapter provider implements the *com.ulcjava.base.server.IModelAdapterProvider* interface. By default ULC uses the *com.ulcjava.base.server.DefaultModelAdapterProvider* to create new model adapters. The *getModelAdapterProvider()* method on *com.ulcjava.base.server.ULCSession* returns the installed model adapter provider.

To register own model adapters, ULC offers two possibilities as described in the following sections.

5.6.1 Generic Approach

Instead of the *DefaultModelAdapterProvider* it is possible to configure a custom model adapter provider. This custom model adapter provider has to implement the *IModelAdapterProvider* interface.

To configure a modelAdapterProvider add a *<modelAdapterProviderClassName>* element in the *ULCApplicationConfiguration.xml*.

```
<ulc:modelAdapterProviderClassName>
   sample.CustomModelAdapterProvider
</ulc:modelAdapterProviderClassName>
```

5.6.2 Specific approach

The interface class *com.ulcjava.base.server.IModelAdapterProvider* declares the method *registerModelAdapter()* to register a model adapter for a given model type and a given model object. This method is useful to register single specialized adapters for special models. When using this approach it is important to registere the custom model adapter before the model is used i.e. before setting it to a corresponding component.

5.7 Implementing Custom Renderer and Editor

The appearance of ULC widgets such as *ULCComboBox, ULCList, ULCTable, ULCTableTree* and *ULCTree* can be customized using renderers and editors. ULC provides support for the following components to be used as a renderer and/or editor: *ULCLabel*, *ULCCheckBox*, *ULCComboBox*, and *ULCTextField*. However, some time there may be a need to use some other *ULCComponent* as a renderer or an editor. This section explains basic concepts about renderer/editor and describes:

1. How to extend the *ULCProgressBar* to be able to use it as a renderer in a *ULCTable*
2. How to extend the *ULCSlider* to be able to use it as an editor in a *ULCTable*

For the purpose of simplicity only the implementation of renderer and editor for *ULCTable* has been described. Nevertheless, the same concepts and mechanisms apply to renderers/editors for components *ULCComboBox*, *ULCList*, *ULCTree*, and *ULCTableTree*.

5.7.1 Renderers and Editors in ULC

Renderers in a table are used to define the visual representation of the table cell. The renderer is responsible to return a renderer component for each table cell. For the purpose of performance optimization, when a renderer is defined for a table, only specific attributes of the renderer component are sent to the client and not the renderer component itself. For instance, if a *ULCComboBox* is returned for a table cell then the entries of the *ULCComboBox* are sent to the client in addition to the visual attributes like the font, the background and foreground colors etc. On the client side, a new component (e.g. a new *UIComboBox and its BasicComboBox*) is created and configured with these attributes. It is this component that is used as renderer component on the *BasicTable*. Note that ULC does not send information such as listeners attached to the renderers

Renderer components are uploaded to the client in a lazy manner. They obtain the values to be displayed from the client side model. Therefore setting values on the renderer components on the server side has no effect.

In addition the value displayed on the renderer component on the client side is not in sync with the value on the renderer component on the server side and that value cannot be accessed from the server side renderer component. This is because ULC only sends renderer component templates to the client and not the actual component. On the client, based on the attributes of the templates, a new appropriate component is generated and used as renderer/editor. This new component is not linked in any way to the original renderer component on the server side.

The same mechanism as for renderers is used for editors (they define the visual representation of the table cell in edit mode) and for all other cell based widgets: combo box, list, tree, and table tree.

5.7.2 *ICellComponent* Interface

ULC minimizes the number of renderer/editor components it must create and upload to the client by reusing the components of the same type that have the same visual attributes. ULC uses the server-side *IRendererComponent* and *IEditorComponent* interfaces to manage templates for the renderer/editor components. It maintains a cache of renderer/editor components it has already uploaded, and uploads a component only if

it differs in at least one visual attribute value from the components in the cache. Therefore, ULC needs some way of asking a renderer/editor component whether it is visually different from another component. For this purpose, ULC provides the *ICellComponent* interface to be implemented by ULC component that is to be used a renderer/editor component.

The *ICellComponent.areAttributesEqual()* method compares the attribute values in order to determine if a renderer/editor component with a given set of attributes has already been created.

The *ICellComponent.attributesHashCode()* method returns a hash code of attribute values that is compatible with *areAttributesEqual()*. It helps to speed up the cache lookup.

The *ICellComponent.copyAttributes()* method copies the attributes of the provided source object into the call target. This method exists because ULC typically obtains the renderer and editor components for all visible table cells in a single server roundtrip, and must handle the case that an *ITableCellRenderer/Editor* implementation always returns the same component instance, but with different visual attributes each time. To accommodate this, ULC clones the component returned by *getTableCellEditorComponent()* / *getTableCellRendererComponent()* if necessary, and it uses *copyAttributes()* to do this. The clone is instantiated using *Class.newInstance()*, so an *ICellComponent* implementation must have a publicly accessible no-arg constructor.

The contract for an *ICellComponent* implementation also entails that the client half object implement the client-side *com.ulcjava.base.client.IRendererComponent* and/or *com.ulcjava.base.client.IEditorComponent* interfaces, which define methods for retrieving the renderers and editors that are actually installed on the underlying JTable. ULC uses the client-side *IRendererComponent* and/or *IEditorComponent* interfaces to create Swing renderers out of the ULC components and therefore in the implementation of these interfaces one has to return the correct Swing renderer/editor for the actual ULC component state.

5.7.3 Extending *ULCProgressBar* for use as a cell renderer

To use a *ULCProgressBar* as a renderer component you need to write an ULC extension, which involves extending both *ULCProgressBar* and its client side proxy *UIProgressBar*.

In the code below, we extend *ULCProgressBar* and make it implement three methods of *com.ulcjava.base.application.IRendererComponent interface,* namely *areAttributesEqual(), attributesHashCode(),* and *copyAttributes().*

```
public class ULCCellRenderableProgressBar extends ULCProgressBar
          implements com.ulcjava.base.application.IRendererComponent {
    public void copyAttributes(ICellComponent source) {
        ULCCellRenderableProgressBar sourceProgressBar =
                                (ULCCellRenderableProgressBar)source;

        setBackground(sourceProgressBar.getBackground());
        setFont(sourceProgressBar.getFont());
        setForeground(sourceProgressBar.getForeground());
        setMaximum(sourceProgressBar.getMaximum());
```

```
        setMinimum(sourceProgressBar.getMinimum());
        setOrientation(sourceProgressBar.getOrientation());
        setToolTipText(sourceProgressBar.getToolTipText());
    }

    public boolean areAttributesEqual(ICellComponent component) {
        if (!(component instanceof ULCCellRenderableProgressBar)) {
            return false;
        }

        ULCCellRenderableProgressBar other =
                              ULCCellRenderableProgressBar)component;

        return equals(getBackground(), other.getBackground()) &&
                 equals(getFont(), other.getFont())&&
                 equals(getForeground(), other.getForeground()) &&
                 (getMaximum() == other.getMaximum()) &&
                 (getMinimum() == other.getMinimum()) &&
                 (getOrientation() == other.getOrientation())&&
                  equals(getToolTipText(), other.getToolTipText());
    }

    public int attributesHashCode() {
        int result = 17;
        result = 37 * result +
                 (getBackground() == null ? 0 : getBackground().hashCode());
        result = 37 * result + (getFont() == null ? 0 : getFont().hashCode());
        result = 37 * result +
                 (getForeground() == null ? 0 : getForeground().hashCode());
        result = 37 * result + getMaximum();
        result = 37 * result + getMinimum();
        result = 37 * result + getOrientation();
        result = 37 * result +
                 (getToolTipText() == null ? 0 : getToolTipText().hashCode());
        return result;
    }

    protected String typeString() {
        return "sample.client.UICellRenderableProgressBar";
    }
}
```

On the client side, we extend *UIProgressBar*. It implements *com.ulcjava.base.client.IRendererComponent* interface and *getTableCellRenderer()* method that returns an instance of *MyTableCellRenderer*, which in turn implements Swing's *TableCellRenderer.*

```
public class UICellRenderableProgressBar extends UIProgressBar
                 implements com.ulcjava.base.client.IRendererComponent {
    private TableCellRenderer fTableCellRenderer;
```

```
    protected void postInitializeState() {
        super.postInitializeState();
        fTableCellRenderer = new MyTableCellRenderer();
    }

    public TableCellRenderer getTableCellRenderer() {
        return fTableCellRenderer;
    }

    public TreeCellRenderer getTreeCellRenderer() {
        throw new UnsupportedOperationException();
    }

    public ListCellRenderer getListCellRenderer() {
        throw new UnsupportedOperationException();
    }

    public TableTreeCellRenderer getTableTreeCellRenderer() {
        throw new UnsupportedOperationException();
    }

    private class MyTableCellRenderer implements TableCellRenderer {
        public Component getTableCellRendererComponent(JTable table,
                    Object value, boolean isSelected, boolean hasFocus,
                    int row, int column) {
            Number numberValue = (value instanceof Number) ?
                                    (Number)value : null;

            JProgressBar result = (JProgressBar)getBasicComponent();
            result.setValue(numberValue.intValue());
            result.setOpaque(true);
            result.setBorderPainted(false);

            return result;
        }
    }
}
```

5.7.4 Extending *ULCSlider* for use as a cell editor

To use a *ULCSlider* as an editor component you need to write a ULC extension, which involves extending both *ULCSlider* and its client side proxy *UISlider.*

In the code below, we extend *ULCSlider* and make it implement three methods of *com.ulcjava.base.application.IEditorComponent interface,* namely *areAttributesEqual(), attributesHashCode()*, and *copyAttributes().*

```
public class ULCCellEditableSlider extends ULCSlider implements
                            com.ulcjava.base.application.IEditorComponent {
    public boolean areAttributesEqual(ICellComponent component) {
        if (!(component instanceof ULCCellEditableSlider)) {
            return false;
        }
```

```
    ULCCellEditableSlider other = (ULCCellEditableSlider)component;

    return equals(getBackground(), other.getBackground())
           && equals(getFont(), other.getFont())
           && equals(getForeground(), other.getForeground())
           && (getMaximum() == other.getMaximum())
           && (getMinimum() == other.getMinimum())
           && (getOrientation() == other.getOrientation())
           && equals(getToolTipText(), other.getToolTipText())
           && (getPaintTicks() == other.getPaintTicks())
           && (getPaintTrack() == other.getPaintTrack())
           && (getInverted() == other.getInverted())
           && (getSnapToTicks() == other.getSnapToTicks())
           && (getPaintLabels() == other.getPaintLabels())
           && (getMajorTickSpacing() == other.getMajorTickSpacing())
           && (getMinorTickSpacing() == other.getMinorTickSpacing());

}

public int attributesHashCode() {
    int result = 17;
    result = 37 * result + (getBackground() == null ? 0 :
                            getBackground().hashCode());
    result = 37 * result + (getFont() == null ? 0 :
                            getFont().hashCode());
    result = 37 * result + (getForeground() == null ? 0 :
                            getForeground().hashCode());
    result = 37 * result + getMaximum();
    result = 37 * result + getMinimum();
    result = 37 * result + getOrientation();
    result = 37 * result + (getToolTipText() == null ? 0 :
                              getToolTipText().hashCode());
    result = 37 * result + (getPaintTicks() ? 1 : 0);
    result = 37 * result + (getPaintTrack() ? 1 : 0);
    result = 37 * result + (getInverted() ? 1 : 0);
    result = 37 * result + (getSnapToTicks() ? 1 : 0);
    result = 37 * result + (getPaintLabels() ? 1 : 0);
    result = 37 * result + getMajorTickSpacing();
    result = 37 * result + getMinorTickSpacing();

    return result;
}

public void copyAttributes(ICellComponent source) {
    ULCCellEditableSlider sourceSlider = (ULCCellEditableSlider)source;

    setBackground(sourceSlider.getBackground());
    setFont(sourceSlider.getFont());
    setForeground(sourceSlider.getForeground());
    setMaximum(sourceSlider.getMaximum());
    setMinimum(sourceSlider.getMinimum());
```

```
        setOrientation(sourceSlider.getOrientation());
        setToolTipText(sourceSlider.getToolTipText());
        setPaintTicks(sourceSlider.getPaintTicks());
        setPaintTrack(sourceSlider.getPaintTrack());
        setInverted(sourceSlider.getInverted());
        setSnapToTicks(sourceSlider.getSnapToTicks());
        setPaintLabels(sourceSlider.getPaintLabels());
        setMajorTickSpacing(sourceSlider.getMajorTickSpacing());
        setMinorTickSpacing(sourceSlider.getMinorTickSpacing());
    }

    protected String typeString() {
        return "sample.client.UICellEditableSlider";
    }
}
```

On the client side, we extend *UISlider*. It implements *com.ulcjava.base.client.IEditorComponent* interface *and getTableCellEditor()* method that returns an instance of *MyTableCellEditor*, which implements Swing's *TableCellEditor*.

```
public class UICellEditableSlider extends UISlider implements
                          com.ulcjava.base.client.IEditorComponent {
    private TableCellEditor fTableCellEditor;

    protected void postInitializeState() {
        super.postInitializeState();
        fTableCellEditor = new MyTableCellEditor();
    }

    public ComboBoxEditor getComboBoxEditor() {
        throw new UnsupportedOperationException();
    }

    public TableCellEditor getTableCellEditor() {
        return fTableCellEditor;
    }

    public TableTreeCellEditor getTableTreeCellEditor() {
        throw new UnsupportedOperationException();
    }

    public TreeCellEditor getTreeCellEditor() {
        throw new UnsupportedOperationException();
    }

    private class MyTableCellEditor extends AbstractCellEditor
                                    implements TableCellEditor {
        public Component getTableCellEditorComponent(JTable table,
                    Object value, boolean isSelected, int row, int column) {
            getBasicSlider().setValue(((Integer)value).intValue());
            return getBasicSlider();
```

```
        }


        public Object getCellEditorValue() {
            return new Integer(getBasicSlider().getValue());
        }
    }
}
```

5.7.5 Using the custom renderer and editor

Now we can use the *ULCCellRenderableProgressBar* and *ULCCellEditableSlider* as follows:

```
…
ULCTableColumn progressColumn =
       table.getColumnModel().getColumn(MyTableModel.PROGRESS_COLUMN);
progressColumn.setCellRenderer(
       new ProgressCellRenderer(0, model.getRowCount() - 1));
progressColumn.setCellEditor(
       new SliderCellEditor(0, model.getRowCount() - 1));
…
    private static class ProgressCellRenderer
       extends ULCCellRenderableProgressBar implements ITableCellRenderer {
        public ProgressCellRenderer(int minimum, int maximum) {
             setMinimum(minimum);
             setMaximum(maximum);
        }

        public IRendererComponent getTableCellRendererComponent(
             ULCTable table, Object value, boolean isSelected,
             boolean hasFocus, int row) {
            setBackground(isSelected ? table.getSelectionBackground() :
                 table.getBackground());

            return this;
        }
    }

    private static class SliderCellEditor extends ULCCellEditableSlider
              implements ITableCellEditor {
        public SliderCellEditor(int minimum, int maximum) {
            setMinimum(minimum);
            setMaximum(maximum);
            setPaintTicks(true);
            setPaintLabels(true);
            setPaintTrack(true);
            setSnapToTicks(true);
            setMajorTickSpacing(1);
        }

        public IEditorComponent getTableCellEditorComponent(
                          ULCTable table, Object value, int row) {
            return this;
```

```
        }
    }
```

5.8 Integrating a Container

As described in chapter 4.6 in more detail, a basic component is decorated with an instance of *JXLayer* in case an extended visual effect is set on the corresponding server-side *ULCComponent*. The support for extended visual effects leads to the following rules a developer integrating a container has to adhere to:

1. The client-side container extension must use *UIComponent.getBasicComponentForContainer* instead of *UIComponent.getBasicComponent* in order to get the correct component to be added to or removed from the basic container. This rule is automatically ensured by *UIContainer*, if the client-side container extension adds and removes child components using the inherited methods *addChildComponent*, *addComponent*, *removeChildComponent*, and *removeComponent*. The same holds for setting layout constraints as in *UIContainer.addChildConstraints*. Please note, that formerly client-side container extensions did not need methods to add and remove components as in this case ULC invokes the corresponding methods directly on the basic container. With extended visual effects however, this is no longer possible and corresponding methods need to be introduced on the client-side extension. If this rule is not fulfilled, setting extended visual effects on the container's children do not have any effect.
2. The client-side container extension must internally store the layout constraints used when adding a component. Only then, a component can first be decorated by a *JXLayer* and then be replaced within the container using the exact same layout constraints. Again, this rule is ensured by *UIContainer* if the client-side container extension adds and removes child components using the inherited methods *addChildComponent*, *addComponent*, *removeChildComponent*, and *removeComponent*.

References

[1] Jakarta Tomcat servlet container
http://jakarta.apache.org/tomcat/

[2] Java Web Start
http://java.sun.com/products/javawebstart/

[3] J2EE tutorial on web application archives
http://java.sun.com/j2ee/tutorial/doc/WCC3.html

[4] Eclipse
http://www.eclipse.org/

[5] Jakarta ORO regular expression package
http://jakarta.apache.org/oro/

[6] HTML 4.0.1 specification
http://www.w3.org/TR/html4/

[7] Java Secure Socket Extension (JSSE) 1.0.3
http://java.sun.com/products/jsse/index-103.html

[8] Java Security Architecture
http://java.sun.com/j2se/1.3/docs/guide/security/index.html
http://java.sun.com/j2se/1.4.1/docs/guide/security/index.html

[9] JNLP forum thread describing the protocol handler workaround
http://forum.java.sun.com/thread.jsp?forum=38&thread=71335

[10] Java Plug-In tags
http://java.sun.com/products/plugin/1.3/docs/tags.html

http://java.sun.com/products/plugin/versions.html

[11] Using the conventional APPLET tag with Java Plug-In 1.3.1 and 1.4
http://java.sun.com/products/plugin/1.3.1_01a/faq.html
http://java.sun.com/j2se/1.4/docs/guide/plugin/developer_guide/applet_tag.html

[12] "usePolicy" Permission
http://java.sun.com/products/plugin/1.3/docs/netscape.html#use

[13] LiveConnect
http://developer.netscape.com/docs/technote/javascript/liveconnect/liveconnect_rh.html

[14] Calling an applet from Java Script
http://java.sun.com/products/plugin/1.3/docs/jsobject.html.

[15] InstallAnywhere
http://www.installanywhere.com/

[16] InstallShield
http://www.installshield.com/

[17] Initializing an initial context with applet parameters
http://java.sun.com/j2se/1.3/docs/api/javax/naming/Context.html#APPLET

[18] Canoo Engineering AG contact address
ulc-info@canoo.com

[19] Introduction to servlet technology
http://java.sun.com/products/servlet/index.html

[20] Servlet specifications
http://java.sun.com/products/servlet/download.html