

**AN EXTENSIBLE JAVA™
USER INTERFACE
FRAMEWORK
FOR CONTROLLING
DISTRIBUTED SYSTEMS**

Glenn Eychaner

Interferometry Systems and Technology
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive, M/S 171-113
Pasadena, CA 91109-8099
Glenn.Eychaner@jpl.nasa.gov

OBJECTIVE

Develop, deploy, and maintain graphical user interfaces (GUIs) for a number of distributed systems (in this case, interferometer test beds) consisting of:

- Remote objects (used to command the systems)
- Telemetry streams (carrying data from the systems)

Since each distributed system has different objects and telemetry, each GUI requires custom control panels and telemetry displays. In addition, a particular distributed system may require customization of other aspects of the GUI (e.g. the processing of incoming telemetry).

IMPLEMENTATION

Despite their superficial differences, the user interfaces for the distributed systems will contain a lot of common functionality, and are built on a unified *framework*.

The framework is itself nearly a complete GUI, and can be run independently of the code for a particular system. It creates, displays, destroys, and otherwise manages the GUI elements (control panels and telemetry displays) and other custom code specified by a particular system.

The custom classes for a particular system implement abstract classes provided by the framework. These

implementations are specified to the framework at runtime by class name (on the command line or in an initialization file), and are then loaded and instantiated dynamically by the framework. The framework itself may provide some implementations.

The abstract classes in the framework allow methods, fields, or properties of arbitrary objects (e.g. the remote objects for commanding the system) to be connected to arbitrary GUI elements (e.g. buttons, input fields, and pull-downs) by introspecting the objects at run-time.

This flexibility is not without cost; the framework sections that interact with the distributed system must be carefully designed to permit the runtime customizations.

EXAMPLE: CONTROLLING A REMOTE OBJECT

Consider a distributed system that contains a simple camera. The remote object that represents the camera (through which the camera is controlled, as written in CORBA™ IDL) might look like:

```
interface Camera {  
    void On();  
    void Off();  
    void TakePicture(double exposure);  
}
```

A control panel for this camera (generated using the drag-and-drop GUI designer in the Borland[®] JBuilder[™] development environment) might look like:

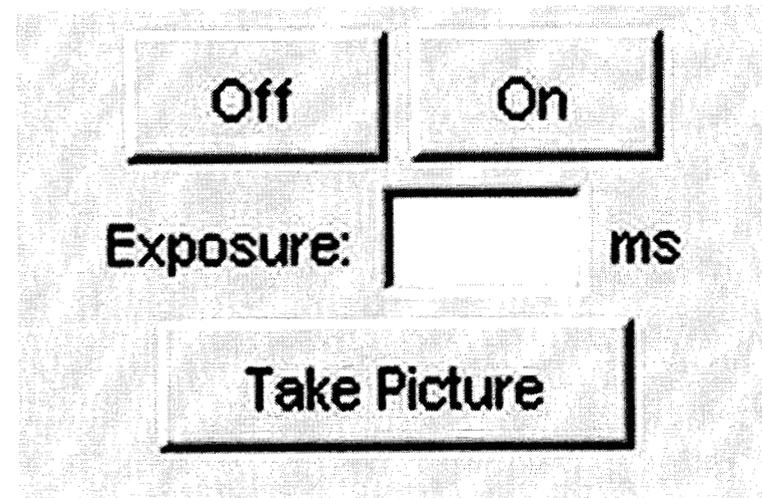
```
import java.awt.*;
import javax.swing.*;

public class CameraPanel extends JPanel {
    FlowLayout flowLayout1
        = new FlowLayout();
    JButton jButtonOff = new JButton();
    JButton jButtonOn = new JButton();
    JLabel jLabel1 = new JLabel();
    JTextField jTextFieldExposure
        = new JTextField();
    JLabel jLabel2 = new JLabel();
    JButton jButtonTakePic = new JButton();

    public CameraPanel() {
        try {
            jbInit();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    void jbInit() throws Exception {
        jButtonOff.setText("Off");
        this.setLayout(flowLayout1);
        jButtonOn.setText("On");
        jLabel1.setText("Exposure:");
        jTextFieldExposure.setText("");
```

```
jTextFieldExposure.setColumns(5);
jLabel2.setText("ms");
jButtonTakePic.setText("Take Picture");
this.add(jButtonOff, null);
this.add(jButtonOn, null);
this.add(jLabel1, null);
this.add(jTextFieldExposure, null);
this.add(jLabel2, null);
this.add(jButtonTakePic, null);
    }
}
```



To display the control panel, an object that implements the abstract framework class `Commander` must be created and specified to the framework at initialization:

```
public class CameraCommander extends Commander {
    protected ControlSet createControlSet() {
        return new ControlSet(this) {
            private CameraPanel panel = new CameraPanel(this);
            private CameraMenu menu = new CameraMenu(this);

            protected JMenu getCommandMenu() {
                return menu();
            }

            protected JComponent getMainControlPanel() {
                return panel;
            }
        };
    }
}
```

(Note that a menu can be displayed in addition to the panel; the menu implementation is not shown.)

[One detail has been left out of the `CameraCommander` class shown above; it must also connect to and disconnect from the remote camera object in addition to creating and returning the control panels.

However, connecting to the remote object is specific to a distributed system's architecture and not particularly interesting. In an actual GUI based on the framework, the superclass of `CameraCommander` would extend `Commander`, and implement the `connect()` and `disconnect()` methods specific to the architecture. This superclass may be provided as part of the framework (as it is for a CORBA-based system). Once the remote object is connected, the framework manages it internally, and no further effort must be expended.]

The buttons in the `CameraPanel` are configured to call the methods of the remote camera object (and the input field similarly sets the input to the `takePicture()` method) by modifying the panel's constructor.

```
public CameraPanel(ControlSet c) {
    try {
        jButtonOn.setAction(c.bindMethod("On"));

        jButtonOff.setAction(c.bindMethod("Off"));

        ExecuteAction a = c.bindMethod("TakePicture");
        a.bindInput (jTextFieldExposure, 0);
        jButtonTakePic.setAction(a);

        jbInit();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

It seems simple only because the framework does the real work. The `bindMethod()` methods, provided by the framework, introspect the remote object for the named method and return an `Action` that calls the remote object's method when the button is pushed.

Similarly, the `bindInput()` method, also provided by the framework, introspects the input field and the remote object's method. The framework creates temporary objects that set the remote method's parameters from the input field (with appropriate type conversion).

Using introspection, the framework can connect arbitrary GUI elements to arbitrary objects, creating a flexible GUI while minimizing the amount of custom code.

ACKNOWLEDGEMENT

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

NOTICES

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries. CORBA is a registered trademark of Object Management Group, Inc. in the United States and/or other countries. Borland Trademarks and logos are trademarks or registered trademarks of Borland Software Corporation in the U.S. or other countries and are used under license.

[Poster layout by page]

	1	
2	3	4
5	6	7
8	9	10
	11	