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

ABSTRACT

Graphical user interfaces (GUIs) are the preferred method of interacting with a distributed system such as a spacecraft ground system. While GUIs are often difficult to develop and maintain, the capabilities of the Java language and JavaBeans™ component architecture can be used to create a generic framework that allows easy and rapid development, deployment, and maintenance of user interfaces to control a variety of distributed systems.

1. INTRODUCTION

To control a distributed system, a user typically interacts with a GUI that sends commands to and displays telemetry from the system. These GUIs are complex, yet are often custom built for each new system to be controlled at considerable expense. In creating GUIs for a set of interferometer test beds, we are developing a Java framework that all the systems can share. Dynamic class loading, reflection, and other run-time capabilities of the Java language and JavaBeans component architecture allow the GUI for each system to be created with a minimum of custom development. The framework imposes a minimal set of requirements on the systems, yet allows them considerable flexibility to extend its capabilities.

Using the framework, GUI components in control panels and menus can send commands to a particular distributed system with a minimum of system-specific code. The framework receives, decodes, processes, and displays telemetry data; custom telemetry data handling can be added for a particular system. In addition, the framework supports saving and later restoring user configurations of control panels and telemetry displays with a minimum of effort in the system-specific code.

While this framework is currently written in Java Swing and used in a CORBA® (Common Object Request Broker Architecture) environment, the techniques and overall design of the framework can be easily adapted to other Java GUI toolkits (such as AWT or SWT) and other distributed system architectures. The framework is written entirely in the Java language so that GUIs can be deployed to any operating system with a Java Runtime Environment without recompilation or code changes. The framework is fully multi-threaded, with performance comparable to applications written in platform-dependent languages.

This paper will describe the framework's interaction with the system-specific code as well as some of its internals, and contains a number of Java code samples. For the sake of brevity, these samples omit details of the actual implementation (such as scope qualifiers of class and method definitions). The code for the working system can be made available on request.

2. COMMANDING A DISTRIBUTED SYSTEM

First, let us examine how the framework enables a user to send commands to a distributed system. The framework requires the system to expose its command interface as a set of objects; connecting to these objects and calling their methods executes the commands. This allows the framework to leverage existing Java functionality for dynamically creating, manipulating and introspecting objects. If a distributed system uses a Java-compatible object-oriented architecture, it will probably already exist as a set of distributed objects, and the commands can be executed by calling methods of the distributed objects directly. In a CORBA-based system, for example, the interfaces to the distributed objects are written in a language-neutral Interface Definition Language (IDL). Tools supplied with the CORBA ORB (Object Request Broker) being used (an ORB is supplied with the Java Runtime Environment) enable the framework to interact with them as Java objects.

If an object-oriented architecture is not already being used, exposing the system's commands to the framework may

require the creation of adapter objects. The framework can execute commands by calling methods of these adapter objects that translate the commands to the format required by the system. These adapter objects are generally simple, and can often be generated automatically from templates (just as the CORBA IDL compiler generates the Java code for objects from the IDL files).

The system may expose its commands as the methods of any number of objects. However, it is recommended that the classes of these objects follow established object-oriented development paradigms and patterns, with each class containing a related set of commands for manipulating a part of the distributed system. While the framework does not permit method overloading, the classes may extend or implement any other classes and their methods make take any parameters and return any values desired.

Once a system's commands are available as methods of objects, we need to make those command objects visible to the framework. For each class of command object, an implementation of the abstract framework class `Commander` must be created:

```
class Commander {
    ControlSet createControlSet();
    String getObjectLocation();
    void setObjectLocation(String s);
    Object connect();
    void disconnect(Object o);
    boolean isConnected();
}
```

The `Commander` class contains methods for setting the location of, connecting to, and disconnecting from the command object. The implementation of the connection between the framework and the distributed system is specific to a particular architecture and not particularly interesting; the methods are implemented in the superclass of the `Commander` classes for a particular system, which may be provided as part of the framework (as it is for a CORBA-based system).

3. DISPLAYING CONTROLS

Now we need to create the user interfaces for interacting with the `Commander` objects. The user interface for a `Commander` object is encapsulated in the `ControlSet` object returned by the `createControlSet()` method:

```
class ControlSet {
    JComponent getMainControlPanel();
    JMenu getCommandMenu();
}
```

A `ControlSet` object contains a control panel and a menu. The control panel can contain buttons that send commands to the system, input components (text fields, combo boxes, checkboxes, etc.) that set the parameters of the methods, and output components (labels, text areas, etc.) that display the return values of the commands. The menu can likewise contain menu items that send commands to the system, set input parameters, or display return values. The control panel and menu can be created using any desired tool, such as the drag-and-drop GUI editor included in many Java development environments. The framework displays the control panel in a window and adds the menu to an existing menu bar (along with menu items to reconnect to or disconnect from the command object) when the `Commander` object is created.

The `ControlSet` object is responsible for connecting the GUI components to the methods of the command object and their input parameters and return values. This is where the real power of the framework lies; each GUI component can be configured to call a method, set an input parameter of a method, or display a method's return value with only one or two lines of code.

4. EXECUTING COMMANDS

To demonstrate the power of the framework, let us consider a user interface for controlling a camera. The distributed system contains an object with a `takePicture()` method, which will command the camera to take a picture. The `ControlSet` object returned by the camera's `Commander` object contains a control panel with a single button and a menu with a single menu item:

```
class CameraControl extends ControlSet {
    JPanel ourPanel = new JPanel();
    JMenu ourMenu = new JMenu();

    CameraControl() {
        ourPanel.add(new JButton());
        ourMenu.add(new JMenuItem());
    }

    JComponent getMainControlPanel() {
        return ourPanel;
    }

    JMenu getCommandMenu() {
        return ourMenu;
    }
}
```

To configure the button and menu item to call the `takePicture()` method of the camera, we call a method of the `ControlSet` object:

```
ExecuteAction bindMethod(String method);
```

The `method` argument is the name of the method of the command object to be called, and the returned `Action` object can be set as the action of the button or menu item. (Java GUI components that perform an action, such as

buttons and menu items, typically have a `setAction(Action)` method.) When the action of the button or menu item is performed (i.e. the button is pressed or the menu item is selected by the user), the method of the command object will be called by the action. We can add this to the `CameraControl` class's constructor:

```
CameraControl() {
    ExecuteAction a =
            bindMethod("takePicture");
    ourPanel.add(new JButton(a));
    ourMenu.add(new JMenuItem(a));
}
```

It seems simple because the real work is taking place in the framework; in particular, inside the `ExecuteAction` object. To allow the `ExecuteAction` object to access the command object (the camera), the framework insures that each object contains an internal reference to the object that constructed it. Thus, the `ExecuteAction` object has an internal reference to the `CameraControl` object that constructed it, which has a reference to the `Commander` object that constructed it, which has a reference to the command object (once it has been connected). By delegating up the chain, the `ExecuteAction` object can access the command object. All this is handled internally, and is not exposed to the system-specific code.

To call the method named in the `bindMethod()` call, the `ExecuteAction` object first uses introspection on the command object to search for a method with the requested name. The resulting `Method` object is stored in the `ExecuteAction` object. (If no matching method is found, the action is disabled and an error message is output.) To avoid the overhead of finding the `Method` object every time the action is performed, it is located once when the `Commander` object is connected (or when the `ExecuteAction` object is created if the `Commander` object is already connected).

When action is performed, the `actionPerformed()` method of the `ExecuteAction` object calls the `invoke()` method of the `Method` object, executing the method of the command object. This takes place in a separate thread so as not to interfere with the operation of the GUI. (Swing, like most GUI toolkits, uses a single thread to handle user-generated events such as button presses; if the `invoke()` method were called in this thread, the GUI would become unresponsive until it returned.) The distributed system thus receives and executes the command.

5. INPUT PARAMETERS

Merely connecting a method of a command object to a button or menu item is only sufficient if the method has no parameters. If this is not the case, however, the framework can connect other GUI components in the control panel to the input parameters of the method. Suppose the `takePicture()` method has the exposure time as its only input parameter:

```
void takePicture(double exposure);
```

If a text field for inputting the exposure time is added to the control panel, a method of the `ExecuteAction` object can configure the text field to set the input parameter:

```
void bindInput(JTextField t, int param);
```

This sets input parameter number `param` in the method's parameter list (the first parameter is number zero) from the text in text field `t` each time the `ExecuteAction` object's action is performed. This adds a few more lines to the `CameraControl` class's constructor:

```
CameraControl() {
    ExecuteAction a =
            bindMethod("takePicture");
    JTextField t = new JTextField();
    a.bindInput (t, 0);
    ourPanel.add(t);
    ourPanel.add(new JButton(a));
    ourMenu.add(new JMenuItem(a));
}
```

Again, it seems simple because the real work is taking place inside the framework. The `bindInput()` method shown above delegates internally to this more general method of the `ExecuteAction` object:

```
void bindInput(JComponent c,
               String property,
               int param,
               Format format);
```

This sets the specified input parameter from any property of any GUI component. (A property is the JavaBeans design pattern for reading and writing an object's internal state through special *getter* and *setter* access methods.) In our camera example, the first input parameter of the `takePicture()` method will be set from the "text" property of the text field (which contains the text typed into the field).

When setting an input parameter of a method, the framework automatically handles conversions between Java primitive types or `String` objects. If the input parameter is not one of these types (that is, it is any non-string object), an optional `Format` object can be supplied to the `bindInput()` method as shown above to convert the value returned from the input component to an object of the type required.

The framework also has a more powerful mechanism for handling input parameters that are objects. Suppose the

`takePicture()` method of our imaginary camera has a single parameter of a class `PictureParams`:

```
void takePicture(PictureParams p);

class PictureParams {
    double exposureTime;
    double fStop;
    boolean compress;
}
```

We can use an alternate form of the `bindInput()` method to cause any property of any input component to set any field (or subfield) of any input parameter when the action is performed:

```
void bindInput(JComponent c,
               String property,
               String field);
```

The field is specified in standard Java notation, beginning with an array index indicating the parameter number. Thus, to cause our text field from the `CameraControl` constructor above to set the `exposureTime` field of the first (and only) input parameter of the `takePicture()` method, only a small change to our constructor is required:

```
a.bindInput (t, "[0].exposureTime");
```

The string designating the field is parsed and used to recursively introspect the input parameter to find the indicated field. The input parameter may contain objects or arrays at any depth; the string `"[0].a[1].b"` will bind to field `b` of the second element of array field `a` of the first input parameter. Intermediate objects and arrays are constructed dynamically as necessary, and must have public constructors that take no parameters (this is the default constructor in Java if none is specified in a class).

The implementation of all this introspection is, of course, much more complex than the description; the framework code responsible is considerably longer than this paper. However, this complexity is all hidden from the implementers of the specific systems; they only need to know how to use the `bindInput()` methods, not how they work internally.

Thus far, we have ignored the menu item in this discussion. Since, as we have implemented it above, it is using the same `ExecuteAction` object as the button, it will have the same behavior; it will set the `takePicture()` method's input parameter from the text field in the control panel and then call the method. However, suppose we instead want it to prompt the user for the exposure time with a dialog containing a text field. First, we will have to configure the menu item with a separate `ExecuteAction` object, so that it can execute differently from the button. Second, we will have to create a `JOptionPane` object containing a text field for the dialog, and bind the new text field in the same way as the previous one. Finally, we will have to instruct the menu item's `ExecuteAction` object to display the dialog before getting the input values from its components using this method of the `ExecuteAction` object:

```
void setInputOptionPane(JOptionPane p);
```

The framework will then automatically display the option pane in a dialog when the action is performed, before setting the input parameters. The constructor for the `CameraControl` class now looks like this:

```
CameraControl() {
    ExecuteAction a =
            bindMethod("takePicture");
    JTextField t = new JTextField();
    a.bindInput (t, 0);
    ourPanel.add(t);
    ourPanel.add(new JButton(a));

    ExecuteAction a2 =
            bindMethod("takePicture");
    JTextField t2 = new JTextField();
    a2.bindInput (t2, 0);
    JOptionPane p = new JOptionPane(t2);
    a2.setInputOptionPane(p);
    ourMenu.add(new JMenuItem(a2));
}
```

We now have a control panel and menu that can command our camera, and the finished code is not particularly complicated. The framework is doing much of the work for us, allowing GUIs to be deployed to many systems, even those undergoing rapid development, easily and quickly.

6. RETURN VALUES

As one would expect, displaying the return values from the command object's methods is very similar to setting the input parameters. The `ExecuteAction` object contains methods for setting a property of a GUI component from the return value and setting a dialog to be displayed when the method of the command object returns:

```
void bindOutput(JComponent c,
                String property,
                Format format);
void setOutputOptionPane(JOptionPane p);
```

These work in a manner very similar to the methods for setting the input parameters. When the method of the command object returns, the return value is used to set the indicated property of any connected GUI component, and if an option pane has been set, it is displayed to the user as a dialog. (The framework also transparently handles the threading issues of the Swing threading model.)

7. ADDITIONAL CONTROL PANELS

It is sometimes useful for a command object's controls to be displayed in multiple control panels. For example, a set of related, but seldom used, controls can then be displayed in a second control panel apart from the main control panel, and this control panel can be opened and closed as needed. To support this, the `ExecuteAction` object has a method that creates an `Action` object to display an additional control panel:

```
Action bindControlPanel(JComponent pnl);
```

The action of a button or menu item can then be set to this `Action` object. The framework will handle placing the panel in a window and displaying it to the user when the action is performed.

8. CREATING THE `COMMANDER` OBJECTS

We have seen how the `Commander` classes are defined, how they create controls to be displayed to the user, and how those controls send commands to the distributed system. Now we have to answer the question of how the `Commander` objects are created by the framework in the first place. Fortunately, this is relatively simple, at least in terms of the system-specific code. Unfortunately, it adds some additional requirements to the distributed system.

In the current operation of the framework, the distributed system is required to provide a list of all the locations of objects available to be commanded. The GUI is configured at startup (in a system property, an initialization file, or the command line) with the location of this list. For example, in the current CORBA framework, all the available objects are entries in a Naming Service. The GUI is given the location of the Naming Service; the objects there can be easily listed. The GUI is also configured with a list of all the available implementations of the `Commander` class (the actual implementation class names). Finally, each implementation of the `Commander` class must have a public constructor that takes no arguments.

To connect to an object, the user selects a location and a corresponding `Commander` implementation from a dialog. Once a location and `Commander` implementation have been selected, a factory object in the framework dynamically loads the selected `Commander` class by name and constructs a new instance of the class. The factory then passes the location to the `setObjectLocation()` method of the `Commander`, and connects the `Commander` to the object by calling the `connect()` method.

The user could be spared from having to select the `Commander` implementation by having the framework connect to the object to be commanded and discover its type before creating the appropriate `Commander` object. However, this would make changing the implementation for connecting to the command objects more difficult, and also make commanding heterogeneous systems considerably more complicated. Requiring the user to select the `Commander` implementation has not been an issue in the GUIs deployed so far.

The framework then displays the main control panel and menu for the connected object, and the user can interact with it. Once the user has all the desired control panels open, the state of the entire GUI can be saved to a file. This state can be restored the next time the GUI is started by specifying the file name on the command line; the GUI will then reconnect to all the objects and redisplay all the control panels exactly as they were before. This greatly increases the ease of use of the GUI by allowing individual customization of the control panel layout.

9. PERSISTENCE

Implementations of the `Commander` class need not contain any code to support saving and restoring the GUI's state; the persistence of the GUI is handled entirely within the framework. A careful examination of the framework reveals how this is accomplished. First, the connections to the distributed system can be saved and restored by saving the locations the user has connected and the name of the `Commander` implementation each location is using. Second, the `Commander` implementations only provide the contents of the control panel windows and menus. The windows and menus themselves are created and displayed by the framework and managed internally, so that their size, position, and other state can be saved and restored entirely within the framework. Finally, the framework tracks the bindings of the control panel components made by the implementation. Since these bindings encompass all the relevant properties of the control panels that can be changed by the user, and the bindings give easy access to the properties, saving and restoring them can be handled entirely within the framework.

10. TELEMETRY DISPLAYS

In addition to enabling the user to send commands to the system, the GUI must also display telemetry from the system. While the current telemetry framework has some flexibility, it is not as advanced as the command framework previously described and requires more system-specific code and effort. However, it still contains a number of useful abstractions that I will briefly describe.

Just as the implementations of the `Commander` class can be specified as part of the configuration of the GUI at startup, so can the telemetry display classes be specified. In the simplest design, each telemetry display class must

extend the JComponent class so that it can be displayed as the contents of a frame, have a constructor that takes no arguments, and implement an interface that allows it to receive telemetry data:

```
interface Receiver {
    void handleData (ValueEvent e);
    String getName();
}
```

The display object can manipulate the data internally in any way it sees fit and display it to the user. If the display object is not capable of displaying the telemetry it is receiving (for example, it does not know how to handle the data type, such as an image sent to a text display), it should display a useful error message to the user. The framework itself contains common display classes, such as an image viewer, a strip chart, and a table for displaying text or numeric data in rows and columns.

In actual practice, the problem is more complicated. For example, some telemetry displays might be able to display multiple channels simultaneously (such as a strip chart) while others can only display a single channel in a useful manner (such as an image display). Also, some telemetry displays often have internal state that is not immediately visible to the framework, but can adjusted by the user and must be persisted along with the rest of the user interface. For example, a strip chart should allow the user to adjust the bounds of the axes and the colors of the individual traces. The current framework requires telemetry displays to implement an extensive interface, but provides a number of helper classes to which system-specific displays can delegate the implementation of the interface. We are currently working on a more general solution.

11. TELEMETRY DATA

To receive telemetry, the framework requires some standardization of the data. It expects that the telemetry data will be sent from the system as one or more data streams, and requires that the telemetry from each element of the system can be uniquely identified. The stream of data being sent from an individual element of the system is referred to as a telemetry channel. The framework requires the data to be sent in packets; each packet must include a description of the type of data contained within it as well as the data itself.

For example, in the current CORBA implementation, the telemetry is sent using asynchronous CORBA Event Channels. The GUI requests data from a particular channel by creating a PushConsumer object that connects itself to the event channel. As the distributed system produces data, the push(Any) method of the PushConsumer object will be called by the distributed system to send the data to the GUI. The data arrives as CORBA Any objects, which automatically contain both the data they were created to hold and a TypeCode object describing the contents of the Any object.

A different telemetry system, such as telemetry sent directly over a network socket, would require a different Consumer implementation in the same way that a different command system requires a different connection implementation in the Commander superclass. The Consumer implementation for a network socket might need to filter the data if multiple system elements (i.e. multiple channels) send data over the same socket. The framework around the Consumer and Commander objects that handles their creation, destruction, and persistence remains the same, however.

The Consumer objects also act as local multiplexers for the telemetry data to minimize the load on the distributed system. Only one Consumer object is created for each channel. If more than one telemetry display is subscribed to the same channel (the displays might be of different types, such as a text display and a graph), they are both connected to the same Consumer object by the framework.

Just as it must make a list of command locations available to the framework, the system must also make a list of telemetry channels available, and the GUI is configured with the location of this list at startup. (In the CORBA implementation, the system includes an object in the Naming Service that can be queried for this list.) The user selects a channel from the list and a display type. The framework then creates the display object of that type (in exactly the same way a Commander is created), displays it in an internally managed frame, and connects it to the Consumer for the channel.

12. UNPACKING TELEMETRY

In discussing the telemetry displays, we specified that they would receive telemetry data as ValueEvent objects without discussing how they are created from the incoming packets. In fact, the ValueEvent objects are just convenient wrappers created around the data packets. The Consumer object creates a value event for each data packet received and passes it to the displays. The ValueEvent object has a getData() method that unpacks the data in the packet, caches it internally, and returns it. Since Consumer objects send the same ValueEvent object to all the displays subscribed to a channel for each packet received, this insures that a given packet is only unpacked at most once, and is not unpacked at all if it is not displayed, maximizing the performance of the GUI.

The actual work of unpacking the telemetry packet is done by a set of factories; the classes of these factories, as might be expected, are supplied to the GUI in its configuration at startup, and each factory class must have a no-argument constructor. Each must also implement an interface that allows the framework to find out what types of packets it can unpack and to perform the unpacking:

```
interface TelemetryFactory {
    boolean canCreateValue(Object type);
    Value createValue(Object data);
}
```

The parameters for the factory methods must be as general as possible, and therefore are of class `Object`. The `ValueEvent` objects are created with a reference to the array of available factory objects by the framework, and can then delegate the unpacking of the data packet to the appropriate factory.

Scanning the factories for each data packet to be unpacked introduces extra overhead in the telemetry system. However, the `canCreateValue()` methods of the factories will be simple enough that this overhead will be minimal. Scanning the factories allows the telemetry framework not to need to know what type of data is going to be sent on a channel until data actually starts arriving. Scanning also allows a channel to carry more than one type of data, which is useful for sending errors without using a second channel or having special "out-of-bounds" values.

The factories return the unpacked telemetry as `Value` objects. These objects encapsulate the data and provide convenience methods that allow the telemetry displays to access the unpacked data in commonly desired forms:

```
interface Value {
    ArrayValue asArrayValue();
    double doubleValue();
    Object getValue();
    Number numericValue();
    String stringValue();
    String toString(Format[] formats);
}

interface ArrayValue extends Value {
    Object getValue(int index);
    Value value(int index);
    Value[] values();
    Number numericValue (int index);
    Number[] numericValues();
    double doubleValue(int index);
    double[] doubleValues();
    int length();
}
```

To add support for a custom telemetry data type, implementations of the `Value` interfaces that encapsulate the raw data and a factory that can unpack the data from a packet must be created. The framework provides implementations for numeric values and arrays, strings, images, and other common data types, as well as factories to unpack them from CORBA `Any` packets.

13. CONCLUDING REMARKS

Dynamic class loading, reflection, and other run-time capabilities of the Java language and JavaBeans component architecture can be used to create a framework that allows rapid development, wide deployment, and easy maintenance of system-specific user interfaces, allowing the developer to focus on the distributed system rather than the user interface. In particular, determining the class of a few key objects in the framework at run-time allows the framework to be easily modified to work with a wide variety of distributed systems. The requirements imposed on the systems by the framework need not be very extensive, and various adapter strategies can be used to assist in meeting these requirements. Though the relative simplicity of the system-specific code comes at the price of complexity in the shared framework, the powerful capabilities of the Java language to create a generic, adaptable framework reduce the effort necessary to build and maintain the framework considerably.

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