CHAPTER 19

User Interfaces for Jini Services

SOME EARLIER CHAPTERS HAVE USED CLIENTS with graphical user interfaces to services. Clients may not always know which is the most appropriate user interface, and sometimes may not even know of any suitable user interface. Services should be able to define their own user interfaces, and the question of how they should best do this is explored in this chapter. We'll also look at how clients can discover, download, and use these user interfaces.

User Interfaces as Entries

Interaction with a service is specified by its interface, and the interaction will be the same across all implementations of the interface. This consistency doesn't allow any flexibility in using the service, since a client will only know about the methods defined in the interface. The interface is the defining level for using this type of service.

However, services can be implemented in many different ways, and service implementations do in fact differ. For example, one service may be offered on a "take it or leave it" basis, while another might have a warranty attached. This does not affect how the client calls a service, but it may affect whether or not the client wants to use one service implementation or another. There is a need to allow for this, and the mechanism used in Jini is to put these differences in `Entry` objects. Typical objects supplied by vendors may include `Name` and `ServiceInfo`.

Clients can make use of the type interface and these additional entry items primarily in the selection of a service. But once clients have the service, are they just constrained to use it via the type interface? The type interface is designed to allow a client application to use the service in a programmatic way by calling methods. However, many services could probably benefit from some sort of user interface (UI). For example, a printer may supply a method to print a file, but it may have the capability to print multiple copies of the same file. Rather than relying on the client to be smart enough to figure this out, the printer vendor may want to call attention to this option by supplying a user-interface object with a special component for the number of copies.

NOTE *In this chapter I talk about interfaces we have been using throughout the book, and also about user interfaces. To avoid possible confusion, in this chapter I will use the term type interface to refer to a Java interface as used in the rest of this book, and user interface for any sort of interaction with the user.*

A client can only be expected to know about the type interface of a service. If it uses this to build a user interface, then at best it could only manage a fairly generic one that will work for all service implementations. A vendor will know much more detail about any particular implementation of a service, and so the vendor is best placed to supply the user interface. In some cases, the service vendor may be unwilling or incapable of supplying user interfaces for a service, and a third party may supply it.

When your video player becomes Jini-enabled, it would be a godsend for someone to supply a decent user interface for it, since the video-player vendors seem generally incapable of doing so! The `Entry` objects are not just restricted to providing static data; as Java objects, they are perfectly capable of running as user-interface objects.

User interfaces are not yet part of the Jini standard, but the Jini community (with a semi-formal organization as the "Jini Community") is moving toward a standard way of specifying many things, including user-interface standards and guidelines. Guideline number one from the serviceUI group is this: user interfaces for a service should be given in `Entry` objects.

User Interfaces from Factory Objects

In Chapter 13, some discussion was given to the location of code, using user-interface components as examples. That chapter suggested that user interfaces should not be created on the server side but on the client side—the user interface should be exported as a factory object that can create the user-interface on the client side.

More arguments can be given to support this approach:

- A service exported from a low-resource computer, such as an embedded Java engine, may not have the classes on the service side needed to create the user-interface (it may not have the Swing or even the AWT libraries).

- There may be many potential user interfaces for any particular service: Palm handhelds (many with small grayscale screens) require a different interface than a high-end workstation with a huge screen and enormous numbers of colors. It is not reasonable to expect the service to create every one of these user interfaces, but it could export factories capable of doing so.

- Localization of internationalized services cannot be done on the service side, only on the client side.

The service should export zero or more user-interface factories, with methods to create the interface, such as `getJFrame()`. The service and its user-interface factory will both be retrieved by the client. The client will then create the user interface. Note that the factory will not know the service object beforehand; if the factory was given one during its construction (on the service side), the factory would end up with a service-side copy of the service instead of a client-side copy. Therefore, when the factory is asked for a user-interface (on the client side), it should be passed the service. In fact, the factory should probably be passed all of the information about the service, as retrieved in the `ServiceItem` from a lookup service.

A typical factory is the one that returns a `JFrame`. This is defined by the type interface as follows:

```
package net.jini.lookup.ui.factory;

import javax.swing.JFrame;

public interface JFrameFactory {
    String TOOLKIT = "javax.swing";
    String TYPE_NAME = "net.jini.lookup.ui.factory.JFrameFactory";

    JFrame getJFrame(Object roleObject);
}
```

The factory imports the minimum number of classes needed to compile the type interface. The `JFrameFactory` above needs to import `javax.swing.JFrame` because the `getJFrame()` method returns a `JFrame`. An implementation of this type interface will probably use many more classes. The `roleObject` passes any necessary information to the UI constructor. This is usually the `ServiceItem`, as it contains all the information (including the service) that was retrieved from a lookup service. The factory can then create an object that acts as a user interface to the service, and can use any additional information in the `ServiceItem`, such as entries for `ServiceInfo` or `ServiceType`, which could be shown, say, in an "About" box.

A factory that returns a visual component, such as a `JFrame`, should not make the component visible. This will allow the client to set the `JFrame`'s size and placement before showing it. Similarly, a "playable" user interface, such as an audio file, should not be in a "playing" state.

Current Factories

A service may supply lots of these user interface factories, each capable of creating a different user interface object. This allows for the differing capabilities of viewing devices, or even for different user preferences. One user may always like a Web-style interface, another may be content with an AWT interface, a third may want the accessibility mechanisms possible with a Swing interface, and so on.

The set of proposed factories currently includes the following:

- `DialogFactory`, which returns an instance of `java.awt.Dialog` (or one of its subclasses)
- `FrameFactory`, which returns an instance of `java.awt.Frame` (or one of its subclasses)
- `JComponentFactory`, which returns an instance of `javax.swing.JComponent` (or one of its subclasses, such as a `JList`)
- `JDialogFactory`, which returns an instance of `javax.swing.JDialog` (or one of its subclasses)
- `JFrameFactory`, which returns an instance of `javax.swing.JFrame` (or one of its subclasses)
- `PanelFactory`, which returns an instance of `java.awt.Panel` (or one of its subclasses)

These factories are all defined as interfaces. An implementation will define a `getXXX()` method that will return a user interface object. The current set of factories returns objects that belong to the Swing or AWT classes. Factories added in later iterations of the specification may return objects belonging to other user interface styles, such as speech objects. Although an interface may specify that a method, such as `getJFrame()`, will return a `JFrame`, an implementation will in fact return a subclass of this, which also implements a role interface.

Marshalling Factories

There may be many factories for a service, and each of them will generate a different user interface. These factories and their user interfaces will be different for each service. The standard factory interfaces will probably be known to both clients and services, but the actual implementations of these will only be known to services (or maybe to third-party vendors who add a user interface to a service).

If a client receives a `ServiceItem` containing entries with many factory implementation objects, it will need to download the class files for all of these as it instantiates the `Entry` objects. There is a strong chance that each factory will be bundled into a `jar` file that also contains the user interface objects themselves, so if the entries directly contain the factories, then the client will need to download a set of class files before it even goes about the business of deciding which of the possible user interfaces it wants to select.

This downloading may take time on a slow connection, such as a wireless or home network link. It may also cost memory, which may be scarce in small devices such as PDAs. Therefore, it is advantageous to hide the actual factory classes until the client has decided that it does in fact want a particular class. Then, of course, it will have to download all of the class files needed by that factory.

In order to hide the factories, they are wrapped in a `MarshalledObject`. This keeps a representation of the factory and also a reference to its codebase, so that when it is unwrapped, the necessary classes can be located and downloaded. Clients should have the class files for `MarshalledObject`, because this class is part of the Java core. By putting a factory object into entries in this form, no attempt is made to download the actual classes required by the factory until it is unmarshalled.

The decision as to whether or not to unmarshall a class can be made on a separate piece of information, such as a set of `String`s that hold the names of the factory class (and all of its superclasses and interfaces). This level of indirection is a bit of a nuisance, but not too bad:

```
if (typeNames.contains("net.jini.lookup.ui.factory.JFrameFactory") {
    factory = (JFrameFactory) marshalledObject.get();
    ....
}
```

A client that does not want to use a `JFrameFactory` will just not perform the preceding Boolean test. It will not call the unmarshalling `get()` method and will not attempt the coercion to `JFrameFactory`. This will avoid downloading classes that are not wanted. This indirection does place a responsibility on service-side programmers to ensure that the coercion will be correct. In effect, this is a maneuver to circumvent the type-safe model of Java purely for optimization purposes.

There is one final wrinkle when loading the class files for a factory: a running JVM may have many class loaders. When loading the files for a factory, you want to make sure that the class loader is one that will actually download the class files across the network as required. The class loader associated with the service itself will be the most appropriate loader for this.

UIDescriptor

An entry for a factory must contain the factory, itself, hidden in a `MarshalledObject` and some string representation of the factory's class(es). It may also need other descriptive information about the factory. The `UIDescriptor` captures all this:

```
package net.jini.lookup.entry;

public class UIDescriptor extends AbstractEntry {

    public String role;
    public String toolkit;
    public Set attributes;
    public MarshalledObject factory;

    public UIDescriptor();
    public UIDescriptor(String role, String toolkit,
                        Set attributes, MarshalledObject factory);

    public final Object getUIFactory(ClassLoader parentLoader)
           throws IOException, ClassNotFoundException;
}
```

There are several features in the `UIDescriptor` that we haven't mentioned yet, and the factory type appears to be missing (it is one of the `attributes`).

Toolkit

A user interface will typically require a particular package to be present or it will not function. For example, a factory that creates a `JFrame` will require the `javax.swing` package. These requirements can provide a quick filter for whether or not to accept a factory—if it is based on a package the client doesn't have, then it can just reject this factory.

This isn't a bulletproof means of selection. For example, the Java Media Framework is a fixed-size package designed to handle lots of different media types, so if your user interface is a QuickTime movie, you might specify the JMF package. However, the media types handled by the JMF package are not fixed, and they can depend on native code libraries. For example, the current Solaris version of the JMF package has a native code library to handle MPEG movies, which is not present in the Linux version. Having the package specified by the `toolkit` does not guarantee that the class files for this user interface will be present. It is primarily intended to narrow lookups based on the UIs offered.

Role

There are many possible roles for a user interface. For example, a typical user may be using the service, in which case the UI plays the "main" role. Alternatively, a system administrator may be managing the service, and he or she might require a different user interface, in which case the UI then plays the "admin" role.

The `role` field in a `UIDescriptor` is intended to describe these possible variations in the use of a user interface. The value of this field is a string, and to reduce the possibility of spelling errors that are not discovered until runtime, the value should be one of several constant string values. These string constants are defined in a set of type interfaces known as *role* interfaces. There are currently three role interfaces:

- The `net.jini.lookup.ui.MainUI` role is for the standard user interface used by ordinary clients of the service:

```
package net.jini.lookup.ui;
public interface MainUI {
    String ROLE = "net.jini.lookup.ui.MainUI";
}
```

- The `net.jini.lookup.ui.AdminUI` role is for use by the service's administrator:

```
package net.jini.lookup.ui;
public interface AdminUI {
    String ROLE = "net.jini.lookup.ui.AdminUI";
}
```

- The `net.jini.lookup.ui.AboutUI` role is for information about the service, which can be presented by a user interface object:

```
package net.jini.lookup.ui;
public interface AboutUI {
    String ROLE = "net.jini.lookup.ui.AboutUI";
}
```

A service will specify a role for each of the user interfaces it supplies. This role is given in a number of ways for different objects:

- The `role` field in the `UIDescriptor` must be set to the `String ROLE` of one of these three role interfaces.

- The user interface indicates that it acts a role by implementing the particular role specified.

- The factory does not explicitly know about the role, but the factory contained in a `UIDescriptor` must produce a user interface implementing the role.

The service must ensure that the `UIDescriptors` it produces follows these rules. How it actually does so is not specified. There are several possibilities, including these:

- When a factory is created, the role is passed in through a constructor. It can then use this role to cast the `roleObject` in the `getXXX()` method to the expected class (currently this is always a `ServiceItem`).

- There could be different factories for different roles, and the `UIDescriptor` should have the right factory for that role.

The factory could perform some sanity checking if desired; since all `roleObjects` are (presently) the service items, it could search through these items for the `UIDescriptor`, and then check that its `role` matches what the factory expects.

There has been much discussion about "flavors" of roles, such as an "expert" role or a "learner" role. This has been deferred because it is too complicated, at least for the first version of the specification.

Attributes

The `attributes` section of a `UIDescriptor` can carry any other information about the user interface object that the service thinks might be useful to clients trying to decide which user interface to choose. Currently this includes the following:

- A `UIFactoryTypes` object, which contains a set of `Strings` for the fully qualified class names of the factory that this entry contains. The current factory hierarchy is very shallow, so this may be just a singleton set, like this:

```
Set attribs = new HashSet();
        Set typeNames = new HashSet();
        typeNames.add(JFrameFactory.TYPE_NAME);
        attribs.add(new UIFactoryTypes(typeNames));
```

  Note that a client is not usually interested in the actual type of the factory, but rather in the interface it implements. This is just like Jini services themselves, where we only need to know the methods that can be called and are not concerned with the implementation details.

- An `AccessibleUI` object. Inclusion of this object indicates that the user interface implements `javax.accessibility.Accessible` and that the user interface would work well with assistive technologies.

- A `Locales` object, which specifies the locales supported by the user interface.

- A `RequiredPackages` object, which contains information about all of the packages that the user interface needs to run. This is not a guarantee that the user interface will actually run, nor a guarantee that it will be a usable interface, but it may help a client decide whether or not to use a particular user interface.

File Classifier UI Example

The file classifier has been used throughout this book as a simple example of a service to illustrate various features of Jini. We can use it here too, by supplying simple user interfaces to the service. Such a user interface would consist of a text field for entering a filename, and a display to show the MIME type of the filename. There is only a "main" role for this service, as no administration needs to be performed.

Figure 19-1 shows what a user interface for a file classifier could look like.

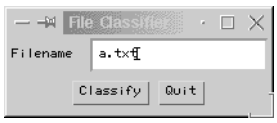

Figure 19-1. FileClassifier user interface

After the service has been invoked, it could pop up a dialog box, as shown in Figure 19-2.

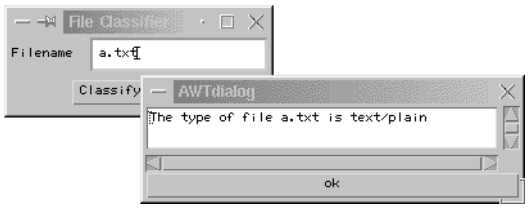

Figure 19-2. FileClassifier return dialog box

A factory for the "main" role that will produce an AWT `Frame` is shown next:

```
/**
 * FileClassifierFrameFactory.java
 */

package ui;

import net.jini.lookup.ui.factory.FrameFactory;
import net.jini.lookup.entry.UIDescriptor;
import java.awt.Frame;
import net.jini.core.entry.Entry;
import net.jini.core.lookup.ServiceItem;

public class FileClassifierFrameFactory implements FrameFactory {

    /**
     * Return a new FileClassifierFrame that implements the
     * MainUI role
     */
    public Frame getFrame(Object roleObject) {
        // we should check to see what role we have to return
        if (! (roleObject instanceof ServiceItem)) {
            // unknown role type object
            // can we return null?
            return null;
        }
        ServiceItem item = (ServiceItem) roleObject;

        // Do sanity checking that the UIDescriptor has a MainUI role
        Entry[] entries = item.attributeSets;
        for (int n = 0; n < entries.length; n++) {
            if (entries[n] instanceof UIDescriptor) {
                UIDescriptor desc = (UIDescriptor) entries[n];
                if (desc.role.equals(net.jini.lookup.ui.MainUI.ROLE)) {
                    // Ok, we are in the MainUI role, so return a UI for that
                   Frame frame = new FileClassifierFrame(item, "File Classifier");
                    return frame;
                }
            }
        }
        // couldn't find a role the factory can create
        return null;
    }
```

```
} // FileClassifierFrameFactory
```

The user interface object that performs this role is as follows:

```
/**
 * FileClassifierFrame.java
 */

package ui;

import java.awt.*;
import java.awt.event.*;
import net.jini.lookup.ui.MainUI;
import net.jini.core.lookup.ServiceItem;
import common.MIMEType;
import common.FileClassifier;
import java.rmi.RemoteException;

/**
 * Object implementing MainUI for FileClassifier.
 */
public class FileClassifierFrame extends Frame implements MainUI {

    ServiceItem item;
    TextField text;

    public FileClassifierFrame(ServiceItem item, String name) {
        super(name);

        Panel top = new Panel();
        Panel bottom = new Panel();
        add(top, BorderLayout.CENTER);
        add(bottom, BorderLayout.SOUTH);

        top.setLayout(new BorderLayout());
        top.add(new Label("Filename"), BorderLayout.WEST);
        text = new TextField(20);
        top.add(text, BorderLayout.CENTER);

        bottom.setLayout(new FlowLayout());
        Button classify = new Button("Classify");
        Button quit = new Button("Quit");
        bottom.add(classify);
        bottom.add(quit);
```

```
        // listeners
        quit.addActionListener(new QuitListener());
        classify.addActionListener(new ClassifyListener());

        // We pack, but don't make it visible
        pack();
    }

    class QuitListener implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            System.exit(0);
        }
    }

    class ClassifyListener implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            String fileName = text.getText();
            final Dialog dlg = new Dialog((Frame) text.getParent().getParent());
            dlg.setLayout(new BorderLayout());
            TextArea response = new TextArea(3, 20);

            // invoke service
            FileClassifier classifier = (FileClassifier) item.service;
            MIMEType type = null;
            try {
                type = classifier.getMIMEType(fileName);
                if (type == null) {
                    response.setText("The type of file " + fileName +
                                     " is unknown");
                } else {
                    response.setText("The type of file " + fileName +
                                     " is " + type.toString());
                }
            } catch(RemoteException e) {
                response.setText(e.toString());
            }

            Button ok = new Button("ok");
            ok.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    dlg.setVisible(false);
                }
            });
```

```
            dlg.add(response, BorderLayout.CENTER);
            dlg.add(ok, BorderLayout.SOUTH);
            dlg.setSize(300, 100);
            dlg.setVisible(true);
        }
    }

} // FileClassifierFrame
```

The server that delivers both the service and the user interface has to prepare a `UIDescriptor`. In this case, it only creates one such object for a single user interface, but if the server exported more interfaces, it would simply create more descriptors. Here is the server code:

```
/**
 * FileClassifierServer.java
 */

package ui;

import complete.FileClassifierImpl;

import net.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceRegistrar;
import java.rmi.RemoteException;
import net.jini.lookup.ServiceIDListener;
import net.jini.lease.LeaseRenewalManager;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.DiscoveryListener;
import net.jini.core.entry.Entry;

import net.jini.lookup.ui.MainUI;
import net.jini.lookup.ui.factory.FrameFactory;
import net.jini.lookup.entry.UIDescriptor;
import net.jini.lookup.ui.attribute.UIFactoryTypes;

import java.rmi.MarshalledObject;
import java.io.IOException;
import java.util.Set;
import java.util.HashSet;
```

```
public class FileClassifierServer
    implements ServiceIDListener {

    public static void main(String argv[]) {
        new FileClassifierServer();

        // stay around forever
        Object keepAlive = new Object();
        synchronized(keepAlive) {
            try {
                keepAlive.wait();
            } catch(InterruptedException e) {
                // do nothing
            }
        }
    }

    public FileClassifierServer() {

        JoinManager joinMgr = null;

        // The typenames for the factory
        Set typeNames = new HashSet();
        typeNames.add(FrameFactory.TYPE_NAME);

        // The attributes set
        Set attribs = new HashSet();
        attribs.add(new UIFactoryTypes(typeNames));

        // The factory
        MarshalledObject factory = null;
        try {
            factory = new MarshalledObject(new FileClassifierFrameFactory());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(2);
        }
        UIDescriptor desc = new UIDescriptor(MainUI.ROLE,
                                             FileClassifierFrameFactory.TOOLKIT,
                                             attribs,
                                             factory);

        Entry[] entries = {desc};
```

```
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            joinMgr = new JoinManager(new FileClassifierImpl(), /* service */
                                      entries /* attr sets */,
                                      this /* ServiceIDListener*/,
                                      mgr /* DiscoveryManagement */,
                                      new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public void serviceIDNotify(ServiceID serviceID) {
        // called as a ServiceIDListener
        // Should save the ID to permanent storage
        System.out.println("got service ID " + serviceID.toString());
    }

} // FileClassifierServer
```

Finally, a client needs to look for and use this user interface. The client finds a service as usual and then does a search through the `Entry` objects, looking for a `UIDescriptor`. Once it has a descriptor, it can check whether the descriptor meets the requirements of the client. Here we shall check whether it plays a `MainUI` role and can generate an AWT `Frame`:

```
package client;

import common.FileClassifier;
import common.MIMEType;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ClientLookupManager;
import net.jini.core.lookup.ServiceItem;
import net.jini.lease.LeaseRenewalManager;
import net.jini.core.entry.Entry;
```

```
import net.jini.lookup.ui.MainUI;
import net.jini.lookup.ui.factory.FrameFactory;
import net.jini.lookup.entry.UIDescriptor;
import net.jini.lookup.ui.attribute.UIFactoryTypes;

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

import java.util.Set;
import java.util.Iterator;
import java.net.URL;

/**
 * TestFrameUI.java
 */

public class TestFrameUI {

    private static final long WAITFOR = 100000L;

    public static void main(String argv[]) {
        new TestFrameUI();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(2*WAITFOR);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public TestFrameUI() {
        ClientLookupManager clientMgr = null;

        System.setSecurityManager(new RMISecurityManager());

        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            clientMgr = new ClientLookupManager(mgr,
                                                new LeaseRenewalManager());
```

```
    } catch(Exception e) {
        e.printStackTrace();
        System.exit(1);
    }

    Class [] classes = new Class[] {FileClassifier.class};
    UIDescriptor desc = new UIDescriptor(MainUI.ROLE,
                                         FrameFactory.TOOLKIT,
                                         null, null);
    Entry [] entries = {desc};

    ServiceTemplate template = new ServiceTemplate(null, classes,
                                                   entries);

    ServiceItem item = null;
    try {
        item = clientMgr.lookup(template,
                                null, /* no filter */
                                WAITFOR /* timeout */);
    } catch(Exception e) {
        e.printStackTrace();
        System.exit(1);
    }

    if (item == null) {
        // couldn't find a service in time
        System.out.println("no service");
        System.exit(1);
    }

    // We now have a service that plays the MainUI role and
    // uses the FrameFactory toolkit of "java.awt".
    // We now have to find if there is a UIDescriptor
    // with a Factory generating an AWT Frame
    checkUI(item);
}

private void checkUI(ServiceItem item) {
    // Find and check the UIDescriptors
    Entry[] attributes = item.attributeSets;
    for (int m = 0; m < attributes.length; m++) {
        Entry attr = attributes[m];
        if (attr instanceof UIDescriptor) {
            // does it deliver an AWT Frame?
```

```
                checkForAWTFrame(item, (UIDescriptor) attr);
            }
        }
    }

    private void checkForAWTFrame(ServiceItem item, UIDescriptor desc) {
        Set attributes = desc.attributes;
        Iterator iter = attributes.iterator();
        while (iter.hasNext()) {
            // search through the attributes, to find a UIFactoryTypes
            Object obj = iter.next();
            if (obj instanceof UIFactoryTypes) {
                UIFactoryTypes types = (UIFactoryTypes) obj;
                // see if it produces an AWT Frame Factory
                if (types.isAssignableTo(FrameFactory.class)) {
                    FrameFactory factory = null;
                    try {
                        factory = (FrameFactory) desc.getUIFactory(this.getClass().
                                                                   getClassLoader());
                    } catch(Exception e) {
                        e.printStackTrace();
                        continue;
                    }

                    Frame frame = factory.getFrame(item);
                    frame.setVisible(true);
                }
            }
        }
    }


} // TestFrameUI
```

Images

User interfaces often contain images. They may be used as icons in toolbars, as general images on the screen, or as the icon image when the application is iconified. When a user interface is created on the client, these images will also need to be created and installed in the relevant part of the application. Images are not serializable, so they cannot be created on the server and exported as live objects in some manner. They need to be created from scratch on the client.

The Swing package contains a convenience class called `ImageIcon`. This class can be instantiated from a byte array, a filename, or most interestingly here, from a URL. So, if an image is stored where an HTTP server can find it, the `ImageIcon` constructor can use this version directly. There may be failures in this approach: the URL may be incorrect or malformed or the image may not exist on the HTTP server. Suitable code to create an image from a URL is as follows:

```
ImageIcon icon = null;
        try {
            icon = new ImageIcon(new URL("http://localhost/images/mindstorms.ps"));
            switch (icon.getImageLoadStatus()) {
            case MediaTracker.ABORTED:
            case MediaTracker.ERRORED:
                System.out.println("Error");
                icon = null;
                break;
            case MediaTracker.COMPLETE:
                System.out.println("Complete");
                break;
            case MediaTracker.LOADING:
                System.out.println("Loading");
                break;
            }
        } catch(java.net.MalformedURLException e) {
            e.printStackTrace();
        }
        // icon is null or is a valid image
```

ServiceType

A user interface may use code like that in the previous section directly to include images. The service may also supply useful images and other human-oriented information in a `ServiceType` entry object. The `ServiceType` class is defined as follows:

```
package net.jini.lookup.entry;

public class ServiceType {
    public String getDisplayName();     // Return the localized display
                                        // name of this service.
    public Image getIcon(int iconKind)  // Get an icon for this service.
    public String getShortDescription() // Return a localized short
                                        // description of this service.
}
```

The class is supplied with empty implementations, returning `null` for each method. A service will need to supply a subclass with useful implementations of the methods. This is a useful class that could be used to supply images and information that may be common to a number of different user interfaces for a service, such as a minimized image.

MindStorms UI Example

In Chapter 17, an example was given, in the "Getting It Running" section, of a client supplying a user interface to a MindStorms service. This client not only knew that the service was a MindStorms robot, but that it was a particular robot for which it could use a customized UI. In this section, we'll give two user interfaces for the MindStorms "RoverBot," one of which is fairly general and could be used for any robot, and another that is customized to the RoverBot. The service is responsible for creating and exporting both of these user interfaces to a client.

RCXLoaderFrame

A MindStorms robot is primarily defined by the `RCXPort` interface. The Jini version is defined by the `RCXPortImplementation` interface:

```
/**
 * RCXPortInterface.java
 */

package rcx.jini;

import net.jini.core.event.RemoteEventListener;

public interface RCXPortInterface extends java.io.Serializable {

    /**
     * constants to distinguish message types
     */
    public final long ERROR_EVENT = 1;
    public final long MESSAGE_EVENT = 2;

    /**
     * Write an array of bytes that are RCX commands
     * to the remote RCX.
     */
```

```
    public boolean write(byte[] byteCommand) throws java.rmi.RemoteException;

    /**
     * Parse a string into a set of RCX command bytes
     */
    public byte[] parseString(String command) throws java.rmi.RemoteException;

    /**
     * Add a RemoteEvent listener to the RCX for messages and errors
     */
    public void addListener(RemoteEventListener listener)
        throws java.rmi.RemoteException;

    /**
     * The last message from the RCX
     */
    public byte[] getMessage(long seqNo)
        throws java.rmi.RemoteException;

    /**
     * The error message from the RCX
     */
    public String getError(long seqNo)
        throws java.rmi.RemoteException;

} // RCXPortInterface
```

This type interface allows programs to be downloaded and run and instructions to be sent for direct execution. As it stands, the client needs to call these interface methods directly. To make it more useable for the human trying to drive a robot, some sort of user interface would be useful.

There can be several general purpose user interfaces for the RCX robot, including these:

- Enter machine code (somehow) and download that.
- Enter RCX assembler code in the form of strings, and then assemble and download them.
- Enter NQC (Not Quite C) code, and then compile and download it.

The set of RCX classes by Laverde at `http://www.escape.com/~dario/java/rcx` includes a standalone application called `RCXLoader`, which does the second of these

options. We can steal code from RCXLoader and some of his other classes to define an RCXLoaderFrame class:

```
package rcx.jini;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.UnknownEventException;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import rcx.*;

/*
 * RCXLoaderFrame
 * @author Dario Laverde
 * @author Jan Newmarch
 * @version 1.1
 * Copyright 1999 Dario Laverde, under terms of GNU LGPL
 */

public class RCXLoaderFrame extends Frame
    implements ActionListener, WindowListener, RemoteEventListener
{
    private String      portName;
    private RCXPortInterface     rcxPort;
    private Panel       textPanel;
    private Panel       topPanel;
    private TextArea    textArea;
    private TextField   textField;
    private Button      tableButton;
    private Properties  parameters;
    private int inByte;
    private int charPerLine = 48;
    private int lenCount;
    private StringBuffer sbuffer;
    private byte[] byteArray;
    private Frame opcodeFrame;
    private TextArea opcodeTextArea;

    public static Hashtable Opcodes=new Hashtable(55);
```

```
    static {
        Opcodes.put(new Byte((byte)0x10),"PING            ,void, void,P");
       Opcodes.put(new Byte((byte)0x12),"GETVAL         ,byte src byte arg, short
val,P");
        Opcodes.put(new Byte((byte)0x13),"SETMOTORPOWER   ,byte motors byte src
byte arg, void,CP");
       Opcodes.put(new Byte((byte)0x14),"SETVAL          ,byte index byte src byte
arg, void,CP");
        // Opcodes truncated to save space in listing
    }

    // added port interface parameter to Dario's code
    public RCXLoaderFrame(RCXPortInterface port) {
            super("RCX Loader");

        // changed from Dario's code
        rcxPort = port;

        addWindowListener(this);

        topPanel  = new Panel();
        topPanel.setLayout(new BorderLayout());

        tableButton = new Button("table");
        tableButton.addActionListener(this);

        textField = new TextField();
        // textField.setEditable(false);
        // textField.setEnabled(false);
        // tableButton.setEnabled(false);
        textField.addActionListener(this);

        textPanel = new Panel();
        textPanel.setLayout(new BorderLayout(5,5));

        topPanel.add(textField,"Center");
        topPanel.add(tableButton,"East");
        textPanel.add(topPanel,"North");

        textArea = new TextArea();
        // textArea.setEditable(false);
        textArea.setFont(new Font("Courier",Font.PLAIN,12));
        textPanel.add(textArea,"Center");
```

```
add(textPanel, "Center");

textArea.setText("initializing...\n");

Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();
setBounds(screen.width/2-370/2,screen.height/2-370/2,370,370);
// setVisible(true);

// changed listener type from Dario's code
try {
    // We are remote to the object we are listening to
    // (the RCXPort), so the RCXPort must get a stub object
    // for us. We have subclassed from Frame, not from
    // UnicastRemoteObject. So we must export ourselves
    // for the remote references to work
    UnicastRemoteObject.exportObject(this);
    rcxPort.addListener(this);
} catch(Exception e) {
    textArea.append(e.toString());
}
tableButton.setEnabled(true);

/*
if(rcxPort.isOpen()) {
    textArea.append("RCXPort initialized.\n");
    String lasterror = rcxPort.getLastError();
    if(lasterror!=null)
        textArea.append(lasterror+"\n");
    textField.setEditable(true);
    textField.setEnabled(true);
    textField.requestFocus();
}
else {
    if(portName!=null) {
        textArea.append("Failed to create RCXPort with "+portName+"\n");
        textArea.append("Port "+portName+" is invalid or may be ");
        textArea.append("currently used.\nTry another port.\n");
        textArea.append("Edit or create a file named parameters.txt ");
        textArea.append("that has:\nport=COM1\n(replace COM1 with ");
        textArea.append("the correct port name)\n");
    }
    else
        textArea.append("Please specify a port in parameters.txt\n");
```

```
            textArea.append("Create a file that has:\nport=COM1\n");
            textArea.append("(replace COM1 with the correct port name)\n");
    }
    */
}

/*
public void receivedMessage(byte[] responseArray) {
    if(responseArray==null)
        return;
    for(int loop=0;loop<responseArray.length;loop++) {
        int newbyte = (int)responseArray[loop];
        if(newbyte<0) newbyte=256+newbyte;
        sbuffer = new StringBuffer(Integer.toHexString(newbyte));

        if(sbuffer.length()<2)
            sbuffer.insert(0,'0');
        textArea.append(sbuffer+" ");
        lenCount+=3;
        if(lenCount==charPerLine) {
            lenCount=0;
            textArea.append("\n");
        }
    }
    if(lenCount!=charPerLine)
        textArea.append("\n");
}
*/

/*
public void receivedError(String error) {
    textArea.append(error+"\n");
}
*/

public void actionPerformed(ActionEvent evt) {
    Object obj = evt.getSource();
    if(obj==textField) {
        String strInput = textField.getText();
        textField.setText("");
        textArea.append("> "+strInput+"\n");
        try {
            byteArray = rcxPort.parseString(strInput);
        } catch(RemoteException e) {
            textArea.append(e.toString());
```

```
                }
                // byteArray = RCXOpcode.parseString(strInput);
                if(byteArray==null) {
                    textArea.append("Error: illegal hex character or length\n");
                    return;
                }
                if(rcxPort!=null) {
                    try {
                        if(!rcxPort.write(byteArray)) {
                            textArea.append("Error: writing data to port
"+portName+"\n");
                        }
                    } catch(Exception e) {
                        textArea.append(e.toString());
                    }
                }
            }
            else if(obj==tableButton) {
                // make this all in the ui side
                showTable();
                setLocation(0,getLocation().y);
            }
        }

        public void windowActivated(WindowEvent e) { }
        public void windowClosed(WindowEvent e) { }
        public void windowDeactivated(WindowEvent e) { }
        public void windowDeiconified(WindowEvent e) { }
        public void windowIconified(WindowEvent e) { }
        public void windowOpened(WindowEvent e) { }
        public void windowClosing(WindowEvent e) {
            /*
            if(rcxPort!=null)
                    rcxPort.close();
            */
            System.exit(0);
        }

        public void notify(RemoteEvent evt) throws UnknownEventException,
                                                   java.rmi.RemoteException {

            long id = evt.getID();
            long seqNo = evt.getSequenceNumber();
            if (id == RCXPortInterface.MESSAGE_EVENT) {
```

```
            byte[] message = rcxPort.getMessage(seqNo);
            StringBuffer sbuffer = new StringBuffer();
            for(int n = 0; n < message.length; n++) {
                int newbyte = (int) message[n];
                if (newbyte < 0) {
                    newbyte += 256;
                }
                sbuffer.append(Integer.toHexString(newbyte) + " ");
            }
            textArea.append(sbuffer.toString());
            System.out.println("MESSAGE: " + sbuffer.toString());
        } else if (id == RCXPortInterface.ERROR_EVENT) {
            textArea.append(rcxPort.getError(seqNo));
        } else {
            throw new UnknownEventException("Unknown message " + evt.getID());
        }
    }

    public void showTable()
    {
        if(opcodeFrame!=null)
        {
            opcodeFrame.dispose();
            opcodeFrame=null;
            return;
        }
        opcodeFrame = new Frame("RCX Opcodes Table");
        Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();
        opcodeFrame.setBounds(screen.width/2-70,0,screen.width/2+70,screen.height-
25);
        opcodeTextArea = new TextArea("   Opcode          ,parameters, response,
C=program command P=remote command\n",60,100);
        opcodeTextArea.setFont(new Font("Courier",Font.PLAIN,10));
        opcodeFrame.add(opcodeTextArea);
        Enumeration k = Opcodes.keys();
        for (Enumeration e = Opcodes.elements(); e.hasMoreElements();) {
            String tmp = Integer.toHexString(((Byte)k.nextElement()).intValue());
            tmp = tmp.substring(tmp.length()-2)+" "+(String)e.nextElement()+"\n";
            opcodeTextArea.append(tmp);
        }
        opcodeTextArea.setEditable(false);
        opcodeFrame.setVisible(true);
    }
}
```

RCXLoaderFrameFactory

The factory object for the RCX is now easy to define—it just returns a `RCXLoaderFrame` in the `getUI()` method:

```
/**
 * RCXLoaderFrameFactory.java
 */

package rcx.jini;

import net.jini.lookup.ui.factory.FrameFactory;
import net.jini.core.lookup.ServiceItem;
import java.awt.Frame;

public class RCXLoaderFrameFactory implements FrameFactory {

    public Frame getFrame(Object roleObj) {
        ServiceItem item= (ServiceItem) roleObj;
        RCXPortInterface port = (RCXPortInterface) item.service;
        return new RCXLoaderFrame(port);
    }

} // RCXLoaderFrameFactory
```

Exporting the FrameFactory

The factory object is exported by making it a part of a `UIDescriptor` entry object with a role, toolkit, and attributes:

```
Set typeNames = new HashSet();
typeNames.add(FrameFactory.TYPE_NAME);

Set attribs = new HashSet();
attribs.add(new UIFactoryTypes(typeNames));
// add other attributes as desired

MarshalledObject factory = null;
try {
    factory = new MarshalledObject(new
                                   RCXLoaderFrameFactory());
} catch(Exception e) {
    e.printStackTrace();
```

```
    System.exit(2);
}

UIDescriptor desc = new UIDescriptor(MainUI.ROLE,
                                     FrameFactory.TOOLKIT,
                                     attribs,
                                     factory);
Entry[] entries = {desc};

JoinManager joinMgr = new JoinManager(impl,
                                      entries,
                                      this,
                                      new LeaseRenewalManager());
```

Customized User Interfaces

The `RCXLoaderFrame` is a general interface to any RCX robot. Of course, there could be many other such interfaces, differing in the classes used, the amount of international support, the appearance, etc. All the variations, however, will just use the standard `RCXPortInterface`, because that is all they know about.

The Lego pieces can be combined in a huge variety of ways, and the RCX itself is programmable, so you can build an RCX car, an RCX crane, an RCX maze-runner, and so on. Each different robot can be driven by the general interface, but most could benefit from a custom-built interface for that type of robot. This is typical: for example, every blender could be driven from a general blender user interface (using the possibly forthcoming standard blender interface :-). But the blenders from individual vendors would have their own customized user interface for their brand of blender.

I have been using an RCX car. While it can do lots of things, it has been convenient to use five commands for demonstrations: forward, stop, back, left, and right, with a user interface as shown in Figure 19-3.

Figure 19-3. User interface for MindStorms car

In Chapter 17, this appearance was hard-coded into the client. Since the client was just searching for *any* MindStorms robot, it really shouldn't know about this sort of detail and should get this user interface from the robot service.

CarJFrame

The `CarJFrame` class produces the user interface as a Swing `JFrame`, with the buttons generating specific RCX code for this model.

```
package rcx.jini;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.UnknownEventException;
import java.net.URL;
import java.rmi.RemoteException;

class CarJFrame extends JFrame
    implements RemoteEventListener, ActionListener {

    public static final int STOPPED = 1;
    public static final int FORWARDS = 2;
    public static final int BACKWARDS = 4;

    protected int state = STOPPED;

    protected RCXPortInterface port = null;

    JFrame frame;
    JTextArea text;

    public CarJFrame(RCXPortInterface port) {
        super() ;
        this.port = port;

        frame = new JFrame("Lego MindStorms");
        Container content = frame.getContentPane();
        JLabel label = null;
        ImageIcon icon = null;
        try {
```

```
    icon = new ImageIcon(new
                URL("http://www.legomindstorms.com/images/home_logo.ps"));
    switch (icon.getImageLoadStatus()) {
    case MediaTracker.ABORTED:
    case MediaTracker.ERRORED:
        System.out.println("Error");
        icon = null;
        break;
    case MediaTracker.COMPLETE:
        System.out.println("Complete");
        break;
    case MediaTracker.LOADING:
        System.out.println("Loading");
        break;
    }
} catch(java.net.MalformedURLException e) {
    e.printStackTrace();
}
if (icon != null) {
    label = new JLabel(icon);
} else {
    label = new JLabel("MINDSTORMS");
}

JPanel pane = new JPanel();
pane.setLayout(new GridLayout(2, 3));

content.add(label, "North");
content.add(pane, "Center");

JButton btn = new JButton("Forward");
pane.add(btn);
btn.addActionListener(this);

btn = new JButton("Stop");
pane.add(btn);
btn.addActionListener(this);

btn = new JButton("Back");
pane.add(btn);
btn.addActionListener(this);

btn = new JButton("Left");
pane.add(btn);
```

```
        btn.addActionListener(this);

        label = new JLabel("");
        pane.add(label);

        btn = new JButton("Right");
        pane.add(btn);
        btn.addActionListener(this);

        frame.pack();
        frame.setVisible(true);
    }

    public void sendCommand(String comm) {
        byte[] command;
        try {
            command = port.parseString(comm);
            if (! port.write(command)) {
                System.err.println("command failed");
            }
        } catch(RemoteException e) {
            e.printStackTrace();
        }
    }

    public void forwards() {
        sendCommand("e1 85");
        sendCommand("21 85");
        state = FORWARDS;
    }

    public void backwards() {
        sendCommand("e1 45");
        sendCommand("21 85");
        state = BACKWARDS;
    }

    public void stop() {
        sendCommand("21 45");
        state = STOPPED;
    }

    public void restoreState() {
        if (state == FORWARDS)
```

```
        forwards();
    else if (state == BACKWARDS)
        backwards();
    else
        stop();
}

public void actionPerformed(ActionEvent evt) {
    String name = evt.getActionCommand();
    byte[] command;

    if (name.equals("Forward")) {
        forwards();
    } else  if (name.equals("Stop")) {
        stop();
    } else  if (name.equals("Back")) {
        backwards();
    } else  if (name.equals("Left")) {
        sendCommand("e1 84");
        sendCommand("21 84");
        sendCommand("21 41");
        try {
            Thread.sleep(100);
        } catch(InterruptedException e) {
        }
        restoreState();

    } else  if (name.equals("Right")) {
        sendCommand("e1 81");
        sendCommand("21 81");
        sendCommand("21 44");
        try {
            Thread.sleep(100);
        } catch(InterruptedException e) {
        }
        restoreState();
    }
}

public void notify(RemoteEvent evt) throws UnknownEventException,
    java.rmi.RemoteException {
    // System.out.println(evt.toString());

    long id = evt.getID();
```

```
        long seqNo = evt.getSequenceNumber();
        if (id == RCXPortInterface.MESSAGE_EVENT) {
            byte[] message = port.getMessage(seqNo);
            StringBuffer sbuffer = new StringBuffer();
            for(int n = 0; n < message.length; n++) {
                int newbyte = (int) message[n];
                if (newbyte < 0) {
                    newbyte += 256;
                }
                sbuffer.append(Integer.toHexString(newbyte) + " ");
            }
            System.out.println("MESSAGE: " + sbuffer.toString());
        } else if (id == RCXPortInterface.ERROR_EVENT) {
            System.out.println("ERROR: " + port.getError(seqNo));
        } else {
            throw new UnknownEventException("Unknown message " + evt.getID());
        }
    }
}
```

CarJFrameFactory

The factory generates a `CarJFrame` object, like this:

```
/**
 * CarJFrameFactory.java
 */

package rcx.jini;

import net.jini.lookup.ui.factory.JFrameFactory;
import net.jini.core.lookup.ServiceItem;
import javax.swing.JFrame;

public class CarJFrameFactory implements JFrameFactory {

    public JFrame getJFrame(Object roleObj) {
        ServiceItem item  = (ServiceItem) roleObj;
        RCXPortInterface port = (RCXPortInterface) item.service;
        return new CarJFrame(port);
    }

} // CarJFrameFactory
```

Exporting the FrameFactory

Both of the user interfaces discussed—the RCXLoaderFrame and the CarJFrame–can be exported by expanding the set of Entry objects.

```
// generic UI
Set genericAttribs = new HashSet();
Set typeNames = new HashSet();
typeNames.add(FrameFactory.TYPE_NAME);
genericAttribs.add(new UIFactoryTypes(typeNames));
MarshalledObject genericFactory = null;
try {
    genericFactory = new MarshalledObject(new
                                RCXLoaderFrameFactory());
} catch(Exception e) {
    e.printStackTrace();
            System.exit(2);
}
UIDescriptor genericDesc = new UIDescriptor(MainUI.ROLE,
                                            FrameFactory.TOOLKIT,
                                               genericAttribs,
                                            genericFactory);

// car UI
Set carAttribs = new HashSet();
typeNames = new HashSet();
typeNames.add(JFrameFactory.TYPE_NAME);
carAttribs.add(new UIFactoryTypes(typeNames));
MarshalledObject carFactory = null;
try {
    carFactory = new MarshalledObject(new CarJFrameFactory());
} catch(Exception e) {
    e.printStackTrace();
    System.exit(2);
}
UIDescriptor carDesc = new UIDescriptor(MainUI.ROLE,
                                        JFrameFactory.TOOLKIT,
                                        carAttribs,
                                        carFactory);

Entry[] entries = {genericDesc, carDesc};

JoinManager joinMgr = new JoinManager(impl,
                                      entries,
```

```
                                        this,
                                        new LeaseRenewalManager());
```

The RCX Client

The following client will start up all user interfaces that implement the main UI role and that use a `Frame` or `JFrame`:

```
package client;

import rcx.jini.*;

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

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.UnknownEventException;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import net.jini.core.entry.Entry;
import net.jini.core.lookup.ServiceMatches;
import net.jini.core.lookup.ServiceItem;

import net.jini.lookup.entry.UIDescriptor;
import net.jini.lookup.ui.MainUI;
import net.jini.lookup.ui.attribute.UIFactoryTypes;
import net.jini.lookup.ui.factory.FrameFactory;
import net.jini.lookup.ui.factory.JFrameFactory;

import java.util.Set;
import java.util.Iterator;

/**
 * TestRCX2.java
 */
```

```
public class TestRCX2 implements DiscoveryListener {

    public static void main(String argv[]) {
        new TestRCX2();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(1000000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public TestRCX2() {
        System.setSecurityManager(new RMISecurityManager());

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);

    }

    public void discovered(DiscoveryEvent evt) {

        ServiceRegistrar[] registrars = evt.getRegistrars();
        Class [] classes = new Class[] {RCXPortInterface.class};
        RCXPortInterface port = null;

        UIDescriptor desc = new UIDescriptor(MainUI.ROLE, null, null, null);
        Entry[] entries = {desc};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       entries);

        for (int n = 0; n < registrars.length; n++) {
            System.out.println("Service found");
            ServiceRegistrar registrar = registrars[n];
            ServiceMatches matches = null;
```

```
            try {
                matches = registrar.lookup(template, 10);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                System.exit(2);
            }
            for (int nn = 0; nn < matches.items.length; nn++) {
                ServiceItem item = matches.items[nn];
                port = (RCXPortInterface) item.service;
                if (port == null) {
                    System.out.println("port null");
                    continue;
                }

                Entry[] attributes = item.attributeSets;
                for (int m = 0; m < attributes.length; m++) {
                    Entry attr = attributes[m];
                    if (attr instanceof UIDescriptor) {
                        showUI(port, item, (UIDescriptor) attr);
                    }
                }
            }



        }
    }

    public void discarded(DiscoveryEvent evt) {
        // empty
    }

    private void showUI(RCXPortInterface port,
                        ServiceItem item,
                        UIDescriptor desc) {
        Set attribs = desc.attributes;
        Iterator iter = attribs.iterator();
        while (iter.hasNext()) {
            Object obj = iter.next();
            if (obj instanceof UIFactoryTypes) {
                UIFactoryTypes types = (UIFactoryTypes) obj;
                Set typeNames = types.getTypeNames();
                if (typeNames.contains(FrameFactory.TYPE_NAME)) {
                    FrameFactory factory = null;
```

```
                    try {
                       factory = (FrameFactory) desc.getUIFactory(this.getClass().
                                                                  getClassLoader());
                    } catch(Exception e) {
                        e.printStackTrace();
                        continue;
                    }
                    Frame frame = factory.getFrame(item);
                    frame.setVisible(true);
                } else if (typeNames.contains(JFrameFactory.TYPE_NAME)) {
                    JFrameFactory factory = null;
                    try {
                       factory = (JFrameFactory) desc.getUIFactory(this.getClass().
                                                                  getClassLoader());
                    } catch(Exception e) {
                        e.printStackTrace();
                        continue;
                    }
                    JFrame frame = factory.getJFrame(item);
                    frame.setVisible(true);
                }
            } else {
                System.out.println("non-gui entry");
            }
        }
    }
} // TestRCX
```

Summary

The serviceUI group is evolving a standard mechanism for services to distribute user interfaces for Jini services. The preference is to do this by `Entry` objects that contain factories for producing user interfaces.