CHAPTER 24

# User Interfaces for Jini Services

Jini is designed to allow client programs to discover and interact with services. There is no user interface explicitly involved in this, although many clients will have a user interface to talk to a user. However, services may wish to offer a user interface for a client to show to a user. This could be for many reasons, such as extended functionality, or to give the service a particular look and feel.

In this chapter, we look at how a service can advertise user interfaces, how a client can choose an appropriate interface, and how the client can use a service's user interface.

## User Interfaces As Entries

Interaction with a service is specified by its interface, and this is the same across all implementations of the interface. Just using the known interface doesn't allow any flexibility in using the service, because a client will only know about the methods defined in the interface. The interface is the defining level for using any service that implements the interface, but services can be implemented in many different ways, and service implementations do, in fact, differ. There is a need to allow for these differences, and the mechanism used in Jini is to put the differences in `Entry` objects. Typical objects supplied by vendors may include `Name` and `ServiceInfo`.

Clients can make use of the interface and these additional entry items, primarily in the selection of a service. But once clients have the service, are they 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. 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 the capability by supplying a user interface object with a special component for "number of copies."

The user interface for a service cannot expect to have all details supplied by the client—at best, a client could only manage a fairly generic user interface. The user interface should come from the vendor, or maybe even a third party (when your video player becomes Jini enabled, for example, 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 from Factory Objects

Chapter 15 discussed the location of code, using user interface components as examples. The chapter suggested that user interfaces should not be created on the server side but on the client side. So 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:

- 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 (e.g., it may not have the Swing or even the AWT libraries).
- There may be many potential user interfaces for any particular service. For example, the Palm Pilot, with its small grayscale screen, requires a different interface from a high-end workstation with huge screen and enormous number of colors. It is not reasonable to expect the service to create every possible interface, 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 entry 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 it was given one during *its* construction (on the service side), it would end up with a service-side copy of the service instead of a client-side copy. So when it is asked for a user interface (on the client side), it should be passed the service as well in a parameter to user interface creation. In fact, it should probably be passed all of the information about the service, as retrieved in the `ServiceItem` obtained from a lookup service.

A typical factory is the one that returns a `JFrame`. This is defined 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 for the interface to compile and be exported. An implementation of this interface will probably use many more. The `roleObject` passes in any necessary information to the UI constructor. This is usually the `ServiceItem`, which contains all the information (including the service) that was retrieved from a lookup service. The factory can then create a UI that acts as an interface to the service, and it 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 this should not make the component visible, in order to allow the component's size and placement to be set before showing it. Similarly, a "playable" UI, such as an audio file, should not be in a playing state.

# Current Factories

A service may supply lots of factories, each capable of creating a different user interface object. This is to allow for the differing capabilities of viewing devices, or even for different user preferences. For instance, one user may prefer 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 factories currently includes the following:

- `DialogFactory` returns an instance of `java.awt.Dialog` (or one of its subclasses), which depends on AWT but not Swing.
- `FrameFactory` returns an instance of `java.awt.Frame` (or one of its subclasses), which depends on AWT but not Swing.
- `JComponentFactory` returns an instance of `javax.swing.JComponent` (or one of its subclasses, such as a `JList`).
- `JDialogFactory` returns an instance of `javax.swing.JDialog` (or one of its subclasses).
- `JFrameFactory` returns an instance of `javax.swing.JFrame` (or one of its subclasses).
- `JWindowFactory` returns an instance of `javax.swing.JWindow` (or one of its subclasses).
- `PanelFactory` returns an instance of `java.awt.Panel` (or one of its subclasses), which depends on AWT but not Swing.
- `WindowFactory` returns an instance of `java.awt.Window` (or one of its subclasses), which depends on AWT but not Swing.

These factories can be extended by any user, but to allow wide understanding, any new factories should be approved by the Jini community.

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 interfaces will be known only 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 may 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 wireless or home network link. It may also cost memory, which might be scarce in small devices such as PDAs. So 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.

Factories are wrapped in a `MarshalledObject` so they are hidden. The `MarshalledObject` 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. By putting the factory into entries in this form, no attempt is made to download its classes until it is unmarshalled.

The decision as to whether or not to unmarshall a class can be made based 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 it's manageable:

```
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 this test, unmarshalling, or attempted coercion. Using strings isn't really type-safe, and it does place a responsibility on service-side programmers to ensure this coercion will be correct. In effect, this maneuver circumvents the type-safe model of Java purely for optimization purposes.

There is one final wrinkle involved in loading the class files for a factory: a running JVM may have many class loaders. When loading the files for a factory, you will 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 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 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;
}
```

I haven't mentioned several features in the `UIDescriptor` yet, and the factory's type appears to be missing (it is one of the `attributes`). The second constructor has four parameters: `role`, `toolkit`, `attributes`, and `factory`. We have already discussed the factory. The following sections now discuss the other three.

## Toolkit

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

This isn't a bulletproof means of selection. For example, the Java Media Framework (JMF) is a fixed-size package designed to handle lots of different media types. So if your user interface is a QuickTime movie, then you might specify the JMF package. However, the media types the JMF handles are not fixed and 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. So 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 for narrowing lookups based on the UIs offered.

## Role

There are many possible roles for a user interface; the `role` is intended cover this. The role is specified as an interface, with one field, `role`, that is the fully qualified path name of the interface. There are currently three 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 is presentable 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 user interface 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 the role interface.
- 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 `UIDescriptor`s it produces follows these rules, but how it actually does so is not specified. There are several possibilities, including the following:

- 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 `roleObject`s are (presently) the service items, it could search through these for the `UIDescriptor`, and then check that its `role` matches what the factory expects.

There has been much discussion about role "flavors," such as an "expert" role or a "learner" role. This discussion has been deferred, as it's 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 is deemed useful. Currently this includes the following:

- A `UIFactoryTypes` contains a set of `String`s for the fully qualified class names of the factory this entry contains. The current factory hierarchy is very shallow, so this may be just a singleton set, such as `{JFrameFactory.TYPE_NAME}`.

  There is an unfortunate wrinkle with Java sets and Jini entries: the lookup matching mechanism for entries tests byte equality of serialized forms, and most implementations of the `Set` interface are not constant across all JVMs. So instead of common set types such as `HashSet`, you should use special types such as `com.artima.lookup.util.ConsistentSet`. This type has a constructor that takes a `Set` in its constructor, which makes already messy code a bit more messy:

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

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

- Inclusion of an `AccessibleUI` object is a statement that the user interface implements `javax.accessibility.Accessible` and that the user interface works well with assistive technologies.
- A `Locales` object specifies the locales supported by the user interface.
- A `RequiredPackages` object 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 is it a guarantee that the interface will be usable! But it may help a client decide whether or not to use a particular user interface.

# File Classifier 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 the file classifier here, too, by supplying simple user interfaces into the service. Such a user interface consists of a text field to enter a file name and a display of the MIME type of the file name. There is only a "main" role for this service, as no administration needs to be performed.

Figure 24-1 shows what a user interface for a file classifier could look like.

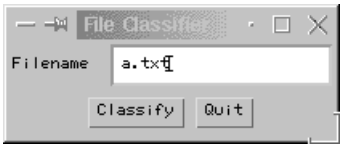


**Figure 24-1.** *File classifier user interface*

After the service has been invoked, it could pop up a dialog box as shown in Figure 24-2.

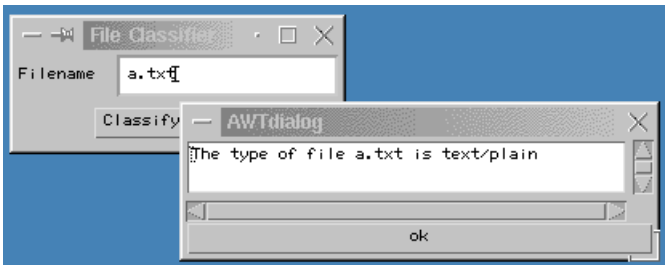


**Figure 24-2.** *File classifier return dialog box*

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

```
/**
 * 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 following is the user interface object that performs this role:

```
/**
 * 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);
        this.item = item;
        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 creates only one such object for a single user interface, but if the server exported more interfaces, then it would simply create more descriptors.

```
/**
 * FileClassifierServer.java
 */
package ui;
import complete.FileClassifierImpl;
import java.rmi.RMISecurityManager;
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;
import com.artima.lookup.util.ConsistentSet;
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() {
        System.setSecurityManager(new RMISecurityManager());
        JoinManager joinMgr = null;
        // The typenames for the factory
        Set typeNames = new HashSet();
        typeNames.add(FrameFactory.TYPE_NAME);
        typeNames = new ConsistentSet(typeNames);
        // The attributes set
        Set attribs = new HashSet();
        attribs.add(new UIFactoryTypes(typeNames));
        attribs = new ConsistentSet(attribs);
        // 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,
                                           null);
            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 it does a search through the `Entry` objects looking for a `UIDescriptor`. Once the client has a descriptor, it can check if that descriptor meets the requirements of the client. Here, we will check if 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.ServiceDiscoveryManager;
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() {
        ServiceDiscoveryManager clientMgr = null;
        System.setSecurityManager(new RMISecurityManager());
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            clientMgr = new ServiceDiscoveryManager(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 = null; // {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);
        }
        if (item.service == null) {
            // found a broken service
            System.out.println("service is null");
            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 UIDescriptor's
        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;
                }
                System.out.println("calling frame with " + item);
                Frame frame = factory.getFrame(item);
                frame.setVisible(true);
            }
        }
    }
}

} // TestFrameUI
```

■ **Note** To use the service UI, additional classes are required that are not in the standard Jini distribution. These need to be downloaded from `http://www.artima.com/jini/serviceui/index.html`.

# Images

User interfaces often contain images. They may be used as icons in toolbars, for general images on the screen, or for the application's desktop icon image. 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 the convenience class `ImageIcon`. This class can be instantiated from a byte array, a file name or, most interestingly, from a URL. So if an image is stored where an HTTP server can find it, then the `ImageIcon` constructor can use this image directly. There may be failures in this approach: the URL may be incorrect or malformed, or the image may fail to exist on the HTTP server. Suitable code is as follows:

```
ImageIcon icon = null;
try {
    icon = new ImageIcon(new URL("http://localhost/images/mindstorms.jpg"));
    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

User interface code exported by the server may use the preceding code to directly to include images. Alternatively, the service may supply useful images and other human-oriented information in a `ServiceType` entry object and leave it to the client to use it.

```
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 of `ServiceType` with useful implementations of the methods. However,`ServiceType` is a useful class that could be used to supply images and information that may be common between a number of different user interfaces for a service, such as a minimized image.

# Summary

The Jini specification from Sun did not include a specification for Jini user interfaces. The work described in this chapter was developed as a Jini Community standard. The key is that a user interface will be passed in an `Entry`. However, there are many considerations when taking this approach, such as not downloading classes that the client cannot handle. This chapter has examined these often subtle issues and shown how you can build user interfaces for services that can be used properly by clients.