The ServiceUI Project

Jini Service UI Draft Specification

Version 1.0

Revision History

- 1.0 - Mon, Apr 24, 2000, Made no changes other than to change the version number of the specification from 0.9.2 to 1.0, to match the version number of the code that is now released under the Sun Community Source License (SCSL).
- 0.9.2 - Sun, Mar 12, 2000, Fixed typo: Changed `Administratable` to `Administrable`. Fixed grammer: In last paragraph of section 3.1, changed "had" to "had had".
- 0.9.1 - Mon, Mar 6, 2000, Made no changes to the specification other than changing the version number from 0.9.0 to 0.9.1 to match the revised version number of the code. I did make changes to the code in version 0.9.1, to fix some bugs, which are described on the Code Access page.
- 0.9.0 - Tue, Feb 22, 2000, Fixed four grammar typos, which did not change meaning of the specification. In section 3.2, paragraph 2, altered second sentence so that the service item with which a main UI is associated must be accepted as the role object, rather than is required as the role object. In section 3.3, paragraph 2, altered second sentence so that the service item with which a admin UI is associated must be accepted as the role object, rather than is required as the role object. In section 3.4, paragraph 2, altered second sentence so that the service item with which a about UI is associated must be accepted as the role object, rather than is required as the role object. In section 3.7, paragraph 2, changed "simply shouldn't extend" to "simply shouldn't in general extend". In section 3.7, added paragraph 3, which starts, "Nevertheless, if some party...".
- 0.8.1 - Wed, Feb 9, 2000, In `MainUI`, `AdminUI`, and `AboutUI`, changed constant `role` to `ROLE`. In all factories, changed `toolkit` to `TOOLKIT` and `typeName` to `TYPE_NAME`.
- 0.8.0 - Tue, Feb 8, 2000, One last pass. Mostly minor tweaks except for the last few sections, which had some major surgery.
- 0.7 - Mon, Feb 7, 2000, Made a once over and performed much clean up. A few sections near the end are still a bit rough. Will post version 0.8 tonight, which will serve as the starting point for the two-week serviceui review phase.
- 0.6 - Sat, Feb 5, 2000, Fleshed out remaining sections, though some of it is rough.
- 0.5 - Thu, Feb 3, 2000, Fleshed out material in most sections. Added a new section on third-pary UIs.
- 0.4 - Wed, Feb 2, 2000, Incorporated changes decided upon at the 2nd meeting with Jini architects. Document now gives a good general overview.
- 0.3 - Mon, Jan 31, 2000, Fixed a few typos in Main UI semantics.
- 0.2 - Sun, Jan 30, 2000, Added introduction, revised semantics of Main/Admin/About UI roles, dropped `Flavor` and `FlavorSelectorUI`.
- 0.1 - Fri, Jan 28, 2000, First version.

Table of Contents

1. Introduction

Traditionally, desktop applications have been designed with a user interface (UI) built into the application. The code for the UI of the application is often highly coupled to the code that implements the functionality of the application. Over time, tentacles of UI code may burrow deep into functionality code, and tentacles of functionality code may burrow back into UI code. In the traditional desktop application, therefore, the UI code and functionality code are usually inseparable, married for all eternity.

Jini's service object architecture encourages a different way of thinking about UI and functionality:

- UI and functionality should be kept separate.
- Both UI and functionality are encapsulated in objects.
- The point of coupling between UI code and functionality code should be the service object interface.

A Jini service object should represent the pure functionality of the service, expressed via the methods of its object interface. The service object interface for a toaster service, for example, should express or model the pure functionality of the conceptual toaster service -- the interface should say "what it means to be a toaster." The service object should not supply any way for human users to access to the toaster service. Such access should be provided by a separate "UI object."

1.1. Direct-Use Clients

One reason for keeping UI out of the service object is that it enables client programs (clients) to access the service without human intervention or supervision. Clients written by programmers who knew about the (potentially well-known or standard) interface of a particular service object can interact with a service directly. As shown in Figure 1, client code can interact with a service directly by invoking the methods in the service object interface. Such a client is called a "direct-use" client, because the programmers of such a client can use their knowledge of the service object interface to write code that uses the service directly via that interface.

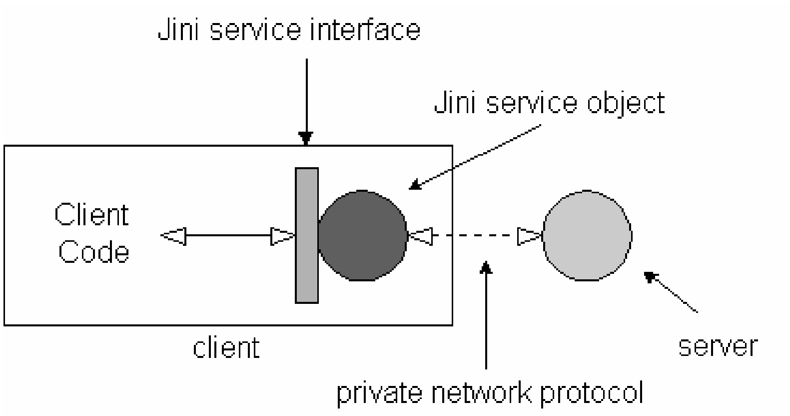

Figure 1. A "direct-use" client talks to a service through its object interface.

Direct-use clients need not be completely devoid of a user. For example, a user could be using a device that acts in certain cases as a direct-use client. Were the user to request that the device save some data in a file, the device could go get a storage service object and directly invoke methods on the storage service object to save the data in a file, with no further intervention of the user. In this case, the device acts as a direct-use client of storage services even though it has a user, because the programmers of the device had prior knowledge of the storage service object interface and used this knowledge to program the device to use storage services directly.

On the other hand, direct-use clients can also operate entirely independently of human supervision or intervention. Such clients act as "autonomous agents," which decide for themselves when to enlist the help of services. When an autonomous agent uses a service, it invokes the methods offered by the service object interface directly. Thus, the programmers of an autonomous agent must have prior knowledge of the service object interfaces their agent uses. (Although it is actually the human programmer that has the prior knowledge when he or she writes the client code, in this document the code itself will often be said to have prior knowledge.)

1.2. User Adapters

When you design a Jini service, you should attempt to capture the entire functionality of the service in the service object interface. To access any aspect of the service you are providing, a direct-use client should need only one thing: a reference to the service object. The service object should not include any code that defines a user interface to the service, just code that provides the functionality of the service at the level of method invocations.

To provide a user interface for the service, you should encapsulate the user interface code in a separate "UI object." As shown in Figure 2, a UI object grants a user access to some aspect of a Jini service. Think of a service UI object as a "user adapter" -- an object that *adapts* the interface of the service object, which a human user can't interact with directly, into some form that a human user can interact with directly. Sitting between the interface offered by the service object and a human user that wants to use the service, a UI object grants a user access to a service.

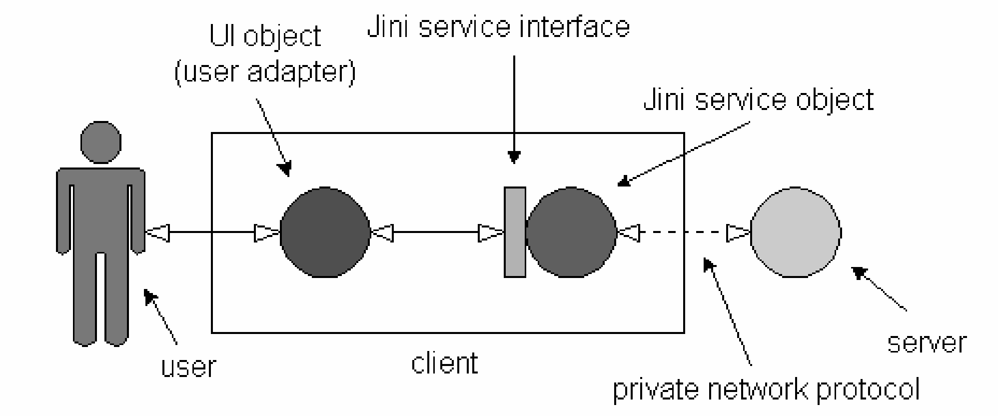

Figure 2. A user interacts with a service via a UI object.

A UI object can represent a graphical UI component, such as an AWT `Frame` or Swing `JPanel`, but need not necessarily be graphical. A UI object could also represent a speech interface, a text interface, a combination of speech and graphics, or a 3D immersive world. (The UI object is called "UI object," rather than "GUI object" or "view object," because the object is intended to represent user interface components in general, not just graphical user interface components.) Any kind of Jini service UI, even a 3D immersive world with speech and a virtual text display, should be represented by a single UI object, which is distinct from the service object.

One advantage of this architecture, in which UI and functionality are kept separate, is that multiple UIs can be associated with the same service. Associating multiple UIs with a single service enables different UIs to be tailored for clients that have particular UI capabilities, such as Swing or speech. Clients can then choose the UI that best fits their user interface capabilities. In addition, it may be desirable to associate with a single service different UIs that serve different purposes, such as a main UI or an administration UI. Often, it will be desirable to associate multiple UIs with a single Jini service, where each UI has a particular purpose and is geared towards a particular set of client capabilities.

1.3. Jini Service UIs

This specification defines a standard way for UI providers to associate a UI (user adapter) object with a Jini service, and shows how client programs can find the best fit UI among multiple UIs associated with the same service. To associate a UI with a Jini service, UI providers (primarily, service providers, but also possibly third parties) must supply three items:

- the UI itself
- a "UI factory" that produces the UI
- a "UI descriptor" that describes the UI

A UI factory is an object that has one or more factory methods that produce and return a UI object. A UI descriptor, an instance of `net.jini.lookup.entry.UIDescriptor` (`UIDescriptor`), serves as a container for the UI factory and other objects that describe the UI produced by the factory. Because `UIDescriptor` implements `net.jini.core.entry.Entry` (`Entry`), a `UIDescriptor` can be included in the attribute sets array of a Jini service item, thereby associating the UI with the service. A UI descriptor contains four public fields:

- `factory` -- a reference to a `java.rmi.MarshalledObject` that contains the UI factory object in marshalled form
- `attributes` -- a `java.util.Set` of attribute objects that describe the UI produced by the factory
- `toolkit` -- a `String` that gives the name of the main package of the UI toolkit required by the UI
- `role` -- a `String` that gives the fully qualified name of a Java interface type that represents the role played by the UI

The purpose of the `attributes`, `toolkit`, and `role` fields is to describe the UI produced by the marshalled UI factory. Using these fields, clients can choose from among multiple UIs associated with a service. Once they make a selection, clients can use the other field, `factory`, to create the UI object they selected.

To associate a UI with a Jini service, the UI provider must first fill a UI descriptor with a role `String`, a toolkit `String`, a set of attributes, and a marshalled UI factory. The UI provider must then place the UI descriptor in the attribute sets of the service item of the service with which the provider wants the UI to be associated.

A UI's role indicates both its purpose and the portion of its semantics that is related to its role. Each UI role is defined as a Java interface type, which is implemented by UI objects that play that role. The `role` field references a `String` whose value is the fully qualified name of the role interface type. For example, the purpose of a `net.jini.lookup.ui.MainUI` role UI is to provide general access to a service. The purpose of a `net.jini.lookup.ui.AdminUI` role UI is to enable a user to administer a service. Although most of the UI object semantics of interest to the client program is usually defined by the return type of the UI factory, some semantics may be tied to the role. For example, for a role that represents a dialog with the user, the role interface might include methods that allow the client to determine the result of the dialog.

The `attributes` field of the UI descriptor holds a reference to a `java.util.Set` of serializable objects that contain information about the UI represented by the descriptor. These objects can be of any class, so long as they are serializable. Clients can search through the attributes set for attribute types about which they have prior knowledge, and use those objects to help them decide whether or not to attempt to use the UI represented by the UI descriptor. Several attribute classes, which are defined in the `net.jini.lookup.ui.attribute` package, are described later in this document.

The toolkit field serves to facilitate matching when doing queries on the Jini lookup service. For example, if a client is interested in all services that have main UIs that work with the Swing toolkit, it could form a net.jini.core.lookup.ServiceItem (ServiceItem) whose attributeSetTemplates array includes a UI descriptor template. If the UI descriptor template's factory and attributes fields were set to null, role field were set to "net.jini.lookup.ui.MainUI", and toolkit field were set to "javax.swing", the lookup would return only services that had at least one main UI for Swing.

The factory field contains the UI factory object inside a MarshalledObject so that the class files for the factory and the UI it produces can be placed in a different codebase than the service item containing the UI descriptor. By placing the class files for UI factories and UI objects in separate codebases than that of the service item, clients can download a service item without needing to download all the JAR files containing all the class files for UI factories and UI objects, most or all of which the client may never use. Only if the client decides to attempt to use a UI need it unmarshal the UI factory, which would trigger the downloading of the JAR file containing the class files for the factory, and possibly the class files for the UI the factory will generate.

To use a UI, a client program must have prior knowledge of both the UI factory type (including of course the type of UI object returned by its factory methods) and the UI role type. These types define the UI object semantics, which client programmers must understand before they can write code that will properly interact with the UI. A client need not have prior knowledge of all the types of attributes that appear in the attributes set. To the extent the client does have prior knowledge of the attribute types, it can use that prior knowledge to help it select an appropriate UI from among multiple UIs associated with a service.

2. UI Descriptors

A net.jini.lookup.entry.UIDescriptor (UIDescriptor) is a net.jini.core.lookup.Entry (Entry) that enables a UI for a service to be associated with the service in the attribute sets of the service item. A UIDescriptor holds a marshalled UI factory object in the factory field, a role type name in the role field, a toolkit package name in the toolkit field, and set of attributes, which describe the UI generated by the factory, in the attributes field.

The purpose of a UIDescriptor is not only to provide a place for a marshalled UI factory that can produce a UI, but also to describe the produced UI to help clients decide whether or not to unmarshal the UI factory. The description of the UI is contained in the role, toolkit, and attributes fields.

The public parts of class UIDescriptor look like this:

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

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

```
public class UIDescriptor extends net.jini.entry.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 {...}
}
```

Because the marshalled UI factory is referenced from a public field, to unmarshal the factory client programs can simply invoke `get()` directly on the marshalled object. Clients must take care, however, to ensure that the class loader that loads the class files for the UI is also able (likely via a parent class loader) to load the class files of the service object with which the UI will be interacting. The `UIDescriptor` includes a convenience method named `getUIFactory()` to help clients unmarshal the UI factory with the proper class loader context.

The `getUIFactory()` method saves a reference to the current context class loader, sets the context class loader to the class loader passed as `parentLoader`, invokes `get()` on the marshalled object, then resets the context class loader to the saved reference before returning the object produced by `get()`. Because setting and getting the context class loader is a privileged operation, the `getUIFactory()` method wraps those operations in `AccessController.doPrivileged()` calls.

The class loader passed to the `getUIFactory()` method in the `parentLoader` parameter should be able to load types (classes and interfaces) needed when the UI interacts with the "role object," which is passed as the first parameter to any factory method. The semantic description of a UI role indicates, among other things, what object should be passed to the factory as the role object. For the `net.jini.lookup.ui.MainUI` role, for example, the role object is the service item. Thus, to unmarshal a UI factory for a main UI, the class loader passed to `getUIFactory()` should be able to load types needed when the UI interacts with the service item. For example, the client could pass to `getUIFactory()` the class loader the client previously used to load the service item.

The `String` referenced from the `toolkit` field, which gives the name of the main package of the primary UI toolkit used by the UI, is determined by the UI factory type. The semantics of each UI factory type should include a toolkit string, so UI providers will know what string to put in the `toolkit` field for their selected UI factory type. Two toolkit strings currently defined are `"java.awt"`, for graphical UIs that depend on AWT but not Swing, and `"javax.swing"`, for graphical UIs that depend on Swing (and AWT, since Swing is built on top AWT).

3. UI Roles

The `role` field of the `UIDescriptor` gives the fully qualified name of the interface that represents the role of the UI generated by the marshalled UI factory. If the client program unmarshals the UI factory and invokes a factory method, the UI returned by the factory method must implement the role interface specified by `role`.

For a client program to be able to use a UI, the client has to have prior knowledge of the UI semantics, a portion of which is defined by the UI's role type. Thus, for a client to be able to use a UI, the client must understand the semantics of the type whose fully qualified name is appears in the `role` field of that UI's `UIDescriptor`.

For example, three role types that are defined in the `net.jini.lookup.ui` package by the Jini Service UI Specification are `MainUI`, for a main UI to a Jini service; `AdminUI`, for an administration UI; and `AboutUI` for an about UI. Other role types may be defined by future incarnations of the Jini Service UI Specification, by individual Jini service API specifications, or by any other party. The fully qualified names of UI role types should, as with any other type, follow the recommended naming convention for unique packages outlined in the Java Language Specification:

> You form a unique package name by first having (or belonging to an organization that has) an Internet domain name, such as sun.com. You then reverse this name, component by component, to obtain, in this example, com.sun, and use this as a prefix for your package names, using a convention developed within your organization to further administer package names.

For example, printers in general don't have a main UI, but they may have a UI for printer setup, a UI for print job setup, and a UI for administering the printer. The printer working group, therefore, could define two role types as part of their Printer API definition, one for printer setup and one for print job setup. (The printer's administration UI could likely use the already existing `net.jini.lookup.ui.AdminUI` role.) The printer working group would start their role type names with some package prefix to which they have been given or have obtained control. For example, if the Jini Community grants the printer working group the right to place their APIs in the `net.jini.print` package, the printer working group could name their `role` types: `net.jini.print.ui.PrinterSetup` and `net.jini.print.ui.PrintJobSetup`. Each working group or other entity that defines Jini Service APIs can include as part of their specification any new UI role types that are useful in the context of their kind of service.

3.1. Displaying Roles to Users

As the strings referenced from the `role` field are Java type names, they are intended to be manipulated by client programs only. They should not be shown to a user. A client may, nevertheless, display localized strings representing roles about which the client has prior knowledge.

For example, imagine a Jini browser client shows a list of services using icons and names provided by `net.jini.lookup.entry.ServiceType` (`ServiceType`) entries. Such a Jini browser could, when the user double-clicks on a service icon or name, attempt to display a `MainUI` for the service. In addition, such a Jini browser could, when the user right-clicks on a service icon or name, pop up a list of verb strings, one for each UI role that is offered by the service in a form the client believes it can use. For example, imagine that among the UIs offered by the service upon which the user right-clicked are three `java.swing.JFrame` (`JFrame`) UIs (produced by `net.jini.lookup.ui.factory.JFrameFactory`s), one for each of three roles defined in the `net.jini.lookup.ui` package: `MainUI`, `AdminUI`, and `AboutUI`. If the client is able to use `JFrame`s, the client could display three verbs on the pop-up list when the user right-clicks. In the

UI-English locale, the verb strings could be: `"Open..."`, `"Administer..."`, and `"About..."`. If the user selects `"About..."`, the client could display the `AboutUI` `JFrame`.

Note that the strings shown to the user -- `"Open..."`, `"Administer..."`, and `"About..."` -- were not provided by the service. These strings were decided upon by the developers of the Jini browser client. Given that these developers had prior knowledge of the `MainUI` role, they were able to decide that in their client, `"Open"` would be a sensible verb string for `MainUIs` in the US-English locale. Of course, those developers may have selected other verb strings for other locales. German users of the same browser, for example, could potentially be presented with the verb string `"Jetzt Geht's Los..."` for `MainUIs`, `"Was Ist Das Ding..."` for `AboutUIs`, and `"Spielen Sie Mit Die Viele Kleine Nummern"` for `AdminUIs`, etc.

Because the verb strings are provided by the client program, and not by the Jini service, the client will be unable to show a verb in its list for any role about which it did not have prior knowledge. Thus, if the service upon which the user right-clicked also offers a `JFrame` UI with the role `net.jini.blender.ui.FrappeUI`, the client would be able to display a localized string for that UI in its list of verbs only if the developers of the client had had prior knowledge of that role. If the client program was not endowed with prior knowledge of `FrappeUIs` by its programmers, the client will not be able to list a verb for that UI in its pop-up, and therefore the user will not be able to select it. This prior-knowledge requirement is intentional, because as mentioned previously in this document, the role defines a portion of the semantics of the UI. Before the client program can properly use a UI, it must have prior knowledge of the role interface. If a client doesn't know how to use a UI, it makes no sense to let the user select that UI.

3.2. The `net.jini.lookup.ui.MainUI` Role

`net.jini.lookup.ui.MainUI` (`MainUI`) is a UI role interface implemented by main UIs, which enable client programs to grant users general access to a service. If a UI descriptor's UI factory produces a UI that implements this interface (i.e., produces a main UI), the UI descriptor's `role` field must reference a `String` with the value `"net.jini.lookup.ui.MainUI"`.

The first parameter of any factory method declared in a UI factory type is an object called the "role object." Any factory method that produces a main UI must accept as the role object the service item (the `net.jini.core.lookup.ServiceItem`) of the service with which the main UI is associated.

Main UIs should allow clients to configure them before they begin. For example, main UIs produced by `FrameFactory`, `JFrameFactory`, `WindowFactory` and `JWindowFactory` (all members of the `net.jini.lookup.ui.factory` package) should not be visible when they are returned from the factory method. This allows clients to set the UI's position and size, for example, before making the UI visible by invoking `setVisible(true)` on the UI.

A client should be able to invoke a main UI factory method multiple times sequentially. In other words, if a user uses a service via a main UI, then says exit, then double clicks once again on the service icon, the client can simply invoke a UI factory method again, and get another main UI for the same service. Main UIs, therefore, should be written so that they work no matter what state the service object happens to be in when the main UI is created.

It is recommended that clients use multiple main UIs for the same service only sequentially, and avoid creating multiple main UIs for the same service object that operate concurrently with one another. But because some clients may create and use multiple main UIs at the same time for the same service object, providers of services and main UIs should program defensively, to ensure that multiple main UIs for the same service object at the same time will all work together concurrently.

Here's the `net.jini.lookup.ui.MainUI` tag interface:

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

public interface MainUI {

    String ROLE = "net.jini.lookup.ui.MainUI";
}
```

3.3. The `net.jini.lookup.ui.AdminUI` Role

`net.jini.lookup.ui.AdminUI` (`AdminUI`) is a UI role interface implemented by admin UIs, which enable users to administer a service. If a UI descriptor's UI factory produces a UI that implements this interface (i.e., produces an admin UI), the UI descriptor's `role` field must reference a `String` with the value `"net.jini.lookup.ui.AdminUI"`.

The first parameter of any factory method declared in a UI factory type is an object called the "role object." Any factory method that produces an admin UI must accept as the role object the service item (the `net.jini.core.lookup.ServiceItem`) of the service with which the main UI is associated.

Admin UIs have precisely the same semantics as main UIs. The only difference is their purpose. Here's the `net.jini.lookup.ui.AdminUI` interface:

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

public interface AdminUI {

    String ROLE = "net.jini.lookup.ui.AdminUI";
}
```

3.4. The `net.jini.lookup.ui.AboutUI` Role

`net.jini.lookup.ui.AboutUI` (`AboutUI`) is a UI role interface implemented by about UIs, which enable users to view (or in some way experience) information about a service. If a UI descriptor's UI factory produces a UI that implements this interface (i.e., produces an about UI), the UIDescriptor's `role` field must reference a `String` with the value `"net.jini.lookup.ui.AboutUI"`.

The first parameter of any factory method declared in a UI factory type is an object called the "role object." Any factory method that produces an about UI must accept as the role object the service item (the `net.jini.core.lookup.ServiceItem`) of the service with which the main UI is associated.

About UIs have precisely the same semantics as main UIs. The only difference is their purpose. Here's the `net.jini.lookup.ui.AboutUI` interface:

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

public interface AboutUI {

    String ROLE = "net.jini.lookup.ui.AboutUI";
}
```

3.5. Defining New Role Types

As mentioned previously, any party may define new role interfaces. New UI role interfaces will likely often be defined in conjunction with the definition of new Jini Service APIs, and many of those roles will likely represent service-specific dialogs with users. As used here, a "dialog" is a short conversation with the user, usually to obtain some information from the user. Although for graphical toolkits, a dialog would often be implemented with a dialog box, such as AWT's `Dialog` or Swing's `JDialog`, the term dialog is used here in the generic sense, not strictly in the graphical sense. Service-specific dialog UIs will enable clients to enlist the help of a user at various points throughout an otherwise direct use of a service.

As an example, consider a user asking a word processor to print a document, where the word processor has prior knowledge of a well-known Jini Printer API. (Note that the types that appear in this example were invented for illustration purposes. At the time of this writing, the Jini Printer Service API had not been finalized.) To print via the Jini Print Service API, the word processor first obtains a `net.jini.print.service.PrintService` (`PrintService`) object and invokes `createPrintJob()` on that object to obtain a `net.jini.print.job.PrintJob` (`PrintJob`) object. A purely direct-use client must do two things with the `PrintJob` object, configure the print job and supply the print data, before invoking `close()` on the print job, thereby queueing the job for printing. Although the word processor could potentially do both of these jobs directly, the word processor may not have prior knowledge of the portion of the `PrintJob` object that enables the client to configure the print job, and besides, users are accustomed to configuring print jobs. Given that a user is available, the word processor would likely want to provide a dialog UI that enables the user to configure the print job. Once the user completes his or her configuration and dismisses the dialog UI (with an "OK", not a "Cancel"), the word processor could supply the print data and invoke `close()` on the `PrintJob`.

Given that a dialog UI that enables users to select configuration parameters for print jobs will likely be a common desire of clients of the Jini Print Service API, the printer working group could decide to define a new UI role type for that purpose. The printer working group would likely place the role interface in some subpackage of the main package for their API. For example, they could define an interface named: `net.jini.print.ui.PrintJobSetup` (`PrintJobSetup`).

A `PrintJobSetup` UI would represent a dialog with the user that enables the user to select configuration parameters for the print job, such as page orientation, etc. Once the user is finished selecting configuration parameters, the user dismisses the dialog UI, and the dialog UI in some way communicates back to the client what the user's configuration wishes are (likely by invoking methods on the `PrintJob` object itself, which is likely the role object of for the `PrintJobSetup` role) and indicates to the client that the dialog is finished. The semantics of the `PrintJobSetup` interface would define how the client interacts with the UI, such as how the client knows the UI has been dismissed, and whether or not the user dismissed the UI with an "OK" or a "Cancel". If the client gets an OK, the client could then continue on and write print data to the `PrintJob` object, and invoke `close()` on the `PrintJob`, thereby sending it to the printer.

The way in which the `PrintJobSetup` UI interacts with the client (i.e., communicates the configuration data and indicates the dialog has been dismissed) is defined by the semantics of the `PrintJobSetup` role interface. As the dialog may require some sophisticated interaction between UI and client, the role interface may be more than just a tag interface. The role interface may include methods that define how the UI and client interact. The reason the word processor client would know how to put up this `PrintJobSetup` dialog, is that the `PrintJobSetup` role interface, and its semantics, would be defined as part of the Jini Print Service API, of which the word processor client has prior knowledge.

3.6. Working with Existing Role Types

Besides creating new role types for service-specific dialogs, it is also conceivable that a working group for some Jini service API, or a provider of an implementation of some Jini service API, may want to provide multiple incarnations of an already-defined role. For example, what if the manufacturer of a combined printer/scanner/faxer/copier product (a "four-in-one box") wanted to deliver four `AdminUI`s, each dedicated to administering one of the four functions of the product? Such a manufacture could take several approaches.

First of all, assuming the Jini community has adopted a standard API for each of the four functions of the product (i.e., a Jini Print Service API, a Jini Scanner Service API, a Jini Faxer Service API, and a Jini Copier Service API), the manufacturer of the four-in-one box could, rather than registering a single Jini service whose service object implements four interfaces (such as `Printer`, `Scanner`, `Faxer`, and `Copier`), could simply offer four separate Jini services, a each of which implements only one of those interfaces. In this approach, each service could offer an `AdminUI` dedicated to that one function.

Alternatively (or in addition), the manufacturer could register a single Jini service whose service object implements all four interfaces. The `AdminUI` for that four-in-one service could present some user interface device (such as a tabbed pane, if the UI is graphical) that lets users select between the four main functions. In this way, a single actual `AdminUI` exists, which in some way grants users access to the four conceptual administration UIs.

Lastly, the manufacturer could go to the Jini community and propose a Jini Four-In-One Service API, and propose that four new role types, each dedicated to administering a single subset of the product, be included as part of that API. The role types could be named `PrinterAdminUI`, `ScannerAdminUI`, `FaxerAdminUI`, and `CopierAdminUI`, each of which is place in some package that is controlled by the Four-In-One Working Group that is defining the Jini Four-In-One Service API. Clients with prior knowledge of the Jini Four-In-One Service API, would also have prior knowledge of the four role types, and could therefore offer verb strings for them. Providers of Four-In-One services could offer those four kinds of admin UI, but should also likely offer a basic `net.jini.lookup.ui.AdminUI` as well, which as described previously, grants a user access to all four kinds of admin UI. Including an `AdminUI` enables clients that don't have prior knowledge of the Four-In-One service to offer a UI that will allow their users to administer the Four-In-One service.

3.7. Choosing Superinterfaces for New Role Types

Role interfaces should not in general extend other role interfaces, because only one role interface is described in the `role` field of the UI descriptor. For example, the `FaxerAdminUI` from the previous section seems like it could extend the `AdminUI`, because a `FaxerAdminUI` could be considered to be a kind of `AdminUI`. However, only one role interface can appear in the `role` field of the UI descriptor. If a UI is both a `FaxerAdminUI` and an `AdminUI`, then which role would go in the `role` field? If the same UI is registered twice, once with `AdminUI` and once with `FaxerAdminUI`, then what is the point of `FaxerAdminUI`? In other words, if a `FaxerAdminUI` is the "default" admin UI of the four, then the four-in-one working group needn't have bothered defining a `FaxerAdminUI` in the first place. All they'd need to do is mention that the `AdminUI` should just behave as an admin UI for the fax machine.

Role interface can extend other interfaces -- they simply shouldn't in general extend other role interfaces. If the semantics of a dialog UI role are already defined in some interface, then the role interface could extend that interface and thereby inherit those methods and semantics.

Nevertheless, if some party does decide to create a role interface that extends another role interface, any factory method that generates the subinterface UI role type must accept all the same categories of role object accepted by factory methods that generate the superinterface UI role type. For example, if `FaxerAdminUI` were declared as a subinterface of `AdminUI`, any factory method that produces a `FaxerAdminUI` would have to accept as the role object the service item with which the `FaxerAdminUI` was associated, because that category of role object is accepted by factory methods that produce `AdminUI`s.

It is recommended that all role interfaces include a compile time `String` constant named `ROLE`, which gives the fully qualified string type name of the UI role interface, as it should appear in UI descriptor's `role` fields. Such convenience constants not only help reduce typing and increase code readability, they also leverage compile-time type checking to help minimize the chance of typographical errors in `role` strings.

4. UI Attributes

The UI descriptor's `attributes` field references a `java.util.Set` of objects that describe the UI generated by the marshalled UI factory. Any object can appear in this set, so long as it is serializable. (Note that if any object in this set is not serializable, the entire set will not appear at the client side -- the UI descriptor's `attributes` field will be `null`.)

If a UI provider wishes to register a UI descriptor that has no attributes, it may register the UI descriptor with either a reference to an empty `java.util.Set` or `null` in the UI descriptor's `attributes` field. Nevertheless, because clients would know so little about the UI represented by a UI descriptor with no attributes, many clients would likely ignore the UI descriptor entirely.

Although all attributes are optional, this specification recommends that all UI descriptors include at least the attributes (all of which are members of the `net.jini.lookup.ui.attribute` package) described in this list:

- One `UIFactoryTypes` attribute, which lists Java types of which the marshalled UI factory is an instance.
- One `RequiredPackages` attribute, which lists packages the UI requires to be already installed at the client.
- One `Locales` attribute, which lists the locales supported by the UI.
- If (and only if) the UI supports the accessibility API, one `AccessibleUI` attribute.

The remainder of this section describes these four attribute classes in detail.

4.1. The `AccessibleUI` Attribute

`AccessibleUI` is a UI attribute that indicates a generated UI implements the `javax.accessibility.Accessible` interface and that the designer of the UI did the necessary work to make sure the UI would work well with assistive technologies that are aware of the Java Accessibility API.

The attribute should appear in the attributes set of a `UIDescriptor` only if the marshalled UI factory produces a UI that supports the Accessibility API. The presence of this attribute in an attributes set means the produced UI will work well with assistive technologies that are aware of the Java Accessibility API.

The `AccessibleUI` attribute looks like this:

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

public class AccessibleUI implements java.io.Serializable {
}
```

4.2. The `Locales` Attribute

`Locales` is a UI attribute that lists locales supported by a generated UI.

Zero to many `Locales` instances may appear in the attributes set of a UI descriptor. UI providers are encouraged to provide in the attributes set of any UI descriptor a single `Locales` object that contains a complete set of the locales supported by the UI. Given that UI providers are not required to give any information about locales, complete information about locales, or even accurate information about locales, clients should program defensively and consider the supported locales to be a strong hint at what locales are supported by the UI, but not necessarily 100% complete or accurate.

The public portion of `Locales` looks like this:

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

import java.util.Locale;
import java.util.Iterator;
import java.util.Set;
import java.util.List;
```

```
public class Locales implements java.io.Serializable {

    public Locales(Set locales) {...}
    public boolean isLocaleSupported(Locale locale) {...}
    public Locale getFirstSupportedLocale(Locale[] locales) {...}
    public Locale getFirstSupportedLocale(List locales) {...}
    public Iterator iterator() {...}
    public Set getLocales() {...}
}
```

4.3. The `RequiredPackages` Attribute

`RequiredPackages` is a UI attribute that enables clients to get a list of the fully qualified names and version numbers of packages required by a UI.

Zero to many `RequiredPackages` attributes may appear in the attributes set of a `UIDescriptor`. Client programs interested in a UI may wish to verify that they have all required packages mentioned in the `RequiredPackages` attributes (if any) contained in the UI's `UIDescriptor`, before they attempt to create the UI. If the client is lacking any required packages (either because the entire package is absent or because the package is present but of an incompatible version), the client will not be able to use the UI.

The intent of this attribute is to provide a quick way for a client program to determine that a UI is unusable by a client, not to provide a guarantee that a UI is definitely usable by the client. If a client is missing a required package, or has an incompatible version of a required package, the client cannot use the UI. But if the client has compatible versions of all required packages listed in a `RequiredPackages` attribute, the client may or may not be able to use the UI.

UI providers should attempt to list in a `RequiredPackages` attribute all packages that must already be installed at the client for the UI to work, so that if the client discovers it has compatible versions of all listed packages and attempts to generate the UI via the factory method, the client will likely succeed. (Note that packages used by the UI that could potentially be installed at the client, but are also available at the UI's or service's codebase should not be listed in a `RequiredPackages` attribute. Such packages are not actually required of the client, because if the client doesn't have them, the client can download them.)

Client programmers should bear in mind that a `RequiredPackages` attribute doesn't necessarily list *all* required packages. As a result, satisfying all required packages doesn't absolutely guarantee the UI will work on the client. Client programs should therefore program defensively. (For example, clients should probably catch `LinkageError` in appropriate places when dealing with UIs, even if they find they have compatible versions of all required packages listed in `RequiredPackages` attributes.)

The version numbers listed in `RequiredPackages` attributes must take the form of "specification version numbers," as used by the `java.lang.Package` class:

> Specification version numbers use a "Dewey Decimal" syntax that consists of positive decimal integers separated by periods ".", for example, "2.0" or "1.2.3.4.5.6.7". This allows an extensible number to be used to represent major, minor, micro, etc versions. The version number must begin with a number.

Here's what the public portion of `RequiredPackages` looks like:

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

import java.util.Map;
import java.util.Set;
import java.util.Collection;
import java.util.Iterator;

public class RequiredPackages implements java.io.Serializable {

    public RequiredPackages(Map packages) {...}
    public Iterator iterator() {...}
    public String getVersion(String packageName) {...}
    public Map getRequiredPackages() {...}
}
```

4.4. The `UIFactoryTypes` Attribute

`UIFactoryTypes` is a UI attribute that allows client programs to determine Java types of which a UI factory (marshalled in the same `UIDescriptor`) is an instance.

Zero to many `UIFactoryTypes` may appear in the attributes set of UI descriptors. The UI factory contained in marshalled form in the `factory` field of the same UI descriptor should be an instance of each type that appears in the list stored in a `UIFactoryTypes` attribute.

Of all the attributes that could appear in the attributes set, `UIFactoryTypes` is perhaps the most important. If a UI descriptor's attributes set includes no `UIFactoryTypes` attribute, then the only way for clients to know what kind of factory a UI descriptor represents is to unmarshal it. But unmarshalling a UI factory will likely involve downloading code, which a client wants to avoid unless it is fairly certain it will be able to use the UI. As a result, UI providers are strongly encouraged to include in every UI descriptor a single `UIFactoryTypes` attribute that includes those Java types (of which the UI factory is an instance) that clients would likely be looking for. In general, clients will be looking for well-known factory interface types, such as those that appear in the `net.jini.lookup.ui.factory` package.

The public portion of `UIFactoryTypes` looks like this:

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

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

public class UIFactoryTypes implements java.io.Serializable {

    public Types(Set typeNames) {...}
    public boolean isAssignableTo(Class classObj) {...}
    public Iterator iterator() {...}
    public Set getTypeNames() {...}
}
```

5. UI Factories

The UI factory object should implement one or more UI factory interfaces appropriate to the type of UI generated (the return value of the factory method) and the options clients have at creation time (the parameters the client can and must pass to the factory method). Factory interfaces can be implemented in many ways, but in general, factory objects should use lazy instantiation of the UI object. For example, rather than instantiating a UI object when the factory is instantiated, and returning that already-instantiated UI when a factory method is invoked, a factory object should wait until a factory method is actually invoked before intantiating the UI object. Lazy instantiation enables the factory object to be marshalled without requiring the image of the UI object to be included in the marshalled image of the UI factory.

UI factories define methods that indicate by their return type the Java type of UI generated and indicate by their parameters the objects and primitive values that can and must be supplied by the client to the factory. The first parameter to any UI factory method is an object called the "role object". The category of this object (but not the type) is determined by the UI's role. As used here, category means how the client program gets ahold of the role object. For example, a role might specify that its role object is the service item, or the service object, or an object that is obtained by invoking a method on the service object, etc. The role of the UI describes how the client gets ahold of the object that must be passed as the first parameter to the factory method. For example, the semantics of the `MainUI` role indicates that the role object is the service item.

UI factories should be written such that their factory methods can be invoked multiple times. A client may wish to show a user the same UI several times, and therefore may invoke a factory method on the same factory object multiple times. Each time the factory method is invoked, it should produce and return a new copy of the UI object.

Eight factory interfaces are defined in the first version of the Jini Service UI API:

- `DialogFactory` - Returns an instance of class `java.awt.Dialog`, or one of its subclasses, that depends on the AWT, but not the Swing, libraries.
- `FrameFactory` - Returns an instance of class `java.awt.Frame`, or one of its subclasses, that depends on the AWT, but not the Swing, libraries.
- `JComponentFactory` - Returns an instance of class `javax.swing.JComponent`, or one of its subclasses, that depends on both the AWT and Swing libraries.
- `JDialogFactory` - Returns an instance of class `javax.swing.JDialog`, or one of its subclasses, that depends on both the AWT and Swing libraries.
- `JFrameFactory` - Returns an instance of class `javax.swing.JFrame`, or one of its subclasses, that depends on both the AWT and Swing libraries.
- `JWindowFactory` - Returns an instance of class `javax.swing.JWindow`, or one of its subclasses, that depends on both the AWT and Swing libraries.
- `PanelFactory` - Returns an instance of class `java.awt.Panel`, or one of its subclasses, that depends on the AWT, but not the Swing, libraries.
- `WindowFactory` - Returns an instance of class `java.awt.Window`, or one of its subclasses other than `Frame` or `Dialog`, that depends on the AWT, but not the Swing, libraries.

Other UI factory interface types may be defined by future incarnations of the Jini Service UI Specification or by any other party. The fully qualified names of UI factory interface types should, as with any other type, follow the recommended naming convention for unique packages outlined in the Java Language Specification:

> You form a unique package name by first having (or belonging to an organization that has) an Internet domain name, such as sun.com. You then reverse this name, component by component, to obtain, in this example, com.sun, and use this as a prefix for your package names, using a convention developed within your organization to further administer package names.

Here's an example UI factory interface:

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

import javax.swing.JFrame;

public interface JFrameFactory extends java.io.Serializable {

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

    JFrame getJFrame(Object roleObject);
}
```

Given that only one toolkit field exists, all `UIFactory` interfaces implemented by a UI factory class will need to have the same toolkit.

5.1. Defining New Factory Types

Although anyone is allowed to define new factory types, to prevent a chaotic explosion of factory types, new factory types should be defined only with the utmost care. In general, new factory types will be justified only when new UI toolkits appear on the scene. If possible, new factory types should be agreed upon by the Jini Community and placed into the `net.jini.lookup.ui.factory` package, so they are easy to find. For example, if a toolkit is ever defined for speech-only UIs, a set of factory methods for speech-only UIs could be added to those already in `net.jini.lookup.ui.factory`. If a graphical toolkit is defined for extemely small screens, a set of factory methods for small-screen UIs could be added to those already in `net.jini.lookup.ui.factory`.

A UI factory must be an interface, and each method in the interface must take an `Object` called the `roleObject` as its first parameter. Other than the role object, nothing should be passed to a factory method except information or context required by the toolkit.

It is recommended that all UI factory interfaces include two compile time `String` constants:

- `TOOLKIT` -- gives the package name of the toolkit used by the produced UI which should appear in the UI descriptor's `toolkit` field
- `TYPE_NAME` -- gives the fully qualified string type name of the UI factory, as it should appear in `UIFactoryTypes` attributes

Such convenience constants not only help reduce typing and increase code readability, they also leverage compile-time type checking to help minimize the chance of typographical errors in `toolkit` strings and type names added to `UIFactoryTypes` attributes.

6. UI Objects

UI objects should be user adapters, as shown in Figure 2. Two temptations that service and UI providers may have are:

- putting UI code in the service object
- putting service code in the UI object

Both of these temptations should be avoided. The Jini service object should contain the functionality of the service, the whole functionality of the service, and nothing but the functionality of the service. No user-interface code should appear in the service object, and no functionality of the service should appear in the UI object unless it also appears in the service object. In short, Jini service UIs should merely serve to make available to users all or part of the functionality available to direct-use clients via the methods of the Jini service object.

Note that even though service item may be passed as the role object for some roles -- as it is for example to factories that produce `MainUI`s, `AdminUI`s, and `AboutUI`s -- the full *functionality* of the service should nevertheless be available exclusively through the service object. The service item may be passed as the role object to UI factories because extra information (but not extra functionality) could be embodied in the other parts of the service item, the service ID and attribute sets, and the UI may wish to display that extra information to the user.

It is strongly encouraged that service and UI providers decouple the class files required by a particular service item from the class files required by any UIs that may be associated with that service via UI descriptors in the service item. If the class files for the service item and those of all the UIs associated with the service via UI descriptors are placed into the same JAR file, then a client will have to download all the class file for all the UIs just to download the service item, even if the client never uses a single UI.

6.1. What to Provide

This specification does not mandate any types of UIs to be associated with services. Service providers are free to provide any types of UIs, or no UIs, with their services. Nevertheless, some discussion of client expectations may help service providers decide what types of UIs to supply.

First of all, if a service provider only supplies one UI, it should likely be an embeddable component, such as an AWT `Panel` or Swing `JComponent`. Although Swing UIs can be more attractive than AWT UIs, to have the greatest reach with a single UI type, an AWT `Panel` may make the most sense.

Some clients may wish to show a UI without popping up another window. A `Panel` allows them to do that, whether they support AWT or Swing. If a client wants to pop up a UI, they can embed a `Panel` UI in a new `Frame` and pop that up. If a client wants to display a title and icon for the `Panel`, the client could use the localized name and icon provided by a `ServiceType`.

A `Panel` doesn't usually provide a way for the client to exit, and one wouldn't normally expect to see a menubar at the top of a `Panel`. So UI providers may also want to provide either a `Frame` or a

`Window` in which the `Panel` is placed, and to which other UI elements such as menu bar, status bar, and title, etc., are added. If this approach is taken, a single factory object could implement both the `PanelFactory` and the `FrameFactory` interfaces, because their toolkit is the same. This might make sense if the `Panel` is used in the `Frame`, and the `Frame` doesn't add many classes to the panel. Alternatively, the `Panel` and `Frame` could be produced by two different UI factories sitting in two different UI descriptors.

A UI provider may also wish to support Swing in addition to, or instead of, AWT. If so, at a minimum a Swing `JComponent` UI might make sense. Or, both a `JComponent` and either a `JFrame` or a `JWindow` might make sense. If the second approach is taken, a single factory object could implement both the `JComponentFactory` and the `JFrameFactory` interfaces, because their toolkit is the same. Alternatively, the `JComponent` and `JFrame` could be produced by two different UI factories sitting in two different UI descriptors.

On the other hand, for those UI roles that represent dialogs with the user, an AWT `Dialog` or Swing `JDialog` may make the most sense.

Currently no direct support exists in the Jini Service UI API for speech-only interfaces, but an AWT or Swing UI could be enhanced with speech capabilities, produced by an appropriate AWT or Swing UI factory type, and identified as speech-enabled with the appropriate required package names (such as `javax.speech`, etc.) in the `RequiredPackages` attribute. If a speech-enhanced GUI is provided, it is likely a good idea to also provide a GUI-only version, as many clients won't have speech capabilities.

6.2. Avoiding Service-Level Data and Client Context

Two kinds of data may be tempting to pass to UI objects, either directly or via factory methods (which would then make it tempting to define new factories) is service-level data and client-context data. It is strongly recommended that both of these kinds of temptation be avoided.

For example, imagine a service provider requires a user name and password for its service. The service provider wants clients to be able to save a user's name and password locally and enter them automatically the next time the user uses the service. If no user name and password were previously saved, the service UI would prompt the user to type them in with a logon dialog box. But if the user name and password were saved, the user would never see the logon dialog box requesting the user name and password. Rather, the client program would log on automatically with the saved user name and password, and the user would just see the service UI that appears after a successful logon.

To implement this functionality, a service provider might be tempted to define interfaces the UI object implements that allow the client to pass a saved user name and password to the UI. Alternatively, a service provider might be tempted to define new factory methods that enable clients to pass a user name and password to the factory, which could then hand them to the UI.

The trouble with either of these approaches is that functionality is being put into the UI that isn't in the service. As mentioned previously, the full functionality of a Jini service should be modeled in the service object, so direct-use client code can just use the service directly. If a service requires a user name and password, the interface of the service object should allow the client to provide the user name and password. Thus, a client with a user could give the user name and password to the service object first, then create the UI. The UI could ask the service object if it has a user name and password yet. If not, the UI would show the logon dialog box prompting for the user name and password. Otherwise, the UI wouldn't show the logon dialog box.

The recommended way to design a Jini service and its UIs is to make sure the full functionality is available via the service object interface, and keep the UIs focused on being user adapters -- adapting some or all of the functionality that is available via the service object interface, but offering no more functionality than is available via the service object interface.

Service providers are not the only members of the Jini Community that will face UI-related temptations in the years ahead, client providers will also face temptation. In particular, client providers may find it tempting to pass client context data to UI objects.

For example, a client provider wishing to differentiate his or her product from the competition might add a toolbar and status bar to the client program. The client provider might find it quite tempting to define interfaces that UIs could implement that allow the client to pass references to the toolbar and status bar to the UI. Alternatively, the client might find it tempting to define new factory interfaces whose factory methods include parameters that allow the client to pass references to the toolbar and status bar to the factory, which could then pass it along to the UI. UIs that obtain references to the toolbar and status bar in either of these two ways could then put buttons on the toolbar and write messages to the status bar.

The trouble lurking behind this seemingly innocuous value-add on the part of the client provider is that the toolbar and status bar references represent client context data, which complicates the job of UI providers. If other clients devise their own toolbar and status bar interfaces, and perhaps even up the ante by also defining other client context innovations, UI providers will have to worry about which kinds of client context to support. To the extent a UI provider supports various kinds of client context, that UI provider's test matrix will expand. To the extent that client context becomes complicated, such as the invention of a general compound document model for Jini service UIs, everyone's job, and the user's experience, becomes more complicated.

Client providers, therefore, should be strong and avoid the temptation to invent client context for Jini Service UIs. The only context data that should be passed to UI factory methods is context required by a UI toolkit, such as, for example, the `Frame` or `Window` reference required by an AWT `Dialog`. Similarly, the only context in which Jini Service UIs should be embedded is context provided by the toolkit. For example, a client could embed a Jini Service UI `JComponent` in a `JFrame`, and pop up the `JFrame`. This embeddability of the `JComponent` is provided by its toolkit, Swing.

UI providers can do their part by ensuring that their UIs don't depend on any context other than that provided by the UI toolkit. Jini Service UIs should be self-contained and not care anything about context other than their toolkit. If a UI provider wants its UI to write messages to a status bar, the UI should not go looking for a status bar provided by the client, the UI should itself include a status bar.

7. Adding Third-Party UIs

To add a third party UI to the attribute sets of an existing service, you must have the cooperation of the service. The service can veto your request at any of several points in the process. The steps are:

1. Check to see if the service object implements `net.jini.admin.Administrable` (`Administrable`).

   If the service object does not implement `Administrable`, the service provider has already thwarted your wishes. You won't be able to add a third party UI directly to the attribute sets of the service item.

2. Cast the service object to `Administrable`, and invoke `getAdmin()`.
3. Check to see if the admin object returned by `getAdmin()` implements the `net.jini.admin.JoinAdmin` (`JoinAdmin`).

   If the admin object does not implement `JoinAdmin`, the service provider has thwarted your wishes. You won't be able to add a third party UI directly to the attribute sets of the service item.

4. Cast the service object to `JoinAdmin`, and invoke appropriate methods to add or modify one or more `UIDescriptor`s.

   You add or modify `UIDescriptor`s in the same manner as you would add or modify any kind of `Entry` in the service's attribute sets. The `JoinAdmin`'s `addLookupAttribute()` method allows you to add new `UIDescriptor`s. Its `modifyLookupAttribute()` method allows you to modify existing `UIDescriptor`s. Once again, the service provider can veto your request even at this step, by throwing an exception from these methods.

Once a new `UIDescriptor` for a third-party UI is successfully added via the process described previously, the third party need never add it again. From that point forward, whenever a service registers itself with a lookup service, its service item will include the third-party `UIDescriptor`. It will also reregister itself with any lookup services with which it is currently registered, so that the new `UIDescriptor` will appear there as well. Even if the service provider crashes and restarts, it is supposed to remember the new `UIDescriptor` and include it in its registrations after the crash and restart.

On the other hand, the UI, UI factory, and possibly the attributes, require class files to be available in some codebase. Although the third party need never worry about the `UIDescriptor` being included in future registrations of the service, the third party does need to make sure the class files are forever available at some codebase.

One potential problem with adding a third party UI is that in a strict client side security policy, downloaded code will by default only have permission to connect back to the host from which the code was downloaded. In that case, if the third-party UI's codebase is on a different host than the service object's codebase, the service object will not be allowed to talk back to its host at the behest of the UI. Therefore, for a third-party UI to work in practice, either the codebase of the third-party UI must be on the same host that serves the codebase of the service, or the client must specifically relax its security policy in order to use the third-party UI.

8. Evolution and Prior Knowledge

In the future, the Service UI API can evolve in several different ways. New role types and the meanings of those role types can be defined by any party. New attribute classes can be defined by any party. New factory types can be defined, and new toolkit types can be identified. For clients to be able to user these new types, the new types will need to be publicized. Before a client will be able to make use of a role, factory, or attribute type, the programmers of that client would need to have had prior knowledge of the type.

Other than service-specific UI roles created as part of Jini Service APIs, new roles should come from the Service UI project at the Jini Community. Service Providers should resist the temptation to define new role types, as that complicates the job of clients. Clients should be able to program to the interface (particular Jini Service APIs about which they have prior knowledge), rather than being forced to program to one or more implementations of that interface. Whether role types are being defined for particular Jini Service APIs, or globally for the Jini Service UI API, the preferred way to define the role interfaces is via the Jini Community process.

As mentioned previously, new UI factory interface types should be defined with great care. The main justification for a new set of UI factory types is when new UI toolkits appear on the scene. The preferred way to define these interfaces is via the Jini Community process. Similarly, although anyone can define new attribute types, the preferred way to define these types is through the Jini Community process.

The reason the Jini Community is the preferred way to define new role, factory, and attribute types is that it is the best way to avoid an explosion of types. For example, if a certain attribute type is needed, and twenty different parties define basically the same attribute, but with twenty slightly different names, twenty slightly different behaviors, and placed them in twenty different packages, then clients that want to use that kind of attribute will need to support all twenty variations. It would be far better if those twenty parties would get together and agree on a single attribute class. Enabling such collaborations is the main purpose of the Jini Community organization. If a single attribute class can be agreed upon by all twenty parties, then clients would need to keep track of, understand, and use less information (one attribute class versus twenty), and the resulting client programs would likely work more reliably.