CHAPTER 4

Entry Objects

A service is exported to lookup services based on its class. Clients search for services using class information, typically using an interface. There is often additional information about a service that is not part of its class information, such as who owns the service, who maintains it, where it is located, and so on. Entries are used to pass this kind of additional information about services to clients. The clients can then use that information—as well as class type—to decide if a particular service is what it wants.

Entry Class

When a service provider registers a service, it places a copy of the service object (or a service proxy) on the lookup service. This copy is an instance of a class, albeit in serialized form. The server can optionally register sets of attributes along with the service object. Each set is described by an `Entry` object. What is stored on each service locator is an instance of a class along with a set of `Entry` objects, each of which describes some special additional attributes of the service.

For example, a set of file editors may be available as services. Each editor is capable of editing different types of files, as shown in Figure 4-1.

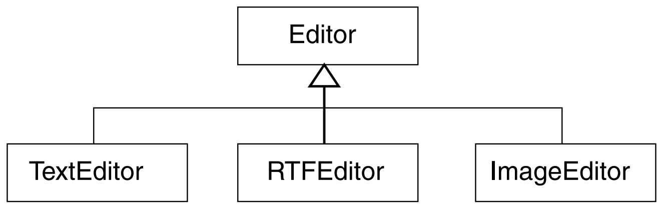

Figure 4-1. Editor class diagram

NOTE *These classes would probably be interfaces, rather than instantiable classes.*

In this situation, a client could search for a suitable editor in two ways:

- By asking for an instance of a specific class, such as `ImageEditor`
- By asking for an instance of the general class `Editor` with the additional information that it can handle a certain type of file

The type of search performed depends on the problem domain, as defined by the services and possible clients. Services advertise themselves by exporting an object that is of a particular class and by exporting additional information along with this object. Jini clients can then search for specific types of services by asking for an object that implements the specific class, such as `ImageEditor`. They can also search for superclass objects and include extra objects to narrow the search, based on additional information advertised by the service. This additional information is given in objects belonging to subclasses of the `Entry` class.

The `Entry` class allows services to advertise their capabilities in very flexible ways. For example, suppose an editor was capable of handling a number of file types, such as plain text *and* RTF files. It could do so by exporting a service object implementing `Editor` along with an `Entry` object saying that it can handle plain text and another `Entry` object saying that it can handle RTF files. The service implementation can just add more and more information about its capabilities without altering the basic interface.

To manage this way of adding information, we could have a `FileType` class, which would give information about the types of files handled:

```
public Class FileType implements Entry {
    public String type; // this is a MIME type

    public FileType(String type) {
       this.type = type;
    }
}
```

For a text editor, the attribute set would be `FileType("plain/text")`. For an RTF editor, the attribute set would be `FileType("application/rtf")`.

For an editor capable of handling both plain text and RTF files, its capabilities would be given by using an array of entries:

```
Entry[] entries = new Entry[] {new FileType("plain/text"),
                               new FileType("application/rtf")
                              };
```

On the other side, suppose a client wishes to find services that can handle the attributes that it requires. The client uses the same `Entry` class to do this. For any particular `Entry`, the client specifies both of the following:

- Which fields must match exactly (a non-`null` value)
- Which fields it does not care about (a `null` value)

For example, to search for a plain text editor, an entry like this would be used:

```
Entry[] entries = new Entry[] {new FileType("plain/text")};
```

If any editor would do, the following entry could be used:

```
Entry[] entries = new Entry[] {new FileType(null)};
```

Attribute Matching Mechanism

The attribute matching mechanism is pretty basic. For example, a printer typically has the capacity to print a certain number of pages per minute, but if it specifies this using an `Entry`, it actually becomes rather hard to find. A client can request a printer service in which it does not care about speed, or it can request a particular speed. It cannot ask for printers with a speed greater than some value. It cannot ask for a printer without a capability, such as anything except a color printer. An attribute must either match exactly or be ignored. The relational operators such as "<" and "!=" are not supported.

If you want to search for a printer with a particular speed, then printer speed capabilities may need to be given simpler descriptive values, such as "fast," "average," or "slow." Then, once you have a "fast" printer service returned to the client, it can perform a query on the service, itself, for the actual speed. This would be done outside of the Jini mechanisms, using whatever interface has been agreed on for the description of printers. A similar problem, that of finding a physically "close" service, is taken up in Chapter 13.

The attribute matching mechanism that was chosen by the Jini designers, of exact matches with wildcards, is comparatively easy to implement. It is a pity from the programmer's view that a more flexible mechanism was not used. One suggestion often made in the Jini mailing list is that there should be a `boolean matches()` method on the service object. However, that would involve unmarshalling the service on the locator in order to run the `matches()` method, and this would slow the lookup service down and generate a couple of awkward questions:

- What security permissions should the filter run with?

- What would happen if the filter modifies its arguments (deep copying to avoid this would cause further slowdowns)?

The `ServiceDiscoveryManager`–discussed in Chapter 15—has the ability to do client-side filtering to partly rectify this problem.

Restrictions on Entries

Entries are shipped around in marshalled form. Exported service objects are serialized, moved around, and reconstituted as objects at some remote client. Entries are similarly serialized and moved around. However, when it comes to comparing them, this is usually done on the lookup service, and they are not reconstituted on the lookup service. So when comparing an entry from a service and an entry from a client request, it is the serialized forms that are compared.

An entry cannot have one of the primitive types, such as `int` or `char`, as a field. If one of these fields is required, then it must be wrapped up in a class such as `Integer` or `Character`. This make it easier to perform "wildcarding" for matching (see Chapter 5 for details). Jini uses `null` in the fields of `Entry` objects from the client to act as a wildcard. This will work for any class, including wrapper classes such as `Boolean`. The primitive types, such as `boolean`, have no values that can be used as a wildcard pattern, since all possible values (`true` and `false`) could be valid request values.

Jini places some further restrictions on the fields of `Entry` objects. They must be public, non-static, non-transient, and non-final. In addition, an `Entry` class must have a no-args constructor.

Convenience Classes

The `AbstractEntry` class implements the `Entry` interface, and it is designed as a convenience class. It implements methods such as `equals()` and `toString()`. An application would probably want to subclass this instead of implementing `Entry`.

In addition, Sun's implementation of Jini contains a further set of convenience classes, all subclassed out of `AbstractEntry`. These require the `jini-ext.jar` file. They are the following:

- `Address`–The address of the physical component of a service.

- `Comment`–A free-form comment about a service.

- `Location`–The location of the physical component of a service. This is distinct from the `Address` class, in that it can be used alone in a small, local organization.

- `Name`–The name of a service as used by users. A service may have multiple names.

- `ServiceInfo`–Generic information about a service. This includes the name of the manufacturer, the product, and the vendor.

- `ServiceType`–Human-oriented information about the "type" of a service. This is not related to its data or class types and is more oriented toward allowing someone to determine what a service (for example, a printer) does and if it is similar to another, without needing to know anything about data or class types for the Java platform.

- `Status`–The base class from which other status-related entry classes can be derived.

For example, the `Address` class contains the following:

```
String country;
String locality;           // City or locality name.
String organization;      // Name of the company or organization that provides thi
service.
String organizationalUnit; // The unit within the organization that provides this
service.
String postalCode;         // Postal code.
String stateOrProvince;    // Full name or standard postal abbreviation of a state
or province.
String street;             // Street address.
```

You may find these classes useful; on the other hand, what services would like to advertise, and what clients would like to match on, is pretty much unknown as yet. These classes are not part of the formal Jini specification.

Further Uses of Entries

The primary intention of entries is to provide extra information about services so that clients can decide whether or not they are the services the client wants to use. An expectation in this is that the information in an entry is primarily static. However, entries are objects, and they could also implement behavior as well as state.

This should not be used to extend the behavior of a service, since all service behavior should be captured in the service interface specification.

A good example of a “non-static” entry is `ServiceType`, which is an abstract subclass of `AbstractEntry`. This contains “human oriented” information about a service, and contains abstract methods, such as `String getDisplayName()`. This method is intended to provide a localized name for the service. Localization (for example, producing an appropriate French name for the service for French-speaking communities) can only be done on the client side and will require code to be executed in the client to examine the locale and produce a name.

Another use is to define the user interface for a service. Services do not have or require user interfaces for human users, since they are defined by Java interfaces that can be called by any other Java objects. However, some services may wish to offer a way of interacting with themselves by means of a user interface, and this involves much executable code. Since it is not part of the service itself, this should be left in suitable `Entry` objects. This topic is looked at in detail in Chapter 19.

Summary

An entry is additional information about a service, and a service may have any number of entries. Clients request services by class and by entries, using a simple matching system. There are a number of convenience classes that subclass `Entry`.