A DIRECTORY SERVICE FOR THE CERN PS/SL JAVA PROGRAMMING INTERFACE

J.Cuperus, P.Charrue, F.Di Maio, K.Kostro, CERN, Geneva, Switzerland;
W.Watson, TJNAF, Newport News, USA

Abstract

The CERN PS and SL accelerator control groups developed a common application programming interface (API) in Java [1]. Part of this API is a directory service that provides information about the underlying hardware and software. With this information it is possible to write generic programs that do general actions on lists of devices without hard coding of device names. And, starting from a device name, full details about related devices, the device itself and its class and properties, can be obtained, including the meaning of bits and bitpatterns in status words. The interface definition is independent of any implementation but a reference implementation is provided using Java Database Connectivity (JDBC) against a set of tables in a relational database. Data from very different systems can be brought together and presented in a uniform way to the user. The full potential of the directory service is reached when it is used in software components (Java Beans).

1 INTRODUCTION

The Java programmer (the user) sees the accelerator devices through an input/output service [1] that is object oriented in the sense that it calls methods for device classes but it has a narrow interface, meaning that there is a single interface for all classes. Such an interface is well adapted to generic programming but it is not possible to do introspection through it in the Java sense. Anyway, the user should have access to more information about the details of accelerator devices, the properties of their classes, and the relations between devices, than would be possible by introspection of Java classes. To bring this information to the user, the directory service was made (fig.1).

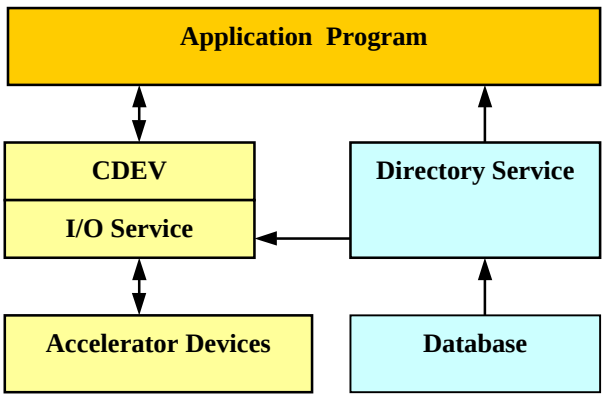

Figure 1 : I/O service and directory service

This service consists of an interface definition that is independent of any implementation plus an implementation that is adapted to the available data. We will first describe the capabilities of the interface.

2 THE DIRECTORY INTERFACE

The directory interface is shown on fig.2.

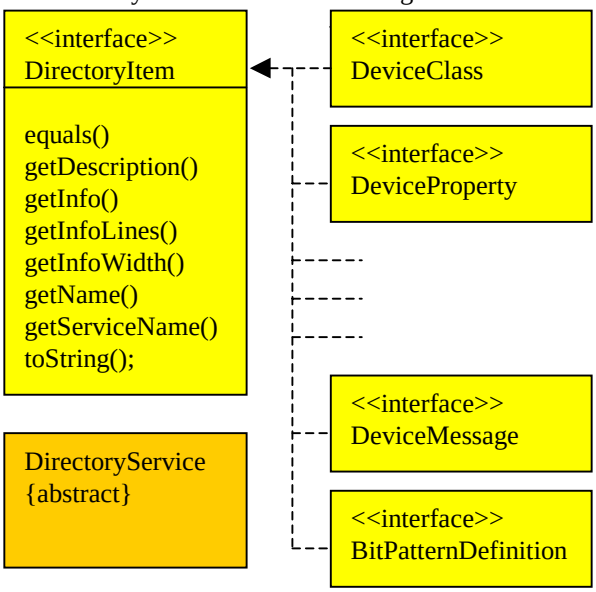

Figure 2: The directory service interface consists of two main parts: the **DirectoryService** class that can return a DirectoryItem object when given an Identifier and the **DirectoryItem** interface, and 7 interfaces that extend it, that return detailed information about the object

Before doing anything else, the user must get a reference to the DirectoryService singleton object with:

dir = DirServiceImplementation.getService();

After this, the user sees only the interface, not the implementation. Having not yet acquired any specific DirectoryItem objects, the user starts with a call to DirectoryService like:

object = dir.getSomething(myString);

where *myString* is an identifier or a query, and *object* is a name, or a DirectoryItem object, or an array of these. The form of a valid query is implementation dependent. In our implementation, it is a SQL query against the underlying relational tables: “devicename like ‘BR3%’ ” or: “classname=’POW’ and accelerator=’PSB’ ”. Where this dependence is undesirable, avoid queries in favour of

more specific identifiers. Suppose now that the user gets a DirectoryItem object like:

DeviceData dd = dir.getDeviceData("BR1.QNO");

The user can now get detailed information from this object with calls like:

DeviceClass class = dd.getDeviceClass().

2.1 DirectoryItem Interfaces

Once the user has obtained a reference to a DirectoryItem object from the DirectoryService interface, he can get further information about these objects through the interfaces that extend the DirectoryItem interface:

- **DirectoryItem**: defines some basic methods, extended by all following interfaces. The getInfo method returns a text object with summary information for browsers, help facilities, and bean editors.
- **DeviceClass**: to get all properties for the class or an ordered named subset of the properties. Classes may be organised with multiple inheritance and abstract classes.
- **DeviceProperty**: to get the attributes (type, dim, ...) and characteristics (min, max, format, units, ...) of the property. The characteristics may return either a value or a property for obtaining the value through the device access interface.
- **DeviceData**: to get values for accelerator device variables. Some of these variables are mandatory (such as the device class) and others are specific for the implementation. Dependence on other devices is obtainable through 'role' queries.
- **DeviceGroup**: to get the composition (title, devices, properties) for the named device group.
- **GroupList**: to get the DeviceGroups that make up the named list. A generic program can acquire a GroupList to get the set of devices it has to work on.
- **DeviceMessage**: to get the attributes (category, number, severity) and text parts (label, short text, long text) of the message. All kinds of messages, including error messages, labels, and help information, are possible.
- **BitPatternDefinition**: to get the meaning of bits, or groups of bits, in a binary word. The data for each bit, or bit pattern, are in the form of a DeviceMessage. In its simplest form, this reduces to a correspondence between integers and strings but considerably more complex patterns are possible.

2.2 Exceptions

Provisions must be made in case something goes wrong. Almost every call can throw a DataNotFoundException with subclasses:
- BadConnectionException,
- BadQueryException,
- NoSuchDataException,
- TooManyValuesException.

Handling exceptions in the application program is a lot of work but, as explained later, the application program will access the directory service mainly through components. It is these components that will do most of the exception handling and recovery.

3 IMPLEMENTATION WITH JDBC

Access to the implementation is through the singleton object of DirServiceImplementation. This object delegates all database access to the appropriate implementations of the DirectoryItem interfaces. Several sets of implementations of the DirectoryItem interfaces can coexist for different parts of the installation. Few installations will have complete data for all methods of the interface. In case of missing data, a suitable default value must be returned or else the NoSuchDataException must be thrown. Caching of certain data in hash tables is optional and objects returned by two identical queries must contain the same information but are not necessarily the same objects.

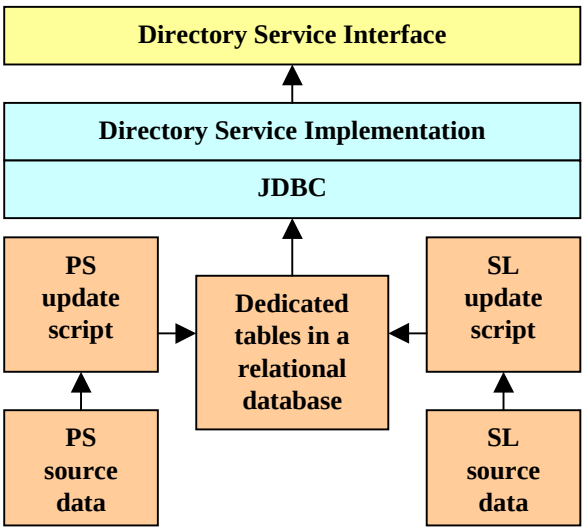

Figure 3: Directory Service Implementation

For the CERN accelerators, there is a single implementation, based on relational database access through JDBC (fig.3). A group of 10 database tables, dedicated to the directory service, covers all the data. Updating these tables from source data is done with scripts using transactions so that all data remain available to JDBC at all times in a coherent form. The source data, and the scripts, are different for the accelerators of the PS injector complex and those of the SL complex but the directory service provides a unified view of those two sites. A large part of the source data existed already, before the directory service project started (see [2] for the PS complex), but some data structures were modified and forms were used to fill missing data by hand.

4 APPLICATION COMPONENTS

The application programmer will normally work with components (Java beans) and will not see the directory service directly. The data in the directory service can be used to initialise, or configure, the components. An example is a component that displays I/O values for selected devices and properties. All configuration data can come from a DeviceGroup object. Scale factors, formats, units, min and max values, can be automatically extracted from the directory service.

Well designed components are extremely important and their easy implementation is the main reason for using the Java language. The close collaboration between components and the directory service needs to be stressed but only the DirQuery bean, which is directly related to the directory service, will be described here.

4.1 DirQuery Bean

The application programmer may occasionally want to see the directory service more directly while still working exclusively with components. An example is the simple data browser shown in fig.4. For these purposes, the DirQuery bean was made (fig.5).

The DirQuery objects receive one main DataEvent and, for some queries, an auxiliary DataEvent. The real input parameters for the query are:

dataEvent.getData().toString;

The main event will start a directory service query with these parameters and according to the query type. The possible query types are given by *final int* variables with names like:

CLASSNAME_DEVICENAMES

CLASSNAME_PROPERTIES . . .

The first part of the name indicates the input and the second part the output. Twenty query types cover most of the directory service.

5 BEAN CUSTOMISERS

A graphical design tool will introspect your bean and use its default so called customiser to present the designer with a GUI editor. The designer can use this customiser to set the configurable parameters of the bean to his needs. This customiser will look different in each design tool and will give only minimal help to the designer.

An accelerator control system will have a limited set of highly specialised beans and it is worthwhile to make a dedicated customiser for each of them, even if this may mean more work than designing the bean in the first place. This dedicated customiser can make heavy use of the directory interface for presenting the user on demand with valid options and useful context information.

6 CONCLUSIONS

The Directory Service is an essential complement to the Java equipment access API and permits generic programming. The full potential of the Java API is realised when used in Java beans for accelerator control systems, both in the hidden inner workings of these beans and in dedicated customisers. In that case, the user can concentrate on what he wants to obtain without bothering with the details of the interfaces or the brand of the design tool. Further effort should go into producing an effective set of such beans and their customisers.

REFERENCES

[1] F.Di Maio, P.Charrue, J.Cuperus, I.Deloose, K.Kostro, M.VandenEynden, W.Watson, The CERN PS/SL Application Programming Interface, this conference.

[2] J.Cuperus, M.Lelaizant, Integration of a Relational Database in the CERN PS Control System, ICALEPCS-97, November 1997, IHEP, Beijing, China.

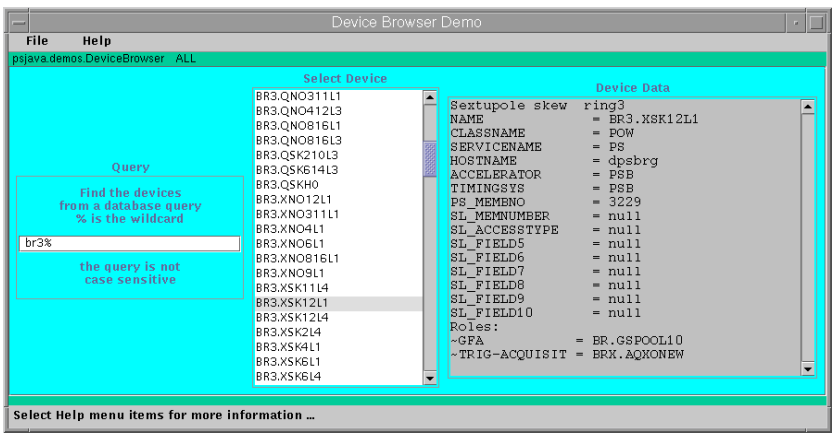

Figure 4: A simple data browser

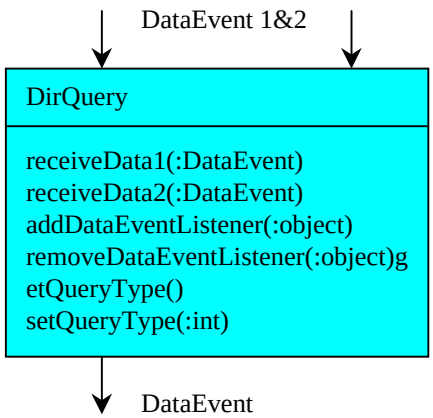

Figure 5: DirQuery bean events & methods