# A REFLECTION ON INTROSPECTION

M. Plesko, G. Tkacik*, Cosylab and J. Stefan Institute, Ljubljana, Slovenia

## *Abstract*

In this article we present our specific solutions of storing implicit knowledge about the control system in the control system itself and utilizing it to create the so called generic applications. We show an implementation of Abeans that allows to move away from naming conventions, channel and property lists and documentation (i.e. implicit knowledge used by human programmers) and rather stores that information digitally. Generic applications can thus access it programatically from code (i.e. what we call meta information). Those applications require absolutely no code porting when used at other control systems, because all system dependence is parametrized in terms of different meta descriptions for each system and managed by Abeans.

The goal of the project is to produce software common to the majority of the communication systems (archivers, browsers, multiple device displays, machine physics applications) that are exactly the same irrespective of the control system (i.e. CORBA ACS, TINE at DESY, EPICS at SNS).

To this purpose, we have extensively studied the nature of meta information and have developed the Abeans Directory, which contains semantics and APIs to manage meta data as part of our Abeans and CosyBeans frameworks. This article describes the interplay of Java Naming and Directory Interface (JNDI), meta descriptor objects, specialized GUI components and strategies of extraction of meta-information from the underlying middleware communication systems (e.g. CORBA, EPICS), that bring us closer to the dream of generic applications.

As an example we discuss ArchiveReader, an application produced in collaboration with DESY, which employs the enumerated concepts to completely separate the presentation of channel history data from the data access. As Abeans Directory is implemented also in full on EPICS and CORBA ACS platforms, the same application will access their archives in addition to the initially supported TINE and Abeans Simulator systems. We believe that the specific example discussed here will make the reflective concepts clearer and will facilitate their faster adoption in the wide control system community.

## INTRODUCTION

This paper should be read as a follow-up article to [1], where we, firstly, give a conceptual sketch of what meta-information is; secondly, discuss the conditions for its deployment in control systems; and lastly, briefly demonstrate how a generic application uses it to learn device structure during run-time. Building on that foundation, we can now focus on two further issues. In Section 2 we delve into some meta-data technicalities of Abeans [2] meta-libraries that were, for the sake of clarity, omitted in [1]. In Section 3 we show how the meta-data mechanism can be used to parameterize not only the basic controlled entities (e.g. devices, channels), but system-wide services as well. In Section 4 we conclude by pointing out directions for further research in reflective control systems.

## ABEANS META LIBRARIES

### *What Entities Do We Describe*

Meta-libraries in Abeans encompass meta-data collected from various sources (such as CORBA Interface Repository, XML structure descriptions, hardcoded info), rules for naming such information with Uniform Resource Identifiers (URIs) and a meta-API for accessing it. While running, meta-data are stored in a directory, called distributed or Abeans directory – consequently standard Java Naming and Directory Interface (JNDI) for accessing entries given their hierarchical unique names forms an integral part of Abeans meta-API.

Although we describe how services can be integrated into such structure only in the next section, *the ability* to integrate various data source descriptions (not only those of remote devices, but also of distributed services and local-virtual entities, for instance) was the major design goal of Abeans meta-libraries. Notice that if we broaden the scope of all describable data sources to include entities beyond devices and channels, we face the following two new issues:

**A possible distributed nature of the data source**. For example, when offering access to a naming service or a remote archive, both of these may be realized on multiple remote machines. To protect the user from this (mainly technical) detail, the meta-libraries do not expose it to the application programmer, but have to manage it internally to know where to look for data. This is, in other words, the issue of *federation*.

**A possible lack of expressive power in existing meta-libraries.** For example, getting archive data could be *functionally* so radically different from physical device data access that meta-API would fail to encapsulate it.

### *What Entities Do We Use for Description*

As opposed to Section 2.1, where we talked about Level 1 (as defined in [1]) content, we discuss Level 2 entities here, especially insofar they address the new potential problems. In Abeans meta-API, Level 2 entities are called descriptors and they are returned to the user when s/he requests meta-information by name from the directory. The canonical form of the interaction between the user and meta-API is as follows:

*E-mail: gasper.tkacik@cosylab.com

```
Directory d = <obtain directory as Abeans service>
URI uri = new URI("abeans-ACS://
server.ijs.si/linac/PSBEND_M.01");
Descriptor desc = d.lookup(uri);
// examine desc to obtain info on PSBEND_M.01
```

Although the steps are not reproduced in full syntax, they clearly show how the directory is to be used. The names themselves are accessible for listing in the directory: it is populated when Abeans plugs start up and examine their specific name servers or object lists.

## *URI names*

Without going into the details of URI specification [3], we will stress the points where using URIs benefit Abeans and potentially other control systems:

1. **Maintenance:** URI manipulation tools are freely available (integral part of Java platform), reducing parsing bugs.
2. **Flexibility:** not all URI parts must be present. In the example above we reference "server.ijs.si", because supposedly another equally named object could have existed on "alternate.ijs.si". If there is a uniqueness guarantee, the server name (formally URI authority part) may be skipped and is understood by default.
3. **Schema specification:** prefix "abeans-ACS" indicates that the data source being examined is in ACS plug running on Abeans platform. "abeans-archive" means that we are going to examine the remote archive part of the Abeans directory. In general, schema explicates the URI to such an extent that no additional information is required to interpret parts that follow (usually such information is implicit and we provide it by passing the name *to the object that knows how to interpret the name*).
4. **Hierarchical part:** apart from introducing hierarchy in an obvious way (even that can be beneficial if Abeans run on platforms that otherwise have flat namespaces), the hierarchy can be used for federation. In DESY, for example, there are different archive servers that we would like to access transparently. We introduce names such as "abeans-archive://TINE/HERA", "abeans-archive://TINE/PETRA" etc. and leave it to the plug to do translation from human-readable names "PETRA", "HERA" into the actual server names. In addition, all archive servers now reside under common "abeans-archive://TINE" directory and can be handled as a group. [4]
5. **Query part:** although absent in the example, Abeans interpret the query part as stating the kind of request that should be invoked on the data source (see the table of descriptor entities in Section 2.4 for details).

By using URIs, we can therefore name all physical entities, virtual entities that exist only locally (e.g. "abeans-archive" itself does not have a separate existence on any machine, it is an Abeans construct) and data in distributed services. Such URI name is by design sufficient to uniquely parametrize Abeans request to any target, and the corresponding response.

## *Descriptor entities*

What are, then, the mysterious *Descriptor* objects from the presented code snippet? They are simply subclasses of a root Abeans descriptor class, implementing a number of *tagging* interfaces. An application may examine descriptors with *instanceof* operator to determine if any particular tagging interface is implemented, and can react accordingly [5] – some tagging interfaces actually also declare methods, but not all of them. The following table briefly summarizes major descriptor tags:

***ConnectableRealization***. If a descriptor is tagged as being a connectable realization then the descriptor name can be used in a name resolution process. In other words, the name can be used to bind Abeans as a client to a remote object, obtaining a remote object reference. This describes CORBA or RMI (or any other) binding capability of a given name.

***LinkableRealization***. If a descriptor is tagged as being a linkable realization then the descriptor name cannot be used in a name resolution process, but nevertheless the named object will represent some resource allocation on a remote machine. For example, a monitor on a current of a power supply can be uniquely named by URI in Abeans and thus has an entry in the directory. However, you cannot bind to a monitor – you bind to a device and create a monitor on its property. Linkable objects, for example, represent monitors as transient resource allocations created in response to a request and lasting for as long as the request is active.

***ClassRepresentable***. If a descriptor is tagged as class representable, this means that there exists a level 1 Java Abeans class, which corresponds to a descriptor (level 2 entity in the directory). For example, a directory reports that a power supply "PSBEND_M.01" contains a "current" and "off". "current" is class representable, because Abeans have a modelling class DoubleProperty through which the current can be controlled. "off" is not class representable, because it is simply a method in PowerSupply class.

***DesignPatternRepresentable***. If a descriptor is tagged as design pattern representable, there is no level 1 Abeans class that models the descriptor. Remember, there are other possible level 1 representations in Java apart from Java class: a method, a Java Beans property or event source and so on. Although there is no class representation, a request may be issued to such entity, for example when "off" method of a power supply is invoked.

***NameContextRepresentable***. If a descriptor is tagged as a naming context representable entity, this means that the name is just a level in hierarchy and that it can be looked up as a directory and will contain a list of other names. In URI abeans-ACS://server.ijs.si/linac/PSBEND_M.01 "linac" is simply a naming context representable entity with no remote function. Naming contexts are automatically name context representable, but other entities may be as well.

***RequestTarget***. If a descriptor is tagged as request target, this means that Abeans Engine request can be directed to it. The request will be named by a full Abeans URI name and will carry parameters, name-value pairs, timeout data, error stack etc. A request is Abeans level 1 object, while Request Target that describes possible requests for a given target is a level 2 entity. It contains knowledge such as what types and number of parameters must be provided, what kind of name-value pairs can be put into the request, if the request will be timed, how many responses will be generated and what kinds of data they will carry and so on.

The described list of tagging interfaces is incomplete. Nevertheless I hope that it shows how a generic application can obtain enough information, by interrogating descriptors, to build requests on the fly and dispatch them through Abeans to the remote targets. Moreover, the enumerated set of these level 2 entities is generic enough to counter the second possible objection in Section 2.1 concerning the expressive power of meta-API.

## GENERIC SERVICE APPLICATIONS

After being given the machinery of meta-API, the design of a generic archive browser becomes a relatively straightforward task. An archive is conceptually a set of data points, indexed by name, time and (optionally) some arbitrary index (if we archive sequences, for example).

We simply mirror the archive entry names into the directory hierarchy itself. So, for instance, when a user does a lookup on “abeans-archive://TINE/HERA”, the directory will: 1) from schema and authority deduce that it has to contact the Abeans archive service running in TINE plug and forward the request there; 2) from “HERA” part the TINE Abeans archive will deduce that it has to contact the server corresponding to “HERA” and return a list of archived channels. Notice that although the name is *uniform*, each step in its resolution is actually processed by a different Java (or even remote!) object (namely the main directory, the TINE Abeans archive service and the remote TINE archive).

A generic GUI tree component that knows how to display JNDI trees will immediately know how to browse our archive. Once the leaf node is identified, the directory will return a specialized *ArchiveDescriptor*. This descriptor implements some tagging interfaces *and* contains methods that explain two remaining indexes, namely index by time and arbitrary other index set. In other words, the descriptor will tell us the time range, delta time step, type of the stored data, if it is single-value or array type, what are the additional indices and so on.

Because the structure of archive data is the same on different machines in the sense that it is usually indexed by similar procedures [6], we believe that an archive reader can be made into a generic application. *ArchiveReader* designed for DESY proves this concept and shows even the same application accessing, *at the same time and in the same way,* a TINE remote archive and Abeans Simulator archive.

## CONCLUSION

We believe that generic applications are possible, because functionality of the control systems is comparable. We are starting to prove this statement by actually producing first generic applications. Work remains to be done on several fronts: 1) improving GUI presentation for both request construction – given meta-data from the directory – and result display, especially by storing also the preferred way of visualizing data into the directory; 2) extending the role of the directory in data exchange even between local (Abeans Java objects), treating Abeans console logging service, for example, in the same way as the remote archive service; 3) fine-tuning of what data has to go into the descriptors and the ways of putting it there, so that it can be used effectively by applications; 4) putting additional subdirectories into Abeans directory if needed; for example, a type directory (as opposed to instance names directory), that would describe if there are type-instance relationships present in the underlying control system; or a virtual device directory; and 5) extending list capability of the directory to perform searches given a certain set of criteria.

## REFERENCES

[1] M. Pleško, G. Tkačik, “Where and What Exactly is »Knowledge« in Control Systems”, ICALEPCS 2003, Gyeongju, October 2003

[2] I. Verstovsek et al, “Abeans: Application Development Framework for Java”, ICALEPCS 2003, Gyeongju, October 2003

[3] URI RFC specification http://www.ietf.org/rfc/rfc2396.txt?number=2396

[4] This is only one approach to federation, because child archive servers are independent. There may be other federation scenarios, such as the presence of multiple remote logging services (and logs have to be forwarded to only one of them), or federation with replicated data. We are still pursuing research in those directions.

[5] The “tagging interface” approach is similarly used by normal Java serialization mechanism, where serializable classes implement *java.io.Serializable*, although that does not bind them to implement any actual function (it is just a design contract).

[6] Naturally, the implementations differ and it is up to the plug to translate from the archive directory form to a control-system specific form. However, this translation happens in a well-defined (in terms of input and output) piece of Java code in the plug and is therefore maintainable.