BEATING COMMERCIAL PRODUCTS: CONTROL SYSTEM OFFICE AND INTEGRATED DEVELOPMENT ENVIRONMENT ARE THE WAY TO GO

G. Tkačik, Cosylab, Ljubljana, Slovenia

Abstract

The success of office suites is based on the three fundamentals: consistent look and feel across applications, a common set of data manipulations (navigation, clipboards, undo/redo, copy and paste, find and replace etc) and the ability of different applications to process the same data (a table can be a part of a document in a word processor and a spreadsheet). This article discusses the same fundamentals in the context of control system software and shows that the "office paradigm" is relevant to it.

To support office-like functions, data beyond raw measured quantities is required: we call this data meta-data, and it encompasses – for example – machine-readable information about how different channels are logically organized into parallel hierarchies, how single values can be combined into group displays and how data items are interpreted differently according to the user interface context. We enumerate meta-data, describe how it is uniformly managed behind the scenes by Abeans, and how it enables generic applications such as table application, chart, archive or even IDE, to interoperate seamlessly as parts of an integrated suite.

INTRODUCTION

The purpose of this paper is to propose one possible architecture that realizes the *Three fundamentals* of an office suite: consistency of user experience, shared actions over data and uniform interpretation of data. The proposal is placed into the context of existing EPICS protocol for reliable and efficient data exchange [1], and Abeans for meta-data representation and management [2]. Steps in the direction towards integrating the two have already been made [3], along with the exploration of meta-data concepts in general [4]. Another contribution [5], presented at PCaPAC 2005, provides a follow-up on this work, detailing in particular how new Java technologies help in realizing design ideas presented here.

LESSONS LEARNED SO FAR

While we will not address the business case question of cost vs. benefit of a control office suite (COS), we nevertheless examine the issues related to Abeans and the Java platform to see how they support the COS idea. Although there have been successes with integration of Abeans and various other control systems [6], there are two major drawbacks in the current third release, namely:

- **Sheer size and complexity of the code.** Abeans have traditionally covered application framework and services in addition to data exchange; furthermore, for control system market, the design is probably over-generic. As a consequence, the desired functionality is either difficult to master (if it exists) or it is non-trivial to add for someone who is not familiar with the intricacies of the code.
- **Deployment is difficult**. The code is not modularly packed; the development of each application is still a compiled-code development and not script-driven / interpreted.

The reasons for these shortcomings are twofold. Firstly, the development of Java-related technologies has made much of Abeans redundant. Secondly, during the last years, scenarios for Abeans use considered by Cosylab have matured to the point where we can say what CS libraries can assume about underlying systems, and what has to remain flexible.

Evolution of Java platform and Abeans

Even if we postpone the discussion of Java 1.5 features, recent years have seen numerous developments relevant to office suite. An increasing number of application services are now provided by Java platform: logging is a standard package in *java.util.logging*, application name-value preferences (on system and user basis) are provided in *java.util.prefs*, exception chaining is defined through *initCause()* method of *Exception* class[*]. There are still areas where Abeans provide superior functionality, such as in resource loading (resource location makes transparent switching between local file system, remote URI, and other modes of storage easy) and property handling (Java System properties are complemented with command-line overrides, file configuration and so on) [6].

More importantly, a fair fraction of Abeans code (about a third) implements the *framework*, a system of child-parent relations coupled to lifecycle management, including the means of modifying and transversing it, and componentizing its parts. A number of similar schemes has emerged either as part of Java (JMX most recently), Web technologies (Jboss / Tomcat) or IDEs (Eclipse platform)[†]. Even though Cosylab can maintain the full framework codebase, the effort might be better spent on focusing on our specialty knowledge, i.e. control system data flow.

Finally, there has been a profound shift in how people view dynamic capabilities of Java, such as introspection and reflection[‡]. In Java 1.2 reference literature one can still read warnings on how to "use introspection sparingly, and only when there is no other way", citing the lack of type-safety at compile time as the reason for being

[*] Currently, the exception handling mechanism of Java is still much poorer than Abeans exception services, but the direction of development of Java platform is clear.

[†] We have examined Eclipse, Tomcat and JMX for use in the future.

[‡] After Java 1.2, features such as Dynamic Proxy and privileged access have considerably extended the reach of reflective programming.

cautious. Today, dynamic capabilities are widely used, most productively in scenarios where XML files are employed to define the structure of an application by enumerating modules and components that are then dynamically instantiated and installed by the framework. XML is thus used as a medium that makes loose biding and deployment of components possible, and through its schemes provides some replacement for tight type-checked coupling of a compiled code[§].

Data-flows and Abeans

By itself the conclusions of previous section are not striking and do not make up a COS: after all, an office is not a set of unrelated components living in a container. What is specific to a COS is the knowledge of how to describe, display and manage the data flow between the user and the controlled objects. To this end Abeans have promoted "URI name – Directory Descriptor – Request / Response" triad as a basic element of interaction with the control system:

1. Each data source can be a target for a request submitted to the underlying layer, capable of producing a response;
2. Request is targeted with a unique name, expressible in a standardized hierarchical form (URI);
3. An entity independent of the target, namely directory, can provide descriptors (also addressable by URI) for each target, containing information on what input data a request must contain and what kind of output the response can deliver.

In Abeans 3, these concepts are reflected in the code by interfaces in packages *abeans.models*, *abeans.engine* and *abeans.models.meta*.

Currently, all elements of the triad live within the same JVM. However, by using a framework technology that allows distributed objects, such as JMX mentioned in the previous section, the elements can become decoupled[**]; most notably, the directory with its descriptors can run in one location, independently of the actual Abeans engine. This reduces the overhead of instantiating and populating the directory, which can in addition store also (semi)persistent state about the objects (their availability, status and dependencies).

Soon, however, we realized that the directory can accommodate much more than solely data related to each target alone. For example, while the hierarchical names describe one possible object hierarchy, such as *ANKA/Booster/PowerSupply1*, we can imagine the same *PowerSupply1* being a member of *PowerSupplies* group parent; or alternatively, being part of a group *CompanyAPSs*, which are distinct from *CompanyBPSs* (and possibly controllable in a different way). To generalize, there might either be alternative hierarchies (in the strict sense of each object having exactly one parent), or alternative groupings (each object being tagged by arbitrary number of identifiers). Furthermore, how does one store the information that *PowerSupply1* and *VirtualDevice1* are actually the same underlying device? Or that *PowerSupply1* is entity of type *PowerSupply* and that all *PowerSupplies* have the same structure?

The answer is to store not only entity descriptors, but also relationship descriptors into the directory. We believe that if we attempt to make a control office suite, the relationship directory will turn out to be its cornerstone.

ABEANS DIRECTORY FOR OFFICE SUITE

Data-source entries

Currently, the nodes in the directory are organized in the following way:

```
abeans-EPICS
        domain1
                <device1, …>
        domain2
                <device2, device3, …>
abeans-types
        type1
                <type1-composition>
        type2
                <type2-composition>
abeans-relationships
        aliases
                device3 – isAliasOf – device2
        isInstanceOf
                device1 – isOfType – device1
        …
```

For each pluggable subsystem, there is a node *abeans-<plugname>* in the tree, which contains the basic, primary hierarchy that maps to URI names. For example, *abeans-EPICS://domain1/device1/channel1* would refer to a descriptor of *channel1* stored 3 levels deep under *abeans-EPICS* node. A direct mapping of primary tree to URI names makes it possible to look up or list all physical data sources in the system in constant time.

If the system supports the distinction between a type and an instance, then all instances listed under primary hierarchy *abeans-EPICS* that are of the same type (such as a *PowerSupply*) share the same descriptor object that is listed in *abeans-types://PowerSupply*. While each single lookup for any instance of power supply in *abeans-EPICS* will still return a descriptor, the directory now contains additional knowledge for applications that are aware of the notion of a "type" (for instance, the device table application), returning the type info and enabling queries against all instances matching a given type.

[§] For example, consider XML deployment descriptors of Tomcat WAR-packed applications.

[**] Note that in architectures such as JMX it is actually indistinguishable to the user whether the directory runs on the same or different machine, and the choice could be affected through configuration without code modification. Even if the architecture does not allow for this, Abeans API itself can be designed always to give the user the feeling of co-location of the engine and directory.

Finally, *relationships* entry contains (binary) relationship instances, organized by the relationship type (such as *aliases*, *isInstanceOf*, *isMemberOf,* etc). A large number of functions, previously handled separately or not provided, can now be uniformly approached with this mechanism. For example, it is now possible to query all objects of type *PowerSupply*, all objects that access the same underlying channel; furthermore, it is possible to declare completely new tree nodes, such as *abeans-virtual://VirtualGroup1* and, through *isMemberOf* relationship, declare that devices *device1, device2, device3* belong to this group.

When a relationship is implemented in Abeans libraries, there is a piece of code that tells the framework how to process the actual relationship. For instance, if *device2* aliases *device1* which is of type *PowerSupply*, it makes sense to treat *device2* as if it were also a *PowerSupply*. Note that for the user, programmatically, the information is accessible through a uniform Java JNDI interface, which is part of the platform.

Data-sink entries

The relatively small investment in the directory implementation (compared to the development of the application framework) gave us great flexibility and transparency in information organization. Although it is tightly coupled to other Abeans R3 classes in the current version, we believe that there is no inherent problem in creating a stand-alone directory component. However, the existing incarnation only describes data sources and their organization, while office is concerned with data sinks, i.e. ways of presenting and manipulating data once it has been acquired from the source.

Data-sinks are conceptually adapters to components that perform display, printing, formatted output (saving, archiving) or data transformations. It is relatively trivial to design a one-to-one source-sink mapping, in which e.g. a gauge GUI component will display the current in a power supply; mapping many sources-to-one sink requires more thought. In addition, it necessitates some parameterization of the content of a data item being transferred: for example, a data-source descriptor may state that *abeans-TINE://Server1/source1* returns an array of doubles, but there is no information that explains whether this array is a profile (such as a power supply current readout over 100 power supplies), a waveform (current values that depend on time), or simply current values indexed by a certain known or arbitrary index.

Currently, there is no implementation whatsoever of data-sink entries, although we are working on their design. We think that the field of control systems is constrained to such a degree that the number of types of content being transferred is relatively limited: there are single-values with all their access modes (get, set, monitor in synchronous and asynchronous versions and with different monitor triggers), tuples of values (usually treating pairs separately *might* still be relevant, but not higher-order tuples, except in specialized cases where whole structures are transferred, in which case this can be viewed as a compound data source or a map of key-value pairs), and arrays of values (where the indexing axis is either time, device ID; or alternatively the result is a tabulated function). In the following sections we will show how data-source, data-sink and relationship descriptors tie together office components.

OFFICE COMPONENTS

One of the successes of CosyBeans visualization libraries was the distinction between the visual container (such as a *CosyPanel*) and a plug-in (logging, exception, navigator plug-in windows etc). Just as a directory serves to organize information flow, the visualization of a COS should be based on an installable GUI framework (either Cosy Beans or 3rd party, such as Eclipse). Graphical components then become plug-ins in the visual hierarchy, but also data sink entries in the directory. In addition, operations on the data get encapsulated into *Actions*, for which Java already defines interfaces in Swing libraries: *ExportToTextFile* action, for instance, could take a data source and pop up a window asking the user for times or triggers when the value of data source should be exported, after which the action is enqueued and starts executing. *ExportToTextFile* is registered as a data-sink in the directory, and can be listed in the pop-up menu when the user right-clicks on the data-source entry in the navigator tree display of the devices[††].

Although the scenario seems contrived and over-simplified, note that there is nothing specific to saving a selected data source to a file: what is important is the fact that the action (the sink) understands the data format that is being produced by the source, and that in order to make this determination enough data is stored in the directory (in a machine parseable format) so that the actual source and sink are not instantiated *before* the compatibility is established. In the same way one could imagine selecting nodes in the tree and displaying them in a chart by selecting “View data in chart” from the pop-up menu. Note that visual application construction from actions and components is not something that Cosylab is promoting as our original idea, but is rather a disciplined exercise in well-known GUI design patterns that should be undertaken regardless of whether a suite is being developed or not.

PUTTING PIECES OF THE PUZZLE TOGETHER

A company of our size could not undertake the effort of implementing COS if the platform itself did not progress through the years. Today, however, this task is doable. If, for example, JMX is adopted as framework platform, one can imagine the following sequence of steps leading the control office suite:

[††] Note that such action would not appear for navigator entries that are not data sources, for instance hierarchical domain names in our example of *abeans-EPICS://domain1*. Context sensitivity is possible because sinks declare in their descriptor what kinds of targets they can consume.

1. Define data-source and data-sink metadata (i.e. what kind of entries the directory contains), and implement the directory as a JNDI accessible JMX component. At this point any client running JMX could query the directory. In particular, there can be clients that actually fill the directory, either by querying XML input files, CORBA Interface Repository or some other source. We have implemented data-source part in R3 and have drafted preliminary designs for data-sinks. We have experience with JMX.
2. Make a JMX entity that encapsulates current Abeans engine. This entity processes Abeans Request objects and turns them into EPICS PV read/write operations, for example. Our new analyses show that the engine and plug of Abeans Release 3 can be fused into a single mechanism[‡‡]. In addition, in advancing past Java 1.2, the performance aspects of coding have changed considerably, with small object instantiation optimized but threading and GUI still consuming a lot of CPU. In effect, Channel Access for Java could process directly the "generalized request and response" form, making bridge from Abeans to EPICS much thinner. Most of these components are operational but would have to be refactored.
3. Adopt or make a GUI skeleton framework, supporting installable plug-in. CosyBeans already exist, but would need to be refactored. Other platforms can be considered.
4. Create GUI components and actions for the office. Some underlying graphical components are already in use (charts, navigator, gauges, tables), but would have to be refactored to use the data-sink specifications.
5. Define the deployment, installation, negotiation and configuration protocols for the sources and sinks.

The last point requires further work, but also offers a great promise that could address the second big Abeans R3 drawback (difficult deployment) and reap new Java technologies. Imagine distributing the office runtime (new Abeans core) only as the basic set of JMX components (engine that talks to the underlying system, directory access, GUI skeleton, bootstrap components) and JVM runtime (with new application services that it provides)[§§]. The GUI skeleton wakes up and contacts the directory, which now contains also a list of applications themselves. When an application is selected, its graphical components get downloaded directly from directory if they are not already present on the client machine (trivially achievable with Java byte-code). The GUI components install themselves into the skeleton (a series of plug-in, action and control screen instantiations), and as they integrate themselves into AWT hierarchy, they register in parallel as data-sinks. The final part of the application specification in the directory is a binding scheme of data-sources to data-sinks, both of which have descriptors in the directory themselves making it conceivable that there is an algorithm that instantiates the sources and starts feeding data into the sinks. Because applications of this design consist of an enumeration of data-sinks and the bindings to the data-sources, they as well can be stored in the directory as descriptors, which completes the scheme.

[‡‡] Because of the restricted scope of cases where we wish to deploy Abeans.

[§§] The JAR size would decrease by a large margin, because the complete JMX framework is part of the core Java platform as of Java 1.5.

CONCLUSIONS

Although by no means simple, I believe that the effort to implement such an architecture is smaller than that of creating Abeans Release 3. It is hard to stress enough how the Java platform has changed and the fact that this (and the focus that office use-case would give to Abeans) makes it possible to reduce the codebase of current release by at least a factor of 2 if not more. Migration to an industry-standard platform such as JMX and deployment akin to Tomcat or Jboss also makes entry for new developers much easier, because there is no longer a monolithic block of code to master, but rather a loosely (but precisely) tied selection of independently installable components. All patterns and supporting classes with generality wider than that of Abeans can be factored into separate APIs, either because they could be usable for other Cosylab projects, or to be replaced by open-source equivalents (such as Apache Commons), where appropriate.

Furthermore, the strategy outlined here allows for incremental progress towards a full-fledged suite, where the only problem to be solved in one go is the specification of descriptors and negotiation, while the implementation can proceed in small pieces and independently. Some existing specifications, such as the [7] or the Infobus / JAF can serve as design guidelines.

Cosylab has the accumulated experience of having implemented various projects on a wide range of control system architectures, experience of trying to introduce meta-data to facilitate generality and maintenance, as well as the benefit of having tracked carefully new technological developments of the Java platform. In my rough estimation a man-year of work investment should result in a working prototype of the control system office.

REFERENCES

[1] http://caj.cosylab.ocm
[2] M. Pleško et al, "Where and What Exactly is "Knowledge" in Control Systems", ICALEPCS 2003, Gyeongju, Korea, October 2003
[3] A. Košmrlj, "New Features for New Applications with Abeans 3.1", PCaPAC 2005, Sokendai, Japan, March 2005
[4] G. Tkačik et al, "A Reflection on Introspection", ICALEPCS 2003, Gyeongju, Korea, October 2003
[5] G. Tkačik, "Generic Types in Java: Abeans<T> Specifically for You, Mr T?", PCaPAC 2005, Sokendai, Japan, March 2005
[6] I. Verstovšek et al, "Abeans: Application Development Framework for Java", ICALEPCS 2003, Gyeonju, Korea, October 2003
[7] XML Datatypes http://www.w3.org/TR/xmlschema-2/