JAVA BEANS OF ACCELERATOR DEVICES FOR RAPID APPLICATION DEVELOPMENT

Mark Plesko[+], Gasper Tkacik[+], Miroslaw Dach[*], Steve Hunt[*]

Abstract

A Java bean is a reusable component that can be manipulated in a visual builder environment, similar to Visual Basic: beans can be graphically arranged and connections between them established. Such environments enable the programmer to build an application without typing a single line of code. Many visual beans exist, such as buttons, gauges, charts etc... However, beans can be also invisible, having pure functionality without graphical representation. We have written a library of invisible Java beans, called Abeans for accelerator beans, that implement controlled objects of an accelerator - devices. The concept is based on ACOP, but goes further such that for each device type there is one corresponding device bean. A device bean encapsulates all remote calls from the client to a device server of the process control layer, e.g. get/set, on/off etc. Thus the network is invisible to the user of device beans. Tasks of a device bean include opening the connection and performing the function calls on remote objects; report and manage all errors/exceptions/timeouts, providing handles for asynchronous messages and the like. Abeans currently support CORBA communication through the ACI interface [1] and CDEV. All the applications for the control system of the ANKA lightsource have been built using Abeans. The SLS plans to use Abeans to connect the same applications to CDEV.
.

1 INTRODUCTION

Abeans were developed to fulfil the following requirements:

- to enable final users - application developers to develop applications for accelerator control easily (advanced programming skills are not required), safely (by delegating as much work as possible to automatic code generators, which are thoroughly tested off-the-shelf products) and quickly (by using complex commercially available components)
- to present a clean, object oriented view of the accelerator to the final user
- to hide implementation details of the lower level subsystem (e.g. network communication)
- to be platform independent (with Java) and pluggable (Abeans can be used with different communication systems, like CORBA, CDEV, RPC)

2 ARCHITECTURE

The Abeans were influenced mostly by the following major design decisions:

2.1 Separation of data and its visual representation

Abeans are beans used to access remote accelerator devices. They have no visual representation. They do, however, expose a number of properties and fire certain events. Visual beans (called awidgets) interact with invisible Abeans to display their data. Visual beans are, for example, standard swing or awt components, commercially available chart and table beans and our own collection of gauges, setters and led panels. In this way we retain the freedom to change the visualisation without changing the Abeans, we are able to present more views of the same data item (like displaying the value of the current in a textfield, chart or on a gauge), and our own awidgets are independent of the control system. The cost of such freedom is the added overhead of the communication between Abeans and awidgets through Java events.

We were faced with the problem of how to efficiently connect awidgets and Abeans in a visual builder environment. While simple in theory, in practice the approach would include making many visual connections. Imagine connecting a gauge bean with Abean representing current in a powersupply: first you must make a connection that synchronizes current value with displayed value on gauge, then you need to set units, name "current", minimum and maximum value etc. on a gauge, obtaining the values from current; this would involve making many connections. The preferred approach is to implement adapters - an adapter extends the awidget and 'adapts' it, so that it knows how to handle an entity like current directly. Then, visual building involves creating a single connection from current of the powersupply to the new, adapted gauge. The latter knows how to handle current internally. In this way the visual programming remains simple without sacrificing the generic nature of the awidgets.

[+] JSI, Ljubljana, Slovenia (e-mail to gasper.tkacik@guest.arnes.si)
[*] PSI, Villigen, Switzerland

2.2 Separation of Abeans implementation and pluggable layer

In cooperation with colleagues at SLS we developed pluggable architecture for the Abeans. Regardless of whether the Abeans are used to access devices through CORBA subsystem or, for instance, CDEV subsystem, their interfaces and implementation does not change. For each subsystem, the so-called proxy interfaces have been implemented along with some specific classes providing information about that subsystem. These classes all reside in a separate java package, called pluggable package. Then, when an application is completely built, the user can decide for each Abean which subsystem it will use to connect to remote accelerator device. At runtime, the whole pluggable package is loaded for the appropriate subsystem allowing Abeans to utilize it. Adding a new subsystem thus requires writing the pluggable package, but does not involve any change at all to other parts of the Abeans code.

3 IMPLEMENTATION

Abean architecture closely matches the Accelerator Control Interface presented by Mark Plesko [2]. For each physical device in the system there is one Abean. As far as the communication with the remote device (the server side) is concerned, an Abean is a collection of methods and properties.

3.1 Methods

A method called on the Abean is called on the remote device. This passthrough is useful, because it brings all methods to the common denominator - they all look like normal java function calls. In reality, the following steps are performed:

- regardless of whether remote method is asynchronous (meaning that it returns immediately and reports its remote completion later through a special callback object - examples are methods which take a long time to complete, like telling the powersupply to ramp) or synchronous (blocks until the remote request is completely carried out) they all appear the same to the user. If it is asynchronous, the Abeans take care of creating callback objects, passing them to the server and receiving network notifications. User can specify whether the Abean should wait for each call to execute completely and to have the fact confirmed by callback before proceeding to the next remote call.
- Abeans check for network errors / timeouts, handle them and report them
- Abeans check for device errors and report them
- Abeans can log remote method calls (a replay function is under development)

3.2 Properties

A property is a bean itself, therefore it is a bean contained in the bean for a device. There is a small fixed number of property types which represent different physical quantities or states of a physical device. For example, we can have DoubleProperty (representing double value, i.e. the current of the powersupply) or PatternProperty (representing bits, describes e.g. the state of a powersupply), in read-only or read-write flavour (current on powersupply can be set / read, but readback on the powersupply can only be read). Apart from the primitive value they contain, these beans provide a wealth of additional information about this value, putting it into a physical context. A RWDoubleProperty can thus be queried for minimum and maximum value, units, name, description, resolution and the like. There is also a number of ways of setting or getting this value (synchronous, asynchronous etc.) Why is it useful to have property beans inside beans representing devices:

- they conceal the data source: physical data comes from the actual device (value of the current), while accompanying information (minimums, maximums, units, names) come from the database. This distinction is completely invisible to the user.
- they manage monitors: a current in a powersupply can be monitored, meaning that the control system periodically sends new values to the client (say every second). The management of the monitor is hidden from the user by the property bean representing the current. Whenever any other bean is interested in receiving current value updates (for example a trend chart that plots current against time) it registers as standard Java beans listener to the bean that represents the current. That in turn automatically creates the monitor and sends monitor callbacks as events to all listeners (the chart). Such dynamical construction and destruction of monitors conserves network bandwidth and server CPU time. The process is further optimized by the events being dispatched to graphical components - awidgets - only when the value changes (because they need to refresh their display) and not periodically, when the control system sends updates.
- they handle alarms and monitoring timeouts through special events

3.3 Abean specific services

In addition to the set of remote device methods and properties, which also communicate with the remote device server, Abeans provide a number of other services to the user which are local to the Abean system. These include:

- configuration management, which takes care of configuration loading (implemented as java resources which are accessible from applets or applications), configuration front end and an infrastructure allowing the custom pluggable subsystems to extend the default list of settings that

must be specified with their own. For instance a CORBA pluggable package needs completely different types of settings than CDEV package; when user selects one or both for his/her application, pluggables request their own specific settings to be queried from the user. By default all Abeans in an application are initialized with settings from the configuration for that application. Thus the user does not need to specify the complete initial state of the Abean programmatically, but just overrides the defaults; furthermore, this approach significantly decreases the amount of information hardcoded into the application.

- timing management: since virtually no assumptions can be made about automatically generated code in a visual builder environment, a way has to be found that uniquely determines the execution sequence of the client process. For instance, some code builders might create all Abeans at the application initialization phase, while the others wait until they the Abeans are needed. The potential problems such differences could cause would be hard to detect, impacting mostly the Abean connection process. By means of grouping Abeans into the so-called families the user can determine time frames during which the Abeans can connect to the remote servers. The default behaviour remains simple (connect-when-ready) but should the need arise, the possibility for finer control over connection exists.
- differentiation between manual use and use in visual builders: while Abeans can be used visually they are naturally also normal java classes that can be used in hand-written code. However the requirements that the user places on the Abeans in both modes of usage differ slightly. In manual mode, for example, handling asynchronous method calls is difficult. Imagine sending asynchronously an 'on' commend and a 'set' command to a powersupply: the first has to complete before the second is sent. Validating this sequence at every step manually requires a lot of programming. Therefore Abeans internally synchronize the calls. In visual mode, on the other hand, asynchronous nature is desired, because it does not block the user interface; special cases, where the order of actions is important, can be handled by careful construction of the user interface (disabling the button for action 2 until action 1 has completed). There are also some other minor differences in this regard.

It is important to note that the large majority of these features are implemented once in a superclass of all device Abeans. Moreover, the number of properties that comprise the Abeans for devices is small and they are already coded. Consequently writing an Abean for a new device involves very little additional work - a simple code generator from IDL to Abeans could easily be written.

4 PLUGGABLE ARCHITECTURE

I have already given a short conceptual overview of the pluggable system in *2.2*. Here I will discuss some of the issues more in depth. The whole Abeans system can be imagined as a two-layered architecture: the upper layer, which the user and the visual builder see that is independent of the communication protocol and passes requests on to the second, pluggable layer. The latter actually executes all communication-system-dependent function calls. Since the upper level is fixed for a given device regardless of the communication protocol and the device model which exist on remote server, it is clear that some 'translation' must occur on this pluggable layer. More specifically, since the Abeans represent the whole accelerator as a collection of devices each with its methods and properties, this object oriented view must be constructed on the pluggable level if it does not yet exist on the remote server (for example, some control systems have servers that expose the control system as a connection of channels, each representing a physical quantity or the device state; there is no concept of device that groups together a number of channels. In such a case, a device would be constructed out of these channels on the pluggable level). If Abeans run on two systems and a device that exists on both does not have the same interface, this poses no problem, since Abeans for new devices are created easily. A more pressing problem presents itself if there is no corresponding data type in the Abeans level that exists on the control system for which the pluggable layer should be created. Up to now no need arose for property types other than double and pattern/integer, but we are addressing also this issue.

Another use of the pluggable layer is the implementation of a simulator. Instead of connecting to a real and existing remote system, Abeans connect to pluggable layer which simulates remote devices. Every panel or application written for the simulator can be used for the real control system simply by telling the Abeans to load, for instance, CORBA pluggable package instead of simulator package.

5 CONCLUSION

The Abeans strive to present the final user with a view of the accelerator as a collection of accelerator beans, thereby reducing the problem of writing applications for the control system to the problem of familiarizing oneself with the ways to use java beans and development tools. To this end as much complexity as possible is hidden, but still accessible on demand. All interfaces that Abeans expose are type-safe (with no 'single function taking control string' methods) and enable one to construct a simple yet powerful graphic application within minutes. The pluggable layer offers the possibility of accessing different remote systems and by its nature of adapting the lower-level remote system interface to the Abean interface enforces uniform and consistent behaviour across a range of possible remote systems.

6 REFERENCES

[1] M. Plesko et al, A Control System Based on Web, Java, CORBA and Fieldbus Technologies, PCaPAC99 workshop, Tsukuba, January 1999.

[2] M. Plesko, Implementing Distributed Controlled Objects with CORBA, PCaPAC99 workshop, Tsukuba, January 1999.

[3] Sun's JavaBeans site: java.sun.com/beans