

Atacama Large Millimeter

KGB-PAP-01/01

Revision: 1.1

2001-09-14

*Abeans White
Paper*

M.Plesko et al.

Abeans White Paper

Building Java Clients

M. Plesko

J. Stefan Institute

G. Tkacik

J. Stefan Institute

I. Verstovsek

J. Stefan Institute

Keywords: Java, CORBA, GUI, Client, AbeansKGB-PAP-01/01

Author Signature:

Date:

Approved by:

Signature:

Institute:

Date:

Released by:

Signature:

Institute:

Date:

Change Record

[illegible]

Table of Content

1 Introduction.....	4
1.1 Scope:.....	4
1.2 Executive summary.....	4
2 Efficiently creating GUI Applications with Abeans.....	6
3 Analysis of Abeans Applications.....	10
3.1 Panels.....	10
3.2 Group Applications.....	10
3.3 Complex Applications.....	10
3.4 General Remarks.....	11
4 Framework for Application Development.....	12
4.1 Architecture.....	13
4.1.1 Separation of Data and Visual Representation.....	13
4.1.2 Separation of Abeans Implementation and Pluggable Layer.....	14
4.2 Implementation.....	14
4.2.1 Methods.....	14
4.2.2 Properties.....	15
4.2.3 Abean specific services.....	15
4.3 Pluggable Layer.....	17

1 Introduction

1.1 Scope:

When developing a control system application one should follow the vision: **To produce a user friendly client, where “user” stands for a physicist that operates the system** and not a computer expert. Therefore, the main characteristics must be that the client is easy to maintain and that it allows non-experts to easily build powerful graphical user interfaces. This document describes our vision how to

- write efficient ACS clients
- create ACS graphical user interfaces in a commercial visual tool with no or few lines of written code
- use strongly typed object oriented components to reduce developing and testing time and increase maintainability

and the concepts how we have achieved this vision. Everything described in this document has been already developed and can be used for ACS 1.0.

1.2 Executive summary

When writing a client application the programmer is most often confronted with numerous problems from the areas of error handling, timeout handling, logging, communication system details, resource initialization and destruction and the like. In terms of lines of code it can easily happen that such functionality easily compares with the core application functionality. The same reasoning applies to GUI programming.

To address these problems in a consistent way, a clean and consistent design of the whole ACS, as it is provided by BACI, is of paramount importance. But this is not enough. Programmer should not have to solve the same problems over and over. On the client level in particular, we need solutions to these problems in a solid framework of components that can be directly assembled into powerful applications. We have developed such a framework with the programming language Java and using its component model, called Java Beans: The ACS distributed objects are wrapped into specially developed Java Beans called Abeans, which provide also a rich set of tools that clients always need. This allows applications to be written easily, even by nonprogrammers.

A Java bean is a reusable component that can be manipulated in a visual builder environment, similar to GUI-builders or Visual Basic: beans can be graphically arranged and connections between them established. Such environments, commercially available at a reasonable price, enable the programmer to build an application without typing a single line of code. Many GUI beans exist, such as labels, buttons, gauges, charts, etc. We use commercial and also homemade beans, where the commercial ones don't provide

the necessary functionality. However, beans can be also “invisible”, having pure functionality without graphical representation. We have written a library of invisible Java beans, called Abeans for ACS beans, that wrap distributed objects of ALMA - devices. For each device type there is one corresponding device bean. A code generator automatically creates beans from the interface definition language (IDL) description of ALMA devices. A device bean encapsulates all remote calls from the client to a device server of the ACS, 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.

The Abeans strive to present the final user with a view of ALMA as a collection of device 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.

In this way, the learning curve for a new programmer is not very steep and demanding. For instance, ALMA control system uses CORBA for communication, but programmers writing the visual components or "views" are completely unaware of CORBA, since the invisible Abeans layer hides it. All interactions with the data sources that the view needs to display its data are performed through simple and well-defined Java Beans interfaces.

Therefore the work done in building a control panel for a device consists mostly of connecting the appropriate device Bean and visual beans in a visual builder. The developing time is low, of the order of hours for a panel, or one or two days for a full-fledged application that instantiates and interconnect many device Beans - even of different devices.

We have chosen Java because it is a modern object oriented programming language, it has well defined data types and API (Application Programming Interface), it allows easy use of graphic widgets, threads and other system tools without having to know the specifics of a given platform. Java is also an interpreted language, so it is a little slower than compiled languages like C++, but we found out that by using JIT (Just-In-Time) compilers it is fast enough for our needs. The design patterns of Abeans are sufficiently mature so that they can be implemented also in another programming language. However, the advantage of using visual builders as for Java Beans is lost in that case.

2 Efficiently creating GUI Applications with Abeans

An example of a commercial visual builder is Visual Age for Java (VAJ), which is a complete integrated development environment. (figure 1)

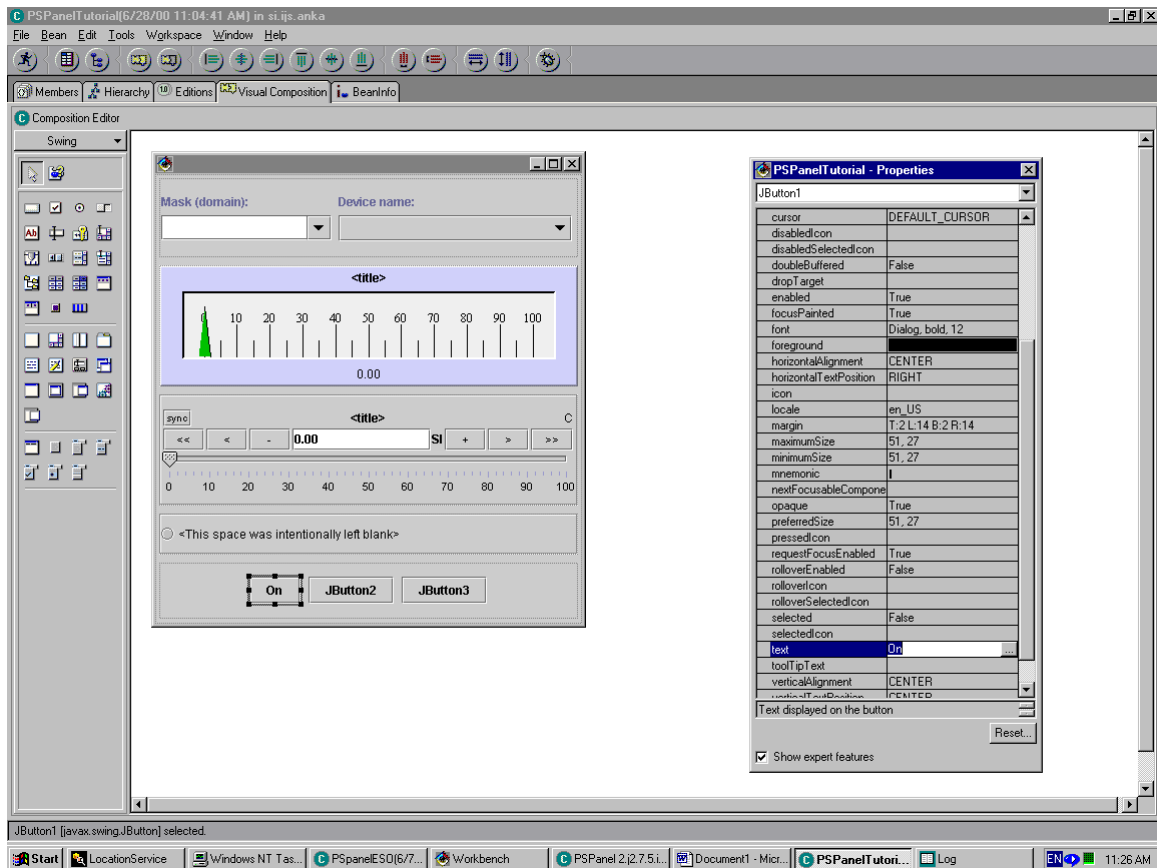


Figure 1: A screenshot of Visual Age for Java. To the right is the property list, where each property of the selected bean can be edited. The pane on the left contains icons for Java Swing Beans. The panel between the property list and the icon pane is a control panel under development.

Our goal was to make the applications be built with ease, without a detailed knowledge of the ACS design. To achieve platform independence, the clients were all written in Java. In order to hide the network from the client programmer, the so-called Abeans were developed. They are a complete framework for the development of applications and also for handling device errors and communication problems that occur. Basically they are Java Bean objects that map to CORBA distributed objects (encapsulate all remote calls from the client to a device server of the ACS layer), following the same rules as defined in BACI. Besides this, they provide the following functionality:

- open the connection and perform the function calls on remote objects
- report and manage all errors / exceptions / timeouts arising from network communication

- provide handles for asynchronous messages, queue / demultiplex responses

To represent the values of the control variables (pointing vector, status of the Mount, etc.), several GUI components were developed: gauger, slider, trend and ledler. They were specifically designed to represent the properties defined in BACI and are therefore tailored to be used with Abeans. It is even possible to create simple applications in a visual builder environment without typing a single line of code: all that needs to be done is to connect a property (e.g. current) from an Abean that represents a certain device (e.g. a power supply) to a corresponding GUI component (e.g. gauger). Another component is the selector, which enables the user to search for all available devices of a given type dynamically at run time and chose one or a group of them. When the choice is made, the Abeans automatically take care of the initialization process and the gauger is immediately showing the correct value. An example of such an application is shown in figure 2.

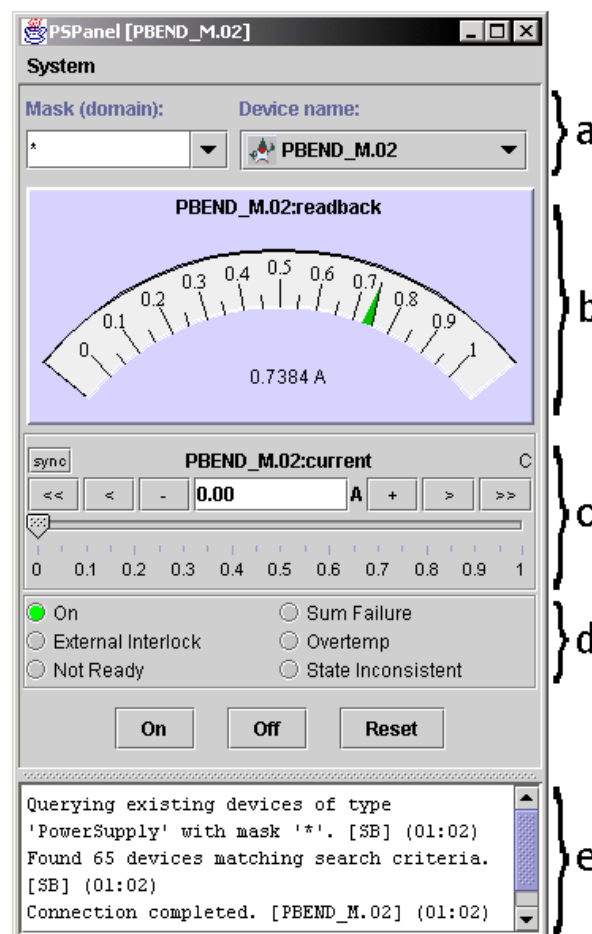


Figure2: Power Supply Panel. This panel was created from Abeans and GUI components in Java visual builder environment without typing a single line of code. The following “ACS standard” GUI components can be seen: a) selector, b) gauger, c) slider, d) ledler, e) ATextPane - a pane with messages for the user.

Great care has been taken in the design of the GUI components. They don’t just display a single value, but provide visualization for all characteristics of a property. Each gauge or

slider can spawn a trend chart. Limits and display precision is read from the property by default but can be set by the user at run-time. Among others, keyboard shortcuts, logarithmic scale, different modes of value get/set, different refresh rates, tool-tip text and alarms are all supported. The trend chart has variable history length, supports zoom and pan, saves data in TAB-delimited format and can convert to a histogram chart.

By using Visual Age for Java, it is possible to create the whole application completely visually, without writing a single line of code, similar to as shown in figure 3. Figure 3 actually also demonstrates the ease of adding event-handling code: the box above the panel refers to a small method containing 3 lines of code. This method is invoked whenever the “bind completed” event is triggered by the power supply Bean, which is nicely shown graphically. The skeleton of the method is generated by the visual environment from a menu selection.

This example really proves the claim in section 1. But there are more complex examples that can be written with Abeans. Even the most complex applications can be built without any knowledge of the design below the level of Abeans. All that is necessary is the knowledge of Java and the organization of Abeans.

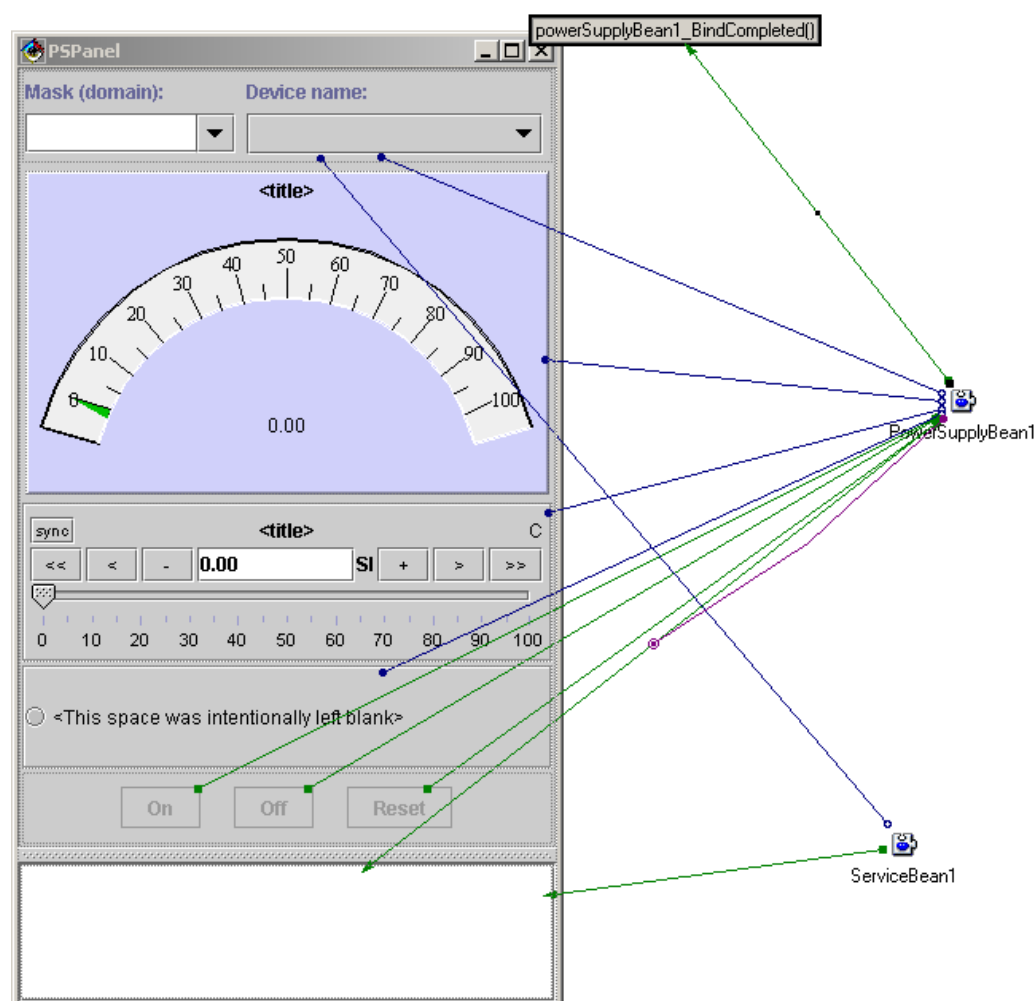


Figure 3: The complete visual composition of the panel shown in figure 2.

Another type of Abeans applications are applications that manage whole sets of devices at the same time, like generic device tables, alarm table, snapshot, etc. We have written several generic applications that can accept any device Bean, even if its type was not known at the time the applications have been written:

- AlarmTable: registers as alarm-listener to all properties. Displays actual alarms; filters and mask alarms; (de)selects device/properties, etc.
- ObjectTable: displays all devices of a given class (type) in a table, one device per row. The columns are values of properties. RWDdouble properties have three rows, for display and setting the values. ObjectTable also features trend-chart (values as function of time) and profile-chart (values as functions of position) of any combination of properties, multiple-device commands, save/load snapshot files, etc.
- Snapshot: displays all active devices in an Windows Explorer like tree, takes the actual set-values of properties for selected devices (by domain, type, etc.) and writes to a text file or into an SQL database. The text file is in TAB-delimited format for easy import into spreadsheets. Existing snapshot files can be examined and selectively (by device type and domain) loaded.

3 Analysis of Abeans Applications

Applications used in control systems can be roughly divided into three classes, namely *panels*, *group applications* and *complex applications*.

3.1 Panels

Panels use a set of widgets (gauges, sliders, charts) to present the state of a device and allow the operator to modify the state. Development of panels is rapid – for a complicated panel it may take some days – and is facilitated by a GUI builder in a RAD (Rapid Application Development) environment. No code is written by hand. The application programmer delegates the task of connection, device query, error reporting etc. to the Abeans libraries.

3.2 Group Applications

Group applications are applications that control a large amount of devices of the same type. Examples used in ALMA are device tables, snapshot and alarm table. Because these applications include a certain amount of logic so specific to the application that it cannot be a part of the Abeans framework, their development cycle is radically different. Only the GUI is designed in the visual builder, while the access to the Abeans and the connection of the invisible beans to the GUI is programmed by hand. Performance often takes priority over the ease of programming and in current Abeans release the application programmer must take care of some performance issues. Nevertheless, a lot of tasks are still done by the Abeans framework, since group applications rely on the operator to respond to control system conditions (the response is not automatic). In other words, the main function of a group application is to aggregate the data and display it, while allowing only a simple and manual way to change the state of the machine.

3.3 Complex Applications

As we do not know the applications necessary for ALMA, we discuss an application we have written for the ANKA particle accelerator, called the Databush. It is a perfect example of an application that qualifies as complex in the ANKA control system: it is a library of machine physics algorithms, for instance orbit correction. Databush connects to both power supplies and beam position monitors. This application requires special access to the Abeans libraries: it is not enough that errors are reported to the user automatically by the Abeans framework, they must also be processed by Databush. Databush also needs fine-grained control of the new value delivery (packed monitor delivers data for all power supplies in one batch), which must be thread-safe. There are also special requirements for synchronization (asynchronous commands must complete with OK status before new commands are sent) and management of a large number of device beans.

3.4 General Remarks

Comparing the requirements of panels and complex applications that use the same Abeans framework, the framework designer is faced with the dilemma of building either a thin, streamlined, “lean-and-mean” version of Abeans or a “has-it-all”, full-featured but heavy library, which offers the access to the devices in any imaginable way. As explained in the following sections, the Abeans library attempts to solve this problem without sacrificing the simplicity of RAD development.

Further analysis has shown that application code that calls the Abeans libraries is small in quantity and easy to write. Most of the time spent while developing a group application goes to tasks such as programming GUI responses and programming data flow (this involves creating Swing models and programming data structures that hold the data obtained from the Abeans framework). A new release of Abeans is being designed to reduce the amount of data copying between different models by providing data to the application in the form that is acceptable without further interfacing. Because data copying is one of the most performance degrading functions in Java, this should result in notable performance increase.

4 Framework for Application Development

A divide-and-conquer solution that ACS client software embraces is the use of the following approaches:

- Control system is comprised of server layers and client layers; the latter consist of framework (invisible) Abeans layer (which in turn consists of server-dependent or pluggable layer and independent layer), and application (visible) layer
- There can be many types of controlled devices and therefore many different interfaces (one device - one class approach), however, each device is constructed of fundamental building blocks - Properties, and there is only a small number of these (RWDoubleProperty represents a physical quantity of double type that can be either changed or retrieved, for instance a current in a power supply; ROPatternProperty, represents a retrievable read-only bit-pattern)
- The core visual components (such as gauges, status displays, trend charts) should be completely independent of the actual control system, so that they can be used by others
- Special adapters have intimate knowledge about the Property and about the visual component, allowing a generic visual component to be "adapted" to the Property. This means that the Property will be an invisible framework class and the generic visual component will, with the help of an adapter, become its view (to use the Model-Controller-View terminology).

All those concepts are implemented in Java with components that adhere to the Java Bean specification. A Java Bean is a component that can be manipulated in a visual builder environment: Beans can be graphically arranged and connections between them established. The latter include, for example, event-to-method connections, where the event in one Bean triggers the method in the other; property-to-method connections, where a change in property triggers the method, property-to-property connections and so on. Such environments enable the programmer to build an application without typing a single line of code.

Any ALMA client application is composed of two types of Beans:

- visual Beans (GUI objects, like windows, buttons, gauges, charts and the like) either commercial products or in-house products of equivalent quality;
- device Beans developed in-house for control systems:
A device Bean encapsulates all remote calls from the client to a device server. Thus the network is invisible to the user of device Beans. Each device (which is presented in the ACS as a distributed object - see BACI) has a

corresponding device Bean. Tasks of a device Bean include opening the connection and performing the function calls on remote objects; report and manage all errors/exceptions/timeouts arising from network communication, provide handles for asynchronous messages, etc.

4.1 Architecture

Abeans were developed to fulfill the following requirements:

- To enable final users - application developers to develop applications for 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 controlled devices 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, RPC, sockets)

The Abeans were influenced mostly by the following major design decisions as discussed in the following subsections.

4.1.1 Separation of Data and Visual Representation

Abeans are beans used to access remote 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 visualization 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 power supply: 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 power supply 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.

4.1.2 Separation of Abeans Implementation and Pluggable Layer

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.

Regardless of whether the Abeans are used to access devices through CORBA subsystem or, for instance, EPICS[†] 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 the remote 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.

4.2 Implementation

Abean architecture closely matches the Basic Control Interface BACI presented in another document. 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.

4.2.1 Methods

A method called on the Abean is called on the remote device. This pass through 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 telescope to move to a given direction) 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

[†] Experimental Physics and Industrial Control System, a commonly used package at accelerators

- Abeans check for device errors and report them
- Abeans can log remote method calls (a replay function is under development)

4.2.2 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 power supply) or `PatternProperty` (representing bits, describes e.g. the state of a power supply), in read-only or read-write flavor (current on power supply can be set / read, but readback on the power supply 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. An `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 power supply 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

4.2.3 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 the simulation 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 hard coded 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 behavior 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' command and a 'set' command to a power supply: 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 full code generator from IDL to Abeans has actually been written.

4.3 Pluggable Layer

A short conceptual overview of the pluggable system has been given already above. Here we 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 control system 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 the types requested consist of double properties, pattern properties, int properties and string properties.

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.