

Atacama Large Millimeter

KGB-DOC-01/05

Revision: 2.1

2001-09-15

*Software
Manual*

Gasper Tkacik

Abeans Programming Tutorial

Software Manual

Gasper Tkacik (gasper.tkacik@ijs.si)

KGB Team, Jozef Stefan Institute

Keywords: KGB-DOC-01/05

Author Signature:

Date:

Approved by:

Signature:

Institute:

Date:

Released by:

Signature:

Institute:

Date:

Change Record

[illegible]

Table of Contents

1 PART I: Introduction.....	5
1.1 Purpose of the Document.....	5
1.2 The Power Supply Example.....	5
1.2.1 Definition of a Power Supply.....	6
1.2.2 The Model of a Power Supply.....	7
1.3 Overview of Abeans.....	9
1.3.1 Java beans, visual builders.....	9
1.3.2 Compile time, design time, run time.....	9
1.3.3 Abeans.....	10
1.3.3.1 Abeans representing devices and properties.....	10
1.3.3.2 What do abeans do?.....	10
1.3.3.3 What Abeans do not do?.....	10
1.3.4 Steps in developing an abean application.....	11
2 PART II: Creating Device Proxies, Implementations and Beans.....	12
2.1 Creating device proxies.....	12
2.1.1 Example.....	13
2.2 Creating abeans for controlled devices.....	15
2.2.1 Example.....	16
2.2.2 Class header and data fields.....	16
2.2.3 Constructors.....	17
2.2.4 Property accessor methods.....	17
2.2.5 Remote methods.....	17
2.2.6 Run method.....	18
2.3 Summary.....	19
3 Part III: Building an application.....	20
3.1 Deciding on how to use abeans: visual mode vs. manual mode.....	20
3.1.1 Finding out the mode of the abean.....	21
3.1.2 Differences between modes.....	21
3.2 SynchronizationLock object.....	23
3.3 Completion objects.....	24
3.4 The connection process.....	24
3.4.1 Manual mode.....	24
3.4.2 Visual mode.....	24
3.5 Our First Abean Application.....	26
3.6 Using Abeans in an Application.....	28
3.6.1 Action command.....	28
3.6.2 Static data item accessors and property accessors.....	29
3.6.3 Accessors, mutators and monitors of the property's value.....	30
3.7 Common events and properties of the abeans library.....	32

3.8 ServiceBean.....	36
3.8.1 Device queries.....	37

1 PART I: Introduction

1.1 Purpose of the Document

The purpose of this document is to illustrate the usage of abeans. It describes step-by-step procedures for creating new device beans and building applications. The main goal is to present several examples with code snippets and screen shots, alert the user about possible caveats and give advice about how to use the software optimally. The document neither explains architecture in depth nor is it a comprehensive API documentation. However, the examples are as broad as possible to cover a wide range of possible applications. It is also advisable to get in-depth acquaintance with the concepts of the basic control interface (BACI), as described in a detailed technical document (BACI Specification) and KGB conference articles.

Part I covers in depth which classes are needed to write a new device bean, revealing some of the principles of the abeans package. By using the implementation of the Power Supply Bean as an example, all necessary steps are explained. This part can be skipped in first reading, in particular if only an application is to be written.

Part II is devoted to the creation of applications in the so-called manual mode, i.e. with textual programming and not using visual building tools to manipulate abeans. Manual application development is preferred in cases where a short test without a GUI is to be run or when a complex algorithm based on many controlled devices is to be implemented. Visual programming, a powerful approach for quickly building GUI-based applications is described in a separate document (Using Abeans with Visual Age for Java Tutorial), as it depends on the particular IDE (integrated development environment). The recommended IDE for Abeans is IBM's Visual Age for Java.

1.2 The Power Supply Example

The fil rouge of this tutorial, used in all examples, will be a power supply. Such devices are abundantly used in particle accelerators to deliver a stable electric current to magnets. Power supplies are well defined objects and are completely independent of other objects, at least in their simplest form, therefore they make good and clean examples. We will describe them here in some detail in particular for those, who do not come from the accelerator community. But also readers familiar with power supplies should read the following few paragraphs in order to get acquainted with the concepts of how abeans are used to model controlled devices.

1.2.1 Definition of a Power Supply

A power supply, as all devices that are modeled by abeans, has two types of items that are controlled remotely from a control system: **properties** and **commands**. The properties of a power supplies are its physical parameters:

- **current:** the reference current, which the power supply should deliver to the magnet. Contrary to conventional power supplies, where the voltage is adjustable (voltage sources), magnet power supplies have adjustable currents (current sources), because the magnetic field depends directly on the current. An electronic circuit with a feedback system constantly adjusts the voltage of the power supply in order to deliver the required current.
- **readback:** the actual current delivered by the power supply, as measured (i.e. read back) by a current transformer built into the feedback loop. Ideally, the values of current and readback are equal, however, due to fluctuations in the feedback loop and measuring imprecision, the readback always differs slightly from the reference current. A case, where the current and readback differ significantly, is, when the power supply is switched off: the readback is zero, independent of what current has been requested.
- **status:** the status is described by a series of alternative conditions: ON/OFF, LOCAL/REMOTE, ALARM/NO-ALARM, READY/NOT-READY, etc.. Each condition is represented electrically by a digital input, or - in software - by one bit, respectively. Depending on the type of power supply, several conditions can be active at the same time. Therefore, the status is a collection of bits, which can be called a bit-pattern. A variable of type unsigned integer is used to keep the values of this bit-pattern.
Note that in order to change the status, one can not simply change the value of a bit; e.g. to change from OFF to ON, the command `on ()` has to be called explicitly (see below). And the transition from NO-ALARM to ALARM is completely out of the hands of the user – an alarm is an external event, for example a burned fuse, a high temperature, etc.

From the above discussion, several concepts can be deduced immediately:

- A property contains exactly one controlled value and is therefore an atomic control point of a device. This concept is known in many SCADA and control systems under the name channel, or tag, respectively.
- A property is described by more data than just its value. There are also limits (maxValue and minValue), some describing text (name, description, units) and other relevant data. They are called **characteristics**. Such data are constants and are usually read from a configuration database.
- The value of a property can be only read, as in the case of readback, or written, as in the case of current. Properties that can only be read are called read-only, or **ROproperty**. Properties that can be written, can always be read – either the

hardware remembers the last set value, or the software keeps it in memory. Therefore such properties are called read-write, or **RWproperty**.

Commands of a power supplies are actions that are requested from the power supply:

- **on**: switches the power supply on
- **off**: switches the power supply off
- **reset**: resets all alarm bits of the status, if the reasons for the alarms have disappeared. The power supply logic is such that if an alarm is triggered by the hardware, the PS goes off and the alarm bits are left on until a reset command has been issued.
- **ramping**: some power supplies have a built in function generator, which is setting the current as a function of time according to a predefined curve. This is usually used to increase the magnetic field along a well-defined monotonic curve, therefore the name ramping.

A Power Supply also has characteristics of its own: name, position, etc.

1.2.2 The Model of a Power Supply

We believe that the most human-oriented way is to represent each accelerator device with a respective Java Bean. With a control system we control accelerator devices, such as power supplies, beam position monitors, etc. So it is natural that we represent those devices in the computer with Beans. In the network, they become CORBA objects.

The example analysis of a power supply in Table 1 shows how a device object looks and what kind of properties it has.

Table 1: properties and commands of a power supply:

Property	Type	Access
readback	double	Read only
current	double	Read/write
status	bits	Read only
Command	Input type	Return value
on	none	void
off	none	void
reset	none	void

In most cases, all properties have the same type of characteristics. There is a difference between read-only and read-write properties, as the former have no command to write. Another difference is between properties of type double and type bits (actually an unsigned integer containing a bit pattern), as the latter have no minimum or maximum. But all these differences can be systematized into a matrix of few classes (see Appendix A and references therein).

An example of a model of a power supply using CORBA IDL (skipping the formalities) is:

```
interface PowerSupply : DO {  
    // properties  
    readonly attribute RWdouble current;  
    readonly attribute RODouble readback;  
    readonly attribute ROpattern status;  
    // commands  
    void on(CBvoid);  
    void off(CBvoid);  
    void reset(CBvoid);  
}
```

The interface of the power supply extends the generic object DO (which has a name, methods to access resources and other general properties and methods) and contains 3 properties: current, readback and status. The power supply can execute 3 commands: on, off and reset.

Most of the device commands and property methods are executed asynchronously by the remote object. The results of the operations are communicated to the client by means of a *callback*. A callback is an object interface that must be implemented by the client, so that it can be invoked by the remote object. During this process, the remote object functions as a client and the client performs as a server.

A client will often need to get the value of a property on a regular basis, either at given time intervals or whenever the value changes. A regular callback with the updated value is invoked by means of a *monitor*. The client creates a remote monitor on the server with a single call, where a reference to a callback is passed as a parameter. Then the monitor on the server invokes the callback whenever the requested conditions are met. An important type of monitors are *alarms*: A client registers a callback which is triggered every time an alarm condition appears, changes, or disappears.

Java Beans use events to communicate data, which are just a special type of callbacks. Therefore all callbacks that have been described above are mapped into Java events, as will be seen in the later chapters.

1.3 Overview of Abeans

1.3.1 Java beans, visual builders

Java beans are components that can be manipulated in a visual builder tool. A visual builder tool, for example IBM's VisualAge for Java, is an integrated development environment with the capability of automatic code generation. Java beans are ordinary java classes and thus retain all advantages of object oriented programming, like code reusability and easier maintenance; they are, however, also a special subset characterized by conformance to java beans design patterns. The most important benefit of java beans architecture is interaction with the visual builder tool. It recognizes beans as a special subset of java classes; it analyses them and offers the user their services. These take the form of events that the beans fire (for instance when their internal state changes) or properties (values that can be retrieved and / or set). To use the beans the user can then graphically establish the connections between them. For instance, drawing a line which begins in one bean and ends in the other might be interpreted by the visual composition editor as follows: when an event has happened in the first bean (if the first bean is a button, someone might have pressed it and thus it generated an event), invoke a method on the second bean (the second bean could be a File Open dialog and the method invoked could be *show*). From this graphical representation the visual builder would then generate the code, which would amount to an application with a button, which, when pressed, shows the File-Open dialog box. Of course the visual builders and the beans offer many more ways of interaction, but it is not our purpose to discuss them here. To summarize: beans in visual builders enable the user to generate the application without writing a single line of code. Moreover, learning how to create an application involves learning how to use the development tool but at the same time reduces the amount of knowledge the user has to have concerning the actual bean.

1.3.2 Compile time, design time, run time

Java and especially java beans have divided the life cycle of an application into three main phases. The compile time and run time are known from classical C/C++ programming: at compile time the syntax of the programs is checked and the output of this process is a valid executable file (in case of java, this is a `class` file). During run time this file is loaded and executed (in Java Virtual Machines some security steps are taken additionally, such as verifying). For our discussion the most important phase is the design time, that is the period in application development, when the beans are actually inside of the visual builder and the application is being built. This is important because the beans can (and do) detect, whether they are currently in design time and can modify their behaviour accordingly. When the application builder is creating a button and dropping it into the frame, for instance, s/he is not just manipulating the picture of a button. A button was actually created by the visual composition editor and is fully functional. Abeans for example detect the environment and if it is design time, they will not attempt network connection to remote objects (or device server).

Since the visual builder and the beans are so interconnected, a tutorial that tries to provide some insight into the visual programming cannot be written without reference to

some specific visual builder tool. Since the author and abeans developers have used IBM VisualAge for Java for this purpose, all of the snapshots and examples in this tutorial refer to it.

1.3.3 Abeans

1.3.3.1 Abeans representing devices and properties

Abeans stands short for Accelerator Java beans. Abeans are java bean components that wrap up all the functionality needed to access the control system. They enable the user to view the control system as a collection of devices (there is one abean type for each device type) and thus provide intuitive approach to application writing. An accelerator device is thought to be a container consisting of simple methods (like `on`, `off` on a `PowerSupply`) and properties. Properties represent physical quantities or device states that need to be controlled. A `current` in a `PowerSupply` is a good example. A current, however, is not a single `double` value, it also has to be put into a physical context: it has maximum and minimum values, it can be periodically monitored (let us say once per second), its value can be retrieved and set, it can trigger alarms if the value is outside a certain range etc. To make things manageable, all methods that concern a property like `current` are grouped into a `Property` class. There is actually a number of such classes, each representing a type, for example `RODoubleProperty` (stands for read-only double property, for instance `readback` of a `PowerSupply`), `RWDoubleProperty` (read-write double property, for instance our `current`), `ROPatternProperty` (read-only pattern property, could represent a bit collection describing the state of a `PowerSupply`), `RO-` and `RWStringProperties` etc..

1.3.3.2 What do abeans do?

Abeans hide network communication details, manage object instantiation, handle errors, exceptions, timeouts, expose a consistent and uniform java beans interface to the user and the visual builder - this means that all methods (regardless of whether they are synchronous or asynchronous in the actual control system implementation) look like ordinary java method calls, that communication with other beans is done through java beans events mechanism etc., and also provide a number of additional services such as logging, persistent settings storage and the like.

1.3.3.3 What Abeans do not do?

Abeans do not visualise data. More simply put, abeans do not have any means to display the data since they do not have any visual representation. An abean is an invisible bean; it leaves the work of displaying the data to other beans, perhaps commercial ones (like trend charts), or standard ones (java swing components), or even components written specifically for interaction with abeans (like `Gauger`). With this approach any visualisation of the data can be chosen and used (even more than one for a single quantity) and as new updated components become available, they can be used easily

without rewriting the code. Abeans and graphical beans communicate through events and properties (sometimes with the help of adapters).

1.3.4 Steps in developing an abean application

To use abeans, the following steps have to be taken:

1. **Installing abeans packages into the development tool:** The procedure involves importing the packages, setting the design time and run time environments. A special document describes these steps for IBM's Visual Age for Java, the preferred development environment for abeans.
2. **Configuring abeans system:** The abeans system provides a centralized mechanism for applications to store configuration data. Therefore it requires a special configuration path to be set aside for the storage of configuration files. This path needs to be present in java classpath system variable. By default, two configuration files are located there: `AbeansSystem.data` and `DefaultAbean.data`. The first file contains only the pointer to the path itself. `DefaultAbean.data` contains default property values used by the abeans system as application options. These are, for instance, default connector type to be used when none is specified explicitly, timeout, all available plug libraries etc. The configuration can be changed by starting the `AbeanCustomizer` application, which comes with the abeans distribution, as do the two configuration files with default values. If an installer has been used, it is usually not necessary to perform any changes. If the configuration path is not present in the classpath, it must be specified explicitly using the command line option `-Dabeans.config` Java System property.
3. **Creating device proxies, implementations and beans:** An abean is a bean representing controlled accelerator device. A proxy is an interface describing the device methods and properties. A proxy is very similar to IDL specification, the difference being only in that it is CORBA independent. There is one proxy implementation per pluggable subsystem. An abean, a proxy interface and a proxy implementation for a given pluggable subsystem must be present for a single device type to be accessible from the abeans system.
4. **Building an application:** The chapter demonstrates some of the most used abean functions and design patterns for an application that uses abeans, discusses issues connected to building applications manually and visually.

2 PART II: Creating Device Proxies, Implementations and Beans

Note: For most of the devices, proxies, implementations and beans are already written. Normally, there is no need to do this by hand, since these classes will all be provided. Therefore you can skip to the next section, as this is provided for the sake of completeness and architecture overview.

Note: It is possible to create code generators that automate the procedures outlined in this chapter. Documentation for code generators is provided in a separate document.

2.1 Creating device proxies

A device proxy completely describes the interface to the device, that is, the interface through which the device is controlled. It also completely determines the interface of the abean that represents the device. This dependence is so strong that it is possible to write a generator that automatically creates an abean from the device proxy.

How can a device proxy describe a controlled device? It does so exactly like an IDL definition, only in Java language instead of IDL. Because the purpose of the IDL definition and device proxy is so similar, abeans adopt the following conventions on how to create a device proxy from the IDL:

- Take the IDL file compiled to java; leave the number and method names intact.
- Perform these transformations:
 - Let device proxy reside in `si.ijs.anka.abbeans.devices` package; this is the default. It is also possible for the device, its proxy and other necessary classes to reside in non-default packages.
 - If the IDL device extends another IDL device `x`, then device proxy extends device proxy for `x`.
 - If the IDL device does not extend any other IDL device, then device proxy extends
`si.ijs.anka.abbeans.pluggable.AbstractDevice`.
 - If an IDL method takes a basic type for a parameter, so does the proxy method (type remains intact)
 - If an IDL method takes a callback for a parameter (for instance `CBvoid` or `CBdouble`), the proxy method takes the corresponding callback from the package `si.ijs.anka.abbeans.datatypes` for the parameter (`VoidCallback` or `DoubleCallback` respectively).

- If an IDL method returns a basic type for a parameter, so does the proxy method (type remains intact)
- If an IDL method returns a class type, the proxy method returns the corresponding type proxy (for instance, if an IDL method returns `RWdouble`, proxy method will return `RWDoubleProxy`; if an IDL method returns `Monitor`, proxy method returns `MonitorProxy`).
- All these proxy return and parameter types are already created and reside in `si.ijs.anka.abbeans.datatypes` package.

2.1.1 Example

Let us create a device proxy from the IDL for `PowerSupply`:

`si.ijs.aci.powerSupply.PowerSupply` generated from IDL2java from IDL for `PowerSupply` (see BACI specification)

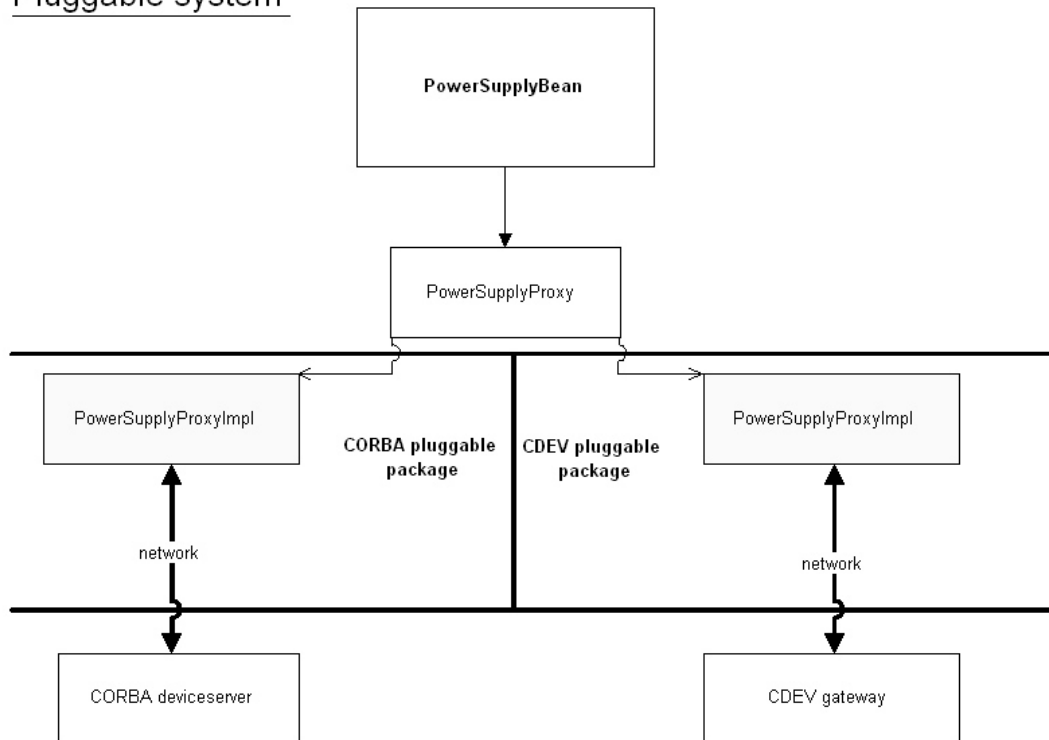
```
package si.ijs.aci.powerSupply;
public interface PowerSupply extends si.ijs.aci.Device {
    public si.ijs.aci.RWdouble current();
    public si.ijs.aci.ROdouble readback();
    public si.ijs.aci.ROpattern status();
    public void off(si.ijs.aci.CBvoid cb);
    public void on(si.ijs.aci.CBvoid cb);
}
```

`si.ijs.anka.abbeans.devices.PowerSupplyProxy`

```
package si.ijs.anka.abbeans.devices;
public interface PowerSupplyProxy extends
si.ijs.anka.abbeans.pluggable.DeviceProxy {
    public si.ijs.anka.abbeans.datatypes.RWDoubleProxy current();
    public si.ijs.anka.abbeans.datatypes.RODoubleProxy readback();
    public si.ijs.anka.abbeans.datatypes.ROPatternProxy status();
    public void off(si.ijs.anka.abbeans.datatypes.VoidCallback cb);
    public void on(si.ijs.anka.abbeans.datatypes.VoidCallback cb);
}
```

You see that there is a complete one-to-one correspondence between the interface created from the IDL and the device proxy. Why then do double work? Because all of the types used in IDL generated file (such as `ROdouble` or `CBvoid`) are finally extended from classes that need CORBA libraries to function. Because abeans want to be CORBA independent and writing proxies is such a simple task, device proxies such as the one described above are used. As you can see, proxies for properties and callbacks already exist (we did not need to write special proxies like `RWDoubleProxy` and `VoidCallback` which we reference, because they are already there). Note the name patterns, they are important: apart from callbacks (which are technically not really proxies, but we will leave that to specialized chapters) all proxy names end with "Proxy".

A `PowerSupplyProxy` is now complete. This proxy serves as a link between the part of the abeans which is independent of the communication subsystem - these will be our `PowerSupplyBean` when we write it) and the part, which is dependent on the communication subsystem (these are classes in the pluggable subsystem packages). Therefore `PowerSupplyBean` stays the same regardless of the communication subsystem, while the abeans mechanism takes care at run time to actually load the correct pluggable package the user wants. For instance, if the user tells the `PowerSupplyBean` to connect using CORBA, abeans mechanism will load the pluggable package `si.ijs.anka.abbeans.pluggable.CORBA` and in it, find the `PowerSupplyProxy` implementation for CORBA.

Pluggable system

The diagram illustrates the relationships between `PowerSupplyBean`, `PowerSupplyProxy` and proxy implementations. The bean only sees a proxy that is independent of the communication subsystem. Then, during run time, the user tells the `PowerSupplyBean` whether s/he will use CORBA, the simulation or some other communication protocols like low level socket protocol to connect. Based on this decision the abean system then loads appropriate pluggable package with the proxy implementation. All remote calls that the `PowerSupplyBean` makes on the proxy are in turn called on the appropriate proxy implementation.

2.2 Creating abeans for controlled devices

As we have already mentioned, a device proxy determines the interface of the bean. Beans, like the `PowerSupplyBean` that we will write, have a number of methods, which can be divided into four categories:

- Constructors (by default, each abean for a device must support at least two constructors)
- Property accessor methods (because abeans for devices contain properties, they have to provide methods for accessing them)
- Remote methods (such method called on the bean is immediately called on the remote object)

- o A `run()` method (which executes the connection code)

By default, beans are placed into `si.ijs.anka.abbeans.devices` package along with their proxies. Let us start with an example. We will be creating a `PowerSupplyBean`:

2.2.1 Example

```
package si.ijs.anka.abbeans.devices;
import si.ijs.anka.abbeans.*;
import si.ijs.anka.abbeans.datatypes.*;
public class PowerSupplyBean extends DeviceBean
{
    private PowerSupplyProxy remote;
    public PowerSupplyBean() throws InitializationException {
        super(si.ijs.anka.abbeans.devices.PowerSupplyProxy.class);
    }
    public PowerSupplyBean(ConnectionParameters connParams) throws
InitializationException {
        super(si.ijs.anka.abbeans.devices.PowerSupplyProxy.class, connParams);
    }
    public ROPatternProperty getStatus() {
        return (ROPatternProperty)getProperties().get("status");
    }
    public void run() {
        try
        {
            remote = (PowerSupplyProxy)getConnector().connect();
            setHandle(remote);
        } catch (Exception e)
        {
            reportError(toString() + " Connection Exception.");
            reportStatus(new StatusReportEvent(this, toString() + " Additional
exception info: " + e, false, true));
            return;
        }
        reportConnectionStatus(si.ijs.anka.abbeans.Constants.CONNECTION_SUCCESS);
        return;
    }
    public void off() throws RemoteException {
        VoidCallback cb = VoidCallback.newInstance(this, "off", this);
        try
        {
            remote.off(cb);
            cb.startTimer();
        } catch (Exception e)
        {
            throwCallException("/off network error.", e);
        }
        return;
    }
}
```

Note: this bean does not implement `on`, `reset`, `getCurrent` and `getReadback` methods, because they are completely analogous to the `off` and `getStatus` methods.

2.2.2 Class header and data fields

Let us dissect it step by step. First, notice how we extend `DeviceBean` class; we do it because in the IDL specification a `PowerSupply` extends `Device` (`Device` has functions like `shutDown`, `startUp` and the like). `DeviceBean` in turn extends `si.ijs.anka.abbeans.Abean` which is the root of all abbeans that represent devices. Then we create a private handle called `remote` to a `PowerSupplyProxy`. You can see

now, why this bean is independent of the communication subsystem. It only uses proxies and these are the same for all subsystems.

2.2.3 Constructors

Two constructors are present. One is the default constructor. It takes no parameters. Such a constructor **must** be present if the bean is to be used visually. A visual builder always instantiates beans with their default constructors. The only task of the default constructor is to pass the proxy class to the superclass constructor. The root class of all beans representing devices `Abean` will introspect this class (at runtime query its methods etc.) to determine from the proxy which properties are contained in the device. In our case, we pass `PowerSupplyProxy` to the superclass `DeviceBean`, which passes it to its superclass `Abean`. `Abean` will inspect `PowerSupplyProxy` and find three methods, namely `current`, `readback` and `status`, which return property proxies. It will thus deduce that it has to create `RWDoubleProperty` called `current`, `RODoubleProperty` called `readback` and `ROPatternProperty` called `status`, it will initialise them and put them into a hashtable, accessible through `getProperties()` function of the `Abean` class.

The other constructor does exactly the same as the default one, with the addition that it passes over to the superclass an instance of `ConnectionParameters`, which describes bean name, family and the like. These concepts will be explained later.

Invoking either one of the constructors determines the bean mode. If the default constructor is invoked, bean will be created in automatic (visual) mode. If the other constructor is invoked, the bean will be created in manual mode. Both modes of usage differ, and their differences are explained in *Deciding on how to use abeans: visual mode vs. manual mode*.

Constructors throw `InitializationException` on invalid parameters or (if in manual mode) if the connection has failed.

2.2.4 Property accessor methods

`getStatus()` method is a property accessor method. It returns an instance of `ROPatternProperty` named `status`, which is consistent with the proxy description. As explained in the constructors, this object has been created by the `Abean` and placed into a hashtable. All we have to do is to take the property out of the hashtable and return it.

Properties are stored in a hashtable with their name as a hash.

Why should the bean have a special method for each property that returns it (we could always access it manually from the hashtable)? It is not just a matter of convenience: if the bean has method without parameters with name that starts with "get" this is a sign to the visual builder, that there is a *java beans property* with name that follows "get" and type that equals the "get"- method return type. This method is therefore a sign (a design pattern) that tells something to the visual builder. We shall see later why that is useful.

2.2.5 Remote methods

`off()` is an example of a remote method. If you look closely into the code, you will see that it calls the `off` method on the `PowerSupplyProxy` remote. In addition it catches exceptions, reports them and re-throws as `RemoteException` – see documentation on

`throwCallException()` (so that all exceptions generated by the method are `RemoteExceptions` and not ones specific to the underlying communication subsystem). Another thing that a remote method must take care of is callback management. Obviously the `off` method is asynchronous: it takes a single `VoidCallback` parameter. When `off` is invoked on a proxy, a proxy invokes it on the remote object (for instance device server). Such a server returns immediately, even before the power supply has actually been turned off. When the request comes to the real, physical power supply and is fulfilled the device server calls back (therefore a "callback") a `VoidCallback` object and reports to it about the final success of the "off" operation.

Our `off` method hides the whole complicated process from the user. But in order to occur, we must create a callback, which is done with static `newInstance` method, pass it to the proxy function (in call `remote.off(cb)`), and start the timer on callback. The timer will notify us through `TimeoutEvent` (to be discussed later) if there was a timeout. It is stopped automatically if there is no timeout. Generally, callbacks are complicated objects and their constructors support a number of different parameters and configurations.

Sometimes we may want to get an event whenever the callback has occurred. Such an event, called `CallbackDoneEvent` is already prepared in the abeans library. To have it delivered when the callback completes (regardless of whether it has completed with an error or no-error code – this can be queried from the event), the callback creation stanza is somewhat different:

```
VoidCallback cb = VoidCallbackInitializer.newInstance(this,
"off", this, this, getCallbackDoneListeners(),
Constants.CB_SIMPLE, "si.ijs.anka.abbeans.CallbackDoneListener");
```

The code segment shows that the callback management is a very powerful feature of abeans: the programmer can select exactly into which listener interface the callback will fire the events when it completes; you must, naturally, also provide an array of listener objects (that is the purpose of `getCallbackDoneListeners()` method in our example). The rules for callback notification dispatching are very strict and are described in the reference manual.

2.2.6 Run method

This method must be implemented in every bean, because the root bean `Abean` declares that it implements the `Runnable` interface. It enables the bean to connect in a separate thread. This method is never called explicitly in the user code; we will see later how to tell the beans when to connect. The heart of connection is the call `remote = (PowerSupplyProxy)getConnector().connect()`. The method `getConnector()` of the `Abean` parent class returns a class that is responsible for connection. All we have to do is call `connect` and cast the return value into `PowerSupplyProxy`. We then pass the handle into the superclass with `setHandle` method. If the connection has failed (an

exception was thrown and `catch` block is being evaluated), the bean reports the fact and returns from `run`, otherwise, we notify the hierarchy that the connection was successful by a call to `reportConnectionStatus` function (this function will also inform, by means of events, other interested parties that this bean has connected).

2.3 Summary

This section has demonstrated how classes that are specific to a device are written. It shows that so much of the functionality is hidden in superclasses and support classes that writing `PowerSupplyBean` is very straightforward and involves writing a lot of code that repeats itself - to see how an `on` function, which is not in the example, would look like, just exchange every occurrence of "off" in `off` function with "on"! Furthermore, you see how an abean, which represents a device, constitutes a central entity in the abeans collection of classes. All other needed classes, like properties, connector classes and the like are created automatically by internal mechanisms.

There exist generators for canonical abeans methods that take proxy declaration as an input and generate bean, beaninfo and proxy implementation. Canonical methods are action methods, which take a single callback as a parameter. Other action methods (arbitrary number of parameters) can then be written by hand. The development of beans for new devices is therefore only a matter of applying an existing template to the bean specific data.

We have also created more powerful generators that produce code directly from the BACI compliant IDL stored in CORBA Interface Repository. Such generators use XML and XSLT technologies to code canonical BACI design patterns and copy non-BACI compliant constructs verbatim to the output.

3 Part III: Building an application

Before trying to build an application, we should explore abeans more in depth. The areas of interest to cover are:

- Choose the mode how to use abeans – either in visual or manual mode
- The connection process, including families, `ConnectionParameters` objects and `ServiceBean` objects
- Abean events and methods
- Property events and methods, including monitors and alarms

Note: Abean spelled with a capital letter denotes class `si.ijs.anka.abbeans.Abean`, the superclass of all abeans that represent a device.

3.1 Deciding on how to use abeans: visual mode vs. manual mode

An abean can be used either in a visual development tool or in a manually written procedure. Both modes differ somewhat in behaviour in order to simplify the most common tasks (like synchronization and connection) that are typically programmed in each mode

Why is it beneficial to the user of abeans to have two modes of usage? The component behaviour differs in procedural programming and GUI programming. GUI should be asynchronous by nature, so that the process execution does not block the graphical components. The goal of the visual mode is to make the system responsive. On the other hand procedural and script programming must guarantee a linear execution flow, where subsequent statements depend on the completion status of the first statement. Consider the following code snippet:

```
...
PowerSupplyBean ps = new PowerSupplyBean(new
ConnectionParameters("PS1"));
ps.on();
ps.getCurrent().set(0);
ps.off();
...
```

It creates a `PowerSupplyBean` in manual mode (it would be visual if default constructor was used), which connects to a device called "PS1". Then, in succession, it calls `on` method, sets the current to 0 and calls `off` method. Do these events really happen in succession? There is no guarantee: all three calls are in fact asynchronous, meaning that `on()` returns before the physical power supply was actually turned on. What would happen if somewhere on the network or field bus the packets got mixed up so that the `set` on current would be called before `on`? Such things could easily happen and the results are very unpredictable. A scenario that is even more probable would be the

successful delivery of the "on" request, which for unknown reasons fails on the power supply device. Further function calls should probably not occur after failure. But you cannot check the successful completion when you are calling `set on current`, because the callback for `on` has not yet been received.

Now these problems could easily arise in manual programming, where the above sequences are common. A means of locking / synchronization is therefore needed: wait at the `on` call until the callback has been received, check the status and if it is OK, continue. Quite another approach is appropriate for graphical panels. There asynchronous calls are desired, because they do not block the user interface. If there is an error, the bean is expected to notify you through event mechanism. If there are any synchronization risks, the user interface should be designed so that it is impossible for the user to do things in the incorrect sequence (disable button for action 2 until action 1 completes, for instance). In conclusion, it is desirable to have the beans behave differently in the two modes. It is understood that by such a course of action some confusion could arise, however, we think that the benefits outweigh the disadvantages, especially if the different behaviours are well documented.

3.1.1 Finding out the mode of the abean

For these reasons abeans behave differently when they are used in manual mode (constructed with parameters, like in the example) or in visual/automatic mode (constructed with the default constructor – no parameters). Once the constructor invoked determines the mode, it cannot be changed. You can query the mode of the bean by calling `boolean isVisibleMode()` function on the abean.

3.1.2 Differences between modes

Manual mode	Visual mode
Set when constructor with parameters is executed	Set when default constructor is executed
Asynchronous calls can block (i.e. return only after the callback has been received from the device server), if the synchronization lock is installed	Asynchronous calls do not block (i.e. return immediately after device server returns)
Property monitors are not created automatically, but must be created manually by calling <code>getMonitor()</code> method on a property class	Property monitors are created automatically; when there are any listeners for the property value, the monitor is created; when there are none, it is destroyed
Connection to a remote device is established in the constructor; if the	Connection is established only when <code>ConnectionParameters</code> are set, the bean family

constructor returns successfully, the handle is valid	is allowed to connect and the application signals that it is ready
---	--

Monitoring and establishing a connection will be discussed later; let us now turn the attention again to asynchronous calls.

3.2 SynchronizationLock object

How exactly do the beans achieve the blocking of the asynchronous function calls? You tell the abean to synchronize on the calls by installing a synchronization lock object (this is done by default if the abean is in manual mode, see the table above). The following code snippet illustrates the principle:

```
...
PowerSupplyBean ps = new PowerSupplyBean();
ps.setConnectionParameters(new ConnectionParameters("PS1"));
ps.setSynchronizationLock(SynchronizationLock.getInstance());
ps.on();
ps.getCurrent().set(0);
ps.off();
...
```

The code has exactly the same effect as the first snippet. It creates a `PowerSupplyBean` object, but notice the use of the default constructor. This puts the abean into visual mode, which means that the asynchronous calls do not block. Thus we call `ps.setSynchronizationLock(SynchronizationLock.getInstance())`, which installs a new lock. The remote calls block from this point forward. We can be sure that the current is set only after the power supply has been turned on. This is not the only use of the lock, though. If we extend the code to look like this:

```
...
boolean timeout;
Completion c;
SynchronizationLock lock = SynchronizationLock.getInstance();
ps.setSynchronizationLock(lock);
ps.on();
boolean timeout = lock.isTimeout();
if (timeout == false) c = lock.getLatestCompletion();
...
```

It demonstrates nicely how we can use the lock to query the results of the last asynchronous call: `isTimeout` tells us if the lock blocked for such a period of time that a timeout was generated, while `getLatestCompletion()` returns the `si.ijs.anka.abbeans.pluggable.Completion` object describing the status / failure information about the last call.

In the above example we dealt with callbacks of type `void`, which return no value. There exist callbacks, however, that can return a value, such as `DoubleCallback`. Hence the lock also implements method `Object getValue()`, which returns the value that was returned through the callback call. You must cast the `Object` to the appropriate type (basic types are wrapped in corresponding java wrapper classes).

You can change the lock on the fly. If you set the lock to `null`, no synchronization locking occurs (identical to visual mode default). You can get the current lock that the abean uses by a call to `getSynchronizationLock()`.

3.3 Completion objects

A `Completion` object is used to return the status of the request to a remote device. `Completion` objects are returned in each callback and also by some synchronous calls (like synchronous `set` on a property). In most cases, they are handled by the abeans automatically: in each callback completions are checked against the error-free value. If no errors are detected, nothing happens. Otherwise the error codes are decoded and a string message is dispatched to all `ReportListeners` of the abean. Usually, these messages are printed out into the text area in a panel.

`Completion` object consists of three components:

```
public class Completion {  
    public short type;  
    public short code;  
    public int time;  
}
```

`Type` and `code` describe the nature of the problem (`type 0`, `code 0` means no problems) or status. `Time` is the time when the action that `Completion` object describes was carried out by a remote server. It is an integer representing a number of milliseconds from a reference date (Unix beginning of time). Descriptions of completion codes can be obtained by querying `ConnectorInformation` class in the pluggable package. It produces a string message from `code` and `type` parameters. Consult the API documentation for details.

3.4 The connection process

3.4.1 Manual mode

If abeans are used in manual mode, their lifecycle is simple: they connect when they are constructed. If they connect successfully, no exception is thrown and a non-null handle returned from the constructor can be used directly to make calls to remote objects. They remain connected until the `destroy()` function is called. This function releases resources connected with monitors and other pluggable dependent resources (network connection etc.). As soon as the clean up phase is complete, the abean is marked as dead - you can query its status by calling boolean `isDead()` method. It notifies other abeans (if any listen) that it is in the dead mode by dispatching `PropertyChangeEvents`. Once the bean is dead, it cannot be connected again; you must instantiate a new bean for that purpose, or, instead of destroying the bean in the first place, tell the bean to reconnect using `reconnect(ConnectionParameters c)` method.

3.4.2 Visual mode

The connection process for abeans used in visual mode is performed and managed automatically and transparently by the abeans libraries. This section briefly describes the process that goes on beyond the scenes in order to give some insight on some more advanced concepts of abeans.

Managing connection is more complicated in visual mode for a number of reasons. To name just the most important ones:

- *Java Beans specification requires that visual builders always instantiate the beans with default constructor.* That means that an abean representing a device cannot have the access to the settings that it needs for connection (such as the remote device object) at construction time. Hence an abean must necessarily wait until the builder sets all the properties that it requires for connection (encapsulated by `ConnectionParameters` object), such as remote object name, timeout, family etc.
- *Construction time of the abean cannot be exactly determined.* The visual builder might create all the beans when the application starts or it might wait until the bean is first used (called by another bean, for instance). In an application where there are many abeans, most of them not used unless the user explicitly requests so, it is undesirable for all the abeans to be instantiated and / or connected at the application initialisation. On the other hand, in a simple `PowerSupply` panel application the simplest scenario from the viewpoint of the user is for the application to automatically create and connect the single `PowerSupplyBean` when the application starts.

In order to resolve new problems in timing connected to the differences in automatic code generation and to allow the user to fine tune the exact moment when the connection is to begin, several new concepts had to be introduced. These are:

- A manager object, called `AbeanInitializer`.
- A family object that groups abeans together, called `AbeanFamily`
- An interface that every application which uses abeans has to implement, called `AbeanApplication`

Therefore a life cycle of the abean created in the visual mode is described by the following sequence:

- An abean is constructed with the default parameters and it waits (default timeout, no remote name, default family, default pluggable type)
- Parameters that are required for connection are set by a call to `setConnectionParameters(ConnectionParameters c)`
 - Abean checks if these parameters are valid. If not, an exception is thrown.
 - If they are valid, a `PropertyChangeEvent("connectionParameters")` is dispatched to all listeners

- By default, a special manager class called `AbeanInitializer` listens to these events. When it receives an event, it inspects it to see to which family the abean belongs. It then adds the abean to the specified family.
- If this manager is notified by the application that the application itself is ready (for example, that it has painted itself), it scans through all existing families
- If a family is a default family (a special family into which all abeans that have not explicitly stated to which family they belong, are automatically placed) the abean begins connection process.
- If the family is not a default family, then the manager looks if this specific family is allowed to connect. If yes, the abean begins connection process. If no, the abean waits for the explicit approval from its family, telling it that it may connect.

Fortunately, all these steps are handled by the Abean framework and the designer of an application in a visual builder environment does not need to worry about them at all. The visual mode will not be discussed further in this document, as the life cycle of beans in visual mode is handled automatically. Other features of working with abeans in visual mode depend on the choice of the visual builder environment. A separate tutorial exists for using Abeans with Visual Age for Java, our official VBE.

3.5 Our First Abean Application

Although the process described in the previous sections seems a bit complicated, it is actually very straightforward to use, which can be seen from the following code snippet, where a power supply abean is created in manual mode:

```
class Test implements AbeanApplication {

public static void main(String arg[]) {
new Test();

    public Test()
    {
        AbeanInitializer.initialize(this);
        try
        {
            PowerSupplyBean ps = new PowerSupplyBean(new
ConnectionParameters("PBEND_M.01"));
            /* operations on the power supply */
            ...
            ps.destroy();
        } catch (Exception e)
        {
            System.err.println(e);
        }
    }
}

public ApplicationIdentifierSupport getApplicationIdentifier() {
```

```
return new ApplicationIdentifierSupport(this);  
}  
}
```

The code snippet shows a fully functional application that connects to a single device of type `PowerSupplyBean` named `PBEND_M.01`. An application that uses the abeans system must implement `AbeanApplication` interface, which declares a single method `getApplicationIdentifier()`. In addition, the first method from the abeans library that is invoked by the program must be the static invocation of `AbeanInitializer.initialize(this)`, passing an instance of `AbeanApplication` as the parameter. The purpose of such program header is the following:

`AbeanInitializer` constructs objects necessary for the abeans system functioning. It queries `AbeansApplication` instance passed as `this` parameter to the `initialize()` call for an `ApplicationIdentifierSupport` object. This object contains the application name (which is by default the class name of the application), application resource path (relative to the abeans configuration directory) and similar settings.

- Based on data stored in `ApplicationIdentifierSupport`, the `AbeanInitializer` will load the configuration data for the abeans system and the given application; it will instantiate a default abean family, which, by default, contains each instantiated abean. If the `AbeanApplication` is actually of `VisualAbeanApplication` subtype, it will register as `ApplicationStatusListener` to the `VisualAbeanApplication`. It is then up to the `VisualAbeanApplication` to inform `AbeanInitializer` about the state of the application (initialising, closing), and the `AbeanInitializer` will respond by either telling all abeans to connect in case of initialising event, or to destroy themselves in case of closing event.
- After the abeans system has been initialised, the abeans components can be used. In try-catch block a new instance of a `PowerSupplyBean` is constructed. Because a non-default constructor is used, the abean is created in manual mode and it connects immediately as a side effect of construction. If the constructor returns without raising an exception, the connection has been successful (remote methods can be invoked). The abean has also been automatically added to the `AbeanFamily.DEFAULT_FAMILY`, i.e. an application-wide container that holds all abeans that have been instantiated and have not yet been destroyed (regardless of whether they are connected or not). Comment lines are where the actual remote command invocations can be inserted. How the actual communication is performed will be demonstrated shortly. Finally, the abean is destroyed by calling `destroy()`. This guarantees that the abean will release all remote resources that it will be removed from any containers and will be put into a state, where no remote function can be invoked. The application programmer

should let such abean be garbage collected. See reference manual for documentation about `AbeanInitializer` and `AbeanFamily`.

3.6 Using Abeans in an Application

When a reference to a connected abean exists, there are several ways of communicating with the remote device:

- Action commands; an action is a remote asynchronous method, which can take any number of parameters. It normally causes an execution of a time-consuming request on remote device.
- Static data accessors; a static data accessor method reads a static data item value from the device server. Since the request propagates to some sort of easily accessible remote database, the call is synchronous.
- Property accessors; a property accessor returns the reference to the remote property that is contained within a device
- Various get / set methods and monitors; since a property encapsulates a value in a context, properties provide a set of accessor and mutator methods that access the value and also provide push style value delivery from the server to the abean, called monitor, based either on periodic timer or delta value change criterion.

3.6.1 Action command

An action command is of the form `void action_name(param1, param2...)`. For instance, a power supply supports action `on`, which is invoked in the following manner:

```
ps.on();
```

When the execution flow enters `ps.on()`, several execution paths are open:

1. Each remote method may throw a `RemoteException` to indicate that the error has occurred somewhere in the pluggable (communications) layer
2. Each remote method is timed and may raise a timeout. A timeout condition is raised when the bean containing the action method (a device bean or a property bean) fires a `TimeoutEvent`. As a timeout side effect the same bean will also fire a `ReportEvent` with timeout error condition.
3. If the method successfully reaches the remote object, is processed there and returns the completion data through the callback object, the action may:
 - Complete without any further notification
 - Complete by firing `CallbackDone` event that carries completion information

- Fire a `ReportEvent` with an error condition as a result of a completion object that indicates such a condition; this event will be fired as a side effect to any of two execution flow options 1) and 2).

Whether the action will fire `CallbackDone` events depends on the specific implementation of the bean. For performance reasons actions, for which the completion will not be checked, should not fire `CallbackDone` events. Some critical actions, however, should do so. The behaviour of the bean is thus discretionary and left to the bean implementer.

The following code segment illustrates a complete treatment of a remote action: The command ramp is called on a power supply. The code handles all three cases 1) 2) and 3) mentioned above. A `ReportEventListener` has not been implemented here, because this is usually done by a GUI component or by the main class:

```
class TimeoutListenerImpl implements TimeoutListener {
    public void timeout(TimeoutEvent e) {
        System.out.println("Timeout while executing: " + e.getCallback());
    }
    public void timeoutsStarted(TimeoutEvent e) {}
    public void timeoutsEnded(TimeoutEvent e) {}
}

class CallbackDoneListenerImpl implements CallbackDoneListener {
    public void callbackDone(PropertyUpdateEvent e) {
        System.out.println("Completed: " + e.getDataSourceName() + " with completion " +
            e.getCompletion());
    }
}

ps.addTimeoutLister(new TimeoutListenerImpl());
ps.addCallbackDoneListener(new CallbackDoneListenerImpl());

try {
    ps.startRamp(rampData);
} catch (RemoteException re) {
    System.err.println("Error while executing remote action: " + re);
}
```

The code creates two listener objects, that handle timeout conditions and invalid completion conditions and it also catches remote exceptions. See reference manual for specific documentation about the supported features and data that can be obtained through events. The try-catch block could have contained a sequence of remote actions and the code would still remain fully functional, including the event handling. The `startRamp()` action used as an example takes `rampData` parameter only. No callbacks are visible at the beans API, because they are handled behind the scenes and reprocessed as events.

3.6.2 Static data item accessors and property accessors

Static data item accessors and property accessors fetch information that is considered static during the lifetime of the remote object. Static data item accessors return a typed value and are synchronous remote calls. Property accessors return references to property objects. Since the data returned is static, it is buffered by abeans system, i.e. only the first call to the method is truly remote. Otherwise the methods return the buffered values with

virtually no overhead. Property references are even fetched during connection time, so that they are available as soon as the bean gets connected. The user may enforce the buffers to be flushed by calling `refresh()` on the `Abean` instance.

3.6.3 Accessors, mutators and monitors of the property's value

Property accessors and mutators behave exactly like action commands invoked on property objects and they will not be described separately. The same rules apply as in the subsection on action commands; even the example just needs to replace the lines

```
ps.ramp();
```

by

```
ps.getCurrent.get();
```

The discussion will concentrate on the monitoring of the property's value. By creating a monitor, the value of the property can be obtained as event notification when the value of the property changes. There are several important points that must be mentioned with respect to monitoring:

- When a user creates a monitor on a property, s/he will receive event notifications about value changes of the property whenever the property value changes, i.e. whenever the abean property receives a new value from the remote CORBA object.
- The user will receive one event immediately after the monitor has been created.
- There are two basic modes of a monitor that differ in the triggering condition: *on-change* and *on-timer*. The setting of the mode influences the **server trigger**, i.e. the condition that has to be fulfilled for the server to send a callback to the bean. Regardless of the triggering condition, the bean will only fire an event if the value changed, i.e. if `oldvalue != newvalue`.
- On-change triggering is useful because the response of the system is very fast, that is, as fast as it is possible. Naturally, that consumes a lot of bandwidth and processing power. On-timer notification guarantees that the bean will receive periodic value updates, which will not be more frequent than the specified interval (so that there is no flooding), but will also not be late (so that the monitoring pulse can be used as a heartbeat).
- The heartbeat (on-timer monitor) is interpreted by the beans in the following way:
 - If the value actually changed, a new instance of `PropertyUpdateEvent` is fired by the monitored property to all listeners. The event data contains the new value, completion and timestamp.

- If the value remains unchanged, no `PropertyUpdateEvent` is fired. However, certain fields in the monitored property bean are updated, for instance the `latestReceivedTime` field, which holds the timestamp of the latest received monitor callback.
- If the bean has received no callback from the remote object in a certain time period, a timeout condition is raised. The property will fire a `TimeoutEvent` (`timeoutsStarted()` method of the listener interface). The timeout condition persists until a callback (heartbeat pulse) is received again by the property, after which it will fire a new `TimeoutEvent` (`timeoutsEnded()` method of the listener interface).
- At any time the property bean may be queried by invoking `latestReceivedTime()`, `latestReceivedValue()` and `latestReceivedCompletion()` methods. **These methods are all non-remote** and will return the latest (most fresh, by timestamp comparison) value that the property holds. This value is therefore the value of the latest monitor. Note that these methods may return uninitialized values, if no monitoring has been enabled on the property. If the monitor existed and has been destroyed, the methods will return the latest available value.

The following code snippet illustrates how to monitor a property value:

```
class PropertyChangeListenerImpl implements PropertyChangeListener {
    void propertyChange(PropertyChangeEvent ev) {
        PropertyUpdateEvent e = (PropertyUpdateEvent)ev;
        System.out.println("New property value: " + e.getDoubleValue() + " time: " +
            e.getDeviceServerTime());
    }
}

RWDDoubleProperty current = ps.getCurrent();

try {
    current.addPropertyChangeListener(new PropertyChangeListenerImpl());
    PropertyMonitor m = current.getMonitor();
    Thread.sleep(100000);
    m.destroy();
}
```

`PropertyUpdateEvents` are propagated as standard java beans `PropertyChangeEvent` subclasses through the `PropertyChangeListener` interface (this facilitates the use in visual builders). The `getPropertyName()` method of the `PropertyChangeEvent` will always be `"latestReceivedValue"` and the `getNewValue()` will return the new value of the property. If, however, the type of the property is known in advance (for instance, if it is a double), a `getDoubleValue()` invocation is much faster (see example). An equally efficient way for the event listener is to query `event.getSource().getLatestReceivedValue()`, since the source of the event will always be the property the value of which has changed. The abeans system

also guarantees that the `getLatestReceivedValue()` invoked on the property will return the new value when queried in response to a `PropertyUpdateEvent`. Note that the actual monitoring starts when `getMonitor()` is called by the user (in visual mode this step is unnecessary). Monitors must be destroyed when the monitor is no longer needed. This can be done explicitly, as in our example, or implicitly. The monitor will be destroyed whenever its corresponding device bean is destroyed. In visual mode, the monitor handling is different (the monitor is destroyed when there are no more property update listeners registered with the property, or when the visual application is closing, which causes all abeans to be destroyed as a side effect).

Note that the complete implementation of the above example should also include timeout handling, but the issue is fundamentally the same, as it was demonstrated in case of action commands.

For more information about monitoring, see reference on `PropertyMonitor` interface.

There also exists a special kind of monitoring that is started on the whole family of abeans. This is called a `PackedMonitor` and enables the application to receive any number of property value updates in a single network call. The use of the packed monitor is the same as for the ordinary monitor, except that the monitor is created on the family object and not on the single bean. When the packed monitor notification arrives, the fields of all monitored properties are automatically updated (new value, new completion, new timestamp).

3.7 Common events and properties of the abeans library

Practically all information in the Abeans system is obtained through Bean events and Bean properties, in order to optimize their use in VBEs. The following two tables list the most useful ones. For a complete list see the API reference.

Event name	Fired by	Signals
<code>AlarmEvent</code>	RO- property types	An alarm condition. This event is fired if alarm monitor has been created on the property. Alarm event can signal either the beginning or the end of an alarm condition. <code>AlarmEvents</code> are subtypes of <code>PropertyUpdateEvents</code> and carry, in addition to property value, completion codes and timestamp, also an alarm description that is obtained by decoding the alarm type and code.
<code>ApplicationStatusEvent</code>	VisualAbean Application	The event is fired when an application is initialised or when it is being closed. The <code>AbeanInitializer</code> catches the event.

		The default response to initialising phase is the connection of the default family. The default response to the application closing is the destruction of all abeans.
<code>ConnectionEvent</code>	<code>Abean</code>	The change in connection status of an abean. The event is fired 1) on successful connection, 2) on failed connection, 3) on disconnection. When the bind has been completed and signalled by this event, you can call remote methods on the abean.
<code>FamilyEvent</code>	<code>AbeanFamily</code>	The change in family membership. This event is fired whenever an abean is added to the <code>AbeanFamily</code> or removed from it.
<code>Property InitializedEvent</code>	<code>Property</code>	The change in property connection status. After the abean has connected, the connection mechanism automatically reinitialises the properties of that abean. Such initialisation may include monitor creation (if any listeners are registered) and buffering of some static data. After this event has been fired, it is safe to call remote methods on the property instance.
<code>PropertyMonitor Event</code>	<code>Property</code>	The change in property monitor status. The event is fired whenever the monitor is created, destroyed, suspended or resumed. Visual components that display the property value may respond by appropriate visual indication of the monitor status.
<code>PropertyUpdate Event</code>	<code>Property</code>	The change in property value. The property value is changed when a new monitor notification arrives. Note that <code>PropertyUpdateEvents</code> subclass <code>PropertyChangeEvents</code> and are fired into either <code>CallbackDoneListener</code> or <code>PropertyChangeListener</code> interfaces. In the first case they signal that a callback connected with an action has completed. In the second case they signal an arrival of a new monitor value; they also carry the new value, completion, timestamp and a

		reference to the data source.
PropertySequence UpdateEvent	AbeanFamily	This event is used for packed monitors and contains an array of changed properties, their corresponding new values and completions.
ReportEvent	Any object, ErrorHandler instances	A general-purpose report event that carries a string message, flagged with a severity flag. Used especially to display status and error messages.
TimeoutEvent	Abean, Property	Fired by any object capable of invoking asynchronous remote actions. Signals a timeout condition. If it is fired in response to an action timeout, a <i>timeout</i> method of the listener interface is invoked. For monitors, timeouts are identified by their starting time and ending time (i.e. timeout conditions are not created periodically, but persist until a timeout is cleared).

Property name	Bean	Description
connection Parameters (R/W)	Abean	Contains the data necessary for the bean to start the connection, including the remote device name, connector type, connection timeout, and bean family name. Setting this property in visual mode starts the connection if the bean belongs to the default family.
debug (R/W)	Many components	If set to true, the component produces additional debug info and dispatches it to the log service.
synchronizationLock (R/W)	Abean	If set to non-null value, the remote asynchronous calls will block until the callback is received or timeout has expired.
alive (R)	Abean	True if the bean has not been destroyed. If false, the remote methods must not be called; otherwise an exception will be thrown.
visualMode (R)	Abean	True if the bean has been created with a default constructor and is operating in a visual mode.
connecting (R)	Abean	True if the bean is currently connecting, but the connection has not yet been completed.
connected (R)	Abean, Property	True if the connection has completed successfully. Remote methods may be called.
latestChange Timestamp (R)	Property	Indicates the time of the latest value change of the property obtained by monitoring.
latestReceived Completion (R)	Property	Contains the latest received completion object obtained by monitoring.
latestReceived Timestamp (R)	Property	Indicates the time of the latest heartbeat. If the <code>latestReceivedTimestamp</code> and <code>latestChangeTimestamp</code> do not

		coincide, a heartbeat monitor was obtained which carried the same value as the previous monitor.
<code>latestReceivedValue (R)</code>	Property	Contains the latest value received by a monitor. Note that the actual source of this value may be the property or packed monitor. The most recent value is retained in either case.
<code>sourceBean (R)</code>	Property	The bean that contains the property.
<code>monitored (R)</code>	Property	True if the property has an active value monitor.
<code>family (R/W)</code>	ServiceBean	Specifies the family for which the ServiceBean is responsible.
<code>allowedToConnect (R/W)</code>	ServiceBean	Specifies if the family of the service bean can begin its connection process.

3.8 ServiceBean

The **ServiceBean** is a general-purpose bean that can be used for the following purposes:

- It enables the user to query for all available devices and device types on the communication system dependent layer.
- It enables the user to control the families (instances of **AbeanFamily** type).

More than one **ServiceBean** can be instantiated. By default, when a **ServiceBean** is instantiated, it ties itself to the **AbeanFamily.DEFAULT_FAMILY** (i.e. the family to which all instantiated beans belong by default). The family for which a given **ServiceBean** is responsible can be changed by calling its `setFamily()` method. Why is it necessary to have a **ServiceBean** operating on the **AbeanFamily** – why not control the family directly? You can, but it is inconvenient to do so in a visual builder (if you are programming by hand, it is easy). You cannot simply drag & drop an instance of **AbeanFamily**, because they are not beans themselves. Therefore a **ServiceBean** acts as a decorator bean for the abean family instance. Aside from the operations enumerated in the reference manual, most important actions that the **ServiceBean** can invoke on its family are:

- It can multiplex `ReportEvents` originating in all family members and forward them. If many beans are instantiated and need to pass `ReportEvents` to a `TextPane`, you do not have to connect each bean to the `TextPane`; rather

instantiate a `ServiceBean` and connect only its `ReportEvent` feature to the `TextPane`. It will collect the events of the whole family and will forward them.

- If you created some of the beans so that they do not belong to the default family in a visual builder, they will not connect automatically when the application has initialised itself. Rather, the beans will wait for the user to invoke:

```
ServiceBean sb = new ServiceBean();  
sb.setFamily("myFamily");  
sb.setAllowedToConnect(true);
```

- The code segment indicates how to use the `ServiceBean` to signal "myFamily" to connect. The family will then begin the connection process for each of its members. The procedure is useful when you want to have explicit control over the bean connection time.
- Suspend and resume operations. If you are programming an application that consists of a set of panels, you may want to conserve network bandwidth by using families and `ServiceBeans`. If not all panels are active at the same time, you should create a separate family that holds beans belonging to each panel. Then, you can instruct the beans to connect only when a user activates the panel. Moreover, when the panel is minimized, you can use the `sb.suspend()` operation on the `ServiceBean` responsible for the panel's abeans. The method will suspend all monitors and thus increase the responsiveness of the application and the network. When the panel is activated again, you can use `sb.resume()` to return the beans to normal operation. This style of usage is very appropriate for visual composition editors, where you do not explicitly control the creation / destruction of the beans, since these features belong under the control of the builder and are automatically generated. Suspend and resume operations also consume fewer resources than explicit construction and bind operation.

3.8.1 Device queries

Device queries are used in powerful generic applications, where the type and name of a device are determined at run-time and are not hardcoded.

A `ServiceBean` instance is also a gateway to the communication dependent (pluggable) layer. While it does not permit the user to detect the actual communication protocol used and keeps the interface uniform, the user has the possibility to query the pluggable layer for data on all available devices (devices to which the bind can be performed) or all available device types (a device type is defined to be equal to the string value of the device bean, with "Bean" suffix removed). Note that the query operations are completely independent from the families, i.e. the methods are grouped in one class just for the sake of convenience. A simple example will demonstrate the usage.

```
ServiceBean sb = new ServiceBean();
DeviceInfo[] infos = sb.queryAllDevicesSynchronous(new
Type("PowerSupply"), "*");
for (int i = 0; i < infos.length; i++) {
System.out.println("Device name " + infos[i].name + " Connection
type: " + infos[i].connType + " Queried type: " +
infos[i].deviceType + " Actual type: " +
infos[i].implementationType);
}
```

There are several things to note in the preceding code segment:

- A query operation returns an array of `DeviceInfo` structures. All of its fields are printed to the console by the `for` loop.
- The method name `queryAllDevicesSynchronous` reveals that asynchronous version of the same query exists: `queryAllDevices()` will return the same result, but the call will not block. The query will be made in a separate thread and the provided callback argument will be invoked to deliver the results. See the reference manual for details.
- The fields of `DeviceInfo` structure are the following: `name`, `connType`, `deviceType`, `implementationType`. `Name` is a string under which the device can be accessed on the communication layer. Parameter `connType` indicates the name of the connector used to access the device, e.g. "CORBA", "Simulator" etc. Parameter `deviceType` is by definition equal to the type passed as a first argument to `queryAllDevicesSynchronous`. Parameter `implementationType` is the actual (run-time) type of the remote object, which could be either `deviceType` or a name of its subclass.
- The second parameter to the `queryAllDevicesSynchronous()` is a string mask. Mask can be either: a string expression, containing "*" and "?" wild chars, or a domain name. The `ServiceBean` will only return those devices that match the mask. For more information on domains see the reference manual.

Device types are queried by a similar call: `queryAllDeviceTypes()`. The semantics is the same as in `queryAllDevices()`.