Data Access Layer API Proposal

Project:	**DESY-CSS**
Document Owner:	Igor Kriznar
Status:	***DRAFT***
Creation:	2006-02-20 (Igor Kriznar)
Revision:	1.0

Copyright © 2001-2005 by Cosylab d.o.o. All Rights Reserved.

Audience

This document is intended for all that are interested in DataAccess API.

Scope

The purpose of this document is to evaluate different approaches and to specify best solution for DAL API.

Document History

Revision	*Date*	*Author*	*Section*	*Modification*
1.0	2006-02-15	Igor Križnar	All	Created
1.1	2006-04-18	Igor Križnar	2	Included comments by Kay-Uwe Kasemir

References

ID	*Author*	*Reference*	*Revision*	*Date*	*Publisher*
1	Cosylab	Datatypes JavaDoc	3.1		Cosylab
2	Gašper Tkačik	Data Access Layer Feature Requests	-		Cosylab
3	Sun	http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html	-	-	Sun

Table of Contents

1. Narrow and Object-Oriented API

So called Narrow control system API has limited set of different API components through which data are exchanged with remote servers. Usually it has two methods: one for remote Get and other for remote Set. Data is exchanged in form of memory structures. Get and Set methods can be used in very different way by combining different parameters.

On the other side Object-Oriented API can only be wide interface. Such API has many components since each object or method can be used in one way only.

In next sections a view on both approaches will be presented with short code examples and possible advantages or disadvantages of both will be discussed.

Successful examples for Narrow API control system are EPICS, Tango and TINE. And for wide API is ACS with CORBA as communication protocol.

For sake of simplicity examples will be given in EPICS JCA and in generic representation of wide OO API, which will hopefully evolve in common control system wide API.

1.1 API Guidelines

It is safe to assume, that most of programmers who will work with such API are not programming professionals. To average control system programmer it is important that learning curve is smooth at beginning and that it is easy to get going and have something working very soon. Below is list of features that are beneficial to such programming experience:

- simple access (API) to the data;
- programming aids (auto-completion);
- compile time error checking;
- easy debugging;
- attractive GUI widgets;
- common look and feel for applications.

Many of this criteria are most easily met with Object-Oriented API design.

1.1.1 Initialization of communication layer

Before connection can be established some preparation must be made on clients side. All further code samples will assume this initialization has been performed.

JCA

```
JCALibrary jca = JCALibrary.getInstance();
Context ctx = jca.createContext(CAJContext.class.getName());
ctx.initialize();
ctx.attachCurrentThread();
/*
 * Creating a channel. We assume this is double channel.
 */
Channel channel = ctx.createChannel("PS_1:current");
```

Object Oriented example

```
AbstractApplicationContext ctx = new DefaultApplicationContext("Test");
PropertyFactory factory =
                    DefaultPropertyFactoryService.getPropertyFactoryService()
                    .getPropertyFactory(ctx, null);

/*
 * Creates a property from a factory. We assume this is double channel.
 */
DoubleProperty prop = factory.getProperty("PS_1:current",DoubleProperty.class);
```

Both examples create channel (or property) from factory type of provider. In both cases we have assumed that we will deal with double kind of data channel. In JCA example we create generic channel object, there is no need to pass information about double nature of channel, but as seen later we will have to use this information nevertheless, in fact each time we want to read or write something to such channel. In second example we have immediately used this information and requested data channel, which has API specialized for doubles. If programmer will try to use accidentally double channel as for example as long, in JCA there will probably be no problem or problem might appear later (e.g. because of rounding not correct value was set on remote object) but in second case programmer is notified about mistake immediately by compiler. If programmer wants to set long value anyway, it is possible, but it must be done consciously.

1.1.2 Value get and set methods

JCA

```
/*
 * Performing simple get of value.
 */
DBR dbr= channel.get();
double value= ((Double)dbr.getValue()).doubleValue();
ctx.flushIO();
/*
 * Simple set of value. First we check upper limit.
 */
DBR_CTRL_Double ddbr = (DBR_CTRL_Double)dbr;
double max = ddbr.getUpperCtrlLimit().doubleValue();
channel.put(value + ( max – value ) / 2.0);
ctx.flushIO();
/*
 * Make single asynchronous get.
 */
channel.get(new GetListener() {
	public void getCompleted(GetEvent e) {
		if (e.getStatus()!=CAStatus.NORMAL) {
			System.out.println("Get failed!");
		}
		double value= ((Double)e.getDBR().getValue()).doubleValue();
		System.out.println("Value "+value);
	}
});
ctx.flushIO();
/*
 * Make single asynchronous set.
 */
channel.put(value+Math.abs(max-value)/2.0,new PutListener() {
	public void putCompleted(PutEvent e) {
		if (e.getStatus()!=CAStatus.NORMAL) {
			System.out.println("Put failed!");
		}
	}
});
ctx.flushIO();
```

Object Oriented example

```
/*
 * Synchronously gets current value.
 */
double value = prop.getValue();
/*
 * Checks if value can be set and then sets it.
 */
double max = prop.getMaximum();
prop.setValue(value + (max - value) / 2.0);
/*
 * Make asynchronous get.
 */
Request r = prop.getAsynchronous(new ResponseListener() {
	public void responseError(ResponseEvent e) {
		System.out.println("Failed!");
	}
	public void responseReceived(ResponseEvent e) {
		double value = e.getResponse().getNumber().doubleValue();
		System.out.println("Value " + value);
	}
});
// Some time later we can check success.
if (r.hasResponse()) {
	Response res = r.responses().next();
	System.out.println("Success "+res.success()+" value "+res.getValue());
}
/*
 * Make asynchronous set. Response listener we already have.
 */
double max = prop.getMaximum();
r = prop.setAsynchronous(value + (max - value) / 2.0, new ResponseListener() {
	public void responseError(ResponseEvent e) {
		System.out.println("Failed!");
	}
	public void responseReceived(ResponseEvent e) {
		double value = e.getResponse().getNumber().doubleValue();
		System.out.println("Value set " + value);
	}
});
```

The code looks quite similar. Main difference is that we had to make an assumption that we are dealing with double based channel only once in case of ObjectOriented API. Only values of double type were use to set or get remote value. If programmer would try to set some other type of value than double (e.g. accidentally by mistake), then code compiler would complain and thus prevent error. Such object could be easily binded to some double based widget. In case of JCA channel we had to make an assumption of double nature of channel every time we were dealing with channels values.

1.1.3 Special methods

It is quite common that each accelerator facility develops special method for acquiring data from remote data channels. These methods are not commonly used and they are connected with some special ability of underlaying control system or hardware. These special abilities are different for different control system. If we want to cover them with some common API, they all would have to be available in that API. Get of value with time stamp in EPICS JCA is presented below.

JCA

```
/*
 * Make synchronous get with time stamp.
 */
DBR_TIME_Double tdbr= (DBR_TIME_Double)channel.get(DBRType.TIME_DOUBLE, 1000);
value= ((Double)tdbr.getValue()).doubleValue();
TimeStamp timestamp= tdbr.getTimeStamp();
System.out.println("Value "+value+" timestamp "+timestamp);

/*
 * Make asynchronous get with timestamp.
 */
channel.get(DBRType.TIME_DOUBLE, 1000, new GetListener() {
      public void getCompleted(GetEvent e) {
            if (e.getStatus()!=CAStatus.NORMAL) {
                  System.out.println("Get failed!");
            }
            double value= ((Double)e.getDBR().getValue()).doubleValue();
            TimeStamp timestamp= ((DBR_TIME_Double)e.getDBR()).getTimeStamp();
            System.out.println("Value "+value+" timestamp "+timestamp);
      }
});
ctx.flushIO();
```

Object Oriented example

```
/*
 * Special synchronous get with timestamp information.
 */
PointAccess paccess = prop.getDataAccess(PointAccess.class);
Point p=paccess.getValue();
System.out.println("Point " + p.timestamp() + ", " + p.value());

/*
 * Special asynchronous get with timestamp information.
 */
paccess.getAsynchronous(new ResponseListener() {
      public void responseReceived(ResponseEvent event) {
            Point p = (Point)event.getResponse().getValue();
            System.out.println("Point " + p.timestamp() + ", " + p.value());
      }
      public void responseError(ResponseEvent event) {
            System.out.println("Request failed!");
      }
}
```

This special request method was added as additional interface. Once such interface is produced from data channel, it provides type safe way to enter or read data with special request method.

1.2 Device modeling

Device is modeled by some physical device, which is controlled remotely, or an abstraction of several such physical devices. It can be archived in several ways. In Tango and ACS is part of API. In EPICS V3, which is completely flat list of channels, can be achieved only by introduction some kind device proxies on client side, which are defined by naming convention or some additional configuration files (e.g. ControlDesk).

Tango can be described as Device-Oriented. Device in Tango provide semi-narrow interface to data channels (or attributes in Tango) within context of device. Data

channels (attributes) in Tango are similar to channels in JCA.

Devices of same type produce same data channels. This is useful for displaying them in table (e.g. one row for each device) or generic higher level applications.

Object-Oriented example could access and use devices very similar to Tango. We assume generic power supply device with command “on” and double data channel “current”.

```
/*
 * Default initialization.
 */
AbstractApplicationContext ctx = new DefaultApplicationContext(
          "Test");
DeviceFactory factory = DefaultDeviceFactoryService.getDeviceFactoryService()
       .getDeviceFactory(ctx, null);

/*
 * Connect to device.
 */

AbstractDevice device= factory.getDevice("PS_1");

/*
 * List available properties and commands.
 */
DynamicValueProperty[] properties= device.toPropertyArray();
Command[] commands= device.getCommands();

/*
 * Switch device on.
 */
Command on= device.getCommand("on");
on.execute();

/*
 * Get current device property.
 */
DoubleProperty current= (DoubleProperty)device.get("current");
double value= current.getValue();
```

If power supply device is declared as interface like this:

```
/*
 * Simple PowerSupply definition.
 */
static interface PowerSupply extends AbstractDevice {
       public void on() throws RemoteException;
       public void off() throws RemoteException;
       public void reset() throws RemoteException;
       public DoubleProperty getCurrent();
       public DoubleProperty getReadback();
       public PatternProperty getStatus();
}
```

than device will be used as before plus with some type save convenience:

```
      /*
       * Do similar thing, only with defined device type.
       */
      PowerSupply ps= factory.getDevice("PS_1",PowerSupply.class);

      /*
       * Switch device on.
       */
      ps.on();

      /*
       * Get current device property.
       */
      value= ps.getCurrent().getValue();
```

2. ObjectOriented API Issues

Design of ObjectOriented API must take into account that generic applications need access to remote object in a way that exposes remote capabilities without requiring generic application to know device or data channel specific interface. OO API must provide layer "narrow" style level with introspection capabilities. Introspection would have to be supported somehow in communication layer: as integral part (e.g. V4 hopefully) or as separate service. This level is used by generic applications and expert programmers. On top is wide interface with additional features: code completion, compile time error checking and intuitive interfaces.

Implementation of generic or “narrow” style layer is possible as common API to all specific interfaces. All different kinds of specific data channels and devices are extended from one common device and channel interface, which provide generic device or channel access.

2.1 Generic Data Channel API

Data channel (simply Channel in JCA or Property in our examples) can provide generic access. Regardless the data type (double, long, string or something else), value from such property must be accessible in unique way so can be used by generic applications.

2.1.1 Value access with Java Generics

Such design can be achieved with help of new Java language feature generics (see reference 3). Code sample that demonstrates this design follows.

```
public interface DynamicValueProperty<T> {
      /* Generic access to values. */
      public void setValue(T value) throws DataExchangeException;
      public T getValue() throws DataExchangeException;
      public void addDynamicValueListener(DynamicValueListener l);
      public void removeDynamicValueListener(DynamicValueListener l);
}

public interface DoubleProperty extends DynamicValueProperty<Double> {
}
```

Interface DynamicValueProperty is defined with type parameter T. Interface DoubleProperty is specialization of this property for double values. If T parameter is not specified, Java automatically assumes that T is Object. As consequence if we use

DynamicValueProperty without declared T parameter, then Get and Set values will operate with Objects. DoubleProperty for instance is declared with T parameter as Double. Same methods for Get and Set will now operate with instances of Double objects. This is demonstrated in example below.

```
DynamicValueProperty<?> prop= factory.getProperty("PS_1:current");
Object value = prop.getValue();
DoubleProperty prop = (DoubleProperty)prop;
double devalue= prop.getValue();
```

We create new property from factory without explicit type parameter. If Get method is used the Object is returned. If explicitly cast this property to DoubleProperty, because it is declared as such, then we can use Get method with double. Java executes the same method in both cases and generics feature makes possible that value is returned in correct type.

2.1.2 Value access by explicit type declaration

If an application has reference to DynamicValueProperty object and wants to access value explicitly with certain type there are two possibilities. Simple solution is to check if return Object is of certain type each time and cast it. Second is to use DynamicValueProperty interface to cast values instead. Example of this possibility is presented below.

```
DynamicValueProperty prop= factory.getProperty("PS_1:current");
Object value = prop.getValue();

StringAccess da = prop.getDataAccess(StringAccess.class);
String value = da.getValue();
da.addDynamicValueListener(new DynamicValueAdapter<String>(){
      @Override
      public void valueUpdated(DynamicValueEvent<String> event) {
            System.out.println("New value as string "+event.getValue());
      }
});
```

From every DynamicValueProperty it is possible to extract simple interface DataAccess, which make best effort to provide access to value in alternative type or format. Implementation of such interface, which converts double value to String is trivial. Advantage is that such approach opens possibilities for nontrivial conversions, such as transformation from one units to other or to different coordinate system.

Below is example of simplified declaration of DataAccess interface and how this would affect DynamicValueProperty declaration:

```
public interface DataAccess<T> {
      /* Generic access to values. */
      public void setValue(T value) throws DataExchangeException;
      public T getValue() throws DataExchangeException;
      public void addDynamicValueListener(DynamicValueListener l);
      public void removeDynamicValueListener(DynamicValueListener l);
}

public interface StringAccess extends DataAcess<String> {
}

public interface DynamicValueProperty<T> extends DataAccess<T>{
      /* Returns native or default data access for this property. */
      public DataAccess<T> getDefaultDataAccess();
      /* Returns particular data access. */
      public <D extends DataAccess> D getDataAccess(Class<D> type) throws
                                                          IllegalViewException;
      /* Returns supported access types. */
      public Class<? extends DataAccess>[] getAccessTypes();
}
```

2.1.3 Property introspection

Generic application can use all DynamicValueProperty features only if property can describe by itself which features can provide. At this moment only features, which are not fixed are characteristics of property: name-value pairs of (in EPICS these are properties of Channel). They tend not to change often and they describe values for configuration values: display limits, control limits, units and such. Below is redefinition of DynamicValueProperty interface, which enables dealing with characteristics in a generic way.

```
public interface DataAccess<T> {
      /* Generic access to values. */
      public void setValue(T value) throws DataExchangeException;
      public T getValue() throws DataExchangeException;
      public void addDynamicValueListener(DynamicValueListener l);
      public void removeDynamicValueListener(DynamicValueListener l);
      /* Access to characteristics. */
      // returns all available names
      public String[] getCharacteristicNames() throws DataExchangeException;
      // retrieves single characteristic
      public Object getCharacteristic(String name) throws DataExchangeException;
      // retrieves multiple characteristics
      public Map getCharacteristics(String[] names) throws DataExchangeException;
}
```

2.2 Generic Device API

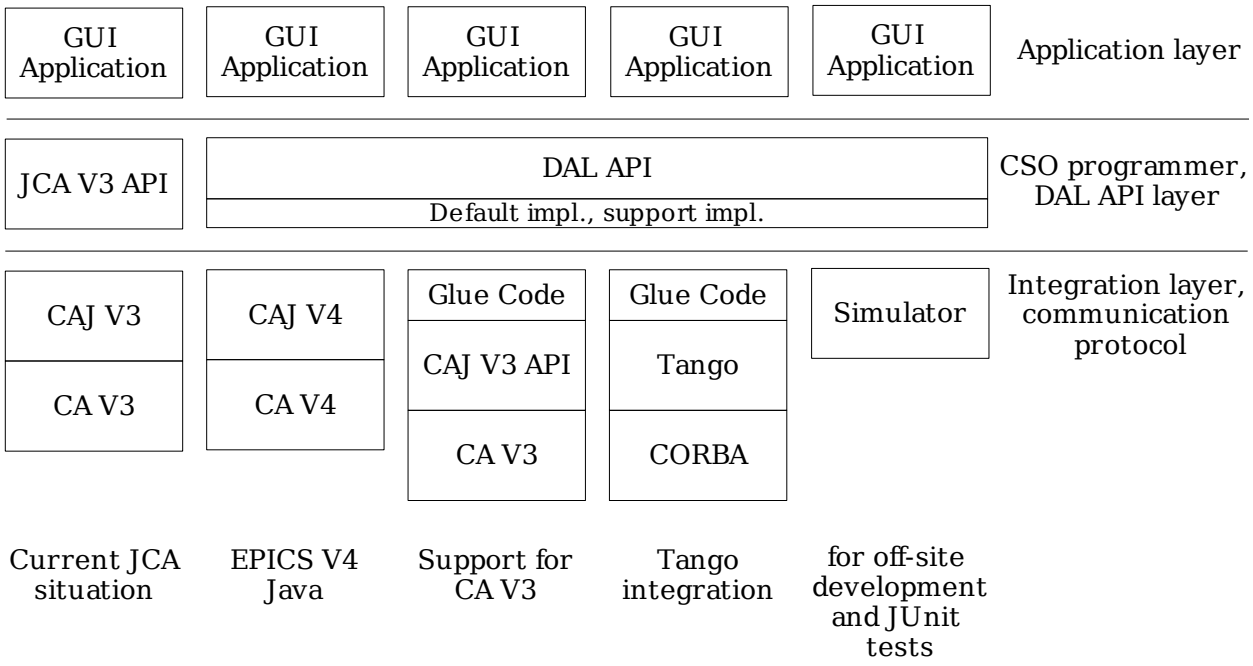
Every specific device (such as PowerSupply already mentioned) has fixed interface which provide type safe and intuitive access to device features. If all device interfaces are extended from same generic device interface, than same features can be accessible in generic way. Device on generic level can be used by generic applications. Key requirement for generic device is usage of “narrow” style access methods and possibility of introspection. There are three groups of features, which requires generic approach:

- properties (or EPICS Channels) that belong to a device;
- characteristics, additional configuration named values describing device (position, description, type);
- commands.

Redefinition of PowerSupply considering this possibilities can be found below.

```
/*
 * Generic device definition.
 */
interface AbstractDevice {
      /* Generic access to properties. */
      // introspection: returns names of all properties
      public String[] getPropertyNames();
      // introspection: returns true if property is contained
      public boolean containsProperty(String name);
      // introspection: returns all properties
      public DynamicValueProperty[] toPropertyArray();
      // narrow access: returns by name requested property
      public DynamicValueProperty getProperty(String name);

      /* Generic access to characteristics. */
      // introspection: returns all available names
      public String[] getCharacteristicNames() throws DataExchangeException;
      // retrieves single characteristic
      public Object getCharacteristic(String name) throws DataExchangeException;
      // retrieves multiple characteristics
      public Map getCharacteristics(String[] names) throws DataExchangeException;

      /* Generic access to commands. */
      // introspection: returns all commands
      public Command[] getCommands();
      // introspection: returns all command names
      public String[] getCommandNames();
      // return particular command by name
      public Command getCommand(String name);
}

/*
 * Since commands are mentioned as objects, we can as well define such interface.
 */
public interface Command {
      public String name();
      public Class[] getParameterTypes();
      public Class getReturnedType();
      public Object execute(Object... parameters) throws RemoteException;
}

/*
 * Simple PowerSupply definition.
 */
static interface PowerSupply extends AbstractDevice {
      public void on() throws RemoteException;
      public void off() throws RemoteException;
      public void reset() throws RemoteException;
      public DoubleProperty getCurrent();
      public DoubleProperty getReadback();
      public PatternProperty getStatus();
}
```

3. Common API implementation



This drawing is result of conversation with Matthias about different levels in vertical view. Current JCA implementation has two levels which divide JCA and CAJ. Similar situation would arise when we make implementation for EPICS V4. We can assume CAJ interface would not change much from V3 to V4.