TRANSPLANTING THE SUCCESS OF ECLIPSE TO CONTROL SYSTEMS

G. Tkačik[1], M. Pleško[1], M. Clausen[2]
[1]*Cosylab d.o.o., Ljubljana, Slovenia* [2]*DESY, Hamburg, Germany*

ABSTRACT

There are two main drawbacks to developing client control system applications independently: from the user's standpoint, the resulting applications miss visual and behavioural consistency and do not support data exchange between applications; from programmer's standpoint, independent development amounts to code replication. Attempts to define a client-side framework can fail because the resulting code is bulky and contains too many dependencies to be easily understandable. In our previous work we have demonstrated the basis for data exchange using meta-data descriptions; here we focus on how Eclipse helps us solve the second, namely the "monolithic framework" problem. Eclipse, an open source Java IDE, has proven to be a huge success, containing code contributions from both small private and major industrial partners. Recently, Eclipse has been split into a Rich Client Platform (RCP) and a set of IDE modules running on top of it. RCP itself forms the "generic Workbench" that includes component framework based on OSGi specification, application services (logging, resources, configuration) and complex GUI abstractions (wizards, preference pages). Everybody is free to take Eclipse RCP and develop applications on top of it, with some big physics labs already having done that (NASA, LANL). We show how using OSGi – the Eclipse component model – one can develop component sets that the framework glues together to form applications. Different visualization components integrate into the workbench as "actions" that can be triggered on the relevant "data sources" (e.g. chart for numeric channels, command execution for devices, etc). New components, both visual and programmatic, can be added by simply putting their respective jars in an "Office" directory, and the framework takes care of loading, updating from the web, dependency resolution, authentication checks and so on. We think that Eclipse, as it did for the IDE world, also brings into control system domain the long-awaited ability to evolve applications as a growing set of components developed by different labs.

INTRODUCTION

In the field of control systems the development of software often has not followed the top-down approach, where the user experience is the driving force behind programming. On the contrary, pushing data from acquisition layer up onto the client machines, frequently in a relatively unprocessed form, seems to be the norm. Consequently the effort expended on studying and implementing "office-like" tasks and gestures (copy & paste, drag & drop, file operations, undo, browsing, common data visualizations, forward and backward navigation; see [1] for functionality breakdown) is wasted, because 3rd party products cannot integrate easily with non-annotated [6] data streaming in from an external source. As a result features are left unimplemented or are added later at a high cost through in-house development.

As part of a project called Control System Office (or CSO; see [4] for the overview) we decided to examine available application frameworks (see [2] for evaluation results) and have selected Eclipse as the platform that both defines the OSGi compliant [3] component architecture, as well as provides complex GUI components, such as property tables, wizards, perspectives and views. The project is currently in the design phase, and is hosted on [5]. Its purpose is to define Eclipse plug-ins that adapt the Eclipse Rich-Client Platform environment for development and execution of control system applications.

This paper discusses the crucial component of the project, the *Data Access Layer* or *DAL*. In its final implementation, DAL will be wrapped as Eclipse plug-in that will bridge the gap between Eclipse framework and a number of specific control system architectures, such as EPICS Channel Access, TINE, TANGO, ACS and possibly others. The primary goal of DAL is to deliver the data from the control system in a form that is easily processed by generic applications and components [7] as well as used in manual programming.

ARCHITECTURAL REQUIREMENTS FOR DATA ACCESS LAYER

The form of DAL is an OSGi conformant API, containing core and pluggable code.
Core DAL, including a Request / Response based interface which we call reactor (see Figure 1), is the code shared between all control systems; glue or pluggable code is the functionality that has to be implemented separately for each system in order to support shared functionality. Plug code sets are different and separate implementations of the same DAL interface set. Distributionwise, this will mean that DAL comes as a core plug-in and a set of dynamically loadable glue-code components that extend the core plug-in. Multiple plug implementations must concurrently execute within the Eclipse environment, if necessary.

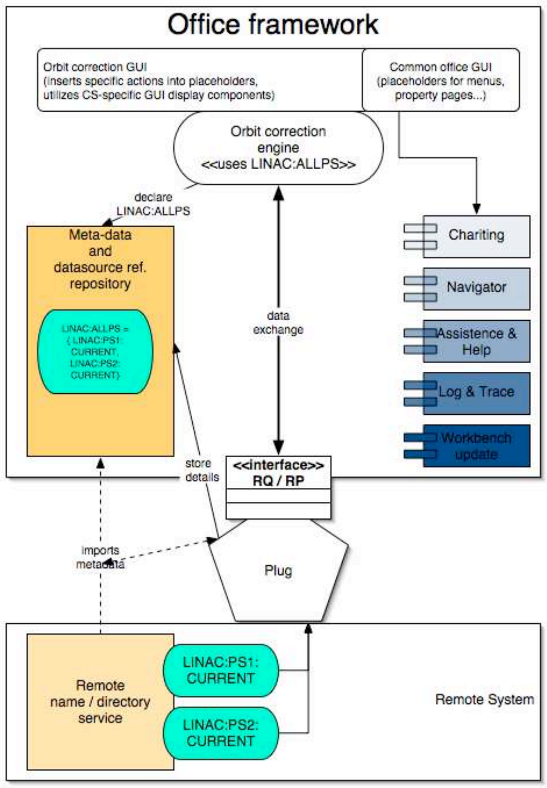
Office framework
Orbit correction GUI (inserts specific actions into placeholders, utilizes CS-specific GUI display components)
Common office GUI (placeholders for menus, property pages...)
Orbit correction engine <<uses LINAC:ALLPS>>
declare LINAC:ALLPS
Meta-data and datasource ref. repository
LINAC:ALLPS = {LINAC:PS1:CURRENT, LINAC:PS2:CURRENT}
data exchange
Charting
Navigator
Assistence & Help
Log & Trace
Workbench update
store details
<<interface>> RQ / RP
imports metadata
Plug
Remote name / directory service
LINAC:PS1: CURRENT
LINAC:PS2: CURRENT
Remote System

Figure 1: A broad outline of the components running within Eclipse framework (upper rectangle): DAL is represented by a pentagon (plug) with Request (RQ) / Response (RP) interface serving as a gateway between the remote system (lower rectangle) and the orbit correction application that uses it.

DAL will address the following functionality:
Direct data exchange. Given a string name that has a meaning in the underlying control system, DAL will be able to fetch or set the related value bound to that name (with all traditional semantics, such as single get/set, monitor etc).
Name resolution. If the underlying system supports name queries and hierarchical namespace as defined by JNDI (Java Naming and Directory Interface), DAL will enable the access to the name service, including querying for the existence of the name, for the type of entity bound to the name, and obtaining a list of entities at certain hierarchical level.
Meta-data interface. Name service can be extended into a directory service by providing a set of attributable data for each named node. In case of control systems and as discussed in the Whitepaper [1], the relevant attributable information is the meta-data description of the entity bound by a certain name. Such information may include, but is not limited to, the notion of type, supported requests and responses for a given entity, type of viewer components to be used for interaction with the entity, semantic description of the provided data (i.e. its changeability, typical duration, composition if it is a structured type and so on). Meta-data directory is also the mechanism for defining virtual data sources [see 12 for definitions]. DAL does not specify the meta-data interface, although its design must enable the support for it, to be added as a separate component later on. Our intention is to design such component as the next step in CSO project.

Design foundations

DAL builds on top of several existing architectures. Firstly, as a requirement, it must provide interface similar to the Java Channel Access in order to ease transition for the programmers. Secondly, it follows the precedent of Channel Access for Java [8] and cleanly separates the reactor (RQ/RP) code from the Channel-like interface sitting on top of it (as will be discussed in the next subsection). Thirdly, it draws ideas from Abeans Datatypes and metadata directory to define data objects and metadata exchanged with the control system, which should enable it to talk to both channel-based (CA, TINE) and device-based (ACS, TANGO) control systems. Lastly, it captures the intuition of TINE that packed data access [see 13 for definition] is an important ingredient of a successful control system.

Different views of the control system

The control system field has witnessed a long-standing debate between wide (e.g. device based) and narrow (e.g. channel based) control systems [9]. In addition to its being a matter of taste, it is also a matter of convenience: while wide interface stresses the *modelling aspect* of the system – how complex devices that have one state are broken down into a number of actions and properties – the narrow interface emphasizes the *operational aspect*, in which most often used actions are carried out on sets of properties that span multiple devices. Let us redefine the narrow interface as the request / response interface, a connectionless data exchange mechanism with flat addressing (i.e. no hierarchy is implied in the modelling objects, such as a device object containing properties; or a channel object containing fields). A request with a named target in the remote layer is submitted into the system, and it generates one or more asynchronous responses as a result. No connection is established beforehand; and no connection needs to be explicitly broken. [10]

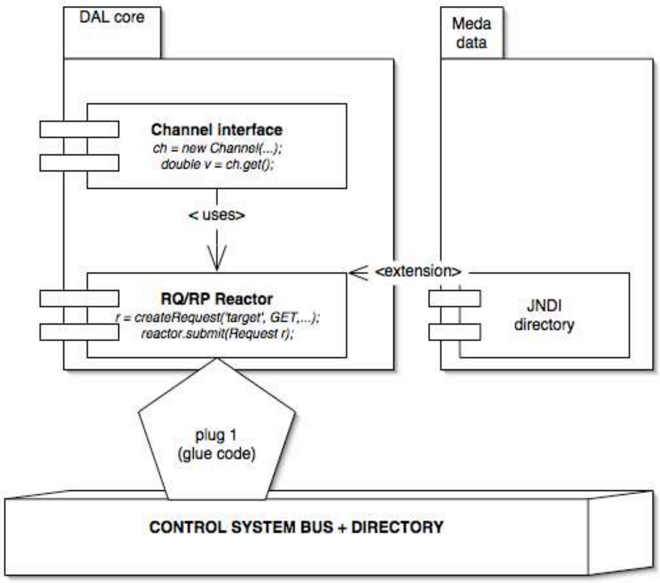

Figure 2: Two-layered structure of DAL. Both channel and reactor interfaces are exposed to other components, one of them being the metadata directory depicted here. The code snippets are illustrations of how each layer can be used.

Clearly any object-oriented interface can be implemented as a veneer on top of such narrow interface: for example, the familiar *Channel* classes produce appropriate requests and unpack the resulting responses, thus exposing a higher level view of the system to the channel user.
However, there is a variety of cases (especially when independently produced components talk to each other), where the low level interface might be preferable: scripting requests from jython or terminal-like component within Office, serializing requests (e.g. transmitting them through the Web by XML encoding), submitting batches of requests at once, using generic applications (e.g. Object Explorer, applications that store state snapshots etc), exchanging data by drag and drop and so on.
Furthermore, an efficient implementation will usually employ some sort of reactor scheme internally, even if it does not expose it. What is being proposed here as an important point in our opinion is to make a formal split between low level reactor (RQ/RP) component and the model (channel, which depends on reactor), and expose **both** interfaces to other Eclipse components, as depicted in Figure 2, so that each one can be utilized as needed for different purposes [1].

DATA ACCESS LAYER FEATURES

The design of DAL requires us to examine the following four aspects. Firstly, specifying *direct data exchange* involves describing what kind of requests and what kind of responses the reactor will process, and how these will map into object-oriented channel interface; in addition, we need to describe the *datatypes*, i.e. objects that are passed as request input arguments and response return values. Next, specifying *name resolution and metadata* requires elucidation of how the remote data sources can be uniquely named; how the implicit hierarchical structure in these names can be explicitly declared in the directory, and how the directory obtains its data from the control system name service through the plug (see Figure 2). Lastly, as DAL is a part of Eclipse infrastructure, we need to state where and which extension points it defines and how they relate to the components that we expect in the suite.

In next subsections we present some features that we found important and a subject of general interest, and we propose a corresponding solution for each; an in-depth analysis is provided in [11].

Data Exchange

1. Optionally stateless connection at Reactor level, explicit connection at Channel level
When submitting requests directly into the reactor, the connection to the remote data source (e.g. channel) need not be established first by a special 'connection' request – it is up to the plug to manage a transparent connect and disconnect if no requests are pending after a period of inactivity. However, an asynchronous connection request is defined: if it is submitted, a response notification must be generated by the plug, the remote resource cannot be released until the explicit disconnection is requested and each failed link must be reported (if detected) as a response to the connection request. Channel class is stateful and issues connection and disconnection requests when its appropriate methods (CONSTRUCTOR, CONNECT, DESTROY) are invoked.
Rationale: Allows both easy interactive scripting and coding; allows execution of blocks of serialized requests without explicitly submitting connection requests; allows transfer of already connected data sources between components in drag & drop and similar operations, because what gets transferred is simply a data source URI with descriptor contained in metadata directory [1], and the drop target can immediately and seamlessly execute requests on it; allows execution of requests on virtual data sources that are composed of atomic data sources that have not been previously connected.
2. Channel is composed of dynamic value (value field), a set of characteristics, and associated accessor / mutator operations of all synchronicities; Reactor is always asynchronous.
Dynamic value has one of the types described in *Datatypes* subsection; each datatype defines a set of obligatory characteristics (such as minimum and maximum for floating datatype); in addition each instance of a channel can have supplementary characteristics. Accessor and mutator operations support both synchronous / asynchronous gets and sets, as well as monitors and (possibly) local history access. All requests and responses between channel model and reactor (and downwards) are asynchronous.
Rationale: If channel model does blocking on reactor asynchronous calls, then user has guaranteed order of execution if s/he executes synchronous methods on channel from within the same thread and such ordering is required; having all reactor calls be asynchronous evades the common pitfalls of blocking on failed connections and allows highly efficient operations at bursts of activity (initialization, shutdown).
3. Reactor supports a small set of requests as a design contract. A request can be issued towards any target that has a well-formed name.
Such requests may include GET, SET, MONITOR, CONNECT, LOOKUP, for instance. If the target is atomic [12] (such as a the channel value) GET, SET and MONITOR have natural interpretations. If the target is a hierarchical component (such as "LINAC" or "LINAC:PS1"), the requests are processed on each element recursively and a compound data type result is generated (see next subsection). CONNECT request has already been mentioned in 1. LOOKUP request resolves the directory structure of the given name, returning the constituents of the name and the corresponding metadata descriptors.
Rationale: Makes the form of requests simple and extensible, while allowing the plug a lot of freedom in how to actually implement these requests. We will see that this feature (specifically the recursive resolution when request is targeted to a packed entity), together with the directory and compound data types, allows systems that support optimized transfer modes natively to use them to the full extent.

Datatypes

1. There is a small number of basic types, two compound types, and a fixed and predefined set of infrastructure types.
Basic types (such as a floating point type, integral type, bit pattern, string, enum) that differ not only in the storage size but also in the set of associated characteristics are separately defined. Infrastructure types (timestamps, error stack, alarm and completion codes) have a fixed form that can accommodate requirements of all the underlying systems. Compound types are arrays and maps of name-value pairs, which can recursively nest. This accommodates structures (that correspond to maps) as well as trees (that are one possible result of issuing a request towards a non-atomic target, such as all entries in directory node "MACHINE1:LINAC").

Rationale: Support for both manual programming using channel interface, generic applications (e.g. fetching all characteristics of X) and efficient coding of packed data transfers [12].

2. Java 1.5 generics can be used to support casting a type to alternative renderings and to use pre-allocated objects; no multiplication of code for different datatypes.

This is an important feature, because it allows inference of the return type from the passed argument and the reuse of existing compound structures (maps) in an efficient way. Also allows the user to define structures and tuples as Java classes and have their member fields filled by the DAL introspectively when they are supplied as a parameter.

Rationale: It is possible to design a value accessor method on the channel with signature `T get(Class<T> type)` or `void get(T instance)` that infers `T` or declares the accessor template method and reuses it without cumbersome type casts. For example, sometimes it is possible to fetch compound data (e.g. by doing GET on a list of channels) into either a map (where keys are IDs of the channels) or array (where ordering is implied), and both 'views' of the same data can be distinguished by `T`.

Universal naming and the directory

1. If metadata directory is present, alternative hierarchies and named virtual data sources (groups, aliases) can be defined; these can be targets for reactor requests.

Virtual data sources can be hierarchical names that contain all their child nodes. More commonly the virtual data source is defined when, say, the user makes a selection of certain channels in the navigator; or when the application at start-up predefines a set of all BPMs in the storage ring as a new virtual data source. Virtual data source can also be an alias (alternative name) or a group of nodes that match some search criteria.

Rationale: Allows treating systems where integration of data is supported natively and systems where it is not, on the same footing (i.e. in one case the data source is virtual and in the other it is real, but DAL machinery remains the same); the requests can be optimized in the plug for systems without native packed support (for example because it is known in advance that the plug has to execute N read requests, one for each BPM in the virtual data source, it can create and flush them in one attempt).

2. Support for a canonical and native names with arbitrary number of hierarchical levels.

Each native name [13] is a valid target for a reactor request. All such names also have a canonical URI rendering that is used to store the data source description in the metadata directory or when serializing requests. In addition, URI names also describe other entities that do not exist natively: non-atomic targets ("LINAC") on which requests can be recursively applied; virtual data sources (such as groups of channels) that are defined in the directory; subparts from which a channel is constructed ("LINAC:PS1_CUR:MINIMUM") and so on.

Rationale: Arbitrary number of levels allows us to cover various systems; canonical rendering makes each target uniquely identifiable in a consistent way that can be parsed by standard Java URI tools; the reactor interface makes no distinction between something that is an atomic data source or a virtual one.

Extensions

1. Reactor defines extension points for plug implementations.
Each plug implements reactor interfaces that perform the actual request processing and communication with the underlying layer.

2. Reactor defines extension points for the interceptors.

Before the received request is submitted to the remote layer, any registered interceptors can examine or modify it (*Chain-of-Command*). This allows requests to be logged, serialized, authenticated, benchmarked and even modified. The interceptor notification procedure is executed when a response is received as well.

3. Metadata directory is an extension that plugs in as an interceptor of the reactor.

If metadata directory is present and as long as the plug keeps the connection to the remote source alive, the URI of the remote source is maintained in the directory along with its descriptor object (see [1]). Whenever the user right-clicks on the data sink that represents the data source (for instance the gauge displayer component), or examines it in the navigator tree, or drags it to another sink etc, the directory can be used to look up the structure of the manipulated object. Generic applications can use descriptors to dynamically invoke requests on the object.

4. Additional request sets are extensions for extension points defined by DAL.
To support functionality that is too specific to be made portable and uniform across different underlying control systems, an extension plug-in can supply a set of request definitions (tags) and their implementations for a specific plug. The application programmer can then use the reactor RQ/RP interface to issue such special requests, which will get routed to the extension for processing, without requiring modifications to the reactor DAL architecture.

CONCLUSION

We believe that Eclipse plug-in system provides the necessary incentive to cleanly split the code that is responsible for communication with the control system into a low-level reactor scheme with its glue-code plugs, the channel model, and the metadata directory. The reactor can run as a stand-alone component; and both reactor and channel model can be used without metadata directory. Using DAL with channel model is similar to current JCA; using all three blocks together opens up the possibility of having compound datatypes returned by a single 'channel'; but with the directory, the 'channel' can now be a real channel or a set of real channels, either of the same type or various types – and no interfaces exposed to application programmers change when such virtual data sources and compound types are introduced. The organization of the data (e.g. whether the naming hierarchy reflects the geographical location or arrangement by device type or functional group and so on) can be formalized in a directory, which can accommodate multiple hierarchies, and consequently – as an example – reading either all properties of "LINAC:PS1" or all CURRENT properties of the form "LINAC:PS*:CURRENT" both turn out to be one-request operations on a specific virtual data source.
By structuring DAL in the way described here we hope to create a consistent and application-oriented foundation that is both small and flexible enough so that the specific features of each separate system do not need to be discarded for the sake of generality.

REFERENCES

[1] Control System Office Whitepaper, http://users.cosylab.com/~kzagar/cso/SPE-Control_System_Office_Whitepaper.html
[2] Control System Office Framework comparison, http://users.cosylab.com/~kzagar/cso/SPE-CSO_Framework_Evaluation.html
[3] OSGi Specification, www.osgi.org
[4] M. Clausen, G. Tkacik: *EPICS Office*, ICALEPCS 2005
[5] DESY Control System Office webpage, http://epics-office.desy.de
[6] In this context, 'non-annotated' data is data for which no meta-information is available in machine parseable form (see G. Tkacik et al, *A Reflection on Introspection*, ICALEPCS 2003); in the absence of such data, any binding between data sink components (displayers, components that integrate data from data sources etc) and data sources has to be manually coded along with manual transfer of the context (for example graphing limits, units etc) from the data source to the displayer.
[7] G. Tkacik et al, *Beating Commercial Products: Control System Office and IDE are the Way to Go*, PCAPAC 2005, Hayama, Japan
[8] Channel Access for Java: http://caj.cosylab.com
[9] P. Duval et al, *The Babilonyzation of Control Systems,* ICALEPCS 2003
[10] There are different possible realizations of this concept in the pluggable code, which we do not discuss here; the implementation freedom that the plug developer has can enable him/her to share connections and manage them through reference-counting.
[11] Discussion of feature requests for DAL, in preparation for EPICS meeting at ICALEPCS 2005
[12] A virtual data source is a data source that does not exist by itself in the underlying system. In this context, an atomic data source is the opposite of a virtual data source: atomic data source is an entity known by its native name also in the underlying system. For example, "LINAC:PS1", although it can be part of the name for "LINAC:PS1:CURRENT", is not itself an atomic data source. However, with the directory describing that "LINAC:PS1" is composed of "CURRENT" and "STATUS", "LINAC:PS1" itself can become a *virtual* data source: a read request invoked on it will fetch both "STATUS" and "CURRENT". Similarly, a completely new data source "Y" could be defined to contain all channels with names that match "LINAC:PS*:CURRENT", which would yield a channel that returns a compound type (double array) of all currents of linac power supplies. TINE supports this operation natively.
[13] Packed access is any atomic and efficient access to items that are accessible as independent data sources by themselves. For example, getting the value of all BPM position readings with one remote call, or getting a packed monitor in which one event lists all readback values, are packed accesses, because reading a single BPM position and doing a single readback monitor are also supported atomically.
[14] A native name is a name that the underlying control system uses for its corresponding atomic data source.