University of Newcastle upon Tyne

submitted under Erasmus exchange to the

Technische Universität München (TUM)

Individual Project as part of the

BSc Degree in Computing Science

Author : Hernan Raffi

Supervisors : Prof. Brüggemann-Klein (TUM)
Dr. Gianluca Chiozzi (European Southern Observatory)

Date of Submission: 10 November 2003

University of Newcastle upon Tyne

“Implementing a configuration database browser”

by

Hernan Raffi

10 November 2003

Supervised by: Prof. Brüggemann-Klein (TUM)
Dr. Gianluca Chiozzi (ESO)

Acknowledgments

I would like to thank the following people for having supported my project.

Prof. Brüggemann-Klein (Technische Universität München) as my supervisor, for having given me the opportunity to develop this project.

Dr. Gianluca Chiozzi (European Southern Observatory) for having provided the requirements and for supervision of the software that I developed.

Table of Contents

ACKNOWLEDGMENTS - 1 -

TABLE OF CONTENTS - 2 -

ACRONYMUS - 3 -

1. INTRODUCTION - 4 -

1.1 THE ATACAMA LARGE MILLIMETER ARRAY (ALMA) PROJECT - 4 -
1.2 INTRODUCTION TO THE ALMA ADVANCED COMMON SOFTWARE (ACS) - 4 -
1.3 PROJECT DELIVERABLE: THE CONFIGURATION DATABASE BROWSER - 6 -
1.4 CHALLENGE OF THIS PROJECT AND EXPECTED OUTCOMES - 7 -

2. THE CONFIGURATION DATABASE (CDB) - 8 -

2.1 INTRODUCTION TO CDB - 8 -
2.2 CDB ARCHITECTURE - 9 -
2.2.1 The Database Access Layer (DAL) - 10 -
2.3 XML CONFIGURATION FILES - 11 -

3. THE CDB BROWSER - 13 -

3.1 THE CDB BROWSER - 13 -
3.2 REQUIREMENTS - 14 -
3.3 USE CASES - 16 -
3.3.1 Browsing through the CDB Tree - 16 -
3.3.2 Editing and resetting the XML String - 18 -
3.3.3 Saving the XML String - 19 -
3.3.4 Editing and resetting the CDB attribute values inside a table. - 20 -
3.4 PROGRAM DESIGN - 21 -
3.4.1 Model View Controller (MVC) - 21 -
3.4.2 How to set up the Tabbed Pane - 23 -

4. CONCLUSION - 25 -

5. REFERENCES - 26 -

6. ATTACHMENTS - 27 -

6.1 USERS MANUAL - 27 -
6.2 DOCUMENTATION - 27 -
6.3 SOFTWARE CODE - 27 -

ACRONYMUS

ESO	European Southern Observatory
ALMA	Atacama Large Millimeter Array
ACS	Advanced Common Software
CDB	Configuration Database
DAL	Data Access Layer
jDAL	Java DAL
DAO	Data Access Objects
DO	Distributed Object
XML	Extensible Markup Language
XSD	XML Schema Definition
CORBA	Common Object Request Broker Architecture

1. INTRODUCTION

1.1 The Atacama Large Millimeter Array (ALMA) Project

The Atacama Large Millimeter Array (ALMA) is a project among astronomical organisations in Europe and North America. ALMA will consist of at least 64 radio antennas, each twelve meters in diameter, operating in the millimeter and sub-millimiter range. The antennas will be located in the Chilean Atacama desert at an altitude above 5000 m and have a total collecting area of 7000 sqm (Ref. [10]).

1.2 Introduction to the ALMA Advanced Common Software (ACS)

The ALMA Common Software (ACS) is an object oriented framework that provides a software infrastructure common to all partners involved in the development of the ALMA software (Ref. [11], [12]). ACS is available under the public licence scheme LGPL[1].
The heart of ACS is an object model based on Distributed Objects[2] (DO's); each Distributed Object is implemented as a CORBA (Ref. [16]) object. DO's are used as basis for the development of control systems for components and devices such as an antenna mount control.

An important module in ACS is the configuration database (CDB), which addresses the problem related to accessing and maintaining the DO configurations of the ALMA system (see Chapter 2). In particular the CDB stores configuration parameters for the DO's and is read, at run-time, when the DO's are started up or reinitialized.

Figure 1 shows the structure of the ACS Packages.
These have been grouped in four layers, each allowed to use services provided by other packages on the lower layer and on the same layer (Ref [1], [7], [9]). The layers are:

1. Base Tools: these are distributed as part of ACS to provide a uniform development and run time environment on top of the operating system for

[1] Lesser GNU Public Licence.
[2] See Chapter 2.1 for more details on Distributed Objects.

all higher layers and applications. These packages provide installation and distribution support.

2. Core Packages: these ensure standard interface patterns and implement essential services. Among these is the Configuration Database.
3. Services: these implement higher level services such as the Management and Access Control Interface (MACI) used to supervise the state of the system.
4. High level APIs and tools: these offer a clear path for the implementation of applications with the goal of obtaining implicit conformity to design standards.

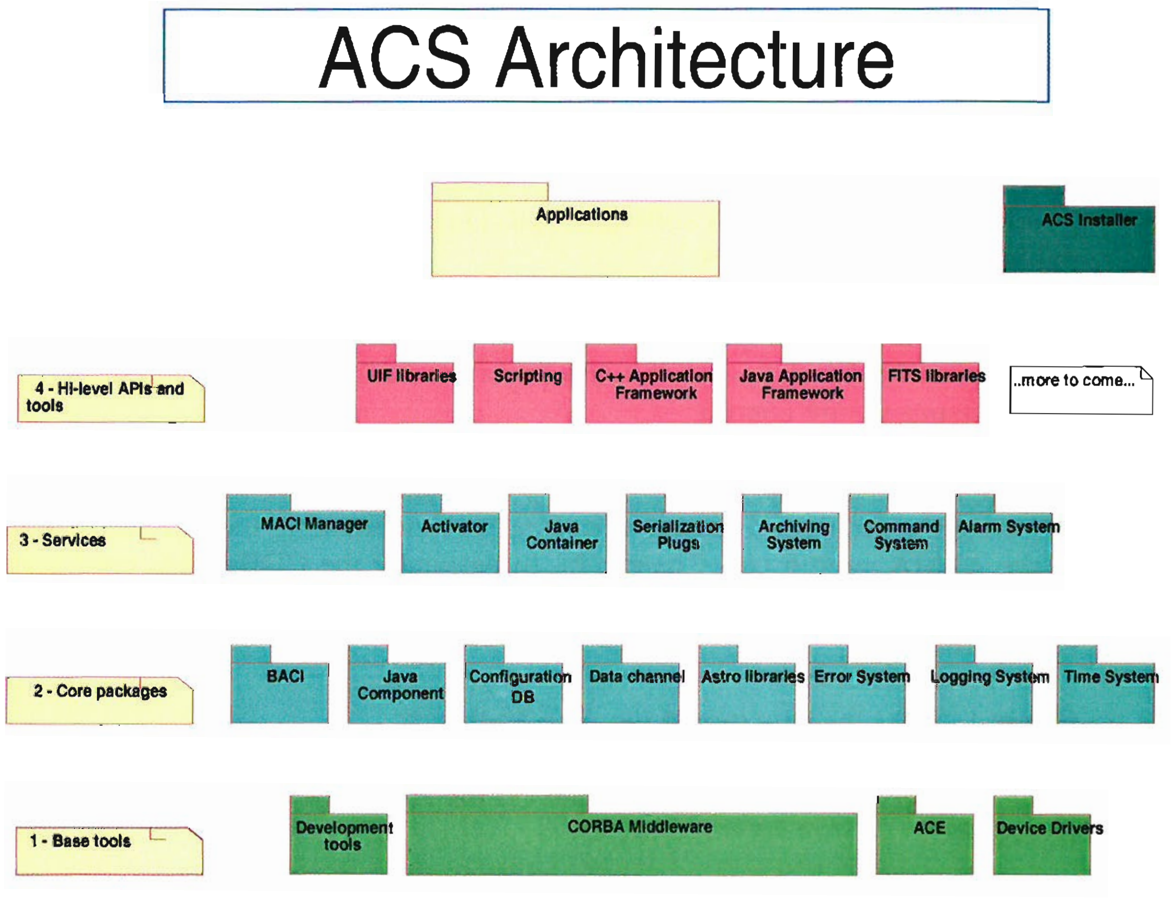
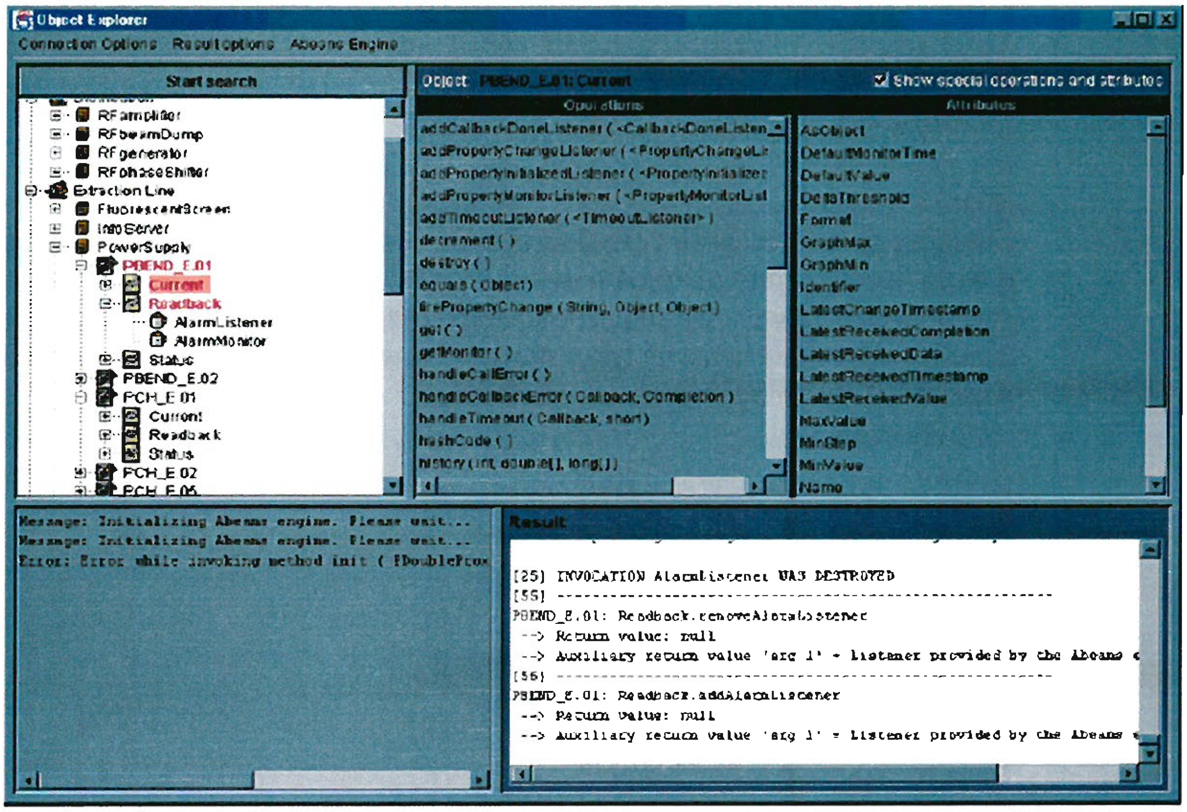


Figure 1: ACS Architecture.

1.3 Project deliverable: The Configuration Database Browser

The purpose of this project was to implement a browser using a Java GUI to visualize the structure of the Configuration database (CDB). The browser enables developers to edit attribute values in the CDB and switch between different views. More precisely the browser shows an 'XML view' of the configuration data exactly as it is stored inside the CDB and a 'Table view' of the same data, presented in an organized table.
As model for the Browsers GUI the existing Object explorer (as shown in Figure 2, Ref [4]) has been used.



```
Message: Initializing Abeans engine. Please wait...
Message: Initializing Abeans engine. Please wait...
Error: Error while invoking method init ( IDoubleProx
```

```
[25] INVOCATION AlarmListener WAS DESTROYED
[55] ------------------------------------------------------------
PBEND_E.01: Readback.removeAlarmListener
 --> Return value: null
 --> Auxiliary return value 'arg 1' = listener provided by the Abeans c
[56] ------------------------------------------------------------
PBEND_E.01: Readback.addAlarmListener
 --> Return value: null
 --> Auxiliary return value 'arg 1' = listener provided by the Abeans c
```

Figure 2: The Object Explorer.

1.4 Challenge of this project and expected outcomes

The main challenge of this project was to be able to develop a useable and complete software package. This has become part of the software framework ACS, which is in regular use by the whole ALMA Computing team consisting of about 50 professional software developers. The CDB Browser is already integrated in the latest ACS Release (3.0).

By implementing this project knowledge in the following areas was acquired:

- Java Swing library (Ref. [14]).
- Eclipse development environment (Ref. [6]).
- XML and XML Schema, as DO's configuration are serialized with XML structures (Ref. [3], [5]).
- Elements of CORBA (Ref. [16]).
- Integration aspects with ACS (now ACS 3.0).
- Documentation in the ACS style, by using doxygen (Ref. [15]). A doc comment is written with JavaDoc syntax, with enbadded HTML. It must precede a class, field, method or a constructor. It is made up of two parts, a description followed by block tags like: '@param', and '@return'. The documentation is retrieved from the source code and ordered into a Web Page.

2. The Configuration Database (CDB)

2.1 Introduction to CDB

ACS is a distributed system, where some parts are installed and run on different platforms simultaneously. It is complex to duplicate the same data on each platform, and also the maintenance and updating would become very hard. Furthermore ALMA is developed in different timescales and places; therefore for testing and developing ACS there will be a multitude of small CDB fragments. It is the developer choice to choose which instance of the CDB to use.

The Configuration Database (CDB) is a module in the ACS system, used to store configuration parameters, i.e. the initial values for **properties** control values and all **characteristics** for properties of ALMA devices (Ref [8]). These devices are also known as Distributed Objects (DOs). Distributed Objects are implemented as CORBA objects that are accessible from any computer in the system; examples of DOs are the antenna mount, antenna control unit, temperature sensor and motor.

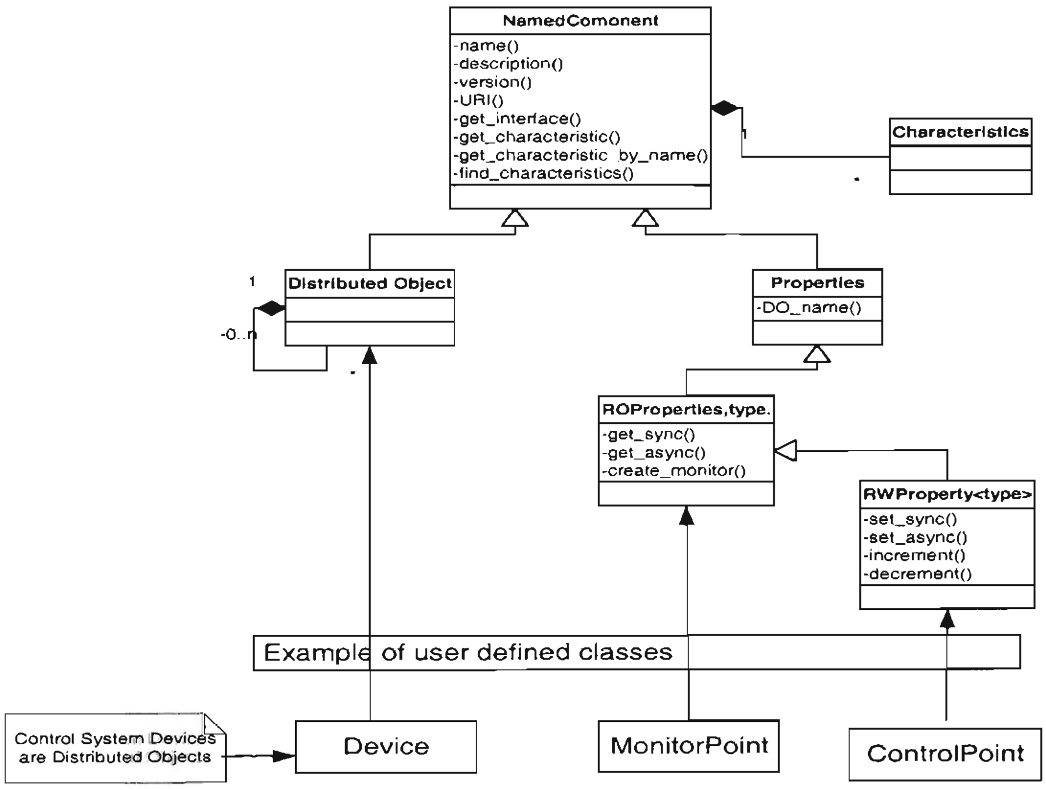


Figure 3: DO Property-Characteristics

As Figure 3 shows each Distributed Object is the base class for any physical/logical device and it can contain reference to other DOs to build hierarchical structures of components. Furthermore each DO is composed of *Properties* such as value, position-control and monitor points. Both DOs and Properties have specific *Characteristics* such as range, unit and default values.
The common behaviour of DO and Property has been factorized in the *Named Component* common base class (see Figure 3). The methods of any Named Component allow retrieval of these Characteristics (Ref [3]).

The configuration data for these DOs consists of a set of text files (XML files[3]) which are retrieved at run-time during start up and used to initialize and configure the DOs.

The CDB system must be modular as many different modules are dependent upon the CDB, and adding one of these modules (i.e. my CDB Browser) should only increase its usability without impacting other existing modules.
The CDB must also provide a clean interface as its functionality will be accessed from different platforms and programming languages; any application can make use of the CDB to get access to configuration information.

2.2 CDB Architecture

The configuration parameters for all DOs are persistently stored in the Configuration Database. There are four issues related to the problem addressed by the CDB:

1. Input of data by the user: Easy and intuitive data entry methods are needed.
2. Storage of data.
3. Maintenance and management of the data: the configuration data changes with time and has to be maintained under control.
4. Loading the data into the ACS Activators: at run time the data has to be retrieved and used to initialize the DOs.

The main objective of the CDB Architecture is to keep these four issues as decoupled as possible.
The high-level architecture is based on three layers (Ref [5], Figure 4):

[3] See Chapter 2.3 for more details on XML configuration files.

1. The Database itself: the database engine used to store and retrieve configuration data. It may consist of XML files in a hierarchical file system or it may be a relational database.
2. The Database access layer (DAL): the DAL is used to hide the actual database implementation from applications. In this way it is possible to use the same interfaces to access different database engines.
3. The Database client that stores and retrieves data using the DAL.

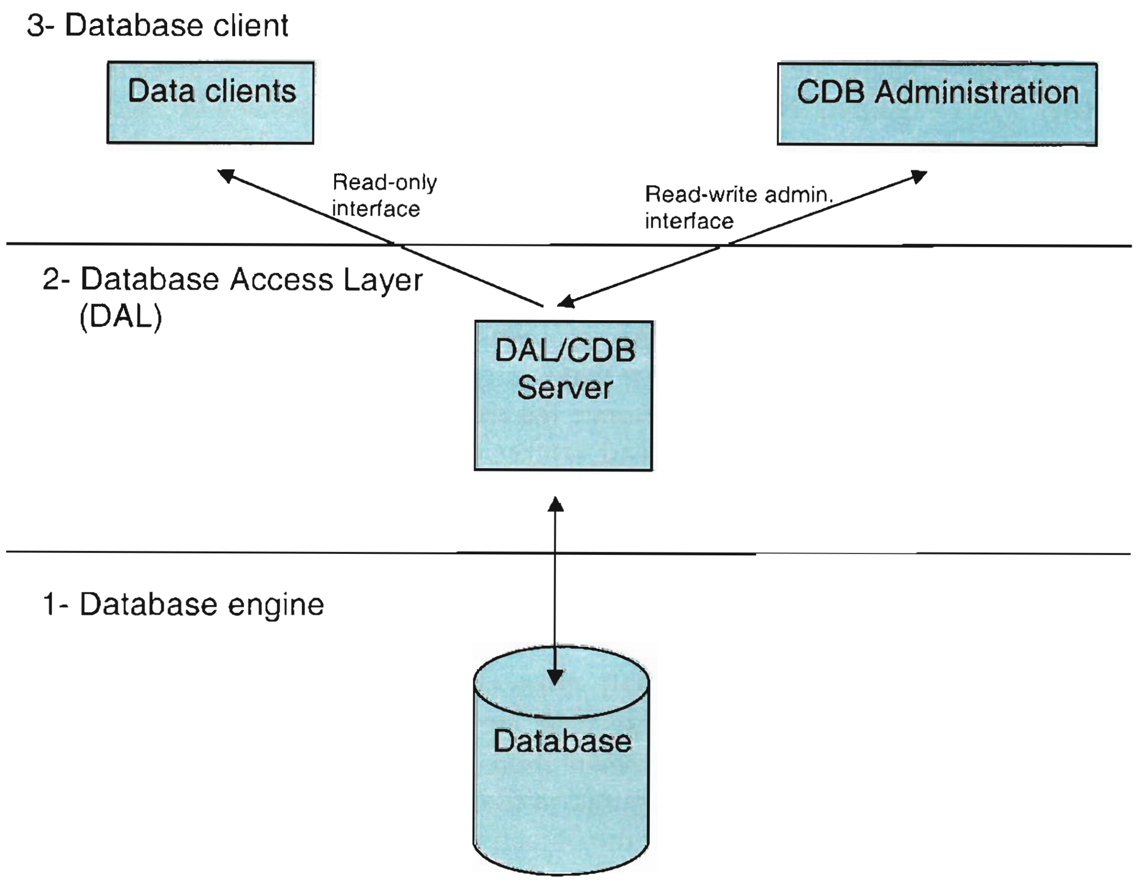


Figure 4: Three-tier database architecture.

2.2.1 The Database Access Layer (DAL)

In the DAL architecture the data is accessed through Data Access Objects (DAO); each record in the configuration database is represented to the client as a DAO. When a request is sent to the DAL server, this first looks into its cache of DAOs.

If the requested DAO is found the request is passed to it and the DAO returns the result. If the DAO is not found the DAL server will first create the DAO.
To clarify the concepts of DAL and DAO lets look at how the CDB Browser retrieves configuration data from the CDB.
The Browser uses a DAL server implemented as a JAVA CORBA server; namely the jDAL. When the CDB is started the jDAL scans for all schema (XSD[4]) files and adds them as external schema location files. The reason for this is the fact that jDAL ensures that all data in the system will be checked against their schema.
The jDAL also initializes the XML parse factory, and configures the XML parser to validate each XML file against its corresponding XML schema. If the parser does not understand the schema language an error message is issued and XML entries will be ignored. Now that the jDAL is running requests can be made to the server. When a request is sent to the jDAL server this first checks to see if the requested XML entry exists. Should this not be the case an exception is raised (RecordDoesNotExist). If the XML file is found, the parsing starts against the corresponding schema. In case a parsing error occurs jDAL throws an XML error exception so that the client can see what went wrong.
If the parsing was successful, jDAL provides an interface to get access to the corresponding DAO. There are three ways to get access to a DAO (see Reference 5). In the case of the CDB Browser once the record in the CDB is found, the DAL creates the interface to the DAO and returns the complete data serialized as an XML string[5].

2.3 XML Configuration Files

As mentioned in Chapter 1.2 the CDB stores configuration parameters for exactly one Distributed Object (DO). These data are organized in XML files under a specific directory in the jDAL implementation. There exists a one-to-one mapping between the path of the XML configuration file and the name of the DOs (ref. [5]). For example, configuration data for the object:

```
/ALMA/Antenna1/Motor3
```

can be found in the file:

```
$SOMEWHERE/ALMA/Antenna1/Motor3/Motor3.xml
```

Each directory representing an object includes an XML file with the same name as the object (e.g. Motor3) containing the data for the object itself and sub-directories for hierarchical sub-objects.

[4] XSD: XML Schema Definition.
[5] To view an XML string see chapter 2.3.

This is how a Database Configuration file for an example DO called LAMP looks like:

```
<?xml version='1.0' encoding='UTF-8'?>
<LAMP id="0">
  <brightness id="1"
      description="brightness"
      units="_"
      min_value="-1.7976931348623157E+308"
      max_value="1.7976931348623157E+308"
      default_timer_trig="10000000"
      min_timer_trig="10000"
      min_delta_trig="0"
      default_value="0.0"
      graph_min="-1.7976931348623157E+308"
      graph_max="1.7976931348623157E+308"
      min_step="0.0"
      archive_delta="0"
      format="%9.4f"
      resolution="65535"
      archive_priority="3"
      archive_min_int="0"
      archive_max_int="0">
  </brightness>
</LAMP>
```

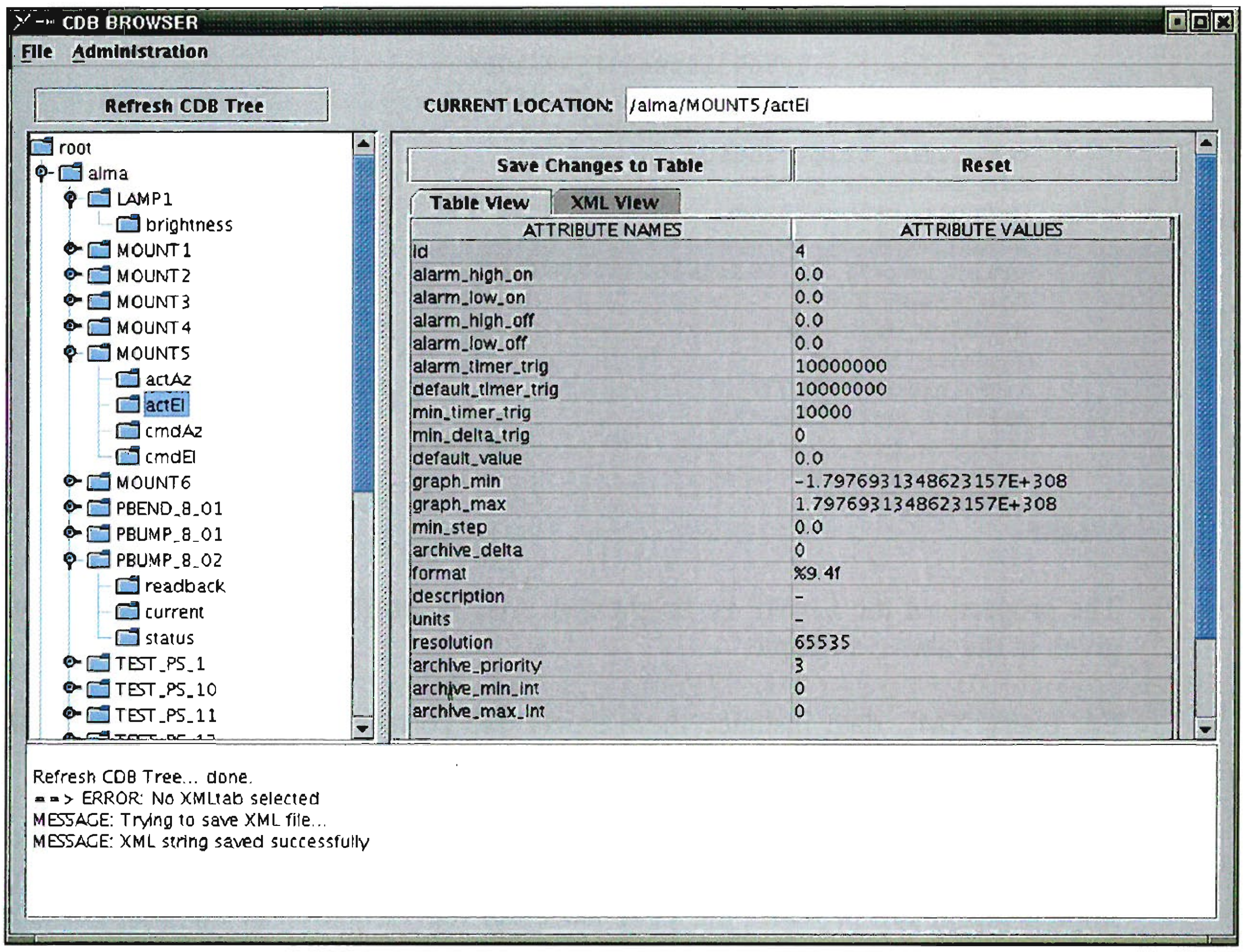
The property of the LAMP is: brightness. Also its attribute names and values are given in the above example.

For every XML data instance there exists one corresponding XML schema file used for validation (Ref [3]).

The physical implementation of a DAO can be different that a XML file, for example it could be stored in a Reletional database. In this case the DAL server implemention still provides the same exsternal representation of the configuration of a DO, also if the physical implementation of the configuration data is changed.

3. The CDB Browser

3.1 The CDB Browser



ATTRIBUTE NAMES	ATTRIBUTE VALUES
id	4
alarm_high_on	0.0
alarm_low_on	0.0
alarm_high_off	0.0
alarm_low_off	0.0
alarm_timer_trig	10000000
default_timer_trig	10000000
min_timer_trig	10000
min_delta_trig	0
default_value	0.0
graph_min	-1.7976931348623157E+308
graph_max	1.7976931348623157E+308
min_step	0.0
archive_delta	0
format	%9.4f
description	-
units	-
resolution	65535
archive_priority	3
archive_min_int	0
archive_max_int	0

```
Refresh CDB Tree... done.
==> ERROR: No XMLtab selected
MESSAGE: Trying to save XML file...
MESSAGE: XML string saved successfully
```

Figure 5: The Browsers GUI.

The CDB Browser GUI is divided into three sections: the CDB tree (left side), the tabbed pane window (right side) and the message text area (bottom), with an additional button (*Refresh CDB Tree*) and display field on top (*location bar*).

3.2 Requirements

Functional Requirements

- The CDB must be accessed via DAL CORBA interface.
- The browser must show the object oriented database structure and more in particular its tree like structure.
- The browser must show the plain XML configuration files as they are stored inside the CDB, using queries towards the Database Access Layer.
- The configuration data inside the XML files must be extracted and made available into an editable table with the following two columns: Attribute Names and Attribute Values. Only the Attribute values can be edited.
- After changes to the table are made a validation must follow, whereby the XML string is updated and validated towards its corresponding XML Schema. Finally the user must be informed if the validation was completed successfully.
- The user must not only be able to save changes to the table, but also be allowed to reset its values to its original status.
- It must be possible to save the XML configuration strings into user defined files. Saving is possible only if all changes to the table have been validated.
- Pop-up dialogs are used to warn the user about actions that are not allowed in a given context, for example if the user selects a node in the CDB tree without having saved the changes made to a previous node.
- The Browser must provide a *'Reset Tree'* button to re-initialise its CDB tree, without the need to shut-down and restart.

Performance Requirements

- There must not be any noticeable delay in displaying CDB data.

GUI look and feel

- The browser panels must have a look and feel consistent with the existing Object Explorer (See Fig. 2 in Chapter 1.3).
- There must be three sections inside the browser window. The left panel must display the database tree structure (same as in the Object Explorer). The right panel must display either the XML string or the table of attributes names and values extracted from the configuration file (a tabbed pane should be used for this). At the button of the GUI there must be a logging and warning window.

- Provide a Reset CDB Tree button (located above display of tree structure).
- Display the selected node path inside a location bar located at the very top of the GUI (similar to address bar in Internet Explorer or Netscape).
- When table view is active Save Changes and Reset buttons should be visible.

Standards to be used

- Implement Browser in JAVA
- Use existing cdb libraries to access the CDB DAL via CORBA.

Design constraints

In order to run the CDB Browser the services provided by the Database Access Layer (DAL) must be activated (Ref. [2]). The command for starting the DAL is:

```
$ cdbjDAL
```

After DAL activation (terminated with message: JDAL is ready and waiting...) the command to start the CDB Browser can be given (normally in a different window):

```
$ cdbBrowser
```

3.3 Use Cases

The CDB Browser requirements can be illustrated by the following USE Cases. The Use Case layout used below is the one utilized by the ACS Use Cases (Ref. [13]).

3.3.1 Browsing through the CDB Tree

Priority: Critical

Performance: Respond to users input in near real time.

Frequency of use: expected to be used every time the CDB Browser is started.

Preconditions: jDAL must be active (so that requests to the DAL server are possible).

Basic Course:

Note: all steps on this course are created to help the development and testing of the CDB Browser.

There are three events on the CDB Tree that the user can activate, namely the 'tree expanded' event, the 'tree collapsed' event and the 'value changed' event.
What follows is a step by step description of what happens when the user browses the CDB Tree and activates one of the three events.

The *tree expanded* event:

1. If the expanded node is an instance of the 'CDBTreeNode' class, than a request is sent to the jDAL server to check if the requested XML entry exists.
2. If the record is not found a 'RecordDoesNotExist' exception is raised.
3. If the record is found the JDAL provides an interface to get access to the corresponding DAO, which returns the requested XML string and the linked hash map. Now the tabbed pane is created (see Chapter: 3.4.2 for more details).
4. The created tabbed pane is ONLY displayed (on the right side of the browser) if no changes from the previous visible tabbed pane are left unsaved (which means that the two buttons on top of the tabbed pane are disabled).

5. If the created tabbed pane is displayed the location bar gets updated with the path of the just expanded node.

The *tree collapsed* event:

The event is called whenever an item in the tree collapsed (children are not shown any longer). The program updates the displayed tree and otherwise ignores the event.

The *value changed* event:

1. First check that the previous tabbed pane (if one was visible) is saved. If this is not the case the user is not allowed to select a new node (a warning pop-up dialog becomes visible).
2. If no changes are left unsaved the four objects representing the current context[6] are updated with the components of the tabbed pane that correspond to the selected node. Finally the location bar gets updated with the path of the selected node.

[6] See Chapter 3.4.2 for more details on how to build a tabbed pane.

3.3.2 Editing and resetting the XML String

Priority: Critical.

Performance: Respond to users input in near real time.

Frequency of use: Perform this Use Case every time the user edits the XML string or presses the *Reset* button.

Preconditions: The user selects the 'XML View' inside the Tabbed pane. The two buttons on top of the table are disabled.

Basic Course:

Note: All steps on this course are created to help the development and testing of the CDB Browser.

1. As soon as the user edits the XML string the program stores (in a temporary string) the value of the XML string before editing occurred.
2. The two buttons ('Save Changes' and 'Reset') get enabled.
3. Now the user is not allowed to select other nodes, as well as to reset the CDB tree (by pressing the '*Reset CDB Tree*' button), save the XML string (Menu -> Save XML String), or close the Browser.
4. The user has the choice to save the changes or resetting the changes to their initial value.
    - Save Changes: see Use Case 'Saving the XML string'.
    - Reset XML string: no interaction with the jDAl is needed. The temporary string that stored the XML string as it was before editing started (see point 1) is copied to the XML tab. (method: selectedXMLArea.setText(oldXML)).
5. The two buttons on top of the table get disabled.

3.3.3 Saving the XML String

Priority: Critical.

Performance: Respond to users input in near real time.

Frequency of use: Perform this Use Case every time the user presses the '*Save Changes*' button.

Preconditions: The user has edited the XML string and the two buttons on top of the tabbed pane are enabled.

Basic Course:

Note: All steps on this course are created to help the development and testing of the CDB Browser.

1. If the user presses the 'save changes' button the edited XML string is sent to the DAL server using ACS function that takes as parameter the XML string and validates it against the corresponding XML schema.
   - If validation fails an error message will be returned and the XML string will be reset (see Use Case: Editing and resetting the XML string).
   - If validation succeeds, the CDB tree has to be fully refreshed and the buttons on top of the tabbed pane get disabled.

Issues to be determined or resolved:

At this point it is not clear what kind of error message will be returned if validation fails. This requires an ACS function that takes as parameter an XML string and validates it against the corresponding XML schema. As this function is under implementation by the ACS team, the validation functionality is not implemented yet.

3.3.4 Editing and resetting the CDB attribute values inside a table.

Priority: Critical.

Performance: Respond to users input in near real time.

Frequency of use: Perform this Use Case every time table editing occurs.

Preconditions: The user selects the 'Table View' inside the Tabbed pane. The two buttons on top of the table are not enabled.

Basic Course:

Note: All steps on this course are created to help the development and testing of the CDB Browser.

What follows is a step by step description of what happens when the user starts editing a row inside a table.

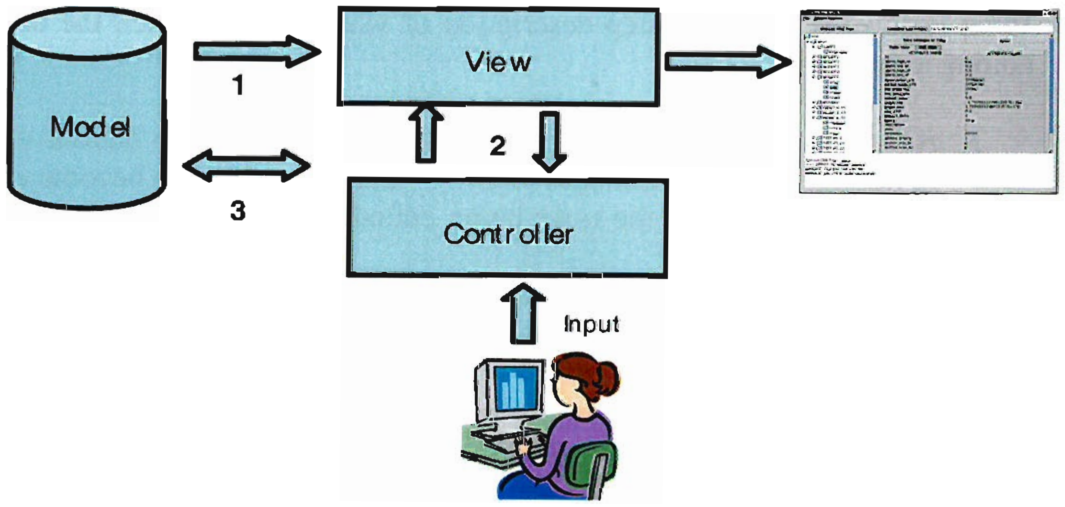
1. As soon as the user starts to edit data in the table the two buttons on top of it, namely the 'Save Changes to Table' and 'Reset' button become enabled.
2. All initial values of the rows being edited are stored (if the user wants to reset the values).
3. The program also stores the number of all rows being edited so that data in these rows can be displayed in red colour to remind the user which rows are changed.
4. At this point the user is not allowed to do any other action than editing more rows, saving the changes or resetting the table.
5. Let us consider the case when the reset button is pressed: at this point all the values saved in point 2 are set in the table. Finally the two buttons on top of the table become disabled and the data that was displayed in red (see point 3) turns to its initial black colour.
6. If the user presses the Save button the changes inside the table are copied into the corresponding XML string. Now the XML string is sent for validation (see Use Case 3.3.3). If some changes to the table cannot be copied to the corresponding XML string, than an error message is printed in the message text area at the bottom of the Browser.

3.4 Program Design

3.4.1 Model View Controller (MVC)

Model View Controller is a design pattern that is especially suitable for GUI programming; it separates application data (contained in the *Model*) from the graphical presentation components (the *view*) and input-processing logic (the *controller*).

- **MODEL**: The Configuration Database (CDB).
- **VIEW**: Graphical representation of the model (tree and tabbed pane).
- **CONTROLLER**: The mouse click button (tree selection or extension).



1. Model data gets exposed.
2. View passes events to the Controller.
3. Controller notifies Model (XML Record is returned if appropriate).

Figure 6: The Model View Controller.

<u>The Model:</u>

In the case of the CDB Browser the *Model* is the Configuration database that contains XML files that the application is reading (see Chapter 2.3). Anytime the

Model is changed (by editing its data) the *View* of the *Model* is notified so that it can change the visual representation of the *Model* on the GUI.

<u>The View:</u>

The *View* implements the visual display of the *model* through a tree and its corresponding tabbed pane.
In the case of the CDB Browser the *View* is implemented in the class `Browser`.
The class is responsible for initialising and placing all components inside the GUI.

<u>The Controller:</u>

The *Controller* receives all input events from the user and translates them into possible changes to the *View*. In the case of the CDB Browser the tabbed pane at the right side of the GUI will change.
In the CDB Browser the *Controller* is implemented in the class `CDBLogic`.
The class handles all the input events from the user and translates them into possible changes on the GUI.

There are two events that the *Controller* can receive: The Tree Expansion Event, which occurs when the user expands the tree and the Tree Selection Event (see Use Case 3.3.2), which occurs whenever the value of the selection changes.

The advantages of having *Model* and *View* separated are:

- The *View* (the Browsers GUI) uses multiple *Models*, as there exists multiple instances of the Configuration Database that have to run on the same *View*.
- The *Model* (or configuration data) needs multiples *Views* as there are many applications that visualize the structure of the same CDB instance in d
- ifferent context.
- Another important advantage that can be deduced from the two above is easier software maintenance; the *Model* and the *View* implementation can be modified separately. The *View* can be changed without changing the *Model*, and vice-versa.

3.4.2 How to set up the Tabbed Pane

It is important to understand a few points about the logic that is behind the tabbed panes visible on the right side of the Browser (see figure below).

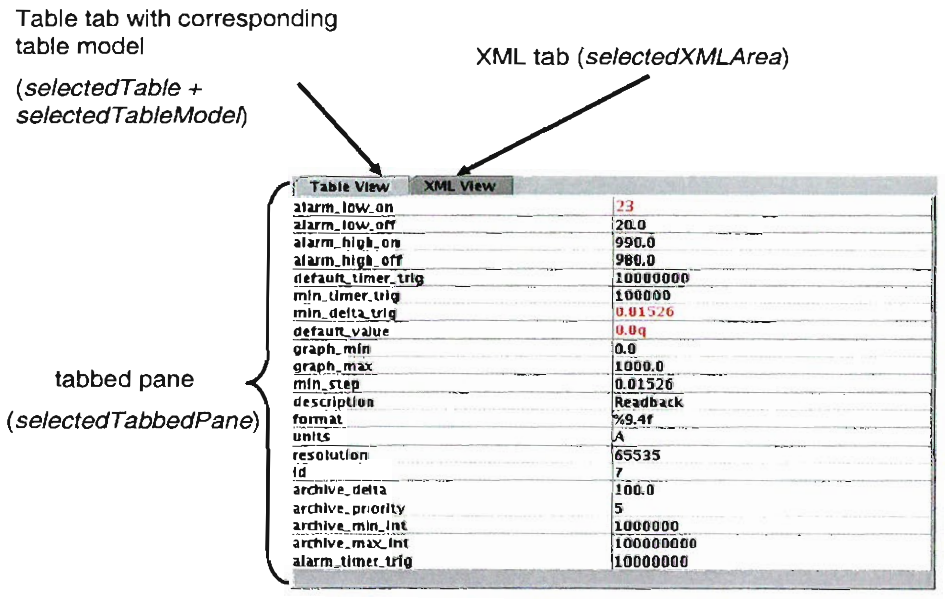


Table tab with corresponding table model
(*selectedTable* + *selectedTableModel*)
XML tab (*selectedXMLArea*)
tabbed pane
(*selectedTabbedPane*)
Table View
XML View

alarm_low_on	23
alarm_low_off	20.0
alarm_high_on	990.0
alarm_high_off	980.0
default_timer_trig	10000000
min_timer_trig	100000
min_delta_trig	0.01526
default_value	0.0q
graph_min	0.0
graph_max	1000.0
min_step	0.01526
description	Readback
format	%9.4f
units	A
resolution	65535
id	7
archive_delta	100.0
archive_priority	5
archive_min_int	1000000
archive_max_int	100000000
alarm_timer_trig	10000000

Figure 7: Components of the Tabbed Pane.

Each tabbed pane consists of at most three components (see figure 7); the *tabbed pane* itself, the *XML tab* (can be null) and the *table tab* (can be null). For the table tab component a CDBTableModel was developed. For each table an instance of the CDBTableModel is used.

Once the user selects a node in the CDB tree and the DAL server successfully returns its record, a tabbed pane (as shown in Fig. 7) is created and made visible on the right side of the browser. Should the user later on reselect the same node the previously created tabbed pane is re-used and made visible, no query is sent to the DAL server this time and no tabbed pane is created. For this reason each component (plus table model) is stored into hash maps; components will be retrieved and set visible when the user reselects the same node.

To simplify the switching between the tabbed panes when the user changes selection the program executes its operations using four objects, namely:

1. the 'selectedTabbedPane' object for the active (visible) tabbed pane .

2. the 'selectedXMLArea' object for the active (visible) XML tab.
3. the 'selectedTable' object for the active (visible) table tab.
4. the 'selectedTableModel' object for the active table model.

These objects have to be updated every time the user changes selection in the tree to set the current context. As these objects are used by the entire program it is not necessary to change the context in every class, but it is enough to update the four objects once.

Precondition: The request to the jDAL interface was successful, and the corresponding DAO returns:

1. The corresponding XML String (can be null),
2. The attribute names and values stored into a Linked Hash Map (which can be null).

What follows is a step by step description on how the tabbed pane is created once the DAO returns the XML string and/or its attribute names and values.

1. The first step is to create a 'JTabbedPane' instance.
2. Now with help of the linked hash map (see precondition) the program tries to create an instance of 'CDBTable' and a 'CDBTableModel'. There are two cases to consider:
   - If the linked hash map contains the attribute names and values of the selected node, then instances of 'CDBTable' and 'CDBTableModel' are created and stored. The created table is added to the tabbed pane (the *'Table View'*).
   - If the linked hash map is empty, than an empty JTextArea is created and stored instead of a CDBTable object. The text area is added to the tabbed pane (*'Table View'*), but is not activated, which means that the user cannot select the 'Table View'. No CDBTableModel object is created; instead a null value will be stored.
3. Now with help of the XML string the XML tab has to be created. Again there are two cases to consider:
   - If the XML string is successfully returned a JTextArea is created containing the XML string. The JTextArea is added to the tabbed pane and stored.
   - If the MXL string is empty an empty JTextArea is created and stored. The JTextArea is also added to the tabbed pane, but it is not activated.
4. At this point the tabbed pane contains two tabs and it is stored into the hash map.

4. Conclusion

In order to implement the CDB Browser I first had to study the ALMA ACS software and in particular the Configuration Database. This was a necessary condition; to fully understand the environment in which I had to develop and integrate the CDB Browser.
Second I had to examine the existing Object Explorer as a model for my Browser, as the CDB Bowser was required to have the same look and feel as the Object Explorer.

During implementation of the CDB Browser I had to deal with CORBA and XML. Mostly I improved my knowledge of the Java Swing library. Important was also to learn how to write professional documentation and user's manual for a specific product.
Interesting for me was also to see how professional software developers work and co-operate with each other to produce a very complex package like ACS.

5. References

[1] Chiozzi G., *CORBA-based Common Software for the ALMA project.* ICALEPC 2001, San Jose 2001.
http://www.eso.org/%7Egchiozzi/AlmaAcs/OtherDocs/spie2002.pdf

[2] Chiozzi G., Sekoranja M. (2002), *ALMA Common Software Overview.* ALMA documentation.

[3] Vitas D. Zagar K. (2002), CDB Tutorial. *Configuration DataBase CDB.* ALMA documentation.

[4] Kadunc M., Tkacik K. (2002), *Object Explorer*, ALMA User's Manual.

[5] Chiozzi G. (2002), *ACS Configuration Database*, ALMA documentation.

[6] Sommer H. (2003), *ACS Java Component Programming Tutorial*, ALMA programmer's manual.

[7] Chiozzi G. (2003) Gustaffson B, Jeram B, *ALMA Common Software Architecture*, ALMA documentation.

[8] Chiozzi G., Fugate D., Gustafsson B., Jeram B., Lopez B., Sivera P., Zamparelli M. (2002), *ALMA Common Software Training-Course. Session 1c.* ALMA training

[9] Chiozzi G., and others, *Common Software for the ALMA project*, ALMA documentation.

ACS Web pages:

[10] The ALMA Web Page: http://www.eso.org/projects/alma

[11] The ACS Web Page: http://www.eso.org/~gchiozzi/AlmaAcs/index.html

[12] The ACS 2.0 Online Documentation:
http://www.eso.org/~gchiozzi/AlmaAcs/Releases/ACS_2_0_Docs/index.html

[13] Use Cases: http://www.nrao.edu/~dshepher/alma/usecases

[14] Java Swing tutorial: http://java.sun.com/docs/books/tutorial/uiswing

[15] JavaDocs: http://java.sun.com/j2se/javadoc/index.html

[16] Object Management Group "CORBA Specification" http://www.omg.org

6. Attachments

6.1 Users Manual

6.2 Documentation

6.3 Software Code