Programmer's Manual

Heiko Sommer (hsommer@eso.org)

ESO

Version	Published	Changed By	Comment
CURRENT (v. 5)	**Apr 13, 2019 22:51**	Jorge Avarias	
v. 4	Apr 13, 2019 22:33	Jorge Avarias	
v. 3	Apr 13, 2019 22:03	Jorge Avarias	
v. 2	Apr 13, 2019 22:00	Jorge Avarias	
v. 1	Apr 13, 2019 22:00	Jorge Avarias	

Purpose and Scope of the Document

The purpose of this document is to give a practical introduction to writing Java components for ALMA software, using the ACS container/component framework. You should have read the section on Technical Architecture in [1] or have learned otherwise about the concepts of container/component and XML binding classes to be used for ALMA software. You should also be familiar with the ALMA build environment (see [8]), even though a few things are explained redundantly in this tutorial.

While describing the steps involved in developing two sample components, I will try to provide information beyond the scope of the demo components in order to help using the framework for concrete ALMA subsystem development. However, this is not a concept or design document for the respective parts of ACS. It should help you to get started nonetheless.

The Java container/component model is fully integrated in ACS. It is meant to be used by ALMA subsystems or the parts of them that don't have real-time requirements and don't directly control hardware devices.

Abbreviations

This document no longer maintains a separate section for abbreviations. Please use the ALMA software glossary [13].

References

1. ALMA Software Architecture, Section 5 (Technical Architecture) http://almaedm.tuc.nrao.edu/forums/alma/dispatch.cgi/Architecture/docProfile/100018/d20030224132141/No/ALMASoftwareArchitecture_1.0.1.pdf
2. ACS Architecture http://www.eso.org/projects/alma/develop/acs/OnlineDocs/ACSArchitectureNL.pdf
3. ACS FAQ http://almasw.hq.eso.org/almasw/bin/view/ACS/AcsFAQ
4. Management and Access Control Interface Specification http://www.eso.org/projects/alma/develop/acs/OnlineDocs/Management_and_Access_Control_Interface_Specification.pdf
5. ACS Command Center User Guide http://www.eso.org/~almamgr/AlmaAcs/OnlineDocs/ACSCommandCenter/Acs_Command_Center_-_User_Guide.html
6. ACS Error System http://www.eso.org/~almamgr/AlmaAcs/OnlineDocs/ACS_Error_System.pdf
7. CDB Tutorial http://www.eso.org/~almamgr/AlmaAcs/OnlineDocs/CDB.pdf
8. ALMA Software Development Tools and Integration Procedures http://www.eso.org/~almamgr/AlmaAcs/OnlineDocs/IntGuidelines.pdf
9. OMG IDL to Java Language Mapping http://cgi.omg.org/cgi-bin/doc?formal/02-08-05
10. Eclipse download www.eclipse.org
11. Eclipse background information and configuration for ALMA http://almasw.hq.eso.org/almasw/bin/view/ACS/FAQJCompEclipse http://www.eso.org/~hsommer/eclipse20020923.htm
12. Castor XML Java binding framework http://castor.codehaus.org
13. ALMA software glossary http://almaedm.tuc.nrao.edu/forums/alma/dispatch.cgi/Glossary

Readers should feel very much encouraged to follow at their computers the steps described in the chapters below, switching between this tutorial and the real software.

Getting Eclipse IDE

The Java parts of the ACS framework were developed using the Eclipse IDE. Eclipse is not one of the "official" ALMA standards, but a majority of ALMA developers finds it so helpful that I like to recommend getting it. You can download Eclipse for free from [10]. The latest milestone build is usually fine, but special plugins may require to use a released version.

We have started a description on how to configure Eclipse to work well together with the ALMA build system [11]. It's expected to grow, feedback is welcome. As background information, notes from a presentation of Eclipse given at ESO are also available on that page.

Getting the source code

The ACS release (currently ACS 4.1.1) contains source code inside the delivered jar files, both for the implementation of the container, compiled IDL files etc, as well as for the examples used in this tutorial.
Automatically generated documentation and the source code for the examples can also be found online at
http://www.eso.org/projects/alma/develop/acs/OnlineDocs/jcontexmpl/doc/html/index.html.
A live web-based CVS view for all relevant ACS modules is at https://bitbucket.sco.alma.cl/projects/ASW/repos/acs/browse/LGPL/CommonSoftware
The modules that play a major role in the Java container/component part of ACS are listed below in compile order, together with the jar files they produce.

jacsutil	Misc. utility classes, test support	jACSUtil.jar
acserr	Exception handling	acserr.jar, acserrj.jar
xmljbind,Tools/castor	Java binding classes for XML (schema compiler, runtime) based on Castor	xmljbind.jar, castor.jar
comphelpgen	Java code generator for component helper classes, see 6	comphelpgen.jar
XmlIdl	ACS IDL compiler (produces XML binding class aware component interfaces)	xmlidl.jar

define	IDL and XML schema files that are part of the framework; Archive IDL	xmlentity.jar, archive_xmlstore_if.jar, systementities.jar
acstestentities	IDL and XML schema files used for demos and testing; forked from ObsPrep	acsTestEntities.jar
acsjlog	Logging	acsjlog.jar
jcont	Java container, component interfaces	jcont.jar
jcontexmpl	Examples used in this tutorial, other examples (e.g. used for testing)	HelloDemo.jar, XmlComponent.jar, BrightLamp.jar, BrightUser.jar, jcontexmpl.jar

Here's an overview of the directories and files in the module jcontexmpl:

/doc		Nothing too exciting since the API documentation is generated by the Makefile and is not supposed to be checked in.
/idl	HelloDemo.idl	definition of the HelloDemo component used in this tutorial
	XmlComponent.idl	definition of the XmlComponent component used in this tutorial
	EventComponent.idl LampAccess.idl LampCallback.idl	other demo components not considered in this tutorial
	ErrorSystemComponent.idl ErrorSystemComponent.xml	definition of errors with certain types and codes, see [6]
/include		not used, but required by build system
/lib	jcontexmpl.jar	compilation of /src, produced by running the Makefile in /src
	HelloDemo.jar	compilation of /idl/HelloDemo.idl
	XmlComponent.jar	compilation of /idl/XmlComponent.idl
	other .jar, .so, .a files	output of Java/C++ IDL compilation of the IDL files
/object		not used, but required by build system
/src		demo components and test clients
/test		tests internal to ACS that use the demo components

Compiling and running the examples

To compile and run the software, you need a current installation of ACS. The basic concepts for Java components have not changed since ACS 3.0 though.

Make

The /src and /test directories each contain a Makefile. The command

```
 make clean all
```

will compile IDL and Java code and pack the results into jar files.

```
 make install
```

will copy IDL, schema, and jar files to the respective sub-directories of your INTROOT directory so that they are available for other modules.
To learn about using the make system, refer to "man acsMakefile" and "man javaMakefile".

Run

To run the container, you must first start the ACS services & manager. Either use the graphical ACS Command Center (see [5]), or the command

```
acsStart
```

Make sure your ACS_CDBvariable points to the CDB installation that you want to use. You find a CDB suitable for the examples under jcontexmpl/test in CVS. You could either use that CDB directly, or merge the entries from Components.xml into your $ACS_CDB/CDB/MACI/Components/Components.xml.

To run the container on the same machine where the Manager runs, use the command

```
acsStartContainer –java <containername>
```

For cases where the manager runs on a different machine, please refer to [3] (“FAQGeneralCompContainerOptions”).

To terminate a Java container, you can use the command line

```
acsStopContainer <containername>
```

or the brute-force approach with Ctrl-C on the container console; in this case the Manager will attempt to reload components in that container next time the container runs.

The rest of ACS you terminate with

```
acsStop (or killACSif all else fails).
```

Using an IDE (Debugging!)

The recommended method is Java remote debugging, which is described in the ACS FAQs [3] under “How do I debug my components?”.

Windows development with limitations

Windows aficionados can run the Java container with its components (or just a client application that accesses components) on a Windows machine that has a network connection to another machine where the rest of ACS is running. Different subversions of JDK 1.4 have worked fine on Windows.

The build process (make) is not available under Windows. Assuming you have ACS installed on a Linux machine, you should keep your source code there, and mount the directories on Windows, e.g. using Samba The ACS FAQ will describe the setup in more detail.. The initial compilation must be done on Linux to compile IDL etc. Subsequent plain Java compilations can then be done from Windows.

There is an issue with CR/LF line termination: if you get ASCII sources from CVS on Windows, they come with Windows line breaks; the build process runs on Linux though, and a few tools will fail without telling you why. The Makefile itself and the IDL files are affected. In case of doubt, run dos2unix on them, e.g.

~/CVSPRJ/ACS/jcontexmpl/src 1001 > dos2unix Makefile

Logging

By default, log messages are sent to the local console and to the central ACS logger. There they simply get absorbed in the void if you don't register for them. For example,

```
 acsStartLoggingClient | tee mylog.xml
```

will send all logs messages to the file mylog.xml and also to the console where you typed the command.

To adjust the log levels that get through, you may provide your own Java logging properties file. It must be compliant to the JDK 1.4 logging spec. Refer to the FAQ for the details. As a template for the properties file, you could use almalogging.properties from acsjlog/src.

There is also a graphical application "LoggingClient" that you can start with the command

```
 jlog
```

During the design of your ALMA subsystem, you should partition your software into one or preferably several components The "subsystem master component" concept may be included in this tutorial in the future. A component will be the smallest chunk of software that can be deployed separately, e.g. in its own process or even on its own machine. Guidelines are that a component should be small enough to be functionally cohesive (having exactly one functional interface with related methods), but large enough to keep the number of components sufficiently small. Typically, a component will be made up of several Java classes, and a subsystem might consist of a handful of components.

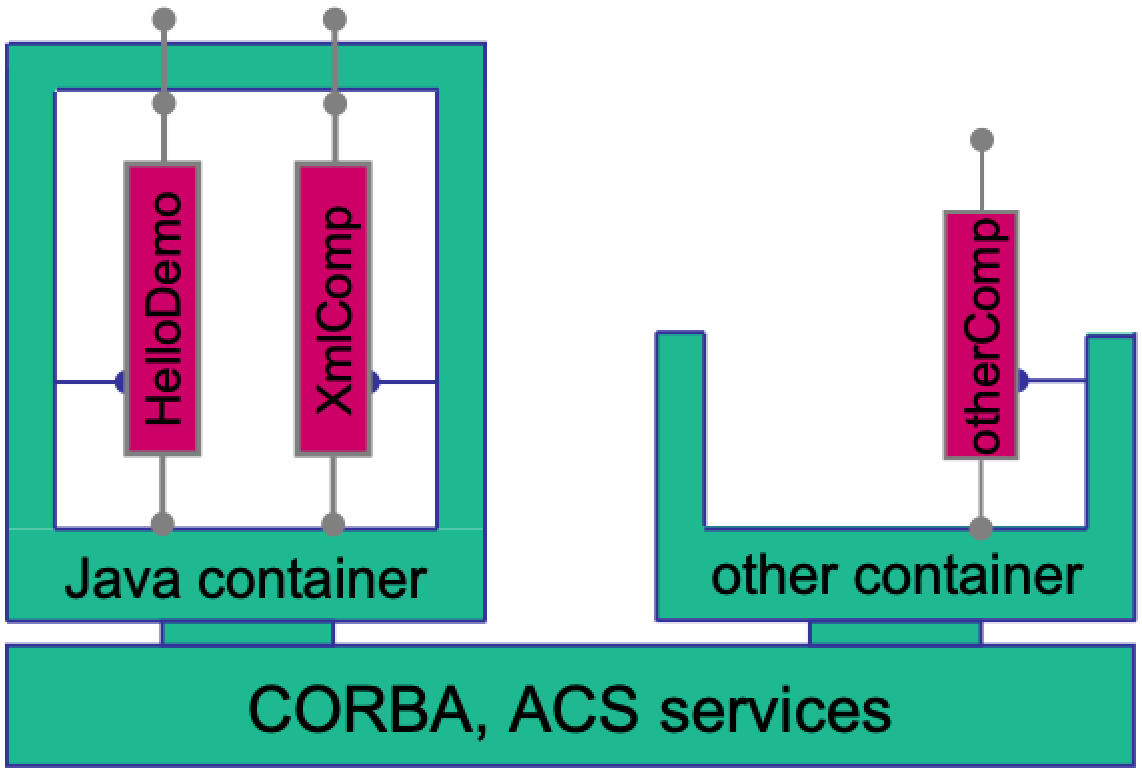
There can be components that will be used by other subsystems, while other components are only used internally by the subsystem. In either case, the functional interface must be specified as CORBA IDL.

The example components that we'll look at in the tutorial are not jewels of OO design, but they are meant to illustrate the steps involved to get to an implementation. The detailed motivation for all methods used in the interfaces is described in the next section about IDL.

We will work with

- a simple Java component “HelloDemo” that implements the notorious sayHello() method, both plain and with primitive dummy parameters
- a fancier “XmlComponent” that demonstrates the handling of XML entity objects with the transparent use of Java binding classes.

The components and their containers can be pictured as follows, with the “antennas” sticking out on top being the functional interfaces.



The two Java components, HelloDemo and XmlComponent, can either run collocated inside one Java container as shown, or run in separate Java containers. The Java container is drawn as a closed box to indicate that it works as a “tight container” (cf. [1]), intercepting all functional calls to any of its components, in addition to starting/stopping the components and providing explicit services to them.

The Java components can talk with, say, a C++ component “otherComp”, which must run inside a C++ container (“other Container”). That container is drawn open to indicate that it is a “porous” kind of container which only takes care of starting/stopping the components, but does not intercept functional calls on the running components.

The ACS containers use CORBA and the ACS services, the ACS Manager, and the Configuration Database. The Manager itself also uses the CDB.

IDL

A component's functional interface must be specified as an interface in CORBA's Interface Definition Language (IDL). An IDL file may contain more than one interface. The Java container framework does not require further restrictions Currently there is a problem with IDL attributes which will be fixed later. on the usage of the more exotic IDL constructs beyond what ACS already mandates for the definition of C++ components.
Using the ALMA directory structure, you should put IDL files in the <xxxx>/idl directory of your module. Our HelloDemo component is defined in jcontexmpl /idl/HelloDemo.idl.
Let's look at the listing of HelloDemo.idl with line-by-line explanations.

```
#ifndef _HELLODEMO_IDL_
#define _HELLODEMO_IDL_
#include <acscomponent.idl>
#pragma prefix "alma"

module demo
{
    // a very simple component
    interface HelloDemo : ACS::ACSComponent
    {
        string sayHello();

        string sayHelloWithParameters(in string inString,
                                      inout double inoutDouble, out long outInt);
    };
};
#endif
```

1, 2, 16	Standard include guard to avoid multiple inclusions of this IDL file
3	Mandatory include of acscomponent.idl (see line 9)
4	Standard CORBA directive that affects generated Java classes At least for what we care about here; primarily the prefix controls the CORBA repository ID.. The prefix pragma must come after the include statements (this restriction might be lifted in the future, but for now the TAO IFR would have a problem otherwise.) It defines the Java package that is prefixed to the package derived from the declared module ("demo") and interfaces, resulting in classes like "alma.demo.HelloDemo". It is an existing ACS practice/restriction to only prefix one package level, e.g. not use "alma.archive" as a prefix. This might be changed in the future when some issues with this have been resolved. The prefix seemed to be transformed differently into Java packages and CORBA repository IDs.
6	Declaration of the CORBA module "demo". A module may contain many interfaces. As a convention, there could be a 1:1 mapping between ALMA subsystems and modules, but this needs further discussions.
9	Interface declaration for the HelloDemo component. Every ALMA component's IDL interface must inherit from ACS::ACSComponent. This ensures that all components have a unique instance name and a state visible to other components.
1 1, 13	Silly methods using CORBA types string, double, and long

After the IDL file HelloDemo.idl has been written, a couple of Java classes must be generated from it.

The ALMA Makefile will take care of this, given the line (see jcontexmpl/src/Makefile)

```
IDL_FILES = HelloDemo
```

The generated Java classes are then automatically compiled and put into /lib/HelloDemo.jar. They belong to the package alma.demo and can be categorized as

Functional interface	HelloDemoOperations.class
Client-side CORBA	_HelloDemoStub.class, HelloDemo.class
Server-side CORBA	HelloDemoPOA.class, HelloDemoPOATie.class
Helper & holder classes	HelloDemoHelper.class, HelloDemoHolder.class

Warning: don't choose the same name for IDL_FILES as for (one of the values of) JARFILES in your Makefile! Otherwise the jar files produced from the IDL and from your module's code have the same name and one gets overwritten by the other.

Component Implementation

The implementation of a Java component is a Java class that must implement

- the functional interface (that corresponds to the IDL definition), and the
- alma.acs.component.ComponentLifecycle interface that the container uses to start and stop the component and to provide a callback handle.

The implementation of a Java component is a Java class that must implement

- the functional interface (that corresponds to the IDL definition), and the
- alma.acs.component.ComponentLifecycleinterface that the container uses to start and stop the component and to provide a callback handle.

The implementation class may inherit from any base class (This is possible because for Java components the framework uses the CORBA tie delegation model; C++ components can use multiple inheritance and extend the CORBA skeleton directly, thus saving one runtime delegation step.); no such base class is required by the framework though, which saves the single inheritance option offered by Java.

For components that don't need inheritance for other purposes, there is a generic base class (alma.acs.component.ComponentImplBase)that provides standard implementations of the required methods from ComponentLifecycle and ACSComponent, just to keep the component code simpler.

In real life we would usually first create the component implementation class with just dummy implementations for all mandatory methods, e.g. using code generation features of an IDE like Eclipse, and then move on to the next steps described in sections 6and below. This document will ignore the iterative process and look at the final implementations right away.

The functional interface for the HelloDemo component is HelloDemoOperations, a Java interface generated by the CORBA IDL compiler.

By convention, we call the implementation class “HelloDemoImpl” and create it in the package “alma.demo.HelloDemoImpl”. It can be found under jcontexmpl/src in CVS.

```
/* legal info cut out
 */
package alma.demo.HelloDemoImpl;
import java.util.logging.Logger;
import org.omg.CORBA.DoubleHolder;
import org.omg.CORBA.IntHolder;
import alma.ACS.ComponentStates;
import alma.acs.component.ComponentLifecycle;
import alma.acs.container.ContainerServices;
import alma.demo.HelloDemoOperations;

/** Javadoc cut out
 */
public class HelloDemoImpl implements ComponentLifecycle, HelloDemoOperations
{
  private ContainerServices m_containerServices;
  private Logger m_logger;

  /////////////////////////////////////////////////////////////
  // Implementation of ComponentLifecycle
  /////////////////////////////////////////////////////////////

  public void initialize(ContainerServices containerServices) {
    m_containerServices = containerServices;
    m_logger = m_containerServices.getLogger();
    m_logger.info("initialize() called...");
  }
  public void execute() {
    m_logger.info("execute() called...");
  }
  public void cleanUp() {
    m_logger.info("cleanUp() called..., nothing to clean up.");
  }
  public void aboutToAbort() {
    cleanUp();
    m_logger.info("managed to abort...");
  }

  /////////////////////////////////////////////////////////////
  // Implementation of ACSComponent
  /////////////////////////////////////////////////////////////

  public ComponentStates componentState() {
    return m_containerServices.getComponentStateManager().getCurrentState();
  }
  public String name() {
    return m_containerServices.getComponentInstanceName();
  }

  /////////////////////////////////////////////////////////////
  // Implementation of HelloDemoOperations
  /////////////////////////////////////////////////////////////

  public String sayHello() {
    m_logger.info("sayHello called...");
    return "hello";
  }

  public String sayHelloWithParameters(String inString,
      DoubleHolder inoutDouble, IntHolder outInt) {
    m_logger.info("sayHello called with arguments inString=" + inString
        + "; inoutDouble=" + inoutDouble.value
        + ". Will return 'hello'...");
    outInt.value = (int) Math.round(Math.E * 10000000);
    return "hello";
  }
}
```

20	The package declaration in accord with the "subpackage-per-component-implementation" convention
21	Import for logging (standard JDK logger configured by ACS)
22-23	Imports for CORBA Holder classes for the OUT parameters in the sayHelloWithParameters method (as defined in [9])
24	Import of the ComponentStates interface, needed to access (and optionally modify) the component state
25	Import for the mandatory Lifecycle interface
26	Import for the ContainerServices interface which the container provides to the component. From this interface, the component gets everything the container can do for it on request (that is, not transparently w/o the component noticing)
27	Import for the HelloDemoOperations interface, which defines the functional methods that a client of our component would use.
38	The component implementation class implements the functional and the lifecycle interface; it does not inherit from any base class (if so, there would be even less here to look at…)
40	Reference to the ContainerServices object that the container will give us (see line 47)
41	Ref to the logger object which we'll get from the container (see line 49)
47-51	The initialize method (from ComponentLifecycle) is called by the container after creating an instance of our component. It provides us with the ContainerServices object. We don't use it for much besides getting a logger and storing that reference in a separate variable for easy access. We could do some component initialization, and could throw a ComponentLifecycleException if this something had failed.
52-54	Implementation of the execute method (from ComponentLifecycle). This method will be called by the container after initialize() has returned. Only few components are foreseen to use both initialize and execute; here we just log a message. The component will not be available to clients before execute() has returned.
55-57	Implementation of the cleanUp method (from ComponentLifecycle). This method will be called by the container when the component is getting dismissed by the ACS Manager, but only after the last call to any of the methods from HelloDemoOperations has returned. This would be the place to free resources (other components etc)
58-61	Implementation of the aboutToAbort method (from ComponentLifecycle). This method may be called asynchronously when the container or the entire ALMA system must shut down without waiting for proper termination of all components. It is meant as a notification to the component to perform the most urgent actions before being forcefully removed at any time.
67-69	Implementation of the componentState() method (from ACSComponent). The component has it's state managed by the container in a default way (but could also change it itself). Here we simply return the state which the container keeps for us; in real-world components, we could check the state of other components that ours depends upon, and then compute the state.
70-72	Implementation of the name() method (from ACSComponent). The name is the instance name of our component, e.g. something we can't know at compile time. We get the name from the container, using the ContainerServices interface. For static components (instances known at deployment), this name is configured in the configuration database (CDB). For dynamic components, it's either specified by the component which creates it, or a default name is chosen by the framework.
78-81	Implementation of the sayHello method which is declared in HelloDemoOperations. Line 79 shows how to log a simple message.
83-90	Implementation of the sayHelloWithParameters method: we set the OUT parameter and leave the INOUT parameter untouched. Note the use of DoubleHolder and IntHolder for outgoing parameters, a concept that is not possible in Java without using these helper classes; they are therefore defined in the CORBA IDL-to-Java mapping spec. [9]

Note that the implementation of the HelloDemo component is done with just one Java class. A real component would rather have one main implementation class that implements the component interfaces and uses other classes to perform the functional tasks.

Unlike the previous HelloDemo, the component XmlComponent uses entity objects that are transported over the network as serialized XML. We will focus on this feature and not repeat things which have already been demonstrated before.

XML entity classes

Overview

The term "entity" refers to non-primitive, non-binary data with identity such as scheduling blocks, system users, correlator configurations etc., c.f. [1]. These data types are defined in XML schemas. Their instances are XML "documents" that get passed around by value and can be stored in the ALMA Archive.

From the XML schemas, we produce Java classes at build-time, based on the Castor framework [12]. These binding classes represent the hierarchical data structure in memory. They have type-safe methods to navigate the tree and to access data items. The binding classes can be automatically instantiated from XML documents that correspond to the same schema, and can serialize their data into stringified XML.

The idea of “transparent XML entities” is to have the entity objects transported by CORBA as language-neutral XML strings. However, the Java component implementations (both client and server involved in the remote call) don’t have to worry about the serialization and parsing of the XML. The main advantage of this is that the framework ensures type safety, from the IDL/schema definitions, all the way down to the Java implementation classes of a component; it’s not possible to be surprised at runtime that the received XML differs in type or version from what the component expected. To achieve this, the implementation classes use modified component interfaces, in which Java binding classes are substituted for the structs with serialized XML. An example with detailed explanation will be given in 5.3.

A Java component that acts either as a client or as a server with respect to some other component, is completely independent in its decision whether to parse XML entity objects “by hand” or to enjoy transparent support for binding classes. The other component may be written in Java, C++, or whatever other language, and it may or may not use binding classes (If this other component is written in Java, the developer has a real choice on this; for C++ at least at the moment there is no support for XML binding classes, so the C++ XML parser (expat) that comes with ACS should be used.).

For example, a Java client that exchanges XML entity objects with a C++ server can work with Java binding classes, whereas the C++ server parses the XML using a SAX parser.

Designing entity schemas

We define complex data types in XML schema and use these types in CORBA IDL similar to how CORBA structs would be used.

To keep this tutorial focused on component development, we only touch on the schema development issue. Rather than showing how to develop a new schema, we use the existing schemas from the module acstestentities.
For XML schemas describing ALMA entities, there are the following conventions:

- One schema per entity. This means that the entity "SchedBlock" must be defined in a different XML schema than the entity "ObsProject". "Different schema" really means "different schema namespace"; on the file level, one schema can be composed of various files connected through <xsd:include>, as long as these files belong to the same namespace. The reason for using included files would be to make large schemas more readable for the human eye.
- XML namespaces: the typical convention would be to use a namespace that looks like an internet URL (e.g. targetNamespace="http://www.alma.org/xmlschemas/obsprep"). This does not work for ALMA though, since "alma.org" is not ours, and "alma.int" conflicts with the integer type in Java. Thus in ALMA, the namespace should be "Alma/<subsystem>/<schemaName>" (e.g. "Alma/ObsPrep/SchedBlock"). Our test schemas use the namespace "AlmaTest/schemaName" instead.
- XML schemas consist of "elements" and "types". Names of schema types must end with a "T", like in <xsd:complexType name="**ObsUnitT**">. This avoids problems with generating binding classes (e.g. alma.mypackage.ObsUnitT), and makes it easier to find a name for a wrapper class that adds functionality to such a binding class (e.g. alma.mywrappers.ObsUnit).
- Top-level entities are defined as the root elements in a schema file, like ObsProject or SchedBlock. To allow the system to store administrational data like ID, version etc., the root element must have a child element of type EntityT (defined in the ACS module "define"/idl/CommonEntity.xsd, with the xml namespace "Alma/CommonEntity"). For example, in the schema file ObsProject.xsd, the schema element ObsProject has a child of type ObsProjectEntityT, which inherits from EntityT.Perhaps it would have been more intuitive to use an inheritance convention instead of composition (SchedBlock could inherit from EntityT), but since XML schema only allows single inheritance, we don't want to use it up for the sake of the framework. In the file SchedBlock.xsd we see that in fact our schema element "SchedBlock" inherits from ObsUnitT, which would not have been possible otherwise.
- From one entity object (that is, an XML document) you can reference another entity object through the ID. In the schema, this must be declared uniformly using (a subtype of) EntityRefT, which declared in the schema file CommonEntity.xsd. For example, a SchedBlock references the ObsProject to which it belongs. The schema element SchedBlock has a child element ObsProjectRef whose type inherits from EntityRefT. Therefore, an XML document for a SchedBlock entity can reference another XML document that is an ObsProject.

Compiling entity schemas

Schema files must be put into the /idl directory of your module, or into the ICD/<yourSubsystem>/ws/idl directory if Like IDL files are meant to be used by other subsystems. Please check with ALMA software engineering.

You must add information to your module's Makefile so that the XML schemas can be compiled into Java binding classes. Let's look at how acstestentities /src/Makefile has it:

```
 XSDBIND = acsTestEntities
```

tells the build process to look for a configuration file ../idl/acsTestEntities.xml which must provide information about the schema files to be compiled (see below). The resulting Java classes will then be packed into ../lib/acsTestEntities.jar.

Schema files can include or import other schema files. In our case, CommonEntity.xsd from the module ACS/define, systementities.jar, is imported. In our Makefile, the line

```
 XSDBIND_INCLUDE = systementities
```

is required so that the build process can set the include path for the schema compiler.

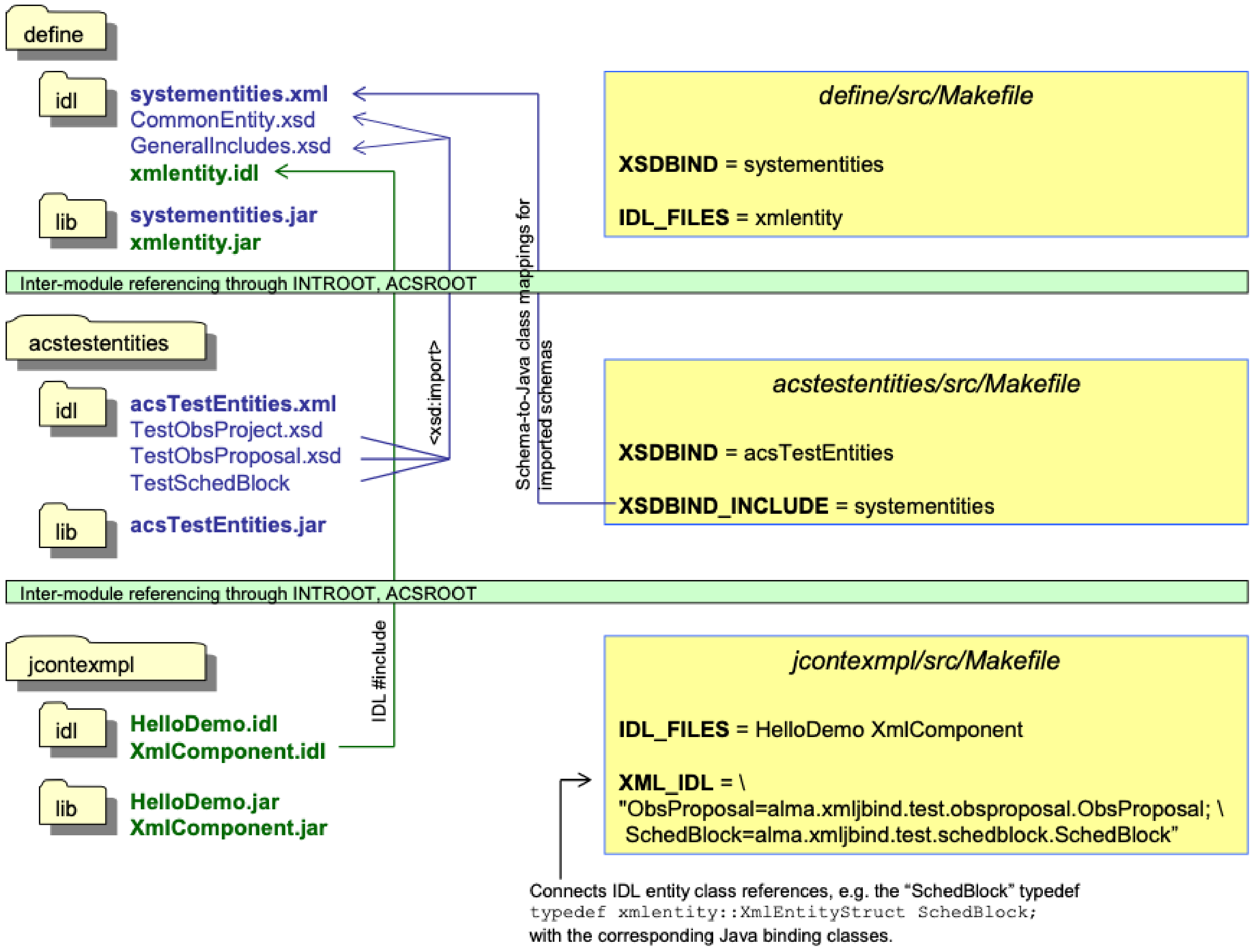
Here's a reduced listing of the configuration file /idl/acsTestEntities.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<EntitybuilderSettings>
  <EntitySchema schemaName="TestObsProject.xsd" xmlNamespace="AlmaTest/ObsProject" />
  <EntitySchema schemaName="TestObsProposal.xsd" xmlNamespace="AlmaTest/ObsProposal" />
  <EntitySchema schemaName="TestSchedBlock.xsd" xmlNamespace="AlmaTest/SchedBlock" />
  <XmlNamespace2JPackage xmlNamespace="AlmaTest/ObsProject" jPackage="alma.xmljbind.test.obsproject" />
  <XmlNamespace2JPackage xmlNamespace="AlmaTest/ObsProposal" jPackage="alma.xmljbind.test.obsproposal" />
  <XmlNamespace2JPackage xmlNamespace="AlmaTest/SchedBlock" jPackage="alma.xmljbind.test.schedblock" />
</EntitybuilderSettings>
```

We see that schema files are listed together with their xml namespaces (<EntitySchema>). Then the namespaces are mapped to Java packages (<XmlNamespace2JPackage>). Files and packages are separated here because several schema files can use the same xml namespace, but the Java package of the resulting classes must depend on the namespace, not the schema file.

Note that the compilation of xml schemas is specified differently in the Makefile, compared with the compilation of IDL files. IDL files are listed individually and are compiled into one jar file each, whereas schema files are only referenced indirectly through their XML configuration file, and are compiled into only one jar file per module.

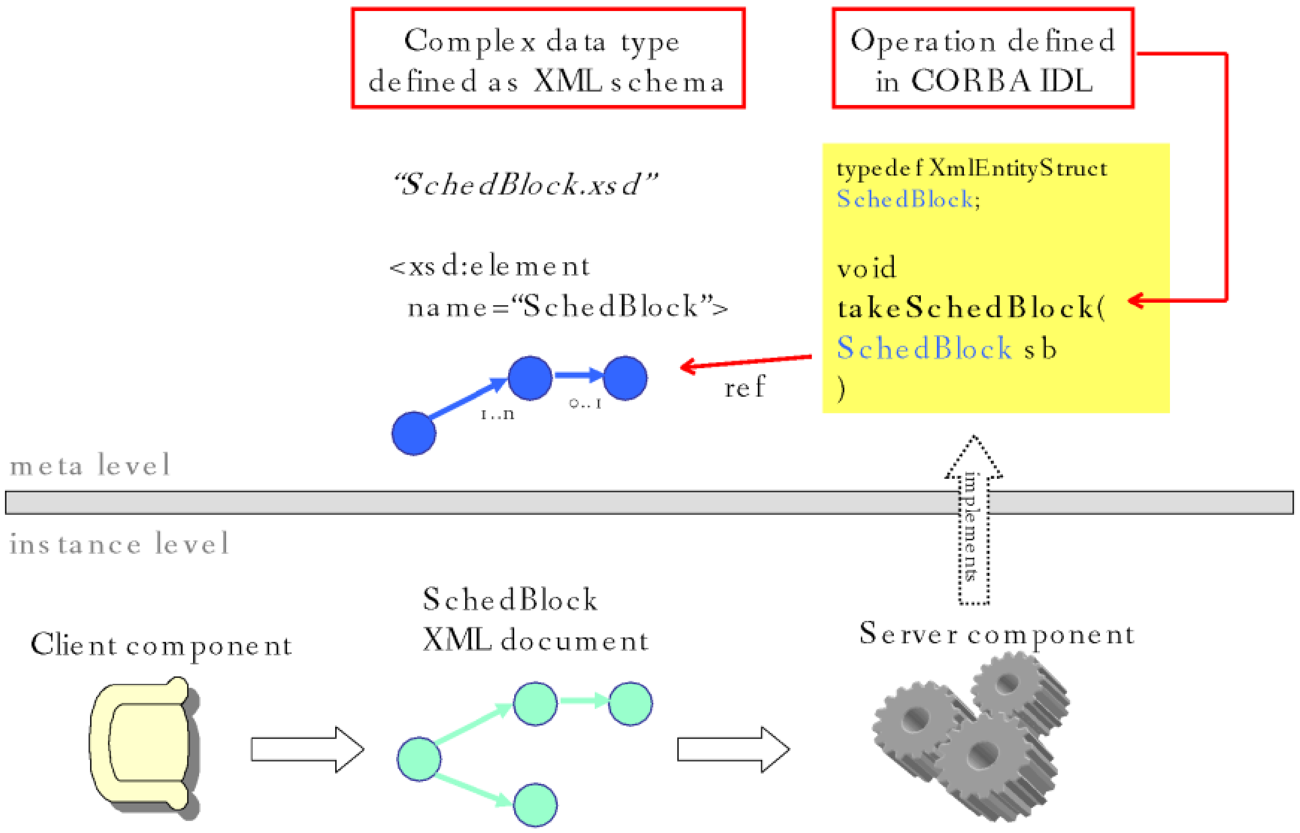
The diagram below illustrates the relationships among the xml schema files, IDL files, and Makefiles from our module "acsjexmpl" and the modules "define" and "acstestentities" from where resources are used by the example module.


define
idl
systementities.xml
CommonEntity.xsd
GeneralIncludes.xsd
xmlentity.idl
lib
systementities.jar
xmlentity.jar
define/src/Makefile
XSDBIND = systementities
IDL_FILES = xmlentity
Inter-module referencing through INTROOT, ACSROOT
acstestentities
idl
acsTestEntities.xml
TestObsProject.xsd
TestObsProposal.xsd
TestSchedBlock
lib
acsTestEntities.jar
<xsd:import>
Schema-to-Java class mappings for imported schemas
acstestentities/src/Makefile
XSDBIND = acsTestEntities
XSDBIND_INCLUDE = systementities
Inter-module referencing through INTROOT, ACSROOT
jcontexmpl
idl
HelloDemo.idl
XmlComponent.idl
IDL #include
lib
HelloDemo.jar
XmlComponent.jar
jcontexmpl/src/Makefile
IDL_FILES = HelloDemo XmlComponent
XML_IDL = \
"ObsProposal=alma.xmljbind.test.obsproposal.ObsProposal; \
SchedBlock=alma.xmljbind.test.schedblock.SchedBlock"
Connects IDL entity class references, e.g. the "SchedBlock" typedef
typedef xmlentity::XmlEntityStruct SchedBlock;
with the corresponding Java binding classes.

IDL

Using XML schema types in IDL – general idea

The diagram below shows how to use the XML schema definition of a SchedBlock in the IDL definition of a component interface.



Complex data type defined as XML schema
Operation defined in CORBA IDL
"SchedBlock.xsd"
<xsd:element name="SchedBlock">
1..n
0..1
ref
typedef XmlEntityStruct SchedBlock;
void takeSchedBlock(SchedBlock sb)
implements
meta level
instance level
Client component
SchedBlock XML document
Server component

XmlComponent.idl

Here's a thinned-out listing.

```
#include <xmlentity.idl>
#include <acscomponent.idl>
#pragma prefix "alma"

module demo
{
       typedef xmlentity::XmlEntityStruct ObsProposal;
       typedef xmlentity::XmlEntityStruct SchedBlock;
       typedef sequence <SchedBlock> SchedBlockSeq;

       exception XmlComponentException {};

       struct ObsProjectTree
       {
              ObsProposal prop;
              SchedBlockSeq schedBlocks;
       };

       interface XmlComponent : ACS::ACSComponent
       {
              long dumbMethod(in string somevalue);
              ObsProposal createObsProposal();
              SchedBlockSeq getAllSchedBlocks();
              void xmlInOutMethod(in ObsProposal opsPropIn,
                                      out SchedBlock schedBlockOut);
              ObsProjectTree getEntireTreeInAStruct();
              void exceptionMethod() raises (XmlComponentException);
       };
};
```

1	xmlentity.idl comes from define/idl and declares the CORBA struct XmlEntityStruct which is used to transport serialized XML together with some meta data. It must be included whenever xml entity objects are used (see I-K). The developer does usually not have to know the details of that struct.
3	The prefix pragma must come after the include statements; this restriction might be lifted in the future, but for now the TAO IFR would have a problem otherwise.
5	We use the same module as for the HelloDemo component
7	Typedefs for xml entity classes. The interface methods that use entity classes as parameters or return types (see H-K) must use these named typedefs like 'ObsProposal' instead of 'XmlEntityStruct'. This not only makes the interface more readable, but is also required for the automatic use of Java binding classes. Notice that the defined names are used to match the XML entity (IDL struct) with its corresponding binding class. For instance, "ObsProposal" will be matched to the Java class alma.xmljbind.test.obsproposal.ObsProposal.
11	Declaration of an exception. (How to use the ACS Error System [6] instead of plain CORBA exceptions may be demonstrated in a future version of this document.)
13	A struct that contains XML entities.
19	Interface declaration for the XmlComponent
21	Just to have something dumb in here
22	Method that returns an xml entity object (as a struct that contains the xml data as a string if the component is accessed as a plain CORBA object, see below.)
23	Demonstrates the use of a sequence (~array) of xml entity objects
24	Demonstrates the use of xml entity objects as an OUT parameter
26	Uses the ObsProjectTree struct, with entities inside
27	Uses an exception (again, later the mechanism described in [6] will be used.)

Compiling IDL

We have to add the new IDL file to jcontexmpl/src/Makefile:

```
IDL_FILES = HelloDemo XmlComponent
```

In addition to running this file through the standard CORBA IDL compiler, it is also fed into the "ACS IDL compiler". This tool gets started by the build process, so you never have to see it; the interested reader can find it in the directory ACS/…/XmlIdl. The ACS IDL compiler creates additional Java classes which the CORBA IDL compiler would not create. They are used for working with Java XML binding classes instead of plain XML strings, see 5.1.1.
In the Makefile, the line

```
XML_IDL = "ObsProposal=alma.xmljbind.test.obsproposal.ObsProposal; \
SchedBlock=alma.xmljbind.test.schedblock.SchedBlock"
```

provides the mapping from IDL typedefs (see lines 6-8) to Java binding classes (conceptually: to the schema that defines the entities).
The Java files that are produced from XmlComponent.idl are

Functional interfaces	XmlComponentOperations.class, *XmlComponentJ.class*
Client-side CORBA	_XmlComponentStub.class, XmlComponent.class
Server-side CORBA	XmlComponentPOA.class, XmlComponentPOATie.class
Helper & holder classes	ObsProposalHelper.class, *ObsProposalHolder.class*, SchedBlockHelper.class, *SchedBlockHolder.class*, SchedBlockSeqHelper.class, SchedBlockSeqHolder.class, *SchedBlockSeqJHolder.class*, XmlComponentHelper.class, XmlComponentHolder.class

The classes shown in normal print are mandated by CORBA standards and generated by the CORBA IDL compiler, while the italicized classes are required by the ACS framework and are generated by the ACS IDL compiler.

Component implementation

In section 5.2, we saw the IDL file with the typedefs for the entity classes, and their use in the methods of the XmlComponent interface.
The following two functional interfaces have been generated for XmlComponent. The component implementation must implement one of them:

- XmlComponentOperations uses serialized XML; if the component implemented this interface, it would have to parse and serialize the xml entity objects "manually".
- XmlComponentJ uses Java binding classes; the container will offer transparent parsing and serialization of XML.

For each component class, a corresponding helper Naming convention: For the component of an interface "xxx", the helper file is named as "xxxComponentHelper". This should not be so easily confused with the IDL-generated "xxxHelper". class is needed. In most cases this helper class will be used only by the framework, which means that the component developer will not have to look at it much.
The framework uses the helper class to

- instantiate the component implementation class
- get information about the component's interface(s)
- get the CORBA POA Tie skeleton class for the component

We will choose the convenience and type-safety offered when implementing XmlComponentJ. The interface generated by the ACS IDL compiler is listed below (Javadoc lines omitted).

```
package alma.demo;

/**
 * XML binding class aware ACS component interface XmlComponentJ */
public interface XmlComponentJ extends alma.ACS.ACSComponentOperations
{
    public int dumbMethod(String somevalue);

    public alma.xmljbind.test.obsproposal.ObsProposal createObsProposal();

    public alma.xmljbind.test.schedblock.SchedBlock[] getAllSchedBlocks();

    public void addNewSchedBlocks(
                 alma.xmljbind.test.schedblock.SchedBlock[] newSchedBlocks);

    public void xmlInOutMethod(
                 alma.xmljbind.test.obsproposal.ObsProposal opsPropIn,
                 alma.demo.SchedBlockHolder schedBlockOut);

    public alma.demo.ObsProjectTreeJ getEntireTreeInAStruct();

    public void exceptionMethod()
        throws alma.demo.XmlComponentException;
}
```

Comparing this with the IDL definition, we see that binding classes like `alma.xmljbind.test.schedblock.SchedBlock`are used whenever the IDL contained a typedef’d entity struct. In addition to this direct substitution, the ACS IDL compiler created the class `ObsProjectTreeJ`which contains binding classes, thus substituting the IDL struct `ObsProjectTree`as the return type of the method getEntireTreeInAStruct();

Let’s look at the implementation of the method createObsProposal().

```
public ObsProposal createObsProposal()
{
  ObsProposal obsProp = new ObsProposal();

  try
  {
    ObsProposalEntityT entity = new ObsProposalEntityT();
    m_containerServices.assignUniqueEntityId(entity);
    obsProp.setObsProposalEntity(entity);

    obsProp.setPerformanceGoals(
           "peak performance enduring a 24-7-365 schedule.");
  }
  catch (ContainerException e)
  {
    m_logger.log(Level.SEVERE, "failed to create ObsProposal. ", e);
  }

  return obsProp;
}
```

We see that the Java class ObsProposal has type-safe methods like “setPerformanceGoals” or “setObsProposalEntity”.

The latter takes an object of type ObsProposalEntityT, also generated by the binding framework. It is used to store administrational information about the ObsProposal object, most important the object ID.

For a new entity object, a valid and unique object ID can be obtained conveniently through a call to ContainerServices#assignUniqueEntityId, for which the container collaborates with the archive. If the archive is not present, it defaults to using random numbers which should be almost always unique (2^64).

Our method implementation simply returns the top-level Java object for the ObsProposal. The container framework will then automatically serialize it to XML, embed the string in the XmlEntity CORBA struct, and send it to the client using CORBA. If the client is another Java component, its container will automatically create an ObsProposal class and fill it with the data from the serialized XML.

To keep the size of this tutorial small, the implementations of the other methods are not listed here. They are meant to serve as more detailed examples, for which the source code of the module jcontexmpl should be retrieved anyway.

Alternatively, the code listing can be found online at the ACS documentation site:

http://www.eso.org/projects/alma/develop/acs/OnlineDocs/jcontexmpl/doc/html/XmlComponentImpl_8java-source.html

To see how the code would look like if the component would implement XmlComponentOperations, take a look at the inner class XmlComponentHelper::IFTranslator.

For each component class, a corresponding helper[1]class is needed. In most cases this helper class will be used only by the framework, which means that the component developer will not have to look at it much.

---

[1]Naming convention: For the component of an interface "xxx", the helper file is named as "xxxComponentHelper". This should not be so easily confused with the IDL-generated "xxxHelper".

The framework uses the helper class to

- instantiate the component implementation class
- get information about the component's interface(s)
- get the CORBA POA Tie skeleton class for the component

A template for the helper class gets generated automatically by the build system if the Makefile contains the line

```
COMPONENT_HELPERS=on
```

The template has a file ending .java.tpl to ensure that no edited .java helper class gets overwritten. It is meant as a convenience for the component developer who will normally just have to remove the .tpl ending and then treat that Java file like his/her own, which requires checking it into GIT.

For our HelloDemo component, the helper class does not need to be changed (some Javadoc lines stripped from the listing):

```
package alma.demo.HelloDemoImpl;

import java.util.logging.Logger;

import alma.acs.component.ComponentLifecycle;
import alma.acs.container.ComponentHelper;
import alma.demo.HelloDemoOperations;
import alma.demo.HelloDemoPOATie;
import alma.demo.HelloDemoImpl.HelloDemoImpl;

/**
 * To create an entry for your component in the Configuration Database,
 * copy the line below into a new entry in the file $ACS_CDB/MACI/Components/Components.xml
 * and modify the instance name of the component and the container:
 * Name="HELLODEMO1" Code="alma.demo.HelloDemoImpl.HelloDemoComponentHelper" Type="IDL:alma/demo/HelloDemo:1.0"
Container="frodoContainer"
 */
public class HelloDemoComponentHelper extends ComponentHelper
{
  public HelloDemoComponentHelper(Logger containerLogger)
  {
    super(containerLogger);
  }

  protected ComponentLifecycle _createComponentImpl()
  {
    return new HelloDemoImpl();
  }

  protected Class _getPOATieClass()
  {
    return HelloDemoPOATie.class;
  }

  protected Class _getOperationsInterface()
  {
    return HelloDemoOperations.class;
  }

}
```

Only in special situations, when a Java component chooses to sometimes use Java binding classes, but other times send or receive serialized XML instead, then the helper class must be modified. This goes beyond the scope of this tutorial and will be described in a more specialized document.

An example for this can be found in alma.demo.XmlComponentImpl.XmlComponentComponentHelper from jcontexmpl/src/.

The ACS configuration database contains information about which components are deployed on the different machines. It assists the ACS Manager to locate components at runtime. All the runtime action is hidden from the programmer by the container.

There are two different types of components:

- static components whose instances are fixed at deployment time. These typically represent (abstracted) hardware devices, or service components for which we choose a fixed number of instances. One component can be deployed many times under different names, like “Pipeline1”, ”Pipeline2”.
- dynamic components whose type (and optionally container location) is fixed at deployment time, whereas the number and names of instances is determined only at runtime. For more details on this, refer to [3], topic “FAQGeneralCompDynamic”.

Let’s make HelloDemo a static component. We create a new line in the file CDB/MACI/Components/ Components.xml with

- The instance name (“Name”), which has the format of an ACS CURL, and is used by a client to get access to the component. We call it HELLODEMO1.
- The component helper class (“Code”, see section 6)
- The CORBA interface repository ID of the component (“Type” – remember that to CORBA a component is just a servant).
- The name of the Java container that will be asked by the ACS Manager at runtime to instantiate our components – called “frodoContainer” in the examples.

Note that the generated component helper class contains these values in a comment (code listing above, line 42). This facilitates adding the CDB entry using copy and paste.

The CDB file Components.xml then looks similar to

```
<?xml version="1.0" encoding="utf-8"?>
<Components xmlns="urn:schemas-cosylab-com:Components:1.0"
      xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
      xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<_ Name="TEST_PS_1"    Code="acsexmplPowerSupplyImpl"
     Type="IDL:alma/PS/PowerSupply:1.0" Container="bilboContainer"/>
         <_ Name="HELLODEMO1" Code="alma.demo.HelloDemoImpl.HelloDemoHelper"
    Type="IDL:alma/demo/HelloDemo:1.0" Container="frodoContainer" />
         <_ Name="XMLCOMP1" Code="alma.demo.XmlComponentImpl.XmlComponentHelper"
    Type="IDL:alma/demo/XmlComponent:1.0" Container="frodoContainer" />
<_ Name="LAMP1" Code="acsexmplLamp" Type="IDL:ALMA/PS/Lamp:1.0" Container="bilboContainer"/>
         <_ Name="OPERATIONAL_ARCHIVE" Code="alma.archive.components.OperationalHelper"
    Type="IDL:alma/xmlstore/Operational:1.0" Container="frodoContainer" />

</Components>
```

ACS ships with a default CDB that contains the entries for the demo components.

You’ll have to add entries for your own components, or create a new CDB structure and reset the ACS_CDB environment variable.

A component can have different kinds of clients which we'll discuss separately:

Generic Client: ObjectExplorer

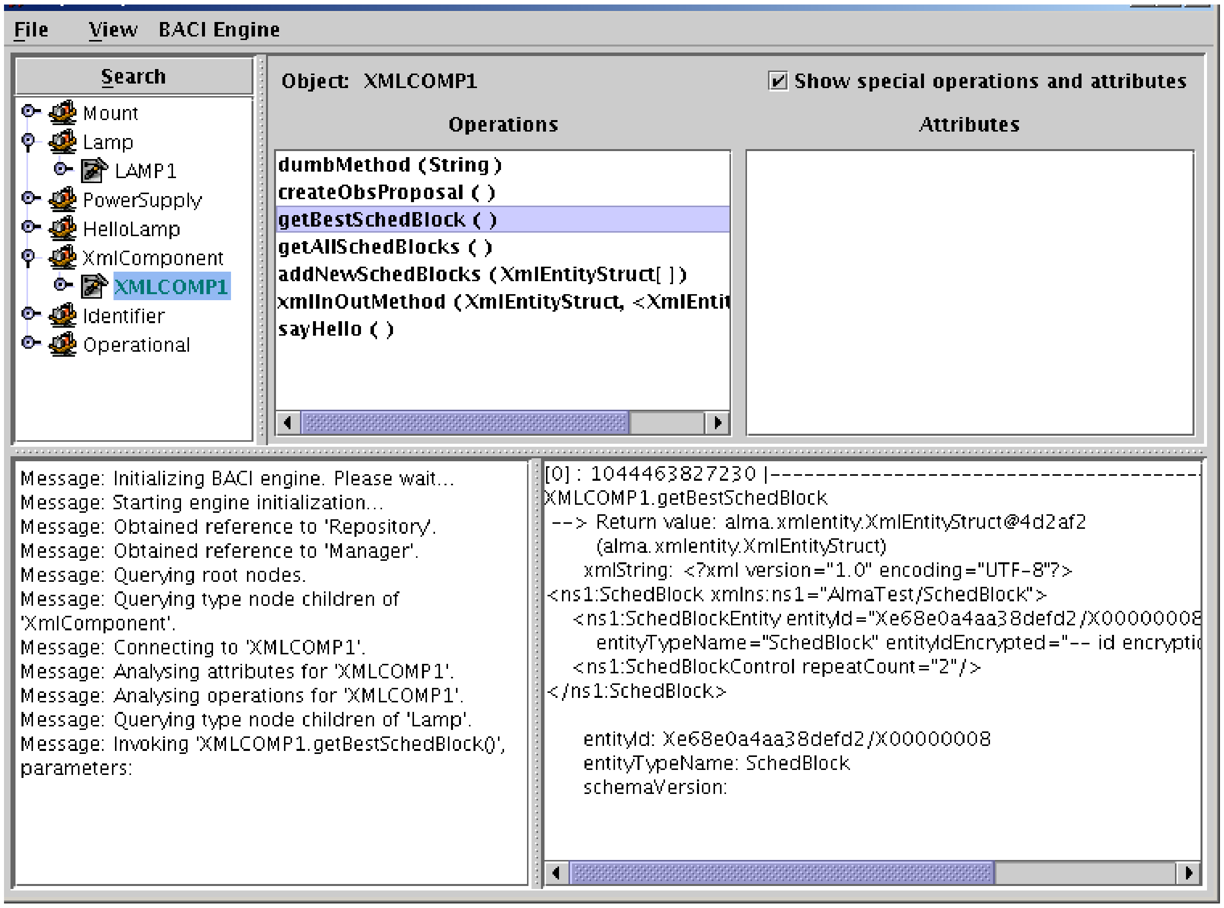
We can use the ACS tool ObjectExplorer as a generic client for all components. Run it with

```
objexp
```

and you should find your Java components (e.g. XmlComponent) and their instances (e.g. XMLCOMP1) listed on the dialog, together with C++ components (e.g. Lamp).

At first, you see only a visualization of the information in the CDB, as opposed to actually loaded components. When you click on a component instance, the Manager will tell the Container to activate the component with the given instance name.

The screenshot shows the output after XMLCOMP1 has been loaded and the method getBestSchedBlock has been called (with View Expand result data checked in ObjectExplorer.)



```
Message: Initializing BACI engine. Please wait...
Message: Starting engine initialization...
Message: Obtained reference to 'Repository'.
Message: Obtained reference to 'Manager'.
Message: Querying root nodes.
Message: Querying type node children of
'XmlComponent'.
Message: Connecting to 'XMLCOMP1'.
Message: Analysing attributes for 'XMLCOMP1'.
Message: Analysing operations for 'XMLCOMP1'.
Message: Querying type node children of 'Lamp'.
Message: Invoking 'XMLCOMP1.getBestSchedBlock()',
parameters:
```

```
[0] : 1044463827230 |---------------------------------------
XMLCOMP1.getBestSchedBlock
 --> Return value: alma.xmlentity.XmlEntityStruct@4d2af2
       (alma.xmlentity.XmlEntityStruct)
     xmlString: <?xml version="1.0" encoding="UTF-8"?>
<ns1:SchedBlock xmlns:ns1="AlmaTest/SchedBlock">
   <ns1:SchedBlockEntity entityId="Xe68e0a4aa38defd2/X00000008
      entityTypeName="SchedBlock" entityIdEncrypted="-- id encrypti
   <ns1:SchedBlockControl repeatCount="2"/>
</ns1:SchedBlock>

     entityId: Xe68e0a4aa38defd2/X00000008
     entityTypeName: SchedBlock
     schemaVersion:
```

Client Component

We've seen the example of a Java component accessing a C++ Component in 4.2, and a Java component accessing another Java component without transparent XML binding support referenced in 5.1.

To use support for xml binding classes on the client side of a call, we first obtain the other component (here: XMLCOMP1)

```
XmlComponentxmlComp=alma.demo.XmlComponentHelper.narrow(
        getComponent("XMLCOMP1"));
```

and then ask the container to wrap it with the more convenient interface that uses XML binding classes, provided in the ContainerServices interface

```
XmlComponentJ xmlCompJ=
(XmlComponentJ)m_containerServices.createXmlBindingWrapper(
XmlComponentJ.class,
                 xmlComp,
                 XmlComponentOperations.class);
```

Now we can keep working with xmlCompJ, and the framework will delegate all calls to xmlComp, translating between XML binding classes and serialized XML presentation in between.

JUnit Test Client

To facilitate the writing of JUnit test clients, ACS (module jcont) offers the class

`alma.acs.component.client.ComponentClientTestCase`

which is a subclass of JUnit's TestCase that does the talking with ACS Manager. It provides most of the functionality that ContainerServices provides for components.

A JUnit test that gets a reference to the XmlComponent component and runs some tests on it is

jcontexmpl/src/alma.demo.client.XmlComponentClient

```
package alma.demo.client;
import alma.acs.component.client.ComponentClientTestCase;
import alma.demo.SchedBlockHolder;
import alma.demo.XmlComponent;
import alma.demo.XmlComponentException;
import alma.demo.XmlComponentJ;
import alma.demo.XmlComponentOperations;
import alma.entities.commonentity.EntityT;
import alma.xmljbind.test.obsproposal.ObsProposal;
import alma.xmljbind.test.schedblock.SchedBlock;

public class XmlComponentClient extends ComponentClientTestCase
{
  private XmlComponentJ m_xmlCompJ;

  public XmlComponentClient() throws Exception {
    super("XmlComponentClient");
  }

  protected void setUp() throws Exception {
    super.setUp();

    org.omg.CORBA.Object compObj = getContainerServices().getComponent(
        "XMLCOMP1");
    assertNotNull(compObj);
    XmlComponent xmlComp = alma.demo.XmlComponentHelper.narrow(compObj);

    m_xmlCompJ = (XmlComponentJ) getContainerServices()
        .getTransparentXmlComponent(XmlComponentJ.class, xmlComp,
            XmlComponentOperations.class);

    assertNotNull(m_xmlCompJ);
  }

  public void testSayHelloUsingHelloDemoComponent() {
    String reply = m_xmlCompJ.sayHello();
    assertNotNull(reply);
    System.out.println("received reply " + reply);
    assertEquals("reply must be 'hello'", "hello", reply);
  }

  public void testCreateObsProposal() {
    ObsProposal obsProp = m_xmlCompJ.createObsProposal();
    assertNotNull(obsProp);

    EntityT ent = obsProp.getObsProposalEntity();
    assertNotNull(ent);

    String id = ent.getEntityId();
    assertNotNull(id);

    System.out.println("received ObsProposal with id " + id);
  }

  public void testXmlInOutMethod() {
    ObsProposal obsProp = m_xmlCompJ.createObsProposal();
    assertNotNull(obsProp);
    SchedBlockHolder sbh = new SchedBlockHolder();

    m_xmlCompJ.xmlInOutMethod(obsProp, sbh);

    SchedBlock schedBlock = sbh.value;
    assertNotNull(schedBlock);

    EntityT ent = schedBlock.getSchedBlockEntity();
    assertNotNull(ent);
    String id = ent.getEntityId();
    assertNotNull(id);

    System.out.println("received out-param SchedBlock with id " + id);
  }

  public void testException() throws Exception {
    boolean gotException = false;

    try {
      m_xmlCompJ.exceptionMethod();
    }
    catch (XmlComponentException e) {
      gotException = true;
      System.out.println("received XmlComponentException as intended.");
    }

    assertTrue("must receive XmlComponentException", gotException);
  }
}
```

The constructor must call the constructor of the base class with a name to be used for this client (here: “XmlComponentClient”). If you override the setup() method, it’s necessary there to call super.setUp() first.

JUnit will call all methods that start with “test” in their name, such as testException().

In the Eclipse Java Perspective, you can run such a class with JUnit. Open the class in the editor, then choose

Menu Run Run As JUnit Test. It will display the results.