An Introduction to Scalable Frameworks for Observatory Software Infrastructure

- SC-644

G. Chiozzi – European Southern Observatory
gchiozzi@eso.org

SPIE 2004
Astronomical Telescopes and Instrumentation
June 23, 2004

Course Abstract

This course provides an analysis of the advantages and requirements for an integrated software infrastructure for observatories and similar scientific facilities. It provides a common framework for application software that can range from control to data analysis applications. Currently available and emerging technologies are evaluated and compared. The course concentrates on the architecture of an application framework necessary for such an infrastructure and on the impact on scalability, maintainability and reuse. Many practical examples will be given based on the ALMA Common Software, a CORBA-based, open source solution used by ALMA and other projects.

Learning outcomes:
This course will enable you to:

•identify the advantages and requirements of an observatory-level software infrastructure

•compare existing and emerging technologies

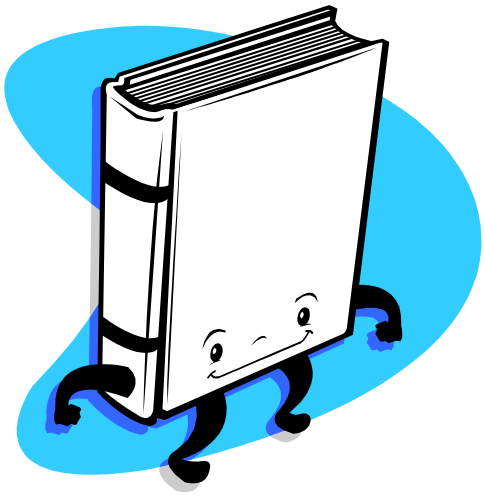
•estimate the impact of introducing a common software framework in a new or pre-existing project

•demonstrate applications implemented using the concepts described in the course

Intended audience:
This material is intended for anyone who is involved in the design and refurbishment of the software architecture of a scientific facility and in the selection of the middle-ware architecture to use. Those who develop applications integrated in the data flow and control infrastructure of an observatory will find this course valuable.

Instructor:
Gianluca Chiozzi currently works at the European Southern Observatory in Munich. For the last 10 years he has been heavily involved in the design and implementation of the VLT Common Software and Telescope Control Software. He is now responsible for the ALMA Common Software architecture and development. Before ESO he worked at the IBM Technical and Scientific Research Center in Milan.

1 - Introduction and Overview



Course Objectives

- Identify advantages and requirements of an observatory-level software infrastructure
- Compare technologies
- Estimate impact of a common software framework in a new or pre-existing project
- Demonstrate applications

This course provides an analysis of the advantages and requirements for an integrated software infrastructure for observatories and similar scientific facilities. It provides a common framework for application software that can range from control to data analysis applications. Currently available and emerging technologies are evaluated and compared. The course concentrates on the architecture of an application framework necessary for such an infrastructure and on the impact on scalability, maintainability and reuse. Many practical examples will be given based on the ALMA Common Software, a CORBA-based, open source solution used by ALMA and other projects.

Learning outcomes:
This course will enable you to:

•identify the advantages and requirements of an observatory-level software infrastructure

•compare existing and emerging technologies

•estimate the impact of introducing a common software framework in a new or pre-existing project

•demonstrate applications implemented using the concepts described in the course

I think it is important to adapt the course to the audience and follow up the questions.

Therefore these course note masters are not cast in the stone and, if necessary and useful, we can decide to go sometimes into deeper details or to skip parts that do not seem particularly interesting for the participants.

By experience, each course is different because of the different mix of participant's knowledge and experience.

Contents

1. Introduction and Overview
2. Observatory architecture
3. Distributed Systems and Middleware
4. Introduction to CORBA
5. Services and Facilities
6. From Object to Component Middleware
7. High Level Framework
8. Development Support
9. Wrapping up

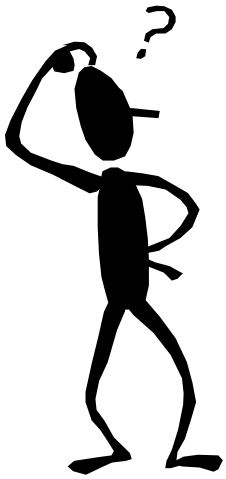
We will divide the course in 9 sections.

At the end of each section there will be time for questions, discussions and, in case, some extra detail.

You can interrupt me at any time for questions.

I will decide case by case if I have to reply immediately to the questions or if it is more efficient to reply later or to let the reply come out by itself from other parts of the presentation.

Let’s introduce ourselves



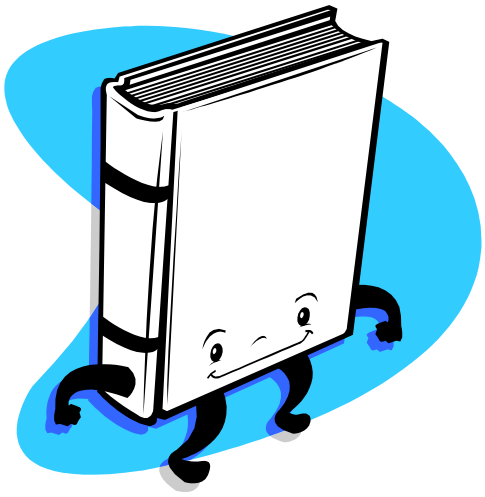
Let’s go around the classroom and introduce briefly ourselves:

•Who are we?

•What do we expect from this course?

•Any suggestions?

2 – Observatory Architecture



The “traditional” Observatory

- In the traditional observatory parts were independent, not integrated
- Complexity was low and there were no requirements of integration
- Astronomer was making observations and taking home the data

A modern Observatory (or any other experimental facility) has a complex integrated and distributed architecture.

In the past, the Astronomer (the main stakeholder for our systems) was traveling to the Observatory, make his observations, store data on tape and go back home to reduce it.

The telescope and, eventually, the instruments had a control system virtually independent from everything else.

Data reduction was done offline after the observation and there was no direct feedback from the observation data to the control system. An experienced observer was just driving the telescope based on his own feelings.

There was no observation data archive, not quality of service measures and constraints, no facility engineering in the terms we think of now.

This has dramatically changed in the past 15 years, with the big observatories like VLT, Keck, Gemini, Hubble and so on.

Since the major observatories are now providing integrated facilities, astronomers expect the same also from smaller ones and the amount of integration required for the new projects like ALMA and the giant optical telescopes will be even more.

The Virtual Observatory is also contributing to this need for integration and quality control adding the intra-observatory dimension to the problem.

Modern Observatory architecture

- Integration of all parts (end to end data flow)
- Archive and virtual observatory
- Feedback

- Compare major projects and small projects.

Now all the systems in an Observatory are fully integrated.

Observation data, weather and telemetry are directly fed back in the control system to obtain optimized performance.

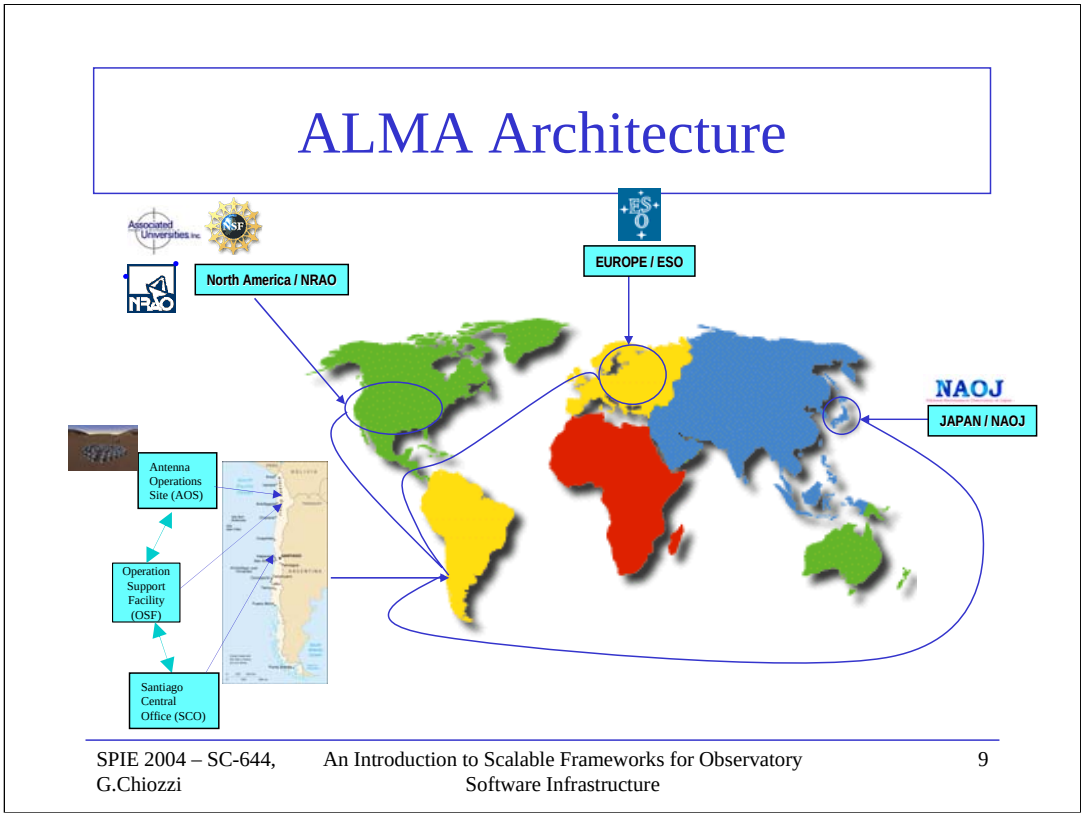
The astronomer is in many cases not even any more going to the observatory, but monitors the observation from his own institute and can ideally interact with the system remotely when the observation is taking place.

The astronomer expects to interact with the system using “standard” Web technology.

Data is archived to be usable by Virtual Observatories and therefore has to contain calibration and quality information.

Engineers and people responsible for the administration and maintenance of the observatory are another important stakeholder.

More and more performance is measured and telemetry information is analyzed daily to perform preventive maintenance and optimize the system or to measure performance trends over long time periods.



We can take as an example the Architecture of ALMA (but we could take the VLT, Hubble or any other major observatory).

We will start analyzing the overall architecture of ALMA and see how it impacts the software architecture.

First of all you can notice that ALMA will be a very geographically distributed system:

- AOS: Chajnantor (5000m)
  - Move antennas (scattered up to 14 km from correlator)
  - Swap in/out hardware modules
- OSF San Pedro (2800m, ~30 km from array)
  - Control & monitor array
  - Repair Hardware
- SOC: Santiago
  - Run pipeline
  - Maintain archive
- Regional Centers: USA, Europe, Japan
  - Accept proposals
  - Deliver data packages
  - Provide User support
- Astronomer's institutes:
  - Submit proposals
  - Monitor and interact with observing projects
  - Data reduction

Observatory Software Scope

- Proposal preparation and review
- Scheduling of observing programs
- Observation
- Calibration and Imaging
- Data delivery and archiving
- Data reduction
- Archival Research and Virtual Observatory compliance

Astronomers expect from the software in an Observatory a wide scope of services.

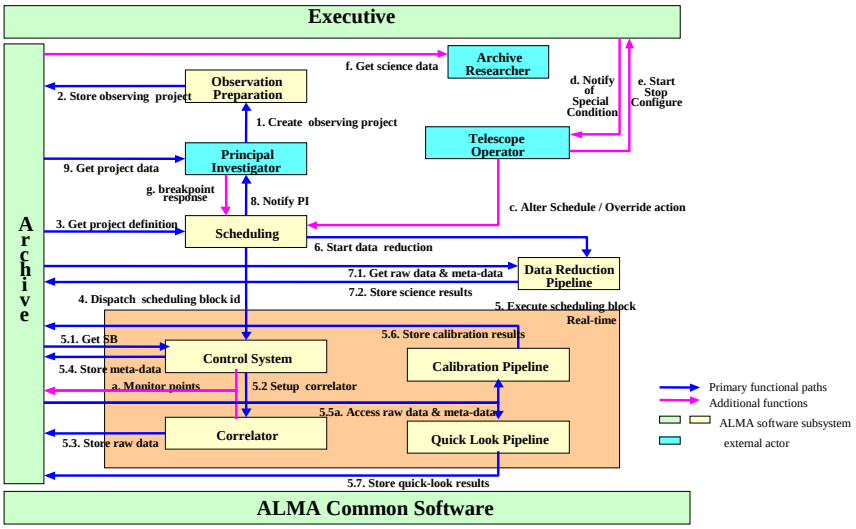
They need to be all integrated together.

Also, astronomical observation is not limited to experts in the field (like infrared or radio astronomers), but it shall be open to chemists, biologists and other multi disciplinary researchers.

The general user should be given standard observing modes to achieve project goals expressed in terms of science parameters, rather than technical quantities. But experts must be able to exercise at the same time full control.

Making things easy and flexible for the astronomer adds up complexity to the software development

ALMA Software Architecture



Again we can take as an example a schematic architecture of the ALMA software (A.Farris, J.Schwarz ALMA HLA team).

This diagram shows the main subsystems in which the software has been divided and the main relationships among then in the form of a collaboration diagram describing the typical lifecycle of a project, starting with proposal submission and ending with a researcher looking for the data in the archive after the observation has been performed.

Functional Architecture

- Software components/subsystems
  - Responsibilities
  - Interfaces
  - Primary relationships and interactions
- Physics and algorithms
- Hardware deployment and distribution

The previous diagram shows the “Functional Architecture” of the system.

The functional architecture is built based on the user requirements.

The functionality that needs to be implemented is assigned to components/subsystems and the architecture describes the responsibilities of each subsystem and the interfaces that are exposed to the other subsystems or to the external world.

Then the relationships between the subsystems (i.e. how these interfaces are used when asking reciprocally services) are described.

The functionality must be implemented according to the physics of the system and must implement specific algorithms that must be described in this architecture. For example scheduling algorithms, control algorithms, data reduction strategies are all part of the functional architecture.

Another essential driving factor is the actual deployment and distribution of the hardware that must be controlled by the software. For example, the physical deployment of motors and sensors and the physical connection of the electronics to the control computers affects the functional architecture of the system. Or the location of the data archives and of the CPU factories for data reduction.

Technical Architecture

- Communication mechanisms and networking
- Database technology
- Software deployment and activation
- Programming model

The “functional architecture” must be supported by a “technical architecture” that describes (and implements) the technical aspects of the software, like the communication protocols used, the threading model, the software deployment (process handling, distribution, activation and deactivation).

The requirements for the technical architecture are mostly derived requirements.

While the user requirements are the basis for the development of the functional architecture, we derive most of the technical requirements from the functional architecture itself: the technical architecture shall enable us to implement the functional architecture.

Separation of concerns

- Functional and technical architecture: two views
- Subsystem teams should concentrate on function
- Technical architecture provides them with simple and standard ways to, for example:
  - Access remote resources
  - Store and retrieve data
  - Manage security needs
- We want to keep the two concerns separate

Functional and technical architecture are two different views of the system.

Subsystem developers should concentrate on the functional aspects of the system, i.e. in the implementation of the physics and algorithms.

They have to be freed from the need of designing and implementing mechanisms for interfacing, communicating, deploying or handling security.

The detailed design of the Technical architecture must be mastered by infrastructure developers.

Application developers are required to understand the just the *concepts* of the both the technical and functional architecture.

Infrastructure Framework

An infrastructure framework is the key to this separation

- Programming model
- Satisfy performance, reliability and security requirements

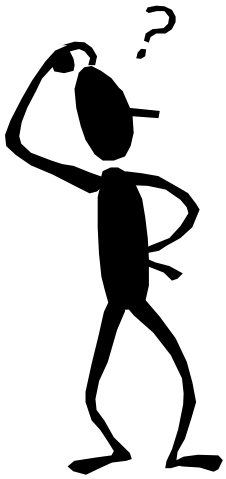
The key to reach this objective is to adopt a Software Framework that provides a consistent infrastructure for the whole observatory.

On one side the framework has to satisfy all the requirements of performance, reliability and security derived from the functional architecture.

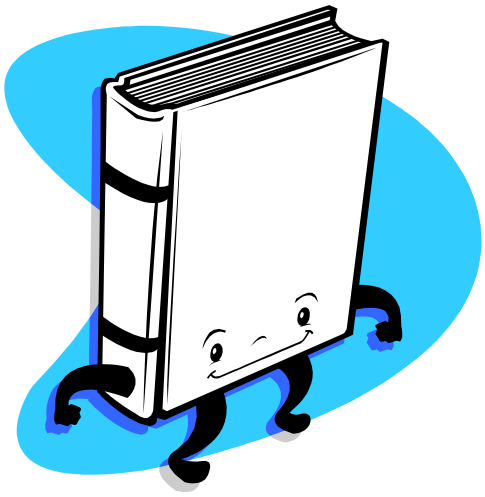
On the other side it must hide as much as possible its own internal complexity to the subsystem developers and provide them with a clear and streamlined programming model.

We will now begin to examine the requirements for such a framework in our domain and what options are available.

Questions (& Answers)



3 – Distributed systems and Middleware



Distributed System

- Requirement: the observatory is a distributed system.
  Servers and clients are distributed on different machines:
  - Possibly in different locations
  - With different purpose and functionality
  - With different requirements on performance and reliability

This actually implies an…..

As can be seen from the previous discussion, the architecture of the observatory is very distributed.

Servers and clients need to be distributed in different locations inside and outside the physical observatory where the telescope reside.

The different parts of the system (that we did not better specify yet) have different purpose and functionality and therefore have different requirements on performance and reliability.

If we take into account that parts of the system are dedicated to real time control of hardware, coordination, database management, data analysis up to the GUIs on the astronomer's desktop, we see that this distribution involves something more than a plain Distributed System.

Heterogeneous Distributed System

- Requirement: the observatory shall be an *heterogeneous* distributed system; servers and clients may use different:
  - Hardware
  - System software
  - Programming languages

What we really have is an *Heterogeneous* Distributed Systems, since the distribution involves different:

•Hardware platforms and architectures. From real time computers to PCs of any kind on the desktops we can have very different hardware architecture (CPU, word size, alignment, memory available…)

•System software. Any of these machine can have a real time operating system, Linux or other variants of Unix, Microsoft operating systems, MacOS or even more exotic software platforms like PalmOS

•Programming Languages. Different programming languages are more suitable for different application domains. For example, C and C++ are most suitable for real time and CPU intensive applications, while Java fits well in coordination, high level or GUI developments. Astronomers will want to write their observation scripts and reduction procedures in high level scripting languages.

Transparent Heterogeneous Distribution

- Separation of Concerns:
  - Developers of clients and servers have to be unaware of the respective architecture
  - It shall be possible to change the architecture of a server transparently to the client
  - Ideally the developer of a client should not even know if a client is local or remote.

In order to achieve the “separation of concerns” objective, applications developers have to be unaware of the architecture (hardware, software, programming language, location) of the servers they interact with.

Having to deal explicitly with network communication protocols, byte order of message data, connection reliability and similar problem would be a major burden on the shoulders of the application developer.

The technical framework has to take up this responsibility and hide all these problems to the functional developers.

It shall even be possible to fully replace the server with a different one without the client noticing.

We could (and this has been often the case in past projects) keep the heterogeneous domains separate. For example data analysis and control system could be implemented using different and independent software infrastructures, but this approach will lead to many problems in the interfaces. In the past, interfaces were limited and this was not an important issue. But the level of integration needed nowadays makes such a choice highly problematic.

The developer of a client application should interact with a server as if it was a local object, not even knowing if it is local or remote. Remote function calls should resemble local function calls.

Using directly low-level network protocols from the application layer (for example using send and receive on socket based communication) does not allow to reach these objective, because the application software has to be fully aware of the network protocols and communication. This code is typically non highly scalable and hard to maintain and change.

The Infrastructure Framework has to take care of these aspects of the system.

Middleware technology

Middleware is a software layer between the application layer and the network services and provides:

- Heterogeneous communication transparency
- Location transparency
- Message delivery and format integrity
- Dynamic invocation of server processes,
- Load balancing
- Security

A special layer of software between client and server processes is needed to deliver the extra functionality.
This software layer is called **middleware**. It hides the complexity of the extra functionality behind a common set of APIs that client and server processes can invoke.

Network protocol functionality only allows data exchange between client and server. More functionality is required for heterogeneous distributed systems, like:

•Location transparency
The application does not want to know the network address of the client it wants to use. A location (naming) service should allow to locate a service over the network by name and/or required functionality.

•Message delivery and format integrity
The system must warranty that messages are not lost or duplicated and that they are delivered uncorrupted.

•Dynamic invocation of server processes,
The client shall not be responsible for starting up the services it need, but the system shall be able to do it transparently

•Load balancing
If needed, the system shall be able to redistribute the services to allow load balancing over the distributed servers

•Security
If needed, the system must be able to handle security of communication using appropriate secure protocols. But without enforcing heavy communication protocols where there is no need and performance is an issue.

Object Middleware

- Most used development technologies and programming languages are Object Oriented:
  - Implement objects
  - Call operations of objects
- Call local or remote objects transparently:
  - Client and object vs. client and server
- Object Reference:
  - first get one,
  - then use it.

Object Oriented concepts are pervasive in current software development: essentially all development methodologies and programming languages used nowadays are to some extent Object Oriented. Even scripting languages like Python provide support for Object Orientation.

Developers are used to think in terms of objects and object oriented programs are based on one side on the implementation of objects (define a class and implement it) and on the other side on clients invoking methods provided by instances of objects.

It is very natural to extend this concept outside the boundaries of a programming language or of a process on one host.

Object Middleware allows to call operations of objects that reside on other systems and are possibly implemented in other languages.

The objective is to make as transparent as possible to the developer calling a local or a remote object.

We replace the client/server model with a client/object model.

In order to be able to access and object, you first must get an “object reference” pointing to it. Calling a local object through a pointer or a remote object through a reference are made to look the same. Task of the Object Middleware is to provide mechanisms to implement and retrieve the references for remote objects and to locate the objects over the distributed system.

We will see how this can be done in the examples.

Object Interface and OO concepts

- Interfaces: strong emphasis
  - Clients only look at interfaces
  - Interface Definition Language
  - Decoupling from implementation
- Encapsulation, inheritance and polymorphism

A client is only concerned with the “interface” of the objects it interacts with.

Object Middleware puts a strong emphasis on the concept of “interface”. Typically a language neutral Interface Definition Language is used to define interfaces and a strong decoupling between interfaces and implementation allows:

- One object to support many interfaces
- One interface to be implemented by many objects in different ways

The concept of interface provides a strong support for encapsulation.

Independent inheritance is typically supported on the side of interface definition and object implementation.

Polymorphism comes naturally from the separation between interfaces and implementation.

Actually, thinking in terms of separation between interfaces and implementation helps a lot in grasping the fundamental OO concepts. This is a major advantage of Java with respect to C++ from the language definition point of view (pure virtual declarations in C++ can be used to emulate interfaces, but are not conceptually equivalent).

Object Middleware Systems

- Common Object Request Broker Architecture (CORBA) from the Object Management Group (OMG)
- Java Remote Method Invocation (Java RMI) from Sun Microsystems
- Distributed Component Object Model (DCOM) from Microsoft

The software infrastructure for an observatory shall be built on top of an Object Middleware.

But there is no point in developing a new one: there are various available on the market and it is just a matter of picking the right one.

We list here just 3 dominant examples: CORBA, Java RMI and DCOM and we will base our examples on CORBA.

There are many parameters to drive the choice, depending on the requirements of the system under development:

•Open or closed standard

•Opens source versus commercial implementation

•Multiple vendors

•Market share

•Costs

•Number of architectures, operating systems, languages supported.

For ALMA we have decided to adopt CORBA because we think its characteristics make it the most suitable for the development of a software system for a large international and open collaboration in the scientific community:

•Very open standard, not controlled by specific vendors

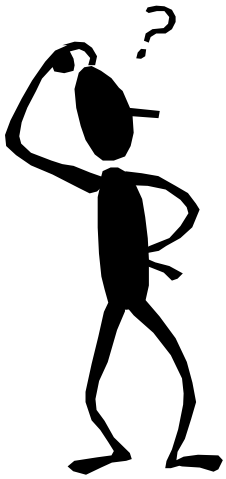
•Wide availability of high quality open source implementations

•Intrinsically operating system, architecture and language independent.

•Vendor interoperable by design, i.e. applications from different vendors will work together

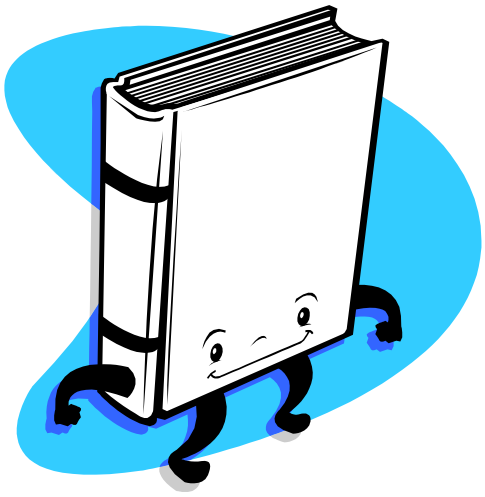
We are happy of this choice and we can state that CORBA maintains what promises.

Questions (& Answers)



A questions and answers session is the best way to clarify this choice before using CORBA as a practical example to describe more in details the characteristics of an Object Middleware.

4 – Introduction to CORBA



What is CORBA

- CORBA is a standard, not a product
- Common Object Request Broker Architecture (CORBA)
  - A family of specifications
  - OMG is the standards body
  - Over 800 companies
- CORBA defines *interfaces*, not *implementations*
- *CORBA core and services*

First of all we have to say what CORBA isn't: CORBA is not a product, but rather a standard specification. The OMG is the standard body for the specification of CORBA (and of UML) and is a consortium of the most important software vendors.

For this reason CORBA specifications define only interfaces and not implementation.

Any vendor should be able to provide an implementation based only on these OMG interface specifications and this implementation should be interoperable and usable together with the implementation of any other vendor, provided that both comply with the same specification level.

Reaching such an interoperability requires unambiguous and complete specifications. This typically requires various iterations. After some initial glitches (like the design of the Basic Object Adapter – BOA – later on superseded by the Portable Object Adapter) the core components of CORBA are now extremely stable and interoperable.

The latest additions and the higher level services still need some iterations to reach the same level of stability and interoperability.

In ALMA and the Alma Common Software we are currently using 5 different open source CORBA implementations:

•TAO for C++

•JacORB for java

•Omni ORB for Python

•Mico for its implementation of the Interface Repository service

•Open ORB for its implementation of an extensible IDL compiler used for code generation

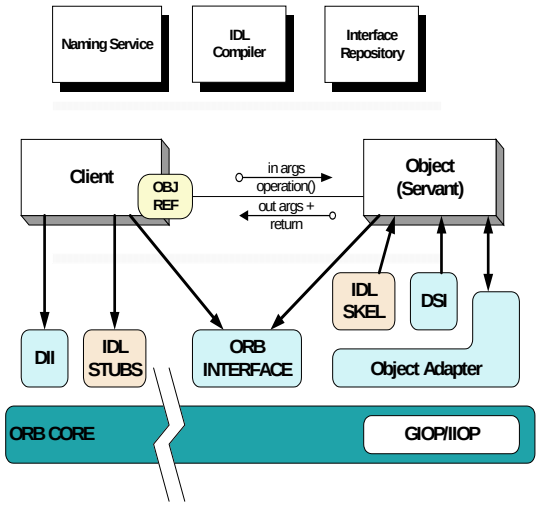
We have also used in the past Orbacus and the native Java JDK ORB and replaced them with one of the ORBs previously listed and we have switched the implementation of some services from one ORB to another, to use the one better fulfilling our needs. And we can use the documentation, manuals, books relative to any of them to learn how to use best another one. This is a very good demonstration of interoperability.

When we discuss of CORBA, we should always keep in mind two levels:

•CORBA core functionality is the set of basic components (like the IDL language, the Object Request Broker and the IIOP communication protocol) that warranty interoperability. Every implementation has to support them.

•CORBA services are additional (but often fundamental) services that are built on top of CORBA by the vendors that want to provide them.

Overview of CORBA



- It simplifies development of distributed applications by automating/encapsulating
  - Object location
  - Connection & memory mgmt
  - Parameter (de)marshaling
  - Event & request demultiplexing
  - Error handling & fault tolerance
  - Object/server activation
  - Concurrency
  - Security

This slide summarizes the overall architecture of the core CORBA.

From the logical point of view, a ***client*** get hold of the ***object reference*** of an Object it wants to talk to, for example using the ***Naming Service***, and then can invoke its operations.

The interface of the object is known to the Client via the ***IDL interface*** published by the object.

Under the hoods, the ***Object Request Broker (ORB)*** transports a client request to a remote object an returns the result. It is typically implemented as a set of client and server side libraries.

Interoperability is warranted by the ***General Inter-ORB Protocol (GIOP)*** and by its TCP/IP incarnation called ***IIOP***.

All vendors are bound to support IIOP but can also implement their own protocol, for example for performance optimization or to exploit specific hardware like ATM networks. There is also a standard secure protocol based on SSH. This allows to exploit network capabilities transparently do the application developers.

On the ***servant*** side the Object Adapter provides the environment in which servants live. In particular it takes care for:

•Mapping of object references into implementation

•Object life cycle

•Threading policy

Compiled interfaces are provided by the ***stubs*** and ***skeletons*** generated by the ***IDL compilers*** (more on this later).

Interpretative interfaces are handled through:

•***Interface Repository***. Repository of the IDL interfaces known to the system. Used for language independent introspection.

•***Dynamic Invocation Interface (DII)*** used on the client side to dynamically generate calls to object operations. Necessary for generic applications and for the implementation of CORBA inside interpreted languages.

•***Dynamic Skeleton Interface (DSI)*** used on the servant side to dynamically implement objects that incarnate a given IDL interface. Necessary for example to implement generic servants like protocol converters or object-to-relational database interfaces.

CORBA server characteristics

- When we say "server" we usually mean server process, not server machine
- One or more CORBA server processes may be running on a machine
- Each CORBA server process may contain one or more CORBA object instances, i.e. "servants", of one or more CORBA interfaces
- A CORBA server process does not have to be "heavyweight"
  - e.g., a Java applet can be a CORBA server
- Clients always talk explicitly to *servants*, and not to *servers.*

When we consider the traditional client-server model, we think of a "client process" requesting a service from a "server process" (or, sometimes, a server machine).

Obviously, also with CORBA the communicating entities are processes running on distributed hosts, but the communication abstraction is higher level.

A CORBA process providing a service to a client contains one or more CORBA object instances, called "servants".

Each servant implement one or more CORBA IDL interfaces and clients do address and communicate explicitly with the servants.

The "server" is only the process inside which the "servant" lives and the client is not aware of that.

Clients always talk explicitly to the servants using the object reference in a fully object oriented model.

Deployment of "servants" in "servers" can be dealt with in a way completely transparent both to clients and servants themselves, as we will see later on.

As we have seen and we will see with more details later, CORBA also uses the term "service" to denote fundamental, almost system-level services to OO applications and their components. Services are specified by means of interfaces, implemented by "servants" and deployed within "servers".

Therefore, make sure to keep always in mind the difference between:

•Servant

•Server

•Service

Interoperable Object Reference (IOR)

- An Interoperable Object Reference is the distributed computing equivalent of a C++ pointer:
  - An IOR uniquely identifies one object instance
  - IORs can be uniquely mapped into a string and back for easy and portable storage.
- An IOR contains:
  - A fixed object key with the fully qualified interface name and an instance identifier
  - Transient information such as the host and port of its server
- An IOR can be persistent
  - Some CORBA objects are transient, short-lived and used by only one client
  - But CORBA objects can be shared and long-lived
- CORBA objects can be relocated
  - The fixed object key of an object reference does not include the object's location
  - CORBA objects may be relocated at admin time or runtime
  - ORB implementations may support the relocation transparently
- CORBA supports replicated objects

Servants are addressed by means of their object references or, more specifically, by their Interoperable Object Reference (IOR).

An IOR uniquely identifies one object instance, I.e. it allows to locate the object in the network and identify what interface it implements.

Interoperability is warranted by representing IORs as strings that can be easily transported and used on heterogeneous systems.

CORBA object references can be persistent.

Some CORBA objects are transient, short-lived and used by only one client.

But CORBA objects can be shared and long-lived

> business rules and policies decide when to "destroy" an object
>
> IORs can outlive client and even server process life spans.This means that once a client has obtained the IOR for an object, it can continue to use it also after a restart of the server, unlike a normal C++ pointer.

CORBA objects can be relocated

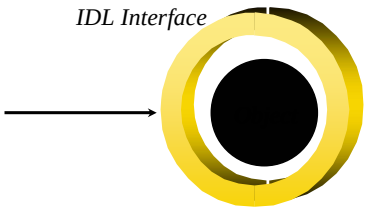
> The fixed object key of an object reference does not include the object's location
>
> CORBA objects may be relocated at admin time or runtime
>
> ORB implementations may support the relocation transparently

CORBA supports replicated objects

> IORs with the same object key but different locations are considered replicas. The same IOR can contain "alternative solutions" for getting in contact with the desired servant.

The flexibility of the IOR specification is one of the keys to CORBA interoperability and scalability.

Interfaces vs. Implementations



CORBA Objects are fully encapsulated

Accessed through well-defined interface

Internals not available - users of object have no knowledge of implementation

Interfaces & Implementations totally separate

For one interface, multiple implementations possible

One implementation may be supporting multiple interfaces

The absolute separation between interface and implementation is another of the CORBA keys to interoperability and scalability.

But it is often cause of confusion for developers approaching CORBA for the first time.

The clients of a servant know and see only its interface and the interface shields completely the implementation underneath.

This makes it possible first of all to implement a servant in any language supporting a CORBA mapping.

But it also means that it is possible:

•To have different implementations for the same interface, if needed in multiple languages. For example one could provide a mock up implementation in Python for testing and an high performance servant in C++ for the final real time system.

•To have one implementation serve multiple interfaces.
For example, access to legacy system could be done defining the interfaces for each subsystem but implementing only one generic servant (for example a sort of protocol converter) able to implement all of them. Another example is a CORBA interface to access a object (or also relational) database. It is not necessary to provide the implementation for each object type (or table) in the database. One single implementation is able to “incarnate” dynamically all interfaces.

•To have one physical instance of a Servant to represent multiple logical instances. Or the other way around. Or any intermediate situation, based on scheduling and load balancing algorithms.

Interface Definition Language: IDL

- CORBA interfaces are defined using the IDL definition language

  IDL forms a 'contract' between client and servant.
- IDL reconciles diverse object models and programming languages
- Imposes the same object model on all supported languages
- Programming language independent means of describing data types and object interfaces
  - purely descriptive - no procedural components
  - provides abstraction from implementation
  - allows multiple language bindings to be defined
- A means for integrating and sharing objects from different object models and languages

CORBA specifies an Interface Definition Language.

Each language supporting CORBA has a formal mapping from the language to IDL and the other way around.

There are for example mappings from and to IDL for C, C++, Java, Python, TCL and many others.

This has the great advantage of reconciling different object models and programming languages allowing them to easily inter operate.

But on the other hand forces some compromises, in particular since some of the supported languages are not object oriented.

For example IDL supports interface inheritance (actually, multiple inheritance) but does not allow overloading, i.e. it does not allow multiple operations with the same name but a different signature. This would not be possible to implement in an easy and intuitive way in the C mapping.

But when thinking about the IDL language definition, it is important to think that is is a language to define INTERFACES and not IMPLEMENTATION. For example there is no point in implementing such things as:

•private operations or attributes
because what is private in Java or C++ is "hidden inside the implementation" and, therefore has no place in the interface

•operation overriding, i.e. redefining the same operation in a sub-class with a different behaviour, since the behaviour is again domain of the implementation.

It is quite common to confuse interface and implementation aspects.

But there are some important limitations in the IDL specification.

IDL Example: Mount

```
#ifndef _ACSCOURSE_MOUNT_IDL_
#define _ACSCOURSE_MOUNT_IDL_

#include <baci.idl>

#pragma prefix "alma"

module ACSCOURSE_MOUNT
{
   interface Mount : ACS::Component
                {
                void objfix (in double az,
                                in double elev);
                attribute long status;
                readonly attribute ACS::RWdouble cmdAz;
                readonly attribute ACS::RWdouble cmdEl;
                readonly attribute ACS::ROdouble actAz;
                readonly attribute ACS::ROdouble actEl;
                };
};
#endif
```


This slide shows a small example of IDL file, representing an abstract (an extremely simplified) telescope mount.

This IDL file defines an interface called Mount that provides:

•one operation called objfix(az, el) to send the mount to a specified azimuth and elevation

•one attribute of type long for the status of the mount

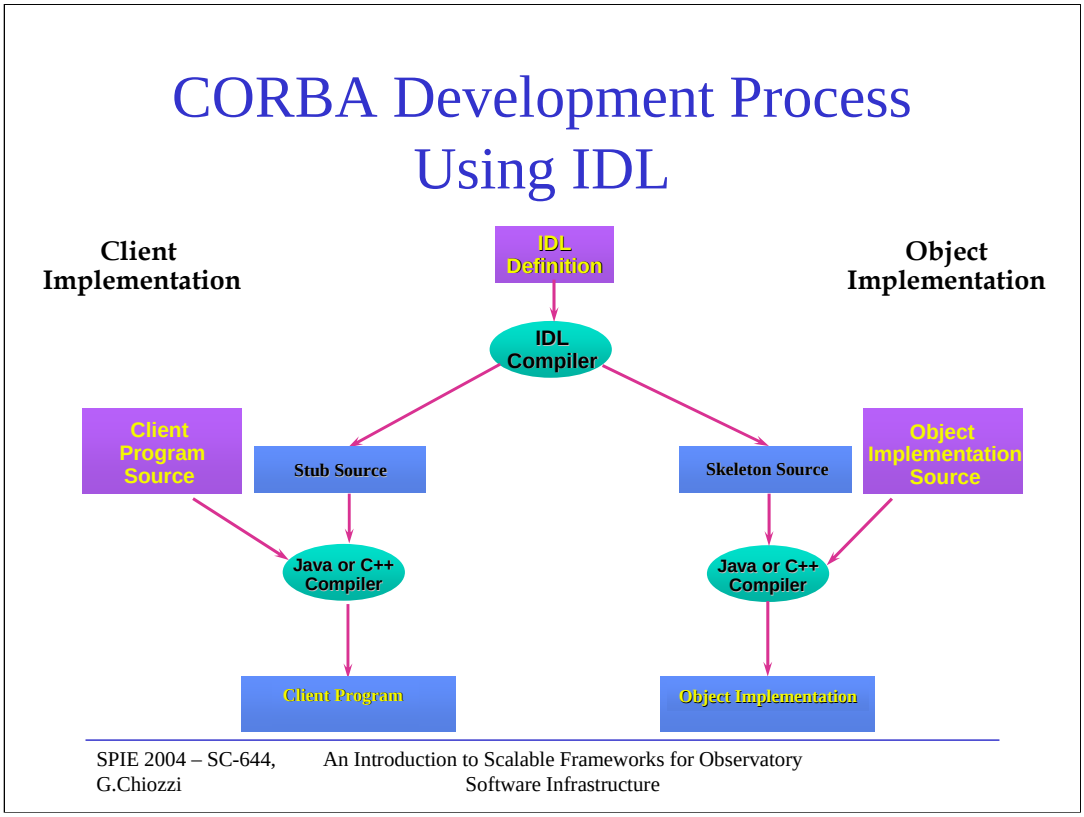
•a few attributes that are complex CORBA objects themselves

As you can see, the structure of IDL is very much taken from C++ include files and even relies on the standard C pre-processor (although some non C++ ORBs like JacORB only provide in practice limited support for pre-processing directives).

The IDL syntax only allows to formally define signatures for operations and attributes with types and names of parameters.

There is no formal provision for describing characteristics of a parameter like ranges or more in general a formal “design by contract” specification.

This would be actually very interesting and IDL could be augmented with comments describing such constraint using OCL (Object Constraint Language, still part of OMG “products” together with UML).



This diagram shows how an IDL definition is used in the code development process.

•The IDL file is processed by one or more IDL compilers

•Each IDL compiler can produce:

- •Stub code
  The stub code is the code that a client has to call to invoke a remote CORBA object.
- •Skeleton code
  The skeleton is the code that has to be used as the bases for the implementation of the servant.

So the development process can be seen through the following steps:

•The IDL interface is defined in agreement between the developers of servants and clients and published as the contract between them.

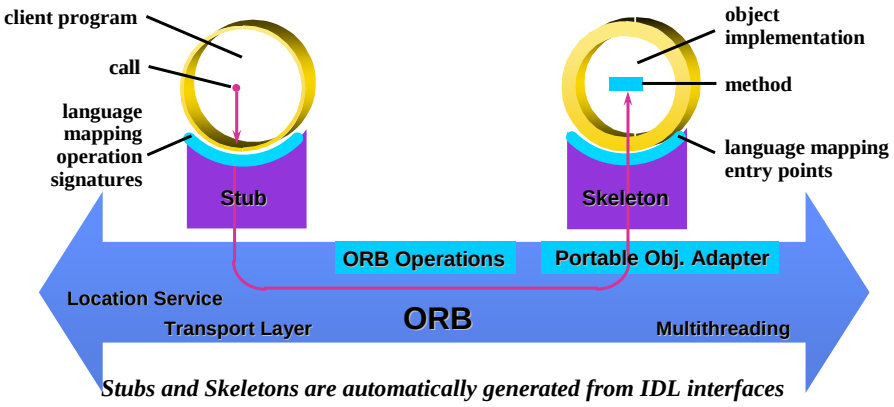
•The developer of the servant:

- •chooses an implementation language (for example C++)
- •selects a CORBA implementation for that language (for example TAO)
- •runs the agreed IDL through the IDL compiler and obtains a Skeleton. In the C++ case, this is an abstract C++ class mapping all operations and attributes of the class into abstract methods
- •subclasses the skeleton and implements all abstract methods

•The developer of the client:

- •chooses an implementation language (for example Python)
- •select a CORBA implementation for that language (for example OmniORB)
- •runs the agreed IDL through the IDL compiler and obtains a Stub. In the Python case, this is a python class mapping all operations and attributes of the class
- •simply calls the stub methods.

Stubs & Skeletons



Stubs and Skeletons are automatically generated from IDL interfaces

The Stubs and the Skeletons contain all the code needed to interface the user code with underlying ORB and CORBA machinery.

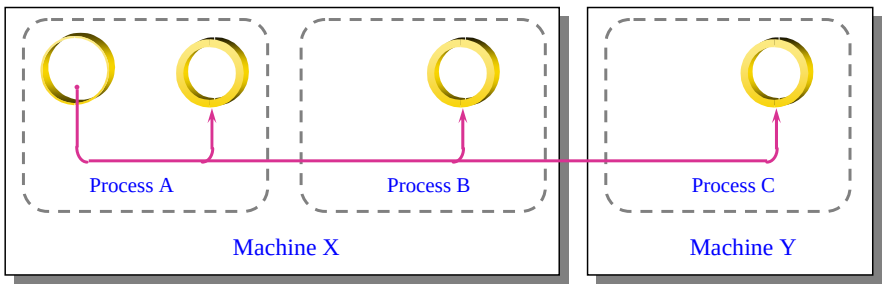
Often the code in the Stubs and Skeletons is ORB dependent and you cannot normally use the code produced by the IDL compiler of one CORBA implementation with the ORB libraries of another one, but this is not important because:

•the interfaces of Stub and Skeleton are based on the formal IDL to language mapping and therefore the user code does not change changing ORB (unless you use vendor extensions)

•the communication between ORBs is also interoperable (unless you use vendor extensions)

This allows to mix and match CORBA implementations based on your needs and to replace them with others.

Location Transparency



A CORBA Object can be local to your process, in another process on the same machine, or in another process on another machine

In order to invokes operations on a servant, a client uses the CORBA reference to obtain a local Stub object (I.e. an object in its own language and instantiated in its own process).

Then it makes native language calls to the Stub.

The Stub and the underlying ORB map these calls into calls to the real Servant, but the client is not aware of where the Servant resides.

It can be a local object as well, an object in another process on the same machine or an object in another host.

There is some overhead in this mapping, but good ORB implementations make this overhead minimal and calls to local Servants can be reduced to a few levels of indirection, avoiding any real inter-process communication.

But this transparency makes it much easier to scale systems and optimise performance by re-deploying Servants on separate processes and hosts or repackaging together Clients and Servants that have frequent interactions.

Summary: requirements and core CORBA

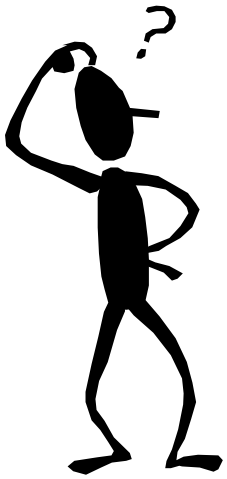
- No need to pre-determine:
  - The programming language
  - The hardware platform
  - The operating system
  - *The specific object request broker*
  - The degree of object distribution
- *Open Architecture*:
  - *Language-neutral* Interface Definition Language (IDL)
  - Language, platform and location transparent
  - *Interoperable*
- *Objects could act as clients, servers or both*
- *The Object Request Broker (ORB) mediates the interaction between client and object*
- *Scalability designed in IOR and ORB specifications*

Here I summarize once more the main design goals and characteristics of CORBA.

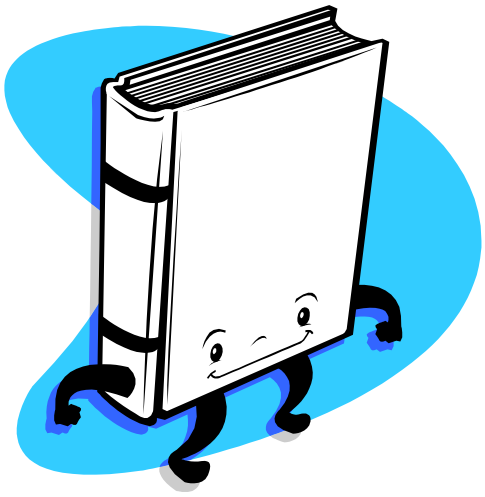
Many of these items have been listed already as common to all major object middleware technologies, but I have marked in colour (italics, for B&W print) the ones that are more specific of CORBA and that many “competitor” technologies do not take into account.

These goals map well with the requirements that have led us to identify the need for adopting a Middleware.

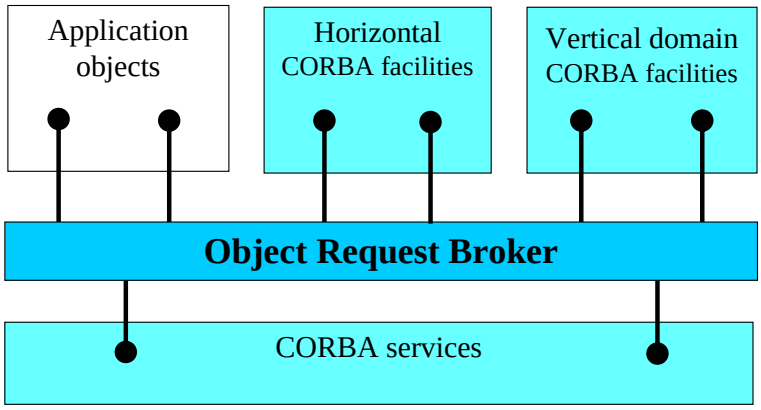
Questions (& Answers)



5 – Services and facilities



Object Management Architecture



The OMG has defined on top of the core CORBA architecture an Object Management Architecture (OMA) with the purpose of providing an architecture and interoperability foundation to allow the development of plug-and-play software.

The basic idea is that when applications provide basic functionality, they shall provide it via standard interfaces.

In this way:

•Multiple, interchangeable implementations of the same functionality can be interoperable but still be characterized by differences in performance, price or adaptation to run on specific platforms.

•Specialized high level components, developed independently and for different purposes, can still be made interoperable because they use the same palette of basic building blocks (interfaces).

Applications - even if they perform totally different business tasks - share a lot of common functionality: objects notify other objects when something happens; object instances are created and destroyed and new objects' references are passed around; operation must be made secure and transactional. Beyond this, applications within a business domain (telecommunication, transportation,…) share even more functionality. The OMA abstracts out this common functionality from CORBA applications into a set of standard objects that perform standard, clearly-defined functions.

As it has been discovered at a high price in the past years, it is not sufficient to write software using object oriented techniques or in any case specific languages to make it reusable and interoperable with other software. Two pieces of software can work together only if they expectations on the environment they want to live in are compatible. Just like two IC chips can live on the same motherboard only if they expect the same kind of power supply.

The OMA defines:

•CORBA Services (COS)
Specify basic services that almost every object needs. This part of the OMA started first and is quite well developed and supported

•Horizontal facilities
Provide intermediate level services common to all applications. They can substantially help to develop applications in any domain but are not strictly necessary.

•Vertical domain facilities
Are specifications for services useful in specific application domains and are defined by Domain Task Forces inside the OMG with focus on a particular application domain, such as telecommunication, Internet, manufacturing and so on. There is an OMG interest group on real time control and there could be one on Astronomy or, more in general, experimental facilities. Some of the facilities developed here have found a widespread usage very well outside the original application domain.

This distinction is useful to clarify who inside the OMG is responsible for the specification of a service or of a facility.

But from the point of view of users of services and facilities it is not really important and there are now vertical domain facilities (like the Telecom Notification Service) that can be actually considered for any purpose of usage plain CORBA services.

Therefore in the coming pages I will not distinguish and only talk in general terms of CORBA services.

CORBA services

- Defined on top of the core CORBA
- OMG defines IDL interfaces and semantic to services in order to ensure interoperability
- To be implemented as standard CORBA objects by vendors
- Vendors choose what services to implement

The services are defined on top of the ORB.

They are defined by means of formal specification documents that include IDL interfaces and semantic description in English text. They shall be implemented as CORBA Objects (or appear as internal CORBA Objects, I.e. CORBA objects that are not accessible from outside the process but only to the local objects).

The vendors or CORBA implementations are free to choose what services they want to implement. But if they implement a service, they are bound to implement it according to the specifications. Some widespread services are implemented by every vendor, but some other are extremely specific and seldom implemented.

But it is important to notice that it is in many cases possible to select any implementation of a service and use it with another ORB, thanks to the fact that interfaces are through IDL and the interoperable CORBA communication bus.

CORBA services

- Naming Service
- Event Service
- Notification Service
- Logging Service
- Audio/Video Streaming Service
- Life Cycle Service
- Concurrency Control Service
- Time Service
- Property Service
- Persistent State Service
- Security Service
- Trading Service
- Transaction Service
- Query Service
- Relationship Service
- Externalization Service

What follows is a list of CORBA services with a brief description:

Naming Service -- Supports both persistent and non-persistent hierarchical mappings between sequences of strings and object references. In addition, the Interoperable Naming Service defines a standard way for clients and servers to locate the Naming Service, as well as any other CORBA service.

Event Service -- Supports decoupled communication among multiple suppliers and consumers using the standard GIOP/IIOP protocol.

Notification Service -- Is a more powerful form of the Event Service that supports filtering and correlation.

Logging Service -- Allows applications to send logging records to a centralized logging server.

Audio/Video Streaming Service – Defines a model for implementing an open distributed multimedia streaming framework.

Lifecycle Service -- Provides a standard means to locate, move, copy, and remove objects.

Concurrency Service -- Provides a mechanism that allows clients to acquire and release various types of locks in a distributed system.

Time Service -- Provides globally synchronized time to distributed clients.

Property Service -- Supports the association of name-value pairs with CORBA objects.

Persistent State Service -- Provides a way to make a service persistent. PSS presents persistent information as storage objects that reside in storage homes.

Security Service – Provides identification and authentication of users and objects, authorization and access control, security auditing, security of communication between objects.

Trading Service -- Implements a mapping between attribute constraints and sequences of object references that match those constraints. Therefore it supports the finding of CORBA objects based on properties describing the service offered by the object

Transactions -- Coordinates atomic access to CORBA objects

Query -- Supports queries on objects

Relationships -- Provides arbitrary typed n-ary relationships between CORBA objects

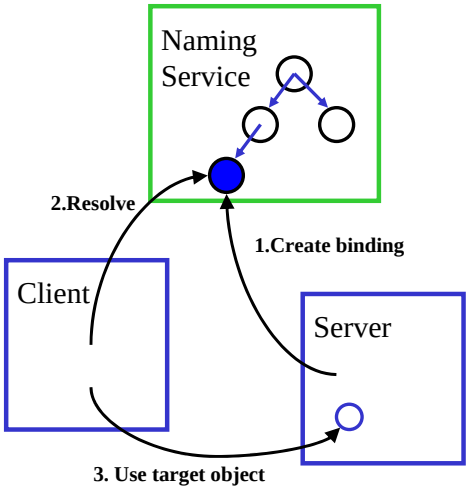
Externalization -- Coordinates the transformation of CORBA objects to and from external media

TAO for example implements most of these services, but other vendors implement only a subset.

In the following pages we will look at some examples with more details, with the purpose of understanding what Services are and what they can bring to the developers.

Naming Service

- maps logical names to server objects
- references may be hierarchical, chained
- returns object reference to requesting client
- allows federation



The naming service is a simple locating service that allows clients to look up an object location using a name as a key. The name can be specified in a human-readable stringified name format or in a raw name format. Typically, a tree-like directory for object references is used, much like a file system provides a directory structure for files.

Before a client can look up an object, the association between the object location and its name must be created. This association is known as an *object binding*, and it is normally made by a CORBA server.

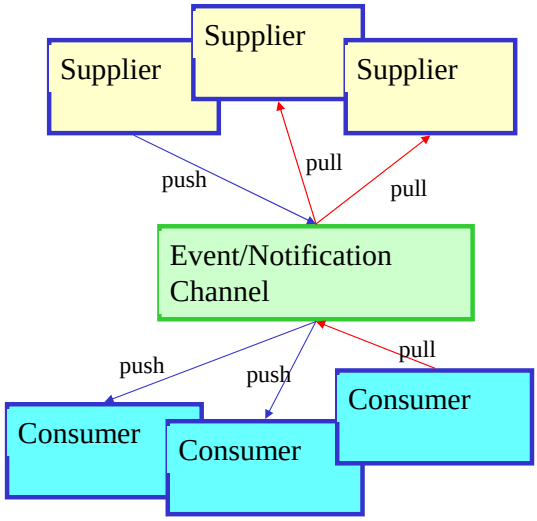
Then a client can *resolve* the name asking the Naming Service by name and receiving back the reference to be used.

The Interoperable Naming Service (INS) is a URL-based naming system on top of the CORBA Naming Service, as well as a common bootstrap mechanism that lets applications share a common initial naming context.

Naming Services can be federated. A federated service provides a single logical service to clients, but consists of a number of physical servers. This allows scalability and redundancy of the system.

Events and Notification services

- Asynchronous messaging
- *Publish/subscribe* paradigm
- Decouple suppliers and consumers of information
- Push and pull models
- Notification adds to Event Service:
  - Filtering
  - Structured Events
  - Sharing subscription information
  - QoS properties



The standard CORBA operation invocation results in synchronous execution:

•Both client and servant must be active

•The client blocks until the operation returns

•Communication is point-to-point

For many application it is required to have asynchronous communication, eventually with multiple suppliers and consumers.

The Event Service provides a model for asynchronous communication based on the "publish/subscribe" paradigm with an *Event Channel* that plays the role of a mediator between suppliers and consumers of events and encapsulates the queuing and propagation semantics.

Some examples are:

•A telemetry system where telemetry data is published and displayed on many consoles, on top of being archived in a central database.

•An alarm system, where alarm conditions can be published by many objects and need to be collected in a central service and dispatched again to many clients.

•Synchronisation events emitted by one object and used to synchronise the action of many other objects. For example a "target reached" event used to start exposure and data collection.

The Notification Service is mostly an extension of the Event Service, but provides very important features.

Filtering is extremely important, because without that the Event Service is actually a broadcast mechanism: all subscribers receive all events published on the channel and have to select themselves the ones they are really interested in.

Logging service

- Logging is fundamental for distributed systems, but complex:
  - Persistent log records
  - Log record filtering
  - Log forwarding for scalability and federation
  - Support for QoS associated with the Notification Service.
  - Administration interfaces
- Based on Notification Service
- Implements CCITT X.735 recommendation

A centralized logging system is essential for the development, monitoring and administration of a distributed system.

Events happen in many different hosts and processes and need to be correlated to be able to understand the inter-relationships between things occurred in different places.

Therefore developers want to be able to log actions and events and collect them in a central place.

It must also be possible to store this information persistently for later analysis.

Also “telemetry” information about the behaviour of the system has the same typical life cycle.

A logging system is really a common service needed by any application and is also very complex if we take into account the requirements for scalability and reliability.

The OMG Telecom working group has defined a logging service supporting the CCITT X.735 recommendation and base on the CORBA Notification Services that is now widely used also outside the Telecom vertical domain and has been implemented by various vendors.

Scalability is based on forwarding specifications that allow log objects to forward messages one to the other building hierarchies and redundant nets.

Quality of Service specifications and a thoroughly defined Administration Interface take care of the reliability requirements.

Summary: CORBA Services

- OMG defines standard specification for common services that go across domain and application borders
- Vendors provide implementation of the service
- These two steps guarantee to the user:
  - Scalability
  - Reliability
  - Interoperability
- Pick the right service for your requirements:
  - Asynchronous communication
  - Security
  - Persistency
  - Load balancing
  - ……

Here I summarize once more the main design goals of CORBA Services.

These goals map well with the requirements that have led us to identify the need for adopting a Middleware.

By picking the right services from the palette of available specifications we can find a ready made solution for wide sets of application requirements.

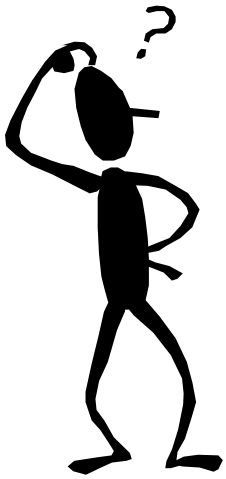
The fact that the services have been specified at OMG consortium level guarantee that they have been thoroughly thought and are coherent and consistent.

Although often the specification appears complicated and over killing, implemented in house what provided by a Service results very often in over simplification of the problem and under estimation of the requirements with the result that it is often necessary to radically extend and change the architecture of the “home brewed” service during development with inconsistent and often not scalable and unreliable results.

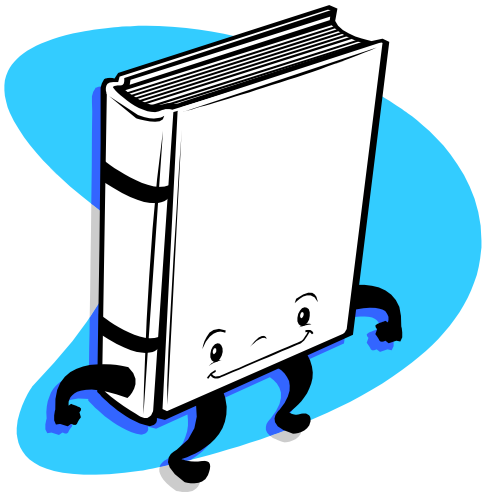
The general (but not absolute) interoperability allows to select the implementation that better satisfies the requirements and there are often available “light implementations” that are simpler and thinner than the full blown implementation at the expense of features (sometimes unneeded in the application domain).

Even in the case where it is not possible to use an existing implementation, it is often very productive to start from the OMG specification and take it at the basis for a home made partial implementation.

Questions (& Answers)



6 – From Object to Component Middleware



Object Middleware shortcomings

- Explicit programming of non-functional properties
- No standard configuration, packaging and deployment facilities

Weak “separation of concerns”

Steep learning curve

The Object Middleware model described in the previous pages has great advantages with respect to the previous approaches.

But experience has also shown some important shortcomings and alone it does not fulfil what it promises.

First of all, using CORBA and services still requires a lot of non-functional programming:

•CORBA needs to be initialized and shutdown

•Services need to be as well initialized and have a wide access API

Middleware specific code appears in many places, so the developers need to take into account (and learn) the functioning of their Middleware.

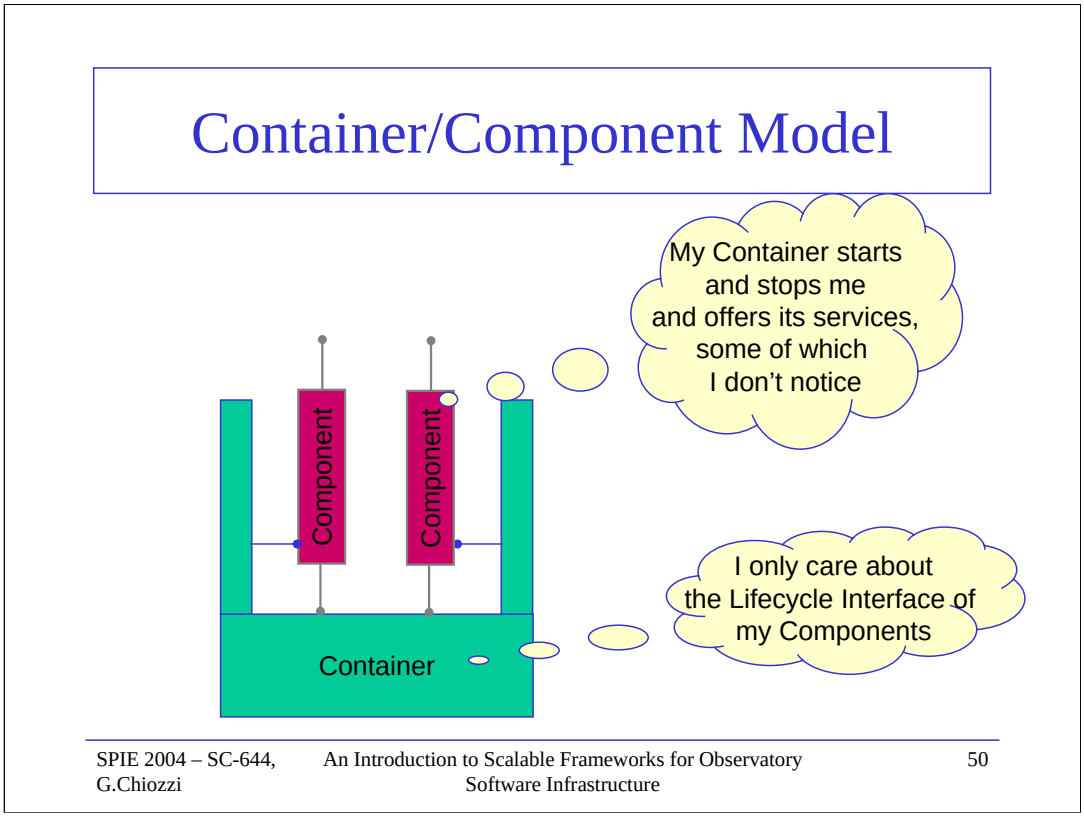
Also the Object model focuses on the objects themselves and not a more global view of the system, i.e. how they are configured, packaged together and deployed. The developers of applications have to take care of this aspect, again mixing functional and technical aspects of the architecture.

The Object Middleware model leaves to the full responsibility of the developers the technical architecture of the system.

We have all the elementary building blocks to build our application, but we still need to find out what of these building blocks we really need in our specific case and HOW to put them together.

All this weakens our capabilities of keeping the desired “separation of concerns” between functional and technical aspects and leads to a steep learning curve for the developers of functional objects, because they still need to learn a lot of the technical aspects of the otherwise powerful and complete middleware.

What we really need is a ***framework***, i.e. a semi-complete application that based on a well specified architecture gives us a general skeleton to support our business logic. Clearly, the framework must be able to satisfy the requirements of our application domain, but at the same time be as close as possible to a “finite” system, since higher flexibility is almost always associated to higher complexity.



Our objectives of separation of functional and technical concerns can be reached "upgrading" from an "object" to a "component" Middleware and adopting a Component/Container Model.

A Component Middleware:

•Creates a standard "virtual boundary" around application component implementations. Functional developers are only concerned by the implementation of their Component code.

•Defines standard Container mechanisms needed to execute Components in generic Component Servers. The Container provides the whole execution environment and access to services for its Components.

•Specify the infrastructure needed to deploy and configure components in a distributed system. Components can be re-configured and moved around in the system without affecting the Component itself or other clients or servants. Configuration and deployment become a separate concern from Component development.

Consequences of this approach for functional developers are:

•Easier learning curve and reduced skill requirements: focus expertise on domain problems.

•Scalability taken care of by the Container and tuneable at deployment time

•Better adaptability and maintainability, with general reduced complexity.

Component/Container: buy vs. build

- Major Component models:

  .NET, EJB, CCM

  - .NET binds to Microsoft platform
  - EJB binds to Java programming language
  - CCM is still immature and there are no reliable free implementations. Implementations are not interoperable.
- Off-the-shelf Component Container implementations require a wholesale commitment from developers to use the languages and tools supplied.
- Focus for these Component/Container implementations are big enterprise business systems
- For ACS we decided in 2000 to go for a custom Component/Container implementation. Recent investigations confirm this choice.
- We aim at staying as much a possible compatible with CMM concepts to allow adopting an implementation, when available.

Commercial implementations of the Component-Container model are now quite popular (EJB, .NET).

A vendor-independent specification, the CORBA Component Model (CCM), is part of CORBA 3 specification, but it is not yet widely used.

There are some free implementations (for example from TAO and MICO), but they are not interoperable and are limited to one single language. The specification still needs to be refined on the basis of the experience with the first implementations and interoperability will come as well.

These Component/Container model are rather comprehensive systems, and require a wholesale commitment from developers to use the languages and tools supplied. In particular,

•.NET binds you to develop in and for the Microsoft world

•EJB binds you to the Java programming language

Once more, only the CORBA CCM really promises vendor, platform and language independence.

At the same time, the focus of these models is on big enterprise business systems and they contain a lot of features that are not needed for our observatory and, more in general, experimental facility environment.

For these reasons when we started to develop ACS in 2000 we decided for a simple custom Component/Container model (that we actually inherited from the work done for the Control System of the ANKA Synchrotron). At that time, CCM was not even a complete specification and there were no implementations available.

Recent investigations done by other teams have confirmed that this decision, taken back in 2000, is still justified.

We keep in any case an eye on the evolution of the CCM and we try to keep as much as possible our system aligned with the CCM concepts, to be able to switch to an implementation at acceptable costs.

The choice of developing a custom Component Model is a typical example where the analysis of advantages and disadvantages between generic and custom implementations has made us decide for the custom solution.

While, in principle, general solutions should always be preferred, our custom implementation has the advantage of being:

•Interoperable

•Lighter and with a smoother learning curve

•Easier to customize to the specific needs of our application domain

In the next pages we will describe the ACS model, but most of what said is applicable to CCM and in abstract to the other Component Container models.

Component

- Developed by Application developers
- Deployable unit of software
- **Focus on functionality with little overhead for remote communication and deployment**
- 1…many Components per subsystem
- Functional interface defined in IDL
- Deployed inside a Container
- Well-defined lifecycle (initialization, finalization)

A Component is the basic deployable unit of software.

It encapsulates a coherent and consistent set of application “business” login functionalities, defined and exported to clients by means of IDL interfaces (and textual descriptions for semantic and constraints).

The designer of a subsystem identifies the functionalities to be implemented and partitions them in 1 or more Components.

This partitioning is based on a logical view of the system and leaves out to a large extent deployment considerations and technical issues.

The interfaces between the Components of the subsystem and the external clients are defined by the IDL interfaces in a formal way.

Notice that the IDL interfaces can be used to implement generic or customized simulators effectively helping in decoupling the development/testing of Components inside the same subsystem or different subsystem.

Once a client has the agreed IDL interface of a component it needs to interact with, it can use a simulator or a mock-up to test its own component to a great extent without having to wait for the implementation of the counterpart.

Components will be deployed inside Containers and therefore will have to satisfy a few specific conditions, in particular about the life cycle, imposed by the need of living inside the Container.

Container

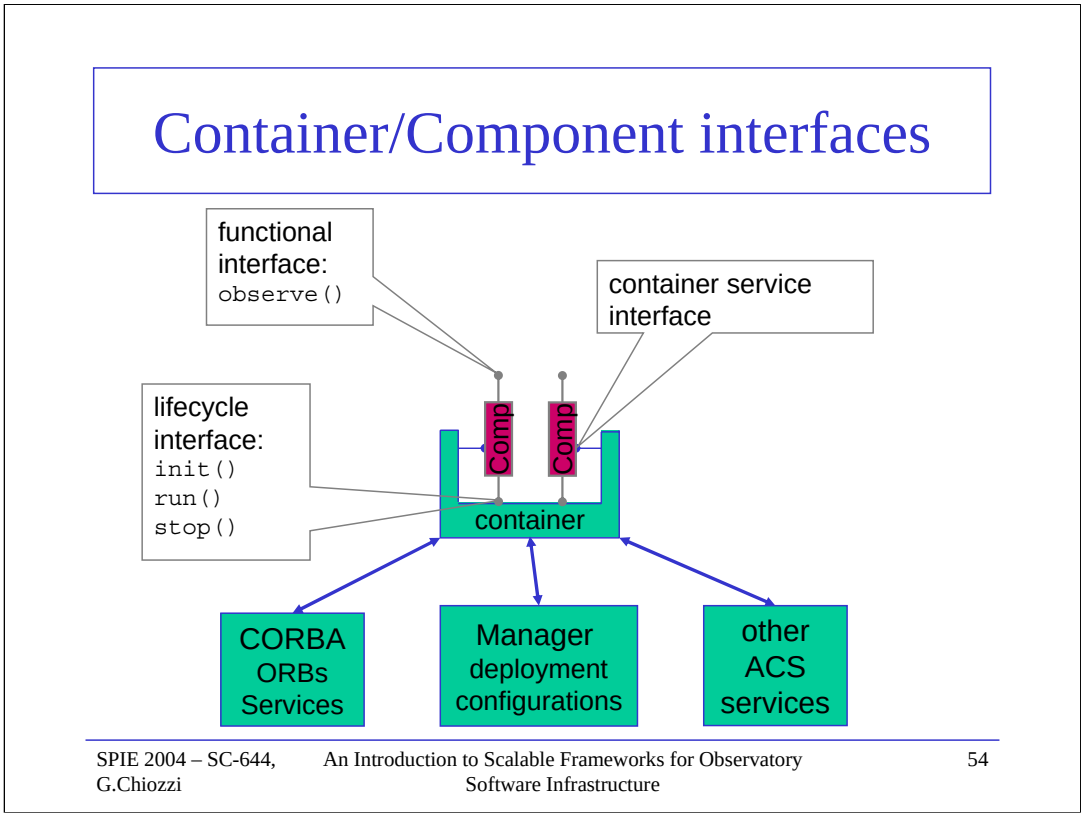
- Developed by the technical framework team
- Centrally handles technical concerns and hides them from application developers
  - Deployment, Start-up
  - Selection and configuration of various ORBs; here CORBA alone is much too complicated.
  - Selection of CORBA Services, integration with other application specific Services (Error, Logging, configuration, …)
  - Convenient access to other Components and resources
- New aspects can be easily integrated in the future, w/o modifying application software

The Containers are generic applications that are implemented by the team responsible for the technical framework, i.e. for the implementation of the Component/Container Model.

They provide the execution environment for the Components and hide from the application developer issues related to deployment, start-up initialization of CORBA and the services as well as convenient access to other Components and system resources.

In principle it would be possible to completely replace the core of the technical framework (for example replacing CORBA with some newer middleware) simply re-implementing the Container.

Also, new aspects (for example security or command parameter checking) can be easily integrated at the Container level without modifying the application software.



This diagram shows the relations between Components and Containers.

First of all, a Component provides a ***functional interface*** to other Components.

This is the part of the Component that the application developer needs to implement to satisfy its own application requirements.

In ACS the functional interface is specified through a standard CORBA 2 IDL interface.

The CORBA 3 CMM specification extends the IDL syntax to allow specifying also:

•What interfaces the component uses, providing therefore a bi-directional specification of the relationships between Components.

•What *events* a component *publishes*

•What events a Component *consumes*

With these very useful extensions with respect to CORBA 2 IDL specifications, Component specifications can explicitly express the connections that they offer to the outside world AND what connections they expect the outside world to offer them.

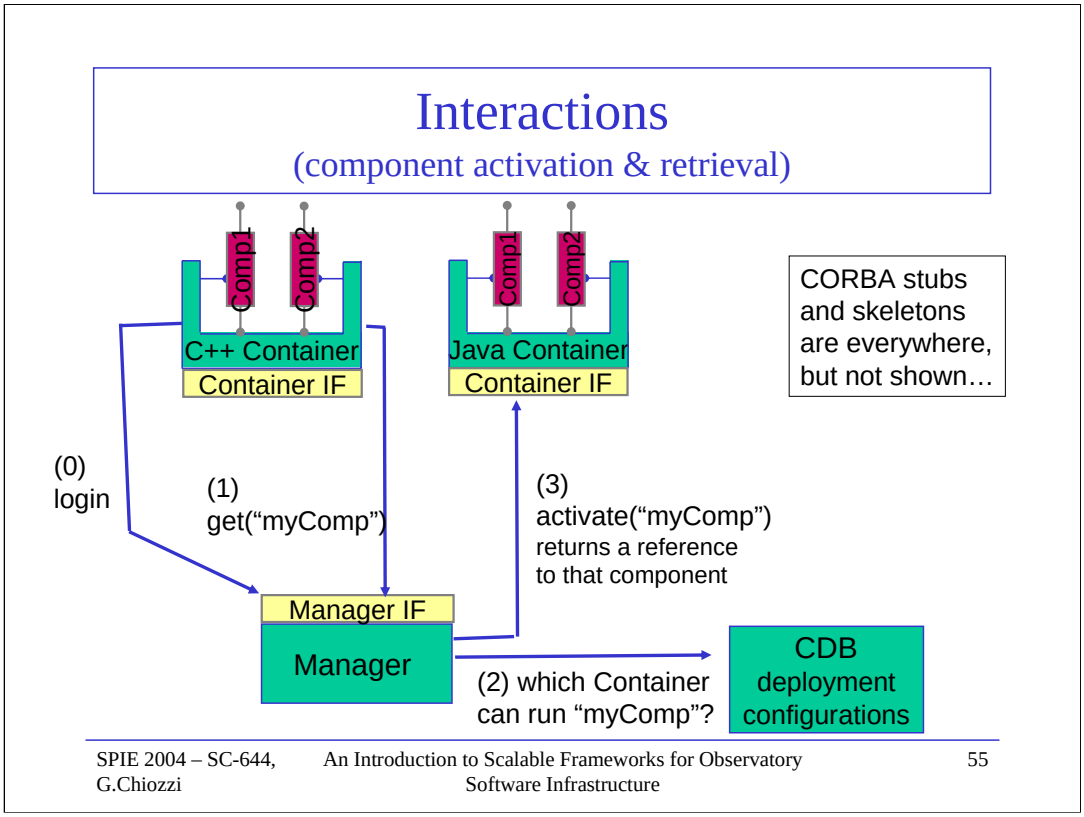
The Container hides as much as possible CORBA and the underlying architecture to the developers of Components, that can concentrate on the functional aspects of their specific Component.

We expect to extend the ACS Component/Container to handle CORBA 3 IDL specifications.

Then the Component is bound to implement a ***lifecycle interface***. In most cases the application developer can simply adopt a default lifecycle behaviour by inheritance or delegation from default Component implementations provided by the Framework.

The division of responsibilities between components and containers enables decisions about where and when individual components are deployed to be deferred until runtime, at which point configuration information is read by the container. If the container manages component security as well, authorization policies can be configured at run time in the same way.

Finally, Containers provide an environment for Components to run in, with support for basic services like logging system, configuration database, persistency and security. A ***container service interface*** is defined by the Container for the benefit of Components to access these services. Developers of Components can focus their work on the domain-specific “functional” concerns without having to worry about the “technical” concerns that arise from the computing environment in which their components run.



The ACS model includes also a **Manager** entity that centralizes deployment configuration, book keeping and system monitoring functionality.

Keeping these functions outside the Container helps significantly in making the model interoperable and language independent, since the Container themselves are simpler and can be therefore easily implemented when a new language or ORB needs to be integrated in the implementation of the Model.

This diagram shows how Components,, Containers and the Manager interact.

When a Container becomes alive, it registers itself with the Manager.

The manager is responsible for keeping a runtime image of the system deployment and for monitoring that all entities are in an healthy condition.

Manager and Container interfaces are also described through IDL interfaces and therefore their implementation language and platform are irrelevant.

Whenever a Component needs another Component, the request goes to the Manager that takes care (using standard CORBA services, like the Name Service) to locate where and if the Component needs to be deployed and, in case, dynamically deploy it and return the reference to the caller.

The Manager takes care also of de-activating Components that are not needed any more in the system.

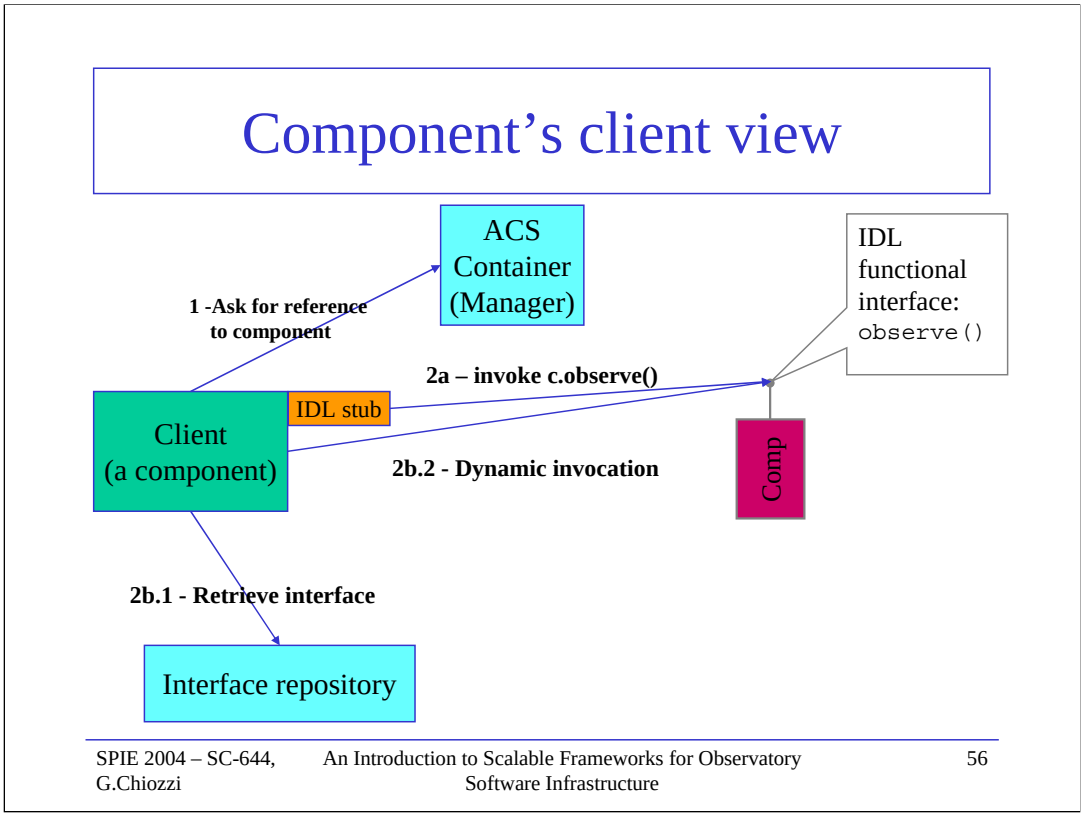
At this point the requesting Component can access its counterpart as needed and be notified by the Manager in case of problems.

The diagram shows how the communication between Components is set up and how it takes actually place.

But from the logical point of view, it is better to keep two separate views:

•The functional view or the application developer implementing the Component

•The technical view of the administrator responsible for configuration and deployment of the system



First of all let's see the system from the point of view of a Component that need to communicate with another Component.

A Component exposes its IDL interface to clients.

A client (possibly itself a Component) that wants to access the component, needs to ask for a reference. The request is done using the Container Services and is internally and transparently redirected to the Manager.

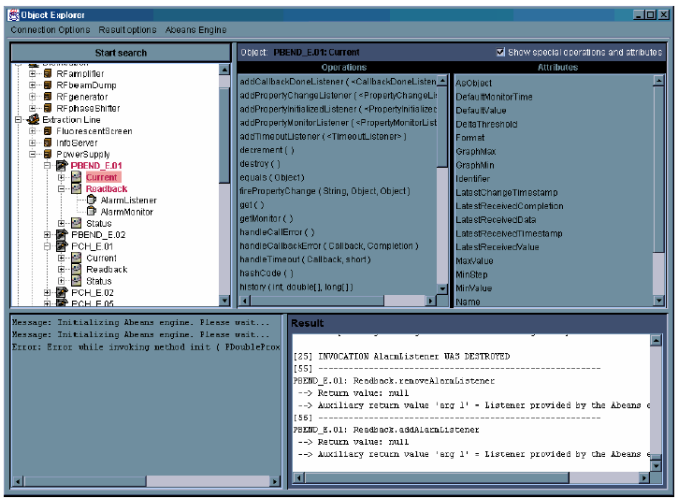
The client is completely unaware of any deployment and lifecycle issues for the Component it wants to talk to.

Once the client has a reference, can call directly the interface via the IDL stubs.

Alternatively a generic client can retrieve the interface from the Interface Repository and use Dynamic Invocation (CORBA Introspection).

The Object Explorer is such a generic client.

ACS Object Explorer



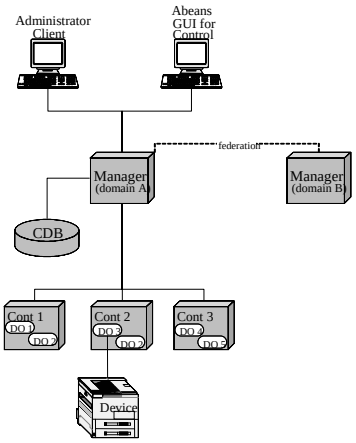
The **Object Explorer (OE):**

• Is a generic tool used for low-level inspection of objects in ACS. It can be used as a debugging or testing tool by the developers and maintainers of a system.

• Allows to interact with any CORBA Object known to the system (I.e. whose reference can be retrieved from the Manager and whose IDL interface can be retrieved from the Interface Repository).

Component's Administrator View

- An administrator defines deployment by customizing the Configuration Database for the Manager
- Manager is responsible for managing and checking the lifecycle of Components
- Containers are directly responsible for the Components that are assigned to them



The administrator of the system has a different perspective.

He is interested mainly in the deployment:

•Where are Containers running?

•What Components are deployed and deployable in which Containers?

•What is the status and health of Components and Containers

•Who is using who?

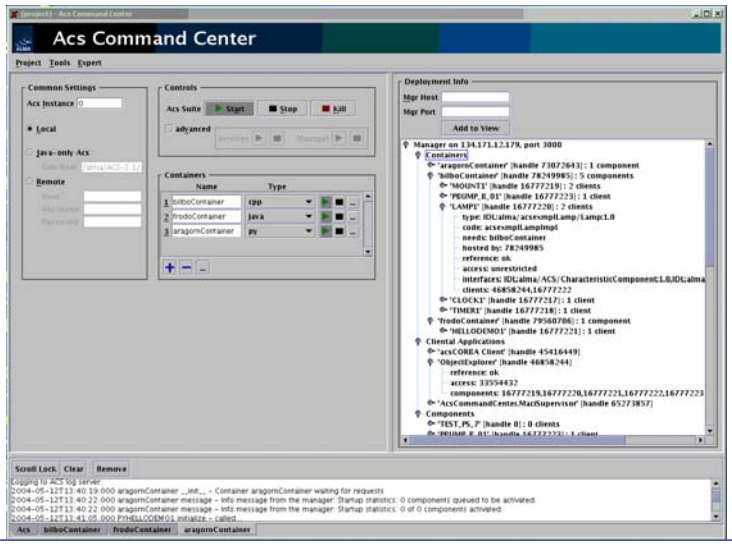
Based on this information, whose static part is kept in a configuration database, it is possible to evaluate and improve the performance of the system or to recover from error conditions. For example it is possible:

•To redeploy Components that have strong and continuous interaction on the same host or Container

•To deploy resource intensive Components on powerful or idle hosts

•Te deploy critical or unstable Components on a separate Container to reduce damage in case of crash.

ACS Command Center



The ACS Command Center is an administrative application used to start and stop ACS services, manager and containers.

It allows to manage the system distributed on several hosts, start tools and inspect the deployment of the system.

The left side allows to control the startup and shutdown of the Services, Manager and Container on distributed hosts.

The tree on the right shows the run time system deployment and the relations between Components and Containers.

Summary: Component Middleware

- Real Framework
- Component-Container based architecture emphasizes Separation of Concerns in two dimensions:
  - Technical and functional development
  - Implementation and deployment/administration
- Scalability
- Maintainability
- Reusability

The adoption of an Architecture founded on the Component-Container model is a big step forward in the direction of a real application framework:

the application are framed inside a well defined technical architecture and a lot of the “do and redo differently” code is completely taken out of the hands of the developer.

The Component-Container emphasizes Separation of Concerns in two dimensions:

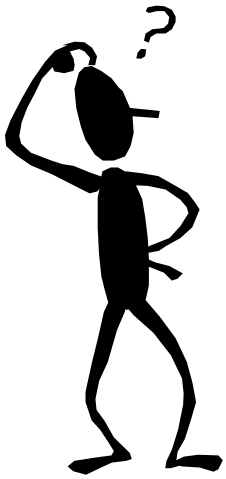
•Technical and functional development
It is possible to split the development team based on the skills in technical experts and domain experts and each developer has therefore a thinner profile.

•Implementation and deployment/administration
In this way it is easier to improve the performance of the system and make it scale up while it grows.

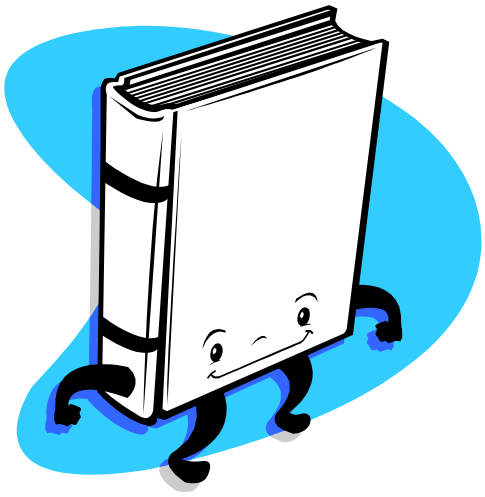
Components are also much easier to reuse and plug into the code, since the three contracts clearly specify what they provide and what they need.

It is also often easier to take legacy code and wrap it into Components that can be integrated into the system independently from the programming language.

Questions (& Answers)



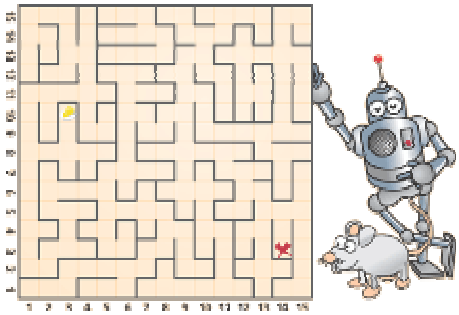
7 – High Level Framework



Even more high level?

- We have a framework but there are still too many ways to do the same thing.
- Take decisions, restrict the paths, show the “best way” for our domain
- Leave alternatives open

We will discuss examples from ACS



We have up to this point identified a very powerful and generic ***framework.***

But still it is too generic: we can still do the same thing in many different ways and probably different developers in the team will take very different roads to the solution of similar problems.

Also, some specifications are still very generic because have to be usable in very different application contexts.

To isolate as much as possible the application developers from the technical concerns, we need to give them a framework closer to the “finite” system and take technical decisions.

The technical framework team has to identify the “best way” among the possible solutions and provide high level framework elements that make very easy to use this now standard solution.

The “best way” always depend on the specific application domain and therefore the choices done at this level always depend on two opposite forces:

•Make it general, so that it applies to a wider application domain

•Make it very specific, so that it fits very well and easy into a problem

This always leads to necessary compromises.

In the design of ACS we have been and are driven by the following considerations:

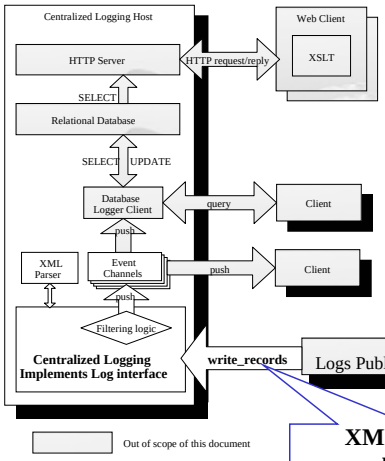
•Our domain is the whole Observatory software. Not just the Control System or the Data Reduction Pipeline. We need to satisfy the needs of all our stakeholders.

•Sometimes the requirements in the sub-domains are very different and there is no “one size fits all” solution. Then we have to provide alternative solutions, but mutually coherent and compatible.

•Some cases are really “special”. We cannot completely close the door. We have to allow going via special paths when justified.

In the next pages we will discuss some of the packages in the ACS high level framework that allow developers to write in an easier way and with better integration and maintainability applications for our “observatory domain”. Depending on the time available we can more or less of these examples and discuss them.

ACS Logging System



- To publish any kind of status and diagnostic information for interested clients and archival.
- Based on CORBA Telecom Logging Service.

C++ API ⇒ ACE Logging

Java API -> java.util.logging

ACS Log Service -> IDL

XML logs follow pre-defined schema

As already said, a centralized logging system is essential for the operation of a distributed system.
The standard CORBA Logging Service provides a very powerful and scalable logging infrastructure.

But this infrastructure is still too generic for our purpose.

In particular is does not provide any guideline on how to structure the contents of the messages.
In ACS we have therefore decided to structure messages using XML and we have defined an XML schema for the contents of logs.

Then we have implemented wrapper APIs in the supported languages and a generic server for other clients that make trivial to use the logging system and generate messages properly formatted according to the schema.
Doing this we have taken into account that Java has a native logging API and that therefore Java developers should have been very happy of being able to use this standard API to log transparently into the centralized logging system.

The driving forces in designing the ACS layer on top of the standard CORBA logging service have been:
•Define how the flexible and generic CORBA logging system shall be used: choose a path for the functional developer
•Make the usage as simple as possible
•Hide native CORBA and make it look like APIs the developers are already comfortable with.

ACS Error System

- We need a unified way of dealing with errors through the system
- CORBA supports “distributed” exceptions
- We need more:
  - Error format standardisation
  - Error handling design patterns
  - Error trace
  - Error logging
  - Synchronous and asynchronous error handling
  - Error browsing and definition tools

THE ACS Error System provides these features

It is extremely important to have a coherent and complete way of handling error conditions all over the system.

This involves handling errors:

•in different programming languages:
an error in a C++ Component has to be propagated and understood by a Java Component

•distributed of the network:
an error in a Component in one host hast to be propagated over the network to a client component on another host, possibly with a different operating system and architecture

CORBA allows defining exceptions in IDL (with some limitations due to the need of supporting non exception-aware programming languages) and throwing exceptions over the call to IDL operations. This means that a remote call can throw an exception that goes back over the wire to the caller and looks the same as a local exception. The exception’s data is handled by CORBA and marshalling is therefore transparent.

The possibility of treating local and remote exceptions in the same way is extremely important in order to build transparency in the distribution of Components, but it is not sufficient.

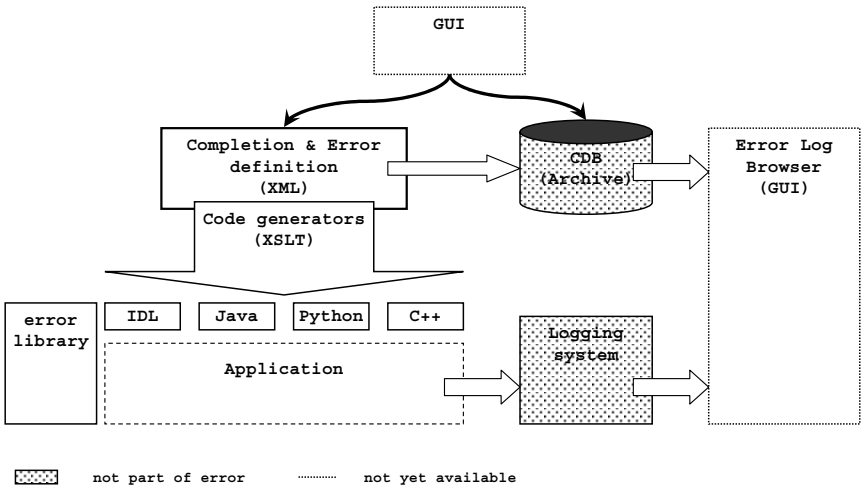
There are many other issues that we need to solve to allow treating efficiently error conditions in Components:

•Error format standardisation
A part from the exception “name”, we often profit significantly from additional context information in the data coming with the exception. But to be able to interpret this information the data structure shall be standardized in the format and contents.

•Error handling design patterns
There are a number of well proven error handling design patterns (see ARCUS Error Handling for Business Information Systems:
http://www.eso.org/projects/alma/develop/acs/Releases/ACS_3_1/Docs/ARCUSErrorHandling.pdf).
Providing a standard implementation for these patterns helps a lot in writing solid applications.

….continues

ACS Error System Architecture



…continues

•Error trace
In a stand alone application running in a single executable, a low level error is propagated up through the call chain until it reaches somebody that is capable of handling it or until the application is terminated. At each level useful context information can be added. Some languages like Java provide native support for retrieving and manipulating the call chain, but others like C++ do not.
This is the Backtrace design pattern and it would be very useful to provide an implementation that works over CORBA network calls.

•Error logging
With a distributed system the Backtrace pattern allows to trace the chain of errors across distributed components, but the errors traces end up all the times in different places, i.e. where the component that finally handles them resides.
It is important to have a centralized place where it is possible to browse and search for errors, with context information allowing to identify where each error occurred.
This can be done sending all error traces to the centralized logging system

•Synchronous and asynchronous error handling
The exception mechanism works for synchronous calls: the execution of an operation fails, an exception is thrown and it is caught by the caller.
But in highly distributed systems many actions have to take place asynchronously: an activity is started by a method call, but the method returns immediately and later on a callback is used to report the result. We need to have a standardised mechanism to report errors also in such asynchronous situations.

•Error browsing and definition tools
It is convenient to have friendly tools to browse the errors and to define the error structures used to report context specific information.

Different middleware systems provide support at different levels for such issues, but they cannot provide a comprehensive solution, because they want to be fully general and here we often have some percolation from the application domain requirements.

ACS for example provides a solution on top of CORBA to these problems taking into account our Observatory/Scientific facility needs.

This diagram shows the architecture of the ACS error system.

Essentially we have:

•Defined a way produce and transport error traces with exceptions and propagate them consistently across languages in CORBA calls.

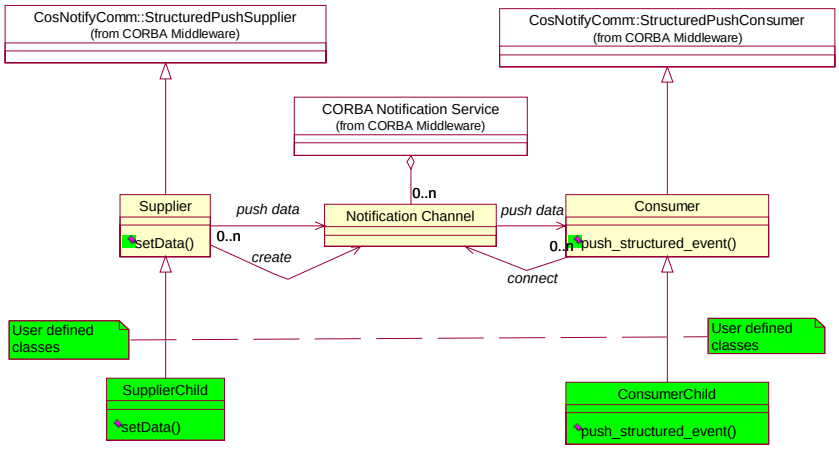
•Designed an XML schema for the definition of error conditions and for their storage in the logging system.

•Implemented code generators that from the XML error definitions produce IDL definitions for the exceptions and convenience support classes in the various programming languages, to overcome limitations in the CORBA support for exceptions.

•Implemented some standard error management design patterns

•Defined how to propagate errors in asynchronous calls

ACS Data Channel



The CORBA Notification Service is another example of a very powerful and generic service.
If on one side it is extremely flexible, at the same time it is quite complex to master.

On the other hand, we have seen that in most of the use cases we have analyzed we have just the following pattern:

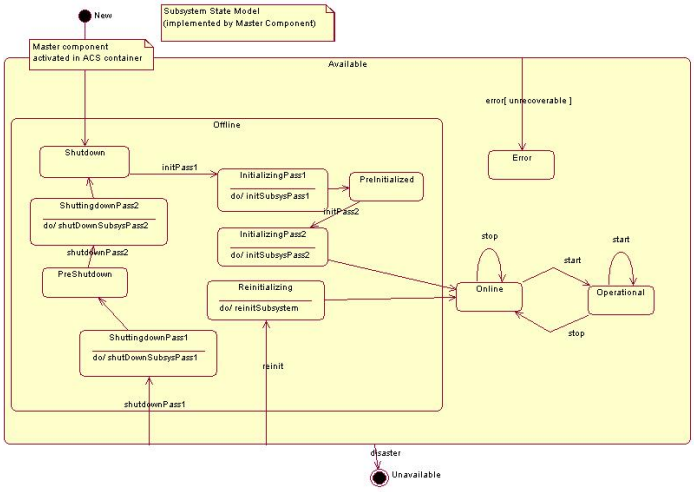
- A data structure is defined
- Whenever data is available the structure is filled in and published
- One or more subscribers receive the data they are waiting for.

Therefore we have provided in the high level ACS framework an implementation for this pattern:

- The event data structure is defined in IDL
- A Supplier class allows to control easily when data is pushed on a channel:
  - Suppliers can create a notification channel
  - Suppliers know when a consumer has subscribed to an event type on the channel it publishes structured events to. A "smart" supplier will only publish events (thereby reducing network traffic) when consumers are subscribed. Only useful in a one-to-many model.
  - Suppliers can automatically execute a method if the connection is ever lost.
  - Suppliers can destroy a notification channel (coordinating with other suppliers when multiple suppliers publish on the same channel).
- A Consumer class allows to control easily when data is given to a client
  - Subscribe to and unsubscribe from all types of events.
  - Filter out structured events they don't want to process.
  - The consumer doesn't have to do anything with the event's data. Can literally be used as a notification mechanism.
  - Specify when they are ready to start receiving events.
  - Suspend and resume their connections to the channel at any time.
  - Notified when a Supplier begins publishing a new type of event and dynamically subscribe to it. The same holds true when subscriptions are no longer offered.
  - Automatically execute a method if the connection is ever lost (i.e., the channel is destroyed).

See the paper "A CORBA event system for ALMA common software", from D.Fugate in this conference for more details.
Our objective here has been to provide a very simple and standardized interface for the most common use case, while complex situations can still be dealt directly with the Notification Service APIs.

Master Component and State Machines



The ALMA architecture is based on subsystems that are “running” independently.

This is a very common architecture and appears in many other scientific (and industrial) facilities.

The subsystems are administrated (started, stopped, checked for health) by an high level coordination application.

This applications does not want to know about the peculiarities of each subsystem and want to be able to tread them all in the same way (well, there are always exceptions, but forget about them for the time being).

It is therefore reasonable to define a standard interface that each subsystem has to implement and expose to the administrator.

The most natural type of interface for such purpose is a state machine and therefore we have specified subsystem level state machine and implemented it in a Master Component.

This is a big help in getting a system that is easy to integrate even if the subsystems are developed by completely independent teams, as it is the case for large international collaborations.

The overall Technical Architecture is specified:

•The system is divided in sub-systems

•Each sub-system has a Master Component implementing a standard State Machine

•This Master Component coordinates the activity of the other Components making up the subsystem

•The Administrator Component deals in a standard way with all the subsystems

Components implementing state machines are very useful also in many other places, but strangely enough we have not been able to find any general state machine implementation in the free source world that we could use for our implementation.

Therefore we have implemented a general solution based on code generation directly from UML models.

Actually, we have realized in the last year that there are very good reasons and even good tools to generate code from UML Models.

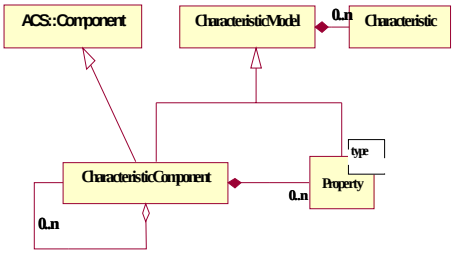
This would even more strengthen the separation between technical and functional concerns by letting application developers design their component and data entities in UML using standard commercial tools and have all the code generated up to filling in the body of the functional operations.

Task of the technical team is then the implementation of suitable code generators.

Some details on this approach have been presented in the ACS papers in this conference.

Devices:
Component-Property-Characteristics pattern

- **(Characteristic) Component**: base class for any physical/logical Device (e.g. temperature sensor, motor)
- Each Component has **Properties** (e.g. status value, position - control/monitor points)
- **Characteristics** of Components and Properties (Static data in **Configuration DB**, e.g. units, ranges, default values)
- **ABeans**



On the Control System side the concept of a Software Device representing physical or logical devices of the system such as antenna mount, antenna control unit, correlator, etc is very common.

It is therefore useful to take a formal design pattern for this model and implement it.

We have taken the *Characteristic Component-Property-Characteristics* pattern.

The Device itself is mapped on a Component in our Component/Container model.

Each *Characteristic Component* implements operations and is further composed of *Properties (representing monitor and control pointes).*

A *Characteristic Component* can also contain references to other *Components* to build hierarchies.

Both *Characteristic Components* and *Properties* have specific *Characteristics*, e.g. a *Property* has a minimum, a maximum, units…. The common behaviour of *Characteristic Component* and *Property* has been factorized in the *Characteristic Model* common base class.

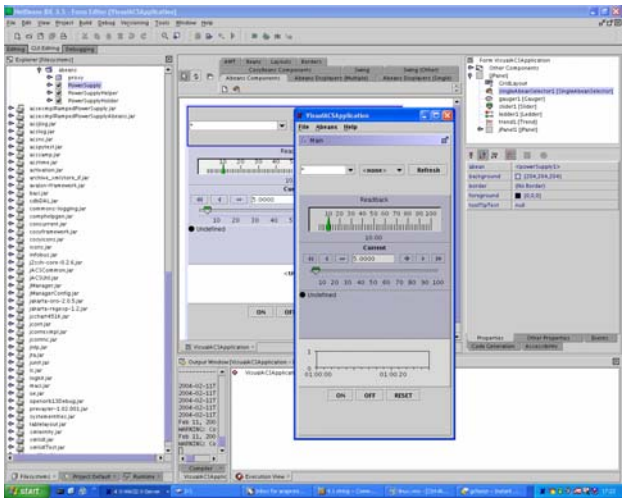
Values of the *Properties* are updated asynchronously by means of monitor objects.

While there are in principle an infinite number of Component types, for example one for each physical controlled device, there are very few different *Property* types

Underneath this high level pattern, design patterns for synchronous and asynchronous value retrieval/setting, monitoring and archiving or alarms are part of the Property definition

The implementation of this pattern provides once more a clear path for the Technical Architecture of the Control System.
The developers responsible for the implementation of the control system have to:
•Identify the hierarchy of logical and physical devices
•Identify the operations allowed on each device and the monitor and control points. This is the IDL interface of the devices.
•Implement the code for the operations.
•Implement the hardware access layer (using the Bridge pattern) to connect the properties with the actual hardware
The framework provides them with standardized configuration and deployment means, automatic monitoring for telemetry and many other facilities.

Abeans and visual editing



An example if the generic ABeans GUI building framework that has been adopted by ACS and by other Control System frameworks.

Abeans are Java Beans that are aware of the Component-Property-Characteristic paradigm.

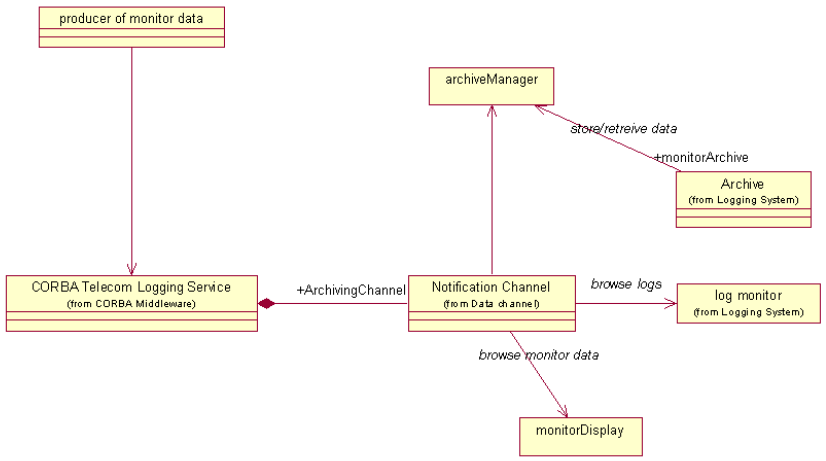
Using Abeans it is possible to use any Java Visual Builder (like Sun Java Netbeans, or upcoming Eclipse extensions) to visually build user interfaces for Components.

A set of graphical Java Beans implements the most useful widgets for the development of Control System applications, aware of the concepts of Components, Properties and Characteristics.

At the same time a code generator produces Java Beans based on the IDL interface of ACS Components. These Beans are therefore automatically integrated in any Visual Builder.

For example, a Gauge widget can be associated to an ACS Property to display the value, draw trend plots and configure automatically itself based on the Characteristics stored in the Configuration Database.

ACS Monitor Archiving system



All Control Systems need to provide telemetry data to monitoring clients and send it to an archive for offline analysis.

Since we map monitor points into Properties, we can implement a generic monitoring system in the properties as a standard service for all developers.

Archiving is enabled/disabled and configured on per-property basis. ACS Properties publish their value on a specific ArchivingChannel notification channel as structured events, by using the ACS Logging System.

The parameters for data publishing are defined in the Configuration Database and it is possible to specify, on a per–Property base:

•Archive priority

•Max time interval between two archive submissions

•Min time interval between two archive submissions

•Min value change that forces an archive submission

Summary: High Level Framework

- Identify and implement:
  - High level Technical Architecture blocks
  - Standard domain design patterns
  - Standard paths when many alternatives are available
- Do not rule out special cases

The developers of the system should be confronted only with the choices connected to the functional aspects of the system.

But most frameworks leave still too much freedom of choice because are thought for “any kind of application”.

Developers then risk to get distracted by technical choices.

In a big projects, different subsystems might take different paths leading to waste of development resources, duplication of effort and interface problems.

It makes therefore sense to identify and implement on top of the all purpose framework solutions that still general for the domain of application and for the whole observatory, although give a clear path to the developers.

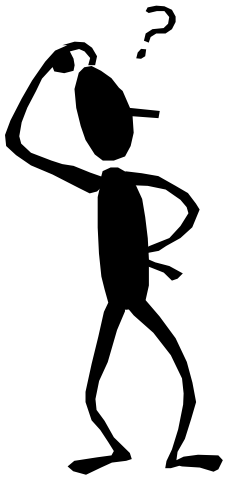
What to do (or to reuse) at this level is really a matter of choice, but there are plenty of examples.

This is also an area where “functional developers” can feel to be strongly limited in their freedom of choice by the “technical team”.

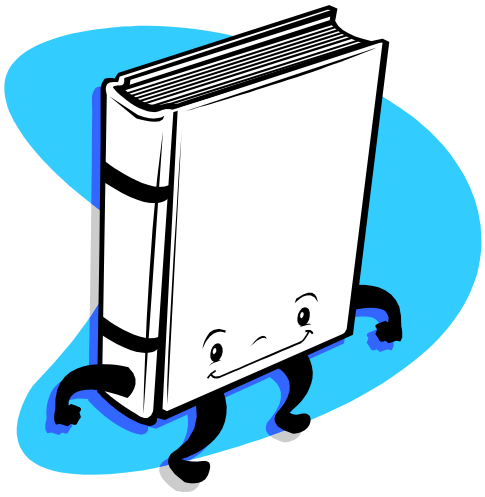
This is often done with a good will and at the advantage of the global project: often it is better a sub-optimal solution working for everybody than many optimal but different solutions that will cost immediately for duplication of effort and in the future for maintenance.

Nevertheless, there are really cases where searching for a “special solution” is not avoidable.

Questions (& Answers)



8 – Development support



Dealing with big projects

- Big projects, big troubles:
  - Ensure that installations are “identical”
  - Integration issues
  - System deployment issues
  - Maintenance issues
  - Personnel turnover

When there is a big project that spans a whole observatory, is developed across many development sites and spans over many years (or combinations of these characteristics) there are many sources of problems:

•The many development machines have to be aligned with the same software at the same level. The framework we have described consist of very many pieces from many sources and it is very easy to encounter incompatibilities between these pieces and the underlying operating system. It is therefore necessary to centralize the definition of the “mix that works” and ensure that everybody gets the right cocktail (possibly being able to check if a configuration is clean or not)

•All the pieces developed in the different sites and by different developers have to come together and be integrated. There should be standard ways of testing the functionality of each single component autonomously and automatically and of integrating them and test them as a unit. If there is an integration team, very often it does not have the knowledge needed to thoroughly test and debug the single components.

•The deployment on the operational system involves many hosts and possibly many sites. Downtime due to deployment problems is very expensive and therefore it is important to have precise deployment and rollback procedures.

•Debugging an maintaining the system can be very expensive.

> •One should get the architecture, design and implementation right in the first place. Therefore it is very important to have means to evaluate (or, better, measure) the quality of the work done and to test it.
>
> •When problems will come out or changes will be needed it will become very important to have a system that is homogeneous and understandable. If the same patterns and tools are reused over and over, everybody in the team knows where to puts its hands.
>
> •Factorizing common code in a single place (the framework) allow to “fix one and cure all”

…. Continues on next page

Software Engineering practices

Software Engineering and Quality Assurance activities:

- Software Process
- Document Reviews, Format, Templates
- Development Environment
- Integration Procedure
- Coding Standards
- Code Inspection
- Configuration Management
- Testing framework and assessment
- Change Management

All these troubles can be mitigated by adopting Software Engineering practices.

Balancing the cost of the overhead introduced with the complexity of the project and the benefits that can be reached will dictate up to which point it makes sense to push for formal practices.

But in any case it is counter productive to simply state rules on paper and ask people to follow them.

It is essential to provide tools and support so that the adoption of the practices and their verification is transparent or becomes second nature.

…. continues from previous page

•For a system that will take many years to implement and that will be operated for many years by different people than the developers it is important not to underestimate the problem of personnel turnover. Personal ownership of the code shall be avoided, because if the person disappear the knowledge will be lost and intervening on the code will require expensive reverse engineering. Founding the architecture on standardized patterns and pushing for factorization and reuse (coupled with code review and team rotation) is of great help.

All these considerations typically appear as requirements for the system to be built.

Can the Framework help?

- Drive technical architecture choices
- Accretion point for factorization
- Distribution vehicle for:
  - Upgrades
  - Controlled versions of the software
  - SE support tools
  - Development and deployment environment configuration

Can the adoption of an observatory wide software framework help with these issues?

Clearly yes.

As we have said already, first of all the whole system structure becomes much more uniform and consistent, because everybody is pushed to use the same architectural and design pattern.

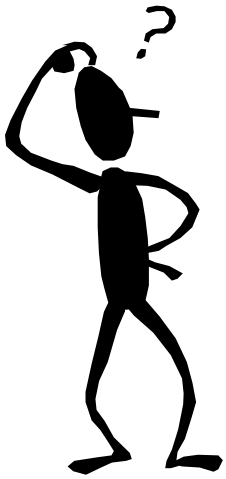
Then new solutions of general usage can be integrated in the framework to be reused by other groups.

But it is also a good idea to integrate in the distribution of the framework also all the tools for software engineering we have described in the previous pages and the configuration of the development environment.

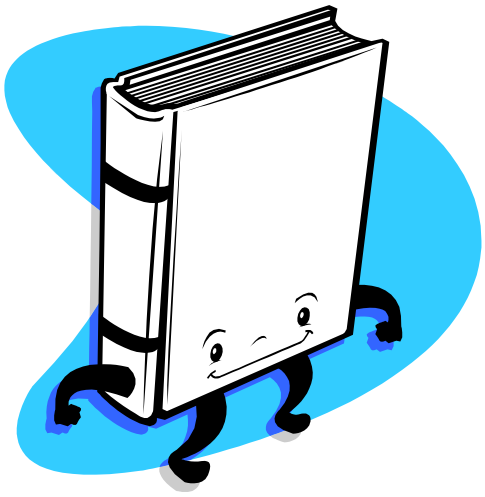
If the installation of the framework on a freshly installed system produces a working environment for development, testing or deployment it is much easy to have reproducible installations.

This approach is again a “global gain” paid at the expenses of personal freedom for the developers and therefore it is important to find the right balance based on the characteristics of the team and of the project.

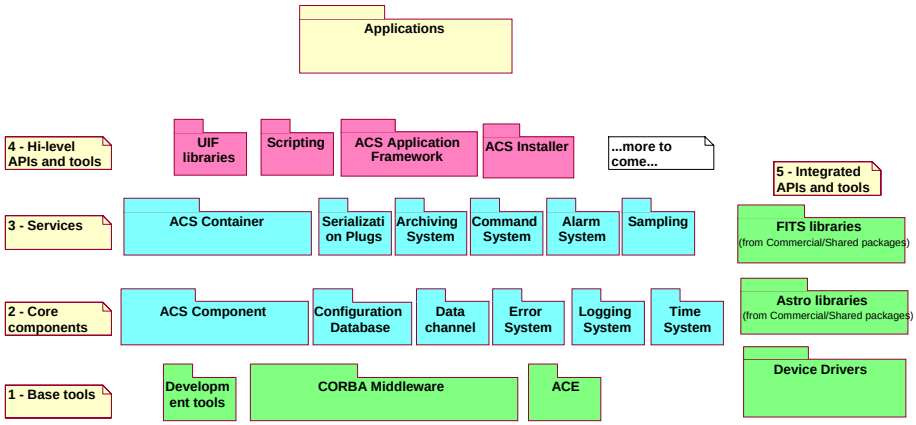
Questions (& Answers)



9 – Wrapping up



ACS global architecture



The ALMA Common Software is an example of the approach described in the previous pages.
This package diagram is a simplified version of the complete ACS Package Diagram from the Architecture document

The architecture is divided in layers and each layer can use the packages in the same layer and in the layers below.
This allows us to keep under control the dependencies between packages.

An important aspect is that the “base tools” layer is a thick and reliable foundation based on CORBA and other “off the shelf” publicly available tools and software packages.
This includes or defines as pre-requisite for ACS installation:

- A standardized set of development tools (like compilers, Makefile extensions, installation procedures and tools, JUnit and other test support tools, emacs configuration and so on)
- CORBA implementation and services for the different languages
- ACE and other public domain libraries used by ACS and available for application developers.

It is a main objective to use whenever possible readily available packages and not to re-implement services that already exist.

But for each service/package, ACS provides and “interpretation” of the way we want it to be used in the terms of design patterns and support code implementing the design patterns to makes it easy to use our “interpretations”. This reduces the learning curve and makes the code more uniform across the distributed development sites.

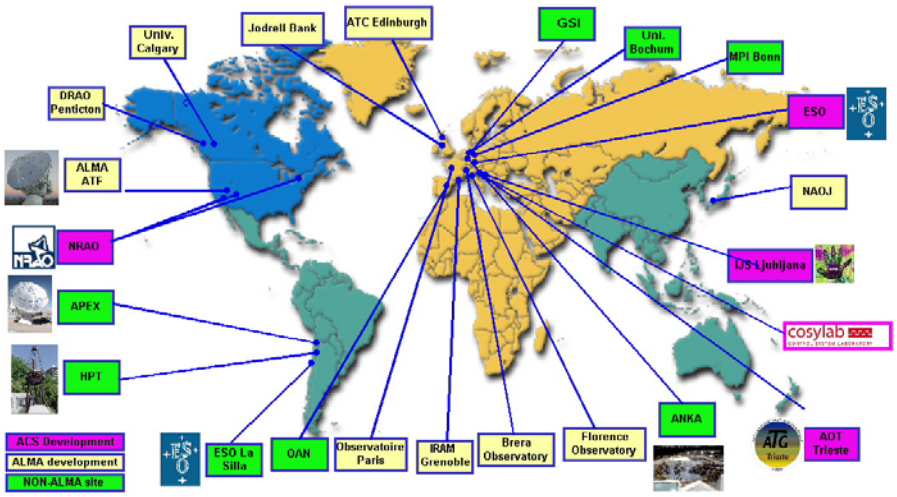
In some case there is really no ready made implementation that we can use and therefore we provide our own implementation, but keeping an eye at the OMG specifications.

We also recognize that this approach heavily constrains the freedom of the developers to choose between the different possibilities of using a service; therefore we allow to “drill a hole” in the upper ACS layers and use directly the underlying layers when this is justified by a real need.
Typically such holes are later on closed again by incorporating the new solution into ACS itself.
An example of this is the ACS “Data channel” that wraps the CORBA Notification Channel to provide very easy access to the push-push model.

The ACS distribution contains as much as possible also all packages of the “Base tools” layer, to make sure that everybody gets a consistent installation, with the proper set of tools in the proper versions. Notice that in some cases it is also necessary to deliver patches to tools and not simply a release downloadable from the web.

The ACS distribution will be used also to package and distribute other APIs and tools that are not part of ACS and that are not used by ACS, but are used by a number of ALMA subsystems and that is therefore convenient to distribute as one single entity together with ACS. This can include for example FITS libraries, Astronomical Calculation libraries or device drivers.

ACS installations and projects



ACS is fully based on public domain software and is in an advanced development phase.

The development is backed by a big project (ALMA) but there is interest in a wider community of users.

Therefore it has a good potential for being adopted by other projects.

ACS is installed in all ALMA development sites, but it is also used or under evaluation by a number of other projects.

Quantifying the advantages

- The feeling is good
- Can we measure the impact of adopting such a framework?
  - Very difficult: all numbers are debatable
  - But we have some good example

All the arguments that have been given in favor of adopting a common software infrastructure for the whole observatory produce the good feeling that it is an investment worth to do.

But can we really have some measure of the advantages?

This is very difficult and numbers here are debatable.

The only real way to get a reliable measure would be to have a big project already done and measured and re-do it from scratch adopting a framework.

For sure there are some successful project that have been done to a great extent in this way: the VLT is an example of well performing observatory whose software has been developed within schedule based on a framework common to a big part of the observatory.

But we cannot really measure:

- Technology is advancing so fast that we cannot really reliably compare any two software works done at 10 years distance
- Nobody has the resources and the interest in developing two parallel systems with and without such a framework.

But there are some partial examples that can be measured and can be extrapolated.

One of these is shown in the next slides and comes from the VLT.

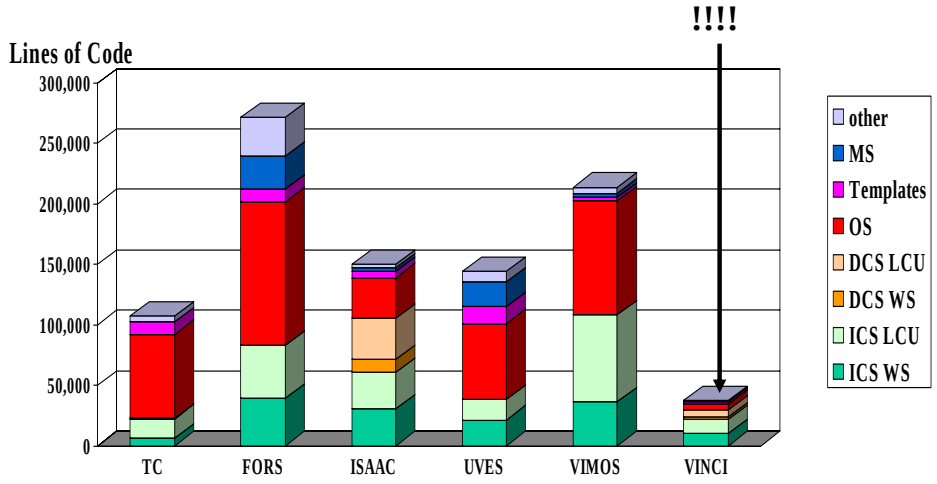
The Control System of the VLT and of its instruments has been developed based on a Common Software.

After having developed a number of instruments it has been realized that that there was a big potential for factorizing major parts of the architecture of the instrument in a Common Instrumentation Software.

This has been done extending the VLT Common Software with Instrumentation Software and new instruments have been implemented using this extension.

Therefore we have the possibility of quantitatively compare instruments developed with and without such a common infrastructure.

Size of Instrumentation code reduced



UVES ⇨ VINCI 70% less

A.Longinotti – ESO Wide Review 2002

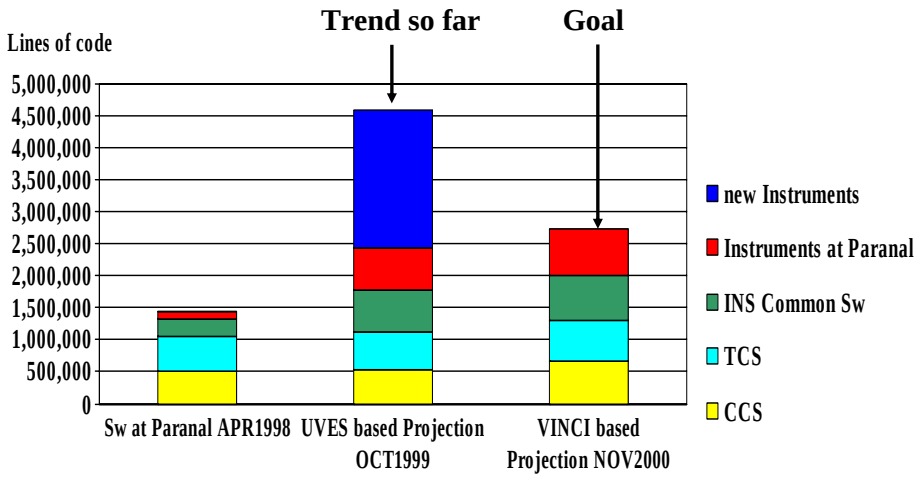
This diagram compares the repartition of lines of code for various VLT instruments.

VINCI has been developed using the Instrument Common Software.

The others without it.

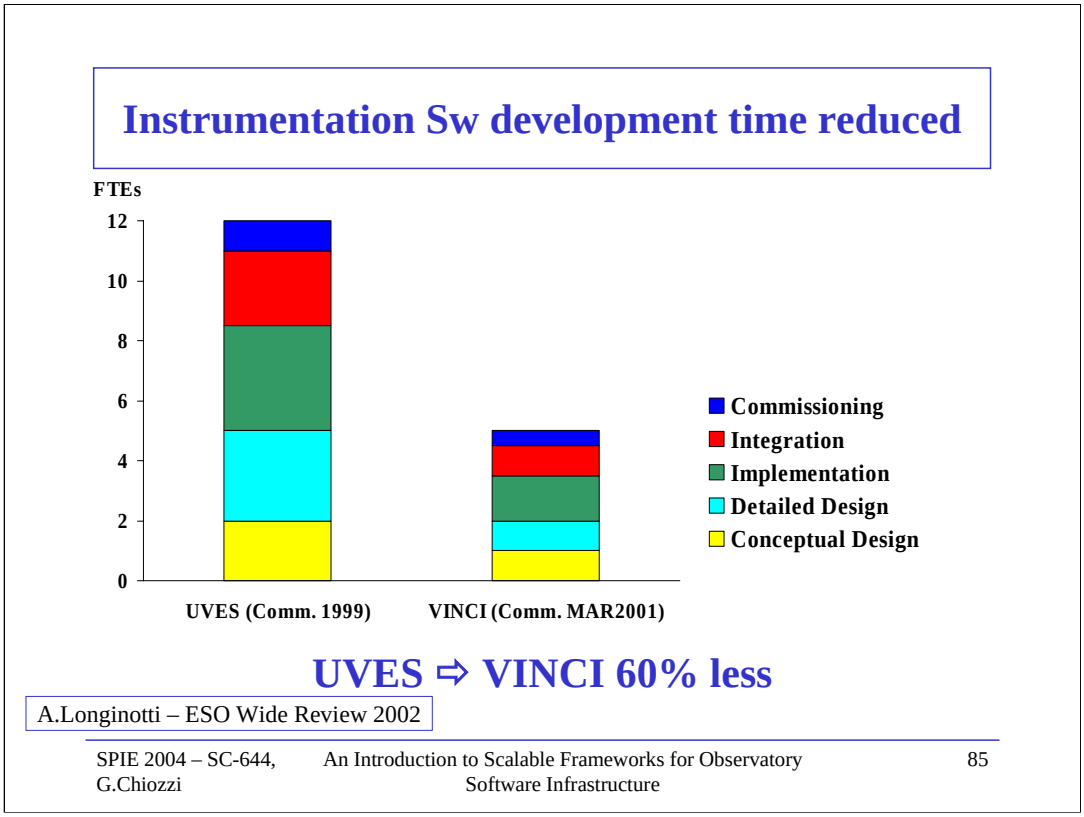
UVES in particular has a comparable complexity.

Instrumentation code more maintainable



OCT1999 ⇨ NOV2000 Projection: 60% less

A.Longinotti – ESO Wide Review 2002



As of today, the number of instruments that are using or will be using the Instrument Common Software is 23.

Comparing the data for UVES (that does not use the INS Common Software) and VINCI (that uses it) we can estimate a gain of 7 FTEs in the development phase of the project.

If we extrapolate over the development of the 23 instruments and take into account the cost of developing the INS Common Software itself (~25 FTEs), we get a total gain for the astronomical community of:

23*7 - 25 = 136 FTEs

Then we have to count the gain in maintenance.

As shown, the INS Common Software allows to reduce of two thirds the lines of application code.

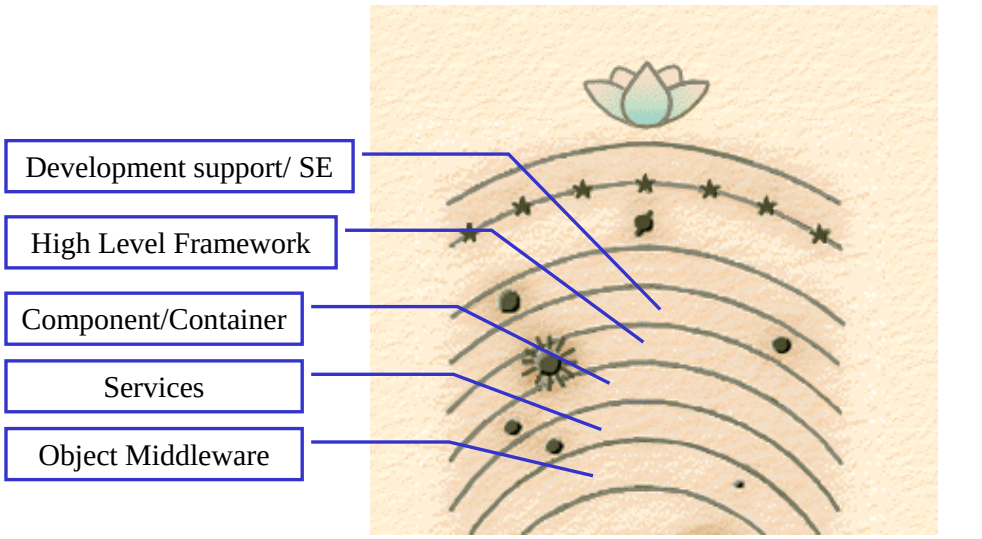
Based on the current situation at Paranal with a mix of instruments with and without INS Common Software and on the resources allocated to software maintenance for the instruments, it is reasonable to estimate a gain of 2.2 FTEs/year.

The maintenance of the INS itself is currently of 1.2 FTEs/year.

Therefore we can round the total gain to something like:

- 135 + (1 * observatory life) FTEs

The path to Heaven?



Let's summarize the steps that have led us to the definition of the components needed by a framework that could be used for the whole software infrastructure of an observatory.

The components are down up in the order that in which we have encountered them following our logical thread.

Who should try this path?

- New big projects:
  - Select a core technology
  - Assign a technical team to the development of the high level framework
- New small projects
  - Select a recent but stable solution (ACS like?)
  - Collaborations
- Upgrade projects
  - Select a recent but stable solution
  - Introduce it in selected areas and let it percolate

Who should try the path of adopting an observatory wide framework like the one described?

I think that every project would find big gains but using different approaches.

A new big project cannot probably avoid to adopt such a solution.

To get the better results, a technical architecture team has to be established.

The team has to select a core technology among the palette of currently available recent but mature choices (avoiding risky cutting hedge technology). Starting from scratch does not make sense.

Then the technical team has to work on defining and implementing the high level framework, trying mainly to do a good work of integration of existing solutions and implementing only what is really necessary.

A new small project should fully adopt an existing solution built by a big project or by a collaboration, trying to behave somehow as a subsystem of the collaboration.

This would allow a very fast startup, with extremely rapid progress on the solid ground of proven solutions.

A small team should concentrate the effort on the functional aspects and not on the technical framework. Nevertheless some specific technical development will be very likely necessary, because each project has some very specific requirements. This work could be done in the form of collaboration with the big project.

What about already existing systems that need to be refurbished and upgraded?

There are many around and in most cases the owners cannot afford to put the resources for developing a completely new system.

But hardware becomes obsolete and cannot be replaced, software maintenance becomes expensive and localized interventions are unavoidable.

In such a case it will be probably most cost effective to take a recent but stable complete solution, just like in the case of small projects. Then apply the solution to the critical parts, for example replacing subsystems whose hardware must be replaced. Or to the parts that have proven weaker and harder to maintain. Then build bridges between the old and new system to allow them to interoperate.

If the system to be upgraded is big the development team can probably give an important contribution in form of ideas and collaborations for the development of new high level framework features to the team developing the adopted framework.

Contour conditions

- It is not easy to adopt a framework in a project.
  - Different background, cultures, experience.
- What contour conditions are necessary?
  - Small motivated team

  Or
  - Strong management and control procedures

Experience shows that it is not easy to get accepted by the developers the introduction of a framework like the one described in this course.

The problem is that each developer or group has it own different background, experience and culture.

As said the framework has the purpose of driving the developers toward narrow but safe technical paths.

Many developers would see this a limitation in their freedom, and this is certainly partially true.

Other would say that they can do the job much better for what they specifically need. And this is also often true.

The problem is that the advantages can be seen from above only and not from the perspective of the single developer:

Non optimal solutions traded for uniformity and coherence

Freedom traded for maintainability

Focus on functional work

Therefore the success is bound to one of two contour conditions:

•The project is done by a small motivated development team that is convinced of the advantages of the framework solution and can push it up

•A strong management imposes the solution to the whole team and establishes control procedures until the project is sufficiently advanced that the gains have become clear to everybody.

Bibliography

- ALMA/ACS Papers in this conference
- ACS Web Page: http://www.eso.org/projects/alma/develop/acs/
- IT Architectures and Middleware, C.Britton, Addison Wesley
- CORBA/OMG web page: http://www.omg.org/
- D.C.Schmidt and TAO web page: http://www.cs.wustl.edu/~schmidt/TAO.html

•This conference contains a number of papers with more details on ACS. You can look in particular at the following papers:

- •The ALMA Software Architecture, J.Schwarz et al [5496-22]
- •The ALMA Common Software: a developer friendly CORBA-based framework, G.Chiozzi et al. [5496-23]
- •Container-Component Model and XML in ALMA ACS, H.Sommer et al. [5496-24]
- •ACS Sampling System: design, I plementation and performance evaluation, P. di Marcantonio et al. [5496-45]
- •A CORBA Event System for ALMA Common Software, D.Fugate [5496-68]

•ACS Web Page: http://www.eso.org/projects/alma/develop/acs/
The ACS Web Page contains a lot of documentation, a detailed architecture description and references to other papers and documents.

• C.Britton, *IT Architectures and Middleware*, Addison Wesley, 2001 ISBN 0-201-70907-4
This is a very interesting (and reasonably thin!) book focusing on requirements and principles of distributed systems, offering an overview of middleware technology alternatives.

•CORBA/OMG web page: http://www.omg.org/
The OMG web page is the starting point to find the CORBA specifications, although what can be found there is too superficial or too detailed for a useful introduction and startup. Better to look in other pages or books.

•D.C.Schmidt and TAO web page: http://www.cs.wustl.edu/~schmidt/TAO.html
Page full of papers on distributed design patterns, CORBA design, high performance and real time distributed systems.D.C.Schmidt is one of the real gurus of the field

•M. Voelter, M. Kircher, and U. Zdun. *Remoting Patterns - Patterns for Enterprise, Realtime and Internet Middleware*, Wiley & Sons, to be published in 2004
This very good book describes the most important patterns used in Object Middleware and compares CORBA, .NET and WebServices.

•M.Henning, S.Vinoski, *Advanced CORBA Programming with C++*, Addison-Wesley, ISBN: 0-201-37927-9
Like the Bible: very old, but still the essential one.

•Communications of the ACM, October 1998. Special issue on CORBA
Old but very interesting collection of introductory papers on CORBA

Acknowledgments

- The material for the course comes from the experience of:
  - ALMA project and in particular the whole ACS team
  - VLT project
- Many ideas for the middleware presentation come from web presentations and in particular from the ACE/TAO web page

Thanks to everybody for ideas, slides and discussions

The material for this course comes from 10 years of experience in the development of Common Software but, most important, of discussions with all people involved in developing and using this software.

All this people has therefore given an important contribution and many have also provided slides or ideas for slides.

First came the VLT project, where in particular K.Wirenstrand, R.Karban, A.Longinotti have to be thanked for the past work together and for the discussions we have all the time to compare VLT, ALMA and to see how the software world is evolving.

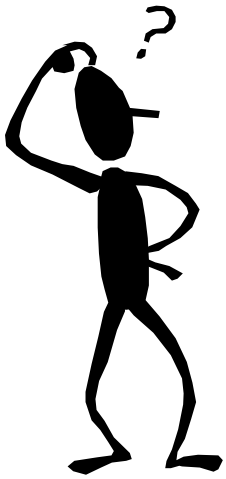
In the ALMA project everybody has shaped a piece of ACS, but a particular thank for the discussions and slides goes to H.Sommer, J.Schwarz, A.Farris, M.Voelter and the other members of the ACS team.

The collaboration with M.Plesko and the Joseph Stephan Institute in Ljubljiana for the design and development of ACS is also of great importance.

Many ideas for the middleware slides come from presentations on the web (cited in the bibliography).

In particular the ACE/TAO web page provides plenty of excellent material and good inspiration.

Questions (& Answers)



Course Evaluation

Take your time to properly fill in the course evaluation