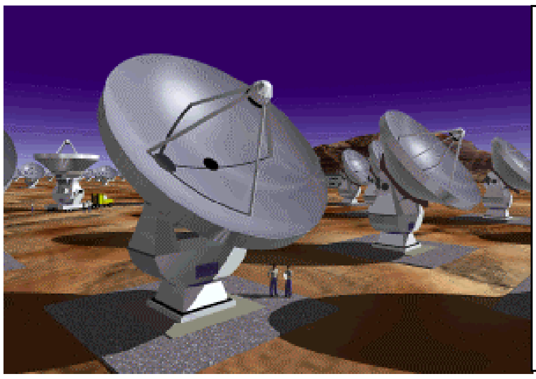

Atacama Large Millimeter Array

ALMA-SW-NNNN

Revision: 0.4

2002-11-17

Software Arch

J. Schwarz

Software Architecture

Software Architecture

J. Schwarz (jschwarz@eso.org), G. Chiozzi, P. Grosbol, H. Sommer
ESO
D. Muders
MPIfR

Keywords: software architecture design ALMA

Author Signature: Joseph Schwarz	Date: 2002-11-17
Approved by:	Signature:
Institute:	Date:
Released by:	Signature:
Institute:	Date:

Change Record

REVISION	DATE	AUTHOR	SECTIONS/PAGES AFFECTED
	REMARKS		
0	2001-12-21	J. Schwarz	Integrated all contributions
0.1	2002-01-25	J. Schwarz	Section 5
	Rewrote & simplified Tech Arch chapter; expanded prototype		
0.2	2002-06-14	J. Schwarz	
	General revision of all chapters, esp. functional & technical architecture		
0.3	2002-10-23	J. Schwarz	All
	Interim version w/changes resulting from reviewers' written comments		
0.4	2002-11-17	J. Schwarz	Sections 4 and 5
	Release for IDR; reworked Functional Architecture; updated Tech Arch		

Table of Contents

1 Introduction 4

1.1 Purpose 4
  1.1.1 Status of this document 4
1.2 Scope 4
1.3 Definitions, Acronyms and Abbreviations 4
1.4 References 5
1.5 Overview 5

2 Architectural Representation 6

2.1 Functional and Technical Architecture 6
2.2 Global Information Flow 6
2.3 ALMA System Overview 9

3 Architectural Goals and Constraints 13

3.1 Runtime Requirements 13
  3.1.1 Functionality 13
  3.1.2 Performance 14
3.2 Development-time Requirements 15
  3.2.1 Distributed development 15
  3.2.2 Modifiability 15

4 Functional Architecture 16

4.1 Global Subsystem Structure 16
4.2 Summary of individual subsystems 19
  4.2.1 Preparation Subsystem 19
  4.2.2 Instrument Operation Subsystems 21
  4.2.3 Data Reduction Subsystems 23
  4.2.4 Archive Subsystem 24
  4.2.5 Administration Subsystem 25
  4.2.6 ACS Subsystem 26
  4.2.7 Interaction between subsystems 27
4.3 Mapping of packages to the WBS 27
4.4 Priorities of package development 28
4.5 Critical issues 29
4.6 Deployment view 30

5 Technical Architecture 35

5.1 Container-Component Model 35
5.2 Definition of entity classes 38
5.3 Generating classes from XML schema 42
5.4 Interprocess Communication and Serialization 42
  5.4.1 Entity Objects in the Instrument Operations Subsystems 45
5.5 Persistence 45
5.6 Versioning 48
5.7 Languages 49
5.8 Bulk data throughput 50
5.9 Security 51
5.10 Technical Prototype 51

Appendix A. Scheduling Block Design: A first sketch 52

1 Introduction

1.1 Purpose

This document provides a comprehensive architectural overview of the ALMA software system, using a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions that will shape the system.

This document is intended for a technical audience including, but not limited to, ALMA software management, ALMA software developers and members of the ALMA Software Science Requirements Committee. It should be read by every software worker new to the ALMA project, or intending to develop pieces of the ALMA software system, because only by sharing the vision conveyed by this document can developers distributed over two or more continents construct a system that meets its functional requirements in a coherent and effective way.

The reader of this document is assumed to be familiar with the contents of the Science Software Requirements Document [SSRD] and the Initial Software Analysis [ISA] referenced below. The present document will be difficult, if not impossible, to follow without such familiarity, since most of the information contained in the preceding documents is not repeated here. A future version of this document will incorporate that portion of the [ISA] that remains valid and is relevant for the architecture.

1.1.1 Status of this document

This document, even after review and approval, remains a work in progress. That is, many of its principles and details will be subject to change and revision during the lifetime of the ALMA software development project. Much architecturally important information cannot be added without the collaboration of subsystem developers who are experts in their individual domains. More changes will arise naturally 1) as subsystem developers encounter the concrete problems of building their subsystems and integrating them into ALMA as a whole; and 2) as the High Level Analysis and Design (HLA) group learns more about the developing system. In particular, a revision of this document should be anticipated for PDR.

1.2 Scope

This document applies to all application- and domain-level software developed for the ALMA project. It affects future versions of the ALMA Common Software (ACS), since it implicitly levies new requirements on the ACS that go beyond those currently implemented or foreseen in the ACS Architecture Document [ACSA].

1.3 Definitions, Acronyms and Abbreviations

A few key terms are defined here. For a complete list of definitions, acronyms and abbreviations, the reader is referred to the Glossary (Appendix XX).

Term	Definition
Architecture	**"…the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the**

	relationships among them." [SAIP]
Package	A grouping of (usually related) classes.
Component	A coarse-grained unit of composition with contractually specified interfaces and explicit context dependencies only. It requires a runtime environment and must be remotely accessible.

1.4 References

1. Lucas, R. et al, *ALMA Software Science Requirements and Use Cases,* Revision 3, ALMA-SW-0011, 2001. **[SSRD]**

2. Schwarz, J. et al, *Initial Software Analysis,* undergoing revision, ALMA-SW-xxxx, 2001. **[ISA]**

3. Fowler, M. and Scott, K., *UML Distilled,* Second Edition, Addison-Wesley, 2000.

4. Bass, L., Clements, P. and Kazman, R., *Software Architecture in Practice,* Addison-Wesley, 1998. **[SAIP]**

5. Chiozzi, G., Gustaffson, B. and Jeram, B., *ALMA Common Software Architecture,* ALMA-SW-0016, 2001. **[ACSA]**

6. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995. **[GOF]**

7. Scott, S. and Myers, S., Momose, M., *Data Rates for the ALMA Archive and Control System*, ALMA Technical Note, 2002. **[SMM]**

1.5 Overview

The ALMA software system is designed to transform user input into an approved observing proposal, to perform the observations specified therein, and to deliver calibrated data (often including further data products such as images and spectra) to the user and to an observatory archive. From the information retained in this archive, not only the original observer, but also other interested astronomers, can do further research and refined analysis of the observational data.

Software to manage and administer proposal preparation, instrument operations and archival use is included in the scope of the software project.

2 Architectural Representation

The main objectives of the architecture are to present the organization of the software system, describe its structural elements and their behavior, and compose these structures into larger subsystems. The general architecture has been driven by the high-level Use Cases defined in the Software Scientific Requirements document [SSRD]. The main structural elements of the system have been identified in the Initial Software Analysis document [ISA], which was based on these Use Cases. For this reason, this document does not include an explicit Use Case View of the system.

2.1 Functional and Technical Architecture

Software Architecture has two facets: one is the functional architecture, describing the components, their responsibilities and interfaces, and their primary relationships and interactions. The technical architecture on the other hand describes technical aspects, such as remote access, streaming, threading, activation, and transactions. Both are architecture: two different views that go hand in hand.

The description of each part, however, is directed towards a different audience. The technical architecture must be understood by infrastructure (container/ACS) developers while the functional architecture needs to be understood by the application developers. A programming model will be extracted from the technical architecture in order to show application developers how to use the provided technical infrastructure.

The detailed descriptions of these two aspects of the architecture are presented in Chapters 4 and 5, respectively. A high-level description of the system, with emphasis on the *functional* aspects is given in the following sections of this chapter. A generic deployment view is provided in Chapter 4 as part of the functional considerations. Explicit process and implementation views are not given at this stage and will first be defined in detail at the subsystem level.

In the logical view, several common high-level design issues are considered. This was done to outline global architectural and design decisions that must be accommodated by all subsystem designs.

2.2 Global Information Flow

The ALMA software system is envisioned as an end-to-end data flow system, which handles the information and operations required to conduct all tasks from the time an astronomer creates an observing proposal until the resulting data are returned. This type of end-to-end observatory system was pioneered by Space Telescope Science Institute for HST and European Southern Observatory for VLT. The experience with these systems has been used as a guide for the ALMA system architecture. A schematic representation of the system is given in Figure 2-1, where its main elements are shown.

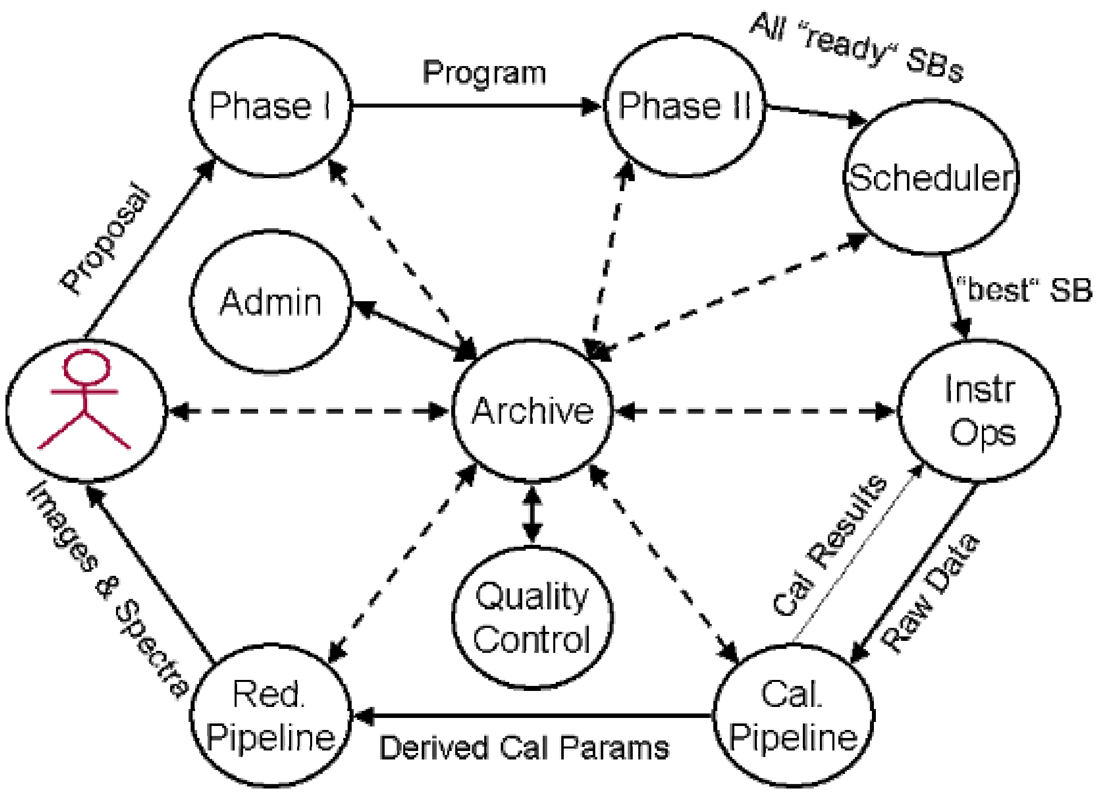

Figure 2-1: ALMA System Dataflow (Schematic). The outer solid lines show the logical data flow; the dashed lines directed to/from the Archive indicate that a) all data is saved in and can be retrieved from the Archive; and b) that the logical data flow may—when appropriate—be handled by the Archive rather than via direct process-to-process communication. Note also the feedback of, e.g., pointing, focus and phase calibration results from the Calibration Pipeline to the ALMA Observing Process.

One may view the system either from the perspective of an end user (*i.e.*, an astronomer) or an observatory. The astronomer is interested in how an observing project flows through the different parts of the system, as illustrated by the outer set of lines in Figure 2-1. The user (shown as the actor symbol on the leftmost side) initiates the cycle by creating and submitting a proposal for ALMA observing time to Phase I Preparation. After the proposal has been reviewed and (hopefully) accepted it is turned into an Observing Project/Program and goes to Phase II Preparation where the actual observations are fully specified by creation of Scheduling Blocks (SB).

Once each SB has been defined it is stored in the Archive and considered for scheduling whenever available observing conditions and array resources make its execution feasible. If all other factors are equal, the ready-to-run SB with the highest scientific ranking will be chosen and executed/observed by the ALMA Instrument Operations subsystem. The execution of an SB generates a set of raw data, which is both saved in the Archive and forwarded to the Calibration Pipeline (in the case of data to be used for calibration) and/or the Reduction Pipeline for processing. The Calibration Pipeline processes, among other things, focus, pointing, and phase calibrator data, feeding the appropriate results back to 1) the ALMA system to modify observing parameters in near real time; 2) the dynamic scheduler to allow it to select the next SB to observe (this latter feedback is not shown in the Figure). It saves these and other results in the Archive for use by the Reduction Pipeline.

The Reduction Pipeline is responsible for producing the quick look and final images and/or spectra, applying whatever corrections are necessary. Finally, the astronomer receives the results of the observing project in the form of calibrated images and/or spectra together with associated information (*e.g.*, logs). As data accumulate in the Archive, Quality Control performs trend analyses of them. The results can then be used to *e.g.*, correct final data products or initiate maintenance actions.

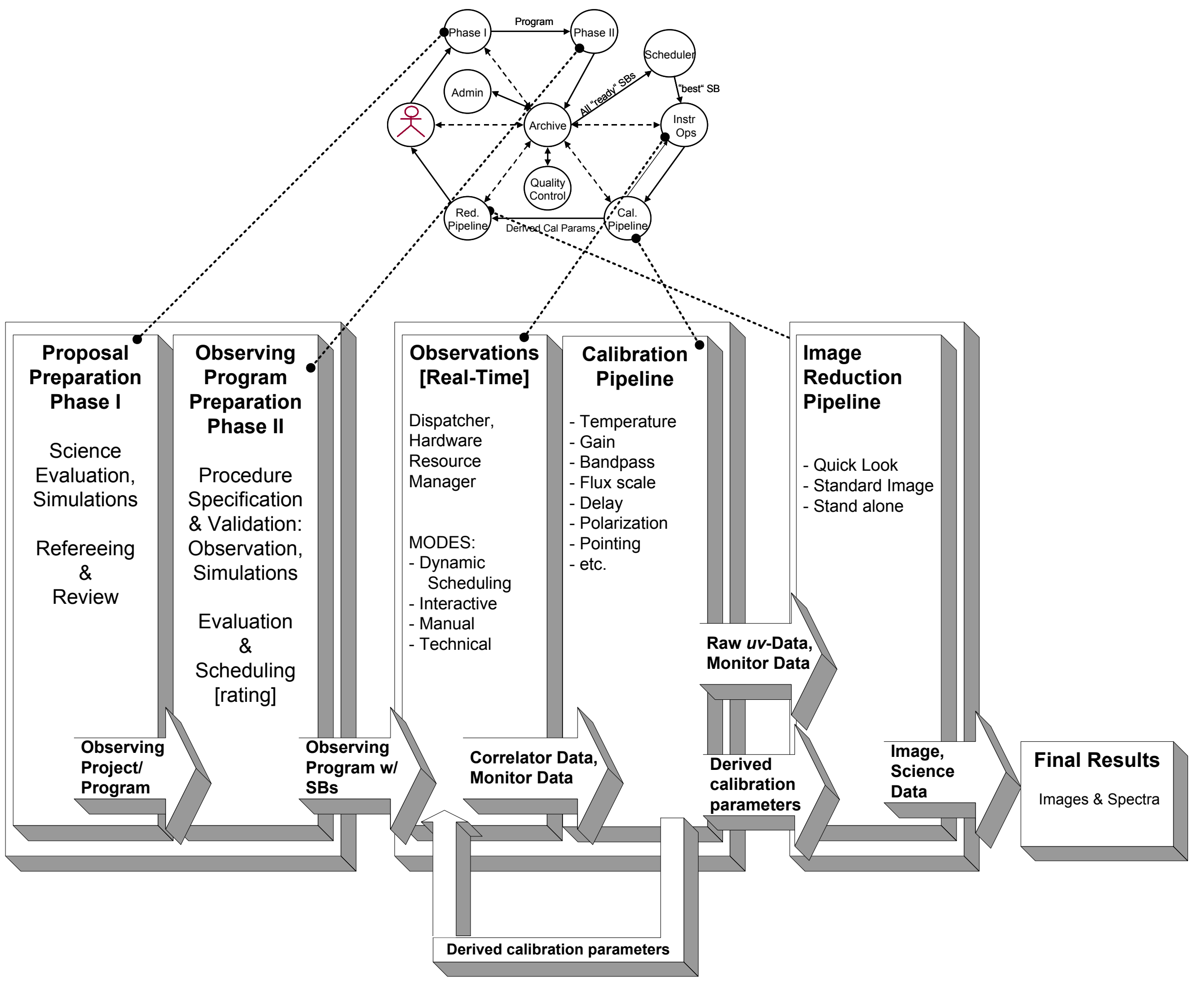

Figure 2-2 ALMA Online Dataflow, with a more detailed view of the processes. The arrows indicate flow of data; the five tall boxes represent major logical groupings of functionality. Although all data is stored in and retrieved from the Archive, this fact does not alter the progressive flow of data from Proposal Preparation to Final Results. The term "Standard Image" is used to indicate that observatory-specified procedures are used to produce the image that is normally delivered to the PI. For simplicity, the Archive itself is omitted from this figure, but the relationship with the preceding diagram is shown at the top.

Another access route for a user is archival research, where requests for data are made directly to the Archive (possibly after some review). In case raw data are requested they may be passed through the Pipeline to yield images and/or spectra, benefiting from calibration and processing techniques that may have been improved since the original data were acquired. Figure 2-3 shows the general flow of data for this case.

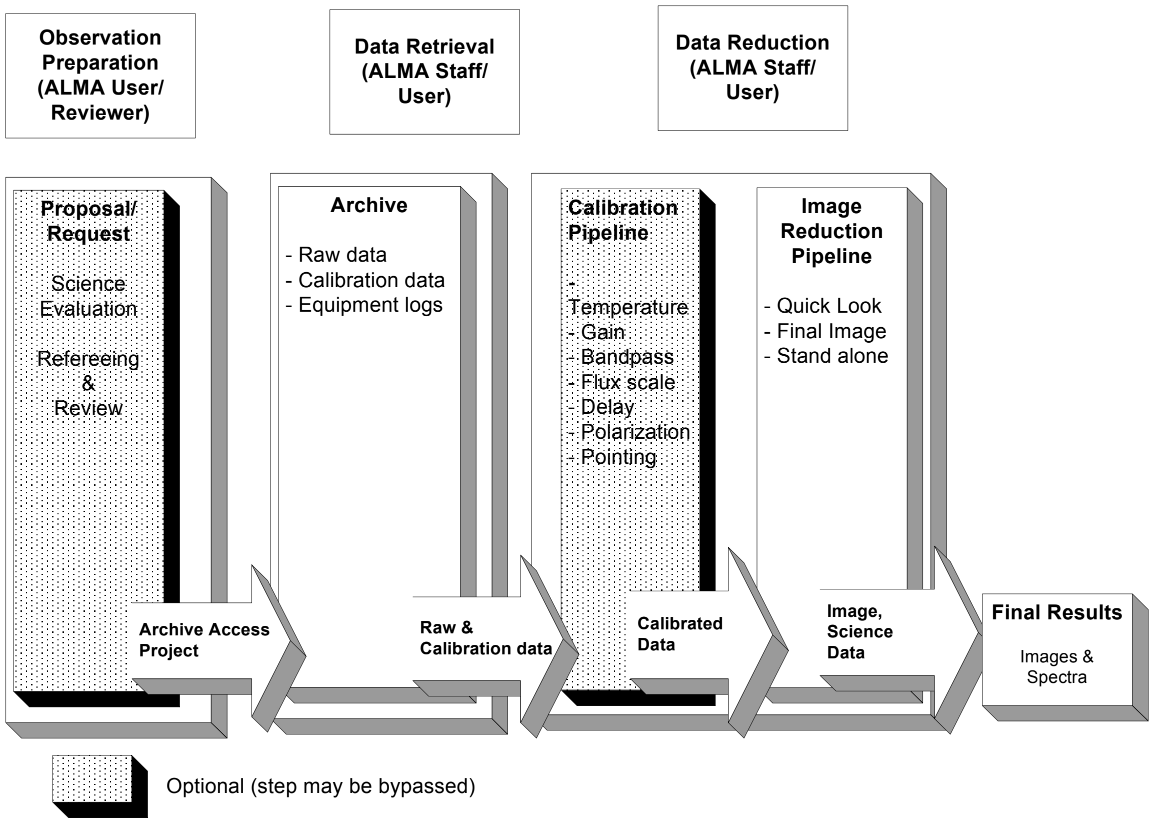

Figure 2-3 ALMA Archival Research (using "Offline System"). As in the previous diagram, arrows represent data flow and tall boxes represent logical groupings of functionality.

The observatory has a somewhat different view of the flow of information, as its prime interest is to optimize the efficiency with which it can process a large collection of observing projects and at the same time guarantee that all scientific requirements are fulfilled. This view is directed principally towards the Archive, which is the central repository for observing projects, data acquired and status and availability of equipment. The main functional elements (*i.e.* Phase I/II, ALMA, Pipelines, Quality Control and Administration) all interact with the Archive to obtain and deliver data but can otherwise operate as independent peers. To ensure high global performance, each element must be optimized individually and have adequate access to the Archive.

The ALMA real-time system should be the only one that limits the total system throughput, that is, all other elements must be carefully programmed and scaled so that they do not constitute bottlenecks. Besides operating the functional elements, the observatory monitors the overall performance of the system through the Administration and Quality Control elements. They provide tools to assess the status of observing projects and to analyze trends to determine data quality and performance of individual parts of the system.

2.3 ALMA System Overview

The general structure of the ALMA software system was derived from the requirements and data flow considerations given above and consists of 5 top-level subsystems with the following responsibilities:

1. **Observation Preparation:** includes the Phase I and II preparation of observing projects and the associated peer review process. It is the main interface for astronomical users to the ALMA facility and provides them with easy to use Graphical User Interfaces (GUIs) for the detailed specification of observations. The main product generated is Observing Projects with their Scheduling Blocks, which contain all information required to perform the corresponding observations.

2. **Instrument Operation:** is responsible for the actual observations specified by Scheduling Blocks and is the only subsystem with true real-time behavior. All observing hardware at the ALMA site (*e.g.* antennas, receivers and correlator) is controlled and monitored by this subsystem. It is the focus of the system since it acquires data. Although it can work automatically by using dynamic scheduling, it will normally be supervised, or at least checked regularly, by a human operator.

   An important goal of instrument operation is to reconcile the real-time constraints of the hardware with the logical time operation of this subsystem. Many hardware components must be synchronized to an array-wide 48 ms timing pulse and the division between real-time and the logical run-time should remain close to the hardware in order to isolate this dependency as much as possible.

3. **Science Data Reduction:** The data acquired are calibrated and reduced by this subsystem, which also checks the quality of the final data products. Its primary role is to process data that have just been acquired (along with archived data belonging to the same project, possibly from a different antenna configuration), but can also reduce data obtained from the Archive. In its observation support role, it will provide Quick Look images and other information to help assess the quality of data just acquired.

4. **Archive:** provides a central repository for all persistent information of the ALMA facility such as Observing Projects, raw and reduced science data, logs of all operations and schedules. Whereas it is logically one repository from which any subsystem can request information, it may well be distributed over several locations in order to optimize performance and increase redundancy. This subsystem also provides external users with facilities to identify and request science related data from the archive.

5. **Administration:** Many administrational tasks have to be performed to ensure that the entire ALMA facility operates smoothly and efficiently. They include long term planning for the array configuration, scheduling of maintenance, management of user accounts and associated privileges, checking the state of Observing Projects, production of data packages for end users and generation of reports on performance of different parts of the facility.

Further, the science software requirements specify that the system must provide a number of key features:

1. to perform all necessary tasks through easy-to-use GUIs.
2. to “provide simple ways for the staff or expert astronomers to refine observing modes and develop new ones” ([SSRD] 3.1.0-R2).
3. to support automatic, interactive, manual and technical modes of operations

Here the Observation Preparation is responsible for generation of an Observing Project and associated Scheduling Blocks, which it places in the archive. Instrument Operations fetches ready Blocks from the archive and schedules them for observations depending on their science rating and needs such as weather conditions and array configuration. This is done by sending commands to the ALMA hardware for performing the real-time control of correlator, antennas and receivers. The data acquired are streamed directly to the Calibration pipeline and to the Science Data Reduction pipeline (operating in Quick look mode) and are saved in the Archive as well. Instrument Operations receives feedback from the Calibration pipeline in order to adjust observing parameters in near real time. In parallel with the observing process, Science Data Reduction generates Quick Look images and data quality information to enable the operator and observer to monitor the on-going observations. From the Archive, Science Data Reduction obtains all data needed to generate the observatory's final[1] data products for storage in the Archive and access by the Proposer/Observer, and—after expiration of a proprietary period—by the astronomical community in general.

The flow of data through the system is checked by Administration, to ensure that Projects are executed in a timely manner. It alerts Operations personnel if Projects are suspended, missing data or otherwise not following the standard path. Finally, it will contact the users when Projects reach breakpoints or are completed and provide all relevant data products to them.

An astronomical User will normally interact with the system through tools provided by Observation Preparation regardless of whether his/her Project is to be observed in dynamically scheduled or interactive mode. The main difference between dynamic scheduling and interactive mode is that Instrument Operation will execute Scheduling Blocks created immediately in the latter case while they will be queued to wait for optimal conditions in the former. In the case of archival research, users may search for and request data directly from the Archive. Operations at the ALMA observing site will be supervised by an Operator, who can check the status of correlator, antennas and receivers, change operation mode (dynamic/interactive) and administer other resources through Instrument Operation. General managerial tasks are performed by administrators through the Administration subsystem, which in turn accesses information in the Archive.

Storing all persistent information in the archive makes the system less coupled so that Observing Preparation, Instrument Operation and Science Data Reduction can work independently as long as they maintain the average flow to and from the archive. This isolates Operations from possible disturbances in other subsystems but makes the archive a critical component, which must have a very high availability.

The only part of the system that has true real-time requirements is Instrument Operation as it controls the antennas, receivers and correlator. Several other processes in this subsystem are also time critical such as calibration procedures and safe storage of acquired data. In order to avoid time limitations due to simultaneous access to the archive, faster more direct access methods (that bypass the archive) may be deployed in such cases.

Whereas the system in terms of transaction rates, security and redundancy does not present major challenges for current software technology, several other properties of the ALMA system must be considered in depth in the detailed design. They include:

[1] "Final," of course, only as far as observatory processing is concerned; the observer will normally need to perform further processing and analysis in the course of his/her research.

1. Very high data acquisition rates. The ALMA baseline correlator produces of order 1 Gbyte/s, and it is the task of the correlator subsystem to reduce this in real time, leaving the rest of the system to deal with a smaller, but still substantial 6 – 60 Mbyte/s.

2. Large volume of archived data (~ 180 Tbyte/year)

3. Real time synchronization of hardware over large distances. ALMA antennas can be spread out over more than 10 km. The correlator itself may be located ~ 50-80 km from the antenna site.

4. Dynamic, optimized scheduling of observations depending on site conditions

5. High processing needs to reduce data

6. Flexibility for developing new observing and data-reduction modes.

3 Architectural Goals and Constraints

The architecture of the ALMA software system must enable the development of software that satisfies both runtime and non-runtime requirements. It is necessary, but not sufficient, to provide a system satisfying the enumerated requirements lists of the [SSRD]. Like all software, the ALMA system will develop and change over time, and its architecture must foresee and support such evolution.

3.1 Runtime Requirements

3.1.1 Functionality

The Science Software Requirements can be summarized as follows: the ALMA software will play a major role in all aspects of the operation (in its most general sense) of the ALMA Observatory. This includes:

- Administration of day-to-day activities on the ALMA site and the Operations Support Facility (OSF)
- Electronic preparation and submission of observing proposals by prospective users of the Observatory
- Validation, review, selection and prioritization from these proposals
- Transformation of observing proposals into ready-to-run observing programs
- Scheduling and execution of observing programs according to scientific priority and suitability of observing conditions
- Near real-time feedback to the operator (and in some cases, to the observer) as observations proceed
- Automatic calibration of raw data and production of final images and spectra.
- Archival of all raw, processed and ancillary (e.g., monitor/engineering) data
- Support for archival research, once the proprietary period for a data set has expired. This will include compatibility and interoperability with other elements of the Virtual Observatory (VO).

3.1.1.1 What's different about ALMA?

While ALMA builds on the experience of existing radio observatories, it nevertheless represents a major step forward, because:

- It must be accessible to the entire astronomical community, including those with no experience in aperture synthesis and mm/submm radio astronomy. While this has been a goal of the WSRT and the VLA as well, ALMA's 2016 baselines and the technological advances of the last 20-30 years should enable it to attract a larger body of non-radio astronomers.
- It must, on the other hand, enable experts to customize observing and reduction procedures.

- It must operate predominantly in service mode, without the controlling presence of the original proposer.
- It must maximize the scientific productivity of the observatory by dynamically scheduling and executing observing projects in response to changing observing conditions.
- It must provide an archive of calibrated and imaged data that can be used by researchers without needing recourse to the specialized knowledge of the original observers once the proprietary period for PI (Principal Investigator)-data access has expired.
- It must deal with a larger variety of projects than previous aperture synthesis instruments
- At $\nu > 345$ GHz interferometry is in its infancy, started recently at the Submillimeter Array (SMA) in Hawaii

Some operations must be available even if other parts of the pipeline are unavailable. The proposal and observation preparation system for example, must operate regardless of whether or not the observation system is running. And the observation system must run, once it has some defined observations, without the proposal system.

3.1.1.2 Physical hardware

Another constraint is the commitment to physical components, some of which we know now [December 2001]: 64 antennas, computers onboard each antenna with a local field bus connecting to devices, high speed communication links from each antenna to the center in a star point-to-point configuration, central correlator, central control computers, communication links between components and to the rest of the world – these show us the scale of the system.

3.1.2 Performance

3.1.2.1 Hard Real-time

3.1.2.2 Near real-time

Results of pointing/focus and observations of astronomical phase calibration sources must be available quickly enough to be fed back to the observing process. In the case of pointing/focus calibrations, further observations of the scientific target cannot proceed until a satisfactory pointing/focus solution has been determined. Rough turnaround times of 0.5 s are needed.

Some quick-look results should be updated and displayed to the operator every scan.

3.1.2.3 Interactive

The proposal tools must give the user immediate feedback and must be generally responsive. When network access (e.g., to catalogs) is unavailable or unacceptably slow, fallback to use of local resources, perhaps with reduced functionality, is required. Array operators should be notified immediately of system failures and should have the means to react quickly.

3.1.2.4 Overall throughput

The requirement here is that the entire processing system be fast enough to keep up with the pace of observations, that is, that the incoming data be calibrated, imaged and archived as fast as it is acquired. Neither calibration nor imaging is allowed to be a bottleneck for the system.

The current specifications for the data rate that must be supported by the post-correlator part of the system are 6 Mbyte/s average and 60 Mbyte/s peak. A change request is pending to increase this to 12 and 72 Mbyte/s, respectively. (Note that the baseline ALMA correlator can deliver 1 Gbyte/s, but the correlator subsystem reduces this to the above rates, which represent the data that is seen by the rest of the system—and represent the data which is archived.)

3.2 Development-time Requirements

3.2.1 Distributed development

The ALMA software will be developed at many institutes, scattered over at least two continents. The architecture is therefore required to facilitate distributed development, promoting independent testing, clarity of interfaces and ease of integration.

3.2.2 Modifiability

The experimental nature of the ALMA project cannot be overemphasized. Some requirements will only become clear as the system comes into operation. Early results will stimulate the development of new requirements, particularly in the relatively unexplored sub millimeter range.

The architecture of the ALMA system should compartmentalize the areas of likely change, so that the impact of future changes will be restricted to relatively few areas of the system. This implies

- High level of flexibility at those levels where considerations of robustness and performance make the effort required reasonable
- Observers and ALMA scientific staff (i.e., those who are not software professionals) should be able to develop and implement new observing methods and reduction procedures, although incorporation into the system as standard observing modes may require intervention of a software specialist.
- The architecture chosen must be scalable enough to allow the addition of new array hardware without a disproportionate amount of effort. Such hardware could include, for example:
    1. A next-generation correlator, with its implied many-fold increase in data rate
    2. Additional receiver bands
    3. A second, smaller array (the ALMA compact array)
    4. Should the ALMA project be expanded to include Japan as a partner, such hardware might come sooner rather than later.
    5. The architecture must permit the retargeting of parts of the system to new computer hardware that will be available over the observatory's lifetime of several decades.
    6. The architecture must permit the migration of archived data to new hardware media and new forms of data access software with minimal impact on the remainder of the system.

4 Functional Architecture

In this chapter we show the internal structure of the global subsystems introduced in chapter 2. The functionality is provided by a number of interacting software packages each consisting of several classes. The main goal was to decouple the major logical subsystems so that they can be developed and operated independently or with few dependencies on other subsystems.

Note that we have not specified the interfaces between individual subsystems. This process is now underway, and involves the subsystem teams directly, as they define what information they will need from other parts of the system. Some key data elements, in particular Observing Projects and Scheduling Blocks, have, however, been given a preliminary definition by the High-Level Analysis group; this definition is currently being elaborated and refined in collaboration with the subsystem teams concerned.

4.1 Global Subsystem Structure

The overall flow of data and control among the ALMA subsystems is shown in Figure 4. It will be useful to briefly state the main functional paths within this system in order to get a clear understanding of the role of the various subsystems.

First, an observer creates an observing project using the Observing Tool, which breaks the project into scheduling blocks, and stores it in the archive. Then, the scheduling subsystem gets project definitions and scheduling blocks from the archive, dispatching them to the control system to be executed. The control system executes the scheduling block by commanding the correlator, which results in raw data and meta-data being made available to the calibration and quick-look pipelines. The completion status of scheduling blocks is monitored by the scheduling subsystem, which starts the science data reduction pipeline at appropriate times. The science pipeline generates data products that are stored in the archive. Again the scheduling subsystem monitors the completion of the science data reduction and informs the PI that data are now available.

Also shown in the figure, although outside the more-or-less automatic operations flow of the system, is access by researchers to data in the Science Research Archive.

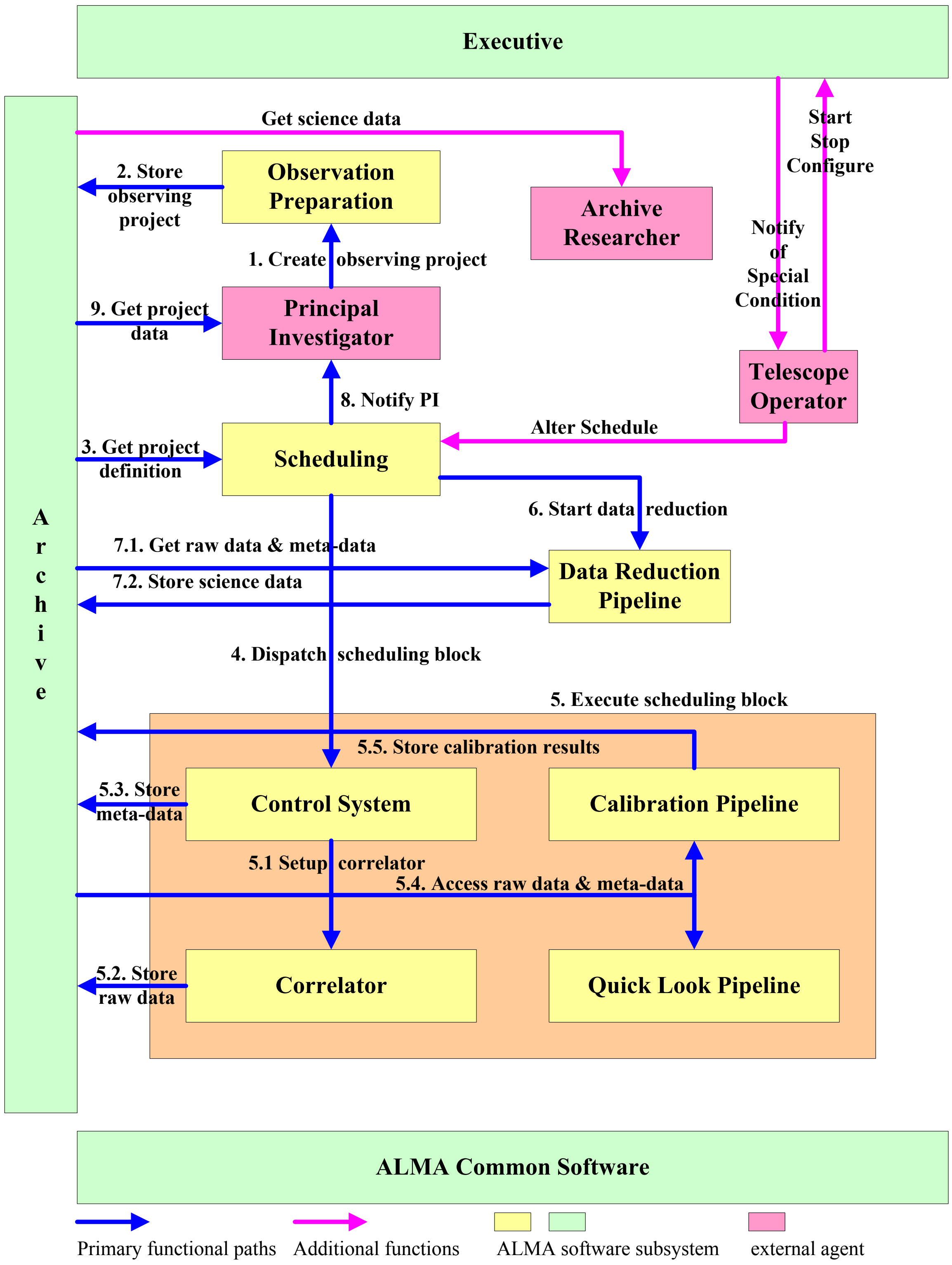

Figure 4: Operation of the ALMA software system. The numbered arrows indicate steps in the creation and processing of an Observing Project from Proposal through to data reduction and storage in the Archive.

The ALMA software architecture (*i.e.*, its software structure, in contrast with its run-time control and data flow) consists of 4 layers: the presentation layer (user interfaces), the application layer (providing the main tasks), the domain layer (commonly used objects) and the system layer (system services, communication between subsystems). The layered structure helps decoupling the subsystems. Figure 4-5 shows the top-level subsystems of those 4 layers. For simplicity the dependencies and interactions are not shown here. They will be discussed later. For a more detailed specification of the packages discussed, see the [ISA].

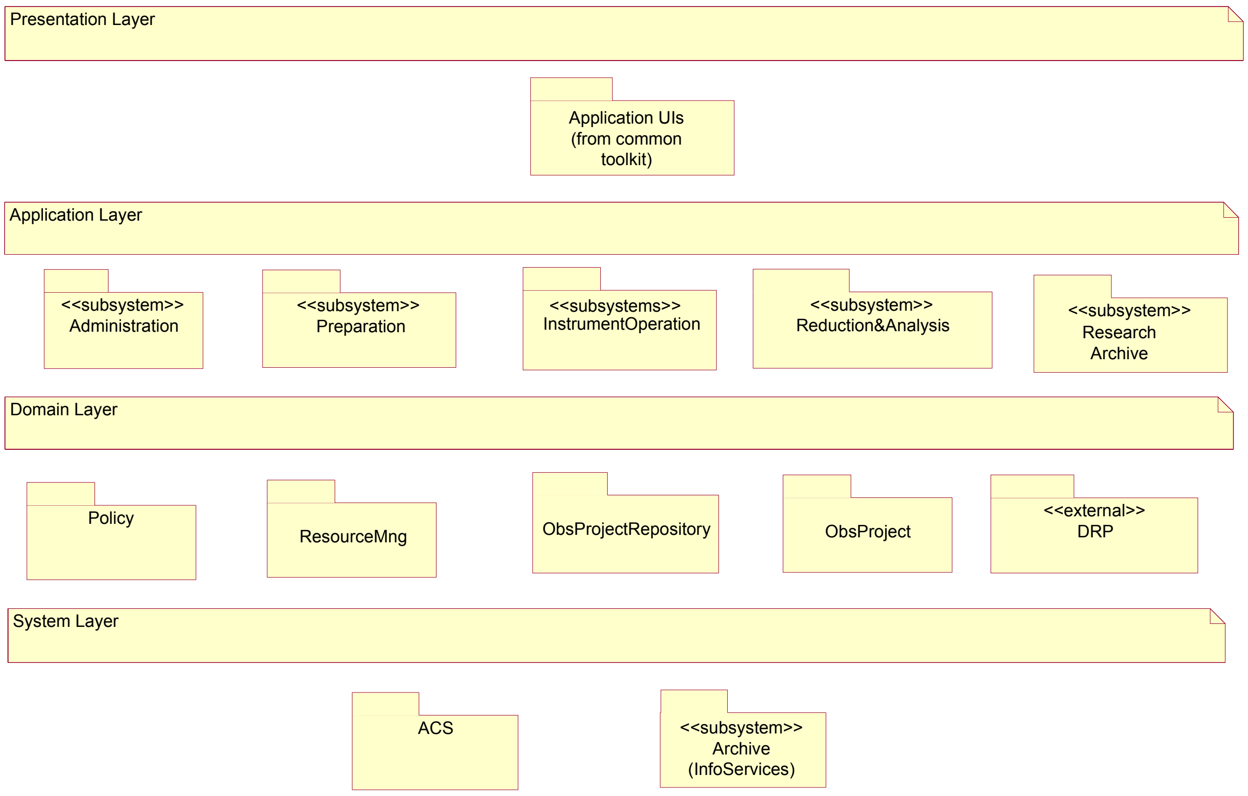
Presentation Layer
Application UIs (from common toolkit)
Application Layer
<<subsystem>> Administration
<<subsystem>> Preparation
<<subsystems>> InstrumentOperation
<<subsystem>> Reduction&Analysis
<<subsystem>> Research Archive
Domain Layer
Policy
ResourceMng
ObsProjectRepository
ObsProject
<<external>> DRP
System Layer
ACS
<<subsystem>> Archive (InfoServices)

Figure 4-5: Global layered subsystem and package structure of the ALMA software system.

The system layer provides the basic communication and archival storage functionality commonly used by almost all other subsystems. This layer will soon have to be stable because everyone else depends on it.

The domain layer provides commonly used entity objects such as the Observing Project and Catalogs as well as the externally-developed Data Reduction Package. The entity objects are also central to the ALMA software system and therefore need to be developed and made as stable as possible early on. This does not exclude, of course, the possibility of changing their definition when experience and/or new requirements make this necessary.

The application layer subsystems implement the main sections from chapter 2, but there is no precise correspondence, because the view in Figure 4-5 is a *static* view which is more relevant to the development of the software than to its run-time organization. “Proposal Preparation” and “Observing Program Preparation” are handled by the Preparation & Administration subsystems, “Observations” are covered by the InstrumentOperation subsystems (a group of six subsystems in their own right, including “AlmaSystem” from the Domain Layer), while “Image Reduction Pipeline” corresponds to the ScienceDataReduction subsystem. The “Calibration Pipeline” and the “Quicklook Pipeline” are—from the run-time point of view—in “InstrumentOperation”, but the software itself is made up of units from the “Reduction & Analysis” package. The

Archive subsystem is the hub around which the system is organized (the Research Archive is itself a client of the general Archive subsystem). “Administration” is also responsible for all other ALMA administrative tasks (see 4.2, Summary of individual subsystems).

Finally, the presentation layer provides the user interfaces. It is planned for the ACS subsystem to provide a general ALMA GUI toolkit that the other subsystem developers will use.

4.2 Summary of individual subsystems

This section describes the structure of the individual subsystems. The definition of a subsystem is mainly based on the five application layer systems, but development considerations and the fact that the project Work Breakdown Structure definition preceded the development of the system architecture have caused us to be less precise in what we consider a subsystem. Only the “ACS” subsystem was kept separate because of its system wide importance and need for stability (see section 4.4). In the figures we already indicate the package development priorities as numbers in brackets (see section 4.4 for a detailed discussion of priorities).

4.2.1 Preparation Subsystem

The Preparation subsystem (figure 4-2) covers the ALMA software from proposal submission to the creation of observable Scheduling Blocks (SBs). Hence, it consists of tightly interacting packages of the observing preparation stage, which make up, with the observing tool at its center, a logical unit that is almost independent from the rest of the system.

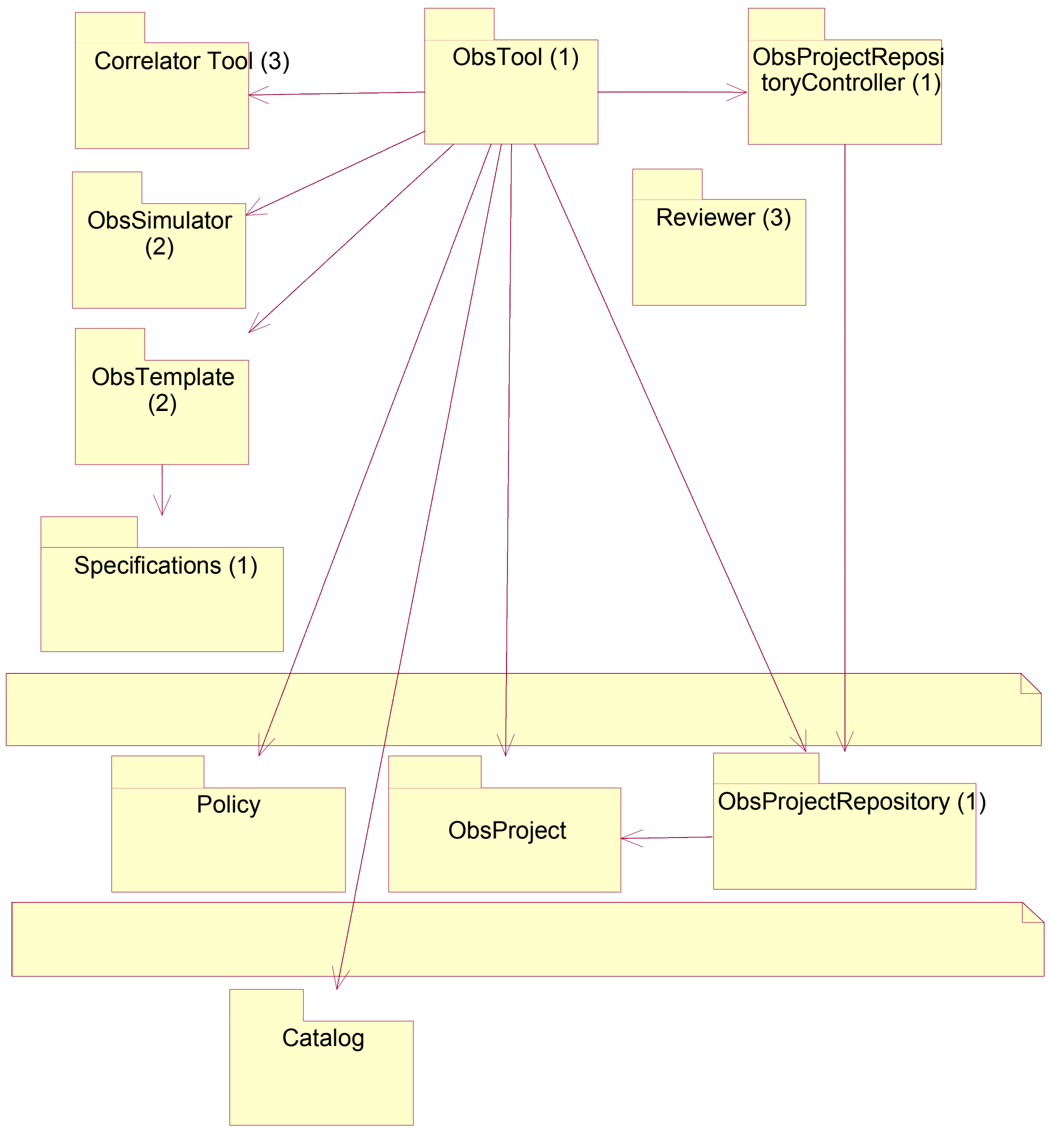

Figure 4-6: Observation Preparation Subsystem (The numbers in parentheses indicate the development priority assigned to each package within the Preparation Subsystem)

The ObsProject portion from the domain layer may need to be exported as a separate subsystem because many other subsystems depend on it. For the same reason the catalog package may need to be kept with the ObsProject package.

We have the following further comments on individual packages: The simulator package that forms part of the Observation Preparation subsystem will be a lightweight version; the subsystem will be able, however, to make use of a full-fledged simulator that will be provided with the offline reduction and analysis subsystem (appearing as a WBS element but not otherwise discussed in this document). The correlator tool and specification packages need a comprehensive knowledge of the instrument’s capabilities. The ObsProjectRepositoryController is a (possibly web-based) controller subsystem that mediates between the application layer and the ObsProjectRepository in the domain layer.

4.2.1.1 Interface to Instrument Operations

The primary interface between the Preparation subsystem and the Instrument Operation subsystem are the Observing Project, along with its linked Proposal, Programs and Scheduling Blocks (see [ISA] for a class diagram of this hierarchy). Each Scheduling Block will contain an observing procedure, either:

- Coded in XML, to allow full validation and consistency with the high-level input given by the PI (see [SSRD], [ISA] and [OTC]) and later compilation into either 1) scripting language commands or 2) a set of compiled language invocations;

OR

- Direct scripting language commands, in particular if an “expert-level” script has been defined by the proposer. In this case, some kinds of validation will not be possible.

A partial and preliminary definition of the content of a Scheduling Block is given in the Appendix.

4.2.2 Instrument Operation Subsystems

The actual ALMA observing process is implemented in the Instrument Operations, which consists of six subsystems.

Alma Executive: supervises and monitors the other subsystems;

Scheduling: determines the next Scheduling Block to execute and updates the status of the SB and its enclosing Observing Project upon success or failure; initiates Science Data Reduction processing and/or notifies the PI when necessary (e.g., upon reaching a breakpoint or completing a significant piece of the Observing Project);

Antenna & Receiver Control (*Monitor & Control?)*: accepts an SB from the Scheduling system and executes its commands to the array hardware, configuring and triggering the acquisition of data by the correlator;

Correlator: accepts data from the correlator hardware, Fourier-transforms it from time- to frequency-space, applies Water Vapor Radiometer (WVR) corrections, and streams its output to the Archive, the Calibration Pipeline and the Quicklook Pipeline;

Calibration Pipeline: calibrates data from the correlator in near-real-time, archiving results and passing them to Antenna & Receiver Control to modify observing parameters, and to the Scheduling subsystem to use in selecting the next SB to execute.

Quicklook Pipeline: applies available calibrations to the data, calculates dirty images and displays these and other data for monitoring by the ALMA staff and, optionally, by the PI.

Operations are supervised by the Alma Executive subsystem, which starts up and monitors the other subsystems and will not be discussed further in this section. The Scheduler provides the dynamic ranking of Scheduling Blocks. The Dispatcher (within the Scheduling subsystem) requests Scheduling Blocks either 1) from the Scheduler when dynamic scheduling is in operation; or 2) from an Interactive Observer, who uses the OT to submit Scheduling Blocks for immediate execution. It then passes the chosen SB to the Sequencer, which executes the observing procedures contained within the SB and sends commands (defined in the Command package) to the Correlator and the Antenna & Receiver Control subsystems to perform the actual observations. (The dashed box labeled "Alma System" denotes the Correlator and Antenna & Receiver Control subsystems, our abstractions for the ALMA hardware and all the distributed objects and control software associated with it.) Upon completion (or abort) of an SB, the Dispatcher must notify the Observing Project Manager (also a package within the Scheduling subsystem) which will update the SB's status, as well as that of its enclosing Observing Project. The Observing Project Manager will determine whether this update, in turn, should trigger any data processing (*e.g.*, final image production when a Project is completed or a Breakpoint is reached at which the PI had requested such processing) and will evaluate any conditions that might cause the status of other SBs in the Project to change (*e.g.*, because a Breakpoint has been reached, or because other SBs will only become ready for execution after the current SB has executed successfully).

The Scheduler itself depends upon the SiteCondition package (see [ISA]) for information concerning weather conditions and instrumental performance necessary to rank SBs for execution.

A crucial part of the subsystem is the calibration package, which will provide all functionality for proper array / instrument / data calibration. The subsystem also consists of several domain layer packages, whose functionality is closely related to the instrument operation.

4.2.2.1 Interface between Correlator/Control Subsystems and Pipelines

In order to meet the requirements for rapid turnaround of calibration and quicklook results after raw data has been acquired, two key design decisions have been made:

Data coming from the Correlator subsystem will not be retrieved from the Archive by the Calibration and Quicklook Pipelines, but will be streamed directly to them, possibly via specialized hardware (see the discussion of the "Fast Data Store" in the chapter on Technical Architecture).

Both the Calibration and Quicklook Pipelines will decide which parts of this data are of interest to them, based on *metadata* inserted into the data stream by the Correlator subsystem, and an additional metadata stream coming from the Antenna & Receiver Control subsystem. This metadata will comprise:

- All parameters characterizing antenna, receiver and correlator configuration during the acquisition of the data that follows:
- An indication of the *scientific intent* (*e.g.*, phase calibration, pointing calibration, pointing scan), which will have been embedded in each scan command before it is executed.

In this way, the operation of these two pipelines will be fully data-driven.

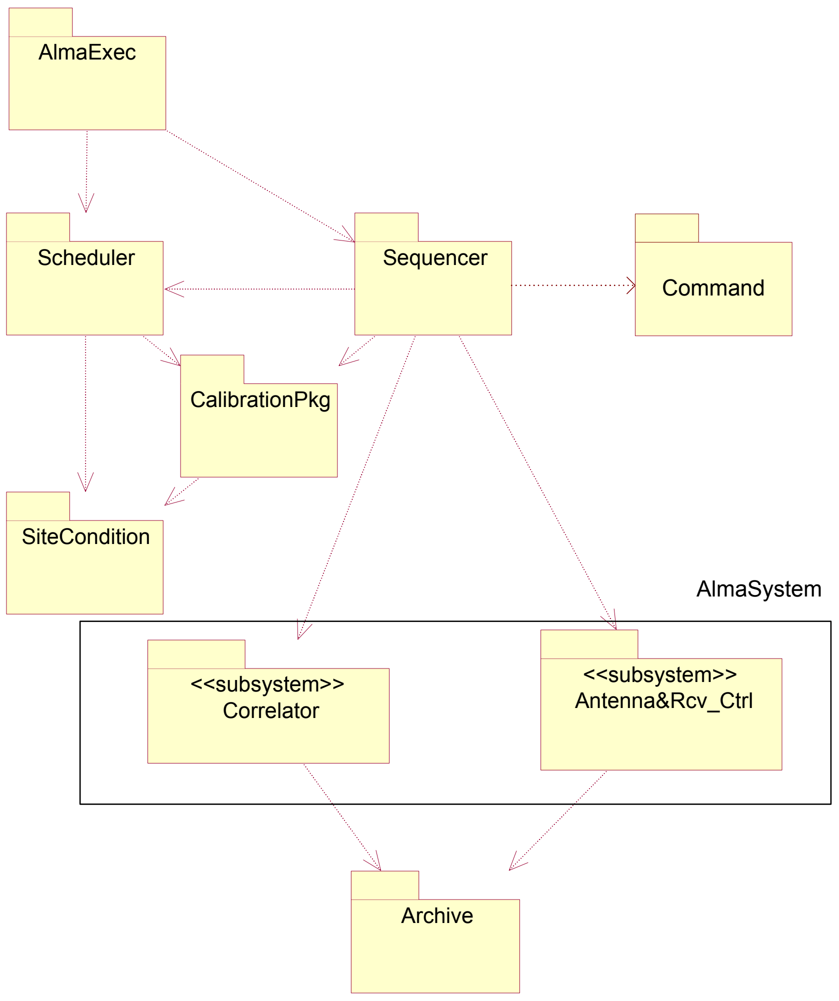

Figure 4-7: Instrument Operations Subsystems. The Sequencer issues configuration and control commands to the Correlator and Antenna & Receiver Control subsystems. Both science (uv & total power) and engineering ("monitor") data produced by these subsystems are made accessible to other subsystems via the Archive (via direct streaming to the Calibration and Quicklook Pipelines, which need to process this data in near-real time). Only the most significant dependencies are shown.

4.2.3 Data Reduction Subsystems

Data reduction in ALMA appears in three guises: 1) on-line calibration to enable evaluation and optimization of on-going operations; 2) generation of quicklook results to give feedback to the operator and/or PI; and 3) generation of "definitive" data products. These three processes will clearly share most of their basic functionality, and will be assembled from pieces of software functionality provided by an existing Data Reduction Package (DRP), probably AIPS++. The Quicklook Pipeline in particular can be regarded largely as a subset of the full Science Data Reduction Pipeline, although it has its own unique requirements for rapid display of results.

Further data reduction and analysis will be done by PIs at the ALMA Regional Support Centers and/or at their home institutions. For this purpose, they will use an ALMA-specific Data Reduction User Interface (DRUI) to allow them to reduce and analyze their data interactively. At least at the beginning, the DRUI will make use of parts of the above ALMA pipelines and the external DRP, although it is intended to be general enough to allow the substitution of a different DRP if the user so desires.

The minimal analysis that has been done on this area of the system is clear from the oversimplified figure. Since it is very likely that AIPS++ will be the DRP selected, the modality for integration with the rest of the ALMA software needs to be addressed in collaboration with experts in AIPS++ internals and with the developers of the Calibration and Quicklook Pipelines and of the DRUI.

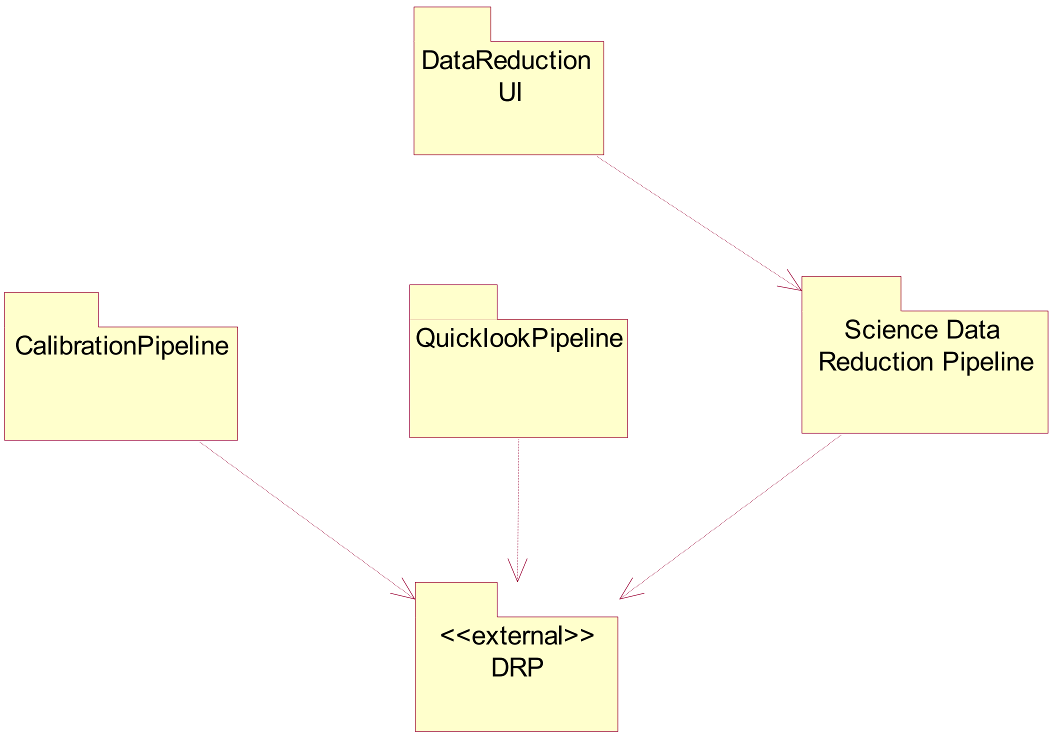

Figure 4-8: Data Reduction Subsystems. The "DRP" (Data Reduction Package) is expected to be already-existing software that has been developed outside the ALMA project, and will have been enhanced where necessary to meet the needs of millimeter and submillimeter radio astronomy. ALMA-specific functionality will be added by the Calibration, Quicklook and Science Data Reduction Pipelines.

4.2.4 Archive Subsystem

The archive subsystem consists of a package in the application and the system layer, respectively. The InfoService package will contain the abstract classes for all data / information services and the services themselves. Hence it will provide for archives, catalogs, repositories, databases, etc. and will be needed by many packages from other subsystems from very early on. The ArchiveResearch package will be responsible for archive queries and data retrieval.

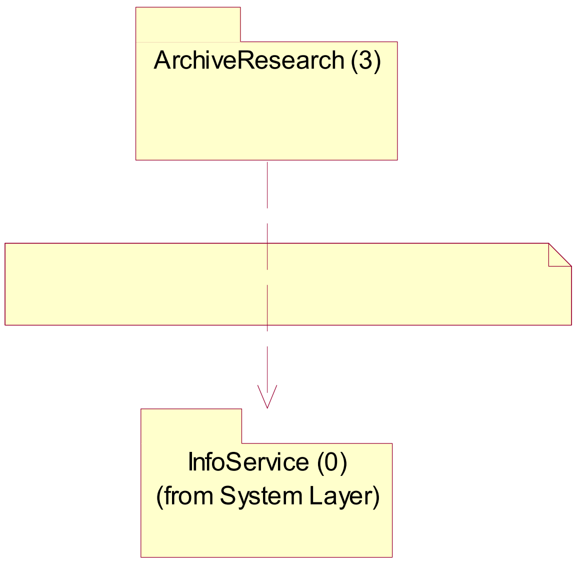

Figure 4-9: Archive Subsystem

4.2.5 Administration Subsystem

At the center of the Administration subsystem are the ProgramTool and AdminTool classes, responsible for the high level administration of observing programs and more general administrative tasks (staff shifts and hardware maintenance schedules, for example), respectively. A Quality Control package was introduced in the subsystem to take care of trend analysis of telescope, array, and receiver properties on the basis of calibration results. The user management package in the domain layer is responsible for managing ALMA user details such as access privileges. This functionality might at some point be completely covered by the ALMA Common Software.

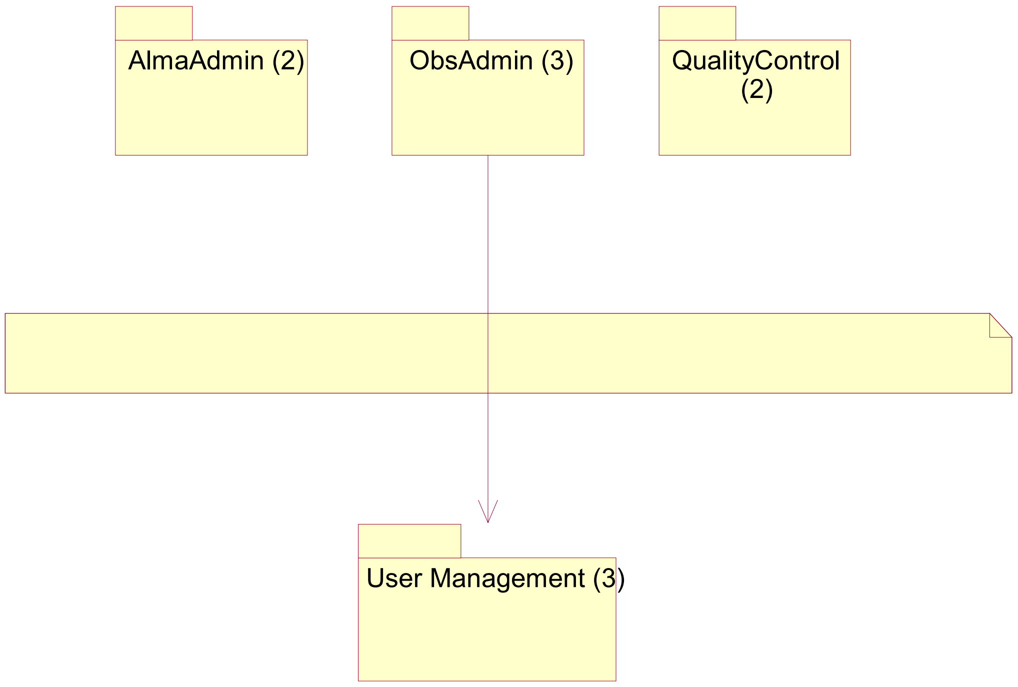

Figure 4-10: Administration Subsystem

4.2.6 ACS Subsystem

The ALMA Common Software (ACS) Subsystem provides the “glue” between all other subsystems. It is based on the CORBA middleware but provides many more features that facilitate subsystem communication, error handling, data collection etc. For a comprehensive summary see the ALMA Common Software Architecture document (ALMA memo 016).

To reflect the existing developments in the ALMA Common Software (ACS) we have added a package called “ACS Core“ which comprises CORBA, the error handling system, etc. The need for the Internet Messaging and Common UI toolkit packages shown in Figure 4-11 was confirmed in the course of our analysis work. We assign them to be part of the ACS subsystem.

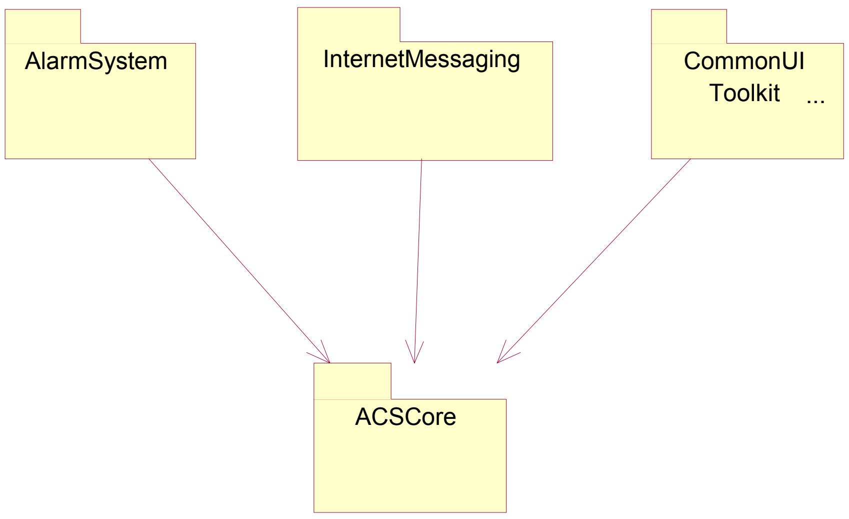

Figure 4-11: ACS Subsystem

4.2.7 Interaction between subsystems

The detailed interactions between subsystems need to be worked out in more detail with the developers of the subsystems involved. In general, coupling of subsystems is minimized by having them interact via the Archive/InfoServices subsystem or via mechanisms provided by ACS as much as possible. Where performance or simplicity make this undesirable, direct interaction is possible. In particular, we expect that raw (and WVR-corrected) data from the correlator subsystem will flow directly to the Calibration and Quicklook pipelines without first being stored and then retrieved from the Archive. While the Archive will be able to accept the full (~10-70 Mbyte/s) data rate in streaming mode, it will be unable to match that speed for retrieval of large quantities of data that it has already stored.

4.3 Mapping of packages to the WBS

Beside the logical structure derived from the required system functionality and from technological constraints, there is also the highly distributed structure of the ALMA development teams and the administratively important Work Breakdown Structure (WBS) which partitions the software development effort among the ALMA partners. Table 1 shows which subsystems and/or packages have been assigned to each WBS element. It also includes the priorities of the individual packages, which will be discussed in the next section.

Table 1: WBS-to-Architecture mapping and priorities. Where a relevant [ISA] reference exists, it is given in the third column.

WBS Item	Subsystem/Package	ISA Ref	Priority
Archiving (2820)	InfoService	2.17	0
	Archive	2.18	0
	ArchiveResearch	2.17	3
Common Software (2740)	CommonUI Toolkit		1
	AlarmSystem		1
	Internet / Messaging Services	2.24	2
	ACS Core		0
Control Software (2760)	Command	2.8	0
	SiteCondition	2.23	0
	Sequencer	2.13	1
	Antenna&Rcv_Cntrl		1
	ResourceMng	2.10	2
Correlator Software (2780)	Correlator		1
Data Reduction User I/F (2890) Offline Data Reduction I/F(2880)	Data Reduction User I/F		2
Executive Software (2750)	AlmaAdmin	2.21	2
	UserManagement		3
	UserSecurity		3
	AlmaExec	2.9	1
Obs. Preparation/Support (2860)	ObsAdmin	2.2	3
	ObsProjectRepositoryController		1
	ObsTool	2.1	1
	Specifications	2.1.1	1
	ObsSimulator[2]	2.1.3	2
	ObsTemplate	2.1.2	2
	CorrelatorSetup	2.1.4	3
	Reviewer	2.3	3
	Policy	2.4	3
	ObsProject	2.9	0
	ObsProjectRepository	2.20	0
	Catalog	2.19	1
Scheduling (2840)	Scheduler	2.12	3
Pipeline – Heuristics (2800)	ScienceData & Q/L Pipelines	2.1.5	1
Pipeline – Infrastructure (2800)	DRP (external)	2.1.5	1
Offline Data Reduction Engines (2880)	DRP (external)	2.1.5	1
	ObsSimulator	2.1.3	2
Tel. Calibration Engines (2900)	CalibrationPipeline	2.14	0
	QualityControl		2

4.4 Priorities of package development

[2] Although the primary responsibility for the simulator has been assigned to the Offline Data Reduction WBS element, a "light" version that, *e.g.*, simulates synthesized beamshapes and estimates required observing time, will be an intrinsic part of the Observing Tool itself.

The developmental stability of the packages and classes must increase from the presentation layer to the domain and system layers because many subsystems depend on functionalities in those layers while higher layers are more independent and can thus be kept more flexible in the development process. The most obvious subsystems with a need for stability are the "Observing Project" and "Observing Project Repository". The domain and system layer subsystems therefore have the highest design and implementation priorities.

We assigned priority levels from 0 to 3 to all packages, 0 being the highest and 3 being the lowest priority (see second column in table 4-1). Note that all packages are necessary for the ALMA system and that these are short-term priorities meant for the design and development in the next 1-2 years, *i.e.* 2002/03. Priority "3" is thus not to be interpreted as "desirable" as in the SSR report but rather "not urgent" while "0" means "crucial".

The priority "0" packages are "InfoService" and "Archive" from the system layer, "Observing Project", "Observing Project Repository" and "Command" from the domain layer and "CalibrationPkg" and "SiteCondition" from the application layer. The system and domain layer packages are important because of subsystem dependencies. The calibration package and the related site condition package were also assigned priority "0" because the need for the development of new algorithms for submillimeter interferometry calibrations makes them critical to the success of ALMA.

Once the level "0" packages have been designed, development for level "1" can begin. Level "1" comprises most of the basic ALMA functionality such as the (basic) observing tool and the array control software. Level "2" deals with systems that are needed later and whose functionality extends the basic level. Level "3" systems add items that will be needed when the interim science phase begins. We assumed implicitly that all user interfaces have priority "3". This only means that the final stable version of the UIs is not an urgent item. There will of course soon be intermediate versions.

4.5 Critical issues

We identified a number of critical issues needing more detailed consideration.

1. To reduce the load onto the system at the ALMA site, observing simulations should be performed offline at remote sites.

2. There is a requirement to support "eavesdropping" on operations (basically from anywhere). This responsibility has not yet been assigned; also uncertain is what kind of access methods and bandwidths to ALMA are needed and how security should be handled.

3. The requested ability to enable a PI to update already-approved Observing Programs / Scheduling Blocks from external nodes can place a large burden on the operations staff, especially if the feedback times must be small (hours rather than days). Such changes might also wreak havoc with a long-term schedule if, for example, new targets are introduced that would change the date in which the observations could be scheduled. Although this is a policy and operational issue, in the absence of definitive guidelines, the software needs to provide a mechanism to make these updates possible, minimizing effects on the operation of the core ALMA system but offering reasonable flexibility to the users.

4. Data reduction algorithms for submm interferometry especially phase calibration correction for long baselines using data from water vapor radiometers are currently still in an experimental stage and development of new algorithms which convert measured water vapor line profiles into phase corrections are crucial right from the beginning of the array operation.

5. Any interactions with the real time system must not lock ALMA operations. Especially the feedback mechanisms of the Calibration Pipeline need to be prototyped early and their performance extrapolated to the full system.

6. The Archive is the central part to decouple the subsystems. Access to the Archive is therefore crucial and it must be able to handle the expected data rates (for science data rates see [SMM]).

7. The requirement for Archival Research adds additional external access (although not to the core system but to remote copies of the Archive) which has to be considered.

8. In general the necessary interprocess communication and network bandwidths have to be estimated. Whether, for example, CORBA middleware provides sufficiently performant facilities for transfer of raw science data needs examination.

4.6 Deployment view

We have made a first cut at allocating ALMA software to individual processors or nodes, and have defined ten general types of nodes or processors to understand deployment and communication issues (see Figure 4-12). They are:

N1) Proposer: Used by the Proposer to prepare observing proposals for ALMA. It will often be external to the secure ALMA network (*e.g.*, because it is on the Proposer's laptop) and exist in several instances.

N2) Reviewer: Used by reviewers during their evaluation of proposals. It may be external to the secure ALMA network and exist in several offline instances.

N3) ALMA Master Control: Controls the ALMA facility and can therefore only exist in one instance. It is naturally in the secure ALMA network and accessed by the ALMA operator.

N4) ALMA Administration: Used by ALMA staff for administration of the general ALMA facility. The node can exist in several instances, but all must be on the secure ALMA network.

N5) Program Administration: Used by staff for performing tasks related to the administration of observing programs.

N6) Data Processing: Performs general pipeline processing of data. It can be both internal and external to ALMA. Any of the various pipelines may, in principle, run on this node; however, the container-component model for the technical architecture (see next chapter) allows us to defer such decisions until run-time—and even to redeploy pipelines to other nodes while the system is running.

N7) Archive: This type of node provides access to the InfoServices. It may exist in several instances but normally on the secure ALMA network.

N8) Observer: Used by interactive observers to specify observations (*i.e.* to specify Scheduling Blocks) which then can be executed immediately. This node is on the secure ALMA network. There may be several instances for the control of several subarrays.

N9) Scientific User: Used by scientific users to perform their work when interacting with ALMA data. It is normally external to ALMA and can exist in multiple instances.

N10) ALMA System: This is a generic node of the ALMA real-time system, *e.g.*, antenna, correlator, real-time computer. It is not anticipated that any of the pipelines will run on this type of node.

Nodes external to ALMA that need remote access to the system raise security issues that need to be considered. Any such remote access must not affect the internal ALMA system performance. Even interactions with the remote DataPipeline and the RemoteArchive, although they don't represent the same potential dangers as interactions with the operations infrastructure itself, still need to be regulated. (Protection of proprietary data and prevention of abuse of resources are two important concerns even at the Regional Support Centers.) In addition, the Proposer and Reviewer nodes may operate offline. This implies the need for synchronization methods between those nodes and the rest of the system.

The activity ("swimlane") diagram shown in Figure 4-13 gives a schematic view of the general order of events in the life of an ALMA program, as well as the parts of the system responsible.

•

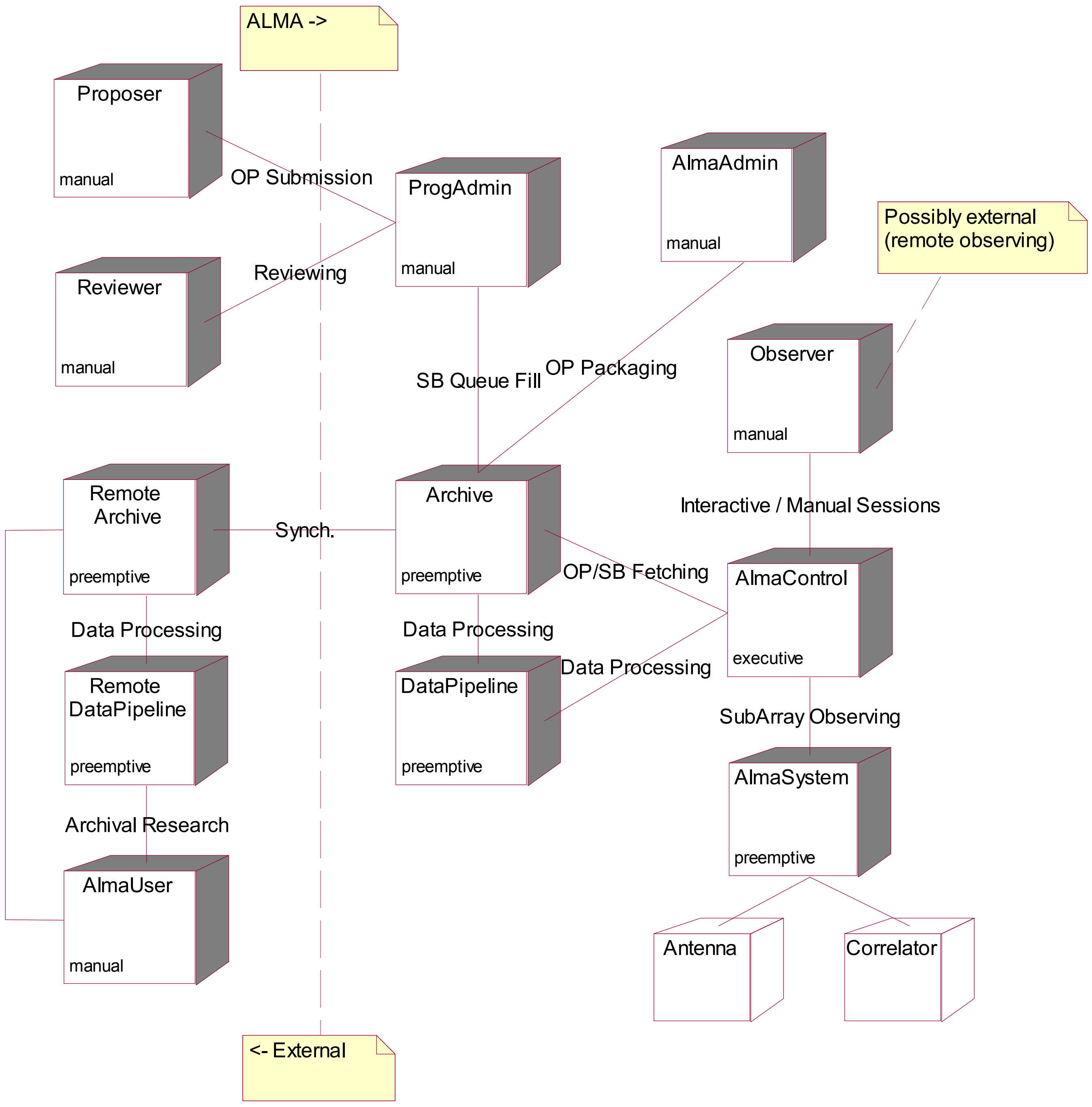

Figure 4-12: Deployment of the ALMA software system on ALMA internal, external and remote nodes.

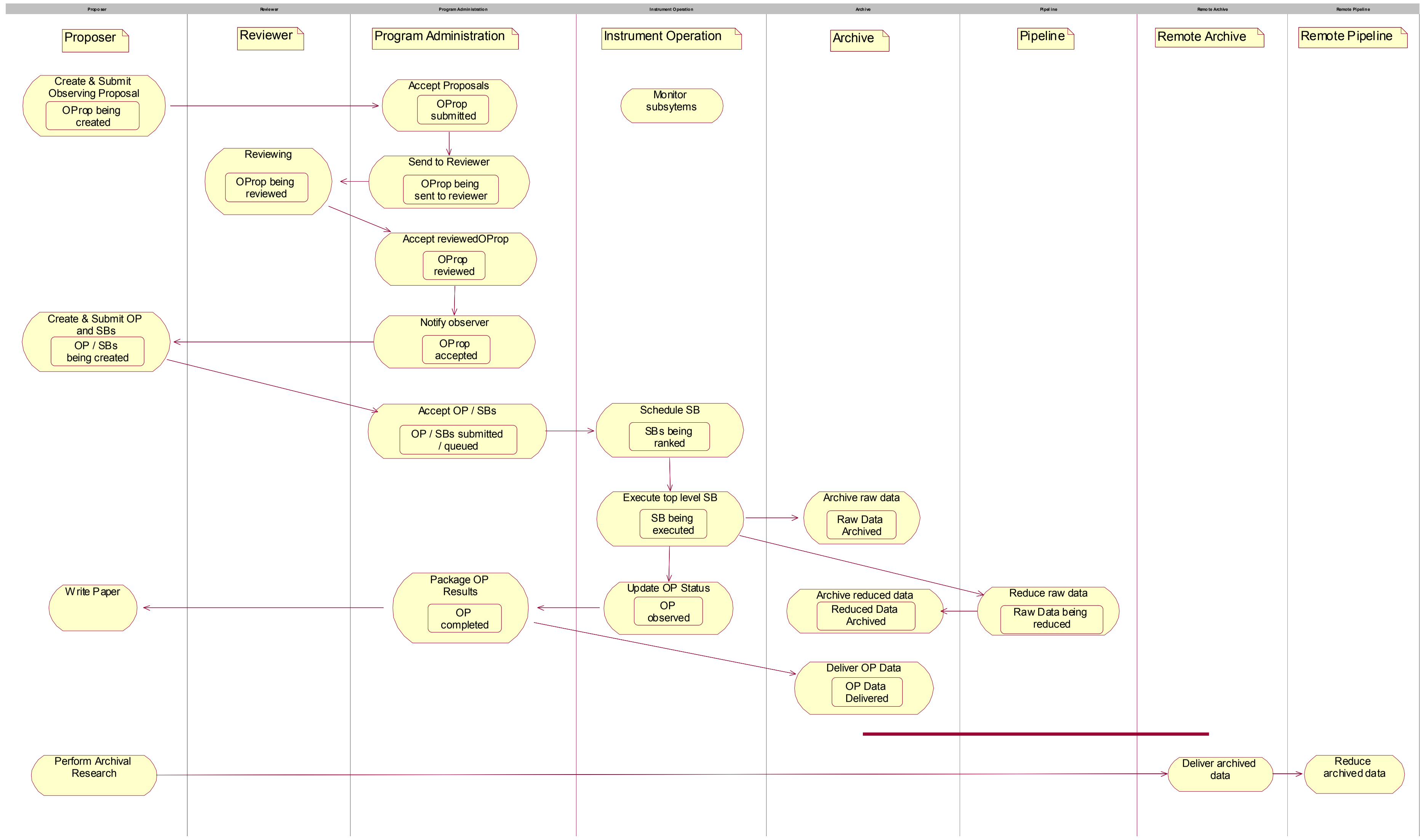
Proposer
Reviewer
Program Administration
Instrument Operation
Archive
Pipeline
Remote Archive
Remote Pipeline
Proposer
Reviewer
Program Administration
Instrument Operation
Archive
Pipeline
Remote Archive
Remote Pipeline
Create & Submit Observing Proposal
OProp being created
Accept Proposals
OProp submitted
Monitor subsytems
Reviewing
OProp being reviewed
Send to Reviewer
OProp being sent to reviewer
Accept reviewedOProp
OProp reviewed
Create & Submit OP and SBs
OP / SBs being created
Notify observer
OProp accepted
Accept OP / SBs
OP / SBs submitted / queued
Schedule SB
SBs being ranked
Execute top level SB
SB being executed
Archive raw data
Raw Data Archived
Write Paper
Package OP Results
OP completed
Update OP Status
OP observed
Archive reduced data
Reduced Data Archived
Reduce raw data
Raw Data being reduced
Deliver OP Data
OP Data Delivered
Perform Archival Research
Deliver archived data
Reduce archived data

Figure 4-13: ALMA swimlane diagram showing the general flow of information through the ALMA software system. Each stage is characterized by a state of most important entity object. Note that the vertical divisions refer to Actors, not the Nodes of the previous diagram.

5 Technical Architecture

The technical architecture for the ALMA software system provides the broad infrastructure necessary to support the functional architecture described in the preceding chapter. As the headings of this and the previous chapter suggest, we want to separate two types of concerns, the domain-specific "functional" concerns, from the "technical" concerns that arise from the computing environment in which the problems of Alma operations, data acquisition and data analysis are to be solved.

This "separation of concerns" is hardly a new concept in software development. Software professionals will recognize in this phrase the same decades-old principle that has guided the development of high-quality, modular code: cohesion of individual modules, and loose coupling between them. It is this principle that teaches us not to combine in a single routine, for example, code to calculate a cosine function with code to perform a numerical integration.

The concerns that we are separating here, however, are somewhat more generic, and our motive for separating them is a precise one: *to allow subsystem developers to concentrate on issues of radio astronomy and interferometry, physics and algorithms, leaving to IT specialists the tasks of providing mechanisms for interfacing, remote access and security.*

5.1 Container-Component Model

The Container-Component model for software organization and development is our primary instrument for achieving this separation of functional from technical concerns.

Voelter (2001) defines a component as a software element that exposes its services through a published interface and explicitly declares its dependencies on other components and services, can be deployed independently and, in addition:

> "is coarse grained: In contrast to a programming language class, a component has a much larger granularity and thus usually more responsibilities. Internally, a component can be made up of classes, or, if no OO language is used, … can be made up of any other suitable constructs. Component based development and OO are not technically related.
>
> "requires a runtime environment: A components cannot exist on its own, it requires a something which provides it with some necessary services." This "something" is called a *container*.
>
> "is remotely accessible, in order to support distributed, component based applications."

The following diagram shows the essential ideas:

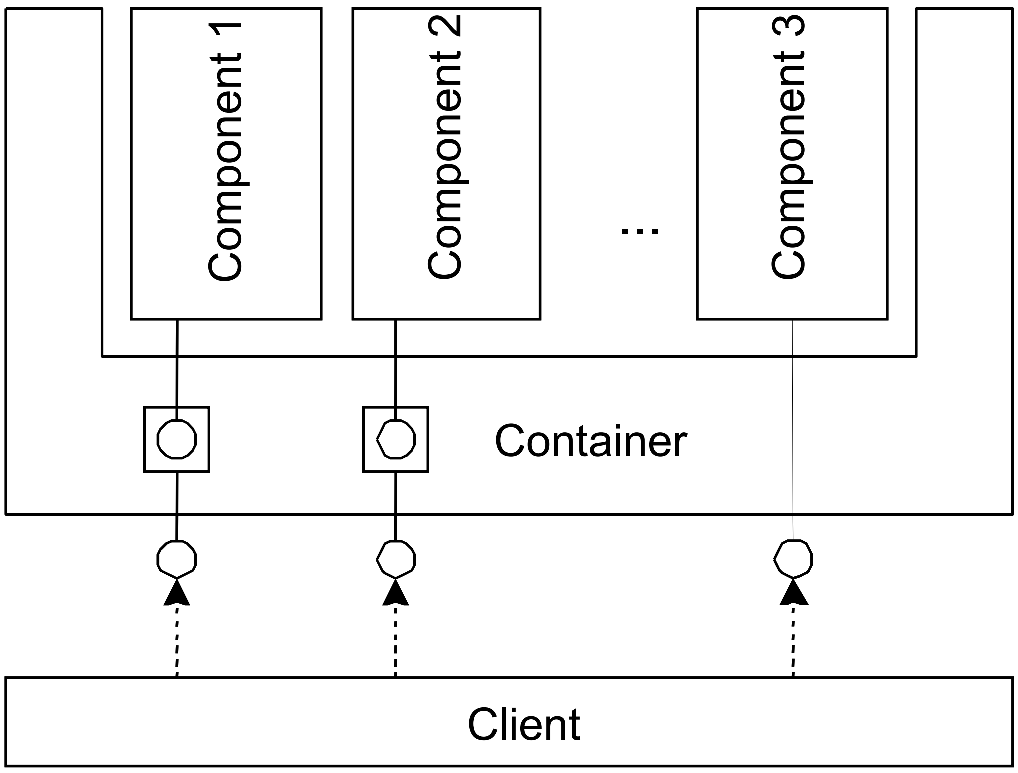

Figure 5-1: A client accesses several service components via an enclosing container.

In this figure, a client (which could be itself a component residing in another container) accesses the services of three components via their published service interfaces. The interfaces published by Components 1 and 2 are shown inside the Container, because in these two cases, access to their services is mediated by the Container itself, which may add additional services such as client authorization verification to those provided by the component. Here, the container is acting as a “tight” container. For Component 3, on the other hand, the container is “porous”; access to the component is direct, although the container will be used to supply the client with the initial reference to Component 3. This might be desirable when performance considerations outweigh the benefits that a tight container could provide in the way of added services.

These two modes, “tight” and “porous”, reflect the expected use of this model in the data flow and control system areas, respectively. The client, however, receives only a reference to its needed component, and does not know which of these two container types it is dealing with.

The division of responsibilities between components and containers enables decisions about where individual components are deployed to be deferred until runtime, at which point configuration information (typically from a file or database, mediated in ALMA by the ACS Manager) 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.

To operate in such an environment, a component publishes two interfaces:

> **Service** interface: the methods and signatures by which the component’s services are available to clients. This interface is written in the CORBA Interface Definition Language (IDL), or derived from it via additional code generation (see section 5.4 for details).

Lifecycle interface: a set of methods that are accessible only by the component's enclosing container. These are, in the order in which they would typically be invoked:

- `setComponentName(String):`
  dynamically assigns an instance name to the component
- `setContainerServices(ContainerServices):`
  callback object through which the component can explicitly use services offered by the container.
- `initialize():`
  tells the component that it's time to configure itself, build up in-memory tables, establish remote connections, etc.
- `execute():`
  similar to initialize(); tells the component to start running methods that are not part of its functional interface. For instance, the Scheduler component would run the thread that maintains the ranked list of SBs. Many components will probably implement only initialize() or execute(), but not both.
- `Cleanup():`
  called after the last functional call has finished; tells the component to release resources so it can be removed safely.
- `aboutToAbort():`
  only called if the container has to forcibly remove the component. Component should make an effort to minimize the damage.

The complete specification can be found in alma.acs.container.ComponentLifecycle.

Note that the implementation of these lifecycle methods is completely at the option of the component, which may decide to provide "null" implementations of any or all of these methods as long as the semantics of the contract between the component and the container are followed.

Other rules for component implementations:

- The constructor of the component implementation class should do close to nothing. Initializations should be carried out in the initialize() and/or execute() methods.
- A component should be implemented thread-safe. This means that any of its methods can be invoked asynchronously in different threads without causing resource conflicts or deadlocks. If thread safety seems really much too complicated to implement, or if very likely only one thread from one client will ever use the component, the container can be configured to synchronize calls to the component to overcome threading issues (at the expense of some performance loss)

Commercial implementations of the Container-Component model are quite popular in industry at present, with Sun's Enterprise Java Beans specification and Microsoft's .NET framework being the prime examples. (A vendor-independent specification, the Corba Component Model (CCM), is under development, but it is not complete, and production implementations do not yet exist.) In any case, all of these are rather comprehensive systems, and require a wholesale commitment from developers to use the languages and tools supplied. For ALMA, which can benefit from an infrastructure that is more lightweight, the Container-Component model will be implemented by ACS, following the structure shown in Figure 5-2. Much of this infrastructure already exists in ACS, so implementing the Container-Component model does not mean that we are starting from a blank slate.

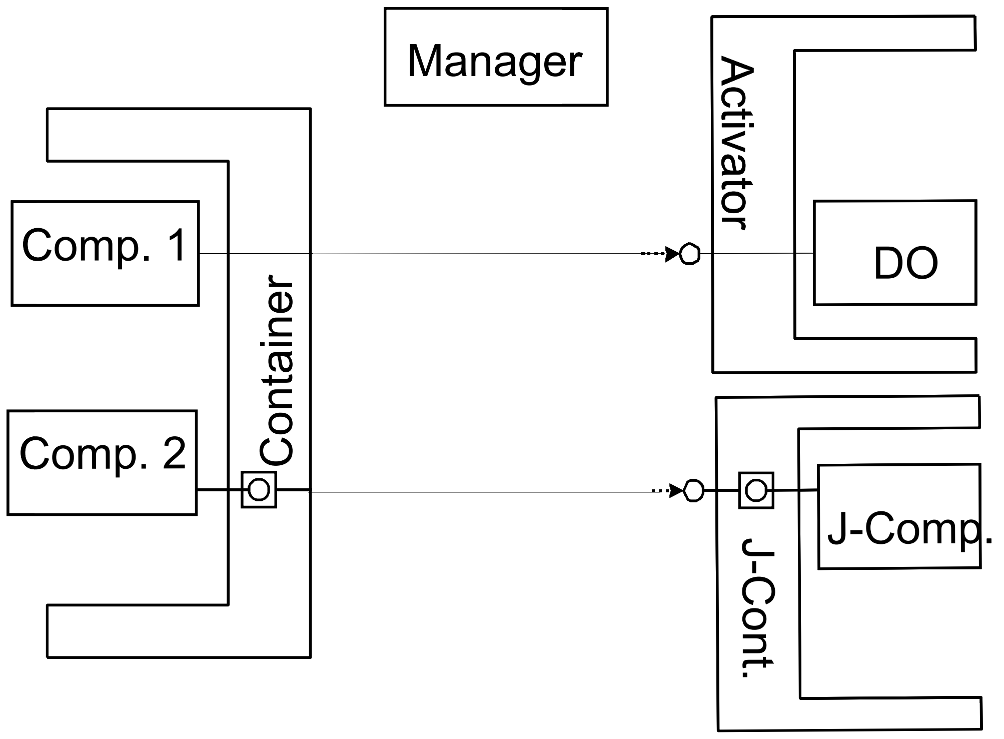

Figure 5-2: The ACS implementation for porous (existing) and tight (planned) Containers. The Manager provides Containers and Activators ("porous" Containers) with configuration information.

In the following sections, we will detail how important technical issues are addressed within the general framework of the Container-Component model: 5.2) definition of entity (or value) classes; 5.5) persistence; 5.6) versioning; and 5.7) choice of programming languages.

5.2 Definition of entity classes

There are many complex data structures that must be defined for the ALMA system to operate smoothly; some will be driven by scientific needs and the requirements of packages outside of our control; examples include:

- complex visibilities;
- images and spectra.

These are typically large (many Mbytes or Gbytes), may have existing formats that are standard throughout the astronomical community (*e.g.*, FITS, Measurement Sets), and are therefore *not* the subject of the discussion that follows.

Other complex data structures drive ALMA operations; they include (see [ISA] for a discussion of this particular class hierarchy):

- Observing Proposals;
- Observing Projects;
- Scheduling Blocks;
- Instrument and operations logs.

We have complete freedom to organize these latter data structures in such a way as to reduce development effort (including the inevitable modifications and revisions that will take place over time) and enhance run-time performance.

In line with the object-oriented philosophy of software development for ALMA, we will speak of *entity classes* and objects. As in the [ISA], we mean here classes responsible primarily for accepting, storing and retrieving persistent data.

In the case of ALMA we often have to deal with structured data, which includes things like configuration parameters and Scheduling Blocks. XML, the eXtensible Markup[1] Language, is a set of rules (one may also think of them as guidelines or conventions) for designing text formats that let one structure one's data. Since its introduction as an open standard, XML has spawned the development of many supplementary specifications and tools that greatly facilitate its use as a data definition language in computer systems development.

We will adopt XML to:

1. define the content and structure of the entity objects that will be passed between subsystems (subject of this section);
2. automatically generate the classes needed to access the data contained in these entity objects, optionally validating every change to a data value; currently validation happens either on explicit request, or automatically on marshalling if desired.
3. serialize these objects for network transmission;
4. facilitate the storage of these objects using the most appropriate and cost-effective database technology at our disposal—Relational and/or Warehouse Database Management Systems today, with perhaps Object-Oriented or XML-based technology in the future. The Archive subsystem already provides the XML database (Xindice) for ALMA subsystem teams to use for their development work; very few (if any) changes will be required to subsystem persistence code in order to use the technology that the Archive eventually adopts.

Although XML documents are written in text format, they are not designed to be reader-friendly. Most editing of XML documents is done via a specialized editor, and the processing itself is hidden from the user, and in most cases, from the developer as well.

[1] In spite of the "Markup" in its name, XML is not primarily for formatting printed documents or Web pages. As its official specification states, XML "can be used to store any kind of structured information, and to enclose or encapsulate information in order to pass it between different computing systems which would otherwise be unable to communicate."

Nevertheless, we provide some “raw” XML examples below to make the basic concepts clear.

XML *schemas* define application-specific rules for the content of an XML document. So, for example, we can define the content of a Scheduling Block via a schema. A part of such a schema will serve to make this clear:

```
<xsd:element name="SchedBlock">
  <xsd:complexType>
      <xsd:complexContent>
          <xsd:extension base="prj:ObsUnitT">
              <xsd:sequence>
                  <xsd:element name="SchedBlockEntity" type="sbl:SchedBlockEntityT"/>
                  <xsd:element name="ObsProjectRef" type="prj:ObsProjectRefT"/>
                  <xsd:element name="SchedBlockControl" type="sbl:SchedBlockControlT"/>
                  <xsd:element name="SchedBlockImaging" type="prj:ImagingProcedureT"/>
                  <xsd:element name="ObsProcedure" type="sbl:ObsProcedureT"/>
                  <xsd:element name="ObsTarget" type="sbl:TargetT" maxOccurs="unbounded"/>
                  <xsd:element name="PhaseCalTarget" type="sbl:TargetT" minOccurs="0"
maxOccurs="unbounded"/>
                  <xsd:element name="PointingCalTarget" type="sbl:TargetT" minOccurs="0"
maxOccurs="unbounded"/>
              </xsd:sequence>
          </xsd:extension>
      </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

This part of the schema states that a Scheduling Block must have the following elements: TargetCoordinates, PhaseCalCoordinates, an ObsScript, Hardware and Calibration Requirements, and references to an ImageScript and something called a TargetPhaseLoop. Of course this is just a simplified example, and an observable SB would have a much richer content.

In any event, both Target and PhaseCal Coordinates are defined to be of the “complex type” SkyCoordinatesT, which is itself defined in detail:

```
  <xsd:complexType name="SkyCoordinatesT">
      <xsd:sequence>
          <xsd:element name="Longitude" type="gen:LongitudeT"/>
          <xsd:element name="Latitude" type="gen:LatitudeT"/>
      </xsd:sequence>
      <xsd:attribute name="system" use="required">
          <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="B1950"/>
                  <xsd:enumeration value="J2000"/>
                  <xsd:enumeration value="galactic"/>
                  <xsd:enumeration value="horizon"/>
              </xsd:restriction>
          </xsd:simpleType>
      </xsd:attribute>
  </xsd:complexType>
  <xsd:complexType name="LatitudeT">
      <xsd:simpleContent>
          <xsd:restriction base="gen:DoubleWithUnitT"> defined in namespace “gen”
              <xsd:minInclusive value="-90"/>
              <xsd:maxInclusive value="90"/>
              <xsd:attribute name="unit" fixed="deg"/>
          </xsd:restriction>
      </xsd:simpleContent>
  </xsd:complexType>
  <--! Similarly for LongitudeT -->
```

While the above schema fragment may look intimidating in its length, the corresponding use of it in an XML document is much simpler. A piece of the XML string or document conforming to this schema could then look like:

```
<TargetCoordinates system="B1950">
    <Longitude unit="deg">24.5</Longitude>
    <Latitude unit="deg">-30.0</Latitude>
</TargetCoordinates>
```

An alternative way of working with xml schemas is to use graphical tools like XMLSpy, where the tree can be manipulated using mouse menus and tables. The SchedBlock with some of its child elements looks like this:

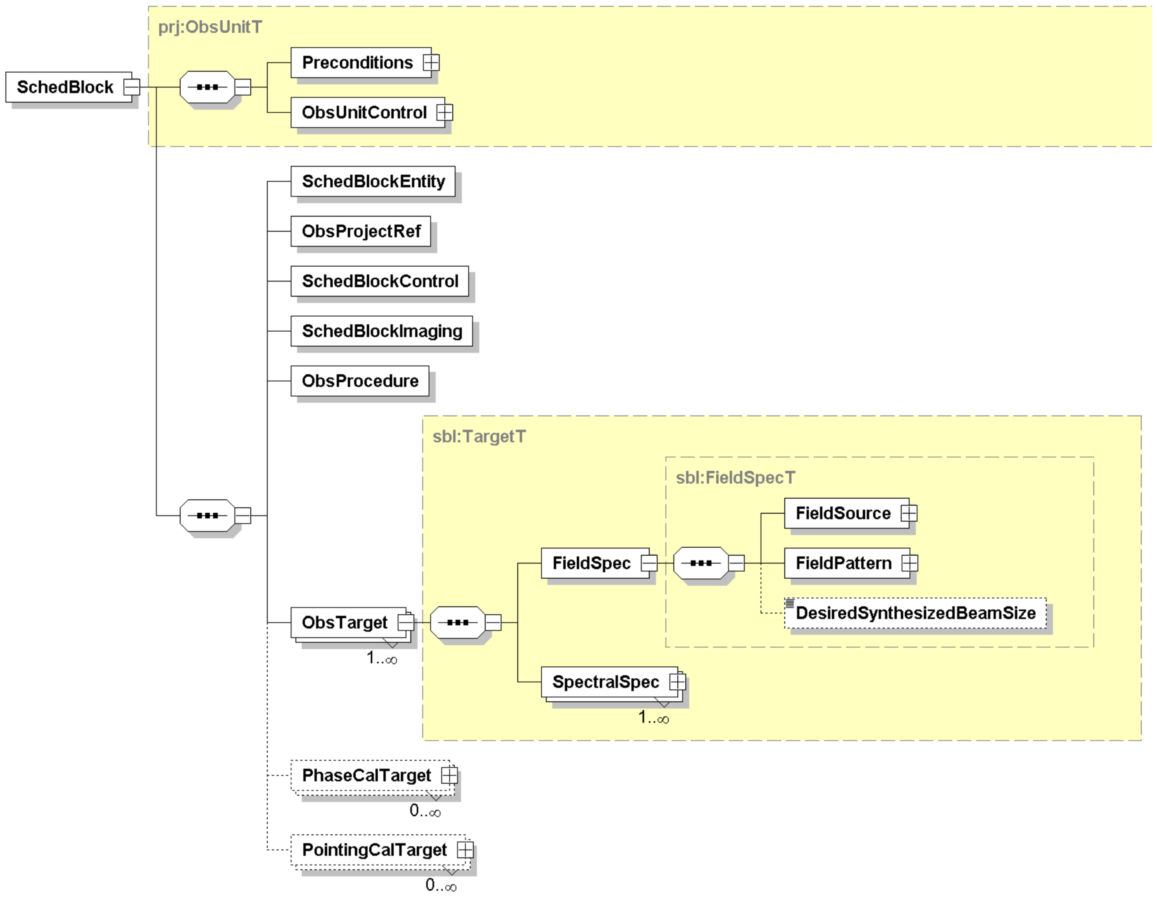

Figure 3: Sample XML schema displayed in graphical form. It shows a hierarchy beginning on the left hand side and branching towards the right. The small squares with "+" signs indicate that more structure is hidden to the right. The rounded rectangles with three bullets in a row indicate that a sequence of elements follows to the right; these elements must be present in the order shown (descending from the topmost element).

XML applications can use the schema to ensure that all data in the corresponding XML document complies in form and content with what is defined there, for instance, that the longitude will be a double precision floating point number between 0 and 360, the latitude will be between –90 and +90, the units will be defined in degrees, and a coordinate system will have been specified that is one of the four allowed (character string) types: “B1950”, “J2000”, “galactic” and “horizon”.

5.3 Generating classes from XML schema

There is no necessity for most subsystem developers to be concerned with, or perhaps even aware of, the XML underpinnings of the entity class definitions (with the possible exception of cases in which entity objects must be passed or returned via remote invocations; see below). This is because we can use tools that automatically generate programming language classes representing these entities straight from the XML schema definition. The subsystem developer can then work directly with the generated class. The process is illustrated in Figure 5-4, where the open-source XML-binding framework Castor is used to transform schemas to Java classes.

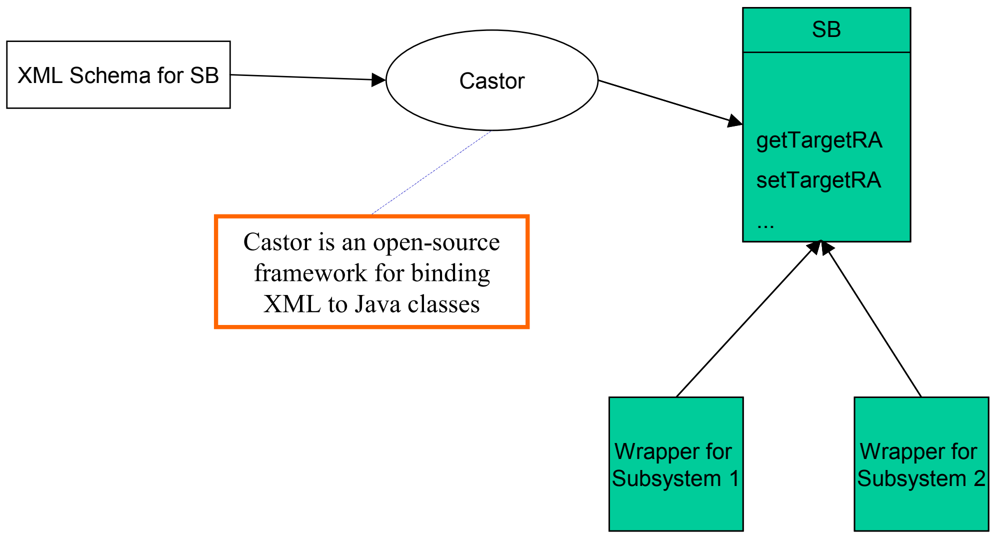

Figure 5-4: Castor generates a Java class from the XML schema for a Scheduling Block. Note the "getter" and "setter" methods supplied as part of the generated class. Subsystem developers can then make "wrapper" subclasses to provide them with any additional specialized functionality that they need.

Not shown in the figure, but produced by the framework as well, is the validation code, which may be optionally enabled to check the validity of all schema-defined entity data. For example, an attempt to set the target latitude to, say, 120 degrees, would cause this validation code to throw an exception. By specifying our constraints in the XML schema and using the framework, we obtain the corresponding validation code for free.

Through development of a prototype, we have already verified feasibility, usability and performance of this technique in Java.

5.4 Interprocess Communication and Serialization

In the usual approach popularized by CORBA, an object in one subsystem that needs the services of an object in remote networked subsystem asks the Object Request Broker for a reference to the remote object, and then invokes methods on that reference, passing additional references in the method's argument list. For those situations in which access to non-trivial portions of an entity object is needed, a pass-by-value mechanism will be preferred, because:

1. Frequent remote method invocations can clog the network, particularly if large amounts of data are passed as parameters or as results.

2. Subsystems should be as independent of each other as possible, so that one can continue to run when another is down. This will be a frequent occurrence during development, and may be common during operations as well.

While CORBA provides a pass-by-value mechanism, it is immature and somewhat shaky, so we will use the same schema and binding framework described in the previous section to allow us to pass entity data by value using an XML serialization technique. These XML strings will then be transported via CORBA.

Figure 5-5 illustrates the principle actors involved. Here, a hypothetical Observing Tool is shown passing a Scheduling Block to a remote Validator component.

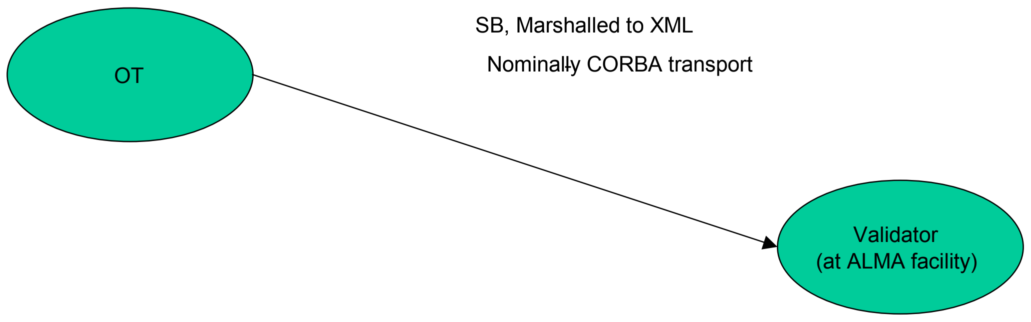

Figure 5-5: The Observing Tool (OT) passes a Scheduling Block across the network to a remote ALMA facility for validation. The SB is serialized to XML before transmission, and de-serialized by the Validator upon receipt.

We use the Interface Definition Language (IDL) to define and publish component interfaces in order to maintain a "Corba-centric" style to the architecture. A component is implemented as a CORBA servant, while the container is implemented as an intercepting layer between the CORBA Object Request Broker (ORB) and the servant. The Container-Component model, built on top of CORBA, hides the details of the implementation of CORBA interfaces from the application developer. The general scheme is shown in the following figure.

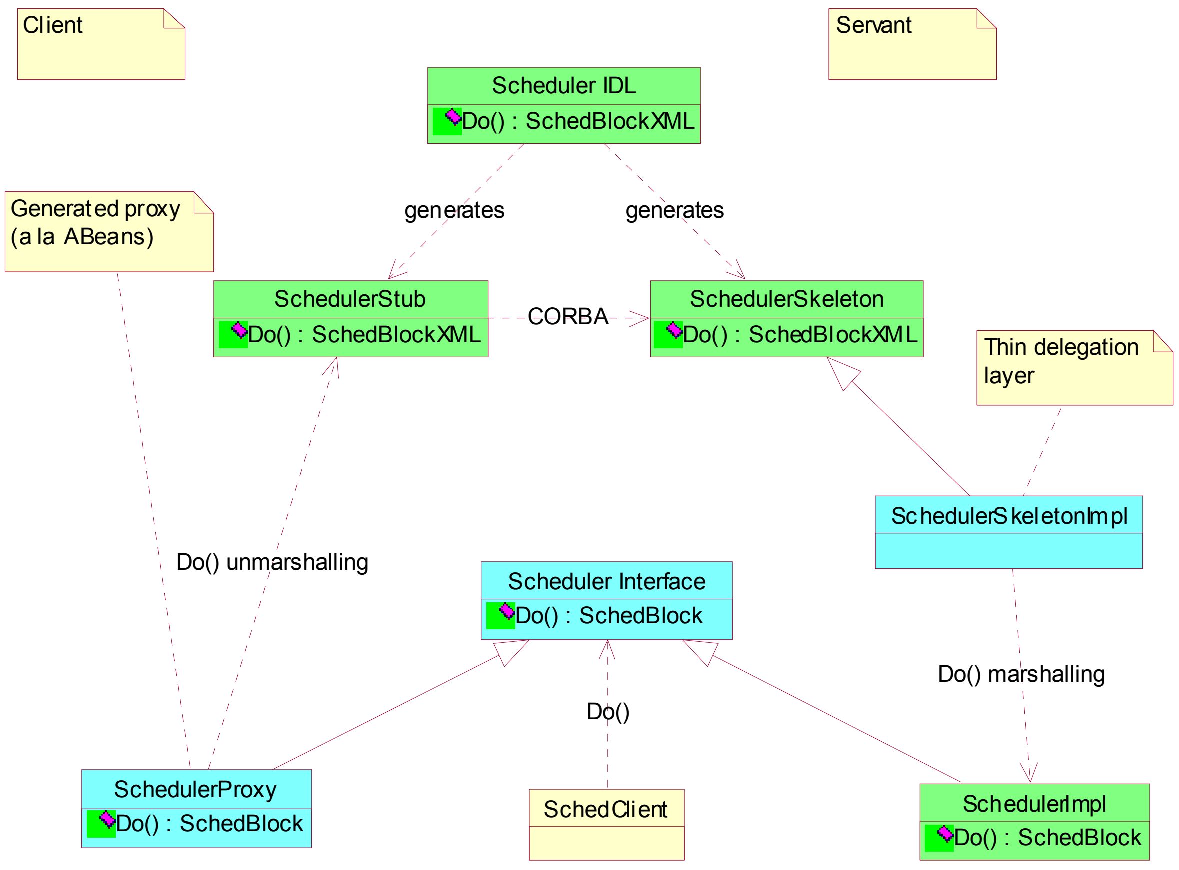

Figure 5-6: Example class diagram for interprocess communication. A client needs to use the Scheduler, which has been implemented as a CORBA servant. Because the Scheduler returns an SB, serialized to XML for transport, both client and server must ensure that the proper marshalling and un-marshalling is done. The classes highlighted in green are automatically generated by an IDL compiler, while those in blue would need to be implemented (either by hand or by a generator).

The IDL interface explicitly uses their XML representation when entity objects appear as method call arguments or return values. This means that clients calling such an interface and servants implementing it must take the responsibility for (un)-marshalling to/from any language dependent representation of the XML data.

In order to make this operation transparent (so that client and server developers do not need to be concerned with the marshalling and unmarshalling of XML), it is possible to provide support classes. In Java, these classes will be generated with the help of the Castor binding framework at compile time, hiding the XML-dependent nature of the IDL definition. The amount of effort to provide similar transparency for other languages is under investigation.

As a further step, we can take advantage of the fact that when the XML representation is hidden by these support classes, the servant implementation class and the generated stub have the same interface. Clients running inside a Container request from the Container references to the Services they want to access. When both client and servant are running inside the same Container, the Container itself can then pass back to the client the actual Programming Language (PL)-native object reference for the servant instead of the CORBA-based stub. This provides an additional optimization; we avoid the XML serialization for the entity data.

This latter feature can be implemented in a later phase without changing already implemented code, because it will be completely hidden inside the Container implementation.

An alternative way of communication is based on the repository. Programs can send around IDs and clients retrieve the data (or the part that they need) from the repository themselves.

5.4.1 Entity Objects in the Instrument Operations Subsystems

The same XML infrastructure can find application in two important areas of the real-time operations of ALMA:

> Commands that pass complex configuration structures to *e.g.*, the correlator, can pass them as XML-schema conformant entity objects, enabling the commanding subsystem to forward this information exactly as it has been prepared by the observer, and the archive to receive directly an exact copy of the configuration as it was used during operations.
>
> The metadata that will accompany the correlator output (see the chapter on Functional Architecture) can be encoded in an XML-schema compliant VOTable (a product of Virtual Observatory development; see http://www.us-vo.org/VOTable/) with a pointer to that portion of the data that will be maintained in binary form for performance and data rate reasons. This offers a more flexible alternative to FITS binary tables and allows the metadata to be stored into the archive *in a searchable format* without format conversion.

In both of the above cases, frameworks that automatically generate C++ classes from the defining XML schema should soon be available.

5.5 Persistence

All needs for persistence in the ALMA system will be handled by the central operational archive subsystem. This includes very different types of data, such as entity objects, science data (raw or processed), and time-stamped logging data. The operational archive will internally use appropriate database or file storage technologies. To the software developers of other subsystems, the storage mechanisms will be shielded by the archive API. For instance, a SchedBlock entity object will be stored in the archive, without the application knowing whether a relational database, an XML database, or any other future technology is at work behind the scenes. With the long development and operations time of ALMA, it is important to avoid or at least encapsulate any such commitment.

The choice of schema-based XML documents as the persistent format of application objects facilitates the storage in a database. Naturally, a real XML database provides easy storage, retrieval and query mechanisms for our XML data. But even if a relational database (RDBMS) were to be used, we benefit from using XML compared with the traditional object-relational mapping model.

When many different views on the same data are required, RDBMSs are at their best because every piece of elementary object data is mapped to attributes in various tables, and can be reassembled into new structures. The downside of this approach is what is know as the “impedance mismatch” between object and relational models. Complicated and time-consuming SQL queries must be executed to get all the relevant record sets that hold the data for the objects we want to retrieve from the database. The same applies to insert and update operations.

While such flexibility might be a requirement for the Science Archive, for the Operational Archive all objects are foreseen to be stored and retrieved using the same data model. We don't need to have one separate field in a relational database table for each attribute in an object. Instead, we can define only those fields that we will need to use for optimized searches, and thus avoid the overhead and performance penalties of supporting arbitrary SQL queries, many of which will never be made.

Therefore we will maintain entity objects as Character Large Objects, essentially text strings of arbitrary length ("CLOB"s; this database field type is defined in Version 3 of SQL). In particular, these CLOBs will be in the XML serialized form that a) conforms to the same schema that was used to define it; and b) is the same as that used for remote communication. These objects will be stored in an RDBMS with carefully selected attributes (redundant data extracted from the object to enable queries based on those attributes). A unique ID will be created and stored for each object. Figure 5-7 shows an example for a simplified Scheduling Block, with Observing Project ID, Target coordinates and maximum allowed Water Vapor column extracted redundantly (these quantities still exist in the XML CLOB) as searchable database fields. In practice, we expect that many more such fields will be needed, but these should not present either storage or performance problems.

This redundancy is visible to and managed by the archive system; update of a CLOB will therefore always result in the corresponding update to any redundant fields maintained in the database table. Maintaining consistency, always a potential source of problems when data is stored redundantly, will therefore be relatively easy to ensure.

This approach offers the advantage of loose coupling between archive and Entity Objects. Changes to the structure of Entity Objects which don't affect the queryable redundant attributes do not require changes to the database and the impact on the DAO's that might result.

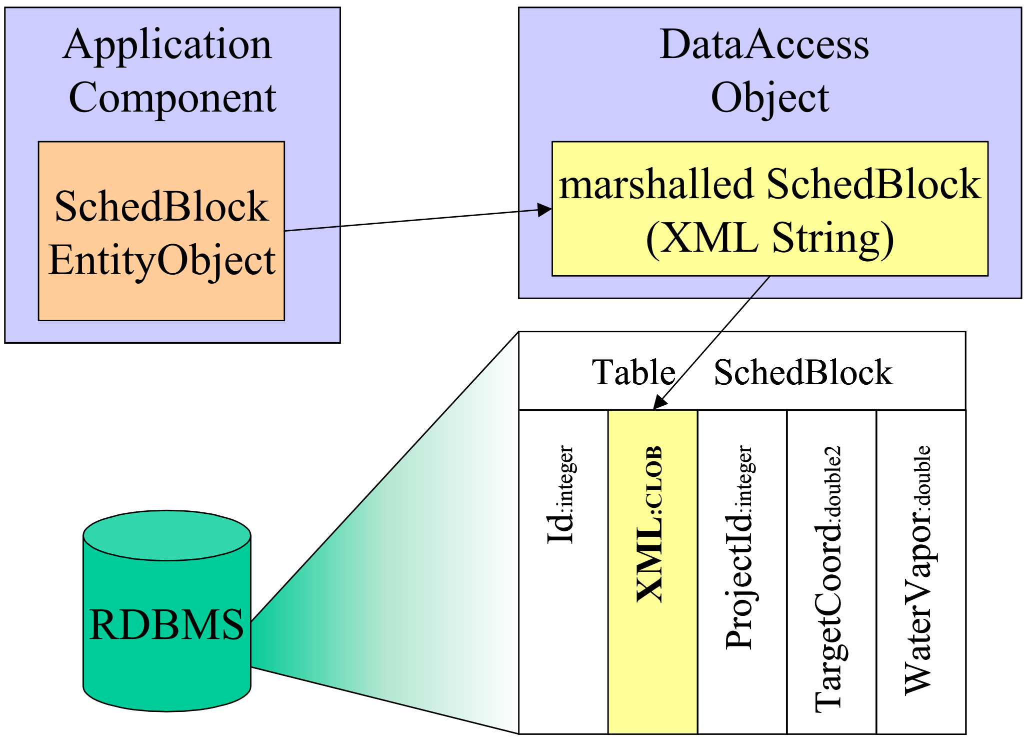

Figure 5-7: Application component accessing a stored Scheduling Block via a Data Access Object (DAO). The SQL machinery is completely hidden from the application component.

Data Access Objects (DAO)

Database access code should be encapsulated inside specialized Data Access Objects. The DAO model offers the required flexibility, without causing the overhead of a full-fledged object-relational mapping layer. By "DAO" we mean a native Programming Language object that has the following responsibilities:

- A DAO is in charge of storing and retrieving one entity object in one type of database.
- It delivers an XML representation of that entity. The same mechanism used for de-serialization can then be used by the client to create the PL-native object.
- For read-only access, the DAO may offer the option to retrieve only part of the object's full XML representation. This is useful for example, when a list of entities is to be displayed to the user, but only one entity is to be selected (and therefore retrieved in its totality) for further processing.
- DAOs are created by a factory[2]. Different implementations of the same DAO functionality allow us to use a different database (or even a file system) without changing the application code. This is especially useful during development (when individual developers might use a lightweight DBMS such as mySQL for testing, whereas the system as a whole will probably rely on an industrial-strength DBMS such as DB2) and for remote deployment (for example, on a notebook).
- Developer groups can be responsible for their own database access code (in particular, subsystem developers will be responsible for writing their own DAOs, in

[2] Standard creational pattern from the book "Design Patterns" by Gamma et.al. [GOF]

consultation with the Archive subsystem team), without depending much on other groups.

Queries

We want to keep application objects as free of SQL or XPath/XQuery and query definitions as possible. Therefore, query definitions should be encapsulated by DAOs

To execute a query, an object calls a query method of a DAO. Letting the object pass binding parameters can provide some flexibility. A query to access all SBs with certain parameters might look like:

SB_DAO.getSBs(status, LST, arrayConfig);

The binding parameters here would be the status (perhaps "ready"), the LST range, and the current antenna array configuration, to look up all Scheduling Blocks that are ready to run with a given LST range and a given array configuration. All query code would be hidden by the DAO.

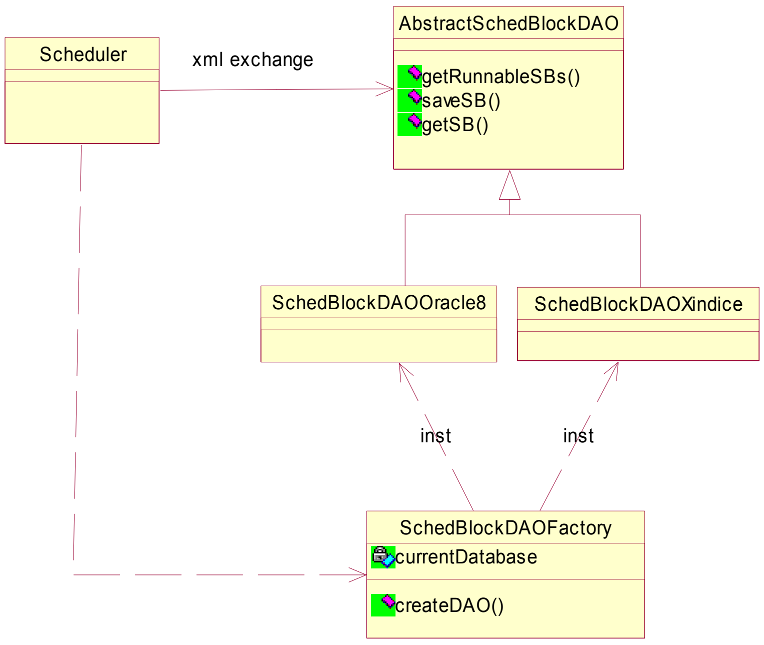

Figure 5-8: Example class diagram for a Data Access Object. At run time, the concrete SchedBlockDAO will be created, depending on what database technology and database are being used. The code of client (the Scheduler in this example) does not change, no matter how radically the underlying database technology may be altered.

5.6 Versioning

The usual way of dealing with changes to software is through a configuration control system and the re-integration of the system when module interfaces change. Similarly, changes to the definitions of entity objects (*i.e.,* to their data formats and contents) usually require modification of the routines that access them, and the transformation of existing objects in, for example, an archive. To avoid massive updates of existing data repositories, we can use the alternative approach of requiring that the application software itself be able to recognize older versions of interacting modules or accessed entity objects and be able to function with the older (reduced) functionality. A "lazy update" mechanism could be used to transform an older version of an entity object to the current version when it is first accessed after a definition change. Such a requirement places a non-trivial burden on the application programmer, but makes updating parts of the system easier and integration of those parts more forgiving. The addition of an attribute to the observing project data structure, for example, could be done without the simultaneous updating of all existing projects in the ALMA archive.

We believe that such backward compatibility might prove desirable for the project, but don't see a clear case for mandating it at this time. To preserve the possibility that such flexibility could be added in the future, we will, however, mandate the inclusion of a version number in each object (particularly entity objects) so that it can be queried at runtime. In the simplest case, a module encountering a version mismatch could exit gracefully.

5.7 Languages

The ALMA software development effort is distributed across nearly a dozen institutes in two continents. In such an environment, unifying influences are certainly welcome. We believe that one such unifying influence would be reliance, insofar as possible, on a single programming language. The ALMA Computing group has already recognized the value of coding standards for each language: it makes code easier to understand, review and maintain by those who did not write it originally. Similarly, the use of a single language would make code sharing, reuse and evaluation easier across the project and reduce the burden of standards and tool support required of the Software Engineering group.

Use of a single programming language is not possible on this project for both historical and technical reasons. ALMA will depend on a good deal of legacy software, written in various compiled and scripting languages. Technically speaking, the hard real-time requirements of the project can be met only with a compiled language that is both predictable and performant at run-time. At the same time, the [SSRD] requires a highly-flexible scripting capability for the development of new observing procedures, as well as user-friendly Graphical User Interfaces for the neophyte ALMA user.

Nevertheless, we see a strong case for encouraging the project to prefer a single programming language for new software development, and to restrict the use of scripting languages to prototyping activities and those that require the kind of flexibility that the [SSRD] calls for. In particular, we recommend the use of Java, outside of the hard real-time and scripting areas. Its strong static typing and run-time error checking makes it a very solid basis for developing code that must be robust and reliable. It is one of the less arcane of modern object-oriented programming languages, has exceptionally good library support, has respectable performance and has the non-negligible advantage that many experienced Java programmers are available.

For the hardware control system, the choice of C++ has already been made. We propose to support the Container-Component model described earlier for both Java and C++, and to support *client* access to Java and C++ components from Python, should this be chosen as ALMA's preferred scripting language. We recommend that Jython (formerly Jpython) be carefully considered as a replacement for Python because: 1) it offers seamless integration with Java and the opportunity to exploit the full range of libraries that are supplied with Java; and 2) it would make it straightforward to migrate observing procedures from scripting to compiled (Java) code once they are mature and fully tested.

5.8 Bulk data throughput

Raw data (coming out from the correlator), as well as reduced data (calibration data or scientific results), can be of the order of gigabytes for a contiguous set.

For performance reasons, bulk data can't be treated like "normal-sized" data, *i.e.,* as typed members in a class or part of an XML file. It must be transported very efficiently. Whether or not this can be done using CORBA techniques (possibly the CORBA Audio Video Streaming service) needs further investigation. In any case, the handling of bulk data should be as object-oriented as possible from the application developer's point of view.

We need to provide a mechanism that

Moves bulk data from where it's created to safe storage.

Never needs to have the entire bulk data chunk in memory at a time (therefore, some Reader/Writer-streaming technique seems useful). In fact, we propose to only provide sequential access to bulk data. If a process needs random access, *e.g.*, for deconvolution of images, it must first copy the bulk data to its local file system.

Allows bulk data to be referenced from objects as member data. The object keeps a link to the bulk data which resides outside the object on a specialized node, called the Fast Data Store (FDS). This allows us to move objects by-value without having to move their associated bulk data.

Integrates the FDS with the operational archive:

The FDS will be critical to instrument operations, while we expect that any data that needs to be accessed from the archive (*e.g.,* for definitive image processing of data sets acquired many days, weeks or months apart) can be accessed in a more normal way. *(We would appreciate comments by the scientific reviewers in particular about whether this assumption is correct.)*[3]

Saving data to the archive: if an object is ingested by the archive, the associated non-transient bulk data must be copied automatically from the FDS to the archive as well. This transfer could actually be anticipated by the archive, which could pull files from the FDS at low process priority, even before the application objects get archived.

Bulk data should be accessed in a type-safe way. All components should share a set of wrapper classes that translate the byte structure to something meaningful like a map of visibilities.

The following items need to be discussed further:

[3] If this assumption is not correct, we could have bulk data be transparently uploaded from the archive to the FDS when an object referencing that bulk data is read from the archive. If the bulk data is accessed after being uploaded completely, then access to that data will be rather fast; if it's accessed during the upload, it should still be piped through the FDS at the rate at which the data comes from the archive.

To allow the generic ACS/FDS/archive software to handle objects with bulk data, each such object must implement an interface that declares the persistent or transient bulk data "members" and gives access to these data.

Do we need to transport bulk data without creating an object beforehand that keeps a reference to the data? An argument could be that raw data must be recorded even if the application software fails. We suggest not doing this, as it would require an additional, non-OO classification/retrieval schema. Software failures should rather be addressed using redundant services and failover strategies.

We assume that bulk data will be immutable, either because 1) it represents raw data, which by definition needs to be preserved as is; 2) it represents a transformation of the raw data, *e.g.*, when visibilities are phase-corrected, in which case a new file/object will be created; or 3) it will be passed as a whole to/from an external system such as AIPS++, which will then be responsible for using it in whatever way is necessary.

5.9 Security

We want to prevent:

Destructive attacks from the internet;

Theft of intellectual property, inside or outside of ALMA;

ALMA users trying to obtain privileges that they have not been granted; here we assume only limited criminal energy (e.g., we don't expect users to deploy their own manipulated client software at the ALMA site and connect to the servers there).

The Container-Component model enables us to set security policies for each Component at run-time, and delegate the task of enforcing these policies to the Container. Within the ALMA software system, we will implement something less demanding than the CORBA security service. There will be a tradeoff between keeping security issues completely outside of the components (thus making component development easier), and avoiding lengthy deployment procedures. As a rule of thumb, IDL interfaces should be designed so that all methods inside are likely to have similar access restrictions.

5.10 Technical Prototype

Earlier in 2002 we developed a first prototype that provided a container-component model and transparent un-/marshalling of XML entity objects. It was based on the assumption that almost all of ALMA software could be written in JAVA, and that JAVA-RMI communication would therefore allow us to elegantly handle the XML communication.

After discussions with several ALMA developers this idea was abandoned in favor of the CORBA-centric model described earlier in this document. It relies on CORBA for remote transport and allows the use of different programming languages, although the number of supported languages must be kept low since every language requires the re-implementation of the framework.

Currently a second prototype is being developed, in accordance with the CORBA-centric model. It will offer the container capabilities of the ACS activator to Java components. Transparent un-/marshalling of XML binding classes will be supported. Deployment information will be stored in the ACS Configuration Database and will be used by the container and the ACS Manager. This approach ensures a seamless integration of the ideas of this document with the existing ACS and its future development; in other words an *evolution*, not a *revolution*, in ACS.

Appendix A. Scheduling Block Design: A first sketch

Since the Scheduling Block is a key, perhaps *the* key, class that will govern ALMA operations, we begin here to sketch the outline of its design. Completion of this design will require collaboration from members of all subsystem teams that depend, in one way or another, on information contained in this class and its constituent pieces.

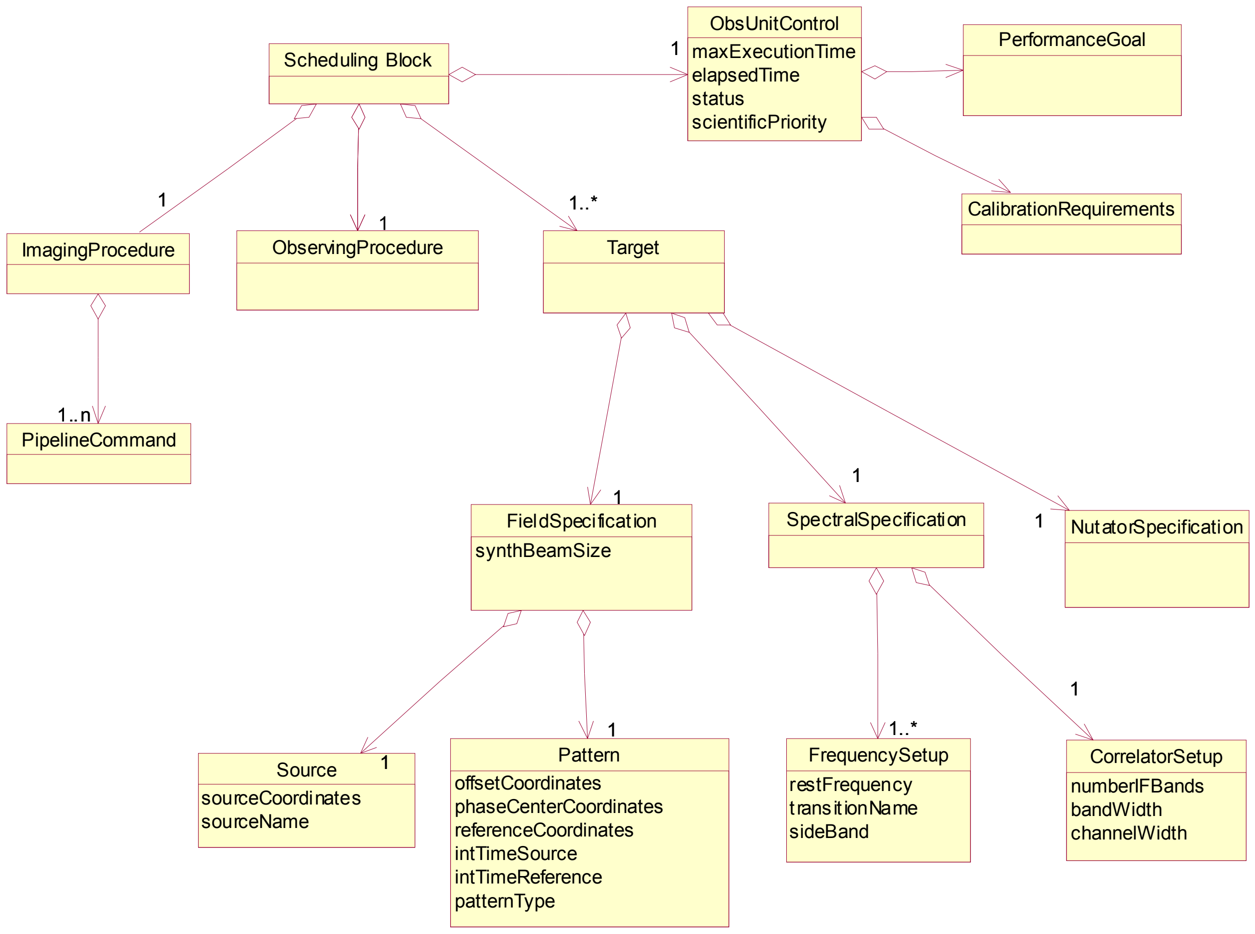

Figure 5-9: Class diagram for a Scheduling Block and its dependent parts. This is a revised version of the diagram in [ISA].

Figure 5-9 shows the class diagram for a Scheduling Block and the classes that it contains or depends upon. Most quantities are specified by the user, normally via the Observing Tool. The ObsUnitControl block will contain, as a rule, parameters that are controlled by the system (status, elapsedTime) or set by reviewers (scientificPriority) and that should not be modifiable by users.

It is clear from the diagram that some of these classes have been developed in more detail than others, and it should therefore also be clear where more work needs to be done. For example:

1. "ObservingProcedure" might be a Python script or a more abstract description of the observing procedure to be followed, perhaps as produced by the Observing Tool and later converted to a script for execution.

2. Patterns need to be defined more generally, perhaps by a class hierarchy.

3. Calibration and Hardware Configuration requirements need to be defined in a way that enables the system to recognize what incremental steps are needed to bring the system to the desired configuration and calibration state in the most efficient way.

We provide below a first sketch of what the XML representation of an ALMA Scheduling Block might look like. (The defining schema for this file will be found in the Interface Control Document for the Observing Preparation subsystem.) The file applies the ideas discussed in the preceding chapter to provide the parameters for an observation of the CO 3-2 transition at ~ 345 GHz in the Large Magellanic cloud. Many things have been left out, for example, an explicit calibrator specification, but this would mimic the layout of the target specification, and so would add bulk but little additional clarity to the example.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XML file generated by XMLSPY v5 rel. 2 U (http://www.xmlspy.com) therefore some values are left as
"String" or "0" which is how XMLSPY constructed them -->
<sbl:SchedBlock xmlns:sbl="Alma/SchedBlock" xmlns:ent="Alma/CommonEntity"
xmlns:gen="Alma/GeneralIncludes" xmlns:prj="Alma/ObsProject"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="Alma/SchedBlock
H:\ALMA\Architecture\SAD\XML\schemas\SchedBlock.xsd">
    <prj:Preconditions>
        <prj:WeatherConstraints>
            <prj:Opacity>TBD</prj:Opacity>
            <prj:Seeing>TBD</prj:Seeing>
            <prj:PhaseStability>TBD</prj:PhaseStability>
        </prj:WeatherConstraints>
        <prj:Polarization>TBD</prj:Polarization>
        <prj:BaselineCal>TBD</prj:BaselineCal>
    </prj:Preconditions>
    <prj:ObsUnitControl scientificPriority="0" runWhenReady="1" schedStatus="waiting">
        <prj:MaximumTime unit="sec">1800.</prj:MaximumTime>
        <prj:ElapsedTime unit="sec">0.0</prj:ElapsedTime>
        <prj:PerformanceGoal>e.g., desired sensitivity level</prj:PerformanceGoal>
        <prj:CalibrationRequirements>
            <prj:PointingAccuracy>0.8 arcsec</prj:PointingAccuracy>
            <prj:Bandpass>TBD</prj:Bandpass>
        </prj:CalibrationRequirements>
    </prj:ObsUnitControl>
    <sbl:SchedBlockEntity entityId="2345" entityIdEncrypted="String" entityTypeName="SchedBlock"
entityVersion="0" isReadOnly="false" lifecycleState="String"/>
    <sbl:ObsProjectRef entityId="1432" entityTypeName="ObsProject" entityVersion='"latest"'/>
    <sbl:SchedBlockControl repeatCount="24"/>
    <sbl:SchedBlockImaging imageScript="String"/>
    <sbl:ObsProcedure obsProcScript="String -- probably a call to a Python procedure"/>
    <sbl:ObsTarget>
        <sbl:FieldSpec>
            <sbl:FieldSource sourceName="NGC19234">
                <sbl:SourceCoordinates system="B1950">
                    <sbl:Longitude unit="deg">71.0</sbl:Longitude>
                    <sbl:Latitude unit="deg">-34.5</sbl:Latitude>
                </sbl:SourceCoordinates>
                <sbl:SourceVelocity>
                    <sbl:CenterVelocity unit="km/s">250.</sbl:CenterVelocity>
                    <sbl:referenceSystem>lsr</sbl:referenceSystem>
                </sbl:SourceVelocity>
                <sbl:SourceProperty>
                    <sbl:SourceFrequency unit="GHz">345.0</sbl:SourceFrequency>
                    <sbl:SourceFlux unit="Jy">0.0015</sbl:SourceFlux>
                    <sbl:SourceDiameter unit="arcsec">0.005</sbl:SourceDiameter>
                </sbl:SourceProperty>
            </sbl:FieldSource>
            <sbl:FieldPattern type="rectangle">
                <sbl:OffsetCoordinates system="B1950">
                    <sbl:Longitude unit="deg">0.002</sbl:Longitude>
                    <sbl:Latitude unit="deg">0.002</sbl:Latitude>
                </sbl:OffsetCoordinates>
                <sbl:ReferenceCoordinates system="B1950">
                    <sbl:Longitude unit="deg">70.999</sbl:Longitude>
                    <sbl:Latitude unit="deg">-34.5</sbl:Latitude>
                </sbl:ReferenceCoordinates>
                <sbl:PhaseCenterCoordinates system="B1950">
                    <sbl:Longitude unit="deg">71.0</sbl:Longitude>
                    <sbl:Latitude unit="deg">-34.5</sbl:Latitude>
                </sbl:PhaseCenterCoordinates>
                <!-- More specs go here -->
            </sbl:FieldPattern>
            <sbl:DesiredSynthesizedBeamSize unit="arcsec">0.05</sbl:DesiredSynthesizedBeamSize>
        </sbl:FieldSpec>
        <sbl:SpectralSpec>
            <sbl:FrequencySetup transitionName="CO 3-2" sideBand="lsb">
                <sbl:RestFrequency unit="GHz">...etc.</sbl:RestFrequency>
            </sbl:FrequencySetup>
            <sbl:CorrelatorSetup>
                <sbl:BandWidth unit="GHz">8.0</sbl:BandWidth>
                <sbl:ChannelWidth unit="MHz">30.0</sbl:ChannelWidth>
                <sbl:NumberIFBands>4</sbl:NumberIFBands>
            </sbl:CorrelatorSetup>
        </sbl:SpectralSpec>
    </sbl:ObsTarget>
    <sbl:PhaseCalTarget>
        <!-- Similar specs to above -->
    </sbl:PhaseCalTarget>
    <sbl:PointingCalTarget>
        <!-- Similar specs to above -->
    </sbl:PointingCalTarget>
</sbl:SchedBlock>
```