

# Proposal and Observing Preparation for ALMA

Alan Bridger<sup>a</sup>, David A. Clarke<sup>a</sup>, Alberto M. Chavan<sup>b</sup>, Joseph Schwarz<sup>b</sup>, Marcus Schilling<sup>b</sup>,  
Leonardo Testi<sup>c</sup> and Heiko Sommer<sup>b</sup>

<sup>a</sup>UK Astronomy Technology Centre, Blackford Hill, Edinburgh, EH9 3HJ, United Kingdom;

<sup>b</sup>European Southern Observatory, Karl-Schwarzschild-Str. 2, D-85748 Garching, Germany;

<sup>c</sup>Osservatorio Astrofisico di Arcetri, Largo E. Fermi 5, 50125 Firenze, Italy

## ABSTRACT

A number of tools exist to aid in the preparation of proposals and observations for large ground and space-based observatories (VLT, Gemini, HST being examples). These tools have transformed the way in which astronomers use large telescopes. The ALMA telescope has a strong need for such a tool, but its scientific and technical requirements, and the nature of the telescope, provide some novel challenges. In addition to the common Phase I (Proposal) and Phase II (Observing) preparation the tool must support the needs of the novice alongside the needs of those who are expert in millimetre/sub-millimetre aperture synthesis astronomy. We must also provide support for the reviewing process, and must interface with and use the technical architecture underpinning the design of the ALMA Software System. In this paper we describe our approach to meeting these challenges.

**Keywords:** Observing Tool, ALMA, Proposal Preparation, Observing Preparation

## 1. INTRODUCTION

The Atacama Large Millimetre Array will fill the plains of Chajnantor in the Atacama Desert in Chile in the latter half of this decade. It is being built by an international collaboration between Europe and North America, with Japan in the process of joining the project. It will be an aperture synthesis radio telescope that will operate at millimetre and submillimetre wavelengths.

It is now commonplace that Observers only rarely travel to the telescopes that they use. Not only do they prepare and submit their observing proposals from their desktops, but increasingly also their actual observing programs. Tools developed by the observatories interpret the observers' program descriptions and create packets of information that are later used to drive the observing system.

The ALMA Observatory will be no exception: it will have an Observing System similar to that used by most of the World's major ground-based Observatories. The preparation of proposals and observing will be supported by software, the observatory itself will operate mostly in a queue-scheduled mode, and some level of processing will be performed automatically on the data at the observatory, the final data products arriving in an archive. The overall ALMA Software System is outlined in Schwarz et al,<sup>1</sup> but some key points are relevant to this paper. Firstly, ALMA Observing is described by *Scheduling Blocks*. A Scheduling Block (SB) is the atomic unit of observing with ALMA. It will have a typical length of around 30 minutes, including calibration observations, and will in most cases provide data only for a part of the intended observation.

Secondly, most information within the ALMA system is held within XML documents. Thus the data model describing the Observing Project, the proposal, the Phase II information, SBs, etc., is defined in terms of XML Schema definitions. These schemas are generated from UML class diagram descriptions (see Sommer et al.<sup>2</sup>).

The two main functions of the ALMA Observation Preparation subsystem are to allow an ALMA investigator to prepare and submit Observing Proposals (commonly known as Phase I) and then to prepare and submit an Observing Program (Phase II), which holds the information required to acquire the intended data with the Observatory. The Observing Proposal and Observing Program are held together in an Observing Project, which is the unit of submission to the Observing Project Repository. In addition to these main functions there is a need to support the reviewing process that sits between Phases I and II, and it is likely that the same software will

---

Send correspondence to A.B., E-mail: ab@roe.ac.uk

provide support for these functions. It is intended that all of these functions will be supported by the provision of an ALMA *Observing Tool* (OT) application.

It is intended that ALMA will begin science Observing with a few commissioned antennas in late 2007, so the Observing Preparation tools must be ready in advance of that to support the first users. However, there is a much closer (late 2004) short term goal for the system is to support observing on the ALMA Antenna Test Facility (ATF), situated at the VLA site near Socorro, New Mexico. This is now driving our development priorities.

In this paper we outline our overall design solution and describe in a little more detail some fundamental infrastructure we have implemented to support later development.

## 2. BASIC REQUIREMENTS AND INTERPRETATION

Creating observing preparation tools for the ALMA telescope presents some novel challenges. A major goal for the ALMA telescope is to widen the field - open it to the many astronomers who are not experts in radio interferometry. Thus an observing preparation tool for ALMA must provide an intuitive science-based interface. However, it is also a requirement that the tool supports “expert” users, those who understand the nuances of aperture synthesis, particularly in the submillimetre. These experts will at least exist in the guise of observatory staff, who will use the tool to create standard observing modes. But this provision will also allow non-staff expert users to maximise the scientific output of their programs.

So, the ALMA OT must support the needs of both “novices” and “experts”. Our interpretation of this is to offer two views on the Observing System, *Science view*, in which users will be able to define their observing in terms of their science goals and astronomical concepts, and a *System view*, which will expose the user to Scheduling Blocks (SBs), the key, indivisible, blocks of observing on ALMA (see Schwarz et al.<sup>1</sup> for a more detailed explanation of SBs). To move from the science view to the system definition we offer a *Program Generator*, which will typically create many SBs from a simple target mapping setup.

Furthermore, it is possible that all information for observing may be provided at Phase I, and even that SBs may be prepared at this stage. To support this we have defined a data model that allows the full observing definitions - both science view and system view - to be described at any phase of the preparation process, though only the Phase II version will be the input to the ALMA Observing System.

Finally the system must also support proposal and observing validation, proposal reviewing and observation modelling (at least exposure time calculation).

In order to support these main functions, there is also a need for a server side component, the Observing Project Repository, which will act as the OT’s agent in submitting and retrieving Projects to and from the ALMA Archive database. Further server-side components provide access to the database of observatory characteristics, and also provide support for SB creation by other subsystems at the site (for manual mode, and the creation of special SBs for long term scheduling).

As the tools must be available to all potential users, regardless of their computing platform, we need to choose a suitable supporting technology. We chose to develop a Java application as the business logic is fairly complex, and the desired GUI interface very dynamic, both of which are not well suited to web services. This also means that users do not require a network connection until they actually wish to submit Projects.

To provide consistency with the rest of the ALMA software we also use the ALMA Common Software<sup>3</sup> (ACS) to support our distributed computing needs.

## 3. TOP LEVEL ARCHITECTURE

The basic architecture of the ALMA Observing Preparation subsystem is that of a “traditional” object-oriented three-tier model: editors present views to and retrieve information from the users; business logic interprets these data, and a persistence layer is responsible for storing the information. However, the ALMA “business logic” layer is, of course, somewhat more complex than would be found in commonplace business systems, and in the presentation layer it is desirable to present highly visual and dynamic user interfaces, in addition to basic “forms-based” data entry.

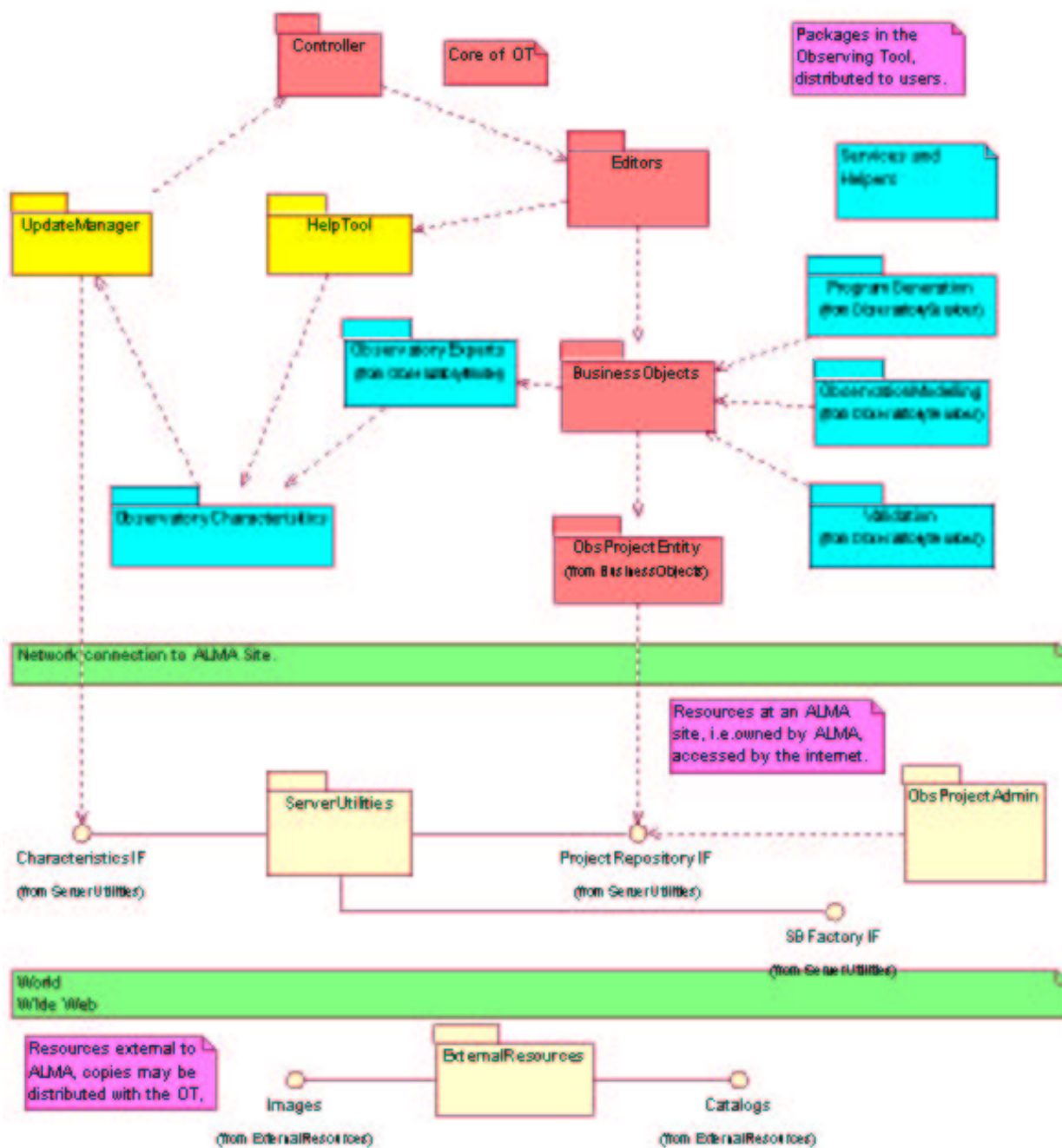


Figure 1. A UML Package Diagram Summarising the Architecture of the Software

The architecture, in the form of analysis packages, is shown in figure 1. Here is a brief description of the packages and their role in the design.

- UpdateManager: Provides updates, including initial installation, to distributed copies of the OT (from the server). We are currently using Java Web Start for this.
- Controller: Entry point for the OT, component and top level framework.
- BusinessObjects: Implementation of the “business logic” layer. The classes here delegate attribute-value storage to the “ObsProjectEntity” classes, whilst adding value in the guise of domain knowledge and the relationship between different elements in the data model. For instance it is at this layer that a “Frequency” class will know that the “Receiver” may change if the frequency value does.
- ObsProjectEntity: This contains binding classes for the XML schema. All persistent data must be in this package. The binding classes and the accompanying XML schema are automatically generated from a UML Class diagram.<sup>2</sup>
- Editors: This represents the presentation layer of the system. It consists of a number of sub-packages each aimed at supporting editing for different parts of the overall data model. Some examples are:
  - ObsProjectEditor: Top level user interface to an observing project.
  - ProposalEditor: The user interface to Phase I specific information.
  - ReviewingEditor: Interface to providing referee and reviewing input.
  - SpatialEditor: Entry of main target and mapping area information.
  - SpectralEditor: Entry of main frequency and correlator setup information.
  - Batchmode: A batch (non-gui) interface to the OT. This is a particularly useful interface for testing.but there are many others.
- HelpTool: Presentation and browsing of User Guide, context sensitive help.
- ObsProjectAdmin: Simple administration of Observing Projects by observatory staff.
- ObservatoryExperts: Classes with intelligence of aspects of ALMA. These helper classes provide straightforward support for simple queries from the business objects, for instance: what is the best receiver to use for this frequency?
- ObservatoryCharacteristics: ALMA repository for array information, site information and observatory policies. These databases provide current information for use by the ObservatoryExperts and Services.
- ObservatoryServices: There are a number of self-contained services for Observing Projects.
  - ProgramGeneration: Provides the algorithms that generate full programs and SBs from astronomical information.
  - ObservatoryModelling: Services supporting the modelling of observations - for instance exposure time calculation.
  - Validation: Provision of services that validate Observing Projects.
- ServerUtilities: The server side components, interface to the project repository etc.
- ExternalResources: Provision of interfaces to external catalogs and image repositories.
- Utilities (not shown): Utilities package to serve the rest of the subsystem. Two examples are the MVC Framework and the Perspectives framework

Work on the software is in progress and will not be complete for several years. In this short paper we can only describe a small part, so we outline below three key areas that are well advanced and are in support of future development.

### 3.1. Business Objects: Adding value to the Data Model

In the business objects each layer of the design adds value to the level below it. The base level of a series of XML schema definitions (in XSD files) define the format of data for interchange between ALMA subsystems.

These XSD files are then processed by the code generator<sup>2,3</sup> to produce Java classes corresponding to the objects defined in the schemas. These classes are referred to as the binding classes.

As their classes are tightly bound to the XML descriptions, objects of the binding classes have several nice properties:

- They can be streamed out and in easily as XML descriptions
- They can be sent from sub-system to sub-system via the mechanisms provided by the ACS.
- They can be stored to and retrieved from the Archive easily.

Whilst they provide a useful basis for representation, storage and communication of data, the binding classes do not have any functionality beyond this. The Observation Preparation subset, however, needs to be able to extend the binding classes to incorporate functionality and intelligence for the subsystem.

A mechanism was required to preserve changes made by the project team to the binding classes, to allow us to extend them to suit ALMA's needs without having to redo the majority of this coding after every minor data model change..

The two mechanisms considered were to subclass the binding classes or to write delegating classes. The mechanism chosen was delegation, as it was felt that we wanted to be able to have an inheritance hierarchy independent of that in the binding classes. For example, the binding classes have a concept of an Entity which is related to the granularity of data within ACS; however, the various classes which are entities are pretty much unrelated to each other within the Observation Preparation subsystem, and therefore it was deemed better to reserve the inheritance hierarchy for other things germane to the Observation Preparation (e.g. the various shapes that the user can specify to scan are all sub-classes of one generic TargetArea class).

An additional benefit was that some of the choices made by the code generation in its conversion of XSD constructs into Java can be "tidied up" by the wrapper classes, making them more programmer-friendly.

The resultant classes form the business layer of the system, and are termed the Business Objects.

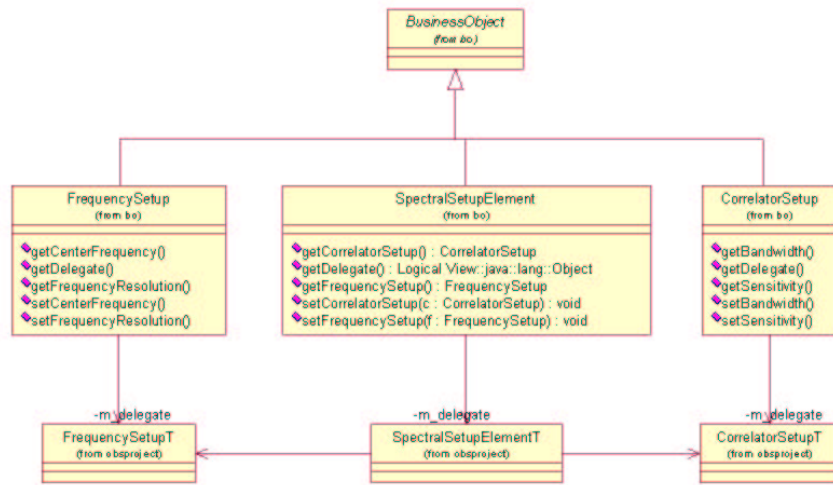
Fig. 2 shows an example of the relationships: the generated binding classes are the bottom layer.

### 3.2. An MVC Framework: The Foundations of Editing

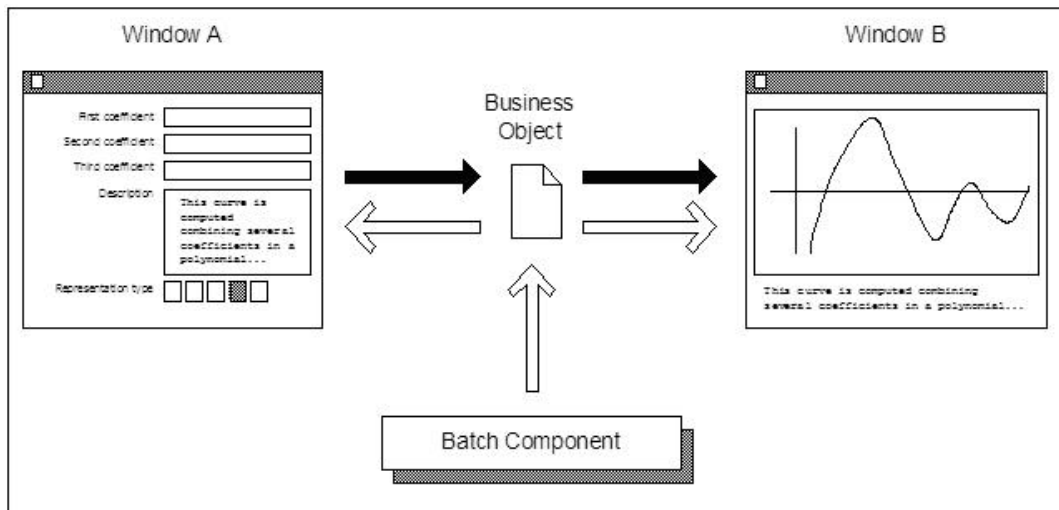
The MVC package implements a Model/View/Controller system; it was designed to allow easy creation of GUIs for business objects. One of the main requirements was that multiple (possibly different) GUIs may render the same run-time business object (instance); GUIs should be kept in sync with the object and with each other at all times. Fig. 3 shows a possible run-time configuration. Two windows render the same object, which is also subject to updates originating from a batch user. If the user modifies the contents of a widget in Window A, that change is reflected in the corresponding Business Object attribute as well as in Window B (solid arrows); likewise, a change caused by the batch user will be displayed in both windows (hollow arrows).

The main players in the package include:

- Document: The document is the actual object being displayed or edited. Any object with public field accessors can play the role of document in an MVC application (as such, one finds no class or interface called Document in the MVC package). Knowledge about the document is restricted to its Model(s). A document can be associated to multiple Models, each offering a different interface to the document.
- Model: A Model implements an interface between the Document and its Views. (A Model can be associated to multiple Views.) The Model holds a reference to the Document and must know how to update the Document's fields, but does not need to know anything about the Views. Communication between Model and View is indirect, via the Controller.



**Figure 2.** Relationship of the Business Objects to the Binding Classes



**Figure 3.** The MVC Concept

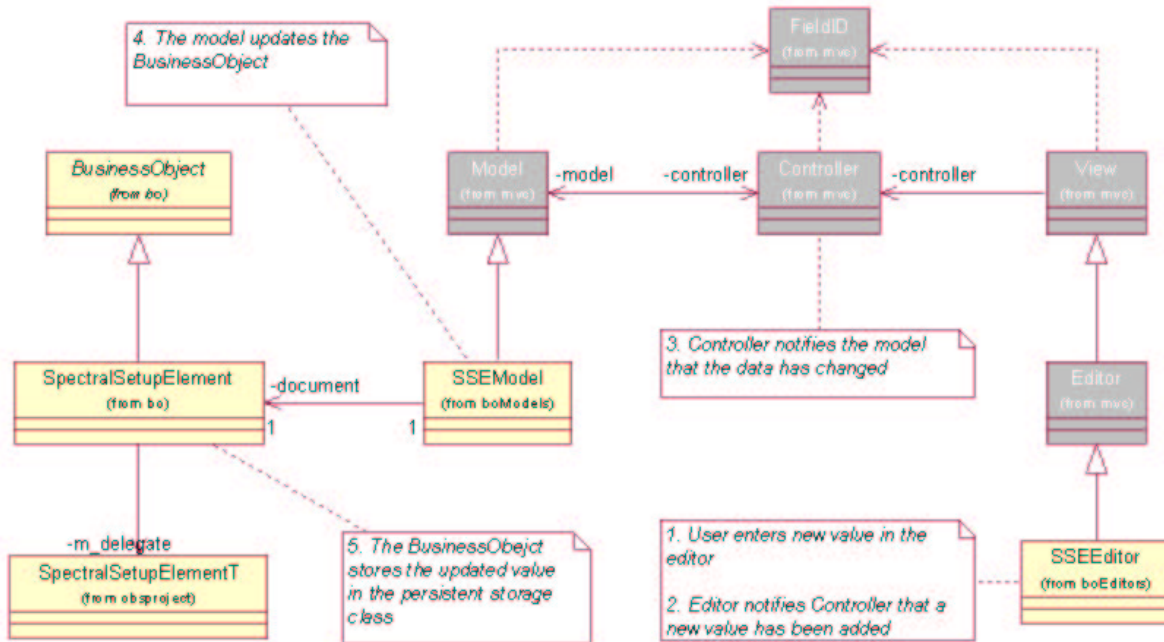


Figure 4. Class diagram illustrating use of the MVC framework

- View: A View renders a Model on the screen. It is implemented as a subclass of JPanel, and can be included in other Views, in top-level windows, etc. An Editor contains a number of Swing widgets, and must know how to update their contents, but it doesn't need to know anything about its Model. An Editor is associated to a single Model.
- Editor: An Editor is a special type of View, allowing the user to alter the Document (via its Model).
- Controller: It performs communication between Models and Editors. It informs the Model when some Editor widget has changed, so that the Model can correspondingly alter a Document's attribute. In turn, the Controller informs all Views and Editors when some Model attribute was modified, so that they can render that change on the screen.
- Field ID: Models know how to map a Field ID to a Document's attribute, while Editors know how to map a Field ID to a GUI widget. In other words, Field IDs couple GUI widgets to Document attributes. When the Model is informed by the Controller, it uses a Field ID to select which data field of the document needs to be updated. When an Editor (or View) is informed by the Controller, it uses a Field ID to refresh the proper GUI widget.

The class diagram in Fig. 4 shows the relationship among these classes, and how business objects interact with them. Notes on the diagram illustrate a typical use.

### 3.3. Perspectives - Supporting User Choice

The ALMA OT uses "Perspectives" to allow user invoked rearrangement of its GUI at runtime. When building a GUI for operation on complex data, it is often not clear how to organize the GUI in an ergonomic way. On the one hand, there typically are many single GUI parts focusing on different aspects of the data that can hardly be all viewed on the screen at the same time for space limitations. On the other hand, different users may have quite different ideas about what is important and what can be neglected in the presentation. Although the developer

tries to arrive at a suitable decision on how to set up the GUI he or she is unlikely to be able to satisfy all users and all needs.

A way to address this is to allow the user to restructure the GUI at runtime. This can be achieved through "Perspectives": they are sets of GUI parts that are semantically related. A developer can predefine perspectives according to, e.g., the different phases a user will go through when working with the data. And he/she can allow the user to redefine the perspectives at runtime.

The user can, via a hotkey, a button, or by implicitly finishing one working phase and starting another, switch the layout and presentation of the GUI. They can also adapt the layout and choose what to see and what not. When the user activates a perspective, the GUI is instantly restructured at runtime.

As the OT is written in Java, this framework is based on the Java Swing GUI toolkit. A Swing GUI is an object tree of visualization components. A perspective is a sub-tree of that. All perspectives have a common root component, the so-called mount point. When switching from one perspective to another, the sub-tree underneath the mount point is replaced by another sub-tree.

Some components (the "views") are logically shared among several sub-trees while others (the "skeleton") are unique to a single sub-tree.

Java provides a standard mechanism for freezing an object graph's current runtime status ("serialization"). When asked to switch from one perspective to another, the Perspectives Framework freezes the current skeleton and copies it to a memory buffer; it then unfreezes the skeleton from another memory buffer and moves the views from the old to the new skeleton (thus, the two skeletons logically share views - physically, a view never exists in two trees at the same time). Finally it replaces the old sub-tree with new sub-tree underneath the mount point in the GUI object tree.

A Perspective can be made persistent by writing the content of its corresponding memory buffer to disk - user preferences can be stored.

Perspectives are a good and useful way of dealing with some of the common problems of complex GUI design. Our Perspectives Framework has a small footprint, offers good performance, and is easy to use.

#### **4. NEAR TERM CHALLENGES**

Towards the end of 2004 it is intended that the ALMA software is used on the ALMA Antenna Test Facility. The OT, and associated software, will play a role in these tests by providing the facilities needed for "expert" users to create observing scripts and define observing modes. The core of this work will be in refining the data model to support observing via SBs and the provision of the editors allowing the users to create and populate the SBs. We also intend first non-prototype versions of our validation and program generation services. In return we expect to gain significant experience in how the OT is used by "real users" and how it can best support submillimetre interferometry.

#### **5. CONCLUSIONS AND THE FUTURE**

We have created the foundations upon which a flexible Observing Tool application can be developed. By the end of 2004 a data model will support the main observing definition structures, and the use of the OT at the ATF will be giving us the opportunity to stabilise the system view of ALMA observing. Following this phase development will shift back to the science view, and we will concentrate on a user friendly interface to observing with what will be the World's largest submillimetre interferometer. The ALMA Observing Tool will present the general users' first view of the ALMA Observatory. It is important that we make it a satisfying experience.

#### **ACKNOWLEDGMENTS**

The design of the MVC framework owes much to work done by Markus Voelter for his FAF package.

Thanks to Steve Scott of Owens Valley Radio Observatory for discussions and input to early concepts.

## REFERENCES

1. J. Schwarz, A. Farris, and H. Sommer, “The ALMA Software Architecture,” in *Advanced Software, Control and Communication Systems for Astronomy*, H. Lewis and G. Raffi, eds., *Proc. SPIE* **5496**, 2004.
2. H. Sommer, G. Chiozzi, and K. Zagar, “Container-component model and XML in ALMA ACS,” in *Advanced Software, Control and Communication Systems for Astronomy*, H. Lewis and G. Raffi, eds., *Proc. SPIE* **5496**, 2004.
3. G. Chiozzi, B. Jeram, H. Sommer, A. Caproni, M. Plesko, M. Sekoranja, K. Zagar, D. Fugate, P. DiMarcantonio, and R. Cirami, “The ALMA Common Software: a developer friendly CORBA-based framework,” in *Advanced Software, Control and Communication Systems for Astronomy*, H. Lewis and G. Raffi, eds., *Proc. SPIE* **5496**, 2004.