ACS Workshop — An Experience Teaching a Complex Framework

Horst H. von Brand  
Mauricio A. Araya  
Jorge Ibsen

October 3, 2008

Abstract

We describe a successful experience teaching the use of a complex framework geared towards distributed control. The framework supports several object-oriented languages, so the participants have to learn how to work in all of them. Furthermore, it is not enough to develop a single piece to learn how to work with the framework, so the design of a relatively complex system is outlined and the objective was to get a prototype of that working.

1 The ACS framework

ALMA (Atacama Large Millimeter/sub-millimeter Array) [8] is a joint astronomy project between organizations in Europe, Northamerica and Japan. ALMA will eventually consist of at least 50 twelve meter antennas with baselines up to 10 [km]. It will be located at an altitude above 5000 [m] on the Chajnantor plateau in the Chilean Atacama desert. ACS (ALMA Common Software) [4, 5] is a distributed object-oriented framework based on CORBA [11], built as a common infrastructure for the ALMA software subsystems. ACS provides a component/container design pattern [7, 13] which enforces a uniform structure on all subsystems and enables developers to focus on domain problems rather than software engineering issues. Also, it provides a clear division between domain code (components) from service code (container), such as socket manipulation, system lifecycle, logging, or data channels [12, 13]. ACS is open source, distributed under the GNU Lesser Public License [6].

The framework offers containers that manage components written in one of several object-oriented languages, namely C++ [9], Python [10] and Java [14]. The container interfaces with the components it manages through entry points defined in CORBA’s IDL language, where each component must define some minimal entry points that allow the container to manage its life cycle. Each
component, in turn, can use the container’s services to find and call methods in remote components transparently, even ones written in another language.

To be able to exploit this framework effectively, the user has to internalize the overall design pattern and become familiar with the multitude of services that the framework offers through its containers. As the framework runs over a network, the development of a distributed application presents further challenges.

ACS development follows an idiosyncratic style, somewhat in the spirit of extreme programming [2] and of test-driven development [1]. As a component is developed, a set of test cases and pieces for an automated test harness should be developed in parallel. A set of mockups of the component in various levels of detail are useful for checking its interactions with other components, and provide “real” test cases for them. Early versions of the component should be integrated into test setups and run against the rest of the system; with real components where available (or possible), with simulated components otherwise.

The ACS framework is complex, and comes with voluminous, detailed documentation, including several complete examples. Unfortunately, much of the documentation is incomplete, and not really oriented towards a beginning developer. Besides, not only the tools used are relevant, the new developer needs to get acquainted with the workflow.

This paper describes the Fourth ACS Workshop, which took place at the Universidad Técnica Federico Santa María in Valparaíso, Chile, on July 5th to 11th, 2007. Previous workshops along similar lines were held at ESO Headquarters, in Garching, Germany. Of the 5 days of the workshop, the course described here took place on July 6 and 9 to 11 (four days in all), with a full day devoted to a description of the ACS framework and about half a day to technical details on developing software under it. A more detailed breakdown is given in section 2.4. Another, informal course along the same lines was held internally in December 2007 in order to bring new participants in our ALMA-CONICYT project up to speed. Also, a similar methodology to the one in these workshops has been successfully applied in an undergraduate lecture called “Taller de Sistemas de Computación”.

2 Organization of the workshop

The workshop has the objective of teaching engineers that are knowledgeable in the supported languages on how to use the framework effectively. Therefore, a team development strategy was set up to enforce distributed development, interface and scope negotiation, and software integration procedures. At the end of the workshop, the participants will have developed a (hopefully functional) prototype of a telescope control system. Each participant was part of the development of only one component of the prototype, but shared interfaces with other components forced interaction as a client or service provider.

ACS is not only a set of libraries to support distributed components, but a collection of common patterns, procedures and software to ensure high quality
software development and integration. Therefore, it is not enough to become familiar with the code and the coding conventions used in ACS, there are also protocols to be followed when submitting a new component for inclusion. For this workshop, a full development environment is set up, complete with a CVS [3] version control system server like the one used to track the official ALMA software sources.

The participants were familiar with at least one of the programming languages supported by ACS, and had had several courses on software engineering. Most participants did not have prior experience working under such a large-scale framework, and did not have much real-world programming experience. Almost none of them had any prior knowledge of astronomy, distributed control systems, CORBA, or writing hardware drivers. All of them had at least a fair understanding of written technical English.

We had two very different hardware examples to be controlled: a Lego-based model of an ALMA antenna with its ACS control components [15] and a Celestron Nextar 4 SE amateur telescope. The workshop was set up as a three-days hands-on course, after a first day devoted to an overall explanation of the ACS architecture, and the discussion of some very simple example containers.

A set of IDL files was provided both as examples and starting points of the developments. These interfaces were based on the initial design written by the instructors for this workshop. The IDL files serve also as counterpart interfaces for testing what is being developed, using component skeletons without useful behavior or simulated components for hardware device control. The instructors took on handling the integration and quality assurance of the pieces provided by the participants.

2.1 Setup of the laboratory

The participants were divided into teams of 3 or 4, each team got one PC assigned. The machines ran Scientific Linux 4.1, one of the Linux distributions on which ACS is supported, and the full ACS software package was installed and configured on it. A CVS [3] server was set up with the ACS software to be used and the initial IDL files for reference by the participants. Here, the participants uploaded what they developed for integration and testing. Each group could select to run their software against the other skeleton components or some of the components being written by the other groups.

Several of the participants had their own machines, which were also integrated into the network, but the main development work was done on the machines provided, as one team member was actually writing, and the others helping from nearby. Care was taken to mix people more experienced with ACS into each group.

2.2 Design of the prototype to be developed

The system to be developed is composed of seven main component interfaces:
**Database:** System entry point for astronomers. It allows the astronomer to set up proposals, insert them into the queue of observations, and query their status. This component was written in Java.

**Scheduler:** It selects proposals from the Database, executes the observations and stores the results. This component was written in Java.

**Console:** The entry point for the telescope operator. It starts and stops the Scheduler's automatic mode, and allows the operator to manipulate the low-level components directly. This component was to be written in C++, but ended up Python code including a rudimentary graphical user interface.

**Telescope:** This component is responsible for managing any type of telescope. It includes the control loop to move the telescope to the requested coordinates and calls the instrument to retrieve an image. This component was written in C++.

**DevTelescope:** A low-level component that communicates with the mount hardware. Each different hardware, such as the Lego or Nextar telescope, should re-implement only this interface. This component is meant to be used by the Telescope component, and should map the general control sequences to specific hardware commands. This component was written in C++.

**Instrument:** This component handles the camera, switching it on and off, and collects an image with a given exposure time. This component was written in C++.

**DevInstrument:** A low-level component to control a specific camera or CCD. The general camera properties are mapped to the specific driver interface to be called by the general Instrument component. This component was written in C++.

Two DevTelescope components with the same interface were to be developed: one for the Lego model and the other for the Nextar amateur telescope.

The components' interfaces and interactions with the rest of the system were given by informal sketches, supported by short prose explanations (see figures 1 and 2) plus skeleton IDL definitions for the components. The whole system was discussed in detail the first hands-on day of the workshop, where some rough spots of the overall design were corrected. To simplify the work to be done, error and exception handling were kept to the absolute minimum.

The roadmap for the workshop was defined on this first day. The next two days each team was to develop the component they got assigned, while the last day was reserved for integration of the results. The development of the components was almost completely independent, except for occasional discussions on interface changes. At the outset it was explicitly stated that if the deadlines could not be met, the requirements would be de-scoped, in order to get some type of complete system working.
2.3 Supporting roles

Several senior participants in the ALMA UTFSM Group, which are already familiar with ACS, set up the laboratory in which the workshop took place with the help of ACS engineers. As this was during winter break at the UTFSM, we could lend computers from a student laboratory and configure them from the ground up for use in the workshop. The more experienced developers (as the Integration and Testing team) also took over roles writing test components, handling the overall configuration of the deployment environment, testing and integrating the components developed by the groups, besides general troubleshooting and helping the participants over rough spots.

The groups where composed by three or four participants, each group in charge of developing one component. The groups have only one computer, and the least experienced member was responsible of actually writing the code. This way, the newcomer can learn about the specific programming API (Application Programming Interface), while the experienced members can discuss the com-
ponent design.

As each group develops a different component, the collaboration of the groups encourages learning; the copy/paste paradigm is not useful in this setup. Also, at partial stages of the projects, the groups may need some functionality from other components to test the current development. This is a fairly high pressure for the development groups, which simulates the real-world stress of a software development process. Therefore, the instructor takes a supporting role rather than just a “punisher” role. In summary, the Integration and Testing group plus the group interaction, develops the needed pressure to the development group.

2.4 Development of the workshop

Before the course itself, a full day was devoted to a detailed explanation of ACS. The points covered were a general overview, an explanation of the lifecycle management of the components by the containers, and a short demo using the Lego model [15]. In the afternoon of this day we got into the details of programming on ACS, how to configure an ACS deployment (the specific configurations of the components, and the assignment of some of the components to specific containers running on particular hosts), the automated test framework ACS uses, and logging and error handling. Most of the material covered here was from previous versions of the workshop. Very rough simulators for the components were written in parallel to the development of the workshop to be used as counterparts for the rest of the system.

The second day (first hands-on day) started with the assignment of participants to the components to be developed, and an overall explanation of the objectives. Here the overall design was presented and discussed. Then each group went on to learn how to start and stop ACS, how to build and test the interface stubs using IDL, and how to connect a component to its container. Most groups wrote simple components in all three supported languages. The rest of the day was devoted to building initial versions of the components, getting them to build and integrate cleanly with the respective containers. The day ended with a Questions and Answers session.

The third day started with a Question and Answers session on the problems encountered during the previous day. This day was devoted to building components, handling their configuration needs using the Configuration Data Base (CDB), automated testing, developing tests, and refining the components started the day before. Some of this work included redefining interfaces among components, as the initial definitions proposed to the class turned out to be incomplete or even contradictory. The interface redesign was done through a discussion including all participants, even if their components were not involved. This was very valuable, as it served to reinforce the concepts involved in the interface definition process, and the compromises and negotiation involved. Also, each few hours the instructors interrupted the work to go through all groups asking for the current status, so everyone knew the general progress. Again, the day ended with a Questions and Answers session.
The fourth day involved building the distributed system by integrating the components, and final tests. In this part the instructors had a lot of work taking the components developed by the individual teams, and integrating them. This helped to show how a particular deployment has to be set up. It provided insight into the testing and quality assurance jobs in the team.

One of the tutors checked the progress regularly, and encouraged groups that were lagging behind to take a look at their work, and either speed up or redefine their objectives. This also helped to keep everybody abreast of the overall progress.

3 Results

The final result was a working prototype of a control system for a telescope, including both mounts (Leggo model and amateur telescope). This was achieved ahead of schedule, and no de-scoping was required. In part this was due to having some participants that already knew ACS. Nonetheless, even for them it was a valuable view of a complete deployment, as they had concentrated only on their specific components before. Also, ACS is rapidly evolving, so the workshop helped the older members of the project team to learn how to use new features effectively.

As expected, most of the problems the second day (first hands-on day) were in software engineering concepts, in handling IDL interface definitions, and the distinction between stubs and implementation in Java. The other languages are simpler to handle.

The third day showed mostly language related issues, and clarifications of some of the requirements. Some error handling and logging was introduced into some of the prototype components, thus needing help in those areas.

The fourth day the components developed had to be integrated, and the support team (Integration and Testing) set up several test run scripts. Some of the test runs failed at first, the problems were solved on the spot by the teams responsible for the affected components. Some error cases were also tried successfully, even though (for simplification) most error handling had been explicitly left out of the design.

The workshop has been very successful in rapidly training interested engineers in the use of the framework. Most of them went on to develop much more complex components with relatively little outside help. Even given the limited time of the workshop, functional prototypes that became the base for later full components were developed from scratch.

Being able to test the component under development by interacting with other components, even very rough simulations, proved invaluable, as did the extensive instrumentation and logging support in ACS. The availability of a generic component with a GUI that allows to set and interrogate any property of a given component was very useful.

In summary, the teaching model behind this workshop was very successful. This model was strongly based on the continuous integration paradigm, where
each group that develops a different part of the software, must integrate their advances every time. Also, to a large degree the success of the workshop was based on the complex, challenging, and interesting example; which is much more motivating than a homework assignment with well-known results or no further use, and which will be forgotten soon.

4 Conclusions

Giving a set of teams a roughly defined design to work on, and encouraging communication between them and with the instructors, testing their work as they progress, helped them to learn how to program successfully in this complex environment. Critical motivation was given in this particular case by the ability to see the results in the form of a Lego model or a real telescope moving according to the commands given.

The example project could still be improved and the work done could be used as a starting point for an advanced track that could include both ACS (properties, alarms, advanced error and logging usage) and software engineering (test driven development, refactoring, agile development) topics.

Similar workshops on ACS were later held at UTFSM as a way of bringing new recruits up to speed with respect to ACS. The students that attended the original workshop and also those who attended the local, cut-down versions, became fully productive ACS developers in a short time. The ALMA project has hired or is considering hiring several of the participants, as the headstart they got from the workshop and further work with ACS in the current ALMA-CONICYT project has meant a distinctive advantage.

Also, due the success of the methodology, the experience was reproduced in an undergraduate course at UTFSM, but for teaching the J2EE infrastructure rather than ACS. Following the same idea, several teams were setup, each one with a different task to solve. At the end of the course, a few integration sessions were held in order to finish the whole software. The novelty here is that the same methodology can be used to teach several complex environments, and is not limited in any case to the ALMA software.

Acknowledgments

This work was partially financed by project ALMA-CONICYT 31060008, and could not have been done without the support of ALMA and ESO. Mauricio Araya's work was supported by a UTFSM Iniciación Científica grant.

References


