ALMA software releases versus quality management: and the winner is …

Erik Allaert*[a], Moreno Pasquato[a], Rubén Soto[b]

[a]European Southern Observatory, Garching bei München, Germany
[b]Joint ALMA Observatory, Santiago, Chile

ABSTRACT

After its inauguration and the formal completion of the construction phase, the software development effort at the Atacama Large Millimeter/submillimeter Array (ALMA) continues at roughly the same level as during construction – gradually adding capabilities as required by and offered to the scientific community. In the run-up to a new yearly Observing Cycle several software releases have to be prepared, incorporating this new functionality. However, the ALMA observatory is used on a daily basis to produce scientific data for the approved projects within the current Observing Cycle, and also by engineering teams to extend existing capabilities or to diagnose and fix problems – so the preparation of new software releases up to their deployment competes for resources with all other activities. Testing a new release and ensuring its quality is of course fundamental, but can on the other hand not monopolize the observatory's resources or jeopardize its commitments to the scientific community.

Keywords: ALMA, computing, software releases, software quality

1. INTRODUCTION

The Atacama Large Millimetre /submillimetre Array (ALMA) consists of 66 high-precision radio-antennas, located on the Chajnantor plateau in the Chilean Andes, at an elevation of 5000 metres. Its construction and operation is a joint effort from partners on 3 different continents: the European Southern Observatory (ESO), the U.S. National Science Foundation (NSF) and the National Institutes of Natural Sciences (NINS) of Japan. The development of the software to operate this sophisticated array, acquire, monitor and evaluate scientific data is equally distributed over these three continents, involving many institutes and software engineers.

The ALMA Observatory was inaugurated in March 2013, and the construction phase has formally been completed, but the software development effort continues at roughly the same level as during construction, to cope with the increased capabilities offered to the scientific community in the context of ALMA's yearly Observing Cycles. This implies that in the course of a year several software releases have to be prepared, dealing with everything from proposal preparation and submission software up to the deployment of the online software commanding the array at the official start of a new cycle at the observatory. These software releases follow a pre-defined process; there are several quality aspects followed up continuously as part of this process, providing a quality status overview to the developers, end-users (scientists and engineers), plus the Release and Acceptance Managers. The corresponding report assessing the software quality of a new release is submitted to the Acceptance Review.

As the ALMA observatory is fully operational, it is used on a daily basis to produce scientific data for the approved projects, and also by engineering teams to extend existing capabilities or to diagnose and fix problems. Hence the preparation of new software releases up to their deployment is "just one more activity", competing for resources with all other activities. Testing a new release and ensuring its quality is acceptable to the observatory is of course fundamental, but can on the other hand not monopolize the observatory's resources or jeopardize its commitments to the scientific community, such as publicly announced milestones for the start of new observation cycles.

* eallaert@eso.org; phone +49 89 32006262; www.eso.org

In this paper we describe the tools and metrics used so far to monitor the quality of ALMA software releases, and the results this has produced. After having applied this for a number of releases, we have also identified several areas within the release or acceptance process where improvement is desired or needed, and where the stumbling blocks are.

2. THE ALMA SOFTWARE DEVELOPMENT AND DELIVERY PROCESS

2.1 Organization

The ALMA software as a whole consists of several "subsystems", like the Baseline Correlator software or the Observation Preparation software. Each of these subsystems has on the computing side a lead-engineer and typically several software developers, who can be working in geographically distinct locations, even in different continents. All ALMA software engineers working across the different executives and the Joint ALMA Observatory (JAO) on one or more of these subsystems make up ALMA's Integrated Computing Team (ICT). For each subsystem there is also on the science side a so-called Subsystem-Scientist (SsS), who acts as the interface between ICT and the Science department. This person assists in clarifying the requirements towards the developer(s), the priorities of the various requests and also coordinates or participates personally to the testing of the new features in the later phases of the development and deployment process.

2.2 Software development

The ALMA software is of course kept under configuration control with the help a Versioning Control System (VCS); currently we use Subversion (SVN). The repository has a tree-structure reflecting the various subsystems. As there are dependencies between several of these subsystems, building and testing the entire ALMA software has to happen in a particular order.

The software is developed and deployed on customized Red Hat Enterprise Linux systems, and relies for most of the subsystems on the in-house developed ALMA Common Software (ACS) as middleware/framework [1]. Subsystems are subdivided into a set of "modules" that provide particular functionality; these modules are organized in a particular directory structure, and that structure is expected by the provided acsMakefile (a makefile infrastructure based on GNU make). One of the subdirectories of a module is supposed to contain software to test the functionality of this module in an automatic way (unit testing), again using the acsMakefile and accompanying tools. The development and maintenance of such unit test is the responsibility of the developer(s) editing the module.

2.3 Build and unit-test infrastructure

Software consists of programs (code), procedures, documentation and data. Therefore, when we refer to software quality we implicitly consider the quality of each of its components.

Software quality assurance uses static and dynamic techniques to inspect the software. It is difficult to automate reviews on procedures and documents, but there are many tools that can help in code analysis. At ESO we use Jenkins [2], a continuous integration tool, to provide services such as code builds, code tests, calculation of code metrics and checks of coding standards.

Code Builds: every hour Jenkins polls all subsystems belonging to the repository Trunk and to circa 8 different releases (the ones actively used at the ALMA Observatory) for code changes. In case of variations, it builds the subsystems and notifies developers per email about eventual failures. If the built subsystems contain code on which others rely upon, Jenkins triggers further builds following the dependency tree. Feedback provided to developers offers them the chance to react fast in order to fix the code.

Code Tests: for each subsystem that is built, Jenkins triggers a test process too. All unit tests are executed and, as for the build, failure notification emails are sent (tests should be developed taking into consideration quality factors and should implement all the requirements classified in them).

Developers are offered the possibility to choose between reports and/or detailed email notifications and to decide whether to build and/or test the code.

Static analysis: ALMA software hosts many different programming languages. Jenkins inspects weekly the code and publishes java/perl/python coding standards and violations reports using 5 different tools (one of them developed in-house). Metrics are stored for 52 weeks and their trend over time is used to evaluate improvement/deterioration of quality.

Different executives may follow different approaches on what part of the ALMA software they build and test, as they may be responsible for a few subsystems only and focus on these only. They could also skip the releases that they are not involved in, or limit themselves to the building of the software. In any case, all executives rely on some version of Jenkins as the tool for triggering builds of their pieces of the ALMA software.

2.4 Issue tracking

ALMA uses in various areas Atlassian's JIRA tool [3] for issue tracking. Also all activities from the ICT group are followed up via a dedicated JIRA project. This allows to track progress on requests for new features, improvements, bug issues, etcetera. The workflow has been configured to our specific needs, and depends on the issue type. Also the fields that need to be filled out depend on the issue type, allowing to extract statistics and relevant information when and where required.

2.5 Software delivery

The software delivery process adopted by ALMA is based on an incremental releases schema [4] that accumulates new functionality to be delivered in a bi-monthly schedule for science commissioning. The release cycle considers development, testing (verification and validation) and integration phases with a formal handover between each phase. This model differs from the paradigm adopted during the construction phase by the smaller amount of features and improvements included per release; however, it puts more emphasis on facilitating the integration, testing and debugging of problems found during each phase. Of course, features and improvements must be scheduled according to the observatory's milestones.

Once development and developer testing has been completed by an ICT group, the relevant tickets are handed over to the Integration & Release Management (IRM) ICT group for integration into the release branch and testing. The developers provide the testing instructions to the IRM group and collaborate with them to ensure a comprehensive and successful testing campaign. Each feature or bug fix must be described in a corresponding JIRA ticket and contain the testing instructions. IRM will collect the tickets for a dedicated incremental release and test them in the given time-window for this incremental release – we call this the "verification phase". This may involve participation from scientific staff or the developers, in particular for new features or when science data is produced or accessed. The outcome is a formal report outlining all contained features, links to test results, bug fixes, and a complete list of open issues.

Once the verification phase is finished, the "validation phase" starts. This phase is directed by the Department of Science Operations (DSO), and involves the subsystem-scientists. It attempts to realistically reflect the final use of the software in the observatory observing process, and apart from validating the new features also looks at eventual side-effects like software performance and stability. Again this leads to a formal test report for each subsystem.

When all features necessary for deployment have been accumulated, eventually via multiple incremental releases, the formal acceptance process starts. The first step is to place the release under change control by the Software Configuration Control Board (SCCB). This means only fixes explicitly requested and accepted by the Release Manager can be committed to the release branch, and major changes will have to be approved by the SCCB. The Acceptance Manager will organize a Test Report Review (TRR), to evaluate the reports from the validation phase and prioritize unresolved issues that need to be fixed before the real Acceptance Testing can start. Finally an Acceptance Review is held, and if the Acceptance Review Committee gives a positive recommendation to the Director, he can formally authorize the deployment of the release at the Observatory.

2.6 Software deployment

The deployment of accepted software is done simultaneously at all ALMA sites, i.e. the JAO and the different ALMA Regional Centers (ARCs). It also includes the application of database model changes required by the new software that are applied at Chile and replicated to the ARCs. This operation usually involves downtime of the applications included in the acceptance. Deployment of new versions must of course take place before the observatory milestone that triggered the acceptance process. Preliminary tests are performed in environments replicated from the production one in order to make sure that everything is ready for the final deployment. All this work is coordinated by the deployment manager who submits a report with the status of the tests and the software deployment at each site. He is also responsible for tracking any issue associated to the deployment of new software.

3. SOFTWARE QUALITY ASSURANCE GOALS

3.1 Definition of quality and quality management

Opinions about what the term "quality" really stands for generally diverge, and the world of software engineering is no exception to that. For our purposes, let's stick to the definition given by Christof Ebert [5]: "Quality it the ability of a set of inherent characteristics of a product, service, product component, or process to fulfil requirements of customers". Consequently, "Quality management is the sum of all planned systematic activities and processes for creating, controlling, and assuring quality". With "customers" we mean the end-users and stakeholders, i.e. the people who matter in a project. A continuous quality management activity should therefore be to interact with the customers, understand their requirements and their level of satisfaction with the fulfilment of requirements (or the level of fulfilment they perceive).

Some authors – like Pressman [6] – see a deficiency in the above definition, as it could be interpreted that it frees the customers from any professional responsibility. That is of course not the idea: the customers are accountable for providing clear, well-founded functional and performance requirements, plus validating that the implemented software complies with these requirements.

3.2 Commitment to quality

If the meaning and proper definition of quality is somewhat controversial, there is unanimity about the fact that quality is a shared responsibility. Goals can be achieved only with contributions from all parties belonging to the project, from the directors and project managers to the developers and testers to the customers. If there is no culture of quality at all levels of the organization, the quality goals are very likely not going to be accomplished.

It should be clear that commitment to quality means in practice we need to find the right balance between quality-related activities and other activities that cannot be postponed. But if it has been decided that a quality goal should be achieved, then the planning should reflect this. This might have consequences on the number of features to implement (reduce the number of features for a certain release to dedicate more time for instance to testing) or on the time to finish them (dedicate more time to complete critical features) or on the resources allocation (get more time at the operational site to run performance tests with real hardware, allocate from developers to this activity, …).

3.3 Metrics

In the process of improving quality, the first step is to take inventory of where we stand: what is OK, what's missing, what can or should be improved. Metrics are needed for that, for measuring deviations, and their history will show the trend – hopefully in the right direction. Metrics can also help to specify where we want to go to, and which road to take to get there. Metrication itself is not a standalone goal, but is a method to obtain certain objectives of quality. Hence what we measure should be driven by what targets we have or what we want/need to improve (attributes to be measured). The metrics should be relevant for all parties involved in the project; rather than rigorously collecting all possible metrics out of the book, we should select the right ones for the project, in the sense that they are meaningful, well understood and able to give useful information to QA responsible persons, developers, project managers and stakeholders. Metrics should be valid (measure the required attributes) and reliable too (produce similar results under similar conditions). Once metrics have been identified, comparative values should be defined based on standards, previous year's performance, etc. Finally, reporting methods and frequency of reporting concludes the process of quality metrics definition.

3.4 Quality goals

For the ALMA project there is an ambitious list of quality goals. These include:

- The metrication program; this includes monitoring the coverage offered by the automated tests, checking compliance with some coding standards, the results of the testing phases, how many issues of which type and severity go into a release. Much of this information can be extracted from JIRA and Jenkins.
- Monitoring the unit tests: do such tests exist in the first place? Do they run automatically? In case of failure is there an acceptable justification?
- Classification of bugs: when bugs are categorized, we will hopefully see the weak spots and concentrate on improving in this area. There is a lot of theory about this and a huge variety of possible categories. For instance the original proposal from Beizer [7] – e.g. "unclear requirement" or "inaccurate design".
- JIRA workflow monitoring: we rely heavily on the use of JIRA to track our activities and monitor the status of issues. It is essential that the type-dependent workflow is followed, the tickets are kept up-to-date within a reasonable amount of time, and tickets progress through the workflow at a pace corresponding to their priority.
- Empower the verification phase: adequate time and resources have to be allocated for an independent group of testers to use real hardware at the Observatory to execute regression testing [8] and verify new features. The earlier on problems are detected, the easier/cheaper it is to cure them.
- Release planning: it is important that developers do not consider their involvement finished until the scientists/customers have validated the software and the release has been accepted; it is equally important to have the customers on-board in the entire development process up to the deployment. For that, both sides need to have a clear view on the planning and know when the software is supposed to be in which phase. We also hold at the end of every acceptance release a "retrospective meeting" between the main Acceptance Review Committee members for better understanding what went wrong and what can be further improved in the software development, acceptance and deployment process.
- Software quality management itself: it is not enough to appoint one or more persons and give them the corresponding title. The quality management team must be empowered to interface with all levels involved in the software development and deployment process, to agree on the quality goals mentioned above, establish and monitor them, guard continuously their achievement and intervene if this would not be the case.

4. REAL-LIFE QUALITY ASPECTS OF SOFTWARE RELEASES

This section outlines a number of software quality issues that we are faced with, in the context of producing software releases in an operational Observatory environment – and how we address them (or intend to address them). These matters are typically identified and written down as the result of retrospective meetings.

4.1 Issues are not current

It often happens that JIRA-issues are updated late compared to some activity involving this ticket, i.e. their status as kept in the JIRA database does not reflect the actual status in reality. People tend to update tickets only as deadlines approach – in some cases only minutes before the deadline – although the related activities took place long before. Anybody looking at the contents or status of such an outdated ticket will end up with incomplete or wrong information. As JIRA is used for metrication and to know how well a release is progressing, it is fundamental that the issues are kept up-to-date in near real-time.

There is in our case only a very limited set of activities that lead to some automatic update of a JIRA-issue (like committing code to SVN). Hence it is important to clarify to all involved that keeping tickets up-to-date is important; from there on we need to rely largely on the self-discipline of all JIRA-users.

4.2 Use of non-authoritative data

JIRA allows to filter out a set of issues, and store them e.g. in an Excel table. Effectively some people tend to extract such lists at the early stages of a release process, and from that point on they keep on working with these tables, without

giving the necessary feed-back into JIRA, nor using newer information from JIRA as the release proceeds. This is similar to – or in fact worse than – the scenario described in section 4.1.

Here too we have to spread awareness about the necessity to use JIRA as the one and only authoritative source of information about an issue, and rely on the self-discipline of the users. In any case, this behaviour may also be triggered by a lack of knowledge on the use of JIRA. It is our task to make this use as straightforward as possible, by providing common or popular filters, Kanban boards etc.

4.3 Late blocking issues

It happens that blocking issues are identified when we are getting very close to the acceptance date. So the risk is that then the Acceptance Review will take place while there are active blockers; the acceptance can then only be provisional, subject to the resolution of these blockers.

To address this, we need to put more emphasis on the verification and validation tests, so we can detect blocker issues earlier in the process. Also, we need to execute more exhaustive regression tests, also early on during the acceptance testing, to have sufficient time to solve eventual problems before the Acceptance Review.

This implies of course that there needs to be sufficient time and resources for these exhaustive tests, and this must be reflected in the planning of the various phases of the software release process.

4.4 De-scoping

When collecting the issues that need or are expected to go into a release, you can easily end up with a bloated wish-list of nice-to-have features, rather than a list limited to the essential, critical issues that can realistically be implemented and tested in the given time with the given resources. And it is in any case often very difficult to give estimates with narrow error margins of the effort required to implement and test an issue. The result is more often than not a too optimistic planning.

There are basically only 2 ways to deal with wit this bottleneck: either with more resources, or with fewer issues. Assuming that extra resources are difficult to obtain, the software release process must then provide options to adjust the content of a release to the available resources. This should happen as soon as it becomes evident that not all issues can make it into the release, typically long before the TRR. At that stage you risk to dilute the efforts of developers and testers over a large number of issues that in any case won't make it all into the release, rather than let them focus on the essential ones. So there needs to be a continuous negotiation about priorities between the SsS and the developers. On top of that, for major releases "priority revision milestone" needs to be added in the release process. This should come together with a consequent de-scoping of the non-essential issues.

4.5 New software delivery process

ALMA has concluded its construction phase and is now moving further into full operation. This transition implies more emphasis on system robustness and stability in order to maintain continuous operations and a reduction of the technical time dedicated to commissioning and verification. For this reason, the software delivery process needs to be adjusted to the current Observatory state. Thus, given the hardware access restrictions, simulation capabilities will play a more prominent role in the verification phase. The number of new features/improvements per release will be reduced, but the emphasis on software robustness will become essential. System stability turns into a critical point in order to maintain the observatory working most of the time. Therefore, the downtime due to new software releases must be strictly controlled and minimized.

During 2015, an agile approach for the software delivery process [4] was proposed to be adopted by the Observatory. This approach is based on the existence of a stable branch, into which new features and capabilities are merged for verification. Developers will commit functionality in separate branches and the verification team will merge them to the stable branch for verification purposes. If verification passed, science testers should validate the same functionality from the scientific point of view. After successful validation, the patch can be integrated into the stable branch and considered ready for science observations. The deployment of validated software in the production environment can be done immediately after validation, instead of waiting for an acceptance process. Bug-fixes for the software deployed in production should follow a similar approach.

This model differs of the incremental approach since the integration of new software into the stable branch is controlled by the verification team instead of the developers. Stability should be also guaranteed since only verified functionality or

bug fixes are included into the stable branch. Features or bug-fixes, which did not pass verification or validation phases, are rejected and scheduled for another iteration. The observatory's technical times are also optimized since only software, which has passed simulation tests, is considered for verification using operational hardware.

4.6 Inactive issues

For the ICT-project in JIRA there are yearly well over 2000 tickets created, distributed over various types (e.g. bugs, improvements, new features) and various software subsystems. Not all of these tickets are urgent, i.e. some of these risk to be postponed (repeatedly) in view of the higher priority ones and a lack of resources, or due to the planning which foresees that some issues should be addressed only later on. We need to review regularly this list of inactive tickets, as the planning may have evolved, and some tickets may not be current anymore due to other issues that have been implemented in the meantime.

There are different approaches on how to handle such inactive issues, and the policy chosen by different organizations varies widely - often even with an obvious lack of consensus within a single organization. Such policy ranges from “do nothing” as one extreme to “close automatically any issue type after a couple of weeks of inactivity” on the other extreme.

The arguments that are raised against automatic closing are typically:

- the fact that some tickets are inactive does not necessarily imply that they would be irrelevant or not real - they may just have a lower priority/urgency than some other tickets. Issues are meant to get fixed, instead of being closed simply because they are "too old".
- the number of open issues is not necessarily a good metric. After all, if one does a clean-up, the metric will appear better, but it does not impact the quality of the software at all - the amount of essential work to be done remains the same.
- it is hard or even impossible to come up with a general automated procedure that does everything right. It may be necessary to actually look at these inactive issues, one by one, and make a call.

On the other hand, the arguments in favour of a (occasional, periodic or continuously automatic) clean-up are:

- open issues that remain inactive for a long period obfuscate the real issues, and make it harder to focus on these real issues.
- if an issue has been inactive for so long, it is (very) unlikely that anybody will ever look at it. So better make that clear by closing it.

ALMA ICT has chosen a middle-of-the-road approach; we identify about twice a year the set of inactive issues, whereby the “inactivity criteria” depend on the issue type and its workflow status. The people involved in these tickets are notified about the inactivity, and get a couple of weeks to react on this (by adding a comment to the ticket, changing its status, priority, due date, or whatever). This is followed by a second reminder. Finally, tickets which still did not receive any attention will be closed. Typically this will result in about half of these tickets being closed (the majority by the users themselves), while the other half will get one or the other update.

5. CONCLUSIONS

Although the JAO is a fully functional observatory, it still requires regular software updates and upgrades, and it will keep on doing so in the foreseeable future. The need to test and deploy these software releases competes to some extent for array time with the scientific observations. However, it is clear that the quality and efficiency of observations depend on the quality of the underlying software. Hence the scientists have an undeniable interest in obtaining software of adequate, proven quality. This also means everyone involved in the development, deployment and use of this software must pull on the same end of the rope, and consider the other side “partner” rather than “competitor”. Using all resources to do science observations would lead to software releases of bad quality, while attempting to build the perfect piece of

software would be prohibitively expensive and take too big a drain on the observation resources. That reality steers to meeting each other half way.

At ALMA the formal application of software quality management is gradually gaining weight as consciousness about quality principles and benefits is spreading. The activities listed in this paper are executed to address some of the issues we have been confronted with. We have made progress, but still see a lot of room for improvement. However, thinking there can be "winners" and "losers" is wrong – if we don't do this right, we'll all be losers.

REFERENCES

[1] Caproni, A., Colomer, P., Jeram B., Mañas, M., Sommer, H. and Chiozzi, G., "ALMA common software from development to operations," Proc. SPIE 9913 (2016).
[2] Jenkins project, "Jenkins Documentation," <https://jenkins.io/doc/> (23 May 2016)
[3] Attlasian Company, "JIRA Software," < https://www.atlassian.com/software/jira> (21 May 2016).
[4] Soto, R., Shen, TC., Ibsen, J. and Saez, N., "ALMA Release Management: A Practical Approach," Proc. ICALEPCS 2015 (2015).
[5] Ebert, C. and Dumke, R., [Software Measurement], Springer, Heidelberg & New York (2007).
[6] Pressman, R., [Software Engineering: A Practitioner's Approach], McGraw-Hill International, London (2000).
[7] Beizer, B., [Software Testing Techniques], International Thomson Computer Press, New York (1990).
[8] Soto, R., Shen, TC., Ibsen, J., et al., "ALMA software regression tests: the evolution under an operational environment," Proc. SPIE 8451, 84511R (2012).