Astronomical Data Analysis Software and Systems X
ASP Conference Series, Vol. 238, 2001
F. R. Harnden Jr., F. A. Primini, and H. E. Payne, eds.

Software Engineering Practices for the ESO VLT Programme

Giorgio Filippi, Paola Sivera, Franco Carbognani

European Southern Observatory, Karl-Schwarzschild Str.2, D-85748 Garching bei München, Germany

Abstract. With the first light of its fourth telescope unit, the European Southern Observatory is successfully completing the commissioning of the main telescopes of the Very Large Telescope program, featuring four 8-meter telescopes. Two of these are already available to the scientific community.

The VLT Software Control group is proud of the fact that VLT commissioning has never suffered delays due to software. A decisive factor in achieving this was the use of software engineering practices, and this paper reports on the experiences of applying these practices to the VLT Control Software and building the group culture as a simple but effective set of standards and tools.

Main areas of interest are analysis and design methodology and tools, software configuration control, software testing, problem reporting and tracking, and documentation. The paper will also discuss what didn't work and changes that are needed and planned.

1. The VLT Project

The European Southern Observatory (ESO) Very Large Telescope (VLT) project consist of four 8-meter telescopes, fifteen instruments, and the VLT Interferometry (VLTI). The VLT Control Software project provides installation of all control and monitor functions can be characterized as follows:

- 3-4 millions lines of code
- 80 workstations for operations, ∼40 for development
- 100-120 software people involved
- 25-30 sites involved in total (12-15 at any given time)
- coding begun in 1991; planned project completion: 2003
- incremental installation and operation.

The VLT Software has been and will be reused on other projects (e.g., NTT, 3.6, 2.2, VST, .etc.).

2. Main Ideas

In 1991 when the VLT development began, the question was whether a traditional, basically no-rules approach would suffice, or whether a more structured

software engineering approach was needed for a project of its size and characteristics. In retrospect, it is easy to identify the decisions responsible for the successful approach:

- The software engineering (SE)/quality assurance (QA) approach was endorsed at the outset of the project and dedicated resources (one full-time person) were allocated.
- In order to allow many people to work at many different sites in a productive way, standards were defined and put in place. Whenever possible, standards were enforced by providing tools, not simply paper definitions.
- VLT Common Software was defined to collect software solutions that could be used across the project. To be effective, the Common Software has been handled like a commercial software product: non-regression testing, backward compatibility, documentation, installation of the operating system (OS) and of third party tools (GNU, etc.) included as part of the VLT Common Software, and distribution to internal and external developers as releases once or twice per year.
- To support diverse needs, SE tools have been selected to allow development loops at different frequencies. The same tools have been used to support both twice-per-year Common Software releases and daily releases of new versions for the integration and commissioning of telescope and instruments.
- The Code Management tool, more than just supporting software development, became a major integration tool.
- Automatic testing was used extensively to validate new releases.

At the beginning very little was available; hence SE had to be put rapidly in place while the project was developing (see Table 1) in order to provide timely standards and tools for early project needs. It was found that by paying attention to the proper synchronization between the two processes, both SE and project needs could be met. This may be of interest for organizations or groups that intend to start but are afraid that SE practices have to be fully in place before starting. Of course this is preferable, but the other approach can also work.

The SE approach applied to the VLT project can be grouped in the following areas that are discussed below: Software Life Cycle, Documentation, Development Environment, (Automatic) Testing, Configuration Management, Releases, and Problem Reporting.

3. Software Life Cycle

The Software Life Cycle provides a reference model for project activities. It defines the phases of the project and the deliverables that should emerge from each of them. In view of the many interesting discussions of the pros and cons of various models described in the literature, it is important to remember that the S/W life cycle is a tool, not an end in itself! The SW Life Cycle has been described in the VLT SW Management Plan (ESO document VLT-PLA-ESO-00000-0006).

For the VLT we used the traditional waterfall model (sequence of phases: requirement, analysis, design, implementation, test, etc.) in an "incremental"

Table 1. SE and VLT Milestones.

'91	Start of a SE/QA role in the software (S/W) team	VLT SW Requirements
'92	Basic development environment Documentation standards	VLT SW Specifications
'93	OS and development tools	Start Coding
'94	First VLT Common S/W release S/W Problem Reports	First external distribution
'95	Code management (cmm) Automatic test support (tat)	First Field Test (NTT)
'96	S/W process audited	Wide external distribution
'97	Use & tune	Installation at Paranal
'98	Use & tune	UT1 First Light, First external Instrument (FORS)
'99	Use & tune	UT1 inauguration, UT2 first light
'00	Use & tune	UT2 inauguration, UT3 and UT4 first light

way, i.e., identical sequences were started at different times for the different parts of the project. Every release of each major package repeated a similar sequence of phases.

4. Documentation

The S/W life cycle defines the types of documents to be produced at each stage. The main types we used were: requirement specification, functional specification (high level design), design specification (detailed design), user manual, and test procedure.

The documents followed the numbering system defined at the VLT level and all had the same layout. A document template was provided for each of the major document types.

Whenever possible, documentation was extracted from the code. This was applied mainly to interface specifications (.h files, command tables, etc.) and for "man pages" used both in the design phase as detailed description of each function and later as user documentation. The "man page" is maintained in the file that contains the corresponding code.

5. Development Environment

A standardized development environment is an essential requirement for managing different people, in different places, who are all developing software for subsequent integration. The cornerstones of the development environment have been:

- A standard variable setup obtained using normal Unix mechanisms (.cshrc, .login) and a combination of project-wide, machine-specific, and user-

specific files. Variable definitions are consistent with other standards (makefile, tools, etc.) and help to reinforce their usage.

- The definition of "Software Module" to group sets of files that provide well-defined functionality. Each software module has a unique name that becomes the stem for all named items that are part of the module (files, variables, etc.). The files are organized in a standard directory structure. The software module is the smallest Configuration Item (CI) used to build the software and therefore the basic concept for integration and configuration management.
- A directory structure as a flexible, controlled mechanism for integrating code and distributing it. It defines different areas (development, integration, release) with similar structure but different responsibility. By selecting the area via environment variables (e.g., INTROOT, VLTROOT) and by consistently applying the precedence rules in all other tools (makefile, PATH setting, etc.), the directory structure provides a flexible tool to support parallel work on different stages of the project.
- Naming conventions and coding rules that speed up communication during integration and reduce maintenance cost later on. This also saves time by providing default answers to many trivial questions.
- A controlled set of development tools (OS, GNU tools, etc.) is essential to document and standardize the installation and configuration of the OS(s) used. The same applies to all tools used to develop the software (VLT uses the GNU family, TCL/TK and some commercial products). The goal is to be capable at any moment of building a known configuration.
- Standard makefiles made up by a project-wide set of rules, centrally stored and managed, to which every developer has only to add the module-specific part: in the simplest case, the names of the files to be treated, or a complete makefile if need be. Implementation based on features of GNUmake is strictly correlated with the environment variable set-up and directory and software module conventions.

6. Testing

The most important aspects of testing are that it be automatic (to the extent feasible) and that it be conducted independently of the person whose code is under test. Test software is made of two parts: (1) a standard framework to provide the general features of creating the needed test environment (processes, communication, etc.), sequentially running a set of test programs, scripts, etc., and gathering the output, and (2) after having accounted for what can change from one execution to the other (e.g., time stamps, machine names, etc.), a means of comparing results with a reference execution.

The developer has to provide at least one test executable (from C/C++ programs and shell, tcl, etc. scripts), ordered in a so-called TestList. Because the test tool compares the current output with a previously generated one, test programs can be quite simple: execute an action and print out the result. No interpretation with error-prone "if-then-else" provisions are needed in the test program itself. Minimising test program complexity is important both to limit

the cost and improve reliability. Make the automatic test affordable, and not a nightmare.

Execution of the test is then accomplished via a standard command: "cd <moduleName>/test; tat". The needed environment is created (every time!) to allow the test to be executed with the same initial conditions. All tests are executed and the result is compared with a reference file. PASSED or FAILED is the only test output. PASSED means the result of the current run is identical to the one defined as reference. FAILED means that something was different and requires that an investigation begin. A very compact output is necessary when tests are used for non-regression during release integration.

Despite the "magic" word "automatic", testing is not without its costs. Dedicated resources, both people and computers, are needed for this task in addition to the work of each developer in preparing and using the test software. The size of test software has been of the same order as delivered software.

In addition to the internally developed tool, commercial software products, like Purify, have been used.

7. VLT Common Software Releases

The VLT Common Software provided the building blocks for all applications and has been used by all internal and external development teams. To be effective it has to be readily available and reliable. This has been achieved by treating it like a commercial product rather than as a scientific prototype.

Key aspects have been: identify all components (OS, tools, ESO-developed S/W), maintain a reasonably paced upgrade cycle (6-12 months), document both in man pages and on paper, enforce backward compatibility, and impose systematic non-regression testing before release.

8. Configuration Management

The Configuration Item is the software module, and a system is made up of a list of pairs "moduleName"-"version". Normally one page is enough to specify a build for a system made up of several thousands of files.

One central archive, accessible to all sites and all people, was essential. The archive is the coordination point: if a software unit is in the archive it means that it is *ready* to be used, because it has been *tested* and all files are *consistent*.

The supporting tool (cmm) is a thin layer of procedures on top of an RCS archive. Its basic rules are:

- All files belonging to a module are handled at the same time;
- Only one person at time can modify a module;
- Branches are supported;
- There is only one archive;
- A client-server protocol implements all operations;
- Commands consist of a very simple set.

Figure 1 gives the total number of transactions executed per month over the last three years. "Archiving" accesses are when a new version of a module is created, "copying," when a read-only copy is retrieved. Note the high number

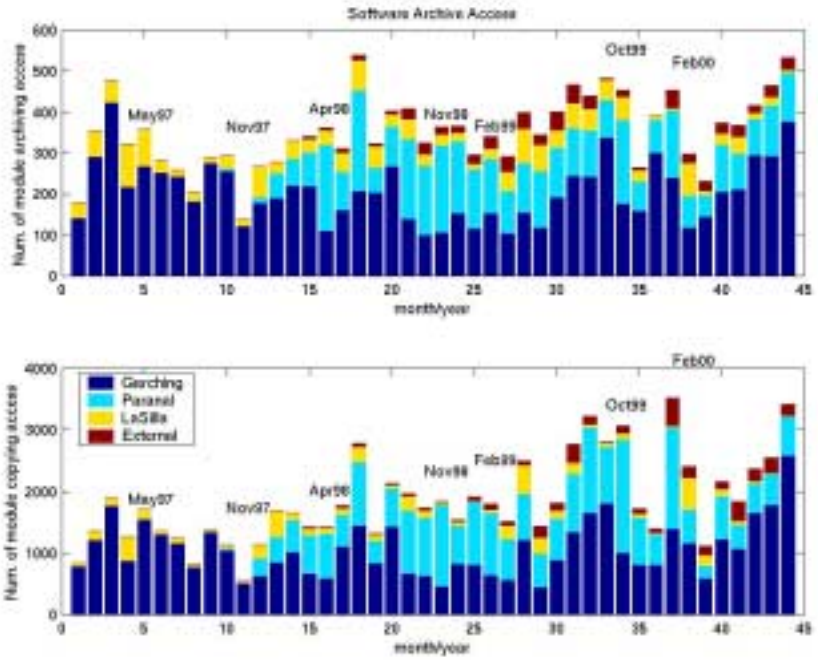

Figure 1. Total CMM Archive accesses by all sites.

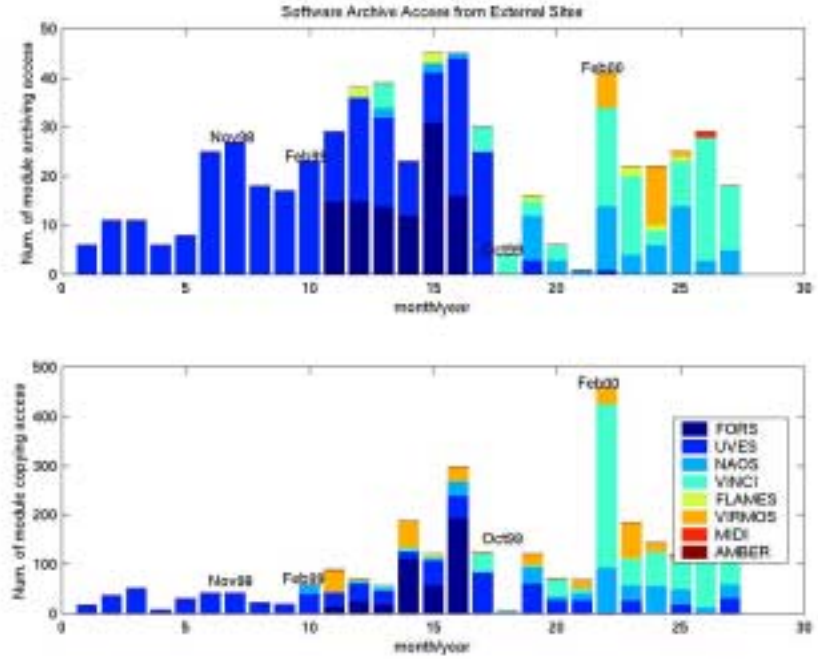

Figure 2. CMM Archive accesses by European Consortia sites.

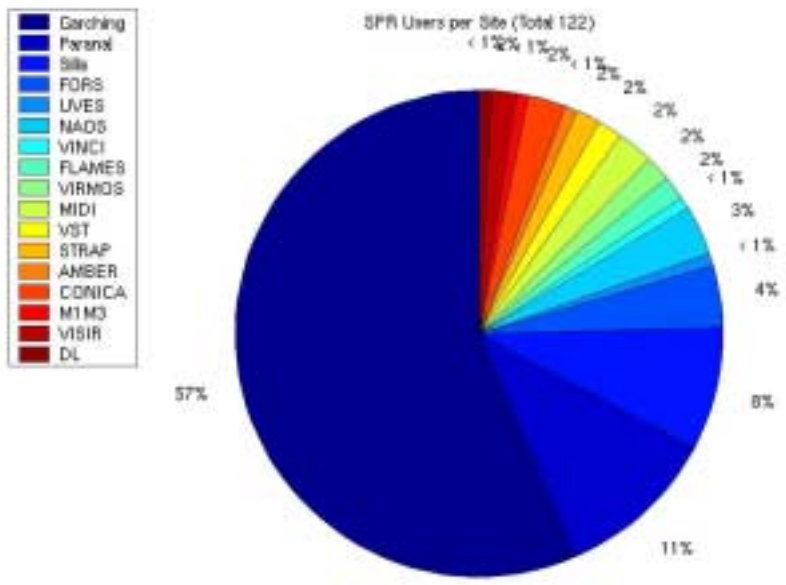

Figure 3. Software Problem Report Users per site.

of read-only copies, especially at the end of the integration process of a release or during first telescope integration and commissioning. This is a sign that regeneration from scratch was the rule. Figure 2 gives the same data but limited to transactions serving non-ESO sites, mainly European institutes that are part of one of the several Instrument Consortia providing the VLT instruments.

9. Software Problem Reports (SPRs)

Tracing bugs and modifications is a necessary complement to Configuration Management. For VLT software, a Web-interfaced central database to submit, query, and modify SPRs has been implemented using the commercial tool Action Remedy. This application supports the following work flow:

- Any user can submit an SPR. Depending on the area, some people are automatically informed.
- Every week or two, the S/W Configuration Control Board discusses all new SPRs and either assigns a responsible person and deadline or rejects it.
- The responsible person produces the fix and after having archived the modified module(s), closes the SPR.
- All status changes are broadcast to all interested parties (a Cc field allows people to follow up).
- Comments can be added by anyone at any time.

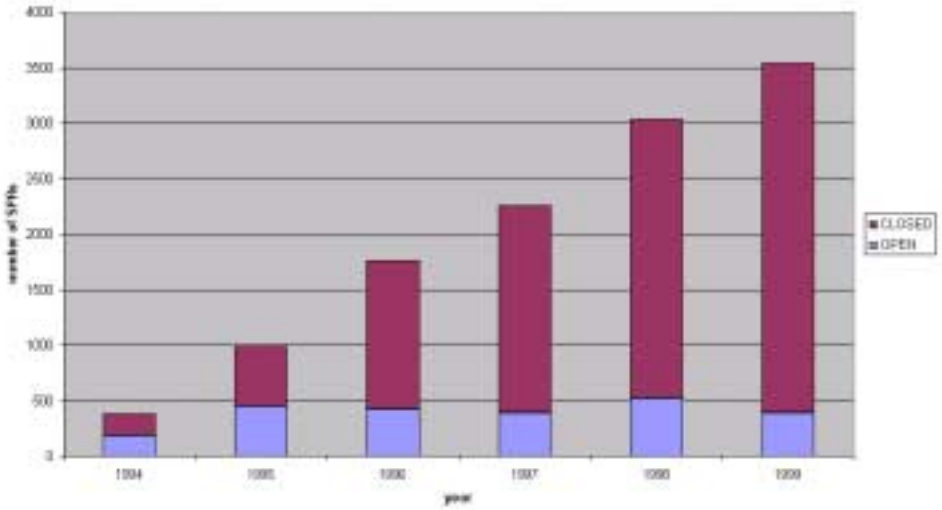

Figure 4. Number of Software Problem Reports per year.

Figure 3 gives the total number of SPR users, i.e., people that submitted a problem or proposed a change. In principle this could be the number of software people that participate in the development and integration of the project. A bit less than 60% were generated by people from the development team (either at the Garching headquarters or on assignment at the Observatory), about 20% by technical support staff based in the Paranal Observatory, and the last 20% by people from Consortia.

Figure 4 give the cumulative number of SPRs each year and shows a continuous increase of the total number due to new software made available each year. Except for the early days, the number of open SPRs at any time was always below a critical level (about 500 SPRs) that was more or less what the team could deal with between releases.

Figure 5 gives the answer to the most important question: Is the system becoming stable, or will a maintenance team bigger than the development team be needed to maintain it? A healthy system should show a peak corresponding to the first integration, followed by a decline. In Figure 5, three different areas are compared over a three-year period: the Common Software had its "glory days" earlier and is declining as a sign of stability, while the Telescope Control Software shows its peak at the integration and commissioning of the first Unit Telescope (1998, UT1 first light). The third group includes other software packages that were at the initial stages of their life cycles in the years shown.

10. User Reaction

In addition to crude numbers, it can also be instructive to report people's reactions. Initial feelings that ranged from indifference to hostility and rejection

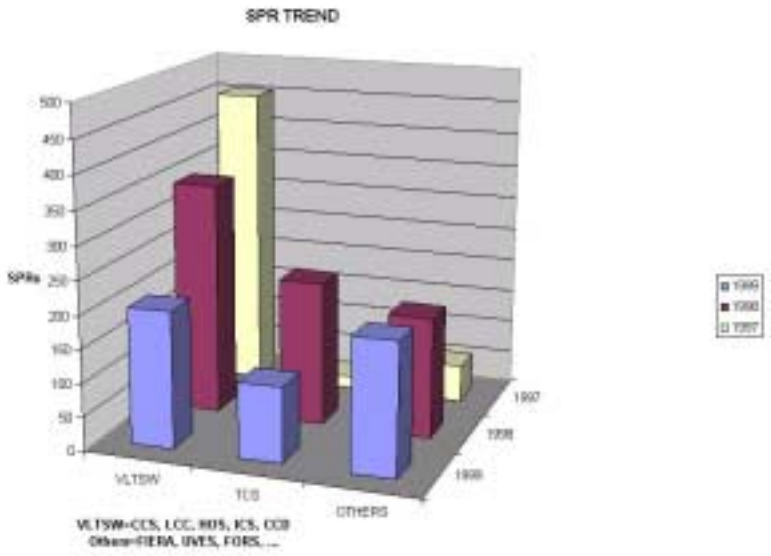

Figure 5. Software Problem Report trends.

have completely faded away a few years later, and the SE has become a strength (and the pride) of the software team. In addition to supporting the work, SE also provided quantitative results, often needed as a way to present work done to upper management.

The established SE system was audited by an external team (in 1996) that found no deficiencies in the approach and in the implementation. Beside this, it has been positively recognized by the integration and commissioning teams of both Paranal and La Silla

A July 2000 survey asked those who had used the VLT Common Software to identify the three best and three worst aspects of the systems. Of a total of 22 responses collected, 15 people mentioned SE practices as one of the best items, while none mentioned SE among the worst aspects. Table 2 contains these reported opinions.

11. The Future

All SE practices described are currently in place and part of the daily routine. Beside keeping the SE tools up to date with the new versions of the underlying free software, there are a few major areas of development. First, the Object Oriented approach is now the rule and analysis and design have to be tuned for that. Unified Modeling Language (UML) and Use Cases are replacing the current practices for the early phases. A second area is use of the Web as the repository for documentation. This tool became widely available in the middle of the VLT S/W project and was not employed as vigorously as it should have been. Third, a stronger "testing" culture will be promoted: more support will be given to developers in terms of both tools and training in writing good test

Table 2. User opinions of VLT Software Engineering.

"Best" Rank	Characteristic
1	Programming standards
1	S/W Dev. Environment + Standards (Makefile,
1	S/W engineering (configuration, SPR, programming
1	S/W engineering, standards
1	The development environment (makefile, file
1	standard for programming, directory structure,
1	std module structure - INTROOT/VLTROOT concept
2	Configuration Management
2	Standard module structures with makefile and
2	Lots of documentation
3	Documentation
3	Test support
3	modular concept
3	cmm, vltMakefile
3	software configuration control and vltMakefile

software. Last but not least, it is time to close the loop with metrics: We have to measure in a quantitative way what we are doing and define numeric goals.

12. Conclusion

Overall, the VLT-SE approach has been a blend of ideas inspired by existing standards and some pragmatic down-to-earth choices. From the implementation point of view, standards have been using commonly available free software (RCS, GNUmake) with minimal "home made" wrappers, allowing an easy and cheap way to serve the quite widespread community that worked on the VLTI.

Standards and practices have been enforced more by means of such tools, than by "police inspections." With the exception of a few key areas in which deviations were simply not permitted, it has been the voluntary behavior of developers that has determined the results.

In summary:

- SE is necessary for a big project and should be used project-wide.
- It comes at a reasonable cost (but not for free).
- It can be accepted (and liked) by developers.
- It can be implemented in gradual steps: it is not necessary to "have it all", but using all that one has, consistently and continuously is essential!