Evolution of the top level control software of astronomical instruments at ESO

Eszter Pozna[*a]
[a]ESO, European Organisation for Astronomical Research in the Southern Hemisphere,Karl-Schwarzschild-Strasse 2, D-85748 Garching bei München,Germany;

ABSTRACT

The Observation Software (OS) is the top level control software of astronomical instruments which is managing the actions during exposures and calibrations carried out at ESO (at various sites VLT, VLTI, La Silla, VISTA). The software framework Base Observation Software Stub (BOSS) provides the foundation of the OS, in use for a decade. BOSS contains 26000 lines of C++ code and covers the functionalities of a simple OS (configuration, synchronization of the subsystems, state alignment, exposure and image file handling). The need for ever increasing precision and speed imposes a consequent increase in complexity on the astronomical instrument control software. Thus makes the OS a critical component in the instrument design. This is reflected by the size of the BOSS applications varying between 0-12000 lines including additional scheduler mechanism, calculation of optical phenomena, online calibrations etc. This article focuses on the progress of OS and BOSS, and their functionality over time.

Keywords: BOSS, observation software, instrument software, control software, exposure control, evolution

1. INTRODUCTION

1.1 Framework Base Observation Software Stub (BOSS)

The Observation Software (OS) of an astronomical instrument is the top level control software that carries out the instructions of astronomers (given as sequential command series) in order to record astronomical images [1][2] .The OS coordinates the actions of the various subsystems and the telescope(s).
The telescope and the constituent parts of astronomical instruments: detectors, groups of mechanical devices, and occasionally components with specific purpose – are controlled by the Observation Software (Figure 1).
The main information flow is starting from the astronomer user reaching the hardware elements via workstation processes and local control units. The commands of the astronomers must be well defined by various tools (P2PP, BOB) before the observation can start. Although these are executed sequentially the OS must be ready to receive simultaneous interactions from GUI or command line as well as catching events from the subsystems. All modes including the observation modes and engineering modes (frequent or non-frequent) must be pre-analyzed and pre-configured for an on-site OS.
BOSS provides a layer of functionality common to the OS of ESO instruments. BOSS has 12 years history and up to now there are more than 30 applications in operation at the VLT, VLTI, La Silla and VISTA sites and in various laboratories. BOSS supports instruments with general structure including multi level instruments (controlled by Super OS). A super OS (SOS) is an OS controlling instrument OS as subsystem besides the traditional subsystems.
The astronomer user does not need to have an insight to the OS or BOSS functionalities, and the developer and the software operator of the OS applications rarely need to look into the abyss of BOSS. Nevertheless, in view of new incoming requirements and also possible development of a next generation of BOSS, this article aims to review the progress of BOSS, its functionalities and also its applications. As a result this analysis should help future planning, and it could serve also as a pre-investigation of similar system to be prepared for the ELT.

[*]epozna@eso.org; phone 49-89-3200-6441;

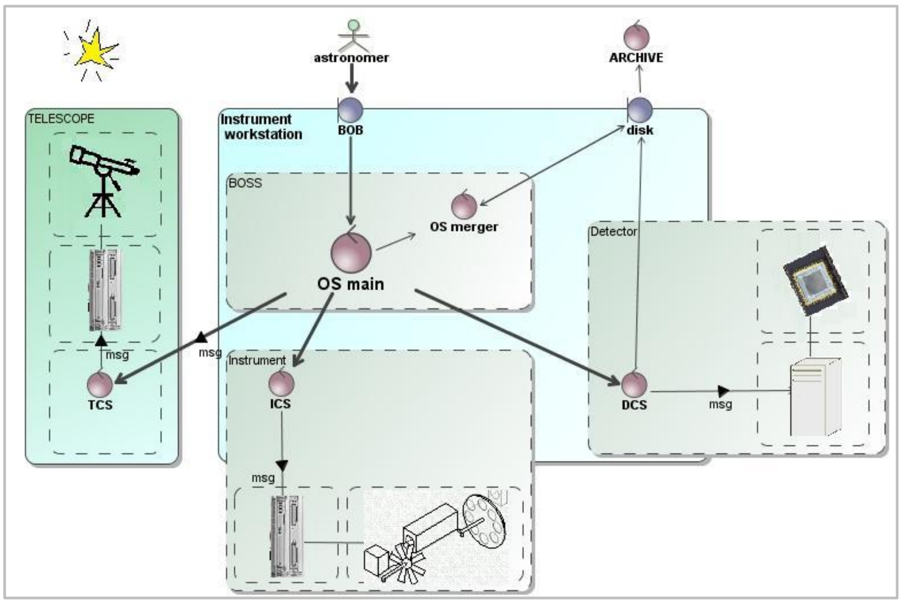

Figure 1. Astronomical instrument controlled by OS.

Herein the article searches for hidden information behind the progress lines: problematic functionalities, difficulties during maintenance, time scale of such project and its applications. The investigation should help to avoid repeating mistakes, and help in the time estimation needed for such projects. (Some aspects might be also helpful for frameworks in general).

1.2 BOSS and software evolution

'Software evolution is the term used in software engineering to refer to the process of developing software initially, then repeatedly updating it for various reasons'[1].

Using Lehman's classification, BOSS falls under the E-type programs. 'E type programs are embedded in the real world and become part of it, thereby changing it. This leads to a feedback system, where the program and its environment evolve in concert [3].' For proprietary software –such as BOSS- Lehman's identified a set of behaviors: Continuing Change, Increasing Complexity, Large Program Evolution, Invariant Work-Rate, Conservation of Familiarity, Continuing Growth, Declining Quality, Feedback System.
A newer model of software evolution called the staged model [4] divides the software life into five distinct stages: Initial development, Evolution, Servicing, Phase-out, and Close-down[1]. Although some concepts in the literature imply commercial environments, following up the progress of BOSS and comparing it to the mentioned stages these laws may help to identify necessary actions before the predicted loss of quality, i.e. helping to preserve its value.

1.3 BOSS and maintenance

'According to several sources, and perhaps counter to intuition, the maintenance of software comprises from 50% to 90% of the overall lifecycle costs (Pigoski, 1997; Lientz & Swanson, 1980)'[2].

Maintenance consists of four parts: '*Corrective* maintenance deals with fixing bugs in the code. *Adaptive* maintenance deals with adapting the software to new environments. *Perfective* maintenance deals with updating the software according to changes in user requirements. Finally, *preventive* maintenance deals with updating documentation and making the software more maintainable. Corrective maintenance is 'traditional maintenance' while the other types are considered as 'software evolution.[6]'

[1] http://en.wikipedia.org/wiki/Software_evolution
[2] http://blogs.msdn.com/b/karchworld_identity/archive/2011/04/01/lehman-s-laws-of-software-evolution-and-the-staged-model.aspx

BOSS is following all the maintenance directives: Over the years BOSS had to adjust to new platforms SUN, HP, LINUX; new version of C++ compilers; new and changing subsystems; requirements from operators, scientist and application developers. Memory test tools[1] are in use. Modular change, bug fixing or code documentation has been also improved, although mostly in connection with user requests. Hence conscious preventive maintenance could be still improved.

2. BOSS AND ITS PREDECESSORS

'Every successful large system is a redesign of a somewhat smaller working system'. Therefore 'whenever possible, design and programming should be based on previous work.' [7]

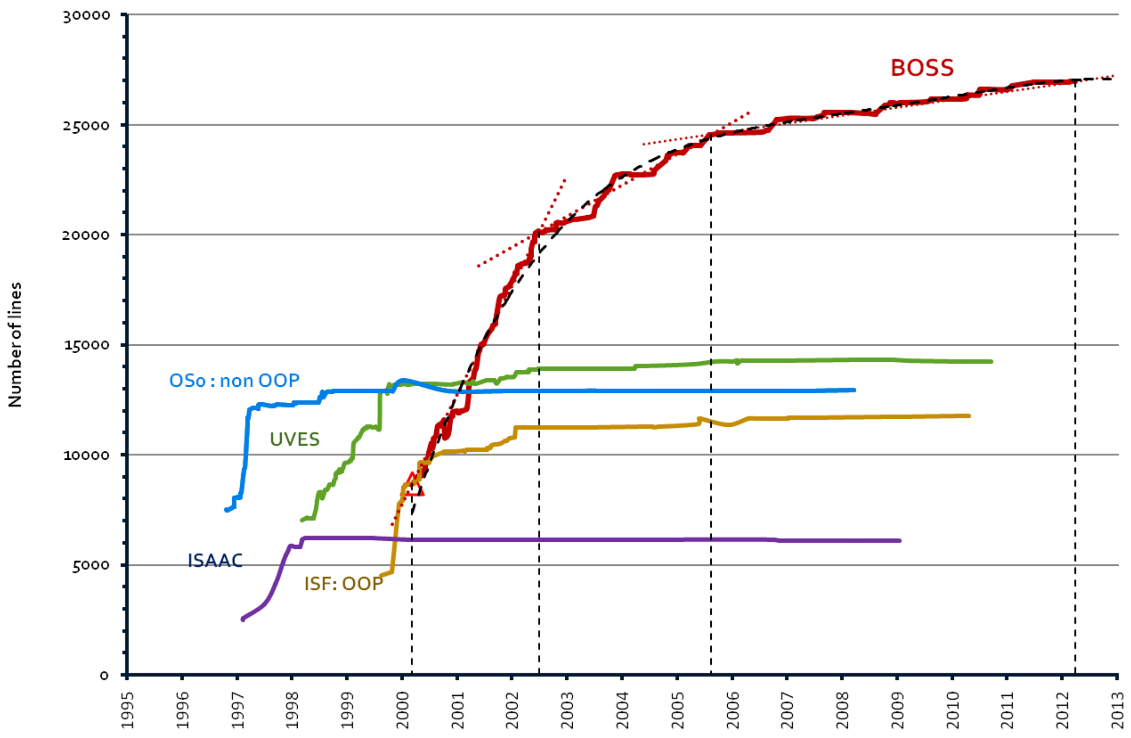

Figure 2. Progress of BOSS and other instruments OS before BOSS[2].

2.1 Progress of OS before BOSS

Before BOSS a few OS have been developed individually. Four of them shown on Figure 2 are: OS0, ISAAC, UVES, ISF[8][9]. These software packages have been based on previous experience and took 2-3 years to develop. The traditional program of OS0 was already configurable, and been in use for 2 instrument applications. The later developed Object Oriented design of ISF is supporting three OS in a combined system (NACO [8][9]).

2.2 Progress of BOSS

BOSS is well following the principle cited at the beginning of chapter 2: it has been started on the bases of existing software (ISF), also ideas from OS0 have been adopted during the early stage of development. Nevertheless BOSS had a long way to go, while releasing earlier constraints, and adjusting to new requirements up to know.

[1] Valgrind recently, Rational Purify earlier.

[2] Number of lines of code is generated using David A. Wheeler's 'SLOCCount.

The shape of the curve of the progress of BOSS on Figure 2 suggests mature software that it is reaching its limit (despite of the incoming new requirements. This curve shows a smooth development which is approximated by a 4th order polynomial (see broken line). The curve however can be approximated also by three well fitted straight lines (see dotted lines), which are dividing the progress line into three phases. It is apparent that these phases are not corresponding to the yearly or bi-yearly software release, but integrate several releases. In each consecutive phase the rate of progress (in term of lines of code) is decreasing. Let us associate these phases with various development/maintenance stages:

Table 1 Development phases and their properties (according to the author's impression).

phases	*Main goals*	*Functionalities added*	*Other aspects*
Phase-1	Specification of generic features; Development of basic functionalities; Standardisation; First test in operation platform; Adjustment to new releases; Bug fixing;	Basic application developer requirements: Standardization of configuration keywords; Additional detector interfaces (CCDs); Generic exposure handling (incl database display); SOS functionalities; Hierarchical keyword handling; Instrument modes; Basic Operational requirements: Further standardization (configuration, exposure life cycle display); auto startup; further clarification of functionalities/ commands; default behavior (header keywords, instrument modes);	Highest rate of code/year; Transition of base code to generic framework; Direct communication with first users (application developer, operator). *Gaining experience*; 4 applications in ~2.5 years.

Phase-2	New and improved functionalities;
	Improved user interface;
	User support;
	Adjustment to new releases;
	Bug fixing;
		Refined requirements:
		Support functions; Additional template functions; Image merging directives (binary table handling, append); Improved performance of merging (merge queue); Improved error handling; Further refining existing commands ; SOS functionalities;
		New requirements: VLTI interface; Independent operation of subsystems; New standard file naming; Performance measurement option; Optimized exposure sequence;
			Medium rate of code/year.
			Evolve to new requirements;
			Architectural changes;
			More applications;
			Direct and indirect communication with users.
			8 applications in ~3 years.

Phase-3	New functionalities
	User support;
	Adjustment to new releases;
	Bug fixing;
		Refined requirements:
		Advanced options for optimized exposure; further refinement of SOS features (e.g. repeated SOS exposure);
		New requirements: Multi-file exposures; image handling for extensions; Parallel command handling; multi telescope system; new NGC subsystems; support of granularity in application.
			Lowest rate of code/year.
			Less bug reports.
			Less user support needed.
			Mainly indirect communication with users.
			18 applications in 7 years.
			(2 application by author.)
			Updates of applications in operation.

During the first phase (approx ~2.5 years) rapid development of new or standardized functionalities took place. The functionalities developed in BOSS at this time were still the basic functionalities needed for the generic framework (see Table 1 for details). Good communication with the first users of BOSS, i.e. developers of the instrument OS applications (for VINCI, GIRAFFE, FLAMES SOS) has helped a great deal to clarify the basic requirements and establish a working framework within these years. Since the core of the software (ISF) has not yet been put into operation the adjustment to the operational environment took place during this time too. (ISF code took separate path in this regards, as it is indicated by its curve). The numerous reports appearing from operation site (Paranal) called to expend more effort to utilize BOSS. By the end of this first phase BOSS passed the basic requirements of the application-developer user and operator user. This marks also –the hidden milestone - the first working version of BOSS as a generic framework.

In the second phase the development continued with a reduced rate. At this time new application started using BOSS. Indirect communications (via problem report system and contact persons) started to take place with application developers; on the other side scientist users started to place their new requirements. During this time numerous new features were still developed as well as existing ones extended. In this stage a more sophisticated synchronization mechanism had been introduced to be able to cope with the new requirements. This has naturally increased the complexity of exposure handling. (See also 2008 SPIE paper on BOSS [2]).

After this period the rate of code/year drops again and BOSS enters the longest 3rd phase: i.e. period of evolution. BOSS in this phase adapts to the necessities of new numerous applications, as well as changes or improves existing features according to user requirements.

Note that these three phases do not correspond to the specification of the pre-alpha, alpha and beta stages[1] (which however might be applicable for the individual times between releases). The phases of BOSS might be classified then as follows: phase-1: transition to generic framework; phase-2: evolve to new requirements, while updating architecture; phase-3: pure evolution. This classification (Table 1) shows more correspondence to that of Bennett's [4].

3. BOSS AND ITS FUNCTIONALITIES

The size of the individual functionalities and their progress in time hides further information about the BOSS software package.

3.1 Main functionalities

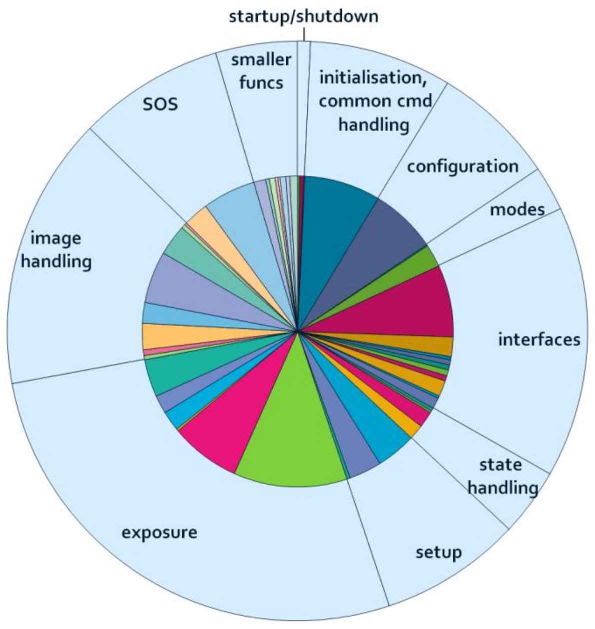

Figure 3. An eye on BOSS shows the basic functionalities and its division into sub-functionalities.

Figure 3 shows the main functionalities of BOSS and their ratio. Figure 4 depicts the main classes representing these functionalities. The inner circle on Figure 3 illustrates the ratio of the constructing classes, some of which are further detailed on Figure 5. The classification here is made to emphasize certain characteristics of BOSS avoiding repetition of the sub-functionalities. Note that certain sub-functionalities have multiple roles therefore alternative categorizations[2] are possible. Some of the sub-functions could be also further divided for more exact picture; however this is outside the scope of this paper[3]. Functionalities with multi-role nature suggest that the software architecture could be adjusted to better reflect the responsibilities. The modularity could be improved at places and also delegating common task into

[1] In alpha stage the first version of software system is still lacking some features which are then developed as part of the initial phase of maintenance. The alpha version is ready for its initial test inhouse which is followed by beta test performed by users at their facilities under normal operating conditions. (PC Magazine: http://www.pcmag.com)

[2] E.g. the OS interface belongs to 'SOS' functionality; the internal archiver interface -via which the secondary merger process of the OS is accessed- belongs to 'image handling' ; hence the 'interfaces' collect the subsystem interfaces of a standard OS.

[3] E.g. the common detector interface contains message handling, detector specific configuration, and synchronization as private nested-class.

lower level libraries could help to decrease complexity. Note that a relatively new DELEG class allows better granularity in the applications (see Figure 4).

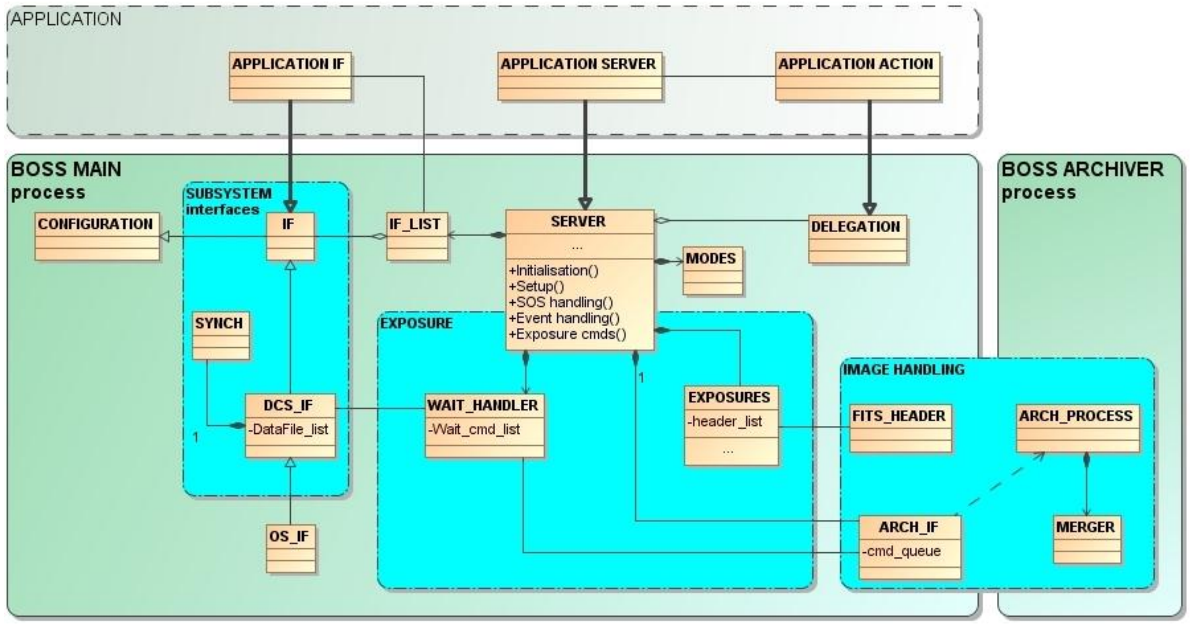

Figure 4 Main element of the class design of BOSS showing also the connection point to its applications.

3.2 The code size of functionalities and their components

Taking a look at the size of functionalities as of today, perhaps not surprisingly (for the developer/author) the EXPOSURE control stands out, that is also the most relevant feature. Second to the EXPOSURE functionality, the IMAGE HANDLING takes places followed by the SETUP and SOS functionalities and collection of INTERFACES.

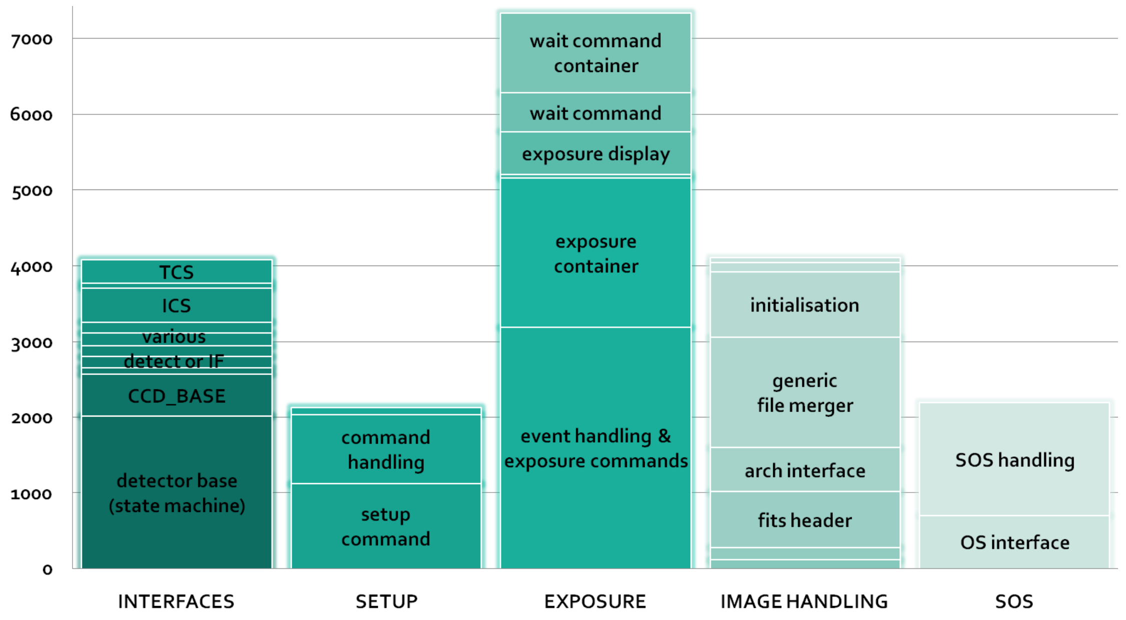

Figure 5. Details of the main functionalities of BOSS.

The EXPOSURE control incorporates the action necessary to carry out single or simultaneous exposures. To do this it monitors the status of the detector and executes various actions accordingly [2]. Exposures are started/interrupted by users via commands. The wait command (giving green light for various user actions) is tightly connected to the exposure status. The concurrent exposures are stored and displayed in database. Similarly there is a dynamic holder for the pending wait commands. The density of these functionalities is reflected by the sizes of the belonging classes displayed on Figure 5. The figure shows that *event handling* stands out with its size which hints that the various action could have been implemented separately increasing the granularity of the system.

The IMAGE HANDLING incorporates the process of assembling the final image file. The main process (Figure 1) makes sure that all information has been collected in form of keywords, binary tables and datafiles. The secondary merger process is responsible for constructing the partial files together according to the directives produced by the main process.

Various dynamic lists handling is incorporated in today's BOSS, which assists the features of multi-image, simultaneous exposures, background image handling. Naturally these features have shaped the architecture of BOSS contributing to increased complexity. The question arose now, whether the complexity is still increasing. The progress lines of the individual functionalities might give answer to this.

3.3 The progress of functionalities and hidden characteristics

'A program undergoes continuing change or becomes progressively less useful. As an evolving program is continuously changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain it or reduce it. [4]'

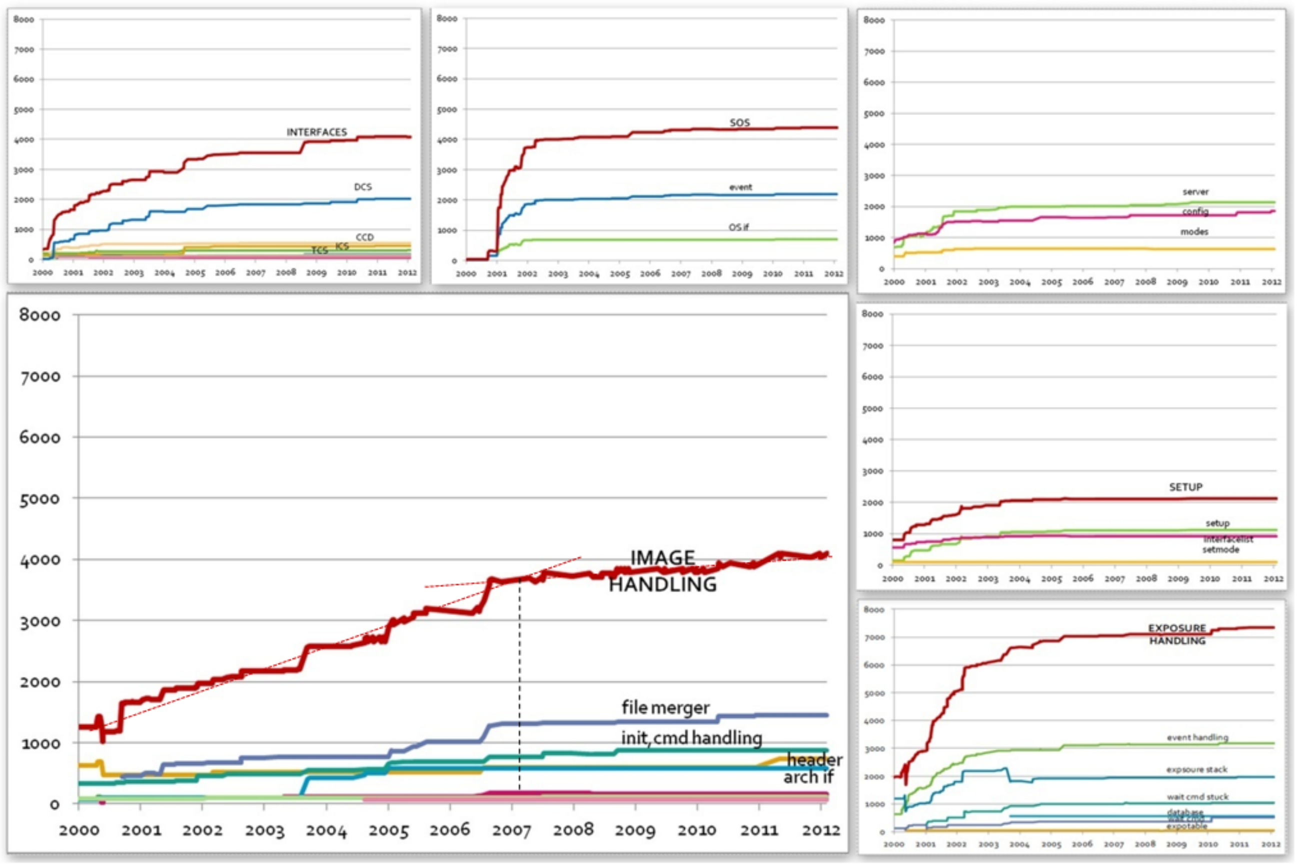

Figure 6. Progress of the functionalities of BOSS. Bottom left: image handling; Clockwise from top left: interfaces; SOS; base functionalities; setup; exposure handling.

Plotting the progress of the separate files/classes (Figure 6) reveals the information about which functionalities are still progressing and which has reached maintenance/mature phase and when. Figure 6 plots the development of the main functionalities (Figure 3) as well as the sub-functionalities illustrated on Figure 5.Within 2-4 years most of the small and average sized functionalities (i.e. SOS, configuration, setup, configuration and initialization) have reached stable state. This corresponds to the global assessment in section 2.2. Interestingly, the biggest functionality –exposure handling- has also reached a steady phase after 6 years, i.e. within the end of global phase-2 (see also Table 1). According to this the time needed to reach maintenance phase of BOSS functionality is proportional to their current size. However the interfaces and image handling are changing continuously up to now...

Image handling stands out with its continuous grows. Looking into the change request tells the causes: one requirement brings on a cascade of related requirements over time. E.g. multi-image systems set off the need for manipulation inside the files (extension header handling; extension sorting algorithms; new directives for the merger; new template functions for user). Another example: the appearance of multi-ICS and multi-TCS system brought on the need for additional configuration and header keywords handling; etc. In these cases analysis helps to go ahead of user requirements and implement the functionalities in advance.

Despite the fact that new interfaces has been added (e.g. for NGC), they are –as child classes- are small, and have little impact on the total size of the interfaces. Figure 5 shows, that it is actually the detector base that changes the most. This class includes functionalities related to the image handling (and exposure handling) which explain this history.

While there are also requirements regarding the exposure handling, they are improving existing functionalities such as the ‘optimised parallel exposure sequence[1]’ (see details in the 2008 SPIE paper[2]). Similarly SOS has been gradually improving in the past years (e.g. adjusted to handle functionalities that been in use for regular OS, e.g. repeating exposures).

Although these improvements sometime sounds major can be done with little effort which indicates an appropriate design.

4. THE APPLICATIONS OF BOSS

4.1 Progress of OS of various instruments based on BOSS

Applications naturally influence the development of BOSS. Figure 7 plots their progress.

Four application stands out with rapid development of larger amount of code. While these should indicate additional functionalities, in case of MIDI also unnecessary function overloads[2] been detected pointing out lack of communication and also improvement option for BOSS to introduce more restriction for user overloads. CRIRES[1][10][11] was developed very fast due to prompt requirement clarifications followed by implementation and (special availability of) immediate verification[3],[4]. The final part of CRIRES where the curve suddenly reaches its limit indicates the final adjustments testing the additional event queue handling together with all extra positioning functionalities. In the other cases it is interesting to notice the several jumps in the code size after the various code enter into maintenance phase. These without doubt indicate new requirement arriving after the system been placed into operation, as part of the maintenance phase.

4.2 Number and type of controlled elements vs the size of OS application

The aim of software evolution would be implementing (and revalidate) the possible major changes to the system without being able a priori to predict how user requirements will evolve [3].
Let us investigate however how some property of the system may influence the amount of code in the applications. One could then foresee where more efforts would need to be invested for future or currently developed instrument OS. May also see for which cases BOSS should provide more support. E.g. looking into the details of the applications detection of generic properties could help preventing to reinvent the wheel at the next instrument.

[1] E.g. parallel exposure handling was extended for being able to catch the end of the first completed exposure allowing further optimization.
[2] This could have been prevented by requesting easy plug-in functions, which were provided along the way using template pattern.
[3] Also before the development of the plotted CRIRES OS the instrument was already in operation (providing the core functionalities).
[4] Developed according to the waterfall model: ‘a tried and tested problem solving mechanism. Documentation is an integral part of the process. [6]’.

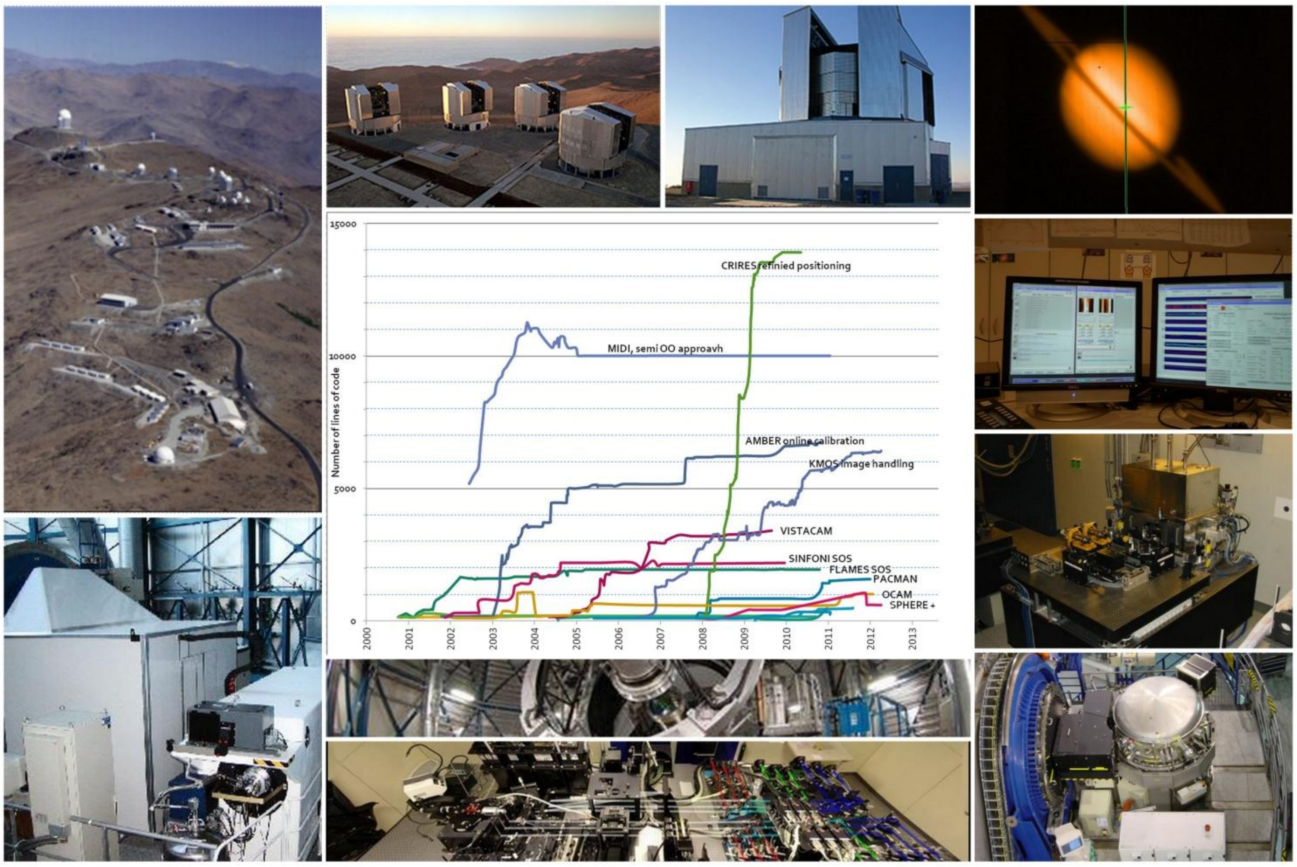

Figure 7. In the middle: Progress of various instruments OS based on BOSS. From top left in clock wise: locations: La Silla, Paranal,Vista; Applications: CRIRES positioning, PRIMA observation; Instruments MIDI, CRIRES, VIRCAM(top), AMBER(bottom), FLAMES.

Figure 8 shows the type and number of subsystems controlled by various instrument OS. Conclusion: infrared instrument[1] with image related functionalities/positioning shows peaks. SOS systems also require more than average work. On the other side - perhaps coincidentally- instruments having the composition of more detectors, one telescope, one ICS as well as the ones with two subsystems appear to be easy cases, adding only a few extras to the provided BOSS functionalities.

Might be interesting to see also how the experience of the users has influence on the OS applications. Some functionality may as well be distributed several ways in the instrument software hierarchy. Other properties of instruments (e.g. number of devices) could be checked how they influence the requirements. These points however are outside the scope of this paper).

[1] In more details: The most generic composition (as shown by Figure 8.) is a 3 unit system containing one detector, one ICS and Telescope system. Six instruments belong here: at VLTI: VINCI; in La Silla: SUSI and HARPS, at VISTA: VIRCAM; in Paranal: VISIR, HAWKI, and KMOS; Apart from Harps all with IRACE detector. Amongst these VIRCAM, KMOS (VISIR) are showing peaks in their code. Instruments having the composition of more detectors, 1 telescope, 1 ICS (in Paranal: TC ; VLTI: MIDI; La Silla EMMI, GROND, XSHOOTER, OCAM) however have all small OS applications regardless which at which stage of the BOSS development cycle they have been created or the type of detectors. Nevertheless SOS are always larger: FLAMES, SINFONI.

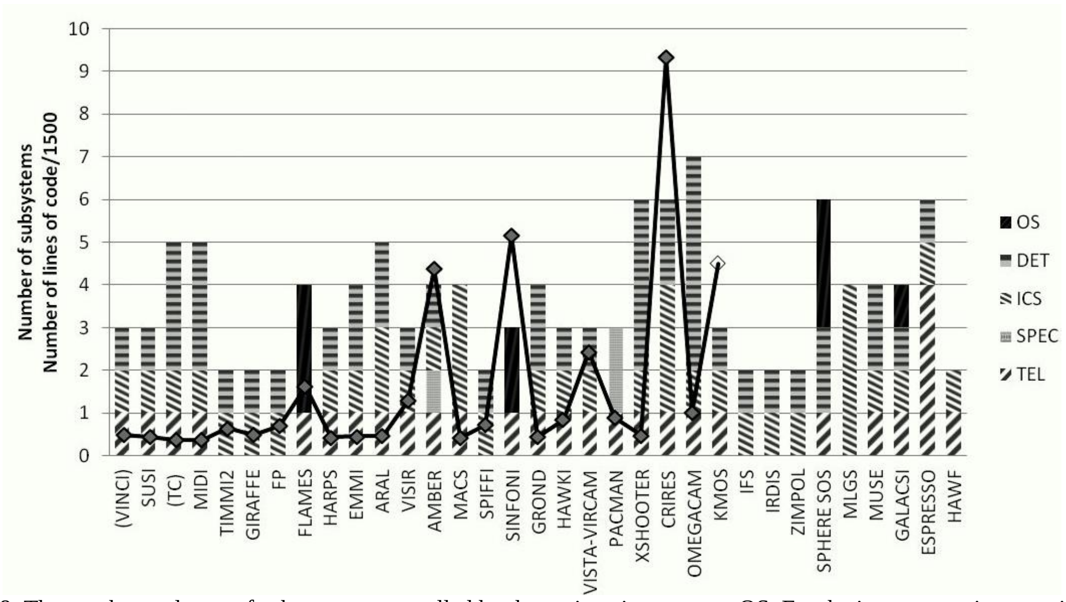

Figure 8. The number and type of subsystems controlled by the various instruments OS. For the instruments in operation the number of lines of code of the OS is also shown superimposed on this figure.

5. CONCLUSION AND FUTURE CONSIDERATIONS

This assessment of the progress of BOSS helps to understand hidden areas of weakness and strength, which in turn can drive the development in the right direction. The progress plots helps to predict future needs, as well as time estimation for BOSS like project and its applications.

5.1 Strength and weaknesses

BOSS in the past few years has proved itself to be easily adjustable to the ever-changing user requirements and operating environment. According to Bennett “for software to be easily evolved, it has to have an appropriate architecture”[4].

Nevertheless, the complexity of an evolving system grows therefore effort should be invested to make sure quality. Possible improvement areas detected for BOSS was to increase its modularity (e.g. using factory method for adding functionalities which may only be required for few instruments) and restrict public access to some of its functions.

5.2 Time evaluation

Three relevant phases of development with different rate of code/year have been identified independently from the yearly releases. Globally it took BOSS 5 years to enter the stage of evolution. However for the case of individual functionalities this varies between 3 to 7 years.

The different phases have been associated with different aspects highlighting the relevance of direct communication with users at the earlier stage.

5.3 Concurrent requirements

The increasing datarate allowed by new detectors rings a bell for further optimization of the exposure control (e.g. MATISSE estimates 100 MByte/s peak performance reaching image files size 2GB expecting simultaneously arriving data with lower rate). Buffered transferring of data from LLCU to IWS is under consideration which would then need to be supported at higher level. The change may not only impact BOSS but may also introduce new operation tactics. The impact on BOSS will result in a change of its internal state machine, also current limitation (given by database display) for exposure queue will have to be reviewed. The handling of large data will require further performance analysis.

As predicted in 2010[1] a generic implementation is now needed to embed RMNREC data in the VLTI observations (for instruments MIDI, MATISSE, GRAVITY in addition to AMBER and PACMAN where access to RMNREC is already available).
Yet to be seen whether generic support to optical phenomena, internal loops, scheduler system (as applied in CRIRES) is of generic interest (as predicted also in year 2010). VLTI systems could be the customer for such functionalities.

5.4 Open Questions

According to C. Diggins 'recycling code', i.e. reusing it in some other project and porting it to 'new operating systems and platforms is almost always inevitable in a successful product' [16] . The future of BOSS in this regards is not yet clear, however according to this study it is time for BOSS to apply a more powerful preventive restructuring and code optimization.

REFERENCES

[1] Pozna, E., Smette A., Schmutzer, R., Roberto, A., Thanh, P. D., Santin P., "New direction in the development of the observation software framework (BOSS)", Proc. SPIE Vol. 7740, 77401Y (2010).

[2] Pozna, E., Zins, G., Santin, P., Beard S., "A common framework for the observation software of astronomical instruments at ESO", Proc. SPIE Vol. 7019, 70190Q (2008).

[3] Lehman, M. M. (1980). "On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle". *Journal of Systems and Software* **1**: 213–221. http://en.wikipedia.org/wiki/Lehman%27s_laws_of_software_evolution

[4] Bennett, K. H., Rajlich, V. T., Wilde, N. "Software Evolution and the Staged Model of the Software Lifecicle", Advances in Computers, Toronto: Academic Press (2002) http://www.cs.wayne.edu/~severe/publications/Bennett.AdvComp.2002.Software.Evolution.pdf

[5] Fred Brooks, The Mythical Man-Month. Addison-Wesley, 1975 & 1995. ISBN 0-201-00650-2 & ISBN 0-201-83595-9.

[6] Prof. Stafford, Erdil K., Finn E., Keating K., Meattle J., Park S., Yoon D., "Software Maintenance As Part of the Software Life Cycle", Comp180: Software Engineering, 2003 http://hepguru.com/maintenance/Final_121603_v6.pdf

[7] Stroustrup B.,"C++ programming language", third edition AT & T Labs, Murray Hill, New Jersey,1997

[8] Zins, G., Lacombe, F., Knudstrup, J., Mouillet, D., Rabaud, D., Charton, J., Marteau, S., Rondeaux, O., Lefort, B., "NAOS Computer Aided Control: an Optimized and Astronomer-Oriented Way of Controlling Large Adaptive Optics Systems", ADASS, Vol. 216 (2000).

[9] Zins, G., Lacombe, F., Knudstrup, J., Mouillet, D., Rabaud, D., Marteau, S., Rousset-Riviere, L., Rondeaux, O., Charton, J., Lefort, B., Rousset, G., Hubin, N. N, "NAOS computer-aided control: an optimized and astronomer-oriented way of controlling large adaptive optics systems", Proc. SPIE Vol. 4009, pp.395-401 (2000).

[10] Käufl, H. U., Amico, P., Ballester, P., Bendek Selman, E. A., Bristow, P., Casali, M., Delabre, B., Dobrzycka, D., Dorn, R. J., Esteves, R., Finger, G., Gillet, G., Gojak, D., Hilker, M., Jolley, P., Jung, Y., Kerber, F., Klein, B., Lizon, J., Paufique, J., Pirard, J., Pozna, E., Sana, H., Sanzana, L., Schmutzer, R., Seifahrt, A., Siebenmorgen, R., Smette, A., Stegmeier, J., Tacconi-Garman, L. E., Uttenthaler, S., Valenti, E., Weilenmann, U., Wolff, B., "CRIRES: commissioning and first science results", Proc. SPIE Vol. 7014, 70140W (2008).

[11] Pozna, E., Smette, A., Schmutzer, R., "Task shyncronisation in the Observation Control Software for the ESO-VLT CRIRES instrument", ICALEPCS2009, Kobe, Japan, THP087 (2009). http://accelconf.web.cern.ch/accelconf/icalepcs2009/papers/thp087.pdf

[12] Abuter R., Sahlmann J., Pozna E., "PACMAN: PRIMA Astrometric Instrument Software", Proc. SPIE, Vol. 7734, 77340W (2010).

[13] Abuter, R., Popovic, D., Pozna, E., Sahlmann, J., Eisenhauer, F., "The VLTI real-time reflective memory data streaming and recording system", Proc. SPIE Vol. 7013, 70134A (2008).

[14] Le Bouquin, J.-B., Abuter, R., Haguenauer, P., Bauvir, B., Popovic, D., Pozna, E. : "Post-processing the VLTI fringe-tracking data: first measurements of stars", Astronomy and Astrophysics, Vol. 493, Issue 2, pp.747-752 (2009). http://adsabs.harvard.edu/abs/2009A&A...493..747L

[15] Santin P., Di Marcantonio, P., Popovic D., Pozna E. , "ESO-VLT Instrumentation the Control Software for the FLAMES-UVES-GIRAFFE observing facility" , Mem S.A.It. Vol. 73, 23 (2002)

[16] Diggins C. ," Stages of Software Development", Artima Developer (2007) http://www.artima.com/weblogs/viewpost.jsp?thread=194223