



# **Memo 50**

## **Software development for the Square Kilometre Array**

T.J. Cornwell - B.E. Glendenning  
06/04

---

---

**Abstract:** *Software development costs for the Square Kilometre Array are likely to be very large – in the range as 1000-2000 person-years. This level of software effort is unprecedented in radio astronomy. Consequently the risk associated with software development is very large. This is common to many large science projects and so we can learn from such projects how to best mitigate against the risk. We present a shopping list of suggestions drawn from the experience in other projects.*

## 1. Introduction

Radio telescopes have grown larger both in ambition and in necessary complexity. As a result, software has come to lie at the heart of radio telescopes, bringing computational power and flexibility. While it once was true that a small group led by a single expert could assemble all the software required, that era is long gone. For example, the ALMA project has software development costs of over 400 person-years. A recent estimate for SKA software development derived by scaling the ALMA budget yields numbers of well over 1000 person-years (Kemball and Cornwell, 2004). If the construction of the software is to be done in 5-10 years, then a team consisting of perhaps 100 or 200 people is needed. The team must have a broad range of skills and knowledge – from programming methodologies to arcane details of interferometry, from parallel computing to user interface design, from code management to people management. It is highly unlikely that the team can reside in one location and geographical separation will be one of the challenges.

To only add to the challenges, it seems probable that development of SKA will occur at a time of large change in the computing industry. Paradigms for computing are changing constantly and the rate of change will probably only increase. Software methodologies continue to evolve in response to sometimes-painful experience: current (or slightly expired) buzzwords are Extreme Programming and Aspect-Oriented Programming, both of which aim at improving programmer productivity. On the hardware side, there is a push towards Grid computing. Finally, it is clear that computing in general and astronomy in particular are moving towards more service-orientation, as in the numerous national Virtual Observatories.

---

*1. The National Radio Astronomy Observatory is operated by Associated Universities, Inc., under cooperative agreement with the National Science Foundation.*

How then can all this activity be managed? The possible failure modes are budget overrun, broken timelines, incomplete software, and drawn-out commissioning, to name but a few. One logical possibility is to radically de-scope – reduce the functionality expected of the telescope. This is often advocated, at least implicitly in statements that SKA software development should not, *a priori*, cost more than X person-years where X is a number drawn from personal experience. We reject this line of argument as being defeatist and unnecessary. Instead we advocate the usual approach to a fresh challenge – learning from our past experience and borrowing best practices, both from the radio astronomy community and from other disciplines.

This memo is written as a collection of suggestions, a good fraction of which are backed up with references. In addition to our own experience and those of people at NRAO and in ALMA that we talk to a lot, we've found the following documents very helpful:

- LCK: An excellent paper by Lewis *et al.* (2002) on “*Do larger telescopes need larger hardware?*”
- PS: Presentations from the NSF-funded series of workshops on Project Science. See <http://131.215.125.172/>
- BPS: the presentation in PS by Kent Blackburn on the LIGO data analysis system.
- RL: A paper by Robert Lupton and others (2001) on the SDSS imaging pipelines. This is notable for Robert's view on the challenges of software development for astronomy. See <http://arxiv.org/pdf/astro-ph/0101420>
- CSNAP: A presentation by Bill Carithers on the SNAP software. See [http://snap.lbl.gov/review/2003\\_11/Carithers\\_Computing.ppt](http://snap.lbl.gov/review/2003_11/Carithers_Computing.ppt)
- LCG: The site for the Large Hadron Collider Grid Computing Project. See <http://lcg.web.cern.ch/LCG/Overview.htm>
- Jim Gray's web site (<http://research.microsoft.com/~Gray>) provides highly insightful commentary on many aspects of computing, but most particularly the economics of large data and distributed computing.
- The entire series of ADASS proceedings at <http://www.adass.org/>. These contain much interesting and relevant experience.

We assume that the SKA will have a strong foundation in project management and so we will mostly confine our suggestions to computing (but see PS for real experience by managers of large science projects, and LCG for an example of a well constructed oversight and review structure).

Many of the things that make working in current projects hard and unappealing are not specific to computing. Examples are:

- Inoperative management and/or management structures,
- Quantity rather than quality of oversight,
- Cultural differences at the team and organizational levels,
- Politicalization of technical designs,
- Drastically insufficient funding.

We hope that none of these will occur in the happy world that is SKA.

## 2. Things not to do

**DON'T Omit software and data analysis** – An example drawn from another field – the LIGO proposal did not specify how the data from the (gravitational wave) interferometers were to be analyzed. This had to be done after the fact.

**DON'T Write unrealistic requirements** – Committees are, of course, prone to add more and more features as meetings continue. Setting priorities (*e.g.* vital, desired, useful) helps but really just amounts to a culling since it is common experience that only the highest rated priorities are actually implemented. Iterating requirements writing with even first-order cost estimation would help.

**DON'T Underestimate the cost of complexity** - Many overly ambitious requirements and bad design decisions really come home to roost in the software. A rule of thumb (Glass, 2002, fact #21) is that a 25% increase in requirements complexity doubles the development cost.

**DON'T Let the hardware design define the software** – It is clear to everyone that software is more difficult than hardware so why do the hardware engineers get to define the playing surface? Often hardware, such as a correlator, is designed and built before the software is even started. Hardware choices often end up constraining the software in very troublesome ways, such as in the lack of real hardware testing capabilities. Let's turn that around and have the software constrain the hardware. If changes must be made as the construction continues, consider changing the hardware instead of the software.

**DON'T Allow “Not Invented Here” thinking** – Reuse if done properly can significantly cut costs. Reuse of known packages should be the default, not the exception.

**DON'T Forget some class of users** – The users of the software will be astronomers, engineers, technicians, operators, the public, *etc.* It's easy to focus on the needs of the astronomers and forget that tools are needed for operations. Software support will also be needed for testing during construction and that need can come to dominate the schedule unless properly planned.

**DON'T Separately manage control and other software** – Separating the “control” software from the rest of the system can result in arbitrary interfaces and complexity in the software.

**DON'T Provide claimed accurate cost estimates too early** – If requirements accepted by the project in all areas (technical, operational, scientific) are not available to base a cost estimate on use a rule of thumb based on total capital cost (10%-20% are typically suggested). Bottoms up costs should only be based on requirements for the current project requirements and software reuse scenarios.

**DON'T Divide the work before the software architecture is defined** – While a geographical distribution of the development team might be inevitable, the work should not be divided until the architecture is defined, and then work should be distributed to minimize the interfaces between groups.

### 3. Things to do

**DO Make software a key part of the project** – software will be vital to making the vision of SKA a reality. The core management team (Manager, Engineer, Scientist) should have deep experience in software. Most likely this will require two Project Engineers and perhaps two Project Scientists (as the EVLA has) with different skills.

**DO Assemble a first rate team** – Most of the productivity comes from a small fraction of a typical team. So it really pays to get the best people, even though it may not be possible to collocate them in one place. The team must be diverse – architects, designers, developers, testers, and managers all have different skill sets, perspectives, and roles. There is a real concern that the talent pool of people working in this area is too small. Lupton (RL) has many interesting things to say about the challenges of staffing a project with good people, especially given the dearth of established career paths for such people. Maintaining the team for construction period and beyond requires some careful choices – form the core from people who are likely to stay around, and give them a good working environment.

**DO Hire a dedicated management team** – There is plenty of experience to show that at least 10% of the budgeted effort should go in management, partly by team members but mostly by experienced software managers. If the current trend continues, there will be a lot of oversight for the managers to respond to, in addition to their internal management responsibilities.

**DO Track risk** – Software will inevitably be an area of high risk. SKA should analyze and track risk at the project level. Some organizations are using a Risk Manager or Risk Office for this activity. The LHC Computing Grid Project has a nice example of risk tracking: see <http://lcg.web.cern.ch/LCG/PEB/risk/>. Such risk estimates can be used in allocating contingency.

**DO Plan for logistical complexity** – Blackburn (BPS) shows that in a typical year only about 30% of developer time went to writing new code. The rest went to activities like code maintenance (14%), testing (10%), documentation (6%), lab support (22%), meetings (5%), and system administration (10%). For a geographically distributed project, the overhead of meetings and travel is likely to be larger.

**DO Engage domain experts** – Domain experts (*i.e.* in the case of SKA, people who know the nuts and bolts of building and operating radio synthesis arrays) are vital to bring perspective and insight to the software development. People with deep knowledge of and experience in running current interferometers wrote the ALMA Science Software Requirements. A substantial number of those people have continued to be involved in the

software development. This will be particularly important for SKA since it will combine techniques – synthesis imaging, mosaicing, wide-field, connected VLBI, pulsar observing, SETI. It also helps to have a few genuine visionaries since, like all big steps, SKA will place us in unfamiliar territory.

**DO Demand operational model early on** – It’s a curious fact that software team is often the first to ask what the operational model is. Definition of the model seems to take 1 – 2 years of committee work so it’s best to start on it early. Include commissioning in the operational model since it will probably continue for quite a few years, and it is a big source of stress and possible disruption on the software development teams.

**DO Manage and document change** – Any substantial changes should be reviewed and accepted (or not) by an appropriate review body. Changes in scope should require approval all the way up to the stakeholders and funders. Changes in interfaces should require approval by an internal review board.

**DO Prototype, test, and simulate** - SKA constitutes a large jump in capabilities. Such jumps are hard to execute and require new technology (see, for example, Ekers, 2002). How can we be sure that it will all work? There are three types of checks available: testing, prototypes, and simulations. All three will be important for SKA, and all three provide ways to validate the software. Prototypes/demonstrators will be built prior to SKA – these provide a place to learn about the software issues. Testing is now an intrinsic, and some would say primary, part of software development methods. Simulators should be built alongside the software development, providing a means to test the software and algorithms via a continuing series of challenges of escalating complexity.

**DO Establish and debug a software process early** – Debug a process with the early design and development team rather than after you staff up the group. This process will probably have to be somewhat formal given the size of the development team and the oversight the SKA project will probably be subject to.

**DO Build a complete system early and often** – Carithers (CSNAP) describes a process in which a complete but stubbed system is built early, thus establishing interfaces first and implementation later. ALMA is also following the same approach, focusing on biannual builds that integrate all subsystems. The early freezing of interfaces allows the entire team to focus on implementation confident that interfaces they use will not change. This evolves into an approach where testing is always the highest priority. This also prevents the common problem where substantial design mistakes are found too late.

**DO Integrate early and often** – integration can be the most difficult part of a project. Do not establish a development model in which separate systems are only brought together in a final commissioning stage.

**DO Invest in infrastructure** – LCK advocate substantial investment in well conceived and executed infrastructure that is relevant to the job at hand. The “Buy-Build-Borrow”

question is key to making a project work. ALMA is reusing three major packages – AIPS++ (pipeline and offline data reduction), NGAST (archiving), and a component-container package built on top of CORBA (used in the ALMA Common Software). The net savings are at least at the 25-50% level. If it is to be useful, infrastructure must be in place early and well executed, otherwise it can act as a shared problem.

**DO Review appropriately** – The traditional NASA approach of PDR, CDR has too much rigidity for the typical software development process. ALMA is using multiple, annual CDRs in recognition of the continuing need to evaluate many aspects of the design. Blackburn (BPS) advocates data challenges in place of a Final Design Review. The review panels should be chosen carefully – the skills to review science, operations, electronics, and software are not often found in one person.

**DO Get help** – SKA will be one of the most impressive scientific projects in history. We should be able to get the interest of internationally known experts in computing. SDSS benefited immensely from the addition to the team of Jim Gray of Microsoft (a Turing Prize winner for his work on databases). The recent partnership between ASTRON and IBM on LOFAR computing is another example.

#### **4. Other related suggestions**

**Algorithm development will be vital for SKA** - It is not the same as computing and should not be done by the same people. Hire experts and support them with a small dedicated computing group.

**Establish a separate testing group** – Getting software tested by knowledgeable people is always difficult. Budget for a small group of core testers, and use external scientists to test targeted areas.

**Plan for continuing software development in operations** – Software systems need maintenance and new capabilities for the lifetime of the project. Haggouchi *et al.* (2004) describe how the Very Large Telescope Data Flow System has evolved since initial deployment, and how the activities of the DFS group has been structure to support that need.

#### **Acknowledgements**

We thank Neil Killeen and Wim Brouw for insightful comments and suggestions.

#### **References**

Ekers, R. D. 2003, in ASP Conf. Ser., Vol. 295 Astronomical Data Analysis Software and Systems XII, eds. H. E. Payne, R. I. Jedrzejewski, & R. N. Hook (San Francisco: ASP), 125

Glass, R.L., 2002, “*Facts and Fallacies of Software Engineering*”, Addison Wesley.

Gray, J., 2003, “*Distributed Computing Economics*”, MSR-TR-2003-24, Microsoft Research.

Haggouchi, K, and 25 others, “*The VLT Data Flow System, a flexible science support engine*,” ADASS 2003 conference, Strasbourg, France, 13-15 October 2003.

Lewis, H., Conrad, A., and Kilbrick, B., 2002, “*Do bigger telescopes need bigger software?*,” Advanced Telescope and Instrumentation Control Software II. Edited by Lewis, Hilton. Proceedings of the SPIE, Volume 4848, pp. 167-174.

Kemball, A.J., and Cornwell, T.J., SKA memo, *in preparation*.