

## Other People's Software

E. Mandel and S. S. Murray

*Smithsonian Astrophysical Observatory, Cambridge, MA 02138, Email:*  
*eric@cfa.harvard.edu*

### **Abstract.**

Why do we continually re-invent the astronomical software wheel? Why is it so difficult to use “other people’s software”? Leaving aside issues such as money, power, and control, we need to investigate practically how we can remove barriers to software sharing. This paper will offer a starting point for software cooperation, centered on the concept of “minimal software buy-in”.

### **1. Introduction**

What shall we do about “other people’s software”? There certainly is a lot of it out there. As software developers, we are well aware of the benefit of sharing software between projects. So why do we resist using other people’s software? How is it that we find reasons to ignore existing software by repeating the mantra, “of course we should share, but in this particular case ...”.

The factors that cause us to ignore the software of others are many and varied. A cynic might argue that it all comes down to money, power, and control. There is truth to such a position, but dealing with economic, social, and psychological factors is well beyond the scope of this short paper – or maybe any paper! Moreover, it may be that these issues always will hinder efforts to share software, unless and until our culture changes or our genes improve. But we still are left with the question of whether we can do anything today to improve the situation regarding software sharing – even while we acknowledge that our efforts may be only chipping away at the edges of a much larger problem.

We can, for example, try to expose technical impediments to sharing software. And surely the first barrier to using other people’s software is “buy-in”, that is, the effort needed to adopt and use such software. At first glance, it would seem that buy-in encompasses technical issues such as:

- the relative ease of installation of software
- the time and effort required to learn enough about the software to be able to use and evaluate it
- the design changes required to utilize the software – that is, the architectural assumptions made by the software itself

But these factors are not the whole story. Our basic attitude toward software buy-in, our very willingness to consider using the software of others, has changed with changing times.

## 2. Three Eras of Software Buy-in

From the Dark Ages until the mid-1980's, we lived in a Proprietary Era that demanded total buy-in. This was the age of isolated mini-computers running closed operating systems that were written in assembly language. Given the enormous cost of these machines and the different interfaces they presented to users and developers alike, there was little possibility of utilizing equipment from more than one vendor.

The astronomical software we wrote was influenced heavily by this proprietary culture: it was tailored to individual platforms with little or no thought to portability. A lot of our code was written in assembly language to save memory in these 64 Kb environments and to increase performance on slow CPUs. We also used FORTRAN compilers that incorporated vendor-specific language extensions, program overlays, and other non-portable techniques.

All of these efforts served the aim of optimizing software meant to be used only within individual projects. The central idea behind our efforts was to build the best possible software for our own systems. Indeed, our community was "data-center centered": astronomers visited data centers and telescopes to utilize project hardware and software on their data. Under such circumstances, buy-in required duplication of hardware and thus was a decision made by high-level management.

The Proprietary Era came to an end in the mid-1980's with the rise of workstations based on the Unix operating system. The central strength of Unix was its portability, and by adopting this little known operating system, workstation vendors changed the climate of software development overnight. We witnessed the rise of the Consortium Era, in which portable application programming interfaces (APIs) were developed by consortia of (often competing) organizations. The best example of this sort of effort was the X Consortium, an alliance of 75 large and small companies who agreed to standardize graphics and imaging for workstations on the X Window System. In doing so, they made possible unparalleled opportunities for developing portable software on a wide spectrum of machines.

In the astronomical community, the new push to portability led to the development of "virtual" analysis environments such as IRAF, AIPS, MIDAS, and XANADU. These systems offered sophisticated analysis functionality for almost all of the popular machines used in astronomy. Designed to be complete environments for user analysis, they offered buy-in at the architectural/API level. Once an analysis environment was chosen for a given project, buy-in was accomplished by mastering that environment and then tailoring the software design to exploit its strengths and evade its weaknesses. API-based buy-in established a new set of expectations for software development and use.

We believe that the Consortium Era is over, and that we have entered into a new Free-For-All Era. The most visible evidence of this change is the recent demise of the X Consortium. But this trend away from consortia-based software

also is seen in the astronomical community, where there has been a weakening of long-term alliances between development groups. This weakening is an outgrowth of the maturity of our current analysis systems: with an overwhelming amount of software increasing the overlap between systems, it has become harder to choose between them. Furthermore, new development tools such as Java and Tcl/Tk have increased the pace of software creation, while shortening individual program lifetimes. The current watch-word seems to be “let’s run it up the flag pole and see who salutes”. Software is created, offered, and abandoned with startling rapidity. It is little wonder that consortia cannot keep pace with this explosion of software. Their decline has given way to temporary alliances that exploit the latest technology offering.

### 3. Minimal Buy-in Software

In such a fast-paced world, everyone is hedging their bets. It is becoming increasingly difficult to choose between software offerings whose longevity is questionable. It is even harder to invest time and effort in rapidly changing software. With so much software and so much uncertainty, we try to use everything and commit to nothing. Having little time or patience to investigate new software, we demand that software be immediately usable, with no buy-in at all, before we are willing to try it out.

For example, distributed objects are being hailed as the key to using other people’s software. And indeed, the concept of hundreds of black-box services being available on a wide area “message bus” is very appealing. It promises a new world in which individual missions and telescopes can offer services specific to their data, while in turn making use of services provided by other groups.

But the reality of distributed objects does not match the advertising. Current schemes (CORBA, ToolTalk, OLE) require substantial architectural or even hardware buy-in. These systems have complex APIs, and some of them even have new language requirements. Such a high buy-in cost presents a dilemma: who will commit first to a complex distributed object architecture? Who is willing to build software that only will run, for example, on a ToolTalk-enabled platform, thereby shutting out users who do not run ToolTalk (for example, nearly all Linux users)? Such a design decision simply is not viable in a distributed community such as ours, where portability is taken for granted. We are lead to the conclusion that community buy-in is necessary in order for current distributed object schemes to succeed – and this brings us back to the original problem!

Perhaps we need a new way of looking at the problem of other people’s software. Perhaps we need “minimal buy-in” software, that is, software that is too easy *not* to try out.

The key to minimal buy-in software is that it seeks to hide from its users the complexity of complex software. This means striking a balance between the extremes of full functionality (in which you can do everything, but it is hard to do anything in particular) and naive simplicity (in which it is easy to do the obvious things, but you can’t do anything interesting). Minimal buy-in acknowledges that design decisions must be made up-front in order to achieve this balance.

Another way of expressing this is to say that minimal buy-in software caters to developers as if they were users. It achieves ease of use for developers by emphasizing simplifications such as:

- easy installation with auto configuration: `untar, make, go ...`
- no special system set-up, and especially no need for root privileges
- immediate functionality on the desktop, so that it can be tried and evaluated easily
- integration with already-familiar user and developer tools
- use of familiar files (ASCII, FITS)

These concepts often will lead to design and implementation features that are different from the usual conclusions of software engineering and computer science. For example, applying minimal buy-in concepts to message buses and distributed objects might lead to “non-classical” requirements such as:

- no need for a special intermediate message-passing process; use “whatever is around”
- configure using ASCII files
- send and receive messages/data at the command line
- utilize a familiar pattern-matching syntax for broadcasting

We need more research in the area of minimal buy-in software. We do not know, for example, how an effort to develop minimal buy-in software relates to more sophisticated implementations. Would such software be seen as precursors to a full system? Or would it be accepted as a full replacement? The balance between functionality and ease of use needs to be explored further to gain experience with minimal buy-in techniques.

At SAO, we are working on minimal buy-in messaging, using the X Public Access (XPA) mechanism as a base-line. We are extending XPA's point-to-point functionality to support broadcast messaging using well known data formats and pattern-matching syntax. We will maintain XPA's popular command-line support (`xpaset` and `xpaget`), which also provides a simple programming interface. Our evolving efforts are available at <http://hea-www.harvard.edu/RD/>.

It should be emphasized once again that the concept of minimal buy-in is only one step toward the software cooperation that is so elusive to our community. Issues of money, power, and control still loom large in the background of any discussion of software sharing. But it remains true that we need to explore such partial solutions while working on the larger issues.

**Acknowledgments.** This work was performed in large part under a grant from NASA's Applied Information System Research Program (NAG5-3996), with support from the AXAF Science Center (NAS8-39073).