Astronomical Data Analysis Software and Systems XXI
ASP Conference Series, Vol. 461
Pascal Ballester, Daniel Egret, and Nuria P. F. Lorente, eds.
©2012 Astronomical Society of the Pacific

IRAF: Lessons for Project Longevity

Michael Fitzpatrick

National Optical Astronomy Observatory, 950 N Cherry Ave, Tucson, AZ 85719, USA

Abstract. Although sometimes derided as a *product of the 80's* (or more generously, as a *legacy system*), the fact that IRAF remains a productive work environment for many astronomers today is a testament to one of its core design principles, portability. This idea has meaning beyond a survey of platforms in use at the peak of a project's active development; for true longevity, a project must be able to weather completely unimagined OS, hardware, data, staffing and political environments. A lack of attention to the broader issues of portability, or the true lifespan of a software system (e.g. archival science may extend for years beyond a given mission, upgraded or similar instruments may be developed that require the same reduction/analysis techniques, etc) might require costly new software development instead of simple code re-use. Additionally, one under-appreciated benefit to having a long history in the community is the trust that users have established in the science results produced by a particular system. However a software system evolves architecturally, preserving this trust (and by implication, the applications themselves) is the key to continued success.

In this paper, we will discuss how the system architecture has allowed IRAF to navigate the many changes in computing since it was first released. It is hoped that the lessons learned can be adopted by software systems being built today so that they too can survive long enough to one day earn the distinction of being called a *legacy system*.

1. Introduction

IRAF ((Tody 1986) and (Tody 1993)) has been a tool familiar to astronomers for the past 25 years, longer if you include the design phase, and remains a highly productive environment today. It was originally designed to meet the needs of astronomers when the computing capabilities and processing requirements were radically different from the present day, however we believe the fundamental design of the system is sound and can evolve to preserve the not only the investments made in IRAF development, but the trust users have in the science results as well. Further, we believe the longevity of the system can serve as an object lesson for projects now in development that might under-estimate how long their software might actually need to be supported.

IRAF was designed from its inception to be a portable system, and while some applications or supported devices are now obsolete, the system as a whole remains highly portable to modern computers and operating systems. Since its first release, some 22 different platforms have been supported; there is little technical reason to expect that IRAF cannot continue to be ported to the next generation of computers and beyond (with varying levels of effort, of course).

IRAF was also designed to be a user-extensible system, indeed far more code exists in the IRAF community than in the core system itself. More than 120 external user-written packages have existed during IRAF's history, however only a few dozen are still in common use or active development. Although current system development is focused primarily on supporting needs within NOAO, new platform ports of IRAF (e.g. the recent 64-bit port) make every effort to require no (or rarely, minimal) changes to external package code in order to preserve the portability of the the packages as well.

The key to the portability of the system is in the architecture and the decision made early in IRAF's design phase to build a complete *environment* for applications development. This then requires only that the IRAF framework be ported to new hardware and allows science applications to ignore the details of the underlying platforms. The result is that literally millions of lines of code can effectively be ported to a new platform by a single developer with less than a few months' effort. When compared to similar projects where substantial effort is/was required for major language updates, a switch in graphics toolkits, adding 64-bit support, or a paradigm shift from multi-user servers to laptops (to the cloud?) computing, this up-front investment in design has paid for itself many times over.

This attention to portability at the beginning of the project is largely responsible for its longevity. IRAF remains a highly productive system today, user statistics support the idea that there are roughly 5000 *regular* users worldwide, and even a casual browsing of journals or *arXiv:astro-ph* will reveal regular mention of IRAF as the system used to reduce data or obtain measurements. Astronomers are now so familiar with the system that phrases such as *"..data were reduced using standard IRAF tasks..."* are universally understood in the community. A search through the *ADS* or *astro-ph* archives show that even with the most conservative estimates, IRAF has been used on average in one *refereed* paper for each day since the first public release, a remarkable statistic for any observatory facility.

2. IRAF and Astronomy Software History

The IRAF project began officially in the fall of 1981 at Kitt Peak National Observatory (NOAO was not yet formed). The preliminary system design was completed early the next year and the first version of the CL command language and applications programming environment were completed in 1982. The *IRAF Group* was officially formed in 1983 and over the next few years IRAF was released internally at NOAO and then to a few outside sites for testing in 1985. The first public release of the system was in 1986 to some 40 sites running both UNIX and VMS. The Space Telescope Science Institute (STScI) selected IRAF as the platform for the Science Data Analysis System (SDAS) in 1983 and carried out the initial port to VMS and extended the CL capabilities. Since then, IRAF has been ported to a wide variety of hardware platforms and operating systems (Figure 1).

The first public release of IRAF contained a basic toolkit of reduction and analysis tools (far fewer than are available today), but more fundamentally it represented a *common system* all astronomers could use. Before the release of IRAF (and similar systems developed about the same time such as MIDAS (Banse et al. 1983), Starlink (Disney & Wallace 1982), etc.) data typically had to be reduced at the observatory using in-house software or with code developed by the astronomers themselves, neither of which was widely distributed or supported for general use. Having a common system meant that

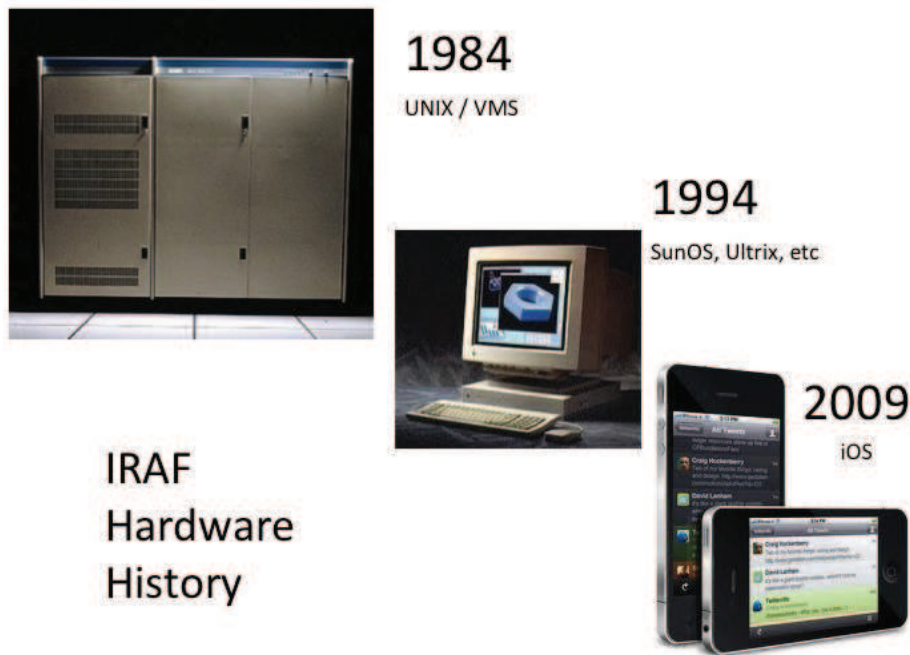

Figure 1. The advancements in hardware since the original IRAF release mean that a system which once required a computer that filled a room, today runs on a mobile phone that fits in your pocket. Applications code is unchanged.

applications could be built and shared easily, and indeed the following years saw an explosion in user-written packages meant to reduce specific instrument data (e.g. locally written tasks optimized for a particular observatory) or address larger science areas (e.g. generalized 2-D Echelle reductions). Shared systems like these changed the way the astronomical community thought about its software and helped form the basis for meetings such as ADASS.

Eventually, however, institutional support for the altruistic development of systems for the wider community was replaced by a model in which software was more tightly coupled to a particular mission or project, often due to budget constraints. This also happened at a time when the choice of programming languages grew, as did the desire for new features, e.g. GUIs replaced command-line apps, Object-Oriented programming became the new standard, and people generally wanted to use the latest technologies. As a result, user-written software began to migrate to these alternatives and there was a decline in the adoption of IRAF as the platform for new development, however many projects are still reliant on IRAF for daily operations.

To be sure, new systems for large projects are still being built in the community; ALMA, LSST, GAIA, LOFAR and many others must meet specific and daunting data challenges in order to be successful. The scale of these projects leads inevitably to a custom solution and this is understandable from many perspectives. Even if the system developed by the project cannot be distributed for community use (e.g. because it requires a super-cluster to run efficiently), there are no doubt many standalone tools that

will be made available for use and adopted by other projects. Although large projects may no longer be funded to build community software per se, they still recognize the benefits and importance of using their projects to support the community by sharing code.

Since the early days of ADASS there have also been discussions about the *Future of Astronomical Data-Analysis Systems* (FADS) (Noordam & Deich 1996), now called the *Future Astronomical Software Environment* (FASE) (Tody et al. 2009) that will someday be the one ring to bind us all. However, the rise of languages such as Python within the astronomical community have begun pushing the pendulum back to the idea of astronomers writing their own analysis code to replace legacy systems. The danger of such a swing is that the community loses not only the trust in the science results from various implementations of the same task, but that the long-term cost of supporting the diversity of applications is not fully appreciated. Given a modern applications framework, most astronomers would likely be happy to adopt it for development, however migrating large projects to a new framework is a major undertaking and fraught with politics. The current funding realities make such a shift to a common *FASE* system challenging unless there is once again an institutional willingness to invest in it for the long-term benefit of the astronomical community.

3. IRAF Architecture and Interfaces

The IRAF architecture is most easily thought of in terms of three layers (Figure 2): the lowest level is the *Host System Interface* (HSI, or *IRAF kernel*) which is responsible for all direct interaction with the host system. Above that is the *Virtual Operating System* (VOS) that provides an abstraction of all the facilities needed by an application (e.g. file and image I/O, graphics, memory and network management, etc). The final layer consists of all the applications in the system (both compiled and scripted), these can use only whatever functionality is provided by the VOS. All software above the HSI is completely portable to any IRAF host. IRAF is ported by implementing the (or porting a similar) HSI for the new host; note that the effort required to port the system is independent of the amount of code above the HSI, and once the *system* is ported, no additional effort is required to port new applications software.

The benefits of this architecture, in particular the use of abstract interfaces, should be readily apparent. Projects may no longer have the luxury of being able to build a virtual operating system to insulate their applications, but building many applications that each rely heavily on the features of a specific toolkit or OS-feature greatly increases the cost of extending the system in the future. For example, consider a suite of applications that use CFITSIO calls directly rather than even a minimal interface to an image data model. Adding support for a new image format or a different FITS library requires substantial effort in all applications instead of just a central library. In practice, experienced programmers will develop library code for common functionality, however recent trends towards *agile* methodologies discourage them from building these interfaces with a fully general design.

3.1. IRAF and Programming Languages

The IRAF Command Language (CL) acts as both the interactive environment for users, and as a script interpreter. As with the science tasks, the CL application is layered on the VOS interfaces and does not communicate with the host system directly. Script tasks

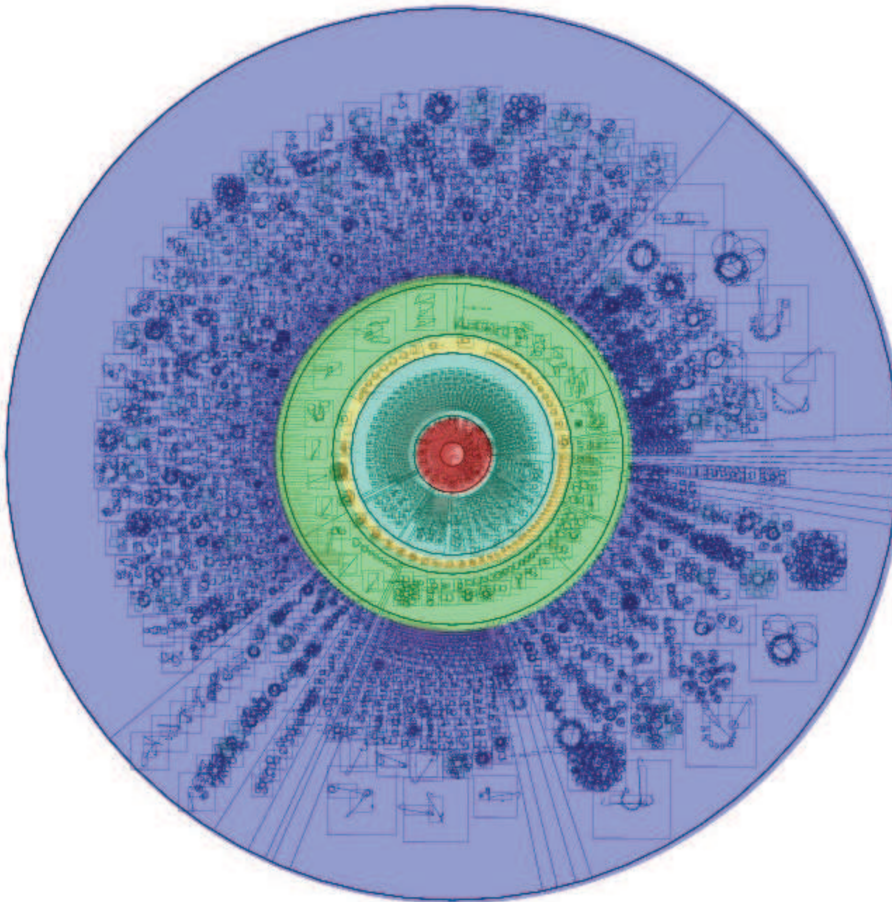

Figure 2. A visualization of the IRAF system architecture. The inner red circle is the IRAF kernel (HSI) code, which is the only code that needs ”porting”. The next layer is the VOS interfaces surrounded by a thin layer of core system applications (graphics, utilities, etc). Science code is show in the next green band, and the outer purple band represents code written by the user community.

are by definition portable and require changes only when the tasks they call are modified (e.g. changes to parameters) between IRAF versions. Although the CL scripting language is fully functional, it has sometimes been stretched beyond its original purpose (i.e. lightweight tasks, not monolithic pipelines) and is seen by some as inadequate as a result. This same observation would likely hold for a comparison of any shell language versus what could be accomplished by languages designed for large-application development, and monolithic applications will continue to be written in both languages.

NOAO has been able to effectively use the CL scripting language to build large pipeline systems (NHPPS) (Valdes & Marru 2011), and despite alternative interfaces and ad hoc methods available for a decade or more, CL remains the dominant language for user-written scripts. The success of the pipeline projects has been a recognition that CL is simply not appropriate for all aspects of the project and a willingness to use other technologies as needed. For instance, a pipeline module built on compiled IRAF tasks

can logically be written in the CL language, however triggering and scheduling of the pipeline stages is better managed in another way.

The CL has always been intended to be a replaceable component of the IRAF system, and at the time of its design the number of scripting languages available (embedded or not) was much more limited. The number of scripts now in regular use however means that even if the interactive CL or the primary scripting language itself were to be replaced, the CL interpreter would need to continue to be maintained and available. Embedding a scriptable language in an application or system is a common method of providing extensibility, what we didn't properly anticipate was that users will often try to use scripts in ways that weren't originally intended.

Compiled applications are written in a language designed specifically for IRAF called SPP (the *Subset Pre-Processor*, *"subset"* because the full language was designed but not fully implemented). SPP code is processed into a standard host language for compilation (C on most systems today, Fortran originally), providing another layer to isolate the application from the host system or the portability issues that accompany host compilers.

There is perhaps no feature as maligned or as closely associated with the idea of IRAF obsolescence, as the SPP language used to implement the compiled applications and system interfaces. There is also, other than perhaps the architecture itself, no feature as singularly responsible for the longevity of the IRAF system. There are undoubtedly *better* languages available, but what's often lost in the animosity towards SPP is the fact that SPP was intended to make the IRAF system portable, not to break new ground in language development.

At the time IRAF was designed, Fortran and C were the established languages for scientific and systems computing. SPP shares many of the constructs available in these languages and most programmers would be able to read the code with little problem, but because it is a pre-processed language we have the ability to control how the SPP is converted to the host language for compilation. This issue was especially important during the 64-bit port where conflicts between the mixed use of integers and pointers in IRAF code, and the 64-bit integer model used by the host compilers, could *only* be resolved by modifying how the target code was generated in the pre-processor. Had the system been written entirely in C, such a port would require modifications to every application in the core system (and external package) and would be far beyond any resources available.

Programming language debates are as old as computing itself, and often serve no useful purpose. Languages are designed to meet the needs of a specific community of users, and when projects make a fundamental technology choice such as the implementation language, they do so to meet the needs of *their* project. Had IRAF been designed twenty years later it is almost certain things would be different, e.g. the argument for an IRAF-specific scripting language isn't as obvious given the choice of embedded languages today.

On the other hand, a core principle of portability would still support the idea of proprietary languages within the system. In the last twenty years we've seen the rise of new languages on a regular basis (Tcl/Tk and Java in the 90's, and most recently Python and web-applications), it is hard to believe there won't be something coming to draw

our attentions in the future. Whether intentional or not, for a project to last decades the foundations on which it is built must survive the same period.

4. Replacement Costs and Considerations

Over the years, many people have wanted to replace IRAF with what they view as something better. These efforts often get started and then fade quickly once the amount of work required becomes clear. The principle problem of course is that *'replacing IRAF"* is not actually anyone's job description at the moment, and so it becomes a part-time effort. Projects such as PyRAF serve only to replace the CL functionality and don't represent complete systems in themselves. While the ability to program IRAF tasks in a new language is a valuable and welcome addition, the majority of work in PyRAF scripting is (by necessity) in writing new applications. A transition to a new language or environment can only be said to be complete once the many established tasks in IRAF are re-implemented.

If you consider the number of tasks in the core system and just a few of the most common external packages, you would find there are more than 4000 tasks that are candidates for a system that might be considered a complete IRAF *replacement*. If you further assume that as much as 99 percent of those tasks are either utilities supplied with the new framework or are simply not needed, the list becomes just a few dozen applications. This seems a manageable number until you look at what the tasks are likely to be: basic CCD processing tools, an echelle reduction suite, aperture and PSF photometry, instrument-specific tasks, image interpolation libraries, and so on. At even a few-months of effort required for each of these tasks the amount of effort required to replace *just 1 percent* of the IRAF task base easily becomes a person-decade (and likely much more).

The benefits of such an effort also aren't immediately clear: Is a e.g. CCDPROC task written in a new language any better than the one we already have? The danger also exists that when re-implementing these tasks bugs are introduced, or the temptation to add new "features" is too great, and the new task must then re-establish the trust by the community in its results. Technologies that bridge older code to new languages or environments, or frameworks that unite systems with a shared infrastructure, are the only practical means of preserving the trust and investment in legacy science code, the cost of replacement is otherwise simply too high in the current (and foreseeable) funding environment.

4.1. IRAF and Project Philosophy

It has long been recognized (DeRemer & Kron 1976) that the process of building large systems is distinctly different from that of building small applications. Building a *system* requires many people, careful planning and highly cohesive modules; the resulting architecture may be complex to understand, but if it is done correctly, will contain modules that do not require major revision in the face of change. In contrast, a small application can be built by a single person or small group, but the process is often less formal and the emphasis may be more on rapid development than generality.

IRAF is an obvious example of the large systems approach (few would argue that it is not complex to understand), and it benefits from the stability granted by a careful design. Open Source projects however tend to be organized using the small applications model; a distribution is created by collecting toolkits to form the framework and

development is focused on applications or libraries (or classes), often by individuals or groups working in a distributed fashion. Such a project may well succeed, but may require more work to maintain in the long term. For example, a core toolkit chosen to provide graphics can potentially affect all applications if a decision is made to later replace it. In the large systems approach, the graphics toolkit is abstracted and can be replaced independently. Even if a project is built using a so-called portable language, it may enjoy only a *false portability* due to dependencies on third-party toolkits making up the core infrastructure, or the highly version-dependent nature of the language itself (e.g. Python).

Institutional support, or at least some form of strong central project management, is required to build a system with a well-designed architecture and the attention to detail that will allow a project to survive for decades. The size of the astronomical programming community means there simply aren't the numbers of people available for a large Open Source effort, no matter how well organized it may be. This again argues against the idea that working and reliable code should be replaced in favor of something new, and *for* ideas that view legacy code as a valuable resource that should be preserved and integrated with new technologies as a community-wide cost-saving strategy.

5. The Future of IRAF

As stated above, there is no *technical* reason IRAF cannot continue to be maintained and distributed for years to come, the system remains as stable and portable as ever. The main threat, as with many projects, is simply a lack of resources. Development at NOAO is focused on how the system is used within other projects rather than on development of the system for its own sake. Given infinite resources there is still much that could be done in terms of developing new science applications and system interfaces, expanding the programming options and moving the system as a whole towards frameworks such as FASE.

One small step being taken on this path, given the resources we *do* have, is a project underway to integrate IRAF with Virtual Observatory (VO) technologies. Aside from being a necessary first step towards true integration of all desktop software in a FASE-like framework, we've been able to use this project to expand the capabilities of *all* tasks in the system to access and utilize remote data, interoperate with other applications, and work with new data formats. Similarly, development of pipelines built on IRAF have driven new capabilities for processing large-format detectors and distributed computing.

Although perhaps not as active as it once was, IRAF remains highly productive and far from obsolete in the astronomical community. (We hope the same can be said about the team that created it).

Acknowledgments. The author would like to thank members of the IRAF Project, and Doug Tody in particular, for helping to create such a stable and productive tool for astronomy.

References

Banse, K., Crane, P., Ounnas, C., & Ponz, D. 1983, Proc of DECUS, 1

DeRemer, F., & Kron, H. 1976, in IEEE Transactions on Software Engineering (IEEE), vol. SE-2 of IEEE Transactions on Software Engineering, 80

Disney, M. J., & Wallace, P. T. 1982, QJRAS, 23, 485

Noordam, J. E., & Deich, W. T. 1996, in ADASS V, edited by G. Jacoby, & J. Barnes, vol. 101 of ASP Conf. Ser., 229

Tody, D. 1986, in Instrumentation in Astronomy VI, edited by D. Crawford, vol. 627 of Proc. SPIE, 733

— 1993, in ADASS II, edited by R. J. Hanisch, R. J. V. Brissenden, & J. Barnes, vol. 52 of ASP Conf. Ser., 173

Tody, D., Grosbol, P., Cotton, W., Fitzpatrick, M., Paioro, L., & Surace, C. 2009, in ADASS XVIII, edited by D. Bohlender & D. Durand & P. Dowler , vol. 411 of ASP Conf. Ser., 514

Valdes, F., & Marru, S. 2011, in ADASS XX, edited by I. N. Evans, A. Accomazzi, D. J. Mink & A. H. Rots, vol. 442 of ASP Conf. Ser., 211