Realizing Software Longevity over a System's Lifetime

Kyle Lanclos,[a] William T.S. Deich[a], Robert I. Kibrick[a], Steven L. Allen[a], John Gates[a]

[a]UCO/Lick Observatory, University of California, Santa Cruz, CA 95064 USA
http://www.ucolick.org

Abstract. A successful instrument or telescope will measure its productive lifetime in decades; over that period, the technology behind the control hardware and software will evolve, and be replaced on a per-component basis. These new components must successfully integrate with the old, and the difficulty of that integration depends strongly on the design decisions made over the course of the facility's history. The same decisions impact the ultimate success of each upgrade, as measured in terms of observing efficiency and maintenance cost.

We offer a case study of these critical design decisions, analyzing the layers of software deployed for instruments under the care of UCO/Lick Observatory, including recent upgrades to the Low Resolution Imaging Spectrometer (LRIS) at Keck Observatory in Hawaii, as well as the Kast spectrograph, Lick Adaptive Optics system, and Hamilton spectrograph, all at Lick Observatory's Shane 3-meter Telescope at Mt. Hamilton.

These issues play directly into design considerations for the software intended for use at the next generation of telescopes, such as the Thirty Meter Telescope. We conduct our analysis with the future of observational astronomy infrastructure firmly in mind.

Keywords: modular design, instrument software, software architecture, software design, Lick Observatory, Keck Observatory

1 INTRODUCTION

Major work performed by the Scientific Programming Group at University of California Observatories, Lick Observatory (hereafter referred to as UCO/Lick) comes in three basic areas: deployment of software for new facilities, incremental updates to existing facilities, and the complete overhaul of facilities that are no longer maintainable. The success of incremental updates often depends heavily on the success of the initial deployment; similarly, the need for a complete overhaul is mitigated by the success of both the initial deployment and subsequent updates.

We assert that software maintainability must be a critical design consideration for any scope of installation, update, or replacement. We offer up our own recent project work to highlight both success and failure in achieving desired levels of maintainability.

We generally see two fundamental approaches with software: monolithic design, and modular design. Modular design typically includes individual components, each performing a discrete task, or small set of closely related tasks; by contrast, monolithic design would contain a single component, or suite of components, that performs a broad set of functions in a tightly integrated package. This observation is not limited to applications; software frameworks and support libraries experience the same basic design conflict.

When working outside the boundaries of a given body of code, the two approaches produce similar observable behavior; however, when the need for internal maintenance occurs, the differences become clear. With small, discrete components, it is much simpler to ensure that a

Send correspondence to K.L.: lanclos@ucolick.org; W.T.S.D.: will@ucolick.org, R.I.K.: kibrick@ucolick.org; S.L.A.: sla@ucolick.org; J.G.: jgates@ucolick.org

minor update does not provoke unintended consequences; with larger, more inclusive components, the possibility of a small change affecting seemingly unrelated code is a very real danger. This additional risk can have a chilling effect on the deployment of any modifications, especially if the existing code base is no longer familiar to the software engineers responsible for its maintenance.

2 SUCCESS STORY: KECK TASK LIBRARY (KTL)

To a first approximation, KTL [1, 2] is a keyword/value application programming interface (API) for communication between servers and clients. A KTL service represents a set of keywords, managed by one or more software daemons (dispatchers); a KTL client is software that reads and/or writes keyword values. Developed primarily for use at Keck Observatory, KTL is also an integral component of modern instrument software at Lick Observatory.

KTL defines a standard interface between a client and a dispatcher. Beyond this interface layer, KTL does not mandate a particular communication implementation between the client and dispatcher. Client software concerns itself solely with the basic client-facing KTL interface. How a particular request is handled, or even where that request is directed, is not visible to the client that made the original request.

The initial development of the reference KTL C interface took place in 1993. Regular, active maintenance of the KTL code base occurs at both Keck Observatory and UCO/Lick.

2.1 KTL as a key enabler of a mechanical upgrade

The hard division between client and dispatcher in a KTL service enables independent modification of one or the other, without requiring simultaneous updates to both. This separation of responsibility resulted in significant time savings for the 2009 upgrade of the Low Resolution Imaging Spectrometer (LRIS) at Keck Observatory [3]. From the observer's perspective, the most prominent component of that upgrade was the installation of a new detector for the red side of the spectrograph; along with the detector and its dewar, a new dewar focus mechanism was also installed. All other keyword-enabled motor control aspects of the LRIS instrument remained unchanged, with the exception of the red side focus stage.

The structure of KTL enabled software engineers at UCO/Lick to logically replace this single motor stage without modifying any of the client software, or any of the existing dispatchers serving other functions for the LRIS spectrograph. Exactly two keywords containing information about the red focus mechanism were provided by the old hardware controller; by modifying the client-facing components of the KTL layer, the requests for those two keywords are now relayed to a unique software dispatcher, whose sole responsibility is the control of new focus mechanism. The new dispatcher provides many additional keywords regarding the state and readiness of the new motor stage, but awareness of this new functionality was not required in order for the existing code base to perform its normal tasks.

This blending of multiple generations of software into a single KTL service is extremely powerful, in that it not only simplifies incremental updates to software, but it also allows for seamless software integration of incremental updates to the mechanical or electrical components of an instrument. The need for this capability becomes more prominent as a given instrument increases both in size and in cost. The more expensive an instrument is, the more attractive it is to perform a modest, discrete modification that allows the majority of the instrument control architecture to remain unchanged.

2.2 KTL Python: the importance of portable implementations

The KTL API itself is largely procedural, but has two logical components that lend themselves to an object-oriented approach: keywords, and services. A service is a defined collection of

keywords; a keyword is a single value, with metadata that indicates what type of value a given keyword represents.

In order to properly enable KTL functionality for the Python programming language, a robust, native Python interface was required. The KTL Python interface began its life as a direct procedural translation of the basic components of the KTL API, prompted by development for the Keck Interferometer in 2004. A major overhaul of this interface began in 2008, including a new object-oriented API for KTL services and keywords. The KTL Python interface was deployed as a key component of software for the upgrade of Lick Observatory's Kast spectrograph in March 2009, followed quickly by use with the Shane AO spectrograph, Prime Focus Camera, and Automated Planet Finder spectrograph.

The incremental addition of the KTL Python layer was simplified greatly by the design decision in the early 1990's to implement the base KTL API in C. This enabled the creation of native Tcl, Java, and now Python KTL interfaces, all without modification to the base KTL API itself. Java and Python were in their infancy as languages when KTL was first established, but this represented no significant barrier for their respective KTL interfaces; it is not unreasonable to expect that languages not yet in existence today could similarly be enabled with a native KTL interface.

The creation of new interface layers in this fashion highlights an interesting class of maintainability. On the one hand, a new interface layer enables new programming languages to interact with a wide range of existing services, without requiring any modification of the services themselves. This may give the old services, and even the KTL API itself, a new lease on life, providing a new mechanism that keeps an older standard relevant for modern computing.

On the other hand, by layering the native KTL Python interface over the established KTL C interface, the KTL C interface retains its relevance not only as a reference implementation, but also as a shared foundation for low-level fixes and enhancements. If the KTL API were defined in the abstract, as opposed to its current form in C code, any fixes or enhancements would be the sole domain of an isolated KTL implementation. As an example, during the development of the KTL Python interface several fixes were made to the routine that translates string values to native boolean, float, and double formats; because these modifications occurred within the fundamental KTL C layer, all language-specific interface layers immediately benefited from the fixes. If the KTL API were defined in the abstract, each native language implementation would require individual fixes in order to remain fully compatible with other KTL implementations.

3 SUCCESS STORY: MUSIC MESSAGING

MUSIC messaging [4] predates the creation of KTL, with initial development and deployment taking place in 1988. The original design was for a simple, lightweight generic messaging system, though the process was shaped by the needs of a new data taking system under development at the time. Early versions acted in a direct point-to-point model, which was followed quickly by a multi-point, broadcast model.

MUSIC messaging is somewhat orthogonal to the KTL API: where KTL normalizes the interaction between servers and clients, MUSIC messaging makes no assertions about the contents of any given communication; where MUSIC messaging explicitly defines a low-level protocol for client-server communication, KTL leaves all communication details up to the developer.

3.1 MUSIC messaging for data taking systems

The original development of MUSIC messaging went hand-in-hand with the development of a new data taking system in the UCO/Lick CCD lab. Different components of this system ran on SunOS workstations, VME crates, and early Linux hosts. This data taking system was in active, nightly use from its initial deployment in 1989 until 2009, when the last of the old camera controllers based on VME crates and/or 6502 systems was retired at Mt. Hamilton.

The data taking systems at Mt. Hamilton began a slow migration in 1996, transitioning to UCAM controllers [5] and a new suite of control software. Deployment started with the three Mt. Hamilton guide cameras in 1996, with the most recent deployment in 2009 for the Kast spectrograph. The suite of software used with UCAM controllers also leverages MUSIC messaging, primarily for communication between different software components running on the same host; the previous generation used MUSIC messaging for communication between distinct hosts.

Similarly, MUSIC messaging is an integral component of data taking systems at Keck Observatory, from the initial deployment of the HIRES instrument [6] in 1993, up through and including the LRIS red side upgrade in 2009. While the systems at Keck also leverage VME crates running VxWorks, and use MUSIC messaging for communication, the commonality between the implementations largely ends there, as the Keck systems use a completely different suite of control software.

By focusing on single task and performing it well, MUSIC messaging was successfully integrated as a stable component of three wholly independent suites of software. This accomplishment alone validated the amount of time taken to develop and implement MUSIC messaging, in that the cost of the initial development has since been amortized over twenty years of productive use at multiple institutions.

3.2 MUSIC messaging: the continuing importance of portable implementations

As mentioned above, the core MUSIC messaging functionality is actively used on a wide range of platforms, though the platforms listed are only a subset of the total range of computing platforms that successfully leveraged MUSIC messaging. This cross-platform compatibility is key to the historical success of MUSIC messaging.

Similar to how the KTL C implementation received translation layers over time, the fundamental MUSIC C libraries are also amenable to adaptation for other languages. A Tcl interface layer for MUSIC started development in 1999, and remains in widespread use as a critical foundation for higher level applications, such as instrument motor control services.

Within the UCO/Lick software environment, there is a small but slowly growing desire for a similar Python interface layer. Thanks to the portable, minimalist nature of the core MUSIC messaging libraries, establishing this new layer is expected to be straightforward and time-efficient, even more so than the development of the KTL Python interface layer.

The portability and ease-of-translation of MUSIC messaging are key prerequisites for any potential expansion of its use. While the range of competing communication solutions is much greater than it was twenty years ago, it remains the case that many off-the-shelf protocols would require additional local engineering to satisfy the use cases within our software environment. This is not to suggest that MUSIC messaging represents a perfect solution, but rather, it benefits greatly by its evolution alongside the evolution of the communication needs of the software deployed by UCO/Lick.

3.3 MUSIC messaging: reusing existing software for new tasks

The ongoing success of MUSIC messaging made it attractive as a communication layer for KTL. Leveraged via fiord, an additional abstraction layer under KTL, MUSIC messaging is the sole communication layer for all KTL clients and servers authored and maintained by UCO/Lick. The deliberate re-use of an existing and well-understood system benefited our overall development effort in two key ways.

The first and perhaps more straightforward benefit was a direct reduction in the amount of time required to develop the communication layer for KTL services. Regardless of the communication protocol, development time was required for integration and testing; by using an

established protocol, no time was required for development and testing of the protocol itself. The overall process was made more efficient by using a familiar protocol.

The second benefit to expanding the use of MUSIC messaging is the increased scrutiny of the core protocol and libraries. Prompted by ongoing development and deployment of KTL services, improvements and bug-fixes in the core libraries have taken place a number of times over the lifetime of the software. These fixes can subsequently be leveraged by the original, non-KTL software. By achieving a critical mass in terms of how widely the software is used, we increase the likelihood that MUSIC messaging will be actively maintained and ready for deployment on modern computing platforms for future projects, which may include recovery from hardware failure as well as planned deployment of new systems.

4 THE PROBLEMS OF COMPLEXITY AND INACCESSIBILITY

With respect to the software that UCO/Lick supports, two factors appear to contribute heavily to whether a given element of software is deemed unmaintainable: complexity, and a lack of accessibility.

By the nature of observational astronomy, not all of our tasks are simple, and a certain minimum of complexity is required for proper operation. It is vital, however, that this complexity not intrude into the other, perhaps simpler aspects of the software. It is especially vital that the overall software architecture not be unnecessarily complicated as a result of an isolated problem. Attempting to provide a broad range of actions or capabilities within a single tool appears to be our most common downfall in terms of generating unwanted complexity.

Whether or not the software is complex, if the maintainers have no visibility into how the software accomplishes its task, the maintainers are already at a severe disadvantage. Proper documentation can mitigate an accessibility problem, but it is by no means a cure-all; lack of proper compilers, or lack of appropriate software licenses in general are trivial examples that reduce the efficiency, or prevent outright the proper maintenance and development of deployed software. Perhaps a more common occurrence is that the documentation is not adequate to the task. When the solution to a problem is not crafted in an easy to replicate form, documentation of that solution can become sufficiently complex that it may provoke more confusion than it alleviates.

4.1 A lack of accessibility: Hamilton spectrograph motor control

The motor control system deployed in 1986 with the Hamilton spectrograph [7] has been and will remain in constant use until its planned replacement in the last few months of 2010. Due to the nature of the computing resources available at the initial deployment, the control system relies heavily on hardware, to the extent that the only interface available for control of the system is based on serial communication, appropriate for use with a serial terminal display.

Each motor in the spectrograph is controlled by a dedicated control card; higher level control is managed by a separate control system written in FORTH. In a remarkable stroke of good fortune, the active code has not required any updates since the commissioning of the Hamilton spectrograph. This fortune is all the more remarkable since UCO/Lick no longer has electronic copies of the top-level FORTH code; all that remains are the active binaries on the controller itself, and an 11x17 tractor-feed printout of the original code.

This represents the first barrier to accessibility: the maintainers of Hamilton motor control system no longer have the ability to generate the running control software from source. Modifications to the running control code are not realistically possible; recovery from any hardware failure can only occur if the existing hardware is available to bootstrap the recovery process. If any serious failures occurred and bootstrapping was not a viable recovery option, the entire control system would have to be replaced.

The second barrier is the exposed interface for the motor control system. There is no readily available mechanism for external software to communicate directly with the individual motor control cards; access to any motor control function is only possible via the end-user interface provided by the FORTH code mentioned above. This puts a burden on software developers, in that any attempt to modernize the observer-facing software would first require a server-side entity to extract the state of the spectrograph via the old user interface, and present that data in a more flexible format. While not an impossible task, it represents a very tenuous link in the chain of software control, and would not provide the level of inspection and flexibility that UCO/Lick considers necessary for robust motor control.

This limited access greatly diminished the options for modernizing the motor control system without replacing it outright. Because of the relatively high cost of replacement, the control system remained unchanged well beyond the point where it should have been modernized; only after the growing trend of intermittent failures and lost observer time due to systematic inefficiency became costly enough did moving forward with the full replacement make budgetary sense.

To be completely fair, one round of modernization did take place: the hardware serial terminal was replaced by a direct serial connection to a Linux host, which runs custom software that presents the old interface within a common terminal window. In that sense, we are fortunate that the discussions over the inclusion of a standard serial connection ended in favor of the serial line: the inclusion of a single, standardized element was enough to extend the functional life of the motor control system by at least a decade.

4.2 Creating accessibility: Hamilton motor control upgrade

All components of the existing motor control system for the Hamilton spectrograph will be replaced in the last few months of 2010. No mechanical modifications are expected as part of this upgrade; while the motors will remain the same, some amount of new wiring will be required to allow the use of new controllers. The new controllers will be off-the-shelf Galil motor controllers, and the motor control software will use the same code base that UCO/Lick deployed for several major instruments, including the Kast spectrograph at Mt. Hamilton and the LRIS red focus mechanism mentioned earlier. Communication between the controller hardware and the software dispatcher will occur via standard TCP/IP protocols, over a standard RJ45 ethernet link.

By using off-the-shelf motor controllers, we greatly improve our ability to recover in case of hardware failure, since spares are easy to acquire and test. By increasing the logical separation between the control hardware and the control software, we improve our ability to modify or replace the software without requiring a simultaneous modification of the hardware. By deploying software that is already in use for multiple instruments, we improve our ability to make fixes and modifications that could benefit a broad range of active installations. In all areas, we increase local access to each component in the motor control process, and by doing so, improve the overall maintainability of the system by decreasing the development cost of incremental updates.

4.3 A problem of complexity: Codegen and Memes

Development of Memes [8] began in 1996, with development of Codegen starting in 1998. The initial work focused on the DEIMOS spectrograph [9], commissioned in 2002 at Keck Observatory; the first deployment of Memes and Codegen was for the ESI spectrograph [10], commissioned in 1999, likewise at Keck Observatory.

In this context, Memes refers to a specific organization of metadata in a relational database. The structure of Memes is largely object oriented, in that all entries are, at their most fundamental level, a Meme, and that specific objects are derived from previous definitions; for example,

a floating point value is a number, and a number is a Meme. The basic structure was augmented by a modest quantity of external tables to establish hierarchies that were not realistically possible as Memes.

The primary function of Codegen is to translate Memes into various header and metadata files for use with KTL clients and servers. It accomplishes this by inspecting Memes for relevant KTL service and keyword definitions, establishing a keyword and data hierarchy based on that inspection, and then generates the various output files based on explicit statements within the hierarchy, as well as a certain amount of implied structure between different elements.

Codegen remained in active use through 2007; as one would expect, regular maintenance and updates were required in order to handle circumstances that were not part of the original scope. Every modification required delicate handling, due to the variety of special cases internal to Codegen that were not fully reflected within Memes. The existence of these special cases heightened the possibility of changes adversely affecting older use cases. Negative repercussions would be mitigated if individual installations of Memes and Codegen were isolated from each other; this was not realistically possible, since Memes only existed as a single, shared database instance. The vulnerabilities were not isolated to Codegen: similar problems could occur if modifications were made to low-level Meme definitions, inviting potential knock-on effects with related Memes.

Beyond the problems with growing Codegen and Memes to handle new tasks, there were also practical matters that hindered their active use. One problem was the reliance on a live, networked database; this mandated that all client development hosts have a software environment supporting access to the database, and the ability to communicate via the network with the authoritative database instance. These dependencies imposed an undesirable drain on developer resources, especially in the absence of unique benefits or functionality to justify the burden.

A second practical problem was basic data entry. Due to the complexity of the database tables involved, the manipulation of old content or the creation of new content could only be accomplished via custom tools. The canonical method involved manually entering data for each individual Meme; several attempts were made at establishing auxiliary tools to allow the batch creation of new Memes, but none of these auxiliary tools were robust or functional enough to provide any real time savings. This inefficiency was especially unfortunate considering that many of our Memes represent individual keywords, and every motor stage under software control has an identical set of keywords to represent all of the data points associated with that stage.

Looking back at the design of Codegen and Memes, one problem is that this system tries to accomplish too many discrete tasks under a single umbrella. By forcing every different type of data into the same form, nuances unique to each class of information are lost, or inadequately represented. For example, it is possible to generate technical documentation from the records contained within Memes; the resulting documents, however, are generally not usable on their own, and require further modification before they are acceptable for use. Similarly, while it is possible to retain configuration values for a KTL dispatcher within Memes, properly expressing the information, and subsequently extracting it into a useful form, requires far too much complexity and indirection for it to be an effective use of resources.

4.4 Reducing complexity: KTL XML

When the author and maintainer of Codegen and Memes left UCO/Lick, the group consensus was that we needed to develop an alternative scheme, and that testing of this new system should occur before any projects prompted further deployment of the old scheme.

Having wrestled for years with the architectural difficulties of using a live, networked database as a storage backend, one design requirement was that all functional data be stored

as human-readable text files. Another key requirement was that all structure be made completely explicit, as opposed to relying on inferences or inheritance. Internal discussions led us towards XML as a possible solution, and work on prototyping began late in 2008.

The initial prototype analyzed the Memes for a given KTL service, and constructed an XML representation of the same data. Separate tools would then directly parse the XML and generate metadata output equivalent to what Codegen produced. This prototype work used the existing Memes for the Automated Planet Finder spectrograph, which is expected to achieve completion and commissioning later in 2010, though software development for the spectrograph began in 2005.

The success of the prototype prompted the deployment of KTL XML for the upgrade of the Kast spectrograph in March 2009. The previous control system for the Kast did not use KTL keywords, thus there were no existing Memes to use as a starting point. As of May 2010, there are 577 keywords associated directly with the Kast spectrograph; though the new XML-based system made it easier to establish KTL keywords, the prospect of entering all these keywords by hand was singularly unattractive. Thanks to the text-like nature of the XML files, we were able to leverage a text preprocessor to generate the full XML from comparatively simple templates. The procedural generation of XML is completely contained within a short Makefile, and runs independent of any other data sources as part of the normal software build process. The time savings on data entry alone, compared to performing the same tasks with Memes and Codegen, were enormous. Given the time savings and reduction in data entry errors, the ability to rapidly deploy keywords in this fashion should be considered a requirement for larger and more complex instruments.

Since March 2009, the KTL XML scheme was deployed for all new or refurbished KTL services maintained by UCO/Lick, representing literally thousands of individual keywords. The cost of small modifications within the Codegen system, even the basic addition of a simple keyword within Memes, are high enough that full conversion to the KTL XML system is attractive in comparison. For large modifications, the initial time cost for a full conversion is quickly absorbed.

The availability of KTL XML enabled very rapid turnaround for two instrument projects with extremely limited resources and even more limited timescales: a motor control software upgrade for the Shane AO instrument, and the creation of a KTL dispatcher for an Omega temperature controller, the latter shipped as part of the LRIS red side upgrade mentioned earlier. In both cases, the available time for basic data entry was measured in hours, not days; in both cases, rapid development of new capabilities was required in order to fully express the capabilities of the hardware itself. The improvement in development efficiency contributed materially to the success of these projects.

With the XML files themselves checked into cvs, there is never any question about stale results or unexpected errors of translation, and there are never issues relating to database connectivity or library support. Once the source code tree is checked out from cvs, the build process for client libraries is completely local; if it worked once for a given input set, it will always work. If development occurs that might invalidate the old schema, existing installations are completely safe, thanks to the independence from any shared source of authoritative data.

With the data in a flat text format, no restrictions exist on the software used to modify the data, enabling direct manipulation by developers and also algorithmic generation of redundant content via external tools. Two common manipulations of the XML in our environment are the text preprocessor templating mentioned above, and simple macro substitution; one example of the latter would be a generated file that has the KTL service name substituted into the XML via a Makefile, rather than state the service name explicitly in the source XML.

Thanks to the structural flexibility of XML, developers are also not forced to adhere rigidly to an existing schema in order to implement desired functionality. The XML representation of

a KTL keyword uses one schema; XML description of the key mechanical features of a motorized stage, however, are not expressed within the keyword-oriented XML. Instead, secondary schema are used for this data. By breaking out the data that is not directly represented by the basic keyword XML, we avoid complicating the keyword XML, and we avoid unnecessary feature creep in the tools that process the XML.

By providing all of the metadata in a relatively accessible format, tools that previously used derivative extracts or direct database queries can instead directly use the local, authoritative XML. This includes information not typically available to a KTL client, such as acceptable position errors, or minimum and maximum values for integer keywords.

One idea raised internally would be to develop a new low-level layer for KTL client libraries that parses the KTL XML directly rather than rely on derived extracts. It would not be unreasonable to have a single KTL client library, regardless of how many services are installed, and change the behavior of the client library at runtime based solely on the content of the KTL service's XML description.

With the development of KTL XML, reducing the complexity of the data description itself not only improved the maintainability of the system, but it also enabled new functionality that was not reasonably possible with the old system. Simultaneously, staff time for repetitive tasks was reduced, and the overall reliability of the build process improved significantly. By mitigating the local cost of developing and maintaining KTL services, local interest in maintaining and expanding our use of KTL has improved.

4.5 A lack of accessibility: third-party software

The modern computing world is littered with solutions. Given the range and depth of what is freely and commercially available, the savings in initial development time due to properly leveraging external software can be substantial. The temptation to drop in a third-party product, especially when project resources are stretched thin, is enormous.

During the decision-making process, great care must be taken to ensure that the incorporation of third-party software has a net positive effect on the overall development. Tight integration with external software can lead to a hard dependency on the fate of a remote project; if the external project decides to move in a new direction, or is abandoned entirely, local developers are forced to adapt to the changes. If adaptation is not possible, the remaining option is to cling to the last supported computing platform for the software in question, and lose all hope of future portability. Remember again that we are concerned with facility lifetimes measured in decades, not just years.

A short support lifetime by no means occurs only with commercial software, though that is where many if not most of the occurrences take place. Even if you restrict the application space to observational astronomy, many interesting, well thought-out designs exist for applications, support layers, and complete frameworks that never gained traction with their intended audience, and have since fallen into disrepair and disuse.

Despite the awareness of the lack of fundamental access to allow direct maintenance of third-party software, there are times when no other options are available. One modern example within UCO/Lick is the Automated Planet Finder dome and telescope; both of these major components are provided by contractors, Electro Optic Systems (EOS), and EOS Technologies (EOST). The user interfaces, server-side control daemons, and the underlying software framework are all custom, closed-source software provided by EOS and EOST. The use of this third-party software is required in order to properly operate the dome and telescope control hardware, much of which contains custom components or firmware that are equally closed to inspection or modification. UCO/Lick has no practical choice but to tightly integrate our own software with the resources provided by EOS and EOST.

5 CONCLUSIONS

No two projects are alike, and the same goes for programmers, programs, and development environments. Trying to establish firm rules about what should or should not be done is folly, but at the same time, experience indicates that certain approaches are more likely to succeed over the long term. With that in mind, we present some of the lessons we hope to carry forward into the next twenty years of observational astronomy.

5.1 Develop for discrete tasks

If a software package performs one task and performs it well, future projects will be more likely to redeploy the known quantity rather than recreate the required functionality from whole cloth. In order for such a redeployment to be attractive, care must be taken to:

5.1.1 Avoid scope creep

When otherwise excellent functionality is clouded by unnecessary additional capabilities, the increased complexity acts as dead weight for future maintainers. A newer, unencumbered implementation becomes attractive once the maintenance burden exceeds the cost of replacing the code entirely.

5.1.2 Be open to unanticipated use cases

While the needs of today's project provide the most immediate metric for success, don't let the current requirements limit the future use of any single tool. By building in an arbitrary limit to what a program can do, you write an invitation for a future project to collide head-on with those limitations.

5.1.3 Solve one problem

It is commonly observed that it is far easier to be good at one thing than it is to be good at many things simultaneously. Solve a single problem to establish a foundation for future solutions, and let those subsequent efforts leverage earlier success as a way to ease the development path.

5.1.4 Solve the right problem

It is easy to have one's intended solution waylaid by tangential concerns. Does the immediate problem warrant the expenditure of resources to address secondary issues? If an existing software infrastructure does many things right, but only a few things wrong, developer time is likely better spent focused on only the unsatisfactory elements, as opposed to architecting and deploying an entirely new software framework.

5.2 Enable re-use via portability

If you already have the best software tools available, but they won't run on the platform that the project requires, you wind up right back where you started. Twenty years in modern computing represents a minimum of three major generational shifts in the underlying technology. In the absence of any guarantees, what can you do?

5.2.1 Avoid delicate build environments

The creation of outstanding software becomes a wholly abstract exercise if the required build environment is no longer available to compile the code into a functioning entity. This is especially difficult with very long-lived code, functioning largely in isolation; perhaps the code compiles with a specific older Fortran compiler, but not with modern Fortran compilers, and the cost of rewriting all of the embedded routines to meet the newer standard is prohibitively high. Perhaps the code only compiles with a proprietary compiler, and that compiler is no longer available. Perhaps the build environment depends on a very particular build utility, and that utility is only developed and tested on Linux hosts. Perhaps the old build environment is already gone, and a cross compiler is now an essential build tool.

5.2.2 Avoid elaborate runtime environments

The more components that are required for a given software package to run, the more likely it is that one of those components will either not be backwards compatible, or missing entirely when a redeployment is attempted on a new platform. The more specialized any single component, the more likely it is to fail when introduced to a new computing environment. By constructing software that uses only the most essential, robust components of a given runtime environment, you reduce the possibility of failure when one of those components changes, or is absent entirely.

5.2.3 Avoid restrictive support libraries

If you use a specialized library as part of your software, will that library still be available in five years? Ten years? Twenty years? If you had to purchase a license for that technology, will there still be a vendor available to provide you with a modern version when you need it most? Will you be able to produce or acquire a copy of the required library when the time comes to migrate to a new computing platform?

5.2.4 Commit early, commit often

Thorough curation of source code is the only way to ensure that you have all the original components of an old software installation. If your deployment procedures do not require that installed code be stored in a revision control system, a disconnect will inevitably occur between what is running, and what is retained.

5.3 Final remarks

Subtle shifts in the computing foundation are inevitable for any project with a long life span; core libraries will change, major versions of popular scripting languages will come and go, entire build environments may come and go as development priorities shift over time.

The projects in our environment that met with the most success were able to keep a strong focus on simplicity and portability. As we turn our collective gaze to the next generation of observational astronomy facilities, it will be an order of magnitude more important that we heed the lessons learned from the successes and failures of today.

References

[1] A. R. Conrad and W. F. Lupton, "The Keck Keyword Layer," in *Astronomical Data Analysis Software and Systems II*, R.J. Hanisch, R.J.V. Brissenden, & J. Barnes, Ed., *Astronomical Society of the Pacific Conference Series* **52**, 203–207 (1993).

[2] W. F. Lupton and A. R. Conrad, "The Keck Task Library (KTL)," in *Astronomical Data Analysis Software and Systems II*, R.J. Hanisch, R.J.V. Brissenden, & J. Barnes, Ed., *Astronomical Society of the Pacific Conference Series* **52**, 315–320 (1993).

[3] C. M. Rockosi *et al.*, "The low-resolution imaging spectrograph red channel CCD upgrade: fully depleted, high-resistivity CCDs for Keck," *Techniques and Components I* **7735**(27), SPIE (2010).

[4] R. J. Stover, "MUSIC a Multi-User System for Instrument Control," *Lick Observatory Technical Reports* **54** (1989).

[5] M. Wei and R. J. Stover, "New design for the UCO/Lick observatory CCD guide camera," *Solid State Sensor Arrays and CCD Cameras* **2654**(1), 226–232, SPIE (1996).

[6] S. S. Vogt *et al.*, "HIRES: the high-resolution echelle spectrometer on the Keck 10-m Telescope," *Instrumentation in Astronomy VIII* **2198**(1), 362–375, SPIE (1994).

[7] S. S. Vogt, "The Lick Observatory Hamilton Echelle Spectrometer," *Astronomical Society of the Pacific* **99**, 1214–1228 (1987).

[8] D. Clarke and S. L. Allen, "Practical Applications of a Relational Database of FITS Keywords," in *Astronomical Data Analysis Software and Systems VI*, Gareth Hunt and H. E. Payne, Ed., *Astronomical Society of the Pacific Conference Series* **125** (1997).

[9] S. M. Faber *et al.*, "The DEIMOS spectrograph for the Keck II Telescope: integration and testing," *Instrument Design and Performance for Optical/Infrared Ground-based Telescopes* **4841**(1), 1657–1669, SPIE (2003).

[10] A. I. Sheinis *et al.*, "ESI, a New Keck Observatory Echellette Spectrograph and Imager," *Publications of the Astronomical Society of the Pacific* **114**(798), 851–865 (2002).

Kyle Lanclos is a member of the Scientific Programming Group at UCO/Lick Observatory. He received his bachelor's in computer science with a minor in astrophysics from the University of California at Santa Cruz in June 2000, and began working with UCO/Lick in December 2000.