Factorizing Monolithic Applications

The Tecolote Team


The POOMA Team


Los Alamos National Laboratory, Los Alamos, NM 87545, USA

Abstract

The Blanca project is part of the US Department of Energy’s (DOE) Accelerated Strategic Computing Initiative (ASCI), which focuses on Science-Based Stockpile Stewardship through the large-scale simulation of multi-physics, multi-dimensional problems. Blanca is the only Los Alamos National Laboratory (LANL)-based ASCI project that is written entirely in C++. Tecolote, a new framework used in developing Blanca physics codes, provides an infrastructure for gluing together any number of components; this framework is then used to create applications that encompass a wide variety of physics models, numerical solution options, and underlying data storage schemes. The advantage of this approach is that only the essential components for the given model need be activated at runtime. Tecolote has been designed for code re-use and to isolate the computer science mechanics from the physics aspects as much as possible — allowing physics model developers to write algorithms in a style quite similar to the underlying physics equations that govern the computational physics. This paper describes the advantages of component architectures and contrasts the Tecolote framework with Microsoft’s OLE and Apple’s OpenDoc. An actual factorization of a traditional monolithic application into its basic components is also described.

Code Reuse

The entire history of computer software design has been targeted towards reducing complexity inherent in computer applications by isolating a single piece of functionality and making it easier for the applications developer to concentrate on only that one construct. Over the years, the methodology for accomplishing this task has moved from isolating blocks of code and accessing them with goto statements, to subroutines, and currently to objects. The inherent claims for increases in reusability and programmer efficiency are well known and will not be rehashed here.
The desire to reuse previously written code has been dominant for decades, and various descriptions of possible techniques have become part of our programming lore. One area in which reuse was truly achieved involved the subroutine libraries for numerical algorithms like CLAMS, or BLAS. Subsequently, we progressed to the more organizationally complex toolboxes used in developing graphical user interfaces (GUIs) such as the Apple Macintosh toolbox or the Unix X-Window libraries. Even though the quantity of subroutines grew in the collections and reuse was improving, an organizing theme was still missing.

One organizing theme that has received a lot of attention is the idea of a framework. In “Design Patterns, [Gamma, 1994]” a framework is defined as a set of cooperating classes that make up a reusable design for a specific class of software. It dictates the architecture of your application, emphasizes design reuse over code reuse and finally inverts control between an application and the underlying software. Generally, an unmodified framework will compile into a do-nothing application. In object-oriented frameworks, the user adds functionality by overriding classes and methods, leaving the original framework unchanged.

There are alternatives to frameworks for achieving code reuse. One of the more important is the idea of component reuse. Instead of building a monolithic specific class of software using a framework, the programmer is free to build components that perform a single task and to combine them. Two examples are OpenDoc from Apple and Object Linking and Embedding (OLE) from Microsoft. A brief discussion of the history of components follows.

**Components**

Early in the development of Unix the concept of pipes was introduced. If properly applied, this single construct would chain together mini-applications (components) that perform only a single function to create complex functionality. Unfortunately, pipes never reached their full potential because the proliferation of components made learning Unix a complex endeavor. In addition the lack of Human Interface Guidelines meant that each component had a different interface, and memory constraints forced programmers to add functionality (feature creep) rather than develop new components with a simpler interface.

Despite these problems, the power of combining components by using pipes and scripting languages such as Perl, Python, or TCL is a major factor in making Unix workstations the preferred choice for scientific programmers. When programming we routinely find ourselves facing one-time tasks (such as a complex global regular expression substitution) across an entire project. Many Unix programmers would be undaunted by the prospect of combining the functionalities of find, xargs, grep, and awk to rapidly accomplish this task.

A system with a GUI has a problem in determining how to integrate scripting and pipes. This is especially true for systems in which there is no legacy commandline, such as the
DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
Apple Macintosh. Traditionally, application developers on such systems have relied on monolithic applications, which result in incredible feature creep. Despite attempts to standardize interfaces across applications, each implementation of the same feature in differing applications appeared—and worse, functioned—differently. If the application you were currently using didn’t perform the desired task, you either had to locate an application that did or do the work manually.

This is clearly unacceptable. Consequently, Apple and IBM, among others, began to design a cross-platform scripting language for GUIs. Like most cross-company collaborations, Script-X was designed and an initial implementation was complete before the project was scrapped. One positive effect of this and earlier scripting efforts was the promotion of factorization for monolithic applications. For scripting to work, it must have access to the basic functionality of an application. Earlier, it was common practice to write a “C” switch statement to handle menu selections. Many operations were so simple that the entire option would be handled by using inline code. To allow a scripting language access to these operations required moving to a slightly more complex scheme in which the switch statement operation after selection merely sent messages to “handler functions.” These handlers were then potentially available to both the GUI menu-handler code (the switch statement) and the script (although some mechanism for registering the entry point and argument list was required). This extra complexity is the main reason most Macintosh applications are not scriptable even today. Good design is difficult to achieve and even more difficult to retrofit.

The term describing this process is “factorizing,” which the dictionary defines as “to break a whole into its constituent parts (components).” Factorizing an application should be one of the overriding principles in software design—even when the desired result is a traditional monolithic application.

Because most good programmers experiment with many different platforms in the course of their careers, it was only a matter of time until someone realized that Unix had more than just scripting to offer. A concerted attempt to integrate cross-platform system-level support for components was initiated at Apple, and became known as OpenDoc. Many of the same issues that face scriptwriters immediately surfaced. The need for factorization was now elevated to the supreme issue. A debate raged about the appropriate level of granularity required before a component became too complex. Should a single component only perform one task, or should design follow the Unix model, in which feature creep led to increasingly complex components?

**The Advantages of Factorization**

Factorizing an application allows the software designer to compartmentalize a given functionality. This methodology is very similar to the idea of data encapsulation in object-oriented programming; it differs only in granularity. Often a factorized component consists solely of one object and there is a one-to-one correspondence. This is not guaranteed to be the case, however; many components consist of multiple objects. Extending the data encapsulation metaphor a little further gives us a little insight into
some expected properties of a component. First, a component should be completely self-
contained, that is, able to create or obtain a reference to any required storage, initialize
any persistent values, and clean up after itself properly. This set of requirements parallels
the construction, usage, and destruction phase of an object.

Second, a component should not exist, as far as the framework is concerned, until it is
registered. This is a slight extension of the data-encapsulation concept, but it allows some
additional security features. Suppose a company was using a component architecture and
they felt it appropriate to distribute the framework and certain nonproprietary parts to the
outside world (for example as a commercial release); however, other parts they
considered company proprietary (trade secrets). An overview component that presented a
summary of expected program-output and reserved space for the proprietary
information’s output might imply the existence of those other components and perhaps
even the methodology being used by those proprietary components. This exposure could
be avoided if the output component contained only generic knowledge of how to do its
job and a registration process existed, in which each component provided its summary
information for output. No registration could potentially occur for models that were not
provided to the user and therefore, no leak could occur. The Blanca project [Holian,
1997; Marshall, 1998] has demonstrated that registration of a component with the
framework need occur in only one place. By removing the component’s files and deleting
the few lines in the registration file, it is possible to remove all evidence of the existence
of that component.

A component’s interface should be designed to allow groups of components with the
same functionality to be interchangeable. This “plug and play” feature becomes more
important when moving from a monolithic application to a scripting environment.
Factorizing allows one to design a code by piecing together components, treating them as
first-class functions, and subsequently running the assembled application. This vision of
software programming has become known as the “Software IC” model, implying the
obvious analogy to hardware integrated circuits (ICs).

**Scripting and the Environment**

An examination of the popular scripting languages Perl, Python, and Tcl shows that more
than just simple glue for components is being provided: a complete environment has been
constructed in which the developer is expected to live. The resultant learning curve
explains some of religious fervor behind some scripting proponents. Once a programmer
in an environment establishes a certain level of comfort, interest in pursuing other,
possibly better, environments wanes.

Current scripting environments also have stark separation in state while inside the script’s
interpreter and while executing a compiled module. Errors detected in a compiled module
must be passed back through the shell to the script. Attempts to capture the state of a
scripted application are very dependent on whether control is local to the script or to a
compiled module. If the granularity of compiled modules is small enough, waiting until
the module returns before capturing the state does not incur a significant penalty. If the
granularity increases significantly, as is likely to occur in numerical simulations, then
waiting until the module returns may pose an unacceptable risk in the amount of work
that could be lost in the event of a failure. Unfortunately, to capture the state of a module
and the surrounding script requires two separate mechanisms.

Never has the issue of granularity been more prominent than in the battle between
Apple’s OpenDoc and Microsoft’s OLE. OLE was created specifically to allow third
party developers to extend Microsoft’s large monolithic applications. OpenDoc was
designed to allow developers to piece together small, single-purpose components that
could compete with large monolithic applications. Both models provided system-level
support to enhance the component developer’s experience, but neither model sufficiently
abstracted away the redundant services. The interfaces were consequently cluttered and
complicated even for skeleton, do-nothing components. Despite the impositions placed
on their developers, both component architectures have inspired the creation of
significant applications. The OLE/ActiveX market is now being touted as the first
commercially viable component-architecture market.

The Tecolote Environment

By making the tough decision to limit full participation to components written in C++,
the Tecolote framework has managed to considerably simplify the API. Registering a
component with the environment can occur in precisely one location. If a component
needs to inform the environment about which variables are to be saved to record the state,
a single table, local to the definition source-code file is created. Full participation in the
framework also requires that a component supply a specific constructor, an “initialize”
method, and a “call” method. Contrast Tecolote’s five requirements to the OpenDoc
“Parts” 66 methods and seemingly infinite caveats that had to be implemented or
complied with. OpenDoc also required the user to implement default behaviors for many
of these methods. For example, highlighting a selected part was left to the part implementer.
This is a service that could easily have been abstracted away. Each part was also
responsible for saving and restoring persistent state variables. Tecolote abstracts all I/O
into a single consistent implementation. A new output format such as HDF is added once
at the framework level, and all components transparently have immediate access.

Almost all of the services required by OpenDoc in its 66 methods could have been
implemented as system services with no loss in generality. The failure to do so meant that
the flagship product “CyberDog” left the one part-one action model and moved toward
parts with multiple functionality because small parts were too expensive in creation,
vtable allocation, and locatability.

The Tecolote framework is being designed by first examining the list of desired
functional behaviors and determining which can be abstracted. A behavior, once
abstracted, is immediately available to all components, both existing and planned. The
environment provides a scripting language that is based on the ideas promulgated by the
functional programming community. This scripting environment is completely integrated
with the component environment, sharing a common state at all times. Each new
component upon registration with the environment, becomes a first-class function (a keyword) along with all its associated persistents.

**Blanca**

![Diagram of Blanca Project with Tecolote and Pooma frameworks]

**Figure 1: Applications in the Blanca Project are layered upon the Tecolote and Pooma frameworks**

The Tecolote environment has promulgated a shift in the way the Blanca project (Figure 1) develops applications. In most projects, even very large ones, only a small number of people (usually one or two) actually work on adding a new feature. In Blanca, new features are invariably added as new components, which allows the development team to take full responsibility for creation, initialization and use of the component. Because a component is an isolated piece that fulfills an entire task, there is little concern about interactions with other components other than to minimize any side effects. In other words, a perfect component does not depend on any other and does not affect any other. This is necessary if we are to reach the ultimate goal of providing a hospitable environment for components that are implementing techniques that have not even been conceived yet.

In a traditional monolithic application, the same new feature requires development of a formal parameter-passing scheme or – worse – addition of a global variable. Such a new public interface has to be agreed upon by all the other developers and a contract is formed. Abrogation of this contract requires agreement by all involved and can have ramifications throughout the application.

The compartmentalization offered by components helps project managers break the organization into small teams of fewer than ten people. Despite this fragmentation and the independent design styles of each team, management can be assured that the final product will plug and play with all the other teams' contributions.
Factorizing an Existing Application

A recent opportunity developed when the code team for an existing traditional application decided to experiment with the "Tecolotization" of their code. The decision to combine an existing code into a complex, albeit feature-rich, environment does not come easily. To make the transition more manageable, the first effort was limited to integrating the entire application as a single component. This mainly involved introducing a wrapper class and replacing the existing parser with that from Tecolote. The immediate benefits realized from just these few steps included:

- gaining access to Tecolote's advanced and feature-rich makefile system
- accessing a sophisticated parser, which considerably reduces the coding overhead
- being able to use existing components designed for a completely different application

The existing components allowed critical input data to be generated on the fly rather than having to rely on output from other programs.

Despite these enormous gains, there were deficiencies in the original code that were now integrated into Tecolote rather than being removed by completing the process of factorization and use of the environment's additional features. The development team fortunately recognized the problems and began the process of complete factorization. The best plan of action would of course be a series of small changes that gradually end with the complete factorization of the application. At every point the application should compile and run correctly before further modifications are added.

The first problem involved removing the dependency on global data that was present in the original code. This step could be accomplished independent of the factorization step. Any component architecture requires a mechanism that allows intercomponent communication. Tecolote's mechanism for providing this service is the DataDirectory, which consists of a hierarchical collection of databases that are indexed on (name, type) pairs. This database provides a strongly typed access mechanism for accessing data on the heap. Using this new technology, the formal parameter list for a Tecolote component can consist solely of an entry point into the DataDirectory hierarchy. The need for global variables is then replaced by DataDirectory accessor functions.

The next decision required delineating how many models were contained in the application. This particular application had three models. The previous instantiation handled persistents for all three models simultaneously leading to a superset of flags and initial values. Division of the application into three models allowed grouping of the persistents for each model. This removed the need for many of the flags which had been used both to determine which model was to be run and how data was to be interpreted by the chosen model. The application designers had already done a good job of deciding how to initialize persistents so that the end-user need only supply deviations from the default values. This resulted in a smaller control script that is considerably easier to read.

Another simplification was that the designers had already adopted the Pooma framework [Reynders, 1996; Humphrey, 1997]. This meant they were already using the same
communications models as Tecolote. Changes in this area were limited to adopting some of Tecolote's simplifications for initializing Pooma Fields. The primary change involves creating TecFields instead of the Pooma Field. This allows usage of some simplified Field constructors that take advantage of a class unique to the Tecolote Framework known as the TecMesh. The TecMesh encapsulates all the data necessary to create a Field such as which boundary conditions to use, the number of guard cells in each direction, the field layout across multiple processors, and the number of cells in each direction for a given centering.

If the existing application had already implemented another communications model or parallel computing abstraction model, integration into the Tecolote framework would have required considerably more far-reaching changes. This could conceivably even impact the original design decisions regarding which algorithm would be optimal. Every decision from that point forward in the design would have to be reconsidered. This is not a consequence of factorization, but of abstraction of parallelism. The Pooma framework on which Tecolote is layered, allows the developer to program using a model which abstracts the actual details of parallelism. The choice of whether to use MPI or another communications library is then made at compile-time and is optimized for the destination hardware. Codes having their own explicit communications model can be integrated into the framework, but, they will be missing “Division of Labor,” one of the greatest benefits of the Tecolote framework.

The benefits of the “Division of Labor” model are enormous. The Tecolote team relies entirely on the Pooma framework to provide a data-parallel Fortran-90-like array syntax. Tecolote developers derive the benefit of being able to code using this beautiful syntax, while Pooma uses complicated C++ mechanisms, including expression templates, to deliver optimal implementation specific object code. Thread based implementations have been demonstrated to be preferable to data-parallel codes on the ASCI Blue Mountain SGI Origin 2000 supercomputers. The Pooma team has designed an abstraction layer for this hardware, which maps the data-parallel syntax used by developers to a thread-based implementation. The only impact to the Tecolote code developers is faster executables. Syntactic changes were limited to those arising from implementation of new features during the rewrite.

The final consideration in “Tecolotizing” this application was in moving from an explicitly 3-D code to a new implementation, which was developed using Tecolote’s mesh- and dimension-independent model. This often involves considerable effort since many models are being implemented in 1-D or 2-D for the very first time. Dimension independence is achieved without using #defines to demarcate implementation-specific, dimension-dependent details. The physics model developer only sees high-level code that appears remarkably similar to the actual mathematics equation that is being modeled.

In all, less than two man-months were required to factorize this application. The resulting models are now available in a stand-alone application and can be used internally by other codes built using the Tecolote Framework. The developers were able to concentrate on the physics unique to their models without having to maintain all of the computer science
boilerplate routines. The net result was only 30% of the original application was unique non-boilerplate code. The application developers had simplified their code tremendously and ended up with a cleaner, more powerful version due to their adoption of Tecolote.

References


This page intentionally left blank.