LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-JRNL-208636

Smart Libraries: Best SQE Practices for Libraries with an Emphasis on Scientific Computing

M. C. Miller, J. F. Reus, R. P. Matzke, Q. A. Koziol, A. P. Cheng

December 16, 2004

Proceedings of the Nuclear Explosives Code Developer's Conference 2004

Smart Libraries: Best SQE Practices for Libraries with an Emphasis on Scientific Computing (U)

Mark C. Miller*, James F. Reus*, Robb P. Matzke*
Quincey A. Koziol, Albert P. Cheng****

*Lawrence Livermore National Laboratory, Livermore, CA. 94550
**National Center for Supercomputing Applications, Urbana-Champaign, IL. 61820

As scientific computing applications grow in complexity, more and more functionality is being packaged in independently developed libraries. Worse, as the computing environments in which these applications run grow in complexity, it gets easier to make mistakes in building, installing and using libraries as well as the applications that depend on them. Unfortunately, SQA standards so far developed focus primarily on applications, not libraries. We show that SQA standards for libraries differ from applications in many respects. We introduce and describe a variety of practices aimed at minimizing the likelihood of making mistakes in using libraries and at maximizing users' ability to diagnose and correct them when they occur. We introduce the term Smart Library to refer to a library that is developed with these basic principles in mind. We draw upon specific examples from existing products we believe incorporate smart features: MPI, a parallel message passing library, and HDF5 and SAF, both of which are parallel I/O libraries supporting scientific computing applications. We conclude with a narrative of some real-world experiences in using smart libraries with Ale3d, VisIt and SAF. (U)

Introduction

It can be difficult to build, install and use a scientific computing application that has numerous and complex dependencies on third-party libraries. In Fig. 1, we have illustrated some of the libraries the Ale3d application can be linked with. The lines in the figure are meant to indicate one library's dependency on another. For example, many of the libraries Ale3d uses depend on the Message Passing Interface (MPI) library.

In using an application such as this, often, it can be difficult simply to get the application to compile and link. Next, although it may compile and link ok, it may not run

correctly. Worse, it could run correctly but produce incorrect results. Or, it could run correctly, produce correct results but run 2-3x more slowly than necessary.

None of these issues should be seen as a flaw in Ale3d. On the contrary, a strength of Ale3d is its ability to take advantage of capabilities available in independently developed libraries. Nonetheless, the more libraries an application uses and in particular the more interdependencies they have, the more challenging it can be to get them all to play well together. Often, many problems such as these are a direct result of mistakes made in compiling and/or linking to the various libraries the application needs.

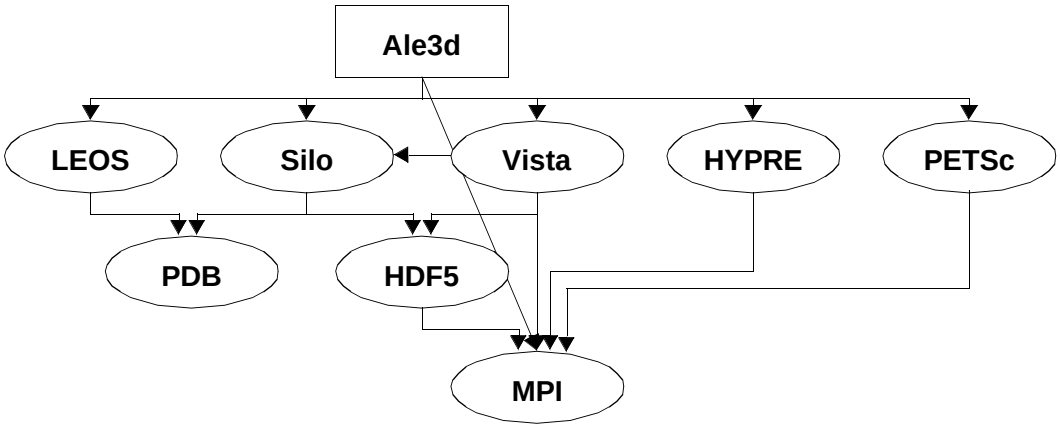

Fig. 1. Some of the libraries Ale3d can be linked with and their interdependencies

In this paper we introduce and describe a variety of Software Quality Engineering (SQE) practices aimed at minimizing the likelihood of making mistakes in using libraries and at maximizing users' ability to diagnose and correct them when they occur. This paper begins with an overview of key differences in SQA standards for libraries and applications. It then introduces and describes several SQE practices under the broad theme of *Smart Libraries*. A smart library utilizes a variety of SQE practices aimed at keeping users and library developers alike from making various mistakes; mistakes in usage, in documentation, in configuration and installation, in dealing with API changes, mistakes leading to performance degradation and/or parallel deadlock and so on. To be sure, making a library smart is not merely a matter of rigorous error checking. While that is part of it, a smart library is also flexible both in how much work it does to detect mistakes and in how much it enables users to diagnose and correct them.

Throughout this paper, we use the term *"Smart Library"* to refer to a library that is developed with these principles in mind. In addition, we use the term *"Client"* to refer to an application that makes use of a library. Finally, when you see the term *"User"* it is important to remember we are referring to a user of a library. We will use the term *"End-User"* when we need to refer to the user of an application itself.

Differences in SQA Standards For Libraries and End-User Applications

Portability, configuration and installation standards are more stringent for libraries than applications because users care about how a library is compiled and installed. For example, users of a library care whether it was compiled for 32 or 64 bit word size, with debugging or not, optimized or not, with support for parallel execution or not, with support for static or dynamic linking, etc.

Furthermore, a user's interest in how a library is compiled and installed extends, recursively, to any other libraries upon which the given library depends. Often, there can be multiple versions of an "other" library and even multiple implementations. For example, there are multiple versions of MPICH, the Argonne portable implementation of the MPI specification. Nonetheless, MPICH is just one implementation of the MPI specification. There are others including LAM, ChaMPIon and vendor specific implementations such as IBM's or SGI's. Obviously, users care about which of version and/or implementation of an "other" library the given library is compiled with.

To assure consistency in name mangling, exception handling and long-jumping, when multiple compilers are available users care which compiler is used. Even with a given compiler, say gcc, there are often multiple versions available causing user's to care about which version is used. In cases where compilers support exotic code-generation options[1], users care which options are used and how.

If for all of the aforementioned options just two choices were available, the example cases we've enumerated here, which are not exhaustive, demonstrate 256 different ways to build and install a single library on a given platform! We call each a *configuration*. SQA standards for libraries involve testing, installing and supporting a very large number of configurations.

To make matters worse, mistakes in configuration are not always detectable at compile-time. For example, how would anyone know that an installed library that was said to be compiled with optimization was accidentally compiled without? Other than lower performance, there would be no indication that something was wrong. Over many, many executions of the client, this might put a substantial and unnecessary drain on computing resources. In other cases, the consequences and cost of a configuration mistake can be more dramatic. For example, in the case where both the client and the library, say `libCool`, both depend on another library, say `libOther`, if the client and `libCool` were compiled with different versions of `libOther`, the client may fail in unexplainable ways triggering an unnecessary debugging effort.

Another difference between SQA standards for libraries an applications is that libraries are entirely unforgiving of application programming interface (API) changes. When the API of a library is changed, clients can cease to function! On the other hand, when the interface of an application is changed, the graphical user interface (GUI) for

1. These are options that effect link- and/or run-time compatibility of compilation units.

example, users may complain about the change but the product will continue to function.

Next, when critical bugs are found and fixed in a library, often they must be fixed in multiple older versions of the library and new releases of these older versions must be made. The same is not true of applications. Why? Libraries are often used in applications with unyielding stability requirements where reproducibility of results over a periods of years is critical. Such applications cannot tolerate having to change to a newer version of a library just to obtain a critical bug fix to the version they are using. Consequently, it becomes necessary for library developers to apply bug fixes to older versions and re-release the bug-fixed older versions. The same cannot be said for applications.

As a final difference in SQA standards for libraries and applications, documentation for libraries is much more technically rich and complex than it is for applications. Furthermore, such documentation is often essential for proper use of the product. By contrast, a key goal of the design of a GUI for an application is to make the interface of an application as intuitive and free from the need for documentation as possible.

In this section, we have outlined many of the important differences in SQA standards for libraries and applications. In the next few sections, we introduce and describe several SQE practices for smart libraries under four broad categories, progenitor practices or those upon which all other practices depend, those that enable clients to deal with change, those that enable clients to detect and diagnose anomalies and, lastly, those aimed at producing quality documentation. In the next to last section we describe several miscellaneous practices.

We include example code for a few of the key practices. Likewise, we've chosen not to discuss practices that we considered to already be in common use such as the use of a revision control system, like CVS, or a configuration tool like Autoconf. We conclude with a narrative of some real-world experiences and how smart libraries have demonstrated big advantages.

Progenitor Practices: Key Practices Upon Which All Others Depend

In this section we describe those practices that are essential in order to implement many of the other practices we describe. We call these *Progenitor Practices.*

Practice 1: Give a meaningful version number to each release of the library

Libraries are often developed and released in increments. Features are added. Bugs are found and fixed. API's are changed and new releases are made. Typically, a library is never really done. It continues to undergo development and evolve even as clients use it. This creates a need for users to be able to identify and distinguish between various versions. We call such identification a *Version Number* and the means by which a version number is assigned to a release a *Version Numbering Scheme.*

A simple way to assign a version number is just to increment a counter each time a release of the library is made to customers. Or, one could use the date of the release as a

version number. For a ridiculous example, one could compute a checksum over all the source files that comprise the library and assign that as the version number. However, none of these version numbering schemes is very useful. Certainly, each scheme permits a user to distinguish one version from another. But, that is about all they are good for with the exception that the counter and date schemes also permit a user to distinguish a newer release from an older one. The checksum scheme doesn't even permit that.

The reason for mentioning these not-so-useful schemes is to underscore the importance of having a version numbering scheme that lends meaning to the version number. A version number can be given meaning by defining for users how changes in the version number relate to events in the software development life-cycle of the library. For example, one digit in the version number could be set aside to indicate bug-fix changes only. That way, if version numbers from different releases match in all other respects except for this digit, a user could know if two versions are link-time compatible. However, this is just one example of giving a meaningful version number to each release.

A smart library utilizes a version numbering scheme that gives meaning to the version number by relating events in the software development life-cycle of the library that are relevant to users to changes in the version number.

Practice 2: Require clients to call functions to begin/end interaction with the library

Libraries can contain a large number of API functions. For example, the MPI library contains over 100 different API functions. If the library is designed such that there is no API function that is culled out and required to be the *first* function that must be called before any other function in the library can be called, then any of the API functions in the library might wind up being the first the client makes during actual execution.

In other words, without requiring clients to call functions that begin and end interaction with a library, library developers have no way of knowing or predicting the path by which clients will enter their library nor when a client is finished using it. This is problematic for managing global settings and resources. It is also problematic for several of the practices we discuss in later sections in this paper.

An example of this practice can be found in the MPI library. The MPI library requires that the client first call `MPI_Init()` before calling any other function in the library and also to call `MPI_Finalize()` when the client is done using the library.

A smart library requires clients to call functions that begin and end interaction with the library. A smart library also includes functions to query if the library has been initialized or finalized.

Practice 3: Design all public functions to invoke common entrance/exit procedures

When a client makes a call into one of its libraries, the client is temporarily transferring control of execution to the library so that the library can perform some service on behalf of the client. This transfer of control is a significant event in the execution of the

client. It make sense that the process by which the library takes control from the client and later returns control to the client be managed formally and be consistent across the entire API of the library.

By managing this transfer of control effectively, there are many services a library can offer. For example, to enable a client to gather some simple performance data, the library might record the time of entrance and exit to each function. Or, on entrance the library might check that is has been initialized with a prior call to its initialization function (Practice 2). Or, the library might record the name of the function so that it can be included in any warning or error messages printed by the library. Another useful application of this practice might be to provide API call tracing. On entrance, each API function can print its name and maybe calling arguments. On exit, it can print the fact that it has successfully returned from the function. Another possibility might be to provide access control for a multi-threaded implementation. Yet another possible application of this practice is to provide an integration point for external instrumentation tools such as Vampir or Pablo.

The point in describing some of these possibilities is that they can be supported only if every function has been designed from the start with this in mind by adding common entrance and exit procedures. An example is illustrated in Fig. 2.

```
int saf_declare_set(SAF_Db db, const char *name, int topo_dim)
{
    int reval = SAF_ERROR;
    SAF_ENTER(saf_declare_set, SAF_PRECONDITION_ERROR);

    /* do the work here */

    SAF_LEAVE(retval);
}
```

Fig. 2. Example code for common entrance/exit procedures

Our experience has been that the macro pre-processor offers the greatest flexibility in controlling the entrance and exit behavior of each function. That is, entrance and exit procedures have their greatest utility when they are expressed as macro'd code-blocks. A consequence of this practice is that early returns from functions cannot be allowed to bypass the normal exit procedures. This can lead to a design where every function exits *through the bottom* often by using much frowned upon but ideally suited for this purpose `goto` statements.

Practice 4: Put all public symbols of the API in their own namespace

In Fig. 1, we show 8 different libraries being linked together into the Ale3d application. This is just a subset of all the libraries Ale3d can be linked with. If each library is developed independently, how can we make sure that public symbols defined in one library do not have the same name as symbols defined in another?

The solution is to put all public symbols of the API in their own namespace. If the client is written in C++, this is relatively easy for the client to do using the namespace features of C++. In general, however, this cannot be achieved by the client alone. Each library must have been implemented with this issue in mind.

The solution is to prepend a 2,3 or 4 character acronym to *every* public symbol in the API. Most developers try to adhere to the this practice. However, almost as many neglect to consider *all* possible symbols and wind up missing some. Developers must remember to consider functions and all global data structures even if those data structures are not directly touched by the client. It includes type definitions, pre-processor macros and enums. It also includes the names of header (.h) files, as well as environment variables and/or command-line arguments, if any, that the library interprets. For example, the HDF5 library defines these symbols...

- `H5Fopen` - function to open a file
- `H5T_NATIVE_INT` - constant for type of native integer
- `H5G_stat_t` - datatype for stat of an HDF5 object
- `H5_VERS_MAJOR` - macro for major version number
- `H5_DEBUG` - environment variable controlling debugging/tracing

In a smart library, every public symbol has prepended to it a 2,3 or 4 character moniker to help keep its symbols from colliding with those from other libraries.

Practice 5: Summary of progenitor practices

In this section, we have described four practices that are essential in order to implement many of the other practices described here. In the next section, we will describe practices aimed at enabling clients to deal with change in a library.

Practices that Enable Clients (and Users) to Deal With Change

As we mentioned previously, a library is never really done. A library continues to undergo development and evolve even as clients use it. Enabling clients to deal with subsequent changes is one of the more important services a smart library provides. In this section, we discuss a variety of SQE practices aimed and providing this kind of service.

Practice 6: Give meaning to the version number in terms of changes' impact on client

Ideally, a library can be improved in arbitrary ways without negatively impacting any applications that depend on it and, for I/O libraries, without negatively impacting any data files generated by it. Nonetheless, this isn't always practical or desirable[1]. When a library is changed, the resulting impact on an application can be minor to severe. The application may simply need to be re-built[2] with the newer library. For more substantial changes, new library calls may need to be added to the client or existing calls re-written[3]. In the most

1. For example, when an application developer works around a bug in a library, once a new version of the library with the bug fixed is available, the work around should be removed from the application.
2. By "re-built" we mean either a re-link or a re-compile and re-link.

severe situations, whole algorithms may have to be re-thought. In a nutshell, to use a new version of a library the user may have to re-build, re-type or re-think portions of the application.

A smart library employs a carefully designed version numbering scheme that helps users make an initial assessment of what is involved in moving to a newer version of a library. A scheme used by SAF is composed of 3 digits, A.B.C, where A is the *major* digit, B is the *minor* digit and C is the *patch* digit. The digits are given meaning by defining the events in the software development life-cycle that trigger a change in a particular digit. This is summarized in the table below.

Table 1: Meanings for various digits in a version number

	A	B	C
	Major Digit	Minor Digit[a]	Patch Digit
In the worst case, changes in this digit *can* mean...	*Everything Minor means &...* Major API changes Major feature enhancements Major file format changes[a]	*Everything Patch means &...* Minor API changes Minor feature enhancements Minor file format changes[b] Performance improvements[e]	Documentation updates Bug fixes API additions Performance improvements[d]
impact on application when digit changes	re-type <= impact <= re-think	rebuild <= impact <= re-type	none <= impact <= rebuild
typical frequency[c]	years	months	weeks

a. Another common practice is an odd/even minor digit to indicate development/production releases.
b. File format issues are specific to I/O libraries. d. High-impact/low-cost. e. Lower-impact/higher-cost.
c. Our experience has been that increment of the release number is often triggered at regular intervals by routine bug-fix work while increment of minor and major number is triggered as planned development activities are completed.

Given this version numbering scheme, it is important to note that a difference in minor and/or major digits is a sufficient but not *always* necessary indicator of link-time incompatibilities.

Practice 7: Provide a version number consistency check between client and library

The client and library are compiled independently. As versions of a library get released and installed, over time various version may exist on a given system. It can become easy to accidentally compile with the header files for one version but link to the object files for a different version. This problem is particularly prevalent for clients that depend directly and indirectly on one library through one or more other libraries. In this situation, although the client may be compiling and linking with the correct header and object files, it may be that one of the other libraries the client is linking with expects a different

3. For libraries written in languages that support polymorphism (e.g. C++ or Java), changes in existing API calls which result in a different function signature are often gracefully handled by the compiler.

version of the given library.

A smart library checks consistency in the version number of the header file(s) a client is compiled with and the object files it is later linked to. There are run-time and link-time approaches to performing this check.

To perform a link-time check, the approach is to define an external symbol in the header file(s) that has a high probability of being referenced by a client. The external reference to this symbol is resolved only when the client is linked with the correct version of the object files. For example, for version 1.2.4 of the MPI library, one could define `MPI_Init()` to be a macro like so

```
#define MPI_Init(Argc,Argv)   (MPI_Version_1_2_4++,mpi_init(Argc, Argv))
```

where `mpi_init()` is the real implementation of `MPI_Init()`. Since all MPI clients must call `MPI_Init()`, each time one is compiled, it will make a reference to the symbol, `MPI_Version_1_2_4`. When a client is linked to the object files for the library, if they do not resolve the reference to this symbol, the linker will issue a fatal error to the effect that "`MPI_Version_1_2_4` is an unresolved symbol". The symbol name is appropriately chosen so that when it is printed in the linker error message, it will indicate the version number of the library the client is expecting to be linked with.

To perform the check at run-time rather than link-time, instead of defining a version specific compile-time symbol, one can pass version information in the initialization call like so

```
#define MPI_Init(Argc,Argv)   mpi_init(Argc,Argv,Maj,Min,Pat)
```

The implementation of `mpi_init()` will then compare the version string with what it thinks is the correct value and either permit the client to proceed or abort with an appropriate error message. The next practice explains why a run-time approach is the preferred approach.

Practice 8: Permit users to override the version check

Recall that differences in the version number are a sufficient but not always necessary indicator of version incompatibilities. The fact is several different versions may be compatible in spite of differences in the minor and/or major version digits. Even if two version are not wholly API compatible, they may be for the subset of API calls for a given application. If strict version number matching is used, (Practice 7), the client may be forced to re-compile when it is not necessary.

A smart library provides an option to override the version check via an environment variable. For example, in versions of HDF5 after 1.4.2, if the environment variable, `HDF5_DISABLE_VERSION_CHECK`, is defined, HDF5 will ignore version number mismatches. Of course, library developers need make no promise that things will operate correctly or to provide support if and when they do not.

Practice 9: Automatically build into the library its version compatibility history

A better solution to the problem of a sufficient but not always necessary version check is for the library to automatically determine and maintain knowledge of its version compatibility history and to use this knowledge when performing its version check (Practice 7). Note that this practice depends on a relatively complete and orthogonal test suite which is also updated to test new features as they are added.

In the development of a smart library, the test suite is used to probe version compatibility. Each time the minor and/or major version digits change, the test suite is compiled and run in the following ways

a) compile old test suite with old lib header files and link to new lib object files
b) compile new test suite with new lib header files and link to old object files

Case a) simulates an old client linking with the new library while case b) simulates a new client linking with the old library. Assuming an orthogonal test suite, tests which exercise new functionality are expected to fail in the link-phase in case b. These are excluded from the new/old compatibility decision because we can safely assume the linker will always fail for any new client using features of the new library but attempting to link to an old library.

For all other situations, failures may occur at link-time or at run-time. Nonetheless, anything less than a completely successfully run test suite is deemed a failure with the caveat that tests expected to fail in case b) are excluded. Pass/fail results for each case are then accumulated and maintained both in the header file and the object file for the library.

```
struct _verCompatInfo {
      int oldmaj, oldmin, newmaj, newmin, compat;
} SAF_VerCompatInfo_t;

SAF_VerCompatInfo_t SAF_VerCompat[] = {
      {1,   0,    1,    1,    1}, // 1.0 & 1.1 OK
      {1,   1,    1,    2,    0}, // 1.1 & 1.2 NOT COMPATIBLE
      {1,   2,    1,    3,    1}};// 1.2 & 1.3 OK
```

The reason for encoding this information into both the header file and an object module within the library is so that we can always guarantee the newest information is available in any given combination of old and new headers and object files.

Practice 10: Provide compile-time symbols client can query for version information

As new functions are added to a library, creating a client that depends on those functions means that said client cannot operate with older versions of the library. This isn't always desirable. To make a client that can be correctly compiled with different versions of the library, it is necessary for the client to conditionally compile code blocks that use newer API functions. However, the client cannot achieve this by itself. It needs help from the library.

A smart library provides compile time constants for its clients to query the version

number. For example,

```
#define H5_V_MAJOR      1
#define H5_V_MINOR      4
#define H5_V_PATCH      2
#define H5_V_GE(A,B,C)\
      ((H5_V_MAJOR==A && H5_V_MINOR==B && H5_V_PATCH>=C) || \
       (H5_V_MAJOR==A && H5_V_MINOR>=B) || \
       (H5_V_MAJOR>=A)) \
```

Suppose a new API function, `H5Pfoo_bar()` was added to the API in version 1.6.3 of the library. A client wishing to use this function but also to be backwards compatible with older versions of the HDF5 library, would be coded like so

```
#if H5_V_GE(1,6,3)
      H5Pfoo_bar(); /* do it the new way */
#else
      H5Pgorfo();   /* do it the old way */
#endif
```

A consequence of this practice is a requirement that the compile-time representation for the version number support not only the `==` operator but also the `>=` and `<=` operators. A string representation for compile-time version information is not acceptable.

Practice 11: Embed string-oriented version information in the library and its clients

Given an application executable file that uses a library, how can users learn which version of the library the executable uses? If library developers left this up to application developers, then probably very few applications, if any, would provide a means for end-users to obtain information about which version of a given library an application uses. What can library developers due to guarentee users can obtain library version information from an application?

The answer is that library developers can embed a string-oriented representation of the version number of the library in any client that uses the library. When this is done, something like the `strings` command on Unix can be used to retrieve version information directly from object and executable files.

This can be easily achieved by augmenting the method (marco) used to initialize the library with a reference to a string representation of the version number such as...

```
#define MPI_Init(Argc,Argv)   \
                              {static char *junk=”MPI library version” \
                               #Maj “.” #Min “.” #Min ; } ; \
                              mpi_init(Argc,Argv,Maj,Min,Pat)
```

In this example we use the C pre-processor’s # operator which turns a reference to a macro into the string representation of the macro along with the C compiler’s ability to catenate literal string tokens together into a single, large string.

With this expression for `MPI_Init`, any client that uses MPI will wind up having

embedded in it a string of the form "`MPI library version A.B.C`" where A, B and C are the major, minor and patch version number digits. By using a unix command such as

```
strings <file> | grep 'MPI library version'
```

where `<file>` is an executable or object file, a user con obtain the version of the MPI library an application executable or object file is actually compiled to use. On a Window's system, a similar operation is possible using the Window's search tools.

A smart library embeds a string-oriented representation for its version number into any client the library is compiled with.

Practice 12: Where possible, pass only language-built-in data-types through the API

The significance of the API of a library is that it represents the boundary between pieces of software (e.g. the library and its clients) that are compiled independently from one another. Any data types that flow through this boundary are shared between the library and the client. Because the client and the library are compiled independently, it is possible for each to have a different interpretation of these shared types. For example, suppose a library had the following member function in its API for initializing the library

```
typedef struct {char c; int i} props_t;
libInit(props_t props);
```

In this example, we have chosen the type `props_t` to be such that alignment of the `i` data member will necessitate 3 pad bytes following the `c` data member. If the library is compiled using default compiler options but some client is later compiled using options that allow these pad bytes to be compressed out, the library and the client won't agree on their respective interpretation of this shared type.

The kinds of data-types that are used in the API have a direct impact on the likelihood of this occurring. If the API is restricted to use only the built-in data-types of the implementation language, then the likelihood of the library and client being compiled in such a way as to have differing interpretations of the types is minimized.

When types more abstract than those built-in to the implementation language are needed, a smart library uses only pointers to these types and provides methods in the library to allocate, manipulate and free them. In other words, to use the `props_t` type, above, a better practice is

```
typedef props_t* props_p;
props_p libCreateProps();
libFreeProps(props_p props);
libPropsSetC(props_p, char c);
libPropsSetI(props_p, int i);
libInit(props_p props);
```

In this way, it becomes impossible for the client to allocate a type, in this case a props_t type, the library is responsible for defining. The only thing the client can do is to

obtain a pointer to the type through appropriate creation/destruction functions and then manipulate its contents through appropriate accessor functions. The only data-type that both the client and library share is the pointer to the type.

Practice 13: Provide data-type size and alignment consistency checks

Many compilers provide a variety of code generation options many of which, if not consistent between client and library, can cause incompatibilities in API data type sizes and alignments. Worse, linkers do not necessarily catch these incompatibilities and refuse to produce an executable. The problem may only manifest itself at run-time.

A smart library performs consistency checks on API types. The approach is to include a small static array of ints in the public header file of the library whose contents are computed at both the compile time of the library and the client. At run time, just as the library checks its version number, (Practice 7), it can also check this array for differences. In the example below, this information is encoded into `SAF_Types[]`.

```
/* datatypes used in API */
typedef struct { int tag; char bits } Foo_t;
typedef struct { double speed; int id; } Bar_t;

struct { char a; Foo_t b; } withFoo;
struct { char a; Bar_t b; } withBar;
#define ALIGN(A)        ((int) ((char*)&(A.b)-(char*)&A))
static const int SAF_Types[] = {
   sizeof(Foo_t), ALIGN(withFoo),
   sizeof(Bar_t), ALIGN(withBar)
};
```

Practice 14: In languages that don’t support polymorphism, fake it

When a library is implemented in a language that supports polymorphism, API changes are less of an issue because the compiler recognizes different versions of the same function based on the function signature. This enables library developers to maintain multiple different APIs for a given function without necessarily impacting any clients. However, not all languages support polymorphism.

When developed in a language that does not support polymorphism, the API of a smart library is designed in such a way as to mimic polymorphism. An approach used in HDF5 is to include a single catch-all argument representing “any additional arguments.” This catch-all argument is a reference to a list of parameter/value pairs called a *property list*. For example, in HDF5’s API function to create a dataset

```
hid_t H5Dcreate(hid_t loc, char *name, hid_t type, hid_t space,
                  hid_t plist);
```

the `plist` is a reference to a data structure that represents a list of additional arguments. As HDF5 evolves and new parameters are needed to affect the creation of datasets, functions to manipulate the contents of `plist` are added. For example, after

HDF5 was designed, the concept of a *fill value* was added to define values in the dataset that were not explicitly written by the client. A new function was added like so...

```
herr_t H5Pset_fill_value(hid_t plist, hid_t type, void *value);
```

The key word in the preceding sentence is *added*. By including the `plist` argument as a stand-in for any additional arguments in the initial design, the API for creating datasets can be changed without breaking version compatibility for existing clients.

Practice 15: Design API changes to break the client's compile

Inevitably, no matter how much time one spends in design, it may still become necessary to change an API that clients are already using. When such changes add or remove arguments from calls, the calls that need to be changed are easily detected and noted by the compiler where they can be corrected by the user.

However, when such changes change the data types flowing between client and library, the compiler won't necessarily detect them as outright errors. In some cases, it may not even generate warnings but could lead to various anomalous behavior during run-time and be difficult to diagnose.

In a smart library, as API changes are made, they are made in such a way as to always break the compilation phase of a client.

A great way to force a compile-time error is simply to change the name of a public symbol the client calls and eliminate the old name. This makes the changes easy to find and avoids unpredictable run-time behavior that may take hours in a debugger to track down.

Alternatively, the old name can be maintained in a *deprecated* state for a few successive releases of the library. When user's reference a deprecated symbol, a smart library generates a warning message -- which can be disabled -- to inform users of the eventual total removal of the symbol from the library.

Practice 16: Make work-arounds conditionally compiled and off by default.

Often, one library uses other libraries. When bugs are encountered in other libraries, they are often handled by work-arounds. A work-around is a block of code that represents an alternative way of achieving the same end. Of course, if there is no good reason to prefer one approach over another, then one cannot be considered a work-around. So, by definition, a work-around is an inferior approach. Eventually, it should be removed from the code and replaced with the preferred approach. Unfortunately, if work-arounds are not managed carefully, it is easy to forget about them among tens of thousands of lines of code. For this reason, work-arounds must be managed carefully.

For example, early versions of MPICH on Sun, Solaris had a bug in which `MPI_Type_struct()` would compute the wrong size for its data type. The code segment below illustrates a work-around for defining the `foo_t` data type in MPICH and returning

a handle to it in `mpiType`.

```
typedef struct _foo_t { float f; char c; } foo_t;
MPI_Datatype mpiType;

#ifdef SAF_WA_mpi_type_struct
      // the work-around
      MPI_Type_contiguous(sizeof(foo_t), MPI_CHAR, &mpiType);
#else
      // the preferred approach
      int           lens[] = {1, 1};
      MPI_Aint     disps[] = {0, sizeof(float)};
      MPI_Datatype types[] = {MPI_FLOAT, MPI_CHAR};
      MPI_Type_struct(2, lens, disps, types, &mpiType);
#endif
      MPI_Type_commit(mpiType);
```

The code in the `#ifdef` clause is sub-optimal because the resulting type is opaque to MPICH. MPICH won't know that the type is really a struct comprised of a float and char data member. This could lead to problems in a heterogeneous computing environment where MPICH may need to convert between different machine's floating point representations when passing messages involving the `foo_t` data type. Thus, the `#else` clause is the preferred approach. However, it relies upon a call to `MPI_Type_struct` which has a known bug.

In a smart library, work-around code is always conditionally compiled. Furthermore, it is turned off by default. This necessitates having to always pass conditional compilation symbols to the compiler to build the library with the work-arounds. Thus, the need for a given work-around is always "in your face" when building and installing the library and it is harder to overlook. In the example code above, to compile the code with the work-around, it is necessary to pass '`-DSAF_WA_mpi_type_struct`' to the compiler. A good practice in makefiles is to do the following...

```
CWAFLAGS="-DSAF_WA_mpi_type_struct ..."
CFLAGS="$CFLAGS $CWAFLAGS"
```

Practice 17: Create tests that independently test the need for work-arounds

Eventually, work-arounds should be removed from a library. A suitably designed test suite can automatically inform developers when a given work-around is no longer necessary.

In the development of a smart library, at the same time a work-around is introduced, a test is added to independently test the need for the work-around. By "independently", we mean the test is designed in such a way that its outcome is not dependent on the presence (or absence) of other work-arounds.

As long as the work-around is required, the test will pass only when the specific work-around it tests is conditionally compiled into the library and fail otherwise.

The indication that the work-around is no longer necessary comes when the library is compiled without the work-around and the test still passes. Consequently, as updates are made to other libraries, a smart library is periodically compiled and tested without its work-arounds to find out which work-arounds are no longer needed.

Practice 18: Provide a means for users to easily identify installed configurations

In the introduction we discussed a number of configuration issues to be concerned with. On any given platform, multiple configurations satisfying a variety of different users' needs may exist. This means, for example, there might be several different `libhdf5.a` libraries for an HDF5 client to link to. How are users to distinguish between them and/or find a configuration they want?

A smart library uses one of a couple of different approaches to help users identify various configurations. The simplest uses the pathname of the `.a` file to identify key configuration options. For example, the following `libsaf.a` file

```
/usr/gapps/saf/saf-1.2.0/IRIX64/gcc-2.96/default/debug/shared/libsaf.a
```

represents version 1.2.0 of the SAF library, compiled on SGI, IRIX 6.4 operating system, with version 2.96 of gcc, using default code generation options, with debugging support and with dynamic linking support (e.g. shared library).

In addition, because it is not practical to capture all the configuration information in the installation directory's pathname, another practice a smart library employs is to create a human readable text file in the directory where the object modules are placed which describes every detail of the configuration. For example, all versions of HDF5 after 1.2.0 include a file called `libhdf5.settings` in the same directory as the `libhdf5.a` (or `libhdf5.so`) which contains all the details about how the library was compiled and installed including the path to the compiler, all the flags to the compiler, flags to the library, etc.

One issue some users have with the preceding scheme is the long pathnames that result. So, an alternative is to flatten out everything after the version number. For example, in the directory `/usr/gapps/saf/saf-1.2.0`, we would have a set of arbitrarily named `.a` files, say `saf_eaRT1b.a`, `saf_ghJK2.a`, and `saf_bbbU7.a`. Since there is no way for a user to determine which `.a` file corresponds to which configuration, this approach also requires a shell-script utility which accepts the desired configuration options as input and then returns the appropriate file name as in...

```
% /usr/gapps/saf/saf-1.2.2/getconfig IRIX64 gcc-2.96 debug shared
/usr/gapps/visit/saf/saf-1.2.2/saf_ghJK2.a
```

The `getconfig` script can be controlled such that either an exact match or the closest link-compatible match is returned. A user can then do the following in his or her makefile

```
SAFLIB=‘/usr/gapps/saf/saf-1.2.2/getconfig IRIX64 gcc-2.96 debug shared‘
SAFINC=/usr/gapps/saf/saf-1.2.2/include
```

Summary of Practices Aimed at Enabling Clients to Deal with Change

In the preceding section, we described several specific practices aimed at enabling clients to deal with change. The practices are primarily aimed at keeping clients from making simple mistakes in the presence of change. In the next section, we discuss a number of practices aimed at helping clients to detect and diagnose anomalous behavior.

Practices that Help Users to Detect and Diagnose Anomalies

In this section, we describe practices aimed at helping users find and correct *anomalies*. We use the term *anomaly* to describe any condition that prevents proper execution. An anomaly may be the result of lack of resources, memory or disk-space for example, improper calling usage on the part of the client, an outright bug in the library, or a failure of some other software component on which the library depends. In our definition, "any" really does mean *any* condition preventing proper execution.

A smart library includes extra code the sole purpose of which is simply to detect commonly encountered anomalies. We call this *anomaly detection* code. On the one hand, anomaly detection code can degrade performance and increase the memory footprint of the library. On the other hand, it can provide valuable diagnostics to a user.

In this section we describe a variety of practices aimed detecting anomalies and providing detailed information to facilitate their diagnosis. Note that if a library does not catch an anomalous condition, often the operating system probably will, typically in the form of some catastrophic error such as a segmentation violation or bus error. This is called an *un-caught* anomaly. However, the point at which the operating system catches such a problem is often logically unrelated to the root cause. Worse, the operating system provides no information to aid in diagnosing the root cause.

Practice 19: Provide at least two quality of service options; debug and production

Many of the practices discussed in this section depend on the user's ability to obtain various qualities of service from a given library. In a smart library, at least two are essential. One is to provide the user with maximum development support while s/he is developing the client. The other is to provide the user with production-level performance once development is completed. We call these *debug* and *production* configurations respectively. A smart library provides at least these two qualities of service.

Note that in both a debug and a production configuration, the library is compiled such that anomaly detection code is also compiled. In a debug configuration, anomaly detection code is enabled by default and a user has to explicitly disable it. In a production configuration, anomaly detection is disabled by default and a user has to explicitly enable it. (see Practice 20). The key point in both configurations is that anomaly detection code is compiled and available in the library. Between a debug and production configuration, only the default state is different.

Additionally, a debug configuration is compiled with the `-g` option. This isn't because

we expect users of our library to debug it. Instead, it is because users are often more successful debugging their own client code when they are also able to see what is happening in the library. Nonetheless, an added benefit is that power users are sometimes able to find and subsequently work around bugs in the library. On the other hand, a production configuration is compiled without the `-g` option and with the `-O3` (or whatever level of optimization the code permits) option.

Finally, although a production configuration is usually sufficiently optimized, it is often useful to provide a third quality of service option in which the library is compiled so as to minimize its memory footprint as well. In this case, the library is compiled as in a production configuration except without any anomaly detection code. Such a configuration is also called an *optimal* configuration.

Since aspects of a given quality of service are often determined at compile time, this practice usually means that multiple different versions of a library need to be compiled and installed on a given platform (Practice 18).

Practice 20: Make anomaly detection code blocks run-time switchable

The user doesn't necessarily want the library performing anomaly detection all the time. Users need to be able to easily disable anomaly detection. On the other hand, whenever users encounter an un-caught anomaly, they often like be able to easily re-run the client with anomaly detection enabled to get more information about where things are going wrong.

The keyword in the preceding paragraph is "easily". If the presence of anomaly detection support in a given library is determined solely at compile time, a client needs to be re-linked, maybe even re-compiled and re-linked, in order to enable or disable it.

In a smart library, anomaly detection is switchable at run-time. Skeptics of this approach might argue that the library still suffers from degraded performance because it winds up doing some work for each check. For example, the library has to examine a boolean value for true or false to determine if it indeed should perform a check or not. However, a good design can avoid even this tiny run-time cost. For example, consider a simple library to create and write and read arrays.

```
Array (*CreateArray)(const char *name, int ndims, const int *dims);
int (*WriteArray)(Array *a);
Array (*ReadArray)(const char *name);

Array createArray(const char *name, int ndims, const int *dims)
{     /* pre-check work */
      _createArray(name, ndims, dims);
      /* post work */ }

writeArray(Array *a)
{     /* pre work */
      _writeArray(fa);
```

```
      /* post work */ }

Array readArray(const char *name)
{     /* pre work */
      _readArray(name);
      /* post work */ }

Array_Init(bool useChecks)
{
   if (useChecks) {
      CreateArray = createArray;
      WriteArray = writeArray;
      ReadArray = readArray;
   } else {
      CreateArray = _createArray;
      WriteArray = _writeArray;
      ReadArray = _readArray;
   }
}
```

Each function in the API is implemented in two pieces. An *outer* piece that does some pre- and post-anomaly detection and an *inner* piece that does the real work but no anomaly detection. By designing the public API to be a set of programmable function pointers, which are set up in the call to initialize the library, `Array_Init()`, even the run-time cost of deciding whether to perform any check or not can be eliminated.

By designing a smart library to support run-time switching of anomaly detection, whenever a client does encounter an un-caught anomaly, the user can re-run the client with the checks turned on to help narrow in on the cause. In an approach where anomaly checks are compiled out, the client needs to be re-linked with a version of the library in which the checks were *not* compiled out. This takes time. In addition, it can cause relocation of code and can wind up hiding the anomaly.

Practice 21: Accept run-time switches for anomaly detection directly by the library

How can the end-user of an application control the behavior of one of its internal libraries at run-time? If the only path into the library to turn anomaly detection on or off is through function calls into the library, then the client must be designed to take responsibility for accepting user input at run-time and calling the appropriate functions in the library. If the developer of the client fails to design it in this way, then there is no hope for the user to control the behavior of the library later on at run-time. Consequently, the library must be able to accept input directly from the user regardless of how the client is ultimately designed.

A smart library is designed to accept input directly from the user for its run-time control parameters. A library can accept user input via command-line arguments, environment variables or some kind of a preferences or configuration file. For example, MPI is designed to accept arguments from the command-line. The call to `MPI_Init()`

takes `argc` and `argv` as its arguments. Nonetheless, there is no way to enforce that the caller pass the actual `argc/argv` of the command-line to `main()`. As another example, the SAF library uses an environment variable, `SAF_ASSERT_DISABLE` to control disablement of assertions. Our experience has been that environment variables work best.

Practice 22: Make Anomaly detection selectively controllable

The application programming interface of a library is often divided into distinct sections for different categories of operations. For example, MPI is broadly divided into point-to-point and collective communication routines. A user might not want to have to take the performance hit for anomaly detection in the collective routines if s/he is in the midst of debugging a problem that is confined to her client's use of the point-to-point routines. For this reason, the user needs to ability to selectively control how and where the library performs anomaly detection.

A smart library is designed to permit selective control of anomaly detection. There are a variety of ways to classify anomaly checks in a library to facilitate selective control. As the previous paragraph suggests, one is to classify according to distinct sections of the API. Another is to classify according to the amount of work a check takes.

For example, in SAF, anomaly checks are classified as being either pre-, post- or invariant-condition checks (Practice 23) as well as being either high, medium, low or no cost. Environment variables, such as `SAF_PRECOND_DISABLE`, then control the cost threshold for a given class of checks. For example, setting `SAF_PRECOND_DISABLE=medium` in the environment of a SAF client has the effect of turning off all pre-condition checks with medium or higher cost. A user can then turn on or off costs above or below a certain cost threshold permitting the user to trade-off performance and robustness of clients that use the SAF library.

Practice 23: Handle Design-by-Contract like all other anomaly detection

It is worth mentioning the Design by Contract programming paradigm and culling it out separately here to emphasize an important point about Practice 20. That is that all anomaly checks, even those having to do with the pre-, post- and invariant conditions of a Design by Contract (DbC) programming paradigm should be run-time switchable. In a production configuration (Practice 19), they should be off by default but nevertheless available to be switched on if the need arises. In a debug configuration, they should be on by default but it should also be possible to switch them off.

Although leaving pre-, post- and invariant condition checks compiled into a production library flies in the face of DbC theory, our experience has been that just because a client meets the conditions of a contract in one execution doesn't necessarily mean it will do so in the next. The data that flows between client and library, the contract that is, may be comprised of end-user input or data read from a file on disk which can surely vary from run to run. Or, it may be the result of a sequence of interactions between client and library which also may vary from run to run. Next, scientific computing

applications are complex enough that it is rarely possible to exercise all possible code paths in testing. It is impractical to confirm that over all possible code paths, the client meets the conditions of a contract with a library it uses. Finally, as new versions of a library are released, it may include new contractual conditions that were either unnecessary or overlooked in previous releases. Thus, the contract the client must meet can change.

As a result, in a smart library even the pre-, post- and invariant conditions of a DbC programming paradigm are run-time switchable. In fact, they are treated uniformly with all other anomaly detection checks the library does. In a smart library, there is nothing really special about checks associated with a DbC programming paradigm.

Practice 24: Never call `abort()`, `assert()`...(Well, almost never, see Practice 25)

When a library aborts due to some kind of anomaly, it is saying there is no hope for execution to proceed normally beyond the point where the anomaly is detected. Nonetheless, it is dictatorially making this decision on behalf of the client. Even if the anomaly turns out to be some kind of internal bug in the library, which obviously cannot be resolved in the current execution, aborting is a bad thing to do. The fact is, a library developer cannot possibly know the fault-tolerant context in which his/her library is being used. The client may indeed be able to recover from the situation even if the library cannot.

For example, consider a scientific computing client that uses multiple solver libraries. One, say `libsolvB.a`, is relatively new and untested while another, say `libsolvA.a`, has been used without error by the client for years. The client may be designed in such a way that if it encounters problems using `libsolvB.a`, it simply falls back to using `libsolvA.a`. However, if `libsolvB.a` aborts due to an anomaly it detects, the client cannot recover. Furthermore, even if there was no `libsolvA.a` to fall back on, there can be other actions a client might like to take in response to an anomaly in `libsolvB.a`. For example, after the client has ground out numerical computation for hours, it may wish save its internal state (make a check point) before terminating in response to an anomaly in `libsolvB.a`. Again, a client cannot do this if a library it uses decides, for any reason, to abort.

For these reasons, a smart library contains no calls to `abort()` or `assert()` or the like.

Practice 25: Abort only to avoid parallel deadlock

For parallel libraries, it is important to recognize that there is indeed a fate worse than abort. It is deadlock. The problem with deadlock is that it is as bad as an abort in that execution ceases with no way to gracefully recover, but worse than abort in that the client doesn't know it has happened.

Smart parallel libraries are designed with this in mind and include checks to detect situations that can or will lead to deadlock and then abort before it occurs. For example,

the call to open a file in HDF5, `H5Fopen()`, is collective across all processors and requires all processors to pass the same name for the file. If they don't, deadlock can occur, maybe within in the execution of `H5Fopen()`or maybe in some later API call involving the file. Of course, to detect this condition, HDF5 would need to engage in communication to confirm all processors agree on the file name before proceeding to open the file and abort if it is detected that they all do not agree on the file name. Since would be costly to do communication to check for this condition all the time, it should be possible to turn it off (Practice 20).

Practice 26: For parallel libraries, check processor agreement in collective calls

In a parallel library, certain API calls may be *collective*. This means that if any processor makes the call, all processors must make the call. Typically, in a collective call all processors must agree on one or more of the arguments in the call. When this is not the case, the behavior can be undefined and can often lead to deadlock. The problem is that the point at which deadlock occurs can be later enough in the execution that finding the actual cause can be very difficult.

A smart library provides checks in collective calls to confirm that indeed all processors do agree on the arguments they are supposed to agree on. Again, although these checks involve additional communication, since they can be turned on or off for any given run, they do not always have to effect performance.

In cases where the data structure that all processors must agree on is large and/or complex, it is easiest to communicate and compare check sums on the data instead of the actual data.

Practice 27: Test the library's ability to catch the anomalies it claims to detect

False positive and false negative anomaly indicators are painfully misleading to the user. A false positive permits execution to continue giving the false impression that all is well. Later on in execution when things really unravel, the last thing the user will think to consider as a root cause is a condition the library claims to catch but failed to report. Likewise, a false negative can cause the user to debug a nonexistent problem. So, its important for anomaly detection code to be correct.

A smart library includes tests aimed at driving the library into anomalous behavior and confirming the library indeed catches it.

Practice 28: Pass all anomaly response through a common handling function

In a smart library, all anomalies are handled by a common handling function. By passing all response to anomalous behavior through this function, a smart library can offer a great deal of flexibility in deciding just what its response needs to be. Will it simply print a warning message and continue? Will it print the function call stack? Will it return control to the client with an error code? Or, will it throw an exception the client has claimed responsibility for catching? Will it pause execution to wait for a debugger to

attach? The point is that there are many, many useful things a library can do in response to an anomaly and it is worthwhile to provide a single point of control for all responses.

In fact, in the case where multiple libraries are involved, it is preferable that anomalous behavior in any one library be handled by the same handling function. Otherwise, the client winds up having to deal with each library's own unique way of responding to an anomaly.

Practice 29: Check for spelling errors in string-based control parameters

If command-line arguments or environment variables are used to control functionality like anomaly detection (Practice 22), API call tracing (Practice 1), and other behaviors of a library, it becomes important for the library to detect when a user might have requested a particular behavior but accidentally miss-spelled the parameter that controls it.

A smart library employs a standard naming scheme for all of its command-line arguments and environment variables (Practice 4) so that miss-spellings can be detected. For example, the SAF library watches for several environment variables that control its behavior. Each is prefixed with 'SAF_'. The SAF library reads all variables defined in the environment looking for all those that begin with 'SAF_'. For each variable that it finds, it checks to see if that variable is one that the library knows about. If not, a message warning of the possibility of a miss-spelled environment variable is displayed. Of course, this cannot catch a miss-spelling in the first 3 characters but hopefully those are easy for the user herself to spot.

Practices that Lead to Quality Documentation

In this section we describe a few other practices related to documentation.

Practice 30: Document compile-time symbols the library defines for its client's to use

A smart library documents the compile-time symbols it defines for its clients to query for the existence of certain features, version information, etc.

Practice 31: Document environment vars and command-line args the library uses

It is common to find in the documentation for an application an explanation of relevant environment variables and command-line arguments. Ordinarily, one would not think of having to do the same thing for a library. However, several of the practices described so far for smart libraries have involved the use of environment variables and command-line arguments to control the behavior of the library. It is important to document these.

A smart library documents environment variables and command-line arguments, if any, that control its behavior.

Practice 32: Document the version in which each public API symbol first appeared

When application developers are using a library and referring to its documentation, if they wish to make their application backward compatible with different versions of the

library, they need to know if certain API functions they're using are only available in certain versions and, of course, which versions.

In a smart library, API functions that were introduced after its first public release are documented with the version number they entered the API and, if applicable, the version number they were deprecated and later removed from the API.

Practice 33: Use a *single-source* auto-documentation tool such as *mkdoc*

There are a number of tools now available that enable library developers to write documentation and maintain it embedded directly in the library source code as comments. However, there is a major problem with almost all of these tools. They rely upon comments that duplicate information already available in the executable statements of the source code itself. For example, one needs to enumerate all of the arguments to a function in the function's comment-block. Over time, as code evolves, its far too easy for the comment and the executable statements of the source code to get out of sync. In short, none of the tools currently available, save one, enforce a truly *single source* for documentation.

A tool developed at Livermore Labs called *mkdoc*, for *Make Documentation*, parses the executable statements of the source code -- functions, data-types, macros, etc. -- to extract symbol names it uses in the documentation it produces. For example, in the source code below...

```
/* Chapter: Sets
   Audience: Public
   Purpose: Declare a new set                                    */
SAF_SetId saf_declare_set(
      const char *name,       /* name of set to create */
      int topo_dim,           /* topological dimension of set */
      SAF_Bool is_extendible  /* is the set extendible or not */
)
{
      int retval = -1;

      SAF_ENTER(retval, saf_declare_set);

      SAF_REQUIRE(...);       /* a pre-condition */
      SAF_REQUIRE(...);       /* another pre-condition */

      /* implementation details */

      SAF_ENSURE(...);        /* a post-condition */

      SAF_LEAVE(retval);
}
```

mkdoc will extract the function's name, `saf_declare_set`, as well as its argument's names and their types such as `const char *name` and the optional comments after the

arguments when it produces documentation. Mkdoc will also extract pre-, post- and invariant conditions encoded as the `SAF_REQUIRE/SAF_ENSURE` macros.

Often, with reasonably chosen symbol names, very little additional documentation is necessary. When it is, *mkdoc* offers a number of features for managing this documentation which make it far superior to most other auto-documentation tools. The reader is referred to (*Matzke, 2000*) for more information about *mkdoc*.

A smart library uses a truly single-source, auto-documentation tool.

Practice 34: Make even API function call instances themselves self-documenting

Often, users learn how to use a library by cutting and pasting code from other example clients that use that library. Consequently, it can be important for each API call instance to be a self-documenting as possible. Consider the following two calls to create a set object in the SAF library,

```
mySet = saf_declare_set(“mySet”, 3, false);
mySet = saf_declare_set(“mySet”, TOPOLOGICAL_DIM_3, IS_NOT_EXTENDIBLE);
```

The second version gives a lot more information about what the arguments mean. It suggests that the second parameter is the *topological dimension* of the set and the third parameter indicates whether the set is *extendible* or not. Of course, in either case without the API reference manual, the user would still have to guess what the first parameter is but most would probably guess correctly that it is the name to be given to the set.

A smart library is designed so that each API call instance can be as self-documenting as possible.

Smart Libraries In Action: Some Real-World Experiences

In this concluding section, we give a short narrative of some real-world experiences in how smart libraries have been a great help in saving time.

A recent build of Ale3d with HDF5

Recently, Rob Neely, an Ale3d developer, was building Ale3d on a new system at LLNL. Rob encountered a problem when trying to run the Ale3d he had built. Ale3d started and ran fine. However, when it began to write a restart or plot dump, it would produce an error message (Practice 7), presumably from the HDF5 library, that looked like...

```
HDF5 Header and Library versions do not agree
Headers say 1.2.0. Library says 1.4.5
```

So, clearly this was a situation in which somewhere in all the code that was linked together to produce the Ale3d executable, one piece of code was compiled with headers for version 1.2.0 of the HDF5 library but everything wound up being linked to version

1.4.5 of the HDF5 library.

We looked over all the makefiles and the actual compiler and link commands used to build Ale3d. They all had include and library specifications for HDF5 of this form.

```
-I/usr/gapps/ale3d/packages/hdf5-1.4.5/include
-L/usr/gapps/ale3d/packages/hdf5-1.4.5/lib -lhdf5
```

Next, we looked at the actual header, `hdf5.h` and library files, `libhdf5.a`, in these directories to confirm that both were indeed version 1.4.5 of the HDF5 library. We visually inspected `hdf5.h` and we used the command (Practice 11)

```
% strings libhdf5.a | grep HDF5 library version
```

all of which confirmed that the header and library files were indeed for version 1.4.5 of the library.

At this point, I suggested that Rob try bypassing the HDF5 version check by setting the environment variable, `HDF5_DISABLE_VERSION_CHECK` when running Ale3d. I explained that if the version check error was a false alarm, this would permit execution to proceed (Practice 8) past the check but that if the version check error was real he would probably encounter all sorts of odd issues when writing a restart file. For this reason, Rob was reluctant to try this. Instead, he wanted to focus on finding out why HDF5 was reporting a header and library version mismatch.

After further thought, we discovered that there was an old version of HDF5, version 1.2.0, installed in `/usr/local`. It was this version's header files that were getting picked up when Ale3d was being compiled. Nonetheless, Ale3d was still being linked to the version of HDF5 in `/usr/gapps/ale3d/packages/hdf5-1.4.5/lib`. Once this was discovered, the problem was resolved.

Alpha version of SIERRA's plot dump to SAF

When SIERRA was first starting to use the Sets and Fields (SAF) scientific database library for its plot dump, occasionally SIERRA would go into deadlock in parallel during one of the calls into the SAF library.

We suspected that the problem was that some processors were not agreeing on collective arguments to collective calls as they should. We wanted to re-run the test problems that hung with some additional debugging features in the library enabled. Fortunately, because of the way SAF is designed, this simply meant setting some environment variables and re-running SIERRA (Practice 20).

We re-ran SIERRA with all of SAF's pre-condition checks turned on by setting the environment variable, `SAF_PRECOND_DISABLE` to `none`. During the execution, SAF reported that the argument passed by two different processors in a collective call to declare a field in SAF were passing different values for the field's name and then aborted (Practice 25). SAF aborted execution when it detected this condition because this sort of condition is exactly the kind that can lead to deadlock.

Summary

As scientific computing applications grow in complexity, more and more functionality is packaged in independently developed libraries. Worse, as the computing environments in which these applications run grow in complexity, it gets easier to make mistakes in building, installing and using libraries as well as the applications that depend on them.

We have shown that there are key differences in software quality assurance standards for applications and libraries. We have introduced a variety of software quality engineering practices aimed at minimizing the likelihood of making mistakes in using libraries and at maximizing users' ability to diagnose and correct them when they occur. We have introduced the term *Smart Library* to refer to a library that is developed with these basic principles in mind.

We then described a number of practices in four broad categories; practices upon which all others depend, practices that enable users to deal with change, practices that enable users to detect and diagnose anomalies and practices that lead to quality documentation.

Finally, we have described some real-world experiences in which smart libraries have demonstrated key benefits.

Acknowledgments

The authors have used information gained from HDF5, MPI and SAF software development projects. In particular, we would like to thank Quincey Koziol and Albert Cheng of the HDF5 team for sharing their experiences and wisdom in developing various versions of HDF I/O software, the entire MPI Forum for its excellent work in designing and documenting MPI, Version 1.1 from which we have also learned much. This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

References

Brand, F., "HDF: the Hierarchical Data Format. " (Technology Tutorial) Dr. Dobb's Journal V23, N5, pp. 42-48 (1998).

Matzke, R.P., "The *Mkdoc* Single-Source Auto-Documentation Tool", (2001) matzke@llnl.gov

Miller, M.C., et al. "Enabling Interoperation of High Performance, Scientific Computing Applications: Modeling Scientific Data With The Sets & Fields (SAF) Modeling System", ICCS-2001 v2. (2001)

The MPI Forum, "The MPI Standard Version 1.1", http://www-unix.mcs.anl.gov/mpi (1995)

Plessel, T., "Design by Contract: A Missing Link in the Quest for Quality Software",

http://www.elj.com/eiffel/dbc (1998).