Smart Libraries:

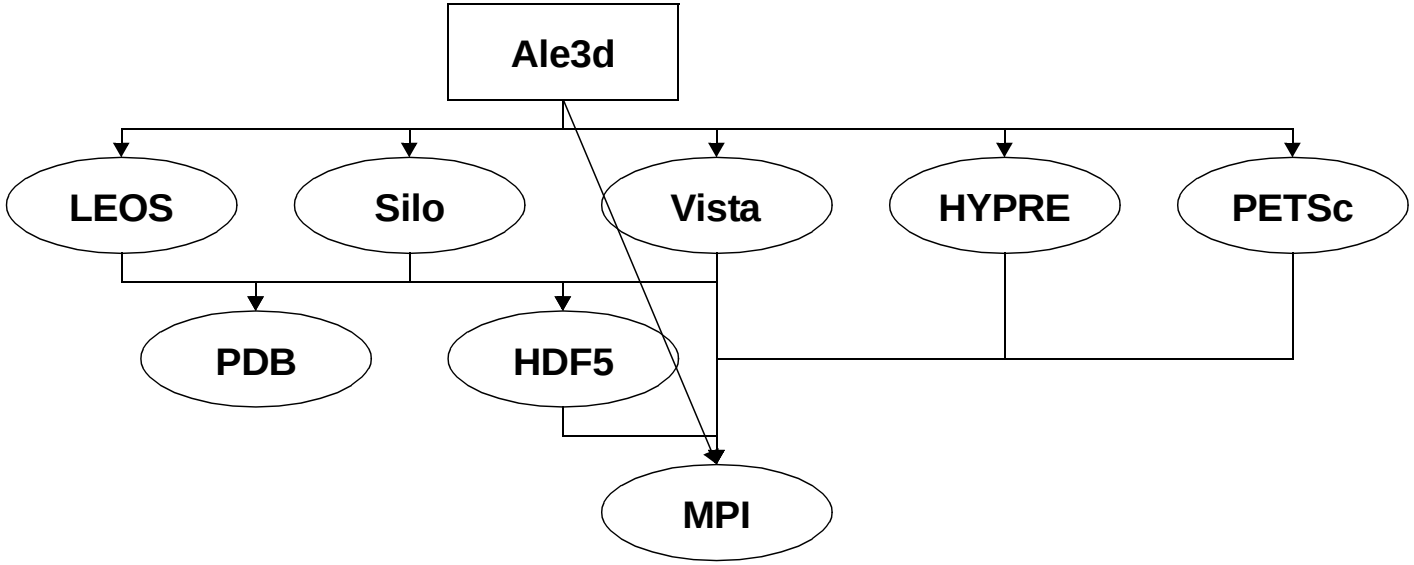
Best SQE Practices for Libraries with an Emphasis on Scientific Computing

Mark C. Miller, James F. Reus, Robb P. Matzke, Lawrence Livermore National Labs
Quincey A. Koziol, Albert K. Cheng, National Center for Supercomputer Applications

Presented at the Nuclear Explosives Code Development Conference
October 4-8, 2004, Livermore, CA.
UCRL# UCRL-PRES-206622 - 17Sep04

Why Do We Need Smarter Libraries?

Problem: Installing/Debugging/Using scientific software in the presence of complex dependencies on third-party libraries



- *Often times, things won't even build*
- *They might build ok, but won't run - can wind up debugging phantom bugs*
- *They might run ok, but give wrong answers*
- *They might run ok, give correct answers and just run 2-3x more slowly than necessary*

Solution: Design libraries to keep users from making mistakes and to diagnose and correct them when they occur

Outline

Introduce Some Terminology

Outline Differences in SQA Standards for Libs and Apps

Four Broad Categories

- *Progenitor Practices: key practices upon which other practices depend*
- *Practices that enable users to deal with change*
- *Practices that enable users to detect and diagnose anomalies*
- *Documentation for Libraries*

Some Real-World Experiences with Ale3d, VisIt and SAF

Some Terminology

Client: some software that makes use of a library

- *It may be an application or it may be another library.*
- *Ex: Ale3d makes use of the Silo library. Ale3d is a client of Silo.*

User: A person who develops client software

- ***The term "user" means "user of a library", not an application.***
- *Ex: Rob Neely uses Silo calls in Ale3d. Rob is a Silo user.*

End-User: A person who is using an app. that uses a library

- *Usually a designer.*
- *Ex: Barbara Kornblum uses Ale3d. Barbara is an end-user of Silo.*

API: Application Programming Interface

- *All the (public) symbols in a library that are available to a client to use*

Differences in SQA Standards for Libraries and Applications

Libraries must be installed and supported in many configurations

Why? Users care about how a library is compiled and installed

- *On the contrary, end-users don't care how an app. is compiled as long as it runs & works*

When compiling a library, user's care about many issues...

- *32 or 64 bit*
- *debug or optimized*
- *parallel or serial*
- *static or dynamic*
- */opt/SUNWspro/bin/cc, gcc-3.2, or gcc-2.95*
- *exception-handling or long-jumping*
- *exotic code-generation options (i.e. -qalign=pack)*
- *MPICH-1.2.4, MPICH-1.2.5, ChaMPIon, LAM...?*
- *HDF5-1.4 or HDF5-1.6*

Different choices lead to different "*configurations*"

Differences in SQA Standards for Libraries and Applications (cont'd)

Libraries require rigid
Application Programming Interfaces (API)

Why? When the API is changed, clients can cease to function!

- *On the contrary, when the interface of an app. is changed (i.e. the GUI or CLUI) the app. still works.*

Differences in SQA Standards for Libraries and Applications (cont'd)

Libraries require multiple, older versions to be maintained and supported

Why? Unyielding stability requirements of apps. that use libs.

- *Apps. cannot be required to upgrade to the newest version of a lib to get a bug fix.*
- *An app. that started using HDF5-1.4.2 in 2002 may need to continue using it in 2010*

Differences in SQA Standards for Libraries and Applications (cont'd)

Library documentation is computer-science-rich and is essential for proper use

Why? Libraries don't come with GUIs

- *On the contrary, the whole purpose of good GUI design is to obviate the need for a lot of documentation*

Progenitor Practices

Require clients to call functions to begin and end interaction with a library

Example: `MPI_Init()` and `MPI_Finalize()`

- *All other functions in MPI check to see if MPI has been initialized before proceeding*

Why is this useful?

- *Provides guaranteed first call from client into library*
- *Key to controlling global state and behavior of library*

Some functions in API may need to be called outside an enclosing `Init()/Finalize()`

- *Example: MPI_Finalized()*

Progenitor Practices

Design all public API functions to invoke common entrance and exit procedures

Example:

```
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);
}
```

Why? There are many, many possibilities that derive from this

- *API call tracing and filtering*
- *Robust error messaging*
- *Integration with performance analysis tools like Vampir or Pablo*
- *Convenient MPI message tags*
- *Detection of conditions leading to deadlock (i.e. early return from collective code blocks)*
- *Others...*

Progenitor Practices

Put all public symbols (e.g. the API) in their own "namespace"

- *No, we don't mean use the C++ namespace keyword*

How? Prepend <u>every</u> public symbol w/ 2,3 or 4 letter acronym

- *functions*
- *global variables and constants*
- *type definitions*
- *pre-processor symbols/macros*
- *enums*
- *include filename(s)*
- *environment variables that library interprets*
- *command-line arguments that library interprets*
- ***Make public only what needs to be public (e.g. NOT contents of config.h)***

Example: HDF5 prepends 'H5' to everything

- `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*
- `HDF5_DEBUG` - *environment variable controlling debugging/tracing*

Enabling Users To Deal With Change

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

Typical 3 digit scheme, A.B.C

- *A is major digit*
- *B is minor digit*
- *C is patch digit*

	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 addition (maybe) Performance improvements[d]
impact on application when digit changes	re-type <= impact <= re-think	rebuild <= impact <= re-type	none <= impact <= rebuild
Events that trigger incriment	cost/impact on users	planned devel. activities	need to role out bug fixes
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 patch 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.

Enabling Users To Deal With Change

Provide a version number consistency check between client and library

Example of link-time approach

- *In* `mpi.h`*...*

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

- *where the symbol* `MPI_Version_1_2_4` *is defined in* `libmpi.a` *for MPI-1.2.4*

Example of run-time approach

- *In* `mpi.h`*...*

```
#define MPI_Init(Argc,Argv)       mpi_init(Argc, Argv, 1, 2, 4)
```

- *where* `mpi_init()` *will check major, minor and patch digits when it is called*

Note: differences in version number are a necessary but not always sufficient indicator of version incompatibilities

- *The run-time approach is more flexible as it can be designed to be overridden*

Enabling Users To Deal With Change

Build Version Compatiblity History Into the Library

Example:

- *put similar code such as this in both header (.h) and object (.o) files*

```
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
   {1, 3,  1,  4,  1}};// 1.2 & 1.3 OK
```

- *In the call that initializes the library, (e.g. SAF_Init()), can check version client was compiled with (from the .h file) against version client is linking to (the .o file)*

Note: With a test suite of sufficient coverage, can automate construction of this history.

Enabling Users To Deal With Change

Provide symbols client can query at compile time for version information.

Example:

- *HDF5 defines these symbols in the* `hdf5.h`

```
#define H5_V_MAJOR   1
#define H5_V_MINOR   2
#define H5_V_PATCH   3
#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))
```

- *If* `H5Pfoo_bar()` *was added to the API in version A.B.C, a client can do the following*

```
#if H5_V_GE(A,B,C)
    H5Pfoo_bar(); /* do it the new way */
#else
    H5Pgorfo();   /* do it the old way */
#endif
```

- *Implicit in 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.*

Enabling Users To Deal With Change

Embed String Representation for Version Information in the Library and Any Client.

Example:

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

Advantage: Can now use ‘strings’ command to query version information from any file (.o, .a, executable, etc.)

```
% strings globe | grep ’HDF5 library version:’
HDF5 library version: 1.6.0
```

Enabling Users To Deal With Change

In languages that do not support polymorphism, fake it

Do this by including a catch-all argument in functions that are likely to change

Example: HDF5's interface to create a dataset

-

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

- `plist` *represents a list of additional arguments.*
- *As dataset creation evolves, new functions are added to manipulate contents of* `plist`
- *For example, after HDF5 was designed, the concept of a fill value was added.*
- *A new function was added like so...*
-

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

Enabling Users To Deal With Change

Make work-arounds[1] conditionally compiled and off by default

Example: `MPI_Type_struct()` on Sun Solaris was buggy

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

#ifdef __sun__ && 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);
```

- *Work-around gets compiled only when on a sun &* `-DSAF_WA_mpi_type_struct`
- *This means you see it in the build-log every time you build it. Its in your face!*

1. By definition, a work-around is an alternative AND inferior approach

Enabling Users To Deal With Change

If API Changes Become Necessary, Design them to Break The Client's Compile

Simplest Approach: Change the Function/Method Name

- *old function's name*

```
set_name(const char *name, const char *title);
```

- *desired new function...*

```
set_name(const char *name, const char *department);
```

- *actual new function's name*

```
set_namedep(const char *name, const char *department);
```

Depricate the Old Function...

```
void set_name(const char*name, const char *title)
{
#warning‘set_name’ is deprecated, use ‘set_depname’
    set_depname(name, “unknown”);
}
```


Enabling Users To Deal With Change

Provide means for users to identify installed configurations

Example of using the pathname

```
/usr/gapps/saf/1.6.0/Linux/mpich-1.2.5/64
```

Example of using a `.settings` file

```
% ls /usr/gapps/saf/1.6.0/Linux/mpich-1.2.5/64
  libsafapi.a   libsafapi.settings
% cat libsafapi.settings
SAF Version:            1.6.0
Configured on:          Tue Jul 15 15:54:47 PDT 2003
Configured by:          matzke@gentoo.llnl.gov
Configure mode:         development
Host system:            i686-pc-linux-gnu
Byte sex:               little-endian
Libraries:              static
Parallel support:       mpicc
Installation point:     /usr/gapps/saf/1.6.0/Linux/mpich-1.2.5/64/debug
Compiler:               /usr/gapps/mpich/1.2.5/Linux/64/debug/bin/mpicc
Compiler switches:      -D_FILE_OFFSET_BITS=64
Extra libraries:        -lz -lhdf5 -lxml -lxml2 -lm
```

Enabling Users (and End-Users) To Detect and Diagnose Anomalies

Anomaly: Any condition that prevents proper execution

- *lack of resources (i.e. memory or disk)*
- *improper calling usage on the part of the client*
- *an outright bug in the library*
- *a failure in some other software component on which the library depends*
- *even the pre-, post- and invariant-condition checks of a DBC programming paradigm*
- *anything else*
- *"Any" really does mean any anomaly*

Many libraries include code to check for anomalies

- *We call this code anomaly detection*

Any anomaly not caught by the library will most-likely manifest itself with catastrophic consequences later in execution.

- *We call these UNcaught anomalies*

Enabling Users (and End-Users) To Detect and Diagnose Anomalies

Provide at least two quality of service configurations; debug and production

A debug configuration...

- *is compiled with anomaly detection included **and it is ON by default***
- *is compiled with symbolic debugging information (e.g. -g flag)*
- *provides maximum development support while user is developing a client.*

A production configuration...

- *is compiled with anomaly detection included **and it is OFF by default***
- *is compiled with optimization (i.e. -O3 flag) and without debugging information*
- *provides production-level performance once development is complete.*

A Third Option: An “Optimal” configuration...

- *no debugging (e.g -g flag not present)*
- *maximal optimization*
- *anomaly detection compiled out*

Enabling Users (and End-Users) To Detect and Diagnose Anomalies

Anomaly detection should be run-time tunable via environment variables[1]

Anomaly detection can degrade performance

- *Users need to be able to easily throttle how much anomaly detection a library performs.*

Users need a way to turn it off to maximize performance

- *In a debug configuration, anomaly detection is ON by default.*
- *So, users need an easy way to turn it off*

Users need a way to turn it on to zero in on an UNcaught anomaly they encounter

- *In a production configuration, anomaly detection is OFF by default.*
- *So, users need a way to turn it on.*

Example: HDF5 API has categories (H5F, H5D, H5P...)

```
% setenv H5_DEBUG="all -dp"
% setenv HDF5_DEBUG="-all sfp"
```

1.If the only path into a library to control anomaly detection is via API calls, every client has to take responsibility for accepting user input and making the appropriate API calls.

Enabling Users (and End-Users) To Detect and Diagnose Anomalies

A library should NEVER call abort (or exit or assert, etc.) (well, almost never)

Library developers cannot possible know the fault-tolerant context in which their library is being used.

Example: What if Ale3d encounters an anomaly in a solver lib.

- *It could fall back to using another solver lib. (Ale3d uses multiple solver libraries).*
- *It could decide to write a restart file and terminate.*
- *If the library terminates execution via abort, assert or exit, Ale3d has no re-course.*

This practice is appropriate even if the anomaly that is encountered represents an internal bug in the library which cannot possibly be resolved in the current execution.

Enabling Users (and End-Users) To Detect and Diagnose Anomalies

Abort only to avoid parallel deadlock

For parallel codes, there is a fate worse than abort; deadlock

- *Its as bad as abort in that execution ceases with no graceful way to recover*
- *Its worse than abort in that you don't know its happened.*

Often, you can detect conditions that will or are likely to lead to deadlock. For example...

- *A call to write a field in the SAF library requires all processors to pass a field handle that is in the same database.*

Documenting Libraries

Document ALL of the library's public symbols

Environment variables that library interprets

Command-line arguments that library interprets

Compile-time symbols client can use to query info about lib.

For any public API symbols
that do not exist in all version of the library
document the version the symbol entered/exited the API

Documenting Libraries

Make API call instances self-documenting

Example: which form is easier to determine what the args are

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

- *An educated guess suggests first argument is the name of the set.*
- *However, it is hard to say what* 3 *and* `false` *represent*

```
mySet = saf_declare_set(“mySet”, SAF_TOPO_DIM_3, SAF_EXTENDIBLE_FALSE);
```

- *In this form, it is easier to guess that the second argument is the topological dimension and the third argument indicates if the set is extendible.*

Often, users learn to use a library by reading example code and cutting and pasting other people’s code.

Documenting Libraries

Use Mkdoc
(its the only auto-doc tool that uses source code directly)

```
/*-------------------------------------------------------------------
 * Audience:    Public
 * Chapter:     Files
 * Purpose:     Open an exiting CBM file
 *-------------------------------------------------------------------
 */
hid_t CBM_Open(
   const char *name,      /* [IN] path name of file to open */
   int flags,             /* [IN] File access flags.*/
   hid_t access_id        /* [IN] file access property list.*/
)
{
    /* implementation of CBM_Open() */
}
```

Mkdoc features

- *Reads function names/arguments directly from source code*
- *Works with C/C++ code*
- *Generates html, tex, text, Maker Interchange Format*
- *Knows when its documenting a lib versus examples in use of a lib and includes links back to the lib from examples*
- *Filters documentation for different audiences*

Summary

Installing/Debugging/Using software in the presence of complex dependencies on third-party libraries is challenging!

- *There are a lot of ways to get it wrong*

It would be nice if libraries kept us from getting into trouble

We've presented some practices that have helped

Some Real-World Experiences

Recent build of Ale3d with HDF5

- *HDF5, version 1.2.0 was installed in /usr/local*
- *Ale3d was getting includes from /usr/local and lib from -L/usr/gapps/hdf5/1.4.2/*
- *At run-time, HDF5 reported inconsistent version*

Alpha version of Sierra's plot dump to SAF

- *Sierra was hanging during I/O of plot-dump to SAF database*
- *re-ran problem with* `SAF_PRECOND_DISABLE=none` *(all pre-condition checks enabled)*
- *SAF reported inconsisted field name for a field that was shared amoung processors*

Some 2002 Benchmarking Tests of Ale3d/SAF in IBM's GPFS

- *numerous env. variables and command-line options controlled various test parameters*
- *enviornment variable* `SAF_PRECOND_DISABLE` *has profound impact on performance*
- *If miss-spelled, you'd get skewed results and not know it.*
- *Because SAF checked spelling of all* `SAF_`*... environment variables, it reported errors*