EOL Software Development Guidelines

Gary Granger, John Allison, Linda Cully

NCAR/EOL

Table of Contents

1. EOL Software Development Guidelines

This living document is an evolving guide to software development practices in the NCAR Earth Observing Laboratory.

This document is also available as a PDF.

Math test:

$$\ker f = \{g \in G : f(g) = e_H\}.$$

$$e_p = \frac{5417.98 e_s}{t_n^2} t_n = t_n + \frac{e_o - e_s}{e_p} t_s = |t_m - t_n| t_m = t_n e_s = 10^{9.4051 - \frac{2353}{t_n}}$$

2. Introduction

2.1 Purpose

These guidelines describe best practices for software engineering in EOL. The purpose of these guidelines is to improve the development experience for users, yield better software for EOL, encourage a common set of good software engineering practices within EOL, and nurture software development skills which will best serve EOL and its scientific community. As we try to improve all of our software developments, these guidelines can be a resource to improve development in similar ways towards similar goals.

These guidelines are not intended to be mandates. The guidelines encourage some practices more strongly than others, but none are absolute requirements. Most of the practices can be adopted individually and incrementally, for any scale of software development, and by anyone developing software, not just software engineers.

In this document, key practices will be emphasized in bold, while further details and discussion may be referenced on a separate page. The bottom of this page provides quick references with links to specific sections and details.

Many people have contributed to this document, in particular Gary Granger, John Allison, Linda Cully, and Sandra Thurn, with helpful input from many others in EOL. It is meant to be an evolving and organic document that all in EOL are welcome to edit. If you have questions about this document, email Gary.

2.2 Principles

The EOL software engineering first principles are the motivation for improving our software development process and the measure of the effectiveness of our software engineering practices. From these principles follow the best practices we want to follow.

EOL software engineering practices emphasize and facilitate utility, efficiency, flexibility, reliability, accountability, and cooperation.

Software development in EOL must be flexible to adapt to evolving research, instruments, deployments, and data. Like the research instruments and research data themselves, software will often be prototypes or cutting edge, but quality and maintainability need not suffer. Development must adapt to advances in software engineering tools and techniques.

Software developments must be efficient to make the most of limited software engineering resources to support EOL's mission. They must accommodate timelines dictated by available instrument hardware and field project deadlines.

The goal of any EOL development is to provide a useful tool or product to the EOL user community. EOL software engineers must be accountable and available to that community. We must work closely with the community to understand needs, solve problems, and fulfill requirements.

The culture among EOL software engineers is based on supporting science and solving technical problems in a spirit of cooperation across all disciplines.

3. Project Management

Some aspects of software development naturally relate to project management, especially when the software is only one part of a larger project.

3.1 Consult on Project Management

Software projects can be completed in an efficient and organized manner by using a project management framework. Project management professionals can provide guidance on the right process level for your project type and size. They are trained to help you select the methods most likely to benefit your project. Whenever it is important to reach a goal quickly, predictably, and with high quality results, project management tools and processes can be used to your advantage.

Two organizations specializing in project management methods and certifications are the Project Management Institute (PMI) and the Scrum Alliance.

3.2 Use Agile Development Practices

Agile software development processes facilitate a cooperative and interactive software development.

- Agile Software Development

3.3 Hold project reviews

For some projects, it may make sense to expand the design review concept into a more general project review. The purpose would be to **review and collaborate on all aspects of the project** , such as the requirements, design, implementation, milestones, priorities, risks, technology selections, and other initial decisions. This would be the opportunity for other experts (like project management experts) and other developers to learn about and help guide the project, so the project benefits from the combined development experience in EOL.

Even without an actual project review meeting, **developers should actively seek input from other developers and other experts**. This seems to be accepted as a good practice in EOL, and it is consistent with the principle of cooperation, but the practice must actively be encouraged. It would be a natural extension to the project checklist (see the Software Project Template) to include which developers have consulted on the design and implementation of the software. Project reviews also give less experienced software engineers a chance to question and learn about good development practices.

3.4 Track issues

At present, issue tracking is not something that is used widely or regularly. Users would not be expected to use it, but certainly it should be kept open and available to everyone. An issue tracking tool also could be an important resource for tracking progress, but only if developers use it and keep it current.

3.5 Track progress

Some of the process guidelines relate to tracking the progress of a project, such as the project overview page and issue tracking. The questions are whether more tracking is needed and what are the best practices for tracking and reporting progress. Project tracking would benefit users, developers, and managers. Here are some notable pieces to the progress tracking picture:

- Participation in regular project meetings
- Milestones on a web or wiki page, with the current status of each milestone
- Issue tracking, possibly with time tracking also
- Emails on commits
- Project overview page, with a field for current status and latest progress
- Blogs, by developers and by project leads

All of these have been used in some form or other in EOL. As a minimum, every software development should have a web site which hosts the project checklist, ideally with something like a blog for status updates. The web site could be a wiki page, an EOL plone page, or a project wiki on the UCAR wiki. A directory of all of these web sites would be a comprehensive list of all software developments in EOL.

Even if a long range schedule and formal milestones are not possible, a software development can still report on the current efforts and impending tasks using the same mechanisms.

Next page: Development Process

4. Agile Software Development

Agile software development is a family of development processes based on a set of principles which align well with existing practice and culture in EOL.

Our software development projects, particularly those that emphasize iteration and evolving scientific requirements and instrumentation, may benefit from adopting aspects of the agile software development methods. As examples, releasing working software early and throughout the development process allows users and domain experts to give continuous, valuable, and relevant feedback. This encourages the exchange of new ideas, a cooperative and interactive software development environment, and should result in the creation of valuable and useful software. Test-driven development emphasizes writing tests first, which helps to clarify purpose and verify that software works as expected.

Agile software development is a people-oriented, collaborative development approach with the flexibility to respond quickly to changes. The Agile methodology family includes techniques such as Extreme Programming (XP), SCRUM (project management framework), LEAN, Feature-Driven development (FDD), Test- Driven development (TDD), and Dynamic Systems Development Method (DSDM). There are many options and no specific engineering practices are prescribed.

Agile Software Development process ideas are based on the Agile Manifesto and its associated basic principles (www.agilemanifesto.org).

Next page: Development Process

5. Development Process

This section outlines phases and infrastructure related to the software development lifecycle, but not a sequential timeline.

5.1 Requirements

Rather than suggest a specific requirements discovery process, here are some issues that should always be considered when identifying the requirements for an EOL software project. These are not necessarily important for all EOL software, but they should not be overlooked. For example, for data-critical applications, the requirements must address data integrity and redundancy. High-bandwidth applications must identify the total throughput needed, so the throughput can be tested and verified.

As part of the requirements process, it is especially important to identify the users, since users must be involved in defining the requirements. In other words, a requirement can be posed as a question, and the users and domain experts should answer it, not the developer. (Unless the developer is a user also, of course.) The intended users also impact the target platforms, implementation language, and dependencies.

As a practical implication, users should be invited to any meeting where requirements will be discussed. [HFSD] and [FOWLER] have straightforward explanations of the necessity and value of close user involvement for iterative requirements analysis and development.

Some statement of requirements is necessary for any size of development, large or small. Larger developments will have more requirements, but they do not need to be specified all at once. Instead they may be discovered as part of an iterative process. The important thing is that they be defined and documented along the way, so the software can be developed with the correct requirements in mind, ideally to the point of developing tests to verify the requirements.

Writing use cases and user stories is good practice for elucidating requirements. They can be written from the user's point of view, and they only present the behavior expected by users without being biased by design or implementation. It is natural to describe software by telling a story about what it does and how the functionality will be used. Stories also force consideration of error handling from the user's point of view: how should the application and user interact when an error case occurs?

Here is a list of requirements to consider, in no particular order:

- **software lifetime**: How long will the software be needed? This requirement distinguishes the one-off fixes from longer term software infrastructure.
- **external support**: Does the software need to be deployed for users outside of EOL?
- **target platforms**: Does the software need to run on multiple platforms, embedded or not, particular operating systems, and so on? Is there a requirement for data portability between architectures?
- **data security**: How robust and redundant does the data manipulation and storage need to be? Raw data recording obviously needs stricter data security requirements than reprocessing software, but someone still needs to decide how many backups are enough.
- **access security**: Do user authentication and authorization need to be built into the software?
- **monitoring**: How will users and operators monitor the status and progress of the software?
- **auditing**: When processing data, what metadata will be preserved and augmented and tracked?
- **data formats**: What data file formats must be supported for input and output?
- **error recovery**: Does the software need to be able to recover quickly from problems, such as during field operations? Perhaps it needs to save checkpoints and be able to resume from the last checkpoint. What errors need to be detected automatically and how will the operator be alerted?
- **data throughput**: What is the minimum data throughput requirement? Even post-processing software may have a throughput requirement, if the users want to process field project data in hours instead of days.
- **algorithm flexibility**: Some software may need special accommodation to support alternative and evolving algorithms.
- **configuration**: How flexible does the software configuration need to be? Will it operate for different field projects, test modes, and instrument modes? Will different users need to run it with their own customizations? Where will it get configuration settings and metadata?

5.2 Design

In any software project, there are many alternative designs which all meet the requirements. The challenge for developers is finding a design which meets the requirements but also can be implemented, communicated, maintained, and elaborated as effectively as possible. The software abstraction should map well to the domain, and the language and concepts in the design should facilitate communication about the problem rather than confuse it. Thus documentation of the design is important.

One best practice seems to be a coherent and consistent partition of the problem into logical components, a basic block diagram, following as closely as possible the vocabulary of the problem domain. The partitioning then maps directly to the namespaces and symbol names used in the code. A design document or block diagram describes the components, their responsibilities, and their collaborations, and as such it serves as an overview of the design as well as a guide to the symbols in the source code.

UML diagramming tools can be helpful here, but so far the experience with them in EOL is limited. As more EOL developers become familiar with UML notation, it will be more feasible to share and discuss designs through UML diagrams.

5.3 Design reviews

Software design reviews have not really been tried in EOL, but they still seem like a good idea. As an action item from this report, we should make an effort to hold design reviews before anything is implemented, even if the review is informal.

Typically, the participants in design reviews will be other developers, as well as perhaps a domain expert to help clarify the requirements. That said, EOL software designs often must compromise between requirements. For example, there may be trade-offs between bandwidth available on the current hardware versus new software that would be needed for new hardware. Also, the priority of each requirement must be decided by the users. (For iterative developments, this relates to selecting which feature to implement in the next iteration.) For these reasons, users often need to be involved in design reviews also.

An iterative development will have multiple reviews, one for each iteration of the design.

5.4 Documentation

Documentation is part of both process and infrastructure. There are certain aspects to a software development that should be documented before coding should continue:

- Requirements
- Design
- User experts
- Guidelines checklist
- Project home page

For smaller projects (or small iterations), the requirements may be just a statement about objective, and the design documentation may not be very involved either. However, even a small project requires decisions about implementation language, data formats, GUI framework (or not), basic classes, and so on. As part of the design documentation, it is helpful to document critical design choices which were rejected rather than only the choices which were selected.

The domain experts are the stakeholders in the software and those who will know the most about whether the software is accurately modeling the application domain. Usually the domain experts are obvious, but it helps to name them explicitly in the documentation, since they are important references for the software developers. As the wisdom goes, software analysis does not happen unless a domain expert is involved. [FOWLER]

An example has been created for the checklist, linked below:

- Software Project Template

The purpose of the checklist is to communicate to others, clarify who has a role in the development and what that role is, provide a quick overview for those not directly involved in the project, and provide a single point of reference for project status and all artifacts of the project. Filling in a checklist assures that important infrastructure is planned for and taken care of ahead of time, such as revision control, wiki pages (or wherever the project checklist will reside), issue tracking, and so on.

Note that for an iterative development, where features are implemented in incremental changes to working software, all of the above documentation requirements still hold. Before a new feature can be implemented, requirements should state what the feature will do, there must be a design in mind for the feature, and user experts must have been consulted about the feature.

5.5 Code reviews

This is another idea from industry, promoted as beneficial and worth the effort, which EOL has never really tried. Emails on code commits allow for sporadic code review of a sort, and we have the Crucible code review tool available in EOL, but we're not taking full advantage of this. The report Software Practice in the Ptolemy Project is very positive about code reviews, and the experience there should be very relevant to EOL.

As a minimal guideline, since every project should be using revision control, every project should likewise be taking advantage of commit emails. Setting up emails on code changes is convenient, and it greatly boosts communication and awareness between developers as far as notifying everyone exactly what is happening in the source tree.

5.6 Releases

Software releases are very ad hoc at the moment across all the projects. We need to adopt a more consistent process for software release. To avoid confusion, the process should be the same for users both inside and outside EOL, even if some software is only available within EOL. More formal releases make for more consistent software quality and revision tracking "in the wild". It also eases the maintenance burden on users, especially "unintended side effects" when production software is upgraded without warning and at arbitrary times. A release should be tagged, tested, packaged, advertised, available, and supported. Release announcements and release notes should be in a single location on the web. Release notes should include a summary of what has changed in the release.

5.7 Acceptance or bug-fix phase

All software when first released goes through a phase where users first start using it, learning it, and discovering bugs in it. We may as well identify this phase and work to make it go smoothly. Many open-source projects (GCC, KDE, Linux Kernel) have a bug-fix-only phase, where work on new features stops in favor of fixing all of the high-priority bugs. This phase can be helped by running (and passing!) tests to verify requirements as well as regression tests to prevent bugs from recurring. Perhaps EOL developments don't warrant a formal or even an informal acceptance phase, but all projects should plan to spend time focusing exclusively on fixing bugs. [See the Joel Test, step 5.]

5.8 Maintenance

In many ways maintenance is its own step in the software development process. Porting to new environments or OS revisions likely does not require analysis or design, but it does require development time. It is difficult to predict the resources needed for maintenance, but some maintenance issues can be seen coming. For example, external dependencies may go away or change (Qt4), formats may change (netcdf 4), hardware may change (64-bit), and new technologies may need to be adopted (web, rpm).

5.9 Pair programming

Pair programming is a technique from agile software development where two programmers work simultaneously developing code, side-by-side at the console, each with a specific role. It has been used in EOL before, but otherwise there has been little experience with it. Pair programming may be a good way to spread and encourage "good coding habits", besides the potential for producing better code.

5.10 Share programming

This is an idea to increase cooperation, collaboration, and mentoring opportunities among EOL programmers by purposely sharing development projects among multiple developers. Unlike pair programming, programmers can still code separately, but they cooperate on the same project and the same source tree. For example, instead of two developers each working alone on a single project, two developers could share development on two projects.

As a general rule, programmers should not hesitate to seek discussion with other programmers to help solve a problem or share techniques. This practice can guard against the single point of failure problem. When only one programmer works on software, then support and fixes fall on that programmer, and that can be a bottleneck. If instead more than one programmer is familiar with the software, then support can be distributed. This may be an item to address explicitly in the project checklist. A project should decide ahead of time how critical cross-training and distributed support will be for the success of the software. If it is critical, then shared programming or other practices could be employed to spread out the support responsibilities.

5.11 Code sprints

Sprints have proved useful in EOL as a way to focus a group of people for a short period on solving a particular problem.

Next page: Code Sprints

6. Code Sprints

Code sprints have been used by many open-source projects and have evolved along with the agile development methods.

Here are a few noteworthy points learned from the Aeros Qt4 porting sprint:

- There is a significant effort to setting up infrastructure before the sprint, such as making sure everyone has a current, working, building source code checkout, with all of the requisite library dependencies installed.
- It takes longer than you think it will.
- There is a benefit to being removed from distractions and working collaboratively in a concentrated group.
- It may help to make more people aware of the sprint, since more people have a stake in the outcome than participate in the sprint.
- It is invaluable to have test suites in place before the sprint which already pass. The tests help verify that the changes have not broken anything that was working before the sprint. The more automated, comprehensive, and convenient the tests are, the more effective they are.

Next page: Coding Guidelines

7. Coding Guidelines

The coding guidelines relate to source code, implementation, and to the artifacts and infrastructure which should be part of development. Of course there is much overlap between process and coding.

7.1 Use automated builds.

Every project should be easy to build from checkout with an automated, batch build process. For C++ applications, SCons is recommended for its existing support of EOL tools, libraries, and common third-party components. Java projects might use ant or Eclipse. Whatever the tool, building and testing should be turnkey. [See the Joel Test, step 2.]

7.2 Use automated testing.

Use a testing framework like boost.test, cppunit, JUnit, cxxunit, or whatever, but integrate the testing into the automated build framework so it is easy to run the tests and detemine either pass or failure, without manually inspecting or comparing the output. A script which runs the program and tests for basic output is still helpful, sometimes called a *smoke test*. The more automated the testing, the more the computer can help by running the tests continuously on every change, often while further development continues simultaneously. One of the ideas behind the test-driven development process is that writing tests also helps the developer think clearly about the scope and the requirements, before writing the code.

7.3 Use continuous integration testing.

Take advantage of buildbot or other tools to run builds and tests whenever code is committed, potentially on multiple platforms, without doing it manually.

7.4 Use compilers effectively.

Most compilers can warn about questionable code constructs, such as missing return statements, unreached code, missing cases, unsafe type conversions. These warnings should always be enabled, and the code should compile cleanly without any warnings. This also works well with automated testing and continuous integration, since the compile step will report warnings whenever suspicious code has been added. When there is a warning, change the code. That way another developer later does not need to wonder whether the warning or the code is correct. GCC has the recommended *-Wall* and *-Wextra* options, but it also has the *-Weffc+* option, which warns about violations of the style rules in *Effective C++* [Meyers].

Very often code quality improves when it must be compiled on different platforms and with different compilers. On Linux, there are compilers available besides GCC, such as Clang from the LLVM project and the Intel compiler, and these may find and report different problems in the source code.

7.5 Use revision control.

There are subversion and git servers already available to use. If any of the guidelines in this document should be an absolute requirement, this is one of them. Beyond just using revision control, there are also good guidelines for commits and commit messages, such as this article On commit messages. [See the Joel Test, step 1.] Here are a couple highlights:

- Do not mix cosmetic changes with functional changes. It is hard to see from a source code diff what behavior changed if many more source lines differ just because of reformatting or reindenting or renaming.
- Commit unrelated fixes separately when feasible. That allows individual fixes to be understood separately and also backed out separately.
- Do not “break the build”. The trunk revision should always build without errors so no one has to fix compile problems just to keep working on their own changes. Commit intermediate, build-breaking changes onto a branch.
- Write a descriptive log message. On many projects the log message will be emailed automatically to other interested persons, so the log message is an easy way to send out a simple notice and explanation of a change.

7.6 Use a logging framework

For the original developer, this may not seem useful at first, but it’s value comes for other developers who later have to learn how the software works. Log messages can be valuable clues into which parts of the code are doing what when, and where a problem may be happening.

Also, when software runs remotely, perhaps even autonomously, logs are invaluable because they can be retrieved by logging into the system or by email from the field operator. A good log can give important diagnostics more completely and more accurately than can be relayed over a phone call.

See Logging Frameworks.

7.7 Use a consistent style

There are many coding styles out there, and we will never settle upon just one, but there are some good conventions to follow. The important thing is to pick a style and be consistent. See [RAL], the Google style guide, and [KDE] for other ideas about style. For the record, here is a basic list of good practices in EOL:

- Differentiate class members from local variables with a naming convention.
- Use a naming convention for class methods and functions, such as camel case (doThat) or underscores (do_that), and then use the convention consistently.
- Use descriptive names. Do not abbreviate too much or leave out arbitrary letters just to have a shorter name.
- Avoid long function definitions.
- Separate interface and implementation. In C++, the header file often can use forward declarations rather than including other header files, which simplifies dependencies and speeds compilation. When implementations are defined in source modules rather than header files, then implementations can change without forcing clients of the interface to recompile. Consider using the pimpl idiom. For languages like Python and Java which force implementation to be defined with interface, use the language to make public interface explicit. The Python convention is to use leading underscores for private methods.
- Favor spaces over tabs. Indenting by 8 spaces is excessive, 2 or 4 is adequate.
- Avoid overcrowding source code. Use spaces around operators.
- Keep source code lines within 80 columns.
- Avoid complicating the flow of an algorithm just to optimize it, unless the performance has been measured. The compiler optimizes better than programmers. Premature optimization is the root of all evil, and often it is also the root of all obfuscation.

7.8 Facilitate code reuse.

There are many existing software implementations that we can use in EOL software projects. We should take advantage of them to avoid duplicating effort. We should also write our own code to facilitate its reuse in other EOL projects. Sometimes tools can help in searching for and identifying existing code. (At one point EOL had an OpenGrok server which could be used to search almost all of the EOL software repositories for specific code symbols or arbitrary text. A capability like that would still be useful.) And of course there is no substitute for just asking around, either in person or on the software engineering mailing lists.

There are many scripts and programs which at first look like one-off tasks, such as data processing specific to a single field project. We know from experience that usually a very similar task comes along, so software gets copied and slightly modified. Code reuse implies avoiding these copies. Instead, make the scripts modular and configurable so code does not need to be copied in whole, but instead code can be shared and maintained for multiple projects and similar tasks. Consolidate similar code into functions, consolidate functions into libraries and packages, share a single code base instead of duplicating it.

This is similar to the coding maxim Don't Repeat Yourself (DRY). Avoid copy-n-paste of more than a few lines of code.

7.9 Deploy tests and logging as part of production software.

It should be possible to test software in its production environment, using the same automated tests used in the development environment. Likewise, the built-in logging capabilities should be available in production and not disabled or compiled out.

7.10 Document.

Short of mandating formal documentation requirements, it would be prudent to at least have some documentation goals. There are two types of documentation to address: programmer guides (API) and user guides. For API, at the minimum, there are almost effort-less tools now for generating API documentation from source code comments. These include doxygen, pydoc, sphinx, and javadoc. Public APIs should be commented, and they may as well be commented in a form from which online documentation can be generated. For user guides, there is no obvious answer. Some projects have used wikis effectively, especially when the wiki content can be downloaded for offline access in the field. The important point is that users should have documentation for basic operations. For operating instruments or processing data in the field, it is crucial that users have convenient and documented methods to verify that the software and instrument are operating normally. This means the software must support diagnosis (see logging) and troubleshooting, and the user guide must document the use and meaning of the diagnostics.

7.11 Do not optimize prematurely.

See all the references on the web about Knuth's quote, the 80/20 rule, and other challenges to performance metrics. Basically, there is little point to optimizing code unless it's performance will be measured accurately both before and after the optimization. Never do for the compiler what the compiler can do for you. In other words, the compiler may already be optimizing code that you think appears slow, so optimizing manually is like fighting the compiler. Don't bother unless you can verify where the compiler actually needs help.

Next page: Tools and Technologies

8. Tools and Technologies

8.1 Editing

There are several tools in use in EOL for editing source code. Besides emacs and vi, there are Integrated Development Environments (IDE) VS Code, Eclipse, and JetBrains. No particular IDE or tool should be mandated or standardized across EOL, since that runs contrary to flexibility, individuality, and the investigation of new technology. Instead, we can continue to encourage communication, share what works, and consolidate tools when prudent.

8.2 Revision Control

There are two major revision control tools in use in EOL: subversion and git. It makes sense to standardize on these two since the infrastructure is in place and there has been plenty of experience with them. Most new projects should use the NCAR organization on github. There is also a subversion server in EOL, but most software has migrated away from that to github. See the EOL wiki for more information on git, subversion, and migrating to git.

8.3 Unit Testing

Test-driven development is a valuable practice that can be used in projects of any scale. It forces developers to consider requirements and expected behaviour from the beginning, and then unit tests verify the behavior and provide some assurances that code still works after the changes. Various libraries and harnesses have been used in EOL to facilitate testing.

Java in general can use JUnit, for which Eclipse has plugins.

For python, there are pylint and unittest and py.test.

C++ projects have used CppUnit, Boost Test, CxxTest, and Google Test.

8.4 Memory Checking

A valuable tool for checking for memory errors in an application is valgrind. It is a very good practice to run compiled applications through valgrind. If the tests can be scripted and automated, then valgrind can easily check for memory errors and memory leaks each time the tests are run. The more source code exercised by the tests (code coverage), the more thorough the memory checking.

8.5 Issue Tracking

The default issue tracking tool in EOL was JIRA, and several projects still use it. However, github has its own issue tracking, and sometimes a wiki is used to track issues. Email does not make a good issue tracking tool, because the thread is spread across emails, it is difficult to catch someone up, and it is difficult to search for past similar issues. Note that field deployment issues can be tracked and not just software issues. UCAR JIRA is now used to track the tasks and problems related to software and systems deployments for various ISF and RAF platforms.

8.6 Continuous Integration

EOL migrated from Buildbot to Jenkins for continuous integration and testing. See the EOL Software Engineering Wiki for information on how to use the EOL instance.

8.7 Builds

EOL software has been known to use make, autoconf, shell scripts, SCons, Visual Studio, cmake, and qmake to build and install. There are many possibilities, but should EOL try to settle on just a few? SCons is used by several projects in EOL, and many of them share extensions to SCons called eol_scons.

8.8 Code Reviews

There seems to be two common approaches to code reviews: emails and github comments. It has been very helpful and effective to send email notifications on each code commit, especially if they contain diffs. Then other developers know what is being changed and can reply with comments about the changes. Github also provides ways to comment on pull requests and directly on code in commits, and that has proved convenient and effective also. Formal code reviews, however, have probably only rarely happened in EOL, if ever.

8.9 Packaging, Distribution, and Installation

Some projects provide targets to assemble an installation package. The package can be an RPM or a tar.gz file. Providing a standard installation package facilitates deployment to multiple field systems and to internal EOL servers. RPM's can be deployed through the EOL YUM repository.

If an application is being installed from the source tree, such as with a build target, or it is being packaged in an archive, then certain conventions should be followed as to where to install the necessary files. Linux has a standard for where files are installed on a system called the Linux Standards Base. Likewise, it is good practice to not install unstable software into production locations. On EOL servers and desktop systems, these directories are for stable, production software: /usr and other system directories, /usr/local, and shared network directories like /net/opt_lnx, /opt/local. Do not install by default into these locations, lest operational versions get overwritten. For Linux installations, follow the standard layout of lib, bin, and include subdirectories beneath a configurable prefix directory.

It is helpful to have a single build target for installing from within a source tree. This allows multiple software packages to be installed into a single integration test tree and run against each other, without affecting the rest of the system. When source code changes in one source tree, the build installs the changed files so that other packages will build against the latest changes from the integration tree.

It is not known what is the best practice for installing libraries and source needed to build software. Sometimes a common prefix can be used for all dependencies, other times packages will have their own separate installation tree, and those directories will need to be added explicitly to the build paths.

There is a recommended installer tool for Windows applications, used in particular by ASPEN. Ask Charlie about it.

8.10 Graphical User Interface Frameworks

The majority of current EOL (non-Web-based) applications use Qt. Should that become the preferred GUI library for EOL? Other GUI frameworks have been used over the years, but experience and current practice suggest that using Qt for GUI applications is a best practice in EOL. That could include Qt bindings in python using either PyQt or PySide.

8.11 Web Application Frameworks

There are many web applications developed in EOL using several technologies, including Ruby on Rails, Groovy with Grail, Django, HTML, Javascript, Tomcat, Python CGI, ION (IDL), PHP, Mapserver, Perl CGI, Java Server Pages, and so on. It would be difficult to standardize on a particular web technology since they change so quickly and since there is no clear trend in EOL. Nevertheless, maybe a few standard web frameworks should be considered. At the very least, before adopting yet another web application platform, consider very carefully the long-term maintenance of the application. Now there is also a variety of JavaScript libraries to choose from.

8.12 Application Configuration

There are many mechanisms and libraries to configure applications at runtime. There are Windows INI files, for which there are few different libraries in use. Qt provides a cross-platform configuration API that stores configuration parameters as INI files. Boost provides

an API for command-line and file- backed program options, as well as a serialization library. There are the standard getopt() and getenv() GNU and POSIX calls. Java now has a standard API for application configuration (Java Preferences API), besides also having core support for serialization (persistence). Some applications store configuration data in XML and use the Apache Xerces-C library to read and write XML files. Here are some guidelines despite all this variety:

- Use a text-based, line-based configuration file format, even if users will never be expected to edit the configuration files. This format can be deciphered by developers when there are problems, and different versions of configuration files can be managed and compared by revision control systems. It is a good practice to store examples of configuration files as well as test and production configurations in revision control.
- Related to revision control, when modifying a configuration in software, avoid gratuitous formatting or structure changes (like changing node order) so that differences between revisions will be meaningful. In the XML DOM this is possible by modifying the document model in parallel with the configuration changes. Other formats might require always ordering nodes alphabetically when writing them out.
- Provide as many reasonable defaults as possible, so software can work as quickly and as automatically as possible without requiring too much configuration from the user.

8.13 Commercial Tools

It seems we should identify some best practices for selecting and adopting commercial tools. There may be IDE's, memory checkers, static analysis and code coverage tools, advanced compilers, web and GUI test harnesses, database tools, and software diagramming tools which would be worth their cost to EOL, but we do not have any methods in place to evaluate them. The best approach may be to suggest that developers seek out tools that may be helpful for a particular project and then report on whether the tools should be used more widely in EOL. EOL has taken advantage of commercial tools for which there are UCAR site licenses, such as the MagicDraw UML tool and the Atlassian products.

Tools like LabVIEW and FPGA compilers are also used in EOL for specialized needs.

8.14 Code Analysis Tools

Memory checking tools have been used in the past, such as Purify and Testcenter, and they proved useful. Perhaps it is time to adopt the latest generation of such tools. CERN has benefitted from the static code analysis tool Coverity, the tool known for checking the Linux kernel. Here is a list of "more well- known" commercial code analysis tools:

- Coverity
- Parasoft
- LDRA Testbed

Likely EOL software would benefit from at least adopting open source and research code analysis tools into the development process. A quick search of the web yields a few possibilities, at least for Linux and C++. There are also many for Java and Python (pylint).

- Splint
- Mozilla Static Analysis Tools
- CppCheck

Next page: Logging Frameworks

9. Logging Frameworks

This is a quick survey of logging frameworks either used in EOL or which seem like good candidates for use.

9.1 NIDAS

https://www.eol.ucar.edu/software/nidas/doxygen/html/d0/d58/group__Logging.html

The NIDAS utility library contains a home-built logging interface. It is self- contained, and it could be possible to adopt it underneath logx to replace other software's dependency on log4cpp. That would go a long way towards trimming down the number of logging frameworks in use in EOL, and maybe the NIDAS log library would see more development if it were more widely used.

- in-house
- simple active() check before generating message
- can be completely compiled out
- fixed set of metadata: time, level, file, function, line, tag,
- not hierarchical
- only single output logger available, either syslog or stream
- multithread only
- fixed format, but fields can be suppressed

9.2 Boost.Log

As of Boost 1.54, the Boost library has an official logging framework:

http://www.boost.org/doc/libs/1_54_0/libs/log/doc/html/index.html

- any metadata
- filter on any metadata
- threading supported by templates
- support wide characters
- formatters associated with each sink
- built-in syslog, stream, and rotating file backends
- built-in timers
- short-circuits message generation without calling a test function

9.3 LOG4CPP

http://log4cpp.sourceforge.net/

- widely-used, familiar, portable API (based on Java log4j)
- categories
- hierarchical
- some standard configuration options and output formats supported (eg, xml)
- particular implementation we've used is not under active development
- Nested Diagnostic Contexts

In EOL, the logx library is a thin wrapper to log4cpp. It adds convenience methods for things like command-line options and a module-scope logging category instance with the LOGGING() macro. logx also adds scoped logging: the ability to log a message when an object comes into scope and then when it leaves scope.

9.4 log4cxx (Apache)

http://logging.apache.org/log4cxx/

I do not know of any uses in EOL, but anything from the Apache project seems worthy of consideration. It is based on the log4x API, and it should be very portable.

- requires APR
- under active development
- Nested Diagnostic Contexts
- built-in configuration

9.5 Log4J

I believe the de facto standard for Java, with concepts like Loggers, Appenders, Layouts, and Categories.

9.6 For Further Thinking

Logging should never be compiled out. If you need it during development, someone else will need it in development, and they should not have to recompile the tree with new cpp symbol definitions just to get new diagnostic logging. If it's useful during development, it will likely be useful for testing and diagnostics on operational code, but only if the logging was left compiled in. A good logging framework allows expensive message generation to be skipped if it will not be logged.

Rather than sending messages with statistics, send the statistics and let the consumer format. Sometimes consumer needed to print to a file or terminal, sometimes it was a GUI needing to fill in a label. Plus it saves space to send just the numbers. This introduces the idea of deferring formatting as long as possible, instead allowing attributes to logged which can be retrieved and handled later in some custom manner. The boost::logging library provides this kind of functionality.

It seems like logging and messaging and monitoring should converge to use the same mechanism. If there's some point in the software with valuable information to report, then the distinctions between a *log message* and an *event publication* and a *status message* become arbitrary.

Other features for which we've had a need: automatic or configurable throttling, dynamic run-time configuration, stack traces, inventory of log points (probably best generated by scanning the source).

Next page: Updating the Guidelines

10. Updating the Guidelines

It is important to review and update these guidelines based on latest industry “best practices” or “good operating practices”, as those practices evolve and as EOL learns how to apply them best to EOL projects.

10.1 Process review

Should there be a process for updating these guidelines? A periodic review?
Should there be a “facilitation team” to help those doing software development (not necessarily just programmers) put the guidelines and suggested practices to good use? Such a team would be the natural place to house infrastructure support for things like subversion, git, jira, websvn, MoinMoin and fisheye.

How should we determine when and how the guidelines should be applied to different projects? Should this report document specific adjustments for different groups? For example, CTM and DMG software projects may have different practices, and some SSG practices may not apply outside of instrument developments. Perhaps the point needs to be made that these guidelines should be kept as universal as possible, so they can be applied consistently throughout EOL.

10.2 Practice priming

New hires should be enculturated into a project on their first day. A mentor should guide them through writing production code following the specific project practices chosen from these guidelines (as described in the project checklist/overview). Old hands on new projects should do similar - either be mentored through their first day on a project, or pair-program and mutually discover/reinforce proper practices. These processes can feed into a project- practice iteration.

10.3 Action items

If there are some specific action items which should follow from these guidelines, then list them here.

Encourage more use of issue tracking, and if necessary consolidate on JIRA as the issue tracking tool. This does not need to include users at this point, but if we’re not keeping track of bug reports and issues from our users, then we’re not doing the best we can to be accountable to them.

Blog? Perhaps a shared blog, or a planet gateway to individual blogs, would be a convenient way to hear about what other developers are doing and how developments are going. It might also be a convenient way to log progress and to report on milestones, and the blogs would be available to anyone interested, including managers and other project team members (not just programmers). The blog could be part of a confluence wiki for EOL software development, perhaps migrated from the Software Engineering Wiki.

10.4 Open Questions

Issues which were considered outside the scope of this report and deliberately were not addressed.

Should there be guidelines about how to decide whether to purchase and adopt commercial software solutions? This could include IDEs, software analyzers and debuggers, and high-performance compilers. Likewise, should there be guidelines about when to use outside code?

How to publish and advertise these guidelines, and to whom (besides software engineers)?

The single point of failure problem is mentioned briefly in the shared programming practice, but what other practices have been used or could be used? Provide training sessions for other developers? Have another developer write the documentation? Get more programmers involved in bug fix phases to learn by doing? Send multiple developers to support a field project (trial by fire)?

Can anyone name a formal process they have used in the past? What worked well, what didn’t? Is there one particular process we should suggest as a model to follow, eg XP, TDD, Scrum?

As an interesting side question, Google development is restricted to three languages: C++, Java, and Python. Unidata several years ago decided to concentrate on Java. Should EOL consolidate on a few languages?

Next page: Resources and References

11. Resources And References

11.1 Staying Informed

Continuing education is important to maintain one's skills, so much so that it's mentioned in the ACM Code of Ethics. Finding better ways to develop software, whether techniques/processes or tools/technologies, is beneficial to the organization. In addition, continual learning is beneficial to one's personal state of mind. Conferences (what is EOL/CDS policy on attendance?), courses, and user groups provide good social exposure. Online resources often provide the most currency. Reading code is a useful and important way to improve skills, as is reading books (and reading books about reading code).

11.2 SEA

- UCAR Software Engineering Assembly

11.3 Stack Overflow

- Stack Overflow (collaborative Q&A)

11.4 Tech News

- Freshnews (aggregator)
- Hacker News (community-moderated bookmarks; some news, some fun, some pointers to new tech or tutorials)

11.5 Professional Organizations

- ACM
- USENIX
- IEEE Computer Society

11.6 Project Template

- Software Project Template

11.7 Joel Test

- The Joel Test: 12 Steps to Better Code
- Joel on Software

11.8 Agile on Wikipedia

- Agile software development article on Wikipedia. See especially the Agile Manifesto section.

11.9 Ptolemy Project

- Software Practice in the Ptolemy Project

11.10 Google Style Guide

- Google style guide

11.11 RAL Wiki

RAL Software Engineering Standards: SEA Presentation by Gerry Wiener and the RAL Software Engineering wiki page.

11.12 Commit Messages

- Peter Hutterer On commit messages

11.13 KDE Policies

- KDE Software Development Policies: See especially the Commit Policy, Library Code Policy, and Kdelibs Coding Style.

11.14 EOL Wiki

The EOL Software Engineering Wiki is an internal resource which only allows authorized access. It has details on specific tools, libraries, and hardware in use in EOL.

11.15 Scott Meyers

- Meyers, Scott. Effective C++: 50 Specific Ways to Improve Your Programs and Design.
- Meyers, Scott. Effective Modern C++

11.16 Design Patterns

- Gamma, Helm, Johnson, Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software.

11.17 UML Distilled

- Fowler, Martin. UML Distilled.

11.18 Head First

- Pilone, Dan and Miles, Russ; Head First Software Development, on Safari Books Online. See Ch 1, pg 10, “Getting to the goal with ITERATION”

11.19 Code of Ethics

- ACM Code of Ethics

11.20 Why scientific programming does not compute

- Computational science: ...Error: why scientific programming does not compute. Published online 13 Oct 2010, *Nature* 467, 775-777 (2010)

11.21 More Books

- Warren, Henry S. Hacker's delight.
- Bentley, Jon Louis. Programming pearls.
- Spinellis, Diomidis. Code reading.
- Kernighan, Brian W. and Rob Pike. The practice of programming.

12. Software Project Template

This is a template for an overview and checklist of a software development project. The intention is to provide a quick summary and an index of important links.

There is a table rendition of this checklist for the CAMS project: CAMS LabVIEW Software Development.

12.1 Software Project Overview Template (SPOT) Fields

Name:

Example: Hello World

Description:

Example: As part of the observation sensing network, this software will monitor hardware sensors and display “Hello World” whenever the instrument is being observed by someone, and otherwise it will do nothing.

Inception date:

Just to give some idea of the age of the project.

Current status:

Use this to distinguish lifecycle phases, from inception to active development to maintenance. The status fields used in the software inventory might already suffice for this.

Release information:

Give the current version, if any, where downloads are available, perhaps upcoming release dates.

Software process:

Identify the parts of the software guideline chosen to be followed for this project, or name a specific process to be followed, like XP or TDD or pair programming. The point is not to restrict or formalize the process, only to think about it ahead of time and to give developers on the project a picture of how the process should work.

Revision control links:

Provide URLs to subversion or git repositories for this software which developers and others should use to access the source.

Links to other software artifacts:

For example, this could be links to web documentation or wiki pages for discussion.

Developers:

List the programmers on this project, who are not necesarily all software engineers.

Users:

Name the intended users of this software. Again, the idea is to be clear and complete about for whom the software needs to work. This could be only a few specific people if it's a one-off data analysis, or it could be the entire research radar community. The users will be scientists or engineers or technicians or the general public, but almost absolutely the users will not be just other software engineers.

Domain experts:

Name specific people who will consult on technical matters related to the software's domain. This could be one or several people. The domain experts may also be users, but not necessarily. These are the people the developers must go to to resolve questions about the requirements, the problem being solved, or the vocabulary and concepts specific to the domain. If the software will process radar data, the domain experts likely will include a radar scientist or engineer. In many cases the developers might also be considered domain experts. The point is that software development requires intimate understanding of the problem domain, and it's important to recognize the role of domain expert and to rely on that resource for development.

Requirements overview:

For simple projects, just state the requirements. For larger projects, provide a link to the requirements discussion or documentation. Obviously it is important to be able to state the requirements to keep the big picture in mind and as a check on the direction of development. For iterative agile development processes, these are not the requirements of each individual development stage. These are the guides by which the final product will be judged whether it works or not.

Design overview:

Provide the basic approach to the design or a link to it. As for requirements, this just gives an idea of the fundamental design. The larger the project, the more complicated this can be. Note that it is useful to document critical design alternatives which were considered but rejected.

Security issues:

Don't forget to think about security implications: data security (including redundancy and access restrictions), system reliability, operator authentication, logging and auditing, user privacy, and so on.

Related projects:

Name projects or provide links to projects which are similar or somehow related to this project. It's important to think about this to identify what parts of the problem might have already been solved or to find out what has been learned from past mistakes.

Implementation overview:

These are nuts and bolts questions to answer at the beginning of the project. They give an idea of the development and deployment environments, and they are a checklist for infrastructure which should be in place when the project begins.

Programming languages:

C++, Java, Ruby, Perl, Python, ad infinitum

Platforms:

Linux embedded, Linux desktop, web server, Windows, Mac, ...

Data formats:

GUI framework:

Build system:

Examples: SCons, make, qmake, automake, cmake, Visual Studio, ...

Sources for test data:

It is important to identify test data from the start, since sometimes it will take a while to obtain them, and sometimes the design of the instrument and the software must specifically be designed to accommodate test data. Virtually all EOL software projects will either consume or produce data, or both, and there must be some way to verify the operation initially, then later also verify that maintenance and improvements do not produce unexpected results. For research instruments collecting one-chance-only observational data in the field, it is especially critical that data not be lost or munged on their way through software.
Having test data easily accessible, perhaps even packaged with the software, makes it easier for others to try out the software and run their own tests.

Test frameworks:

There are many testing frameworks and utilities available. Even if a specific framework is not chosen, it should be possible to identify how tests will be automated, such as with scripts or built into the program.

Other frameworks and libraries:

Examples: Boost, OpenDDS, Qwt, Netcdf, Jambi, and so on.

Links to automated build (CIT) reports:

Examples: buildbot

Links to issue tracker:

This is a reminder to setup a place to record problem reports and feature requests. Once that's created, this is a convenient place to put a link to it.
Examples: JIRA Bugzilla Github