WISE Science Data System Software Management Plan

Contents

1 Signatures

____

Roc Cutri WSDC Manager

____

Tim Conrow WSDC Architect

2 Revision history

2006-03-20: Draft 1 (was version 0,1)

2007-12-07: Draft 2: Imported from Word file. Added Coding Standards section. Many small revisions and rearranging.

3 Document number

WSDC D-M003

4 Scope

This document is about software management, i.e. the handling and disposition of source code and any binary files derived from them. It will cover the following topics:

- Report tracking

- Software and parameter revision control
- Software delivery
- Operational system configuration control, including testing
- Software backup

These topics, though important, are beyond the scope of this document:

- Computing hardware and infrastructure configuration control
- Document revision control
- Reference data (e.g. calibration data) version control

5 Problem Reporting and Tracking

In software management, a problem report (PR) tracking system provides traceability and documentation of the full life cycle of a problem report, from initial report to resolution. It is used for reporting and documenting the characteristics and resolution of:

- Bugs – software doesn’t perform as documented.
- Deficiencies – the software as designed is inadequate.
- Change requests – alter designed behavior.
- Feature requests.

5.1 Reference PR System: Bugzilla

As a reference PR system, we will assume the WISE WSDC uses the open source Bugzilla suite of tools (http://www.bugzilla.org/) for problem tracking. Whatever tool is selected will have the key attributes discussed below. Bugzilla is used as a reference system – i.e. an existence proof – to show that systems meeting the WSDC's needs are readily available without cost. The WSDC may actually use Bugzilla, or may select another compliant system.

Bugzilla is a collection of tools that provide access to a problem report (PR) database. These tools are most conveniently utilized through the Bugzilla web interface web. An example of the web interface can be found at http://landfill.bugzilla.org/bugzilla-2.20-branch/.

Via the web interface, users can

- Authenticate
- Create PRs
- Update previous PRs
- Query the PR database,
- Create summaries of all or a cross-section of PRs
- Trace the history of all action taken on a PR.

When submitting a PR, a submitter categorizes the PR by software category and supplies detailed information, possibly including sample code, data files, environment, etc. Once a PR is completed a predetermined stuckee for each category is automatically notified by email. Either the submitter or the stuckee may subsequently modify the PR.

The PR database may be searched using any of a dizzying array of conditions and filters.

5.2 PR Resolution

All PR’s should be resolved, which may mean the bug was fixed, or that software was used incorrectly (user error), or a feature request was implemented or denied, etc. After resolution, the PR should be formally closed in Bugzilla (i.e. the PR state is changed) and the stuckee should document how it was resolved and why.

5.3 Which Problems Should Be Tracked?

Any problem with operational software should be entered into Bugzilla if the resolution will take longer than a few days, or if the user explicitly wishes to create a problem record, e.g. to document a series of small but ongoing problems with a design.

During development, either prior to the existence of on operational system or in parallel with it, users and developers may enter PRs whenever tracking an issue would be useful. The only limitation is to avoid making PRs so trivial that it is merely a waste time dealing with them (e.g. It's probably excessive to enter document typos as a PR).

5.4 Revision Control

Revision control refers to the ability to label, document, track and recall prior states of a file or related group of files. Any file can be revision controlled, but only files containing source code or parameters are considered in this document.

5.5 Reference Revision Control Tool: Subversion

FThe WISE WSDC uses Subversion (http://subversion.tigris.org) – SVN for short -- for revision control. SVN maintains a central repository into which users check files in and out. For each check-in SVN will record all changes from the previous state of the file, as well as meta-data about the check-in, such as the date and time and any comments from the user. Whatever tool is selected will have the key attributes discussed below.

5.6 Revision Number and Release Tags

SVN maintains a revision number that uniquely identifies the checked-in state of each file. This number is retrievable from the repository as part of a file’s check-in meta-data, and it can be embedded in any plaintext file by including a special text string that SVN will replace with the current revision number upon check-in.

The revision number is strictly file-local; it does not by itself say anything about the developmental state of a file, e.g. whether it’s ready for release. Information about developmental state is explicitly supplied as part of the configuration control process (see below) using release tags.

Tags give a user-supplied name to the state of the entire repository or any part of it and can be applied by a user any time a particular state of (part of) the repository is of interest, such as prior to a release. Thus, a given tag will allow a user to retrieve a particular point in the revision history of any group of files so tagged. Each tagged file will have its own revision number since it has its own, separate history. Thus a tag essentially marks a cross-section of revision numbers within the repository. As an example, prior to a release, the text for a tag will be composed (e.g. “wsdc-release-1_0_2”) and applied to the appropriate source code within the repository. Afterwards, all code associated with this release can be retrieved and examined at any time in the future.

5.7 Branches

Changes to source code which are too speculative, broad, or otherwise risky to be on the main development track (called the trunk in SVN) can be split out onto a side track (called a branch) and developed in parallel. If/when a side track reaches sufficient maturity (as decided by the CCB) it can be merged back onto the trunk. As an example, after a particular release has become operational, further development for the next release will occur on the trunk. But if bugs are subsequently discovered in the ops system, a branch can be split off to work on the bug fix by creating a branch at the release tag for the prior release. That fix can then be applied to the operational system and separately merged into the trunk so it is accounted for in future development.

5.8 Repository Browsing

The SVN repository can be browsed or searched via a web CGI program. Users may access the complete revision history (any state of any file which is revision controlled), including meta-data, about any checked-in file. Repository browsing will be a convenient way to access all WSDC software and parameters without having to navigate the WSDC filesystem.

5.9 What Is Revision Controlled?

All locally developed source code necessary for operations or intended to be used by anyone other than the author should be revision controlled. Also, a user may place any plaintext file desired under revision control for any reason.

Source code developed elsewhere should not be revision controlled as long as it is not modified locally. Externally developed source code that is locally modified might need to be revision controlled. This is an issue, which the CCB (see below) should address on a case-by-case basis.

5.10 Revision Control Instructions

Here are some instructions a user should follow in maintaining a file under revision control:

1. Maintain a repository-friendly directory hierarchy. Details elsewhere.
2. Put the revision ID replacement tag in all files so the revision ID string is present for review in all code and binary objects.
3. Add new software to the SVN repository.
4. Check changes in at least once per week as well as when necessary to prepare for a new delivery (see configuration control below).
5. Periodically do a checkout of all code to test that the repository and a user's work areas are in sync.
6. Add tags whenever they might be useful for future reference (e.g. “before-big-bug-fix” etc.), but always prior to deliveries and releases in accordance with configuration control rules.

6 Build Tools

Build tools are the command line tools used to take source code and produce an executable application or a library and to deliver it to a specified delivery target. All developers at the WSDC will use a common set of build tools, making the build and delivery process uniform.

6.1 Reference Build Tool: make

Building and delivery at the WSDC will be done using the common command line build tool “make.” Specifically, the implementation provided by GNU make is assumed. Other make implementations may be incompatible. Whatever tool is selected will have the key attributes discussed below.

6.2 Use of the Build Tool

The operation of the make tool is controlled by a "Makefile" which contains a specification of targets (things to be built), the dependencies (prerequisites) for that target, and rules used to resolve (provide) the dependencies. Each code unit, or closely related group of code units, will have its own Makefile occupying the same directory as the code.

Makefile can refer to other Makefiles and can thus cascade or chain, potentially building an entire system.

See below for how the build tool will be employed in the delivery process.

6.3 Makefile Templates

Delivery target isolation (see below) and a uniform build and installation process are enforced through the mandatory use of Makefile templates by all developers. These templates are a small set of files of rules, macros and targets to be used by the make utility. All users include these templates in their Makefiles. These Makefiles, in turn, form the complete set of rules by which code is built and delivered when the make utility is invoked.

The make utility employed at the WSDC is GNU make. Making utilities with a different origin may or may not work.

Here are the main areas controlled by the Makefiles:

- Which source files depend on which other source files, and the order in which they must be compiled;
- What external standard libraries are used and their locations;
- Which internal libraries are linked to and their locations (usually the same delivery target);
- The set of compiler flags employed for compilation;
- What is delivered, and where.

Users may override some of the standard behavior, but such overrides are subject to review.

7 Delivery

Delivery is the process of integrating developer – local copies of executable programs and associated files (libraries, include files, etc.) derived from configuration controlled source code, into a common area for use by testers or users. The common area may be an operational release, a public development area, or a private area for integrated testing (see Delivery Target Isolation below).

Delivery proceeds as follows:

1. Stabilize – halt code development and document changes
2. Build – compile source to binary, if necessary
3. Unit test – test at the unit (component) level
4. Check-in – update the source code's revision number
5. Install – move the program to the delivery target
6. Validate – test the integrated code

Both the build and install steps are carried out through the use of standard tools based on the "make" (or the equivalent) utility and makefile templates (see Makefile Templates above). Unit testing is testing on small units of coding prior to delivery – e.g. checking if a change to a particular subroutine worked or not, as opposed to testing that subroutine as part of the fully integrated system. Validation is discussed below.

7.1 Who Delivers?

Software authors are responsible for checking-in their files to the SVN (or equivalent) repository) prior to deliveries. Building and installation to test or development areas may be carried out by a designated build specialist, but can also be done by any developer. Building and delivery to integration and operational areas will be done by a build/delivery specialist in consultation with the CCB and all developers.

7.2 Delivery Target Isolation

In order to test new, developing configurations, developers must be able to integrate their code with code from other developers to allow extensive use of all its features and interfaces without affecting an operational release. This calls for a delivery process that controls dependencies such that delivered programs depend only on libraries, data, parameters, and other programs that are part of the same delivery. The WSDC delivery process ensures this dependency isolation (see Makefile Templates below).

All deliveries specify a delivery target within which all dependencies will be resolved. Any number of separate targets may coexist. Here are the most commonly employed delivery targets and their uses:

7.2.1 The Ops Target

Ops deliveries produce the operational system as used for all WSDC activities other than testing and development. Greatest care is taken in the validation step for ops deliveries and the CCB is always consulted.

There are actually two parallel ops deliveries; one accessible from development machines, and a separate one used by the operations servers. These deliveries are identical.

7.2.2 The Int Target

Deliveries to the int target are made prior to a delivery to ops, for integrated testing of the next ops configuration. The int target thus has only a short lifespan. Use of the int target is controlled by the CCB.

7.2.3 The Dev Target

The dev target is used for ongoing development of the next operational release. Only minimal CCB involvement is required for delivery to the dev area. It is, therefore, an unstable area usually containing the most up-to-date versions of all code.

7.2.4 Other Targets

Other delivery targets can be created at will by any user for special purposes, such as testing new, highly experimental, features or large changes (refactoring, etc.)

== Operational System Configuration Control

For the purposes of this document, the “WSDC system configuration” is the state of all source code (and any derived binary files) which implements the capabilities required at the WSDC. In this context, configuration control is the ability to precisely maintain or restore the behavior of the system until change is deliberately called for, and carefully controlling the change process.

Exerting this control requires these elements:

- A Change Control Board (CCB), which decides when the configuration may change;
- Revision control (see above), to provide traceability of changes and thereby a means of uniquely identifying and, if necessary, recalling a particular state of the configuration;
- Delivery system, to orchestrate the changeover to a new configuration.

Revision control and delivery have already been addressed. The CCB and build validation are discussed below.

7.3 The CCB

The CCB is responsible for deciding when the WSDC system configuration may change and establishing when the change should occur. The criteria for when it is appropriate to alter an existing configuration will be situational, but will include:

- Importance (does it add or fix something important);
- Scope (how much of the system is affected by the change;
- Risk (what might break);
- Nearness of critical mission events;
- Quality (has the change been carefully considered and tested)
- Cost (utilization of project resources).

In addition to controlling the configuration of the operational system, the CCB will also modify the change procedure (see below) as necessary, and decide if development of a proposed change should proceed on the main development track, or a separate branch (see revision control above).

7.4 Configuration Validation Using an RTB

Newly installed configurations are validated through the use of a regression test baseline (RTB) – i.e. running a series of standardized tests on the new configuration and comparing the results with those of previous configurations. For ops deliveries, this comparison is considered by the CCB (see below) as part of the approval process.

The RTB is not a deep and thorough test of all aspects of a particular configuration, but confirms that a set of basic capabilities are present to confirm that the delivery was complete – i.e. it's the delivery process which is being tested, not the code itself. Thorough code testing is assumed to have occurred during the development process.

7.5 WSDC Operational System Build Procedure

Here are the steps to replacing an operational build with a new one.

1. Halt new development.
2. Convene CCB to review changes from current operational state.
3. Make any approved hardware and infrastructure changes to the ops machines. Any such changes will already have been made and validated on development machines.
4. Deliver to integration area on ops machine.
5. Validate int delivery (run RTBs)
6. Tag SVN repository with release candidate tag.
7. Get CCB final approval.
8. Suspend operations

9. Deliver to operational area from a fresh checkout using the release candidate tag. Deliver to both the ops and dev servers so the machines are kept in sync.
10. Check dependencies. Check all scripts, binary programs, and libraries for illegal references outside the ops delivery area.
11. Validate ops delivery (run RTBs).
12. Tag SVN repository with final release tag.
13. Restart operations.

8 Software Backup

The goal of the WSDC software backup process is to ensure the ability to reform an operational system and continue operations and development within TBD days of an ordinary outage (e.g. a disk failure) and within TBD weeks of a disaster (e.g. loss of the WSDC facility). Details of the recovery process are beyond the scope of this document.

8.1 Daily Copying of the SVN Repository

The repository will be copied over the network to backup locations on disk both within the WSDC, and, via sftp, to a remote location. These network backups will occur automatically overnight.

Since all internal software necessary for operations is kept in the repository, this alone should be sufficient to rebuild all necessary executable programs. Calibration data and other reference data will need to be recovered from on- or off-line WSDC backup media.

8.2 Daily Partial And Weekly Full Backups To Tape

As part of the WSDC facility backup system, all code and programs will be backed up along with all other files on a nightly basis. Off-site copies of tapes will be maintained.

9 Coding Standards

9.1 Compilers/interpreters

These are the compilers versions currently accepted for use:

- g95 4.0.3
- gcc 3..4.6 or higher.
- Perl 5.8 or higher.
- IDL 6.2 or higher.

9.2 Code Style

We won't be religious about these. If they rub you the wrong way, don't feel constrained, but at least consider these style points. They are more important for code which more than one person will read. If you want to debate any of them, that's fine, but we should probably keep the religious wars out of real meetings and do it over lunch or something.

- Avoid global variables. Use a dynamic (malloc'd) or automatic context structure and pass it around as a parameter. You'll be happier, I promise. If a problem submits to resolution much easier with globals, fine, but name them with a leading capital letter so they are visually distinct.

- Use all lower case for non-global variables. Separate words in a variable name with an underscore. I know some people love camel-casing their variables, but
- Fit all lines within 79 columns so they can be viewed without wrapping on an 80-column-wide terminal or editor display. I know people find this restrictive, but considering that some people need larger text and will be editing on small laptops, the amount of screen real estate might be more limited than you think.
- Use succinct comments to describe anything non-obvious and to label the important sections of your code. E.g. this is NOT a useful comment:

```
  ++i; /* Increment i */
```

- Do not let comments create so much visual clutter that the code is hard to see, but a new reader should be able to understand the overall flow of a program from the comments.
- **Do not lie with comments.** Keep comments up to date and make sure they're accurate. If being accurate would take too much time, no comment is better than a misleading comment.
- Mostly use /**/- style comments, though C++ //-style comments are good for commenting out code and for brief, trailing comments.
- Stick mostly to 2- or 4-space indents, and don't use tabs.
- Only use bracketless if/while/for blocks if they fit on one line. E.g.

```
if(foo) bar; /* OK */
if(foo)      /* Not OK */
  printf("A really long message that really needs "
         "two lines\n");
if(foo) { /* OK */
  printf("A really long message that really needs "
         "two lines\n");
}
```

- As demonstrated above, keep arguments for function calls always left-right between the parens. E.g.

```
i = foo(x,y,z,
    a,b,c);      /* NO! */
i = foo(x,y,z,
        a,b,c);  /* OK! */
```

Follow similar rules for breaking arithmetic expressions across lines. Do so in logical places:

```
i = a + (b + 2*c) +
    2*pi             /* Better than ... */
i = a + (b +
    2*c) + 2*pi
```

Etc.

- Make nearby similar statements and parts of statements line up in logical ways. E.g.

```
i       = 10;
foo     = -3;
frizzle = 999;
```

- Space out operators and arguments so they are visually uncluttered and order of operations is easy to see. E.g.

```
i = a + b*c - exp(-0.5*(x*x + y*y)/sigsq); /* OK */
i = a+b*c-exp(-0.5*(x*x+y*y)/sigsq);       /* Eew */
```

- Use one of these bracing styles:

```
if() {
}
```

```
if()
{
}
```

- Except for very short and simple routines, don't initialize in the declaration statement, UNLESS you define ALL your initialized variables one per line.

```
int j,i=0,k;   /* NO! */
```

```
int i = 0; /* Yes! */
```

```
int i;
... later ...
i = 0;     /* Yes! */
```

- Take advantage of modern C syntax and keep a variable's declaration lexically close to its use:

```
int i;
... many lines of code ...
if(foo) i = 0 /* Too far */
... many lines of code ...
int i;
if(foo) i = 0; /* Much closer! */
```

- Avoid repeated code. Make a macro (if very simple) or abstract into a function, which could later migrate to a library.
- Avoid unusual gcc coding constructs and syntax not widely supported. Some code might have to migrate some day. Don't get too hung up on portability if it wastes time, but do try to isolate and comment non-portable code if it's fairly easy to do so. E.g. code that depends heavily on word size (like some bitops will), or I/O or

DBMS-driver code.

- Use modern function prototyping and always supply a return type for a function. E.g.

```
int foo(int a, double x)    /* Yes */
{ ...
```

```
int
foo(int a, double x)        /* OK too */
```

```
foo(a,x)                    /* No! */
int a;
double x;
{ ...
```

Along the lines of fatherly advice:

- Make variable names clear as to their purpose, but also as short as possible. Really long names rapidly make code unreadable.
- Compact code is preferred over long-winded code since apprehension is easier without scrolling. But too much density makes the eyes glaze, so seek a nice middle ground.
- Treat pointers like dynamite: They're too useful to avoid using, but you don't want to mishandle them!

9.3 Error Handling

- Check the return values of all system calls and other functions. printf and fprintf are exceptions.
- Make error and warning reports contain enough info to identify the location in the code and the specific values that may have caused the problem. E.g. include the file name and the system error if an open fails:

```
fp = fopen(file, "r");
if(fp == NULL) {
   fprintf(stderr,"*** Foo: Open of '%s' failed; %s.\n",
           file,strerror(errno));
   exit(1);
}
```

- Always send fatal and warning error messages to stderr. Prefix fatal errors with "*** PROG-NAME:" and warning messages with "=== PROG-NAME:".
- Make sure your subroutines return error codes and print useful error messages, and make sure the caller always checks the error codes returned. Think carefully about whether a given subroutine should print a message and terminate on an error, or just return to the caller with an error code.
- Terminate with non-zero exit codes for fatal errors. E.g. 'exit(1)'.

9.4 Parameter Handling

- Make any program executable entirely with command line parameters, except for voluminous data which should be in files.
- If possible use my standard parameter parsing code for Perl, for C and C++. It fun and easy.
- Don't rely on environment var.s to get info into a program. It's OK to use them, but do not demand it. Allow env. var.s to be overridden on the command line and *always* have internal defaults in case a needed parameter isn't given.
- Don't hard code pathnames or filenames unless they can be overridden with parameters.
- Parameter files are OK, but command line parameters are better. Remember that your code will ultimately be run by a Perl wrapper, which will be happy to construct very long command lines without complaint. The Perl parameter handler will be able to read parameter files and construct explicit command lines for your code from them.

If you need to use your own parameter file, let's talk about how to do it in a robust and uniform way.

9.5 Code Building

- Use the makefile link and deliver your code. If I need to expand or modify these macros for you to use them easily, this is better than everyone using a different means of building their code. These macros WILL SAVE YOU TIME!!
- Put any libraries, include files, static data, etc. that your code needs in the standard directory hierarchy so your code can run from anyone's desktop. (The makefile templates used for code builds automate such deliveries.)
- Try to put any functionality that can be generalized or may be needed by other programs to a library, preferably a runtime lib.
- Prepare for the day we start revision control and build control by organizing your code like this:

```
. One library per directory.
. Only closely related executables in one directory.
```

- Put a static SVN revision substitution string somewhere in every file. E.g.

```
      static char __version[] = "$Id$";
```

- Try to get clean compiles: i.e. no warnings.

Retrieved from "http://wise.ipac.caltech.edu/wiki/index.php/Dev::Software_Management_Plan"

- This page was last modified 2007-12-07 19:34:09.
- This page has been accessed 6 times.
- Privacy policy
- About WiseWiki
- Disclaimers