# Software Engineering Design

P.J. Teuben, N.S. Amarnath, & M. W. Pound

Version 0.96
September 24, 2003

**Abstract**

This document describes all CARMA Software Engineering aspects (combining two earlier documents named *Coding* and *CarmaBuilding*) and serves as the final CDR for the **Software Engineering** package.

It discusses how the code is organized, built, documented, and maintained. This also includes CVS practices, such as branching, versioning, distribution, and the bug reporting system, etc.

**NOTE:** Items to be clarified/resolved can be found by searching for ???

## 1 Introduction

CARMA software will be developed using object oriented languages (C++, Java), with some low level microprocessor code in C, and some legacy code in Fortran and C. Coding standards have been defined in order for developers to easily understand each others code accross the project. Documentation will be done using the automatic documentation extractor doxygen. Distributed development of all source code will be done using CVS, though a fair number of support libraries will come from external packages not under our CVS control. An easy mechanism exists to install and maintain these packages.
A standard environment is defined along with our software, such as the OS version, compiler, libraries, build tools, to ensure a stable build. It is expected that we freeze these for a number of years.
Currently this standard environment is Redhat9 linux, with the following annotations:

- kernel 2.4.20 (see also `CARMA/conf/etc/kernelconfigs`)
- gnu compiler 3.2.2 (though 2.95.3 used to work)
- , glibc 2.3.2
- various development tools were used, and of course not well checked on earlier releases how well they work.
  - gnu make, version 3.79.1
  - doxygen, version 1.2.14
  - autoconf, version 2.54 (2.13 also works)

## 2 Computer types

The CARMA system will run mostly on Linux, and provide us with an environment to control the telescopes, do an observation, monitor and inspect data in real-time and even do some calibration/mapping afterwards (maybe even partial maps in real-time).
Although we will probably be running on mainly one architecture (Intel/AMD) within the same operating system (Redhat-Linux), the build system must ensure that is trivial to build CARMA on a number of compiler/operating system combinations[1] The system will be **configured**[2] in a number of modes, as determined by each **type of computer**[3] on the system:

- one for the ACC (Array Control Computer),
- one for each AC (Antenna Computer), possibly in the 3 different flavors, aptly named `BIMA1..BIMA9, OVRO1..OVRO6, SZA1..SZA8`.
- two for each CC (Correlator Computer). the SZA will initially have a different correlator from the OVRO/BIMA combination. On the long term this might become one CC.
- one for the DC (Downconverter Computer)
- one for the CIPC (Calibration and Integration Pipeline Computer)
- one for the DBC (Database Computer)
- one for the ARC (Archive Computer)
- one for the LLC (Line Length Computer)
- one for the LRC (Lobe Rotator Computer)
- one for the EC (Environment Computer) handling weather related things
- several for the UIC (User Interface Computer). This is also the version in which we expect users can run on their private workstations/laptops at remote places, not just the site or valley floor?
- one of the OC (Observing Computer) - or is this the same as UIC???

In addition, a number of hardware modules must be able to run in emulation mode, in order to test some of the run-time behavior of the system off-line for developers. The system must also be able to be built in an incremental way, for example by inheriting most components from an existing build, and only compiling a small number of components with for example debugging turned on.

[1] arguably on e.g. solaris using their native compiler, or the free intel compiler on linux
[2] this is not the `autoconf` based one, but a CARMA configuration, details TBD
[3] see also Carma Software System Diagram, on
`http://www.mmarray.org/project/system/CARMASoftwareSystem_files/CARMASoftwareSystem_frames.htm`

# 3 Directory Structure

A CARMA code tree will look as follows. Directories preceded by a (*) are created during the build process, and by itself can be planted in **the** `/carma` location. Directories marked with (**) can optionally be used during the build process, but can be discarded aftwards. The directory structure in some sense resembles that of the familiar unix /, /opt or /usr/local tree, in the sense that tools like autoconf can use `--prefix=$CARMA` if they want to leave their libraries, binaries etc. within the CARMA tree. Given the number of configurations (read: types of computers) we may wind up with, this greatly simplifies keeping all needed software within one mountpoint on a disk (e.g. /carma).

```
$CARMA/                         # Root (module) name for Carma Software Toolkit %$
    carma/                      # a large tree with all our (namespace carma::) code
    trial/            ???       # a development tree (namespace carma::)
        ...
    conf/                       # configure (autoconf) related things
        etc/                    # config files for the CARMA build system
        opt/                    # simple example package codes
            cppunit/
            log4cpp
            orbacus/
    ...
    scripts/                    # will be in bin/ for use, may have .in versions?
    doc/                        # our own documentation

*   lib/                        # libraries, shared libraries [created w/ install]
*   bin/                        # binaries and scripts [created w/ install]   --help
*   include/                    # include files [copied from carma/ during install]
*   jar/                        # java
*   man/                        # standard unix man pages
*   html/                       # html tree, including doxygen
*   etc/                        # runtime configuration files
*   var/                        # runtime locks, logfiles etc.
*   tmp/                        # any tmp work script and programs can use
**  opt/                        # optional external modules
        OB-4.1.2/                 # TBD during the configure stage
        log4cpp-0.3.4b/
        cppunit-1.8.0/
        miriad/                 # (could also live else on /)
        pgplot/                 # though pgplot also lives inside miriad
...

$CARMA_PKG/                     # directory with all your external package tar balls %$

$CARMA_TOOLS/                   # directory with all the compiled external packages %$
```

# 4 Code Building

The build system must be able to handle the various modes in which we will configure and use the CARMA code. We plan to do this during the final run-time configuration, not the initial configuration and compiling. This has the advantage that any executable or library can in principle be used on any **computer type** we have, but its behavior change willl depend on a run-time configuration. The build procedure is basically the following (the script `conf/install_all` will normally be used with the appropriate options):

1. obtain the CARMA code, via CVS (CVS tags are used to get certain releases or branches) ("`cvs co carma`")
2. configure the code and determine which alien packages were missing. ("`configure --with-PACKAGE-dir=DIR`")
3. install missing alien packages (each alien package is different, but basically something like"`configure; make ; make install`")
4. compile all the carma libraries and executables, install all the documentation ("`make all`")
5. configure the environment for the particular **computer type** (and subtype) choosen

Here are some examples how the `install_all` script can be used to build a system. Since currently only builds from CVS are possible, a CVSROOT environment variable is required (or use the `cvsroot=` command line parameter).

- CARMA as well as CARMA_TOOLS are installed in the same directory. This is not really recommended.

  ```
  install_all carma=/home/carma carma_tools=/home/carma
  ```

- CARMA_TOOLS is installed, though a dummy CARMA environment is needed to do this. It can be discarded afterwards

  ```
  install_all carma=/tmp/carma_tmp carma_tools=/home/carma_tools do_carma=0
  rm -rf carma_tmp
  ```

- CARMA is installed, and a previously installed CARMA_TOOLS is used.

  ```
  install_all carma=/home/carma carma_tools=/home/carma_tools do_tools=0
  ```

Reconfiguring a computer to switch types is not foreseen, but it may very well happen that subtypes (e.g. computers in BIMA9 may be replaced, with maybe even one that was in OVRO6 or SZA8).
It is proposed that all CARMA binaries accept the command line option `--help` to present some form of reminder of what it does, and what the other command line options (if any) are. Those that use `carma::Program` class already get this for free, but a number of script and tools should adhere to the same uniform standard of supplying some minimal form of this type of *inline help*.

## 4.1 configure

We use GNU `autoconf` to configure the CARMA tree for a particular setting of packages. For this we write a `configure.in` file, with directives that a program `autoconf` will transform into a shell script `configure`. This script will then investigate your environment for the presence/absence of things such as compiler features, packages, etc.etc. Based on this it will transform user written files such as `Makefile.in` into `Makefile`, `carma.h.in` into `carma` etc.etc. such that all system dependant settings are determined, and a `make` can now proceed.

```
% configure --help
...
Optional Packages:
  --with-PACKAGE[=ARG]    use PACKAGE [ARG=yes]
  --without-PACKAGE       do not use PACKAGE (same as --with-PACKAGE=no)

  --with-all-pkg=DIR      set a default directory prefix for all packages (none)
  --with-carma-pkg=DIR    Directory where carma_pkg with tar balls is located
  --with-pgplot-dir=DIR   Directory where PGPLOT was installed  (or use: $PGPLOT_DIR)
  --with-orbacus-dir=DIR  Directory where ORBACUS was installed (or use: $ORBACUS_DIR)
  ...

  --with-carma=CARMA      re-use an existing $CARMA tree that is not this one to make CARMADEV

  --with-debug            Turn debugging on (default is optimized)
  --with-cppflags         extra C++ flags..
  --with-cxxflags
  --with-cflags
  --with-ldflags

  --with-doxygen

  --enable-offline        no hardware connected, run in emulation mode where applicable
```

The normal behavior of resolving the location of an alien package is, from highest priority to lowest:

1. an option to configure, e.g. `--with-orbacus-dir=/tmp/orbacus`
2. an environment variable (left over from a previous CARMA, e.g. `ORBACUS_DIR`)
3. pass a global default option to configure, e.g. `--with-all-pkg=/opt/carma_pkg`
4. a number of reasonable predefined search locations by autoconf, e.g. `/usr/local/orbacus`, `/usr/orbacus`, `/home/orbacus`

## 4.2 Build process

After configuration and setting up the (shell) environment , the top level Makefile will hierarchicallly call all Makefile below, as determined by each Makefile. This means that on each (module) level, the owner of that Makefile must ensure that for example a "`make clean`" command not only cleans what needs to be

cleaned in the current directory, but also below. The follow standard makefile targets are used for this tree walk process:

```
all         builds, in the correct order, the libs, bins, and tests

libs        normally the first tree walk, where the libraries are built
bins        normally the second walk, building binaries
tests       compiles and runs all the tests
include     include files to be copied to CARMA/include
doc         any non-doxygen things n
clean       clean up files to ensure a clean rebuild
```

# 5 Alien Packages : CARMA_PKG

We identified many already existing packages that we will be using straight off the shelves. We store them as (compressed) tar balls in a directory referred to as `$CARMA_PKG`. Examples of this are Orbacus, log4pp, PGPLOT, libwcs, etc.etc. The build system will be able to automatically detect if they already are present on the system, and happily assume those. Since it is a fairly large (and still growing) collection, the build system prefers to place them in one common location that can be inside the CARMA tree, but is preferred to be in a different location, referred to as `$CARMA_TOOLS`. For consistency, all of these tools should be under the same directory tree.
Currently used are:

1. `c++`: gnu compiler suite (we designate a specific version to be working, currently 3.2.2) This in case your development machine does not have the correct version, and there is a compiler version conflict.
2. `orbacus`: CORBA/IDL (commercial)
3. `notify`: CORBA notifications (commercial)
4. `log4cpp`: logging
5. `cppunit`: unit testing
6. `janz`: CAN bus drivers (commercial)
7. `gsl`: Gnu Scientific Library, for numerical work
8. `db`: Berkeley db. needed by CORBA's notification services

As mentioned before, the directory `$CARMA_PKG` contains all the external packages we need to fully build CARMA from scratch, assuming a reasonably well populated linux box with development tool installed. This external package repository needs to be kept in sync with the master site (OVRO), for which we use use an rsync based script, `CARMA/conf/carma-package-sync`.

`$CARMA/conf/carma-package-sync}`.

Two configuration files are currently used to control the location and availability of alien packages:

```
# File:    conf/etc/sites.dir
#   SITE directory, with caching locations for packages
#   see packages.dir for package information
#
# Columns:
# 1: your site name
# 2: FQDN
# 3: carma_pkg root directory
#
#   this special one is the master, the rest are all slaves
#   to be updated with rsync (or some mirroring tool of your choice)
master  harkless.ovro.caltech.edu   /sw/carma_pkg
#
cdrom   localhost                   /mnt/cdrom
home    localhost                   /home/carma_pkg
ovro    harkless.ovro.caltech.edu   /sw/carma_pkg
umd     grus.astro.umd.edu          /n/grus/carma_pkg
uiuc    ftp.astro.uiuc.edu          /tmp/carma_pkg
bky     astro.berkeley.edu          /tmp/carma_pkg
uchi    polestar.uchicago.edu       /tmp/carma_pkg

# File:    conf/etc/packages.dir
# Purpose: optional package configuration management
#
#   packagename  URLfmt   version       size-source (MB)  size-compile (MB)  size-install (MB)
#
blitz++  http://www.oonumerics.org/blitz/download/releases//blitz-%s.tar.gz       0.6   0   0     0
cfitsio  ftp://heasarc.gsfc.nasa.gov/software/fitsio/c/cfitsio%s.tar.gz          2430   0   0     0
pgplot   ftp://ftp.astro.caltech.edu/pub/pgplot/pgplot%s.tar.gz                   522   0   0     0
wcstools ftp://cfa-ftp.harvard.edu/pub/gsc/WCSTools/wcstools-%s.tar.gz          3.1.3   0   0     0
orbacus  carma://OB-%s.tar.gz                                                   4.1.2   3   313   82
erit     carma://erit-%s.tar.gz                                                   x.y   0   0     0
cxxtest  http://telia.dl.sourceforge.net/sourceforge/cxxtest/cxxtest-%s.tgz     2.8.0   0   0     0
cppunit  http://telia.dl.sourceforge.net/sourceforge/cppunit/cppunit-%s.tar.gz  1.8.0   0   0     0
log4cpp  http://telia.dl.sourceforge.net/sourceforge/log4cpp/log4cpp-%s.tar.gz  0.3.4b  0   0     0
```

The build process treats these modules as "alien and hostile", i.e. we probably cannot expect them to be deeply integrated into our build process (and probably don't want to), and most of them will have to be pre-installed before you can fully configure and compile 'carma'. A simple 'build' procedure is to be available for those, and before carma is even built the functionality of these modules will be tested (e.g. if they link and run as expected).
It may however be important that (some of) these modules can be compiled in a mode where tools such as debuggers, profilers and memory leak detectors can be used.

The Carma Software Toolkit" lists a number of components, some of them are needed to compile and link carma code, others are useful tools to have around (e.g. the recently mentioned valgrind tool to find memory leaks in your code, and warn of uninitialized variables)

## 5.1 Package Synchronization

Since all sites must have a reasonably up-to-date `$CARMA_PKG` directory, a script is available that can be used (even on a nighly basis) to keep your local site up to date with the master. See CARMA/conf/carma-package-sync for details.

## 5.2 Package Issues

- some packages don't use our .cc,h extensions (xerces: .cpp .hpp)
- some packages don't use -rpath, so we may have to use LD_LIBRARY_PATH[4] (Examples: orbacus, xerces). Some packages are now starting to come out with the -with-rpath configure flag!
- some packages may also be available via CVS (e.g. miriad), though this is left open to the configuration which one to use. The build system only supports the (stable) tar ball types.
- it could be argued not to depend on locally built packages, as subtle compiler and/or flag differences can lead to badly linked code. Use the one that was built by one of the CARMA builds.

  IDL generated code, what to do about that
- the file CARMA/opt/README exemplifies the steps needed if a new package is added to CARMA_PKG.

## 5.3 Adding a new package

Here's a recipe how to add new things to this tree, we'll take the example of adding the janz drivers (private to us, we can't share them with the world):

- create the tar ball, and put it in `$CARMA_PKG`, say `janz-1.0.tar.gz` (the tar file should ideally create a directory with that version embedded in the name, e.g. `janz-1.0`)
- add a line to `conf/etc/packages.dir` ; since it's "our private" probably looks like

```
janz     carma://janz-%s.tar.gz          1.0     0 0 0
```

[4] See also the discussion on `http://www.visi.com/ barr/ldpath.html`

[don't worry about the last 3 digits for now]
Although by default we always read from `$CARMA_PKG`, if it is a public package, it is useful to reference it in the printf-style format in the 2nd column, and the current version in the 3rd column.

- create a directory `conf/opt/janz`, put an install script in there, and optionally a `Makefile`, but at least make the install script such that it will install from scratch. I have some thoughts how to allow all the install's listen to some configure items, so that using a common odd compiler will be standardized, etc.etc. but for now, you can use the model like i use for cppunit or so.

- test it out by typing

```
install
```

  or

```
make install
```

  or whatever you come up with (some general recipe to be given still)

- add the appropriate line to the carma install script

```
conf/install_all
```

- add some documentation to the Carma Software Toolkit page index.html, now in:

```
doc/index.html
```

# 6 Run time configuration

A directory `$CARMA/etc` is created during the installation process, in which the type of CARMA computer is encoded, and from which the software determines its run-time behavior. ??? Nothing has been specified on this.

# 7 CVS practices

Source code management is done with CVS. Since CVS is not a replacement for communication, a number of important ground rules should be adhered to. *CHECK: cvs. or harkless.???.*

1. The common CVS-based environment variables developers probably will need are:

```
setenv CVSROOT     :ext:$USER@cvs.ovro.caltech.edu:/sw/cvscarma
setenv CVS_RSH     ssh
setenv CVS_EDITOR  micro-emacs
```

2. make sure your system clock is reasonable well up to date. E.g.

```
     ntpdate -u ns1.umd.edu
or
     rdate -s earth.astro.umd.edu
```

   particularly for those who travel with laptops, there is some advantage of keeping your system time in UT, instead of some local timezone.

3. as long as you only commit compilable (and hopefully working) code to CVS, you can safely use it as a backup medium. An even safer way is to use branches, though we are putting some control on the naming convention of our branches, see also the next section (cf. Figure ...):

```
official branch tag names:          Release_1_4
private branch tag name:            teuben_2
tag names:                          Release_1_4_1
                                    Develop_1_3_19
package tag versions:               CARMA_<package>_M_N
```

4. nightly builds will ensure close monitoring of the system (see also tinderbox). They will also determine if a tag name becomes a new Release or Develop version (see below)

5. we will strive for a "stable" release every weekend.

6.

## 7.1 Versioning

The proposed CVS versioning scheme for CARMA is the following:

1. we use a version numbering scheme MAJOR.MINOR.PATCH, e.g. 1.3.18 If the MINOR is even, it's a **Release**, if it's odd, it's the (main line, HEAD) **Development**. The MAJOR signifies major milestones, as exemplified in Figure 1.

2. There are 3 ways to test code: you can (CVS) branch the code, and merge back if the experiment was good. You can play in your own CVS sandbox, use a LocalMakefile, and tell nobody about it (or you could, but because of the LocalMakefile it will not be compiled by the system), or you could use `trial/`.

3. the CVS HEAD (main trunk) will compile and run as much as possible, i.e. possibly unstable experiments should be done in branches, more about that later.

4. a nightly build will determine if the system is stable. "hourly" builds are available through `tinderbox.ovro.caltech.edu/tinderbox`. If the system is stable, the patch number is incremented, and tagged, e.g.:

```
carma_set_version 1.3.19
cvs ci -m "new version"
cvs tag Develop_1_3_19
```

   for incrementing version 1.3.18 to 1.3.19.

5. When a stable "Develop" has reached the milestones[5] for a "Release", it will be branched and tagged as such. For example, suppose 1.3.19 was deemed ok, the MINOR release number (3 in this case) is incremented 1 for a "even" release branch:

```
cvs rtag -b Release_1_4 carma

cvs checkout -r Release_1_4  -d carma_1.4 carma
cd carma_1.4
carma_set_version  1.4.0
cvs ci -m "new Release 1.4"
cvs tag Release_1_4_0

cvs checkout -d carma_1.5
cd carma_1.5
carma_set_version  1.5.0
cvs ci -m "new Develop 1.5"
cvs tag Develop_1_5_0
```

6. When changing the MAJOR release, we will change all CVS revision numbers. For example when we decide that the Develop 1.5.x series will not end in Release 1.6, but in Release 2.0 (with a starting Develop 2.1.0 version), we do, for example:[6]

```
cvs tag Develop_1_5_38
cvs commit -r 2.0                  # reset CVS revision numbers to 2.0
cvs update -A -f -R                # remove sticky tags
cvs tag -b Release_2_0             # create the new branch
```

[5] see Table 1

[6] i guess it was never good to start with version 0.x

```
...
carma_set_version 2.1.0            # back in main line
cvs ci ...
cvs tag Develop_2_1_0
...
```

7. When you make a private branch for some experiment, which you expect to succeed and be merged back into the main line, the following may be a useful procedure:

   ```
   cvs tag -b exp1    # create the branch with this tagname
   cvs update -r exp1    # switch current sandbox to the branched one
   cvs tag my_exp1    # tag the start of the branch (really needed??)
   ```

   At this stage the branch can be commited and updated, as if it was a normal sandbox.
   Finally the merging will be done from the MAIN line, i.e.

   ```
   cvs checkout carma
   cd carma
   cvs update -j exp1
   ```

   Now observe all the merging, and notice if there are any conflicts reported. These need to be cleared before the final commit (of this merge) is done to the mainline.

   ```
   cvs commit
   ```

Some remaining CVS questions:

- It is possible to have a sandbox where certain directories are in different branches. This just seems quite messy to me , but may come in handy and manageable if we keep the modules cleanly separated.
- what if somebody wants to 'prematurely' use a bug fix from mainline in branch?
  seems useful to do, so how do you do this?
- what is somebody wants to 'prematurely' commit a branch bug fix to the mainline probably should not do that in the branch, but mainline
- what if two branches want to share a fix
  probably not, fix it in the mainline, and get it back from there in the branches
- what if a release (dead end) branch has a bug.... should be fixed in mainline, but how does this get back to the Release branch.
  put the fix in a branch in hte release, test it, if ok, merge back in (say) 1.0p1, then update the official release, and the observers can use it again.

- There is also the *"Flying Fish"* model of branching (pp192 in Fogel's book)

two types of fixes: those that only apply to the dead end release branch, and those that apply to both could even happen that is will be fixed to Release, but can't be done to mainbranch yet, since it's not quite stable.....

| Version: | Milestones: | Time |
|---|---|---|
| 0.1<br>0.2 Release<br>0.3<br>0.4 Release<br>0.5<br>0.6 Release<br>0.7<br>0.8 Release<br>0.9 | getting the directory structure right<br>build and test infrastructure in place<br>command line interface, basic library elements | dec 2002 |
| 1.0 Release<br>1.2 Release<br>1.4 Release<br>1.6 Release | Basic array control | July 2003 |
| 2.0 Release | SZA | |
| 3.0 Release | BIMA | |
| 4.0 Release | CFL | |

Table 1: Example release timeline for CARMA with rough milestones (details TBD)

# 8 Hardware Emulators

Since much code is developed off-site, we must be able to emulate certain pieces of the hardware in order to excersize as much of the software as possible. Little is determined how this is going to happen at this time. The weather system can be used to easily implement a functional emulator. We need a list of the hardware components that are needed, and if they follow this simple model (simple client writing in shared memory).

# 9 Bug Reporting

A `bugzilla` repository has been setup to handle bug reporting. All developers will have an account, and can setup setup modules. See `http://www.mmarray.org/bugzilla`.

### 9.1 Some notes on 'Products', 'Components', and bugs

'Products' are bugzilla's top level software structure. 'Products' may be comprised of one or more 'Components' but not other 'Products'. 'Components' do not have subcomponents; its a two-level hierarchy. In this sense, MS Office would be considered a 'Product' and Word and Excel would be considered 'Components' of MS Office.
Regular bugzilla users, such as yourselves, can be granted permission to edit 'Products' and 'Components'. This is how bugzilla is currently configured. I have not figured out a way to grant editing permissions for 'Products' only or 'Components' only.
You may create 'Products' and 'Components' as you see fit. Products are owned by whomever creates the Product. You may designate any valid bugzilla user as the owner of a 'Component'. (New 'Products' and 'Components' should show up immediately if your browser cache is kept empty.) Valid bugzilla users are identified by the email address they use to log in. I have not figured out a way to restrict ownership to whomever creates the 'Component'.
You may also create 'Components' for a 'Product' that someone else created. I have not figured out a way to restrict 'Component' creation and editing permissions to whomever created the associated 'Product'.
You may issue bugs against any 'Product' or 'Component'. Once created, only the assignor and assignee of a bug may edit that bug. Only the assignor can remove a bug.
*** 'Products' and 'Components' for which no bugs have been issued may be deleted by ANYONE. You may want to issue a dummy bug against newly created 'Products' or 'Components' to more permanently establish their existance. ***
*** Nobody can delete a 'Product' or 'Component' for which bugs remain unresolved. ***
One alternative to the current configuration is to deny regular users the permission to edit 'Products' and 'Components' altogether. In that case, users with sufficient bugzilla administrative permissions would be responsible for the 'Products'/'Components' structure.

## 10 Managing multiple platforms

We do not plan to add support to maintain multiple platforms within the same source tree. The CVS philosphy is that one simply checks out a new tree for this. This does however imply that given that multiple sandboxes will be present, the CARMA environment (this includes various configuration files and daemons that may be running) must be cleaned up gently to a zero state, and a new one started up.

# 11 Programming

## 11.1 Programming Style Guidelines

We encourage programmers to follow some general programming guidelines to give our code a consistent look and feel, and make it easy for anybody to maintain and upgrade the system. The full document of these guidelines is available in HTML form on
`http://www.mmarray.org/workinggroups/computing/cppstyle.html`,
and covers C++ style guidelines, naming conventions for files, classes, variables etc. From here on we will refer to this as the *Style Guide*. The code review process (see below) is also supposed to help enforcing a reasonable degree of uniformity of the code. The specifics how documentation is added to the code is however covered below.
Source code files in CARMA will use the following extensions:

```
.cc             C++ code   (inlines are in .h files)
.c              legacy C code
.h              C or C++ headers  (also: extern "C" if needed)

.idl            Interface Definition Language
.java           Java code


.f              legacy fortran code (could also have .for)
.inc            fortran include file (could also have .h)
```

## 11.2 Documention: DOXYGEN

Code will be documented using the `doxygen` inline tag system [3], which will generate the online documentation that will be the main documentation center for programmers. Our file headers will thus be somewhat structured and contain the following items:

- CVS key tags: (`$key$`)

**$Id$** this is standard, and catches in one line the last CVS commit time, revision, tagname, last person edited, etc.

**$CarmaCopyright$** we actually still have to define this :-)

- doxygen tags: We will use the @ symbol (`@tagname`, instead of `\tagname`), and especially note the following tags:

`@author` Full name of the person responsible for the code

`@reviewer` The code reviewer, should eventually be there for all code

`@inspector` Code inspector, only for critical code

`@todo` a list of things needed for final cleanup or future enhancements. Can also appear in the .cc files (perhaps even more likely).

- each class definition must have an associated documentation block summarizing the class as a whole block.
- each public and protected function must have a documentation block. Each block must use a `@param` tag for each input parameter [even if it seems obvious].
- each documented function must use an `@exception` (or `@throw`) tag for any exception that:
  - is explicitly thrown by the function for any error that is possible during typical operation of the system. This excludes programmer errors. It usually includes user errors, function input errors, and system failures that could happen from time to time.
  - is documented with an `@exception` tag for a another function called by this function and is subsequently uncaught. [The rationale here is that the `@exception` tag advises the user of problems they need to be prepared for.]

    [April 30] See also Amar's exception proposal!
- use the brief mechanism, either explicitly with `@brief` or implicitly.
- skip the use `@return` only for accessor functions in which the return value is obvious from the function definition. e.g.

```
/**
 * get the i-th antenna
 * @param i  the index of the desired antenna
 */
Antenna& getAntenna(unsigned int i);
```

## 11.3 Unit Testing

A unit test should be written for each class or legacy API. `CppUnit` has been choosen as the framework for testing our classes. For our legacy code a simple `main()` can be defined, either in a separate source (preferred) or inline using `#ifdef TESTBED` (e.g. NEMO and MIRIAD). Such testing should occur inside subdirectories named `Test` below the implementation modules. For us the purpose of a unit test is three fold:

1. ensure the class compiles and links (ok, trivial)
2. be run through `valgrind` to check if the code is leak free
3. be run through `gcov` to check if most code has been excersized

### 11.4 Code Coverage

Although good commercial tools (e.g. `purify` and `TestCenter`) are available, we use the GNU compiler which comes with `gcov`. This can be used quite effectively to evaluate code coverage. Specifically, in `gcov` the code is compiled with special flags `-fprofile-arcs -ftest-coverage` (the Makefile's `coverage` target will take care of this), after which `gcov` can be run on any .cc file to extract useful statistics. We consider the minimum fraction of all source code lines executed (the default in gcov) to be 75%.

```
info gcc gcov                  <-- good background information on gcov

make clean coverage            <== detailed syntax TBD
utLogger
gcov Logger.cc
gcov utLogger.cc
```

In addition, leak detectors such as `valgrind` should be used to confirm the unit test does not leak as the class is exercised. For high performance needs you can optionally use `cachegrind` to evaluate your cache misses, or the standard Unix profiling techniques (gprof,...)

```
make tests TESTS=utLogger       <== detailed syntax TBD
valgrind utLogger
```

## 12 Peer Code Review

It is a common industry practice that code is reviewed and inspected, and we believe this will also be very beneficial for CARMA code. All code will be reviewed, though only critical code will also be inspected.

After a programmer has finished the code (has been committed to CVS, compiles in the nightly compile suite, and has an associated functional unit test), the code is submitted for "review" to the Review Manager. For this the programmer has also assembled the output of `valgrind` and `gcov` on the unit test executable(s) into a file "`Review/`*ClassName*`.review1`" [7].

The Review Manager then assigns one or two reviewers to the code, giving the reviewer(s) a deadline to complete the review based on the LOC and the needs of dependent packages. The review results are passed back to the developer via CVS in the `Review/` file(s). The developer will make the required changes and resubmit for review (by the same reviewer) if need be. For critical code a code inspection will be added to the review, which requires the reviewer to have a more detailed look at the code itself. It is up to an agreement between developer and reviewer to allow the reviewer to add comment into the code via CVS, instead of using the `Review/` file(s).

The developer and reviewer should work out most code issues amongst themselves and then report to the Review Manager. If a code review results in a disagreement between developer and reviewer, another reviewer is added to break the tie. If that results in violent disagreement, the Review Manager will have to intervene. The Review Manager is a rotating job, currently executed by

[7] e.g. `$CARMA/carma/services/Review/Logger.review1`

Steve Scott. Progress of the review process is currently done by Colby Craybill, and will be available on the web[8].
It is up to the discretion of the author of the code to request a re-review when sufficient code has been added or changed to request a new review.

[8] `http://www.mmarray.org/workinggroups/computing/codeReviewStatus.html`

## 12.1 Code submission checklist

In order for code to be submitted for review, the developer should prepare a few practical things (also check out the review and optional inspection checklist below of course):

1. pass the NIGHTLY compile–and–run suite ("smoketest") on ideally more than one compiler/architecture (note: for Review Manager?).

2. The outcome of running `valgrind` and `gcov` on either the test program(s) or unit test should be placed in a file `Review/`*ClassName*`.review1`.

   Also explicitly state which files belong to the review, normally just the two files *ClassName*.{*h*,*cc*}.

## 12.2 Code Review and Inspection checklist

Here is a summary of the steps the reviewer should consider in his report. Note that there are a few items at the end in case the Review includes a Code Inspection stage.

1. Annotate your comments in a file `Review/`*ClassName*`.review`*N*, for each review *N=1,2,3.....* The developer should have left notes on the outcome of `valgrind` and `gcov` in the *N=1* file.

2. Does code adhere to the previously accepted design document? You should be able to do this by looking at the doxygen generated documentation, instead of looking at the individual .h or .cc file (with the right configuration, Doxygen will not show a link if a class is not documented).

3. Check the doxygen generated documentation (see also previous section):

   (a) each class definition has an associated documentation block summarizing the class as a whole block.

   (b) each public and protected function must have a documentation block. Each block must use a `@param` tag for each input parameter [even if it seems obvious].

4. Does the code follow our *Style Guide*? Specifically the ones where we use the word **must** occur in 27 out of 104 cases:

   (a) Naming Conventions (Section 3: cases 1,2,3,5,8,12,15/25,28)

   (b) Files (Section 4: cases 4,7,8,9,11,12)

   (c) Statements (Section 5: cases 2,3,6,8,14,22,23,25,26,27)

   (d) Layout and Comments (Section 6: cases 23,24,26)

   To paraphrase, the reviewer **should** and **can** also look at the remaining guidelines. Both the .h and .cc file(s) need to be examined for this.

5. includes a working unit test such that the class (or function set) can be exercised and passes the unit test as well as be reasonably leak free and have acceptable code coverage. Check the comments in the "review" file the developer left.

*Inspection Only* check for correctness of the code:

   (a) w.r.t. design (what it does)

   (b) w.r.t. design (how it does it)

   (c) reliable/robust (if specified by the design)

   (d) algorithm:

      i. if applicable, is the relevant Design Pattern [8] identified?

      ii. math correct?

iii. numerical correctness?

iv. choice of algorithm correct?

v. proper code reuse, are the CARMA utils used where appropriate, is STL used ?

6. if you give thumbs up, add your name to `@reviewer`, and `@inspector`, if applicable.

It is expected code review to take around 1-2hr/100LOC. The command

```
enscript -Ec -C -r2 code.cc -o junk.ps
```

will pretty-print source code, add line numbers for eased communication, and make lines over 80 characters wide (case #XX) more obvious to the reviewer (or use -r1 and/or smaller font is you need a wider page)

```
enscript -Ec -C -r1 -fCourier7 code.cc -o junk.ps
```

The reviewer should not have to compile or run any code, though is free (encouraged?) to try and excersize the compiler and Carma system with this code.

# Glossary

Here are some CARMA specific entries[9]

**CARMA computer type** CARMA computer type, one of ACC, AC, CC, DBC, LLC, LRC, EC, ARC or UIC! got that?

**CARMA package** independant piece of software that we often place in `$CARMA/opt` or somewhere else on the system. Examples are pgplot, Orbacus, log4cpp, ... We keep a repository of their tar balls for re-installations in a directory `$CARMA_PKG`.

**CVS module** used by CVS to designate 'root tree directories' in the CVS repositories. Examples are carma, miriad, nemo

[9] see also `http://www.alma.nrao.edu/development/computing/docs/joint/draft/Glossary.htm`

## Installation Overview

Details on the installation of the CARMA software can be found elsewhere in this document, this page as a cheat-sheet how to download and install CARMA from scratch:

```
# 1a) get the install script via CVS

setenv CVSROOT :ext:$USER@harkless.ovro.caltech.edu:/sw/cvscarma
cvs co -d tmp carma/conf

#     or via a URL

wget http://www.astro.umd.edu/~teuben/carma/install_all
chmod +x install_all


# 2) if you don't have the carma_pkg, get it 'manually'

setenv CARMA_PKG /tmp/carma_pkg
tmp/carma-package-sync master=$USER@harkless.ovro.caltech.edu:/sw/carma_pkg

# 3) install the tools and carma seperately
#  ahum, this still asssumes you have a carma_pkg 'somewhere'

tmp/install_all carma=`pwd`/carma carma_tools=`pwd`/carma_tools
```

# References

[1] *CARMA C++ Programming Style Guidelines* - Pound. Amarnath, Teuben
[2] `http://www.stack.nl/~dimitry/doxygen/commands.html`
[3] *GNU autoconf, automake, and libtool* - Vaughan, Elliston, Tromey and Taylor (New Riders, 2001) - Chapter 14 and 15 (Writing Portable C and C++)
[4] *ICD/Doxygen* - Amarnath.
[5] *Software Engineering* - Ian Sommerville, 6th ed. (2001, Addison-Wesley)
[6] *C++ Gotchas* - Stephen Dewhurst (2003, Addison-Wesley)
[7] Design Patterns - "gang of four".
[8] The AIPS++ Quality Assurance Group (`http://aips2.nrao.edu/docs/html/qag.html`)
[9] *Open Source Development with CVS* - Karl Fogel. (Coriolis press, 1999)
[10] *GNU autoconf, automake, and libtool* - Vaughan, Elliston, Tromey and Taylor (New Riders, 2001)
[11] *GNU make* - Stallman and McGrath (FSF, 1998)
[12] *LOFAR build document* - Ger van Diepen , Klaas Jan Wierenga.

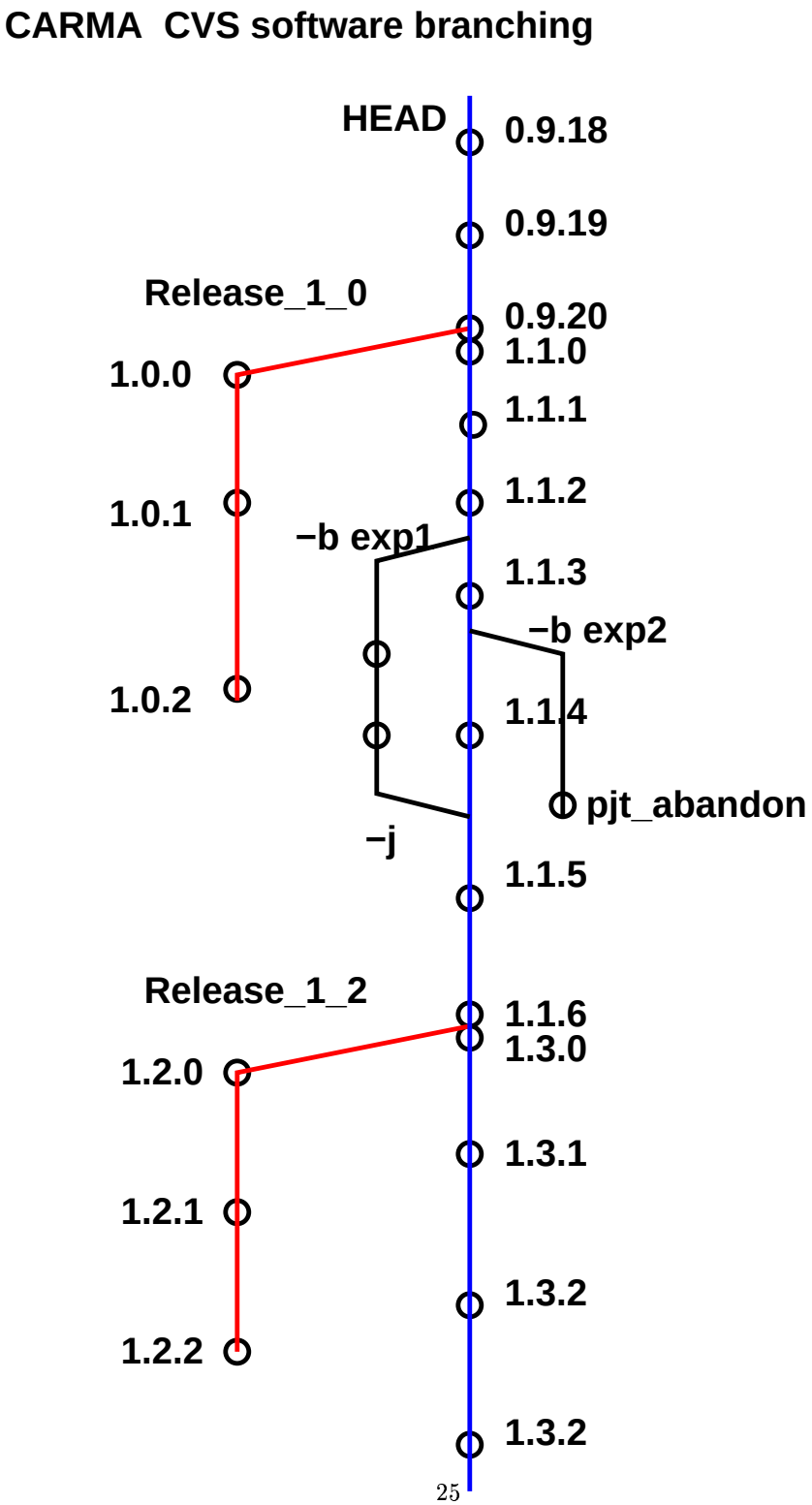




Figure 1: Branching Model: Mainline (HEAD) in blue, dead-end Release branches in red, and experimental branches (dead-end of merged) in black