# UNO: The software working-group

Coordinators: Clark McGrew[*] Chiaki Yanagisawa[†]

February 12, 2004

A web version can be found at `http://nngroup.physics.sunysb.edu/~pdk/`

### 0.1 The Repository

The UNO Software Repository is read-only accessible on the web. Collaborators who wish to work with the UNO Software Working Group can get write-access by contacting Clark McGrew[1]. If you base work on this software, please include a citation to this web page. In the future, a more formal citation will be provided.

### 0.2 Documentation

The UNO Class libraries[2] are documented using doxygen. This document (web-page and hardcopy) is in the software_web module and is written using latex2html.

### 0.3 NEWS

**Nov 16, 2001** Convert to CMT as the primary build system. This will allow the same build files to be used on windows and unix.

## 1 UNO Coding Guidelines

This is an attempt to provide a little bit of unification within the UNO software[3]

[*]mailto:clark.mcgrew@sunysb.edu
[†]mailto:chiaki.yanagisawa@sunysb.edu
[1]mailto:clark.mcgrew@sunysb.edu
[2]classlist.html
[3]UNO stands for Underground Nucleon Decay and Neutrino Observatory. The UNO code is written relatively generically so that SK and K2K members can borrow events from SK and K2K to test any code they write for UNO, and (gasp) use UNO code for SK.

## 1.1 Memory management convention

The basic convention for any code based on unoCore (ie, not CSIM) is:

1. Passing/returning a non-const pointer is transfering ownership.
2. Passing/returning a reference or const pointer is not.

For more details see this page[4].

## 1.2 Coding style

The suggested C++ coding style is based on the one suggested by the ROOT developers and is mostly concerned with naming conventions. Please try to follow these conventions since it really does help make the code readable.

In general, please follow the programming guidelines given in "Effective C++" and "More Effective C++". These books are a nice concise style guide for C++.

**Namespaces** Name spaces are a good idea, but were not well supported when the UNO packages were started. In the future[5], we will choose a standard namespace for all UNO code. The most likely candidate is `UNO` which is already used in `HEPUnits.hxx` and `HEPConstants.hxx`.

**Type and Class Names** A type name should begin with a capital T (for instance "TObject").

**Class Fields** One of the "problems" with C++ is that inside of a class definition there is no easy way to distinguish local variables from class fields. The situation is made much easier if every class field is prefixed with an "f". Fields should either be private (best), or protected.

**Class Methods** The class methods should all start with a capital letter. Also, since none of the class fields are visible outside of the class, you need to provide a routines to get and set their values (named Get*Variable* and Set*Variable*).

**Class Destructors** Since you never know when a class might be used as a base class, always use a virtual distructor. In the (rare) case where a non-virtual destructor is required, clearly document the reasons.

**Function Names** A function name should begin with a lower case and follow the normal C conventions. That means that a long function name might be `a_long_function_name()` with all lower case characters and underscores between words.

[4] unoCoreClasses/index.html#memory

[5] Possible by the time you are reading this.

```
////////////////////////////////////////////////////////////////////
// A standard class.
////////////////////////////////////////////////////////////////////
class TExample {
private:
    int fOneField;
public
    TExample();
    virtual ~TExample();

    void SetOneField(int);
    int GetOneField(void);
    double AnExampleMethod(void);
};

void TExample::SetOneField(int theValue) {
    int anInternalVariable = theValue;
    fOneField = anInternalVariable;
}
```

Figure 1: An partial example of a simple class written using the UNO coding policy.

**Exception Names** An exception is a special type of object so it has a special name. Follow all of the rules listed above, but use "E" instead (For instance, "EAnException"). The ultimate base class of all exceptions should be `std::exception`, but they will typically be more directly derived from an exception class that is part of the project libraries.

**Global variables** Don't use global variables! If you think you need a global variable, write a singleton class instead. If you *absolutely* must use a global variable, reconsider. You *really can* use a singleton class instead. If you still need that global variable, prefix it with a "g" and document it **really really well**. Having created a global variable, consider finding somebody else to program for you.

## 1.3 Library Versions and Releases

CMT provides us with a convenient way to handle library versions and releases, but it doesn't impose any particular version numbering scheme.

An UNO package version has a major, minor and a micro version number (e.g. "v4r3p2" which happens to be the version number of Geant 4 when I wrote this.)

- The major version is used to indicate major changes in the visible interface. For instance, if a new release of a library has a different interface and is

```
//////////////////////////////////////////////////////////////////
// A singleton class.
//////////////////////////////////////////////////////////////////
class TSingleton {
private:
    static TSingleton *fTSingleton;
protected;
    fTSingleton();   // The constructor is protected (or private).
public:
    static TSingleton *GetTSingleton(void);
    virtual ~TSingleton();
}

TSingleton::fTSingleton = NULL;

static TSingleton *TSingleton::GetTSingleton(void) {
    if (!fTSingleton) fTSingleton = new TSingleton;
    return fTSingleton;
}
```

Figure 2: An partial example of a singleton class written using the UNO coding policy.

expected to break existing programs then it should have a new major version number. The major version number should not be incremented for bug fixes even if those bug fixes will break some programs, because it is a bug for a program to depend on a bug.

- The minor version number (or release number) should be incremented whenever a new feature (or set of new features) is added to a library, but the new version is backward compatible.

- The micro version number (or patch number) should be incremented whenever a bug is fixed in a "production version." Development versions should not have a patch number until they are being stabilized to become the production version. The patch number should not be incremented as part of normal development.

#### 1.3.1 Turning a development release into a production release

Once a particular developement release has stabilized and is ready to become the production version, the final development version should be release and tested. If no new bugs are found during testing, then the final development version should be tagged with a stable release number as well as a new development version number.

For example

1. The unoCore CVS head version (assume the current development version is tagged v0r3) is almost stable and we decide to make a production version.

2. The current head is branch tagged v0r3p0 and becomes a "release candidate". The release candidate should be tested and any minor bugs fixed. Assuming that no major bugs are found, this becomes the release. If a bug is found after the package is released, the "release candidate" cycle will need to be repeated. See below.

3. The current head is tagged with the next development version number "v0r4". Development can continue on the v0r4 (the CVS HEAD) as the release candidate is stabilizing.

4. If a small, but critical, bug is found in a production version and it must be fixed, a new patched production version must be created. In our example, the production version number would become v0r3p1. However, a production version should only be patched if a bug is really critical, it cannot be worked around. If the bug is not trivial, then the "production version" should probably be considered "buggy" and dumped in favor of a later release candidate. To create a patched production version, checkout the production branch, create a new branch tag (e.g. v0r3p1), fix the bug.

### 1.3.2 CMT and CVS gymnastics to create a new package version

Creating a new package version is relatively simple, but not very intuitive. A new CMT package version is created each time a CVS tag is added to the cmt/requirements file and you add a CVS tag with a command like "`cvs tag v0r4'`". Assuming we want to change the development version from v0r3 to v0r4 and create a release candidate v0r3p0, we would do the following in the top package directories:

To create a release candidate version, go to "`<packages>/unoCore/v0r3`":

1. Make sure the code is up to dated with "`cvs update`".

2. Add the new branch tag to the files with "`cvs tag -b v0r4p0`". Notice the option "`-b`".

Then, from "`<packages>`":

1. Get the new release candidate with "`cmt checkout -r v0r3p0 unoCore`" which creates "`<packages>/unoCore/v0r3p0`".

2. Edit "`<packages>/unoCore/v0r3p0/cmt/requirements`" to change the package version line and commit the new file.

To create a new development version, go to "`<packages>/unoCore/v0r3`":

1. Make sure the code is up to dated with "`cvs update`".

2. Add the new tag to the files with "`cvs tag v0r4`".

Then, get the new development version from "`<packages>`":

1. Get the new developement version with "`cmt checkout unoCore`" which creates "`<packages>/unoCore/v0r4`".
2. Edit "`<packages>/unoCore/v0r4/cmt/requirements`" to change the package version line and commit the new file.

Notice that the tag denotes the first files that were part of a new version, not the most current (and therefore most relevant files).

# 2 Software Tools

The UNO software is leveraging several popular and well used HEP packages to allow quick development of a complete tool set.

## 2.1 CVS — Source file version control

CVS[6] the "industry standard" file version control system that has become quite popular among physics projects. It is tightly integrated with CMT and is used to track the changes to each piece of software.

## 2.2 CMT — Configuration Management Tool

CMT (Configuration Management Tool)[7] is supported by LAL[8]. To quote the manual: "This configuration management environment, based on some management conventions and comprising several shell-based utilities, is an attempt to formalize software production and especially configuration management around a package-oriented principle.

The environment provides conventions (for naming packages, files, directories and for addressing them) and tools for automating as much as possible the implementation of these conventions. It permits to describe the configuration requirements and automatically deduce from the description the effective set of configuration parameters needed to operate the packages (typically for building them or using them)."

## 2.3 ROOT — A Physics Analysis Tool

ROOT[9] provides the basic I/O and analysis system for UNO. This is the successor to PAW being developed by Rene Brun et alånd is used by several current experiments (eg. MINOS, STAR, NA49, H1, HERA-B, INTEGRAL, to name a few.).

[6] http://www.cvshome.org/
[7] http://www.lal.in2p3.fr/SI/CMT/CMT.htm
[8] http://www.lal.in2p3.fr
[9] http://root.cern.ch

```
#!/bin/sh
. /u/ale/HEP/CMT/v1r9p20011011/mgr/setup.sh
```

```
#!/bin/csh
source /u/ale/HEP/CMT/v1r9p20011011/mgr/setup.csh
```

Figure 3: The setup script for CMT on ale.

## 2.4 GEANT4 — The new version of GEANT.

The GEANT4[10] collaboration has set about re-writing the GEANT structure using modern programing techniques (and the C++ language). This project is in a relatively early stage, but can be used for initial detector studies. Using this package guarrantees that we will gain the expertise of a well respected collaboration and the input from many large experimental groups from around the world to help us develop a good detector simulation.

## 2.5 N Step Program to working software

The UNO software is very much a work in progress and this section is not intended to provide cookbook instructions to install it. However, there should be enough hints and clues to allow you to get it to work.

UNO is depending on several software packages that are produced outside of the collaboration and which must be correctly installed before you can develop, or even compile, new code. As a result, you will need to install these packages and get them configured before you begin to worry about the UNO code. This section discribes how I have installed the code on my portable computer and the machines at Stony Brook. I have found it very helpful to create a new account into which all of the new software will be installed (I have it in "`/u/ale/HEP`" on ale and "`/home/physics`" on my portable).

### 2.5.1 Install CMT

The first step is to install CMT[11]. Connect to the CMT[12] site and download the lastest version (>v1r10) and install it into "`/home/physics/CMT`". There are good instructions on the web page.

After installing CMT, you should add setup scripts "`/home/physics/bin/CMT-setup.sh`" and "`CMT-setup.csh`"
to "`/home/physics/bin/physics-setup.sh`" and "`physics-setup.csh`".

[10] http://wwwinfo.cern.ch/asd/geant4/geant4.html
[11] http://www.lal.in2p3.fr/SI/CMT/CMT.htm
[12] http://www.lal.in2p3.fr/SI/CMT/CMT.htm

### 2.5.2 Install CERNLIB

You should install CERNLIB 2000 or later. I'd suggest that you install it into "`/home/physics/CERNLIB/v2000/...`". Then create scripts in "`/home/physics/bin`" to initialize the CERN, CERN_LEVEL, and CERN_ROOT environment variables and add it to the "`physics-setup`" scripts.

### 2.5.3 Install CLHEP

Download the CLHEP[13] library. You should use version 1.7.0.0. The library should be installed into "`/home/physics/CLHEP/1.7.0.0/...`" by setting the –prefix when you configure and build the library. After you have installed the library you should make a link from the library to libCLHEP.a (e.g. The library will be installed as libCLHEP-g++-1.7.0.0.a and you should make a symbolic link to libCLHEP.a). I suggest that you then copy the source into "`/home/physics/CLHEP/1.7.0.0/source/CLHEP/...`".

Having installed the CLHEP library, you should add a setup script to "`/home/physics/bin`" (see section 2.5.2).

**Creating a shared library** The CLHEP library can be used as a shared library (and probably should be on linux). To create a shared library on linux execute the following commands in "`/home/physics/CLHEP/1.7.0.0/lib`"

```
mkdir shlib
cd shlib
ar x ../libCLHEP.a
gcc -shared -o libCLHEP.so *.o
mv libCLHEP.so ..
```

### 2.5.4 Install Geant 4

I suggest that you install the latest version of Geant 4 (4.3.2 as I write). The installation is not completely trivial, but it's not too bad. The only trick is that you *must* install the global libraries (as opposed to the GRANULAR libraries), and you should make shared, not static, libraries on most unix systems. You also configure to use the "terminal" and "tcsh" GUI. I suggest that you install geant 4 into "`/home/physics/geant4/geant4.3.2/...`".

This section needs to be significantly expanded to cover stuff like GUI and visualization. For the moment, I'll just say that I installed DAWN, OpenGL and Raytracing for visualization, and the Xaw GUI.

### 2.5.5 Install ROOT

The ROOT setup is pretty straight forward and there are pretty good installation instructions there. I suggest that you install the latest root release and install it into "`/home/physics/root/3.??.??`"

[13] http://wwwinfo.cern.ch/asd/lhc++/clhep/index.html

```
#!/bin/sh

# Set the location of CERN
if [ "${CERN}x" = "x" ] ; then
    export CERN=/home/physics/cernlib
fi

if [ "${CERN_LEVEL}x" = "x" ] ; then
    export CERN_LEVEL=v2000
fi

export CERN_ROOT=${CERN}/${CERN_LEVEL}
export CERNLIB=${CERN}/${CERN_LEVEL}

PATH=${CERN_ROOT}/bin:${PATH}
```

```
#!/bin/csh

if ( ${?CERN} == 0 ) then
    setenv CERN /u/ale/HEP/cernlib
endif

if ( ${?CERN_LEVEL} == 0 ) then
    setenv CERN_LEVEL v2000
endif

setenv CERN_ROOT ${CERN}/${CERN_LEVEL}
setenv CERNLIB   ${CERN}/${CERN_LEVEL}

set path = ( ${CERN_ROOT}/bin $path )
```

Figure 4: The setup script for CERNLIB on ale.

# 3 Frequently Asked Questions

## 3.1 Questions about CMT

### 3.1.1 How do I check out the cvs head version of everything?

1. Check out the uno package (`cmt checkout uno`)
2. Check out the packages required by the uno package (`cmt checkout -requirements uno/<version>/cmt/requirements)`) where `<version>` is the version of uno that was checkout out in the first step.
3. Change directory to `uno/<version>/cmt`.
4. Then make sure you have the head version with a broadcast command (`cmt broadcast ''(cd ..; cvs update -A -d)''`)

### 3.1.2 How do I compile a library?

The individual libraries can be compiled by going to the "cmt" subdirectory and running make. For example, here is how to build unoCore:

1. Go to the unoCore cmt directory. (`cd unoCore/<version>/cmt`).
2. Run make (`make`).

### 3.1.3 Is there a "make world" command?

All of the libraries can be build from the uno package using the cmt broadcast command.

1. Go to the uno cmt directory. (`cd uno/<version>/cmt`).
2. Broadcast make (`cmt broadcast make`).

### 3.1.4 What is the copyright on the library?

The copyrights for the routines are held by the original authors or their institutions (depending on the institution rules), however, since parts of the core library derive from ROOT, which in turn derives from GPL code the library it's best to distribute as source code.

### 3.1.5 How do I create a new version of a package?

Creating a new version of a package is covered in section 1.3 of this document. This section includes the CMT and CVS commands that need to be executed.

### 3.1.6 How do I know what packages are available?

If people are diligent they will be listed in `uno/cmt/<version>/requirements`.

#### 3.1.7 How do I get a package?

Go to the top level directory type `cmt checkout <package>`.

## 3.2 Questions about CVS

### 3.2.1 How do I show I'm the author of a commit?

This is broken in the CVS server used.

To have an attribution show up in the email message sent to the commit list, define the `CVS_SERVER` variable by replacing "username" with your user name like:

```
export CVS_SERVER="env LOGNAME=username cvs"  # for bash
setenv CVS_SERVER "env LOGNAME=username cvs"  # for tcsh
```

Otherwise, just stick in an attribution to the commit message by hand.