# Geant4 Development Pocket Guide Draft

Ben Morgan Friday 7th September, 2012

## Contents

# PART 1

## Introduction

This guide aims to briefly document the use of SVN and CMake for development of the Geant4 toolkit itself. It currently acts as a short expansion on topics covered at the Geant4 2012 Workshop at Chartres. Within this context, **it is not a complete or official document**, though there is an intention for this, and other material presented at the Workshop, to form the basis of a future manual. This will be heavily guided by feedback from developers

Due to the draft nature of this document, areas for comment and possible expansion are marked up as follows

> *Topics for review and or expansion, input and discussion at the 2012 Workshop*

> ***Expected major interface or behaviour changes between Geant4 9.6 and 2013 release of version 'X'. Documentation is written from the perspective of the 9.6 interface and behaviour.***

These marked up notes should not be regarded as complete by any means! There are no doubt many areas for expansion and improvement.

The main document has been marked up using XeLaTeX as the author had a suitable template to hand to aid authoring. However, there is nothing to stop the content being migrated to DocBook or other markups. A "pocket guide" format is adopted to provide a clean, compact and easy to browse document. It should be readable as a printed document, online and even on mobile devices. It is hoped that this would be carried through to any future document whatever the markup language used!

# PART 2

# Working with SVN

## Writing Good Commit Messages

The SVN commit log is the primary method for recording what you have changed, and also *why* the changes have been made. Geant4 also uses `History` text files to record changes and tags, usually one per module.

Commit logs are slightly more powerful than `History` files as they are unambiguously associated with particular changesets. One might argue that the changeset diff is enough, yet we have all experienced cases where we go back through *our own code* and wonder why we made a particular change. Good commit messages are therefore critical for recording the what and the why of a change not only for your collaborators but also for you!

What, then, is a *good* commit message? As you can imagine, there are many schools of thought here, though perhaps the clearest and easiest to grasp is the following pattern (adapted from a blogpost by Tim Pope):

```
Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary. Wrap it to about 72
characters so that log formatting is nice even when indented.
The blank line separating the summary from the body is critical
(unless the body is omitted); the summary might be used as an email
subject or as a shortlog in certain repository viewers.

Write your commit message in the present tense: "Fix bug" rather than
"Fixed bug". If you think about it, the message goes with the change,
not after it! This isn't hard and fast as long as you're consistent.

The body can contain multiple paragraphs, separated by blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, preceded by a
  single space, with blank lines in between, but conventions vary here

- Use a hanging indent
```

This is not to say you should make every commit message into a novel as the context will matter. For example, if you have simply

added some Doxygen markup to one class, a perfectly good commit message would be:

```
Add Doxygen markup to G4Foo class and methods
```

In contrast if you are committing a changeset that fixes Bug 42, then a *bad* commit message would be

```
Fix Bug 42
```

which conveys little useful information. A better message would be

```
Fix Bug 42(G4DeepThought returns wrong answer)

Problem arises in AnswerToUltimateQuestion() method when the
G4Question instance cannot be initialized.

Implement new G4ProduceUltimateQuestion() function using the Earth
algorithm to calculate and return an appropriately initialized
G4Question instance. This method may throw a VogonException if
it cannot complete.
```

Whilst the preceeding considerations mean some judgement is needed in how much to write, one hard and fast rule is that you should ***never, ever, make a commit without a message*** [1] . This effectively defeats the aim of having a version control system in the first place! Moreover, your collaborators will not thank you when you introduce a bug and they have zero information on what change may have introduced the bug or whether the change can be reverted!

### Further Reading

The links below contain further discussions and information on using Subversion and general practices with version control systems.

- Version Control With Subversion
- "A Note About Git Commit Messages" by Tim Pope
- "On commit messages" by Peter Hutterer

[1] One rule that teams sometimes use to enforce this is that the author of an empty message has to buy all the other team members a beer!

# PART 3

# Working with CMake

## Getting Help on CMake

Documentation for CMake is now *reasonably* comprehensive, at least for the core functions, variables and modules. The first port of call is therefore Kitware's CMake website, and especially the

- Main CMake Documentation Page.
- CMake Documentation Wiki.
- CMake FAQs.

It's also possible to get help via the `cmake` command line application. For example

```
$ cmake --help-command project
```

to output help on the `project` command to stdout. Use

```
$ cmake --help-commands | less
```

to view documentation on all commands in the `less` pager (or pipe to your favourite text viewer).

> *Expand to cover additional CMake features and usage? Could cover cmake-gui and ccmake?*

A further useful resource is the CMake Mailing List, from which many useful tips and tricks can be found if you use a list viewer with integrated search. The author recommends The Mail Archive which provides a nice search interface and view by date/thread options.

## Supported Platforms and Versions

As of version 9.6, Geant4's buildsystem should support CMake 2.6.4 and higher on Linux, Mac OS X (Lion and Mountain Lion) and Windows 7.

> *This needs review, especially on Mac due to changes in Xcode organization which only newer versions of CMake*

*support. The requirement for 2.6.4 support is kept because we recommend system installs of CMake in the first instance. This is mostly for cluster based Linux installs which may be running older versions of CMake. However, some parts of Geant4's buildsystem, mostly in Tests, will only work with CMake 2.8 and higher.*

Be aware that CMake's API can change even between minor versions. If you are adding CMake level functionality, check the Backward Compatibility Matrix on the CMake Wiki.

### Geant4's CMake System

Geant4 builds functions and macros on top on the core CMake commands to provide a coherent and (we hope) easy to use "API" for integrating your code, data and associated tests plus example applications into the Geant4 toolkit.

*How should this be documented? Later sections can describe each aspect of the API from a usage point of view, but should there also be a full command reference?*

Don't hesitate to contact the Software Management WG for further help, advice or comments/criticisms. Our doors (physical or virtual!) are always open, and it's always better to ask than sit around in confusion.

Please also use Geant4's Bugzilla to report problems, and there is no need to contact us before doing so. It better to report something that turns out not to be a bug than to let something slip through, and in both cases having the record in the system is useful for future reference.

# PART 4

# Developing and Maintaining Geant4 Toolkit Modules

### Layout of Source Code

A "module" is the basic development unit in Geant4 and provides a coherent group of C++ code to provide a specific service in the toolkit. When integrated into the toolkit, the module may be built as as a single library, or it may be merged into a higher level library as a component.

Geant4 enforces a standard layout for each module in the source tree:

```
+- G4mymodule/
   +- sources.cmake
   +- CMakeLists.txt
   +- include/
   |  +- G4Foo.hh
   |  +- G4Bar.hh
   +- src/
      +- G4Foo.cc
      +- G4Bar.hh
```

### The `sources.cmake` File

This file is used to define the following aspects of your module:

- Include paths to API headers used.
- Sources and Headers of the module itself.
- Libraries to link to the module.

It is the responsibility of the module maintainer(s) to keep this file up to date with these values. We'll walk through an example of this for the `G4mymodule` module described earlier. We'll assume this uses two internal Geant4 modules, global and materials, and also uses an external API "UsedAPI".

***The description that follows is for Geant4 9.6, it is expected that a new API will be available in version X. This will be discussed in Chartres.***

Listing 1: sources.cmake file for G4mymodule

```
# - Header paths for external APIs
include_directories(${CLHEP_INCLUDE_DIRS})
include_directories(${USEDAPI_INCLUDE_DIRS})

# - Header paths for internal APIs
include_directories(
 ${PROJECT_SOURCE_DIR}/source/global/management/include
 )
include_directories(
 ${PROJECT_SOURCE_DIR}/source/materials/include
 )

# - Define the module
geant4_define_module(NAME G4mymodule
  HEADERS
    G4Foo.hh
    G4Bar.hh
  SOURCES
    G4Foo.cc
    G4Bar.cc
  GRANULAR_DEPENDENCIES
    G4globman
    G4materials
  GLOBAL_DEPENDENCIES
    G4global
    G4materials
  LINK_LIBRARIES
    ${USEDAPI_LIBRARIES}
  )
```

The file should begin by adding any include paths to headers used by your module with the `include_directories` command. One thing to note here is that you need to also add transient include paths. An example of this is the inclusion of the CLHEP headers in the above example. Even the `G4Foo` and `G4Bar` sources do not include CLHEP headers explicitly, they may include headers that do.

For example, they might include headers from G4globman that include CLHEP headers.

For deeply dependent modules, it's recognized that this can lead to a heavy cascade of include paths. The best strategy for determining include paths for you module is to start by only adding include paths for the headers your code actually uses. If you encounter compile errors with missing headers, add the required paths and rebuild. Iterate until your module builds without such errors.

Use the geant_define_module to declare your module. The NAME parameter sets the name the resultant library will have if it is built in granular mode. The HEADERS parameter should list all the files under the include directory that should be installed by your module. Similarly, the SOURCES parameter should list all the files under the src directory that should be compiled for your module. If your module has headers and or sources that depend on some aspect of the build (e.g. platform), you can create CMake lists containing the required sources and headers prior to calling geant4_define_module, for example

```
set(HDRS G4Foo.hh)
set(SRCS G4Foo.cc
if(WIN32)
  list(APPEND HDRS G4Foo_win32.hh)
  list(APPEND SRCS G4Foo_win32.cc)
endif()

geant4_define_module(NAME G4foo
  HEADERS
    ${HDRS}
  SOURCES
    ${SRCS}
  ...
  )
```

The GLOBAL_DEPENDENCIES parameter should list the names of all the Geant4 libraries your module needs linking to when building in global mode. The GRANULAR_DEPENDENCIES parameter should list the names of all the Geant4 libraries your module needs linking to when building in granular mode. Take care that the names of some granular libraries may be the same as the global library of

which they are part (G4materials in the above example). Finally, the `LINK_LIBRARIES` parameter should list all of the external libraries your module links to.

> ***It's clear this interface can be improved. The major items are a single library structure to avoid the awkward need to declare both granular and global dependencies, and to have a single point of contact to declare a dependency to avoid separate header and library entries. This should ideally handle the include paths and linking for transient dependencies (e.g. CLHEP in the above example). There are several use cases here to consider, for example header only usage, private usage, interface libraries. A secondary item, which is a to-be-decided convention/policy is the explicit listing of sources - see the discussion below.***

When you add or remove dependencies to your module, you should edit the `sources.cmake` file and add include paths and library links as needed. When you add or remove headers and source files, you should edit the `HEADERS` and `SOURCES` parameters to update the contents of your module. In both cases, the buildsystem generated by CMake is aware that your module has changed (the `sources.cmake` file becomes a dependency of your module), so the changes will be automatically picked up the next time you run a build.

Compared with the old GNUmake buildsystem, this puts more responsibility of you as a module maintainer as you must explicitly declare the sources of your module and keep this list up to date. This does mean inconsistent builds can creep in, and you can directly check the build consistency by building the `validate_sources` target. When built, this will report (to terminal or IDE browser) any mismatches between what is listed in the build and what is on disk.

CMake can locate source files using a globbing pattern via the `file` command, but this is strongly discouraged by the CMake developers. The reason for this is described in their documentation for the `file` command (author emphasis):

*GLOB will generate a list of all files that match the globbing expressions and store it into the variable. Globbing expressions are similar to regular expressions, but much simpler. If RELATIVE flag is specified for an expression, the results will be returned as a relative path to the given path. We do not recommend using GLOB to collect a list of source files from your source tree. If no CMakeLists.txt file changes when a source is added or removed then the generated build system cannot know when to ask CMake to regenerate.)*

What this means is that if a glob is used to create a list of source files, you have to rerun CMake manually whenever a file is added or removed from disk. For a purely personal project this might be o.k. In a large collaborative development environment under version control, it can quickly become frustrating as you have to flip between your buildtool and CMake to rerun the latter after every possible change, and this can quickly lead to inconsistent builds. Moreover, globbing doesn't work so well for certain use cases such as generated/platform dependent/optional source files. Having a mix of globbing and explicit listing is worse than either.

Explicit listing is therefore used mainly because it is supported and recommended by CMake. It is also used because it provides a single point of contact and interface for declaring how your module is built (even with globbing, you would still have to edit the file if include paths or linking change). This interface, namely, "edit `sources.cmake`; rebuild" is identical across all buildsystems, and only requires interaction with the buildsystem and version control after CMake is run. A further advantage is that it's transparent to changes in modules you don't maintain. If changes are pulled in after an update, you only need rebuild as the buildsystem will recognize that changes to a module have occurred and will reconfigure itself accordingly.

***The author (Ben Morgan) acknowledges that explicit listing is not to everyone's taste, and some will disagree strongly. The collaboration naturally makes the final decision. The author (Ben Morgan) recognizes that explicit listing has a learning curve for developers accustomed to GNUmake, but feels the effort***

*required to learn this pattern is a small investment for the return of more robust and reliable builds with a faster development cycle.*

### Using External APIs

Think carefully before using any third party APIs in Geant4. Is the functionality already provided internally by the toolkit itself? Is the API available and easy to install on all supported platforms, if not, how difficult is it to port? Are there other APIs that would provide the same functionality and which are more portable/available/supported? Is the license of the API compatible with the Geant4 license?

If after all of that, you still require the API for your code, **you must** contact the Software Management WG as soon as possible. This contact is needed to review the requirement for the API as well as assess the ease of integration into Geant4/CMake (e.g. locating the API on the installation system).

Assuming that the API is simply a header and library, then the general interface in CMake is to provide variables

```
find_package(Frobnicate) # Software Management
...
# Put headers in path
include_directories(${FROBNICATE_INCLUDE_DIRS})
# Build binary
add_library(uses_frob uses_frob.cc)
# Link libraries
target_link_libraries(uses_frob ${FROBNICATE_LIBRARIES})
```

## PART 5

# Developing Example Applications

Geant4 example applications generally use the following layout of source code and build scripts and macros:

```
+- AppName
   +- CMakeLists.txt  # Main build script
   +- AppName.cc      # Application main program
   +- AppName.mac     # Macro for runtime usage
   +- include/
   |  +- AppNameDetectorConstruction.hh  # Implementation
   |  +- ...
   +- src/
      +- AppNameDetectorConstruction.cc  # Implementation
      +- ...
```

This arrangement is required for backward compatibility with the GNUmake system.

The minimal `CMakeLists.txt` for an example application is shown in the following listing, where the comments should be referred to for explanations of the commands used and their purpose. A slight "trick" that should be noted is the use of the `configure_file` command to copy the macro from the source directory to the build directory. This is required if the application codes the execution of the macro into the main program using a relative path and you expect the application to be run from the build directory. Even if your application doesn't execute macros in code, it's helpful to copy all macros to the build directory so the built application can be run directly.

Listing 2: Minimal CMakeLists.txt for a Geant4 Application

```
# - Ensure CMake is >= 2.6
cmake_minimum_required(VERSION 2.6 FATAL_ERROR)

# - Define the project, usually same
# Name is usually the same the as final application
project(AppName)

# - Find Geant4
# REQUIRED argument makes CMake fail if it isn't found
find_package(Geant4 REQUIRED)

# - Include Geant4 "use file"
# This automatically sets required compiler flags
# compile definitions and header paths
include(${Geant4_USE_FILE})

# - Add headers of this project to the include path
include_directories(${PROJECT_SOURCE_DIR}/include)

# - Explicitely list implementation sources
# The headers are also listed so IDEs will include them
set(APPNAME_SRCS
  include/AppNameDetectorConstruction.hh
  # other headers here
  src/AppNameDetectorConstruction.cc
  # other sources here
  )

# - Create the actual executable from the main program
# source file and implementation sources
add_executable(AppName AppName.cc ${APPNAME_SRCS})

# - Link the executable to the Geant4 libraries
target_link_libraries(AppName ${Geant4_LIBRARIES})

# - Copy the macro to the build directory
# Ensures macro is accessible to app in build dir
configure_file(AppName.mac
  ${PROJECT_BINARY_DIR}/AppName.mac
  COPYONLY
  )
```

### Enabling Optional UI and Vis Drivers

As it stands, the minimal `CMakeLists.txt`, will build a working Geant4 application, but it will not include a user interface on any visualization. UI and Visualization can be activated by supplying extra "component" arguments to the `find_package` command. Rather than just doing this directly, we add a user-selectable option and only add these components if the user requests it. This allows the user can build either a batch mode or batch/interactive executable.

Listing 3: Adding UI/Vis Selection

```
...
# - Find Geant4
# REQUIRED argument makes CMake fail if it isn't found
option(APPNAME_USE_UIVIS "Enable Geant4 UI/Vis interfaces" ON)
if(APPNAME_USE_UIVIS)
  find_package(Geant4 REQUIRED ui_all vis_all)
else()
  find_package(Geant4 REQUIRED)
endif()
...
```

When building the application, all available UI and Vis drivers in the build of Geant4 found will be activated by default, as the `APPNAME_USE_UIVIS` option is on by default. This can be changed when running CMake to configure the project.

*How to activate only the required drivers? Can a simple interface be designed?*

***All component options need to be documented fully. It's possible some component names may change.***

### Making Example Applications Installable

There is no requirement to make your example installable.

*Should there be?*

However, it is easy to add this functionality. Simply use CMake's `install` command:

Listing 4: Installing the AppName Application

```
...
add_executable(AppName AppName.cc ${APPNAME_SRCS})
target_link_libraries(AppName ${Geant4_LIBRARIES)

install(TARGETS AppName DESTINATION bin)
...
```

A new `install` target will be added to the buildsystem (so `make install` works with Makefiles!). The `DESTINATION` argument sets the directory where the application will be installed. If a relative path is given, it will be interpreted relative to `CMAKE_INSTALL_PREFIX`.

> ***This does not work if the application uses macros directly, without extra work.***

## PART 6

## Working with Integration Tests

## PART 7

## Working with Unit Tests