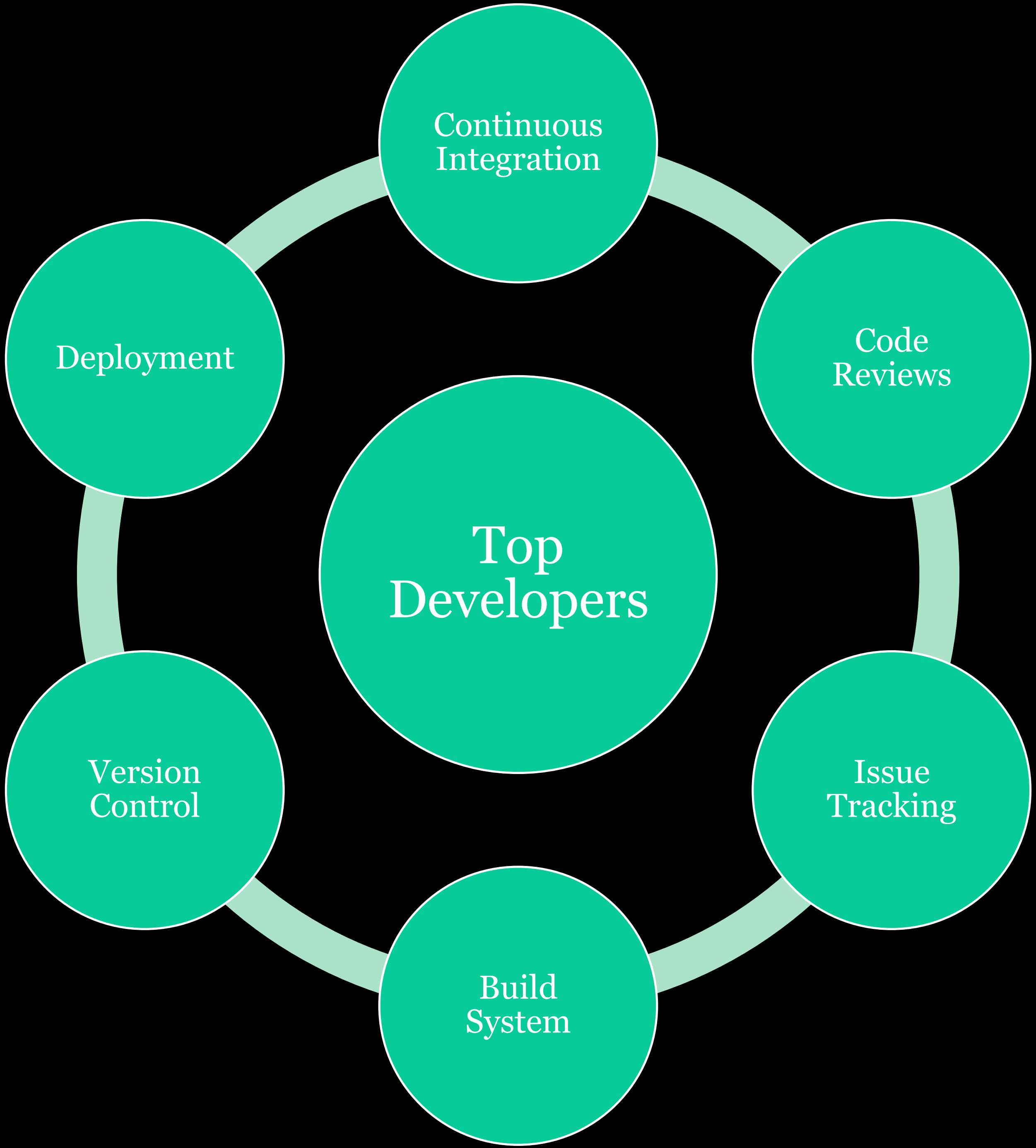
# Large Projects and CMake and git, <u>oh my</u>!

C++Now 2015

David Sankel, Stellar Science

# Recognition

- Will Dicharry (Stellar Science)
- John McIver (Stellar Science)
- Conrad Poelman (Stellar Science)
- K. R. Walker (Stellar Science)



Top Developers

Continuous Integration

Code Reviews

Issue Tracking

Build System

Version Control

Deployment

# So, why version control and build systems?

# We’ve got a problem.

- A dozen build systems, each incompatible with the other.

# We’ve got a problem.

- A dozen build systems, each incompatible with the other.
- A handful of version control systems, each person chooses his favorite.

# We’ve got a problem.

- A dozen build systems, each incompatible with the other.
- A handful of version control systems, each person chooses his favorite.
- Everyone uses each of them differently.

End result?

High cost to use third party code.

Wheels reinvented.

High cost to use third party code.

Wheels reinvented.

High cost to use third party code.

Updates not propagated.

Wheels reinvented.

High cost to use third party code.

Updates not propagated.

Old versions proliferate.

Wheels reinvented.

Buggy code!

High cost to use third party code.

Updates not propagated.

Old versions proliferate.

**a·so·cial** (āˈsōSHəl) *adjective* avoiding social interaction; inconsiderate of or hostile to others.

**et·i·quette** (ˈe-ti-kət, -ˌket) *noun* the rules indicating the proper and polite way to behave

# How to choose?

- Incremental steps from already adopted tools.
- Works for the majority of situations.
- Engineering superiority

Which version control system is the polite one to use for C++?

No!

No!

No!

No!

Git is the only viable option.

But how do we use it?

# Laying out your git project with submodules

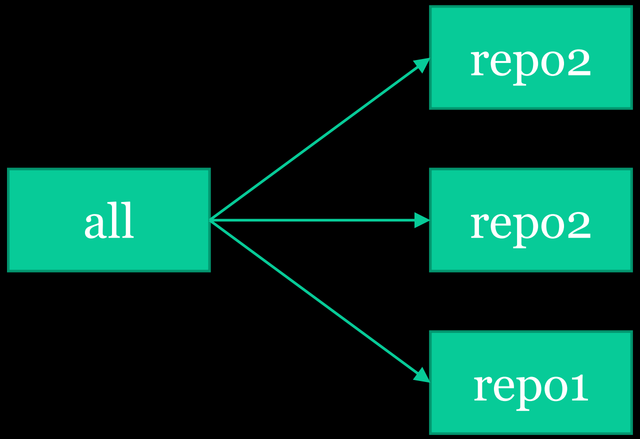
| On your server: | On your client: |
|---|---|
| /git/all.git | all/ |
| /git/repo1.git | all/repo1 |
| /git/repo2.git | all/repo2 |
| /git/repo3.git | all/repo3 |

```
git submodule add –b master ../<newRepoName> <newRepoName>
```

# Some Rules

- The ‘all’ superproject contains everything.
- All your submodules are hosted on your project’s git server.
- Security is handled based on the granularity of repositories.
- Your submodules do not have their own submodules.
- Other superprojects that don’t include everything can be created on an as-needed basis.

# Automatic submodule pointer updates



# Automatic submodule pointer updates

- Add post-receive hook on server to update the submodule pointers.

# Typical Usage

- Initial Checkout

```
git clone ssh://git.wherever.com/git/all.git
cd all
git submodule update –init
git submodule foreach git checkout master
```

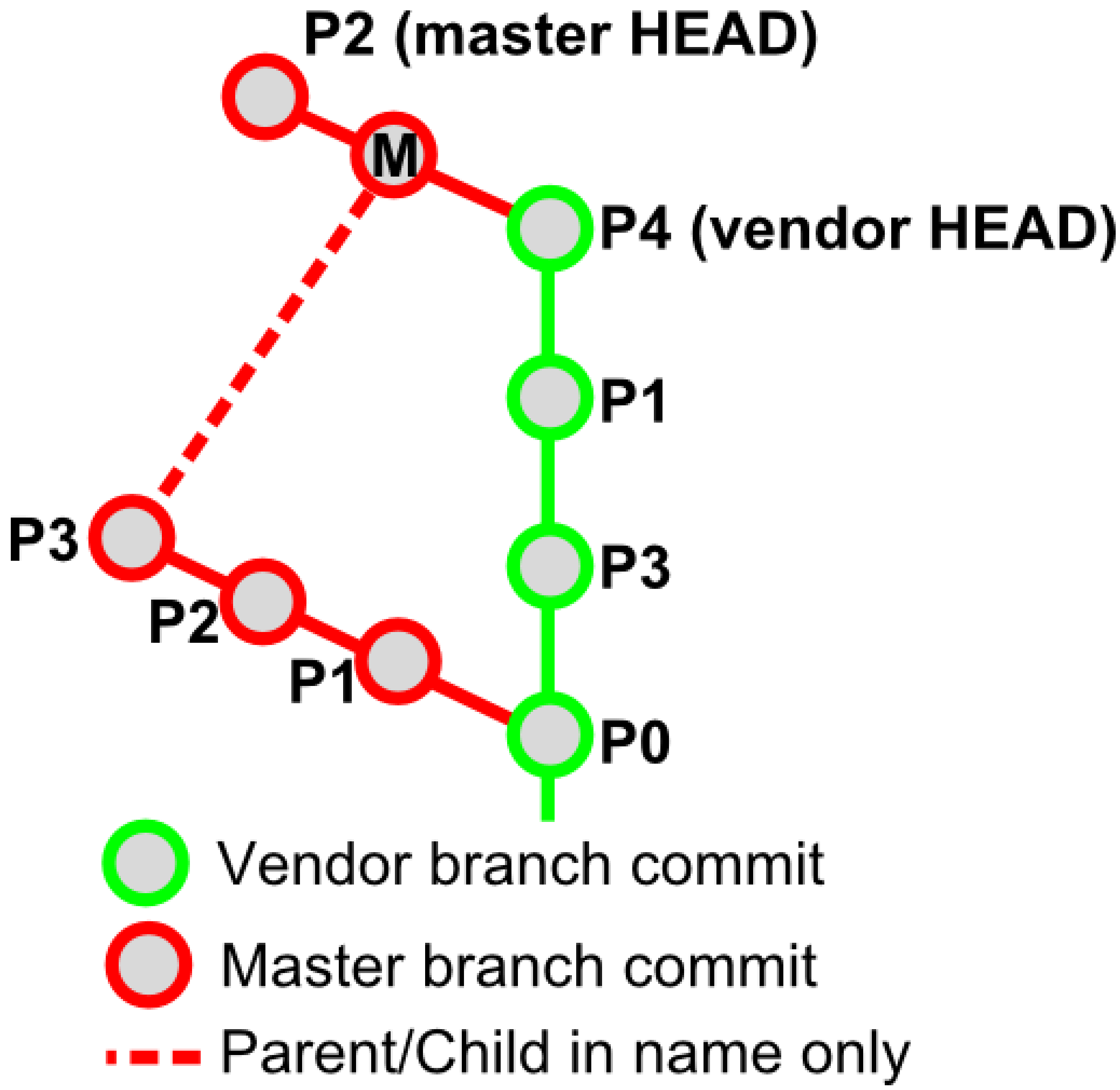
- Commit a modification

```
cd all/repo1
git checkout -b my_branch
# make modifications
git gui # commit locally
git push origin my_branch:my_branch # Push back to server
```

# Third Party Libraries - Requirements

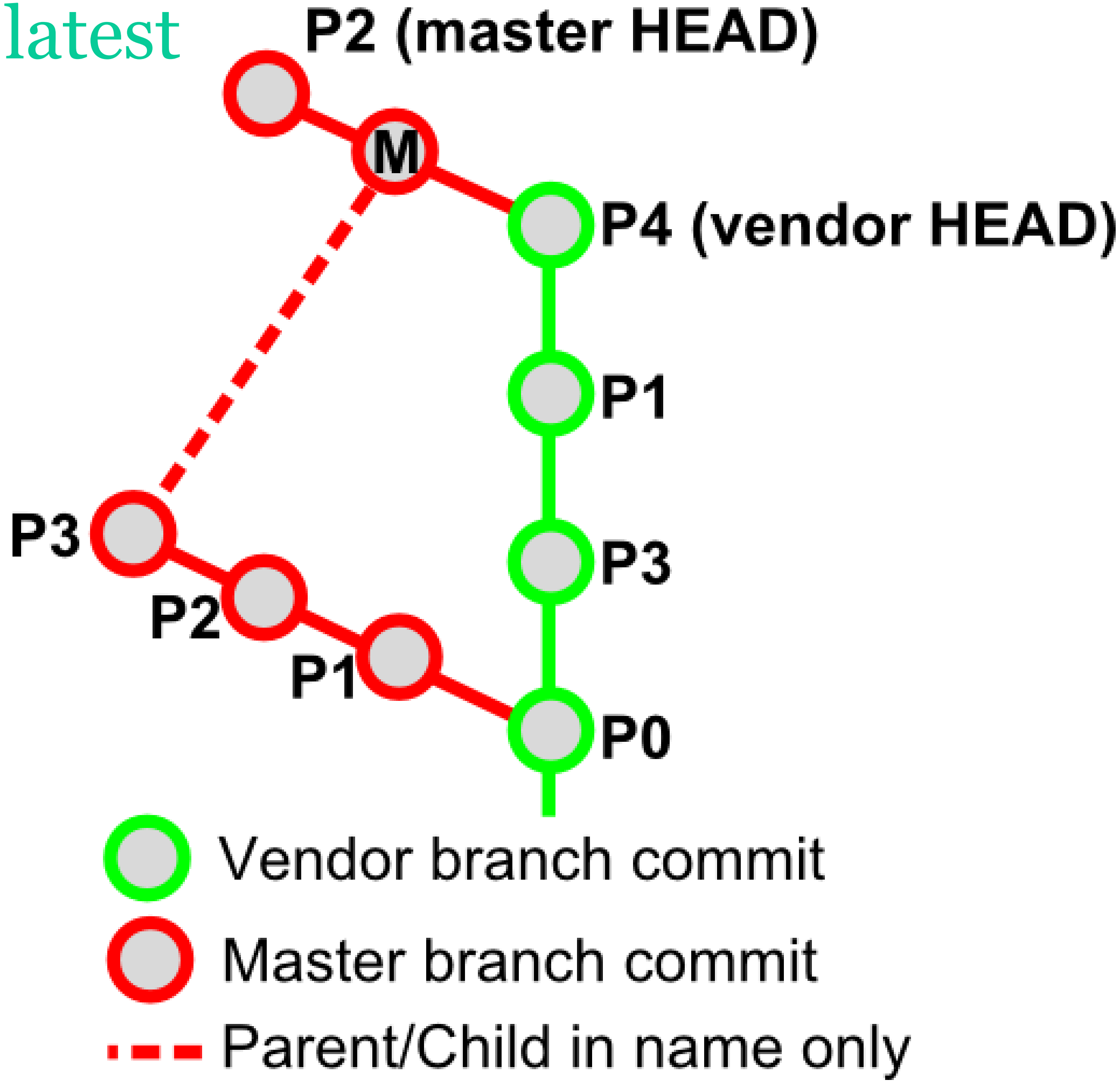
- Seemless to use.
- Host locally.
- Apply fixes before the maintainer applies them.
- Encourage contribution.
- Easy to update.

# Bounce patching



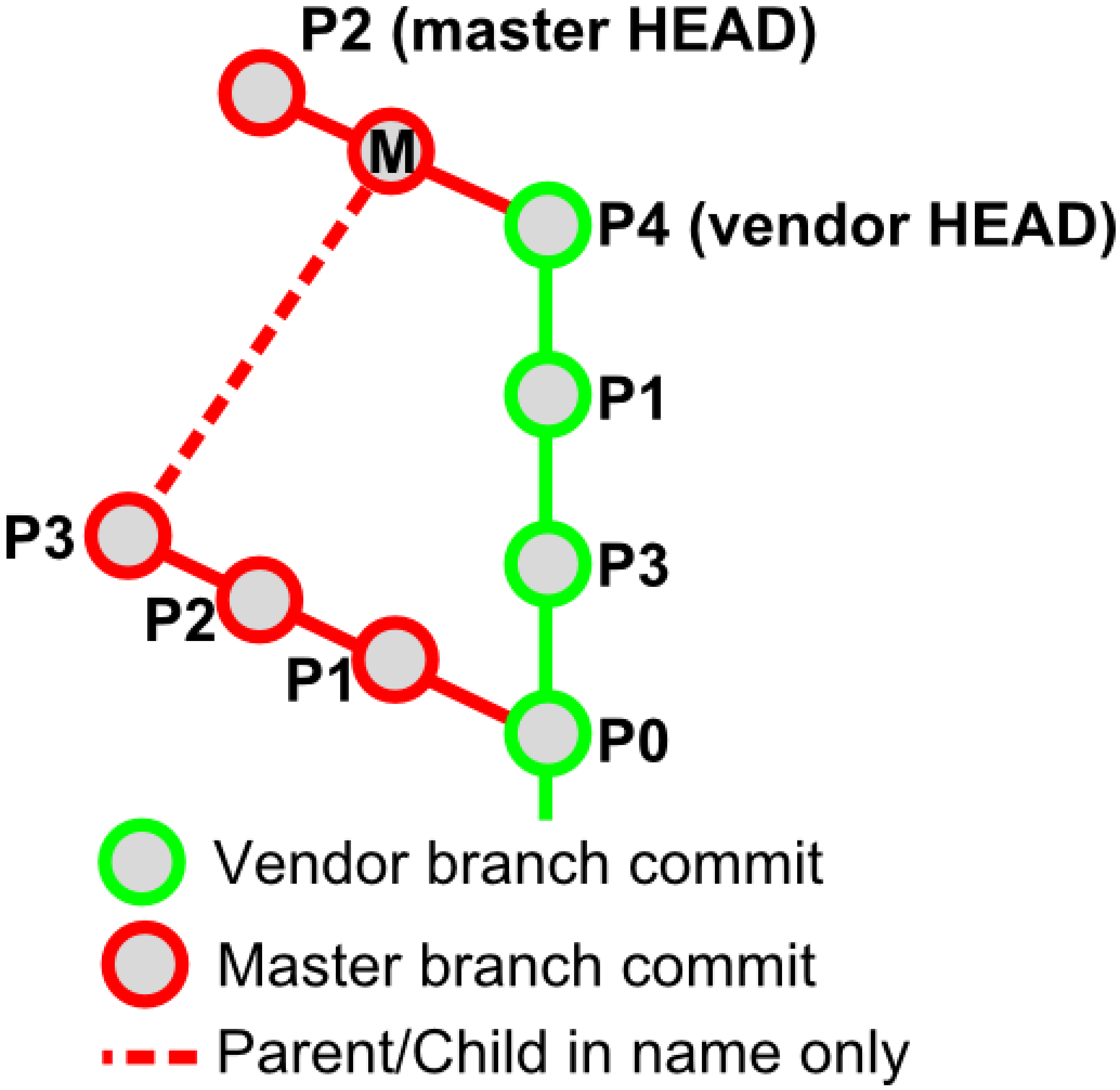
# Update a 3rdparty repo to the latest

```
cd all/repo3
git checkout vendor
git pull
git checkout –b phantom_merge
git merge –s ours master –m “Phantom merge”
git checkout master
git merge --ff-only phantom_merge
git branch –D phantom_merge
git cherry-pick 924989v # etc.
# test
git push origin master:master
```



# Modify a 3rdparty repo

```
cd all/repo3
# make changes
git gui # commit as usual

git checkout vendor
git merge –ff-only origin/vendor
git checkout –b my_change
git cherry-pick 293h39
git push https://github... my_change:my_change
# Use github to make a pull request.
```



# Keep Vendor branch up-to-date always

- Cron job on git server to periodically update the vendor branch to the latest “master” branch on upstream server.

# Non-git libraries ☹

- Make a git repository with a vendor branch corresponding to release dumps.
- Make use the git-svn tools to make a vendor branch and automate keeping it up to date.

So, lets talk about build systems.

Yikes!

# CMake

- Cross compiles.
- Generates Visual Studio projects.
- Generates Ninja files.
- Well supported commercially.
- Kinks worked out over 16 years of use.
- Vast majority of new C++ Open Source projects use CMake.

But, how do we use CMake properly?

Step 1: Write up CMake coding conventions.

# Coding Conventions

- Which CMake version line are you going to use?
- Call functions with UPPERCASE or lowercase names? (lowercase)
- Indentation
- Use `add_definitions` or `target_compile_definitions`? (the latter)
- Use QUIET with find_package.
- Package specific conventions (such as BOOST_ALL_NO_LIB for Boost)
- …

# Dependencies the Wrong Way

- Build each repository in isolation and generate its binaries along with a CMake config file.
- For each project that has dependencies, use ‘find_package’ to load the config file and use the library.

Problem 1: Too tedious. Updating a program implies recompiling its package and then every one of its dependencies manually.

Problem 2: The “export” mechanism used to do this only works with libraries at the base of the dependency graph.

# What we want…

- One build will build the project and all dependencies.
- *Only* required dependencies are built.
- The standard `find_package` libraries (Zlib, Boost, etc.) will also add the source to these to our build, if the source is there, instead of using system versions.

# Example project

```
all/CMakeLists.txt

all/gpc/base/CMakeLists.txt
all/gpc/xio/CMakeLists.txt
...

all/cwf/base/CMakeLists.txt
all/cwf/rf/CMakeLists.txt
all/cwf/rf/Foo.h
all/cwf/rf/Foo.cpp
...
```

# Outline of a CMakeLists.txt file

```
cmake_minimum_required(VERSION 3.1)

project( cwf_rf )

find_package(Boost COMPONENTS regex REQUIRED QUIET)

import( “cwf/base” )
import( “gpc/xio” )

add_library( cwf_rf
  Foo.h
  Foo.cpp
  # ...
)

target_include_directories(cwf_rf PUBLIC ../../ ${Boost_INCLUDE_DIRS})
target_compile_definitions(cwf_rf PUBLIC BOOST_ALL_NO_LIB)
target_link_libraries(cwf_rf
  cwf_base
  gpc_xio
  ${Boost_REGEX_LIBRARY}
)
set_target_properties(cwf_rf PROPERTIES FOLDER “cwf_rf")
```

# Poor man’s import

```
import( “cwf/base” )
```

⬇

```
if( NOT TARGET cwf_base )
  add_subdirectory( “../../cwf/base” “${CMAKE_BINARY_DIR}/cwf/base” )
endif()
```

# Fancy import implementation sketch

**import( <RelativePath> )**

- Use a stack to discover dependency loops.
- Use global properties to determine if a directory has already been added (see get_property function)
- Automatically figure out the binary directory to put the library build files into.
- Call add_subdirectory

**add_import_search_dir( <AbsolutePath> )**

- Add the specified path to the list of import search directories. Akin to “-I”.

# find_package fun

- The default find_package tool was intended to be used for precompiled libraries.
- This was a flawed intention. Lets fix it…

# Example project

```
all/CMakeLists.txt

all/gpc/base/CMakeLists.txt
all/gpc/xio/CMakeLists.txt
...

all/cwf/base/CMakeLists.txt
all/cwf/rf/CMakeLists.txt
all/cwf/rf/Foo.h
all/cwf/rf/Foo.cpp
...

all/zlib/CMakeLists.txt
...

all/CMakeModules/FindZLIB.cmake
```

# all/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.1)

project(all)

list(APPEND CMAKE_MODULE_PATH
  ${CMAKE_CURRENT_SOURCE_DIR}/CMakeModules
)

# ...
```

# all/CMakeModules/FindZLIB.cmake

```
if( NOT TARGET zlib)
  add_subdirectory( “../zlib” “${CMAKE_BINARY_DIR}/zlib” )
endif()

set( ZLIB_INCLUDE_DIRS
  "${CMAKE_CURRENT_SOURCE_DIR}/../zlib”
  “${CMAKE_BINARY_DIR}/zlib”
)
set( ZLIB_LIBRARIES zlib )
set( ZLIB_FOUND TRUE )

set( ZLIB_VERSION_STRING “1.2.8” )
# …
```

# Notes on customized find modules

- Customized find modules must match semantics of original.
  (cmake --help-module FindZLIB).
- Making find modules that work with other build systems requires some more work...

# Special Cases (like Boost & Qt)

Two options:

1. Write CMakeLists.txt files to build the project.
2. Wrap the other build system.

# Notes on wrapping other build systems.

General process:

- Use globbing to construct a list of all the files that are required to build the foreign target.
- Use results of find_package to configure the foreign build system to use libraries generated by CMake.
- Create a custom command and custom target that executes the build. The command is declared to generate the libraries it generates.
- Create an empty library that depends on the library's dependencies.
- Make the custom target depend on the dependency transfer target.
- Use CMake's "client requirements" feature to make users of the custom target link to the libraries it generates.

# Selectively Enabling Projects

- With a large set of projects, it’s nice to select a subset.
    - Quicker build and dependency checking.
- CMake doesn’t have a way to do that.
    - Lets make it happen!

# all/CMakeLists.txt

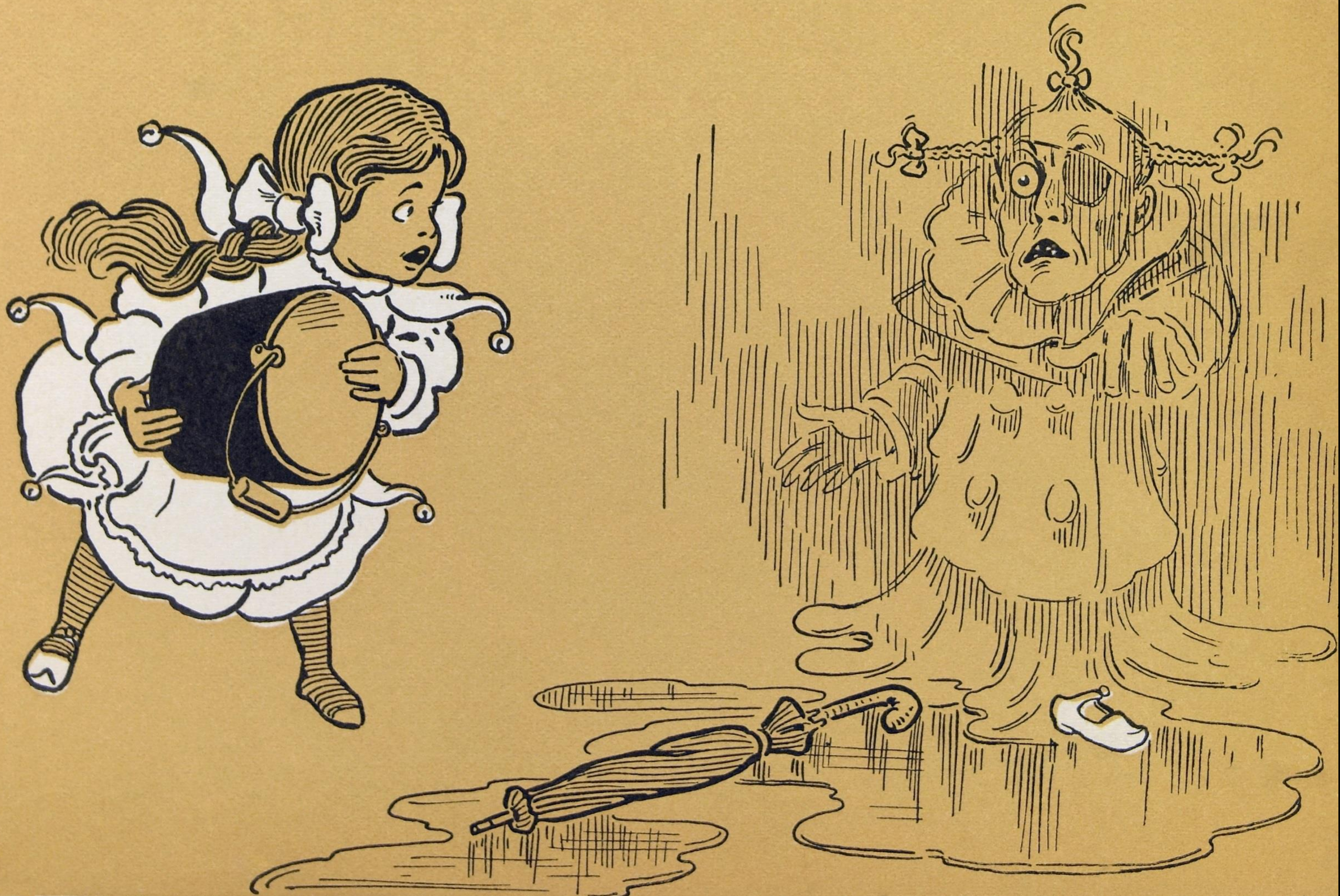
```
cmake_minimum_required(VERSION 3.1)

project(all)

list(APPEND CMAKE_MODULE_PATH
  ${CMAKE_CURRENT_SOURCE_DIR}/CMakeModules
)

file( GLOB project_files
  "${CMAKE_CURRENT_SOURCE_DIR}/*/Project.cmake“
)

foreach( project_file ${project_files} )
  include( ${file} )
endforeach()
```

# all/gpc/Project.cmake

```
option( all_gpc_enable “Enable GPC" FALSE )
if( ${all_gpc_enable} )
  import( “gpc” )
endif()
```

# CMake misc. takeaways

- Enforcement of strict conventions is very important.
- It does the job and, when done carefully, can be pretty nice.



# Some Results from Stellar Science

- System in use now with >40 active developers on dozens of projects.
- Contributions to Open Source projects are frequent.
- Multi-platform development and multi-platform deployment is the rule more often than the exception.

CMake and Git…

Mind your manners.