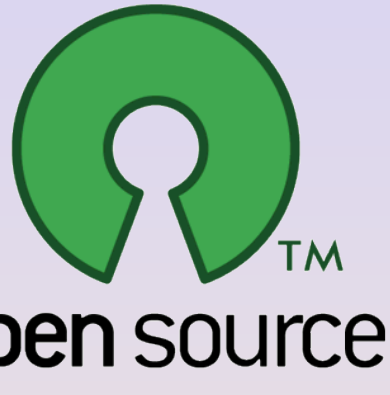
Project Management

The COIN-OR Way

Ted Ralphs

LEHIGH
UNIVERSITY

COR@L
COMPUTATIONAL OPTIMIZATION RESEARCH AT LEHIGH



COIN fORgery: Developing Open Source Tools for OR

Institute for Mathematics and Its Applications, Minneapolis, MN

Preamble

- I'll touch on a ton of stuff here, some of which may already be familiar to some.
- It wasn't clear exactly what order to go in or how much background knowledge to assume.
- Please do stop me as often as you like and we can drill down on topics of interest.
- I may also say something that you disagree with.
- My thoughts on topics like the proper workflow for COIN in the Github environment is still evolving.
- Let's have some good discussion!

Outline

1. Intellectual Property
2. Toolbox
3. Version Control
   - Versioning
   - `svn` and `git`
   - Utilities
4. Dependencies
5. Repository Contents

Intellectual Property: Ownership

- It is important to know who owns your code, since it may not be you.
- In the U.S., your employer probably owns your code, even if you're an academic, but this may not be true in other countries.
- Only the legal owner may license the code, which is a necessary step for making your project open source and useful to others.
- It is important o carefully track contributions to your project by others so that the entirety of your project remains legally licensed.
- COIN-OR requires all contrbutors to sign legal paperwork certifying that they understand all this.
- One of the reasons companies feel comfortable using code from COIN-OR is because we have guidelines in place that ensure the prevenance of code we re-distribute.

Intellectual Property: Licensing

- All open source software must be distributed wth a license.
- Without a license, you are technically not explicitly granting any rights to anyone to do anything with your code.
- All code re-distributed by COIN-OR must be under some open source license.
- The recommended license is the Eclipse Public License 2.0.
  - Originally developed by IBM, but current steward is the Eclipse Foundation.
  - Friendly to commercial use.
  - Standard version is incompatible with the GPL.
  - There is a new secondary license that may be optionally adopted to allow compatibility with the GPL.

Outline

1. Intellectual Property
2. Toolbox
3. Version Control
   - Versioning
   - `svn` and `git`
   - Utilities
4. Dependencies
5. Repository Contents

Toolbox: Infrastructure

- Legacy
  - TRAC for Wiki, issue tracking, source code browser.
  - Centralized SVN repo for source repository and version control.
  - Mailman for mailing lists.
  - Raw HTML (uploaded via SVN) for Web site
  - Jenkins for testing and continuous integration.
  - Binaries uploaded manually via SVN for distribution.
  - Automatic creation and distribution of release tarballs through post-commit hooks.
- Future
  - Github for Web site, Wiki, issue tracking, mailing list, source code browser, source repository, distribution of source tarballs.
  - Travis and Appveyor for testing and continuous integration.
  - Binaries built automatically and uploaded to Bintray for distribution.
  - Zenodo for assigning DOIs to releases (more on this later).

Move to Github

- Per the previous slide, we are in the midst of a move from our own infrastructure to infrastructure hosted on Github.
- It appears that Github is a platform with a stable future that will continue to support open source.
- However, we have to be aware of the possibility that it will either go away or will no longer be free.
- Most (but not all) of what is stored in Github would be easy to move if it came to it.
- An alternative to consider is a self-hosted Gitlab, which provdies similar functionality.
- In any case, an important aspect of all this change is that we will be moving from `svn` to `git`.
- This has far-reaching implications.

COIN-OR on Github

- There are currently three GitHub organizations associated with COIN-OR.
  - `coin-or`: Main organization for hosting projects.
  - `coin-or-tools`: Organization for hosting infrastructure utilities, such as BuildTools and wrappers for third party codes.
  - `coin-or-bazaar`: Envisioned collection of templates, examples, toy codes, and other fun COIN-related stuff.
- Please consider contributing some stuff to the Bazaar, it has not really been advertised or utilized.

Toolbox: Utilities

- There are a number of utilities implemented to automate certain procedures.
- These are mostly part of the `BuildTools` project.
  - Scripts for creating new stable and release versions.
  - Utilities for managing dependencies.
  - Scripts and templates for building versioned libraries and binaries.
  - Post-commit hooks for automatic processing of commits (copying files to Web server, posting binaries, etc.)
  - Wrapper libraries for third-party codes.
- Most of the utilities assume a hosted `svn` repository and will need to be re-implemented for use with `git`.

Outline

1. Intellectual Property
2. Toolbox
3. Version Control
   - Versioning
   - `svn` and `git`
   - Utilities
4. Dependencies
5. Repository Contents

Version Control and Versioning

- COIN numbers versions by a standard semantic versioning scheme: each version has a *major*, *minor*, and *patch/release* number (see `http://semver.org/`).
- All version within a *major.minor* series are compatible.
- All versions within a *major* series are backwards compatible.
- Legacy top-level organization of the repositories (`svn`)

Subversion Repo Layout for Project

```
html/
conf/
branches/
trunk/
stable/
releases/
```

  - `Trunk` is where development takes place (bleeding edge).
  - `Stable` versions have two digits and are continuously patched with fixes and updates.
  - `Release` versions have three digits and are fixed forever.

Libtool Versioning

- Linux distributions use a different, but related, versioning scheme called *libtool versioning*.
- This scheme is based on
  - *current*: the most recent interface number that this library implements.
  - *revision*: the implementation number of the current interface, and
  - *age*: the difference between the newest and oldest interfaces that this library implements.
- In other words, each time the interface changes, we increment *current*.
- If the change is backwards compatible (additions but no deletions), then we increment *age*.
- We increment *revision* for bug fix releases.
- Generally speaking, thee is a one-to-one mapping between these version numbers and semantic version numbers.

Importance of Versioning

- **For users**
  - Allows stability and isolation from breaking changes, but retains the ability to get important patches.
  - Makes it easier to report bugs and get quick fixes.
  - Users obtaining code through version control should generally get the latest stable version.
  - Users downloading a fixed release zip/tarball just want the latest release.
- **For developers**
  - Makes it easier to reproduce bugs reported by users.
  - Makes citation easier for scientific research.
  - Makes reproducibility easier for scientific research.
  - Makes debugging easier generally (going back to working version)
  - A *must* if you want your software packaged!

From svn to git

- This is a big topic so I'm going to come at it a little at a time.
- If you don't already know git and svn to some extent, some things may not make sense right away.
- I'm going to stay away from the religious aspects of the comparison.
- git has a fairly steep learning curve if you learn by Googling.
  - There are at least a dozen completely different ways of solving any given problem.
  - There are a lot of people who know just enough to be dangerous.
  - git is extremely powerful, so it's not that difficult to screw up (at first).
  - svn is far more restricted, but bullet-proof.

How svn Works

- Repo is organized into folders, each containing versioned files.
- Each commit consists of a changeset containing patches to each file.
- The repository is a collection of initial files and a collection of such patches (roughly speaking).
- The revision number of the entire repository is incremented with every commit.
- Copying a file (or folder) creates a new independent version of it.
- Stoarge is centralized and the central repository is the only source of truth.
- Commits are sent immediately to the central repository.
- When you check out part of the repo, you only get the current revision of whatever you check out.
- It is difficult to erase history in an svn repo.

How `git` Works

- `git` is more accurately a *versioned file system*.
- The core is surprisingly simple and straightforward, but the "porcelain" is frustratingly difficult to master.
- Each commit is a snapshot of the entire local filesystem at the time.
- When you clone a repository, you get your own local copy of the entire history of every commit.
- Commits are initially stored locally and may or may not be pushed out to other repositories.
- Each commit has one or more parents and are organized into a directed acyclic graph.
- There is generally no central source of truth (except as agreed upon).
- It is easy to erase history and obliterate a git repository if you don't know what you're doing!

Versioning with Version Control

- Philosophically
  - Generally, there is one long-running “trunk” of development.
  - Stable versions are split off the trunk and continue to receive relevant patches (that may or may not also be committed to trunk).
  - Releases are snapshots of stable versions.
  - Feature branches can also be split off of trunk and later merged back in for the purpose of developing individual features.
- Practically
  - There are lots of variations on this theme.
  - The practical aspects depend on which VCS you’re using.
  - I’ll focus mainly on how to implement this with git

Example Workflows

With svn

```
svn copy https://projects.coin-or.org/svn/CoinUtils/trunk \
  https://projects.coin-or.org/svn/CoinUtils/stable/2.11
svn co https://projects.coin-or.org/svn/CoinUtils/trunk CoinUtils-trunk
cd CoinUtils-trunk
...
svn commit -m "Made some changes"
cd ..
svn co https://projects.coin-or.org/svn/CoinUtils/stable/2.11 CoinUtils-2.11
cd CoinUtils-2.11
svn merge -c 100 ../trunk .
```

With git

```
git clone https://github.com/coin-or/CoinUtils
cd CoinUtils
git branch stable/2.11
...
git add ChangedFile.cpp
git commit -m "Made some changes"
git push
git checkout stable/2.11
git merge master
git push --set-upstream origin stable/2.11
```

Basic `git` Workflow

- Development takes place in the `master` branch.
- Branches named `stable/x.y` are created for stable versions.
- Bugs are fixed in `master` and ported to stable versions either through a `merge` or a `cherry-pick`.
- Releases are tags named `releases/x.y.z`.
- Note that Github also has a separate mechanism for creating a new release that triggers additional operations (more on this later).
- A slight alternative is the “git flow” workflow in which development is done in a separate `devel` branch and master contains only snapshots.
- This has the advantage that when users clone the repo, they get a working version.
- Feature branches can be split from master and merged back later, as needed.

Helper Scripts

- There are several helper scripts intended to ease the burden of creating new stable version and releases.
- There are a number of steps involved and it helps to automate them, as it's easy to forget one.
- These helper scripts currently work with `svn` and need to be re-implemented!
- Steps in creating new release.
  - Determine release number
  - Automatically determine proper dedepencies
  - Modify `configure.ac` with appropriate version numbers (including the correct libtool version)
  - Re-run autotools to create new `configure`, `Makefile.in`, etc. build scripts.
  - Build and run unit tests.
  - Commit code and tag/copy release version.
  - Change version numbers and dependencies backm re-run autotools and commit result.

Drafting a release on Github

- In Github, every tag is listed as a “release”, but there is a mechanism for drafting “official” releases that adds a few more options.
  - Can be labeled as a pre-release.
  - Can attach binaries.
- Drafting a release also kicks of a hook that creates a DOI in Zenodo if integration is enabled.
- This is very useful for allowing citation of your code.

Bug Reports

- Bug reports can be made through Github's issue tracking.
- This is a big improvement over TRAC in general, but needs some tweaking to have the same level of functionality.
- We will want to implement some custom templates for reporting different kinds of issues.
- Github now supports the creation of different templates for different purposes (bug report, feasture request, etc.).

Pull Requests

- A huge advantage of `git` over `svn` is the ability of users to "fork a repository and extend the code independently.
- This cannot be easily done with `svn`.
- In Github, forking a repository is as easy as clicking the "Fork" button.
- This clones the repository into your Github account.
- If you want to submit a fix or extension, you can create a "Pull Request".
- The user's remote branch can be merged as if it were a local feature branch.

Mirroring

- Currently, many projects are still managed using subversion.
- They are being mirrored to Github using a toold called `subgit`.
- This means that any commit directly to github will be overwritten.
- Users can submit pull requests, but they need to be applied as patches in `svn`.
- I hope to switch most projects to `git` management in the relatively short term and disable mirroring.
- Github supports checking out code with a `svn` client and also allows committing through the client, but this should be avoided to the extent possible.

Aside: Command-line versus GUI

- I personally advocate use of the command line for many kinds of operations because it's easier.
- In the case of git, it's very useful to have a GUI interface.
- For lots of reasons, this can streamline work flows.
- I've been using `SourceTree`, which is nicely integrated with `bitbucket`.
- There are many other options out there.

Source Tree Organization

- The source tree for project Xxx looks something like:

Source Tree for Project Xxx Root

```
Xxx/
Yyy/  ⇐ dependency1
Zzz/  ⇐ dependency2
doxydoc/
INSTALL
Dependencies
configure
Makefile.am
...
```

- The files in the root directory are for doing monolithic builds, including dependencies (listed in the `Dependencies` file).
- If you only want to build the project itself and lnk against installed binaries of other projects, you only need the `Xxx/` subdirectory.
- Support for monolithic builds will be going away along with the root directory as we move from `svn` to `git`.

Outline

1. Intellectual Property
2. Toolbox
3. Version Control
   - Versioning
   - `svn` and `git`
   - Utilities
4. Dependencies
5. Repository Contents

Handling Dependencies

- External dependencies are listed in the `Dependencies` file.
- Under `svn`, dependencies were pulled in using the *externals* mechanism.

```
svn propset svn:external -F Dependencies .
```

- The directory structure was designed around the way the legacy build system works.
- It depends on the ability of `svn` to check out subdirectories.
- This cannot be done in a natural way with `git`.
- Until recently, there was no analog of the `svn` externals mechanism that was easy to work with, but now `git` submodules are a good solution.
- For the time being, we would still like to be able to mix external projects in bith `git` and `svn`, however.
- This is accomplished by the `get.dependencies` script.

What Should the Dependencies Be?

- It is a good question what versions of dependent projects your project should depend on.
- The only hard-and-fast rule is that releases should depend on releases.
- Typically, stables should depend on stables.
- Trunk/master may depend on either stables or other trunks, as appropriate.

The `get.dependencies` Script

The `get.dependencies` script is a `bash` script that replaces the externals mechanism and the monolithic build mechanism.

- The `fetch` command gets dependencies (using `git` or `svn` for mirrored projects) and optionally downloads third-party codes.
- The `build` command builds dependencies in order, pre-installs them in the build directory, and optionally runs unit tests.
- The `install` command installs all code in the final location.

To obtain the script, do

```
git clone https://github.com/coin-or-tools/BuildTools
svn co https://projects.coin-or.org/svn/BuildTools/trunk BuildTools
```

Getting bash

- To run the get.dependencies script (and other utilities), you need bash.
- In Linux, you should already be using it.
- In OS X, you are already using it, but the version provided by Apple may be old and should be updated by getting a recent version with homebrew.
- In Windows, there are several options.
  - Cygwin: I no longer recommend and it won't be supported going forward.
  - MSys2: A lightweight Unix environment with a nice package manager for installing commands (port of Arch Linux), highly recommended.
  - Windows Subsystem for Linux: A full Linux installation within Windows, very convenient for using Linux packages in Windows, highly recommended.

Use of pkg-config

- The pkg-config utility is used to track dependencies.
- pkg-config is a widely-used used cross-platform dependency management system.
- Dependencies are recorded in a .pc file.
- Using the build system, these files are generated automatically from dependencies indicated in configure.ac.
- As fallback, if project sources are found in standard locations relative to main source, pkg-config is not required.
- On Windows, there may be problems due to path issues if using the Windows version of pkg-config with builds using bash.
- You should be able to avoid this by installing pkg-config the same way you installed bash.

Contents of a `.pc` file

```
prefix=/home/ted/Projects/OptimizationSuite/build
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include/coin

Name: CBC
Description: COIN-OR Branch and Cut
URL: https://projects.coin-or.org/Cbc
Version: trunk
Libs: -L${libdir} -lCbcSolver -lCbc
Cflags: -I${includedir}
Requires: coinasl osi-clp cgl osi coinutils
```

Using `pkg-config`

`pkg-config` can be used to obtain a list of libraries to link or a list of flags required to build a library/binary.

```
~> pkg-config --libs clp
-L/home/ted/Projects/OptimizationSuite/build/lib -lClpSolver -lClp \
-lcoinasl -lm -ldl -lcoinmumps -lgfortran -lm -lquadmath -lCoinUtils \
-lz -lm -lcoinglpk -ldl -lm -lcoinlapack -lgfortran -lm -lquadmath \
-lcoinblas -lgfortran -lm -lquadmath
```

It can also be embedded into a Makefile to automate building.

```
PKG_CONFIG_PATH=/home/ted/Projects/OptimizationSuite/build/lib/pkgconfig
LIBS = `pkg-config --libs cbc`
```

ThirdParty Dependencies

- There are a number of open-source projects that COIN projects can link to, but whose source we do not distribute.
- We provide convenient scripts for downloading these projects (shell scripts named `./get.Xxx`) and a build harness for build them.
- We also produce libraries and `pkg-config` files.
  - AMPL Solver Library (required to use solvers with AMPL)
  - Blas (improves performance—usually available natively on Linux/OS X)
  - Lapack (same as Blas)
  - Glpk
  - Metis
  - MUMPS (required for Ipopt to build completely open source)
  - Soplex
  - SCIP
  - HSL (an alternative to MUMPS that is not open source)
  - FilterSQP
- The `get.dependencies` will automatically download third party code by default.

Outline

1. Intellectual Property
2. Toolbox
3. Version Control
   - Versioning
   - `svn` and `git`
   - Utilities
4. Dependencies
5. Repository Contents

Source Tree Organization (Project Subdirectory)

- The source tree for project Xxx looks something like:

Source Tree for Project Xxx Subdirectory

```
src/
examples/
MSVisualStudio/
test/
AUTHORS
README
LICENSE
INSTALL
configure
Makefile.am
...
```

- The files in the subdirectory are for building the project itself, with no dependencies.
- The exception is the `MSVisualStudio/` directory, which contains solution files that include dependencies.

README

- It is useful to rename your README as README.md so as to be able to use Markdown to make it display nicely.
- The README.md in your `master` branch is what displays by defult when someone visits your repository.
- Some things to have in our README
  - Build instructions
  - Change Log
  - Badges
    - Build Status (Travis, Appveyor)
    - Code Quality (Codacy, etc.)
    - Download (Bintray)
    - Citation DOI (Zenodo)

Project Web site

- It is simple to host a project web site on Github.
- Simply create a branch (called `gh-pages` by default) and add files there.
- The site will be accessible at the URL

  `https://coin-or.github.com/Xxx`

- It is even possible to use Travis to build a statis site using Jekyll or the like.