Software Carpentry

Tools and approaches for more effective scientific software development.

C. Titus Brown

Bronner-Fraser Lab, Caltech /

/ Michigan State University

Background

- Effective software development is important for research.
- Scientists tend to be poorly trained. (I'm no exception.)
- Relatively small investment in tools and techniques can yield big rewards.

- Secret: industry sucks at developing software, too.

“Software Carpentry”

- Not science per se ;)
- About using free/OSS tools and specific project management techniques to increase speed, reproducibility, and effectiveness of scientific software development.

Outline

1. Build & configure tools
2. Version control
3. Project management
4. Interpreted languages
5. Automated testing for research software
6. screen and VNC: neat UNIX tools

A short quiz

Build tools: make, VC++, Xcode, Eclipse?

Configure tools: autoconf/configure, cmake?

Version control: CVS, svn, etc.?

Project management: Trac, DrProject?

Interpreted languages: matlab, Python, Perl?

Testing?

Configure and build tools

- Automate process of configuring software on multiple computers with different options, libraries, etc.
- Build process automation aims to build software with a single command;
- Build software also can *recompile* minimum parts of software when a source file is changed.

CMake

- Tool for automatically finding libraries and configuring a build, for C/C++ software.
- Produces build files compatible with ‘make’, VC++, and Xcode.
- Works on Windows, UNIX, Mac.

(Demo)

Why build tools?

- (Probably don't need convincing...!)
- As soon as a project moves beyond one file, making sure that compilation stays in sync becomes problematic.
- For many more than one file, can be *slow* to recompile everything every time.
- Build tools automate common tasks.

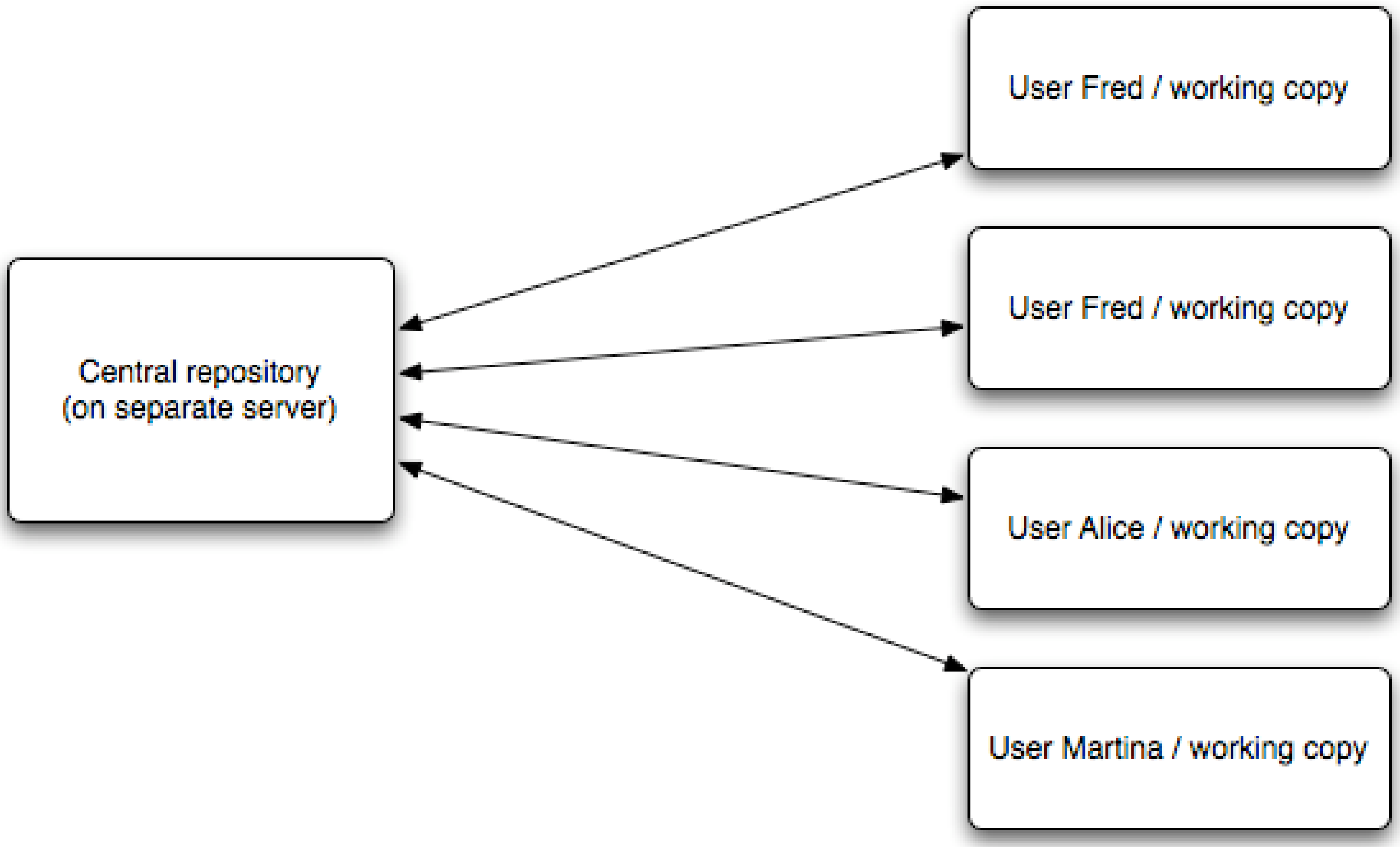
Example: ‘make clean’

- Sometimes you need to start from scratch (stale object file/binaries, new compile options, whatever)
- Figuring out what files to delete can be annoying.
- Automate process => ‘make clean’.

Version control

- Manage “working” vs “in progress” code.
- Coordinate between multiple developers.
- Track entire history of source code.
- Automatic backup to remote machine.
- Determine changes (intended vs unintended…) in latest version.

Version control – one or more users



Example: subversion

Magic commands:

```
svn diff
svn revert
svn status
```

(Demonstration.)

More subversion

Exists for every platform (TortoiseSVN for Windows).

Supports read-only source code export.

Lots of good documentation.

Using a source code control system is critical, even for one-person projects.

Project management

- Manage source code, bugs, feature requests, project documentation...
- A number of open source tools out there; Trac, DrProject recommended. Combines wiki, source browser, ticket system, scheduling.
- “Trac” demo:

Interpreted languages

- “Interpreted” vs “compiled” -- distinction is losing its meaning, but:
  - Compiled: Fortran, C/C++, Java
  - Interpreted: Matlab, Perl, Python

- Interpreted languages are often higher level and always *less formal*.

Python: Eratosthenes' sieve

```
def divides(primes, n):
   for trial in primes:
      if n % trial == 0: return True
   return False

def prime_sieve():
    p, current = [], 1
    while 1:
        current += 1
        if not divides(p, current):
            p.append(current)       # prime!
  return
            yield current
```

A few reasons to use Python

- Easy to learn, with C-like syntax;
- Easily wraps C/C++ code;
- Very good string handling;
- Full built-in library;
- Cross-platform (Windows, Mac, UNIX);
- Fantastic for rapid prototyping;
- Higher-level data types:
  - Dictionaries (hash tables or “maps”)
  - Lists
- Fully object-oriented;
- Interactive interpreter;

A few reasons to use Python

Bold claim:

3-10x productivity increase in software development.

(Software Carpentry online course teaches Python.)

Wrapping C with Python

(For speed and/or interaction with existing code)

Can be done by hand or automatically (SWIG, Boost, SIP)

Lends itself to pipelining (A->B->C), code reuse, testing, and major code reorganizations.

Can dramatically ease parallelization issues.

Examples in “Advanced Software Carpentry” text.

Software testing

What do you think of when you hear "you should test"?

(Manual "smoke tests" -- push buttons, see if smoke emerges?)

Test automation

1. Automate as many tests as possible.

2. Run them frequently.

3. Get a single report.

What do I mean by "testing"?

For research software:

Does my code work?

Can I trust results?

Are *previous results* reproducible as code changes over time?

Types of automated tests

Functional tests: does my code perform specific functions properly?

Regression tests: has the output of my code changed?

(Lots more types, but these are the two types that are most important for research software.)

Functional testing

```
x = 2

# test that square works
y = math.pow(x, 2)
assert y == 4

# test that sqrt works
z = math.sqrt(y)
assert z == 2
```

Functional testing

- “white box” -- asserts things about results of code, inside the code.
- Should be pretty simple.
- Often requires that software be in “known state” -- empty database, specific data files in specific places, random number generator with specific seed set
- Good for complicated/tricky pieces of code that aren’t intellectually challenging.

Regression testing

```
% ./run-program x=5 y=10 z=15 > results.out
% diff results.out results.good
```

(...but completely automated, for several or many different parameter sets; complain if 'diff' is not empty.)

Regression testing

- "black box" - tests exercise compiled code and only need to know input/output.
- Test utilities can be really really simple ("compare these two data files")
- Not very good at pinpointing *why* code is different.
- Incredibly powerful in combination with version control!

Testing summary

Functional and regression tests are two good lines of defense against weird stuff happening in your code.

Functional tests are especially good for slow-changing but annoyingly tricky bits of code (e.g. data file format handling).

Regression tests are excellent for determining when change (wanted or unwanted) occurs.

Getting started is easier than it sounds ;)

Testing rewards

Code becomes more robust.

You are more confident in your results (or, rather, your confidence in your results is more warranted).

Multi-person projects become much easier to coordinate.

Maintenance load (e.g. software/OS/hardware updates) decreases.

screen and VNC

Problem:

Calculations and display can take a long time to finish.

Sometimes you need to be elsewhere (home, conference, ...).

screen and VNC

screen is a free/OSS program for persistent text login sessions on UNIX.

VNC is a free/OSS program for persistent X-window/Mac graphical login sessions on UNIX.

screen Demo

VNC Demo

VNC notes

Secure VNC is more difficult than insecure VNC.

You have to start your session in VNC.

VNC is built into new Mac OS X.

VNC viewers are available for all platforms.

Concluding thoughts

- There are a lot of tools and approaches that can help you do things more efficiently and effectively.
- Automation (configure, build, test) is a great way to free your mind for other problems.
- A little bit of effort can go a long way.
- …it's also true that you can get dramatically sidetracked from your research…

Links etc.

- titus@caltech.edu

- Google "software carpentry" for a free course on specific technologies and approaches.

- Thanks to Greg Wilson, Brandon King, and Diane Trout for Software Carpentry.