An introduction to version control

Author: Lars Kellogg-Stedman <lars@seas.harvard.edu>
IRCS: http://ircs.seas.harvard.edu/
Support: ircshelp@seas.harvard.edu
http://ircs.seas.harvard.edu/userdocs/

This document may be viewed as a slideshow.

Contents

- Who are you?
- Why are we here?
- Why version control?
  - Protection.
- Why version control?
  - Isolation.
- Why version control?
  - Collaboration.
- Vocabulary
- Vocabulary
- Vocabulary
- Vocabulary
  - History of version control
  - History of version control
  - History of version control
  - History of version control
  - git and mercurial
  - Bazaar
  - Darcs
  - Monotone
- Centralized version control
- Centralized version control
- Centralized version control
- Centralized version control
- Distributed version control
- Distributed version control
- Distributed version control
- Distributed version control
- Distributed version control
- Distributed version control
- Centralized vs. Distributed
- svn: Introduction
- svn: Getting help
- svn: Checking out a remote repository
- svn: Adding files
Who are you?

Lars Kellogg-Stedman <lars@seas.harvard.edu>
Senior Technologist, Instructional and Research Computing Services (IRCS)
Why are we here?

- An introduction to version control.
- Some basic recipes for using Subversion.
- Some basic recipes for using git.
- Questions and answers.

Why version control?

Protection.

Version control is like having infinite undo.

- Imagine that you are working on some code. You get it working, but then make a few additional changes. You return to the code later that day to find that you broke it. Version control makes it easy to see what has changed, and possibly revert your code back to a known state.
- You release some code, and a month later someone finds a critical bug. In the intervening month your code has diverged substantially from your previous release, making it difficult to fix what may be a simple problem. Version control makes it easy to return to a previous state of your code in order to fix a problem without losing all of your intervening work.

Why version control?

Isolation.

Version control makes it easy to experiment with your code.

- You can test out experimental features in a dedicated branch of your project without affecting the production code (or the work of your collaborators).

  You can merge your changes into another branch or discard them when you have finished experimenting.

Why version control?

Collaboration.

Version control makes it easier for multiple developers to work on the same project.

- It makes it easier to share code.

  A VCS handles the task of generating patches and merging changes, and makes it easier for a group of people...
• It promotes accountability.

In a large project with many contributors, a version control system keeps track of who has made which changes. If a problem crops up, it's easy to identify the person responsible for the change.

**Vocabulary**

*Repository*
A database containing the files and change history of your project.

*Working tree or Working copy*
A local copy of files from a repository.

**Vocabulary**

*Revision*
The state of the repository at a certain point in time.

*Commit*
To save your changes back to the repository.

*Merge*
To combine two sets of changes to the files in your project.

**Vocabulary**

*Tag*
Identifies a point-in-time snapshot of your project.

*Branch*
An isolated stream of changes to your project.
History of version control

In the beginning (c. 1985), there was RCS.
- uses locking to manage conflicts.

But...
- managed files, not projects.
- everyone worked in the same place.
- locks were inconvenient.

History of version control

RCS begat CVS.
- uses merging and conflict detection to manage conflicts.
- supports distributed operation.

But...
- operations were not atomic
- no support for renaming files/directories

History of version control

CVS begat Subversion.
• designed to address problems in CVS
• command line familiar to CVS users.
• atomic operations, handles directories and renames.

The Subversion documentation describes the development of Subversion thusly:

In early 2000, CollabNet, Inc. (http://www.collab.net) began seeking developers to write a replacement for CVS... CVS's limitations were obvious from the beginning, and CollabNet knew it would eventually have to find something better... So CollabNet determined to write a new version control system from scratch, retaining the basic ideas of CVS, but without the bugs and misfeatures.

**History of version control**

The version control explosion.

- **git**
- Bazaar
- darcs
- Mercurial
- monotone

**git and mercurial**

Git and Mercurial both stem directly from the brouhaha surrounding the adoption -- and subsequent rejection in 2005 -- of the commercial BitKeeper version control system for the Linux kernel.

**Bazaar**

Bazaar was developed in 2005 by Canonical as a replacement for baz, which was itself a fork of GNU arch.

**Darcs**

Darcs was developed in 2002 as a result of the author's experience with Gnu arch.

**Monotone**

Monotone was initially released in 2003. In 2005 it was briefly a candidate for replacing BitKeeper for use in Linux kernel development.
Centralized version control

- CVS, Subversion
- One main repository
- Commits go to central repository

Developers check out working copies.

Someone commits bad code to repository.

Centralized version control
Centralized version control

Changes are visible to everyone.

Distributed version control

Most recent version control systems use a distributed model.

Distributed version control

Developers check out working copies.
Distributed version control

Someone commits bad code to local repository.

Fixes locally and pushes to remote repository.

Everyone is happy.
Distributed version control

There is no spoon.

In the world of distributed version control, the idea of a central repository is a social construct rather than a technical one. While some projects may find it convenient to identify a central repository, git (and other DVC systems) do not enforce a hub and spoke configuration.

For some of my own projects I have something of an "inverted tree": my working copies push to two remote repositories. One is a "personal" repository, which I use to coordinate my work between my office, my laptop, and so forth. The other is a "public" repository, where I push my code when I want others to see it.

Centralized vs. Distributed

It may sound like I am suggesting that distributed version control is generally better than centralized version control.

- I am.
- There are other opinions.

In particular, some of the developers of Subversion have suggested that a distributed model makes it less likely that
people will share code with others (while in a centralized system they are largely forced to if they want to take advantage of the version control system).

**svn: Introduction**

Subversion is a centralized version control system developed to address the shortcomings of CVS.

**svn: Getting help**

- Most commands have built-in documentation you can access with the `--help` option:

  ```
  svn checkout --help
  ```

**svn: Checking out a remote repository**

Use the `svn checkout` command to get a working copy of a Subversion repository:

```
svn checkout URL[@REV]... [PATH]
```

[documentation]

`URL` is often an `http://` URL, but may also be `svn+ssh://` for access over ssh, or `file://` for access to a repository on the local filesystem.

For a local repository:

```
$ svn checkout file://$HOME/repos/hello
```

For a repository hosted on the CIT Subversion server:

```
$ svn checkout \
https://source.seas.harvard.edu/svn/version-control-workshop
```

You can elect to checkout only a particular subtree of a repository. For example, if you really like the images in this presentation:

```
$ svn checkout \
https://source.seas.harvard.edu/svn/version-control-workshop/images
```

**svn: Adding files**

`svn add` schedules individual files or directories in your working copy to be added to the repository next time you commit your working copy:

```
svn add PATH [PATH ...]
```

[documentation]

For example, let's say I wanted to package up the hello project for the Fedora project. I'd need to create a package specification, or spec, file and then add it to the repository:

```
$ svn add hello.spec
```
A         hello.spec
$ svn ci -m 'added spec file' hello.spec
Adding       hello.spec
Transmitting file data.
Committed revision 2.

svn: Renaming files

Use `svn rename` to rename files in the repository:

```
svn rename SRC [...] DST
```

[documentation]

svn: Removing files

Use `svn delete` to remove files from the repository:

```
svn delete PATH [PATH ...]
```

[documentation]

- Removes the file from your working copy and schedules a remove from the repository (next time you commit).
- If you remove the file manually, you will still need to issue `svn delete` after the fact to mark the item deleted in the repository.
- AKA `svn rm`.

svn: What's changed: status

Use `svn status` lists the state of files in your working copy:

```
svn status
```

[documentation]

The `svn status` command shows you what files in your working copy have been changed (added, deleted, renamed, modified). It will also show you which files are not part of your repository. For example:

```
$ svn status
?       subversion-example.rst
M       why_use_vc.rst
M       history_of_vc.rst
M       subversion.rst
A       vocabulary.rst
```

This shows that three files have been modified, one has been added, and one is unknown to the version control system.

svn: What's changed: diffs

Use `svn diff` to display the changes made to your repository:

```
svn diff [\-r N[:M]] [TARGET]
```
• Without any arguments, display all of the uncommitted changes in your working copy.
• Specify TARGET to see just the changes in a particular file or directory.
• Use the -r option to look at the changes between specific revisions.

For example:

```
$ svn diff -r 44:46 README.rst
Index: README.rst
===================================================================
--- README.rst   (revision 44)
+++ README.rst   (revision 46)
@@ -3,6 +3,8 @@
=======================================
 :Author: Lars Kellogg-Stedman <lars@seas.harvard.edu>
 +:Organization: Instructional and Research Computing Services, Harvard
+      School of Engineering and Applied Sciences

 This is a basic introduction to version control with Subversion and Git.
```

### svn: Committing changes

Use `svn commit` to send changes back to the repository:

```
svn commit [PATH ...]
```

Subversion will start an editor allowing you to provide a commit message (you can also provide one with the `-m` option).

### svn: Updating your working copy

Use `svn update` to update your working copy with changes from the repository:

```
svn update
```

You can elect to update only a particular subtree of your repository by passing an optional path argument, e.g:

```
$ svn update images/
```

### svn: Conflicts

A conflict occurs when two people make overlapping changes.

• Detected when you attempt to update your working copy.
• You may discard your changes, discard the repository changes, or attempt to correct things manually.

If you attempt to update your working copy and Subversion detects a conflict, you will get a warning like this:

```
$ svn update
```
Conflict discovered in 'README'.
Select: (p) postpone, (df) diff-full, (e) edit,
(mc) mine-conflict, (tc) theirs-conflict,
(s) show all options: df

Selecting "mine-conflict" will discard the repository changes
(keeping your local changes), while selecting "theirs-conflict" will
discard your local changes.

**svn: Viewing history**

The `svn log` command shows you the history of your repository:

```
svn log [PATH]
```
[documentation]

`svn log` with no arguments will show you the commit messages for each revision in your repository:

```
$ svn log
---------------------------------------------------------------
r4 | lars | 2010-03-18 12:46:35 -0400 (Thu, 18 Mar 2010) | 1 line
    changed GNU to Microsoft
---------------------------------------------------------------
r3 | lars | 2010-03-18 12:46:33 -0400 (Thu, 18 Mar 2010) | 1 line
    made some very important changes
---------------------------------------------------------------
r2 | lars | 2010-03-18 12:46:28 -0400 (Thu, 18 Mar 2010) | 1 line
    import version 2.5
---------------------------------------------------------------
r1 | lars | 2010-03-18 12:46:27 -0400 (Thu, 18 Mar 2010) | 1 line
    create repository layout
---------------------------------------------------------------
```

**svn: Tagging and branching**

- Tags and branches are naming conventions.
- Use `svn copy` to create tags or branches.
- Space efficient via copy-on-write

The conventional layout of a Subversion repository looks like this:

```
/trunk/
/tags/
/branches/
```

You normally work in the `trunk` directory. To create a tag:

```
$ svn copy trunk tags/version_1
$ svn ci -m 'tagged version 1'
```

Or to create a branch:

```
$ svn copy trunk branches/version_1_fixes
$ svn ci -m 'created branch for version 1 fixes'
```
While functionally the same, the convention is that you do not edit things in the tags/ tree, while you may change things in the branches/ tree.

See choosing a repository layout from the Subversion documentation for more information.

**svn: Creating a repository**

Use `svnadmin create` to create a new repository:

```
svnadmin create REPOS_PATH
```

Where `REPOS_PATH` is a local filesystem path.

[documentation]

If you will be accessing the repository remotely (or offering remote access to collaborators), you will need to decide what sort of remote access to provide. Your options are:

- `file://` (local only)
- `svn://` (subversion's native protocol)
- `svn+ssh://` (subversion over ssh)
- `http://` (subversion over webdav)

The Subversion page on choosing a server configuration provides more information about the pros and cons of each of these methods.

E.g:

```
$ svnadmin create ~/repos/hello
```

You could then access this using the URL `file://$HOME/repos/hello`.

**svn: Importing files**

`svn import` imports an entire tree into an existing Subversion repository:

```
svn import [PATH] URL
```

[documentation]

For example, if I wanted to start using Subversion to track changes to the hello project, I might do something like this:

```
$ svn import hello-2.5 https://source.seas.harvard.edu/svn/hello/trunk
```

And the output would look something like this:

```
Adding         gnulib
Adding         gnulib/m4
Adding         gnulib/m4/lib-link.m4
Adding         gnulib/m4/fcntl-o.m4
Adding         gnulib/m4/getopt.m4
Adding         gnulib/m4/localcharset.m4
Adding         gnulib/m4/string_h.m4
Adding         gnulib/m4/mbrtowc.m4
Adding         gnulib/m4/iconv.m4
```
Note that `svn import` does not modify your current directory! You would still need to issue an `svn checkout` to get a working copy of the repository.

**git: Introduction**

Git is a distributed version control system originally developed by Linus Torvalds as a replacement for BitKeeper.

**git: Getting help**

- Most commands have built-in documentation you can access with the `--help` option:
  
  ```
  git init --help
  ```

- Also available via `man`, e.g:
  
  ```
  man git-init
  ```

**git: Creating a repository**

Use `git init` to create a git repository in your current directory:

```
$ git init
```

[documentation]

`git init` creates a git repository (named `.git`) in your current working directory. You will add files to this repository using `git add`. This gives you a repository (the `.git` directory) and a working copy (everything else).

If you are going to start tracking an existing project with git, you will generally start like this:

```
$ git init
Initialized empty Git repository in .../.git/ 
$ git add .
$ git commit -m 'initial import'
```

If you are creating a repository that people will be accessing remotely, you will normally want to create a "bare" repository, which consists of just the contents of the `.git` directory and no working copy. You do this with the `-b` flag:

```
$ git init -b
```

**git: Adding files**

`git add` schedules files to be committed to the repository.

```
$ git add PATH [PATH ...]
```

[documentation]
Unlike Subversion, if you modify a file you (generally) need to `git add` that file in order to make the changes part of the next commit.

Use the `git reset` command to "undo" an add operation:

```
   git reset HEAD
```

This resets the index but leaves your working directory untouched. You can also use `git reset` to revert to a previous commit; read the documentation for more information.

**git: Committing changes**

Use `git commit` to commit files to your local repository:

```
   git commit [-a] [PATH ...]
```

[documentation]

`git commit` by itself will commit any changes scheduled using `git add`. If you would like to commit all locally modified files, use the `-a` option:

```
   git commit -a
```

You may also commit a subset of modified files by specifying paths on the command line:

```
   git commit path/to/modified/file
```

**git: Renaming files**

Use `git mv` to rename files in the repository:

```
   git mv SRC DST
```

[documentation]

Because `git` tracks files by cryptographic checksum, rather than by name, the `git mv` command is not strictly necessary. If you manually rename a file and then do a `git rm file` followed by a `git add file`, `git` will correctly recognize that you have simply renamed it (because the checksum is still the same).

**git: Removing files**

Use `git rm` to remove files from the repository:

```
   git rm PATH [...]
```

[documentation]

**git: What's changed: status**

Use `git status` to see a list of modified files:

```
   git status
```
The output of `git status` will look something like this:

```
$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#   modified:   version-control.rst
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#   examples/
no changes added to commit (use "git add" and/or "git commit -a")
```

The files listed as "changed but not updated" are files that you have modified but not yet added to the repository. "Untracked files" are files that have not previously been added to the repository.

**git: What's changed: diffs**

Use `git diff` to see pending changes in your working copy:

```
  git diff
```

The output of `git diff` is standard diff output, e.g.:

```
$ git diff
diff --git a/version-control.rst b/version-control.rst
index e518192..b1c519a 100644
--- a/version-control.rst
+++ b/version-control.rst
@@ -243,6 +243,34 @@ commit`` to commit them to the (local) repository:``
   Using git: What's changed?
   ===============

   +Use `git status` to see a list of modified files:
   +
   + git status
   +
   +.. container:: handout
   +
   + The output will look something like this:
   +

   You can also use `git diff` to see the changes between arbitrary revisions of your project:

   - Changes in working copy vs. previous commit:
     ```
     git diff <commit>
     ```

   - Changes between two previous commits:
     ```
     git diff <commit1> <commit2>
     ```

**git: Cloning a remote repository**
Use the `git clone` command to check out a working copy of a remote repository:

```
    git clone REPOSITORY [DIRECTORY]
```

`git clone` will clone the remote repository to a new directory in your current directory named after the repository, unless you explicitly provide a name with the `DIRECTORY` argument.

This is analogous to Subversion's `checkout` operation.

You can only clone the top-level repository; unlike Subversion, git does not allow you to clone individual subtrees.

**git: Updating your working copy**

Use `git pull` to update your local repository from the remote repository and merge changes into your working copy:

```
    git pull [REPOSITORY [REFSPEC]]
```

`git pull` by itself will pull changes from the remote repository defined by the `branch.master.remote` config option (which will typically be the repository from which you originally cloned your working copy). If there are multiple remote repositories associated with your working copy, you can specify a repository (and branch) on the command line, e.g., to pull changes from the branch `master` at a remote named `origin`:

```
$ git pull origin master
```

**git: Pushing changes**

Use `git push` to send your committed changes to a remote repository:

```
    git push [REPOSITORY [REFSPEC]]
```

`git push` by itself will push your changes to the remote repository defined by the `branch.master.remote` config option (which will typically be the repository from which you originally cloned your working copy). If there are multiple remote repositories associated with your working copy, you can specify a repository (and branch) on the command line, e.g., to push your changes to branch `master` at a remote named `origin`:

```
$ git push origin master
```

Git doesn't like you pushing into a remote repository that is associated with a working tree (because this could cause unexpected changes for the person who checked out that working tree). You will generally want to create "bare" repositories for remote access (using `git init --bare`).

If you attempt to push to a repository that is newer than your working copy you will see an error similar to the following:

```
$ git push
To dottiness.seas.harvard.edu:repos/myproject
    ! [rejected]        master -> master (non-fast forward)
error: failed to push some refs to 'dottiness.seas.harvard.edu:repos/myproject'
```
To fix this, run `git pull` and deal with any conflicts.

**git: Conflicts**

A conflict occurs when two people make overlapping changes.

- Detected when you attempt to update your working copy via `git pull`.
- You may discard your changes, discard the repository changes, or attempt to correct things manually.

If you attempt to pull in changes that conflict with your working tree, you will see an error similar to the following:

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /Users/lars/projects/version-control-workshop/work/repo2
  4245cb6..84f1112  master     -> origin/master
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.
```

To resolve the conflict manually:

- Edit the conflicting files as necessary.

To discard your changes (and accept the remote repository version):

```
- run `git checkout --theirs README`
```

To override the repository with your changes:

```
- run `git checkout --ours README`
```

When you complete the above tasks:

- add the files with `git add`
- commit the changes with `git commit`.

**git: Viewing history**

The `git log` command shows you the history of your repository:

```
git log [PATH]
```

[documentation](#)

`git log` with no arguments shows you the commit messages for each revision in your repository:

```
$ git log
commit 7c8c3e71893d7481fdd9c13ec8f53cb9c61fac50
Author: Lars Kellogg-Stedman <lars@seas.harvard.edu>
Date:   Thu Mar 18 12:46:46 2010 -0400

    changed GNU to Microsoft

commits 257f2f3ff44c2165c1182d3673a825fcadf121aa
```
git: Tagging and branching

- Git has explicit support for tagging and branching.
- `git tag` manipulates tags
- `git branch` and `git checkout` manipulate branches

**git: Tags**

Create a tag:

```
git tag [-a] TAGNAME
```

- Creates a *lightweight* tag (an alias for a commit object)
- Add `-a` to create an annotated tag (i.e., with an associated message)
- Also possible to create cryptographically signed tags

[documentation]

**git: Tags**

List tags:

```
git tag
```

Information about a specific tag:

```
git tag -v TAGNAME
```

**git: Branches**

List branches:

```
git branch
```

Create a branch rooted at *START*:

```
git branch BRANCHNAME [START]
```
If you omit \textit{START}, the branch is rooted at your current HEAD.

**git: Branches**

Switch to a branch:

```
git checkout BRANCHNAME
```

Create a branch rooted at \textit{START} and switch to it:

```
git checkout -b BRANCHNAME [START]
```

For example, you want to enhance your code with some awesome experimental code. You create a new \texttt{seas-workshop-dev} branch and switch to it:

```
$ git checkout -b seas-workshop-dev
```

You make some changes, and when things are working you commit your branch:

```
$ git commit -m 'made some awesome changes' -a
```

And then merge it into the master branch:

```
$ git checkout master
$ git merge seas-workshop-dev
Updating 1288ed3..33e4a4c
Fast-forward
version-control.rst |    2 ++
1 files changed, 2 insertions(+), 0 deletions(-)
```

**git: the index**

Git is not really just like Subversion (or most other version control solutions).

- The \textit{index} is a staging area between your working copy and your local repository.
- `git add` adds files to the index
- `git commit` commits files from the index to the repository.

**git: the index**

- `git diff` is the difference between your working copy and the index.
- `git diff HEAD` is the difference between your working copy and the local repository.
- `git diff --cached` is the difference between the index and the local repository.

**git: the index**

Refer back to this illustration if you get confused:
**git: Plays well with others**

Git can integrate with other version control systems.

- Can act as a Subversion client (may be the only Subversion client you ever need).
- Can import a CVS repository.

**git: Integrating w/ Subversion**

You can use git as your Subversion client. This gives you many of the benefits of a DVCS while still interacting with a Subversion repository.

**git: Integrating w/ Subversion**

Cloning a remote repository:

```
    git svn clone [ -s ] REPO_URL
```

The `-s` flag informs git that your Subversion repository uses the recommended repository layout (i.e., that the top level of your repository contains `trunk/`, `tags/`, and `branches/` directories). The `HEAD` of your working copy will track the trunk.

This instructs git to clone the *entire* repository, including the complete revision history. This may take a while for repositories with a long history. You can use the `-r` option to request a partial history. From the man page:

```
    -r <ARG>, --revision <ARG>
    Used with the fetch command.

    This allows revision ranges for partial/cauterized history to be
    supported.  $NUMBER, $NUMBER1:$NUMBER2 (numeric ranges),
    $NUMBER:HEAD, and BASE:$NUMBER are all supported.

    This can allow you to make partial mirrors when running fetch; but
    is generally not recommended because history will be skipped and
```
git: Integrating w/ Subversion

Committing your changes back to the Subversion repository:

    git svn dcommit

Before you push your changes to the Subversion repository you need to first commit any pending modifications to your local repository. Otherwise, git will complain:

    $ git svn dcommit
    Cannot dcommit with a dirty index. Commit your changes first, or stash them with `git stash'.
    at /usr/libexec/git-core/git-svn line 491

To fix this, commit your changes:

    $ git commit -m 'a meaningful commit message' -a

And then send your changes to the Subversion repository:

    $ git svn dcommit
    Committing to https://source.seas.harvard.edu/svn/version-control-workshop/trunk ...
    M  seealso.rst
    Committed r38
    M  seealso.rst
    r38 = 03254f2c0b3d5e068a87566caef8445458b85b0 (refs/remotes/trunk)
    No changes between current HEAD and refs/remotes/trunk
    Resetting to the latest refs/remotes/trunk
    Unstaged changes after reset:
    M  git.rst
    Committed r39
    M  git.rst
    r39 = d1f884a3f945f6083541e28ab7a09ca8efc6343b (refs/remotes/trunk)
    No changes between current HEAD and refs/remotes/trunk
    Resetting to the latest refs/remotes/trunk

git: Integrating w/ Subversion

Updating your working copy from the Subversion repository:

    git svn rebase

As with git svn dcommit, you must have a clean working copy before running the rebase command.

git: Integrating w/ CVS

You can import a CVS repository into git (this is a one-time, one-way operation).

The CVS import feature requires cvsps, a tool for collating CVS changes into changesets.

git: Integrating w/ CVS

This may take a while:
export CVSHOME=:pserver:anonymous@example.com
cvs login
git cvsimport -o cvs_head -C my-project

git: Frontends

The git wiki has a list of frontends for git.

More information

We have compiled a list of links related to Git and to version control in general:

- [http://delicious.com/seas_ircs/versioncontrol](http://delicious.com/seas_ircs/versioncontrol)

If you are looking explicitly for additional tutorials:

- [http://delicious.com/seas_ircs/versioncontrol+tutorial](http://delicious.com/seas_ircs/versioncontrol+tutorial)

Metadata

This presentation is available online:


Metadata

The presentation sources are also available online:

- via Subversion:

        svn co \
        https://source.seas.harvard.edu/svn/version-control-workshop/trunk \
        version-control-workshop

Metadata

The presentation sources are also available online:

- via Git:

        git svn clone -s \
        https://source.seas.harvard.edu/svn/version-control-workshop