Visuo - Revision Management Documentation

By: Jonathan Gagné

Ver. 1.0 (Aug, 2012)

Revision management consists of the procedures, policies, and best practices that govern how the developers manage the project’s source code revisions. To facilitate the revision management, the distributed version control system Mercurial is used.

The Definitions Section contains some definitions of some of the terms in this document. The Revision Management Workflow Section contains the policies and Mercurial commands used. Finally, the Best Practices Section details the suggested practices when doing source code revision in Mercurial. Following the sections in this document will provide efficient use of the revision system and will aide in developer collaboration.

# Definitions

Changeset – A particular change made to the Mercurial repository when source code is committed

Kiln – A Mercurial repository web service

Local (Repository) – Any local repository at OPIN or UNBC that pushes to DEV

Remote (Repository) – The Kiln repository (this can be changed to any repository if required)

Repo – A repository

# Revision Management Workflow

Subsection 1 details the initial setup of the repositories. Following completion of the initial setup, the workflow begins a repeating cycle as specified in Subsections 2 through 5, where each alphabetically-bulleted step is listed in order of execution.

Subsection 6 describes how to go back to previous revisions, and Subsection 7 details how to use named branches. Finally, Subsection 8 provides useful commands to list and view the differences between changesets, and contains the command to change the working directory to a particular revision.

## Remote Repository Setup (Only initial user – i.e. Jon G)

1. Set up Kiln account
2. Create remote repository Kiln
3. (Automatic Setup) Download Kiln Client at <https://visuo.kilnhg.com/Tools>  
   **OR**   
   (Manual Setup)   
   - Download Mecurial (with TortoiseHg recommended) <http://mercurial.selenic.com/wiki/Download>  
   - Setup Mercurial as described in the Appendix.
4. Add files to local repository

$ hg add

1. Commit changes

$ hg commit

1. Push to remote (see the appendix if a connection to Kiln has not already been set up or just specify the URL after push).

$ hg push

## Local Repository Setup (All other users of SOIL)

1. (Automatic Setup) Download Kiln Client at <https://visuo.kilnhg.com/Tools>  
   **OR**   
   (Manual Setup)   
   - Download Mecurial (with TortoiseHg recommended) <http://mercurial.selenic.com/wiki/Download>  
   - Setup Mercurial as described in the Appendix.
2. Change directories to your web root directory, and clone the remote repository with Mercurial.

Get the URL from Kiln and the username is your email

$ hg clone https://visuo.kilnhg.com/Code/Project-Visuo/Group/Visuo

1. Add source code files to local repository (e.g. "hg add –I \*.py")

$ hg add # add all files

$ hg add –I <pattern> # add files matching <pattern>

$ hg add –X <pattern> # add all files except those matching <patter>

1. Commit changes

$ hg commit –m "<commit description>"

1. Push to remote (see the appendix if a connection to Kiln has not already been set up).

$ hg push

## Update Local Repository

1. Before changes files, look if there is a more recent version (optional)

$ hg incoming

1. Pull remote changes not on local (if any)

$ hg pull

1. Update file system to most recent revision

$ hg update

## Modify Local Repository

1. Create new branch, if appropriate (see Sect. 6.a)
2. Make changes to project file(s)
3. Test changes and modify if necessary
4. Add new files to repo (if any were created)

$ hg add

1. Commit Changes

$ hg commit -m “<insert message>”

1. If the group of changes for a particular branch are competed and fully tested, merge branch

(see Sect. 6.c)

## Update Remote Repository

1. Look at changes on local that are not yet on remote (optional)

$ hg outgoing

1. Push changes (to remote)

$ hg push

1. If no error, then Repeat at Subsection 2 (i.e. GOTO Step 2.a).

Otherwise, if you receive the error:

“Abort: push creates new remote heads!

(did you forget to merge? Use push -f to force)”

Then do NOT use the -f. Instead, merge remote into local

## Merge Remote Repository into Local Repository

1. Look at the remote changes that are not yet on the local (optional)

$ hg incoming

1. Pull the changes, creating two heads (one for the local changes and one for the remote changes)

$ hg pull

1. Merge the changes

$ hg merge

* In the event of a merge conflict, a use a conflict resolution tool, such as KDiff3, to resolve the conflict before merging.

1. Commit the changes
2. Push changes
3. Repeat at Subsection 2 (i.e. GOTO Step 2.a)

## Undo Changes

In nearly every project, mistakes will be made. Therefore, it is useful to revert to a previous changeset (commit).

1. To revert an uncommited file to the most recent commit

$ hg revert <filename>

Since it is against the design policy of Mercurial to delete anything, the revert creates a files called <filename>.orig. To remove this file, simply delete the old and rename the new to the old

$ rm <filename>

$ mv <filename>.orig <filename>

1. To go back to the last commit

$ hg rollback

## Branches

This section is negotiable; however, to avoid confusion, the type of branching should be used consistently among developers. We suggest using named branches for adding features and other major changes, and using anonymous branches for quick fixes, although if preferred, the use of other branching types (branches or bookmarks[[1]](#footnote-1)) can also be used if agreed upon by the developers. The use of subrepositories for modules is a potentially useful feature, although due to its added complexity, the suitability of this feature over branches will be evaluated at a future time.

Branches should be used extensively. Before adding any major features or modules, create a new branch.

1. To create a new named branch

$ hg branch <branch-name>

$ hg commit –m “new module i18n”

1. To select other branches ([-C] clears uncommitted changes)

$ hg update [-C] <branch-name>

1. To merge branches: switch to the branch you will merge to, merge the other branch, and commit, i.e.

$ hg update <branch-merge-into>

$ hg merge <branch-merge-from>

$ hg commit –m “Merge in the i18n module”

## List, View Differences and Switch Changesets

1. To view a list of the repo’s changesets

$ hg log

1. To see the difference between the working directory and a previous changeset

$ hg diff [-r <revision>] [<file>]

* If <file> is specified, then only that file is compared; otherwise, the all files in the project are compared.
* If the –r flag is specified, the current file(s) are compared with the specified <revision>; otherwise, the last changeset is used.

1. Update working directory to (or switch revisions)

$ hg update [-C] [-r <revision>]

* If the –C flag is specified, the changed files since last commit are removed; otherwise, the files are kept and merged into the files from the previous revision.
* If the –r flag is specified, the working directory is switched to the revision; otherwise, the working directory is updated to the head of the branch.

# Best Practices

## Commit Related Changes

A commit should be a wrapper for related changes. For example, fixing two different bugs should produce two separate commits. Small commits make it easier for other developers to understand the changes and roll them back if something went wrong.

## Commit Often

Committing often keeps your commits small and, again, helps you commit only related changes. Moreover, it allows you to share your code more frequently with others. That way it‘s easier for everyone to integrate changes regularly and avoid having merge conflicts. Having few large commits and sharing them rarely, in contrast, makes it hard to solve conflicts.

## Don‘t Commit Half-Done Work

You should only commit code when it is completed. This does not mean you have to complete an entire large feature before committing. Quite the contrary: split the feature‘s implementation into logical chunks and remember to commit early and often. However, don‘t commit just to have something in the repository before leaving the office at the end of the day. If you are tempted to commit just because you need a clean working copy (to check out a branch, pull in changes, etc.) consider using Mercurial shelve feature instead (requires the ShelveExtension or the AtticExtension).

## Test Code Before You Commit

Resist the temptation to commit something that you «think» is completed. Test it thoroughly to make sure it really is completed and has no side effects (as far as one can tell). While committing questionably functional code in your local repository will only interfere with your own work, having your code tested is even more important when it comes to pushing/sharing your code with others.

## Write Good Commit Messages

Provide specific and detailed commit messages about what exactly was fixed. Refrain from using messages that only you can understand.

## Use Branches

Branches an excellent tool to help you avoid mixing up different lines of development. You should use branches extensively in your development workflows: for new features, bug fixes, ideas, etc.

## Tag Major Changesets

Tag major or important changesets for easy reference.

E.g.,

$ hg tag Version-1.0

Be sure to add the .hgtags file to the repo (if not already done) and when a merge conflict arises, keep *all* lines in both versions of the .hgtags file.

## Don’t Use the Changeset Number when Interacting with Other People/Repos

The changeset number identifies a particular changeset in the Mercurial log (hg log); however, it will likely be different between repos. Therefore, it is useful to use the changeset number when working on a local repo but when working with other people or remote repos, reference the changeset using either the tag id or the changeset hex id.

## Follow the Agreed Workflow

Although there may be times when you implementation requires a procedure more complex than is described in the workflow, using the workflow as a minimum set of procedures will facilitate efficient collaboration.

# Appendix – Setting up a Connection to Kiln

This section is only required if a connection with Kiln has not already been established with a command such as "hg clone <url>" or if you would like your password to be automatically sent when pushing to a remote repository, such as Kiln.

Create or edit the user configuration file (handles ALL repositories) in the user home directory:

In Windows, you can find out your home directory by typing in echo %UserProfile% into a command prompt.

Win7 -> C:\Users\<username>\Mercurial.ini

Linux -> ~/.hgrc

# Mercurial configuration file.  
[auth]  
kiln.prefix = https://visuo.kilnhg.com   
kiln.username = Jonathan Gagne  
kiln.password = timewarp  
[ui]  
username = Jonathan Gagne <jongagne@hotmail.com>

Repository configuration file (one per repository) in “<repo>/.hg/hgrc" :

[paths]  
default = https://visuo.kilnhg.com/Code/Project-Visuo/Group/Visuo  
default-push = https://visuo.kilnhg.com/Code/Project-Visuo/Group/Visuo

With this configuration, "hg push" can be used at the command line and it will handle the push without any additional prompts or required parameters.

1. For a comparison of the pros & cons of using the different types of branches, see

   <http://stevelosh.com/blog/2009/08/a-guide-to-branching-in-mercurial/> [↑](#footnote-ref-1)