OGRE Developer Guide

Steve Streeting <sinbad@ogre3d.org>

Revision History

Revision 1.0	11 March 2010	SS

Table of Contents

This document sets out the principles under which OGRE development will be undertaken, and is designed to be used as an introduction for new developers, and a reference for existing developers.

Note

While this document is primarily aimed at core developers and other contributors to OGRE, regular users might find it interesting too, particularly if they aspire to contribute code eventually. Currently this guide concentrates on Mercurial to help our transition to it, but can be updated to include more later on.

1. Recommended Mercurial Configuration

OGRE uses Mercurial for source control, this is a Distributed Version Control System (DVCS), which means that while there is a central repository which is considered *authorative*, in fact everyone is able to clone their own copy of the repository and commit to it themselves too, with changes being exchanged by *pushing* and *pulling* changesets (commits) between repository clones.

For an introduction to Mercurial, please read the official Mercurial tutorials [http://mercurial.selenic.com/wiki/Tutorial] or Joel Spolsky’s tutorials [http://hginit.com]. However, please note particularly in the case of the latter that the OGRE team’s recommeded approach to using Mercurial differs slightly, with respect to the following cases:

- How to resolve the case where another user pushed their changes to the master before you

- How stable version branches are handled

So please keep that in mind. This document will not teach you the basics of using Mercurial since other documents do that already, but we will cover recommended configurations for OGRE and accepted approaches to common issues.

There are some common settings that you should have set in your Mercurial configuration, which can be made globally in ~/.hgrc on Linux and Mac OS X, or C:\Documents and Settings'User'\mercurial.ini on Windows.

1.1. UI Section

1.1.1. User name

At the very minimum, you need to define your user name and email address, which identifies you:

```
[ui]
username = Joe Bloggs <joe@bloggs.com>
```

1.1.2. Ignore List

Also in this section, you're likely to want to define a global ignore list, so that you don't have to configure ignored files per repository. For example:

```
[ui]
username = Joe Bloggs <joe@bloggs.com>
ignore=c:\hgignore_global.txt
```

So here (on Windows) I'm referencing an ignore list file I've created at c:\hgignore_global.txt. A pretty good start for a global ignore list might look like this:

```
syntax: glob

*~
*.orig
*.rej
*.swp
.#*
*.obj
*.o
*.a
*.ncb
*.ilk
*.exe
*.dll
*.lib
*.manifest
*.pdb
*.idb
*.rsp
*.pch
*.dep
*.so
*.dylib
```

```
*.framework
```

You can obviously add more elements to that if you wish.

1.2. Extensions

We recommend enabling a few Mercurial extensions for OGRE development; this is done quite simply in the configuration file.

1.2.1. Common Extensions

Your configuration file should include at least the following:

```
[extensions]
progress=
rebase=
transplant=
```

The Progress extension just gives you more feedback on the progress of operations.

The Rebase extension allows you re-apply your commits on top of changes in the master repository, when someone else has pushed their changes to the master before you have. This avoids having to explicitly merge your changes with theirs, keeping a messy history; it effectively looks as if you had applied your changes on top of their in the first place. See Pulling Changesets.

The Transplant extension allows you to pull individual commits from one branch to another, should you need to. Typically changes are merged between branches but occasionally there are cases where you need to transplant individual ones.

1.2.2. Diff settings

Mercurial can generate patches in 2 different forms:- the default patch format is compatible with the GNU patch utilities and any other tools that understand them, and is therefore very portable; however it cannot deal with binary files, and can’t cope with renames. To resolve this, Mercurial also supports the Git patch format, which embeds more information and can also handle binaries. This is the **recommended** patch format for OGRE since it is far more functional, although it means you have to use Mercurial itself to apply the patches.

```
[diff]
git = True
```

1.3. Windows Specific

1.3.1. Text Handling

Because Windows handles line endings differently to Linux and OS X, you will need some extra entries in your configuration:

```
# inside [extensions]
hgext.win32text=
[encode]
** = cleverencode:
[decode]
** = cleverdecode:
[hooks]
# Reject commits which would introduce windows-style text files
pretxncommit.crlf = python:hgext.win32text.forbidcrlf
```

This is *very important* - by default Mercurial performs no line-ending conversion. By enabling the win32text extension, you cause all text files to be converted to Unix line endings before commit, and converted to Windows line endings when your working copy is updated. This keeps everything consistent.

The cleverencode and cleverdecode entries determine whether a file is text by looking to see if there are any NUL characters in the file; if there are none, it is assumed to be text, if there are one or more NULs then it’s considered to be binary.

The hook defined at the bottom is included to prevent accidental committing of files with remaining Windows line feeds.

1.3.2. TortoiseHg Specific

If you’re using TortoiseHg [http://tortoisehg.bitbucket.org/], you definitely want to set the following option:

```
[tortoisehg]
postpull = rebase
```

This sets the default action after pulling changesets to rebase your own outstanding changesets on top of it. You can also find it in the Global Options under Synchronize > After Pull Operation.

1.4. Mac OS X Specific

1.4.1. Using FileMerge

Apple’s included FileMerge tool is an extremely powerful 3-way merge tool that you really want to be using for visual diffs and manual merges. It can be launched from the command-line using *opendiff*, but unfortunately if you call it directly it doesn’t block further execution in the console which causes problems for Mercurial. So, the first thing you need to do is create a wrapper script called "opendiff-w" in /usr/bin (or somewhere else on your path), with this content:

```
#!/bin/sh
opendiff "$@" | cat
```

Make sure you chmod +x this file. Then, make the following changes in your ~/.hgrc file:

```
[extensions]
hgext.extdiff =

[extdiff]
cmd.opendiff = opendiff-w

[merge-tools]
```

```
filemerge.executable = opendiff-w
filemerge.args = $local $other -ancestor $base -merge $output
```

The first entry makes the *opendiff* command available so you can call *hg opendiff* instead of *hg diff* and it will open up FileMerge instead of showing you a unified diff on the command line. The merge-tools section makes opendiff the default when merging.

2. Using Mercurial

Hopefully you're familiar with the basic principles of using Mercurial already; if not, please revisit the the official Mercurial tutorials [http://mercurial.selenic.com/wiki/Tutorial]. However, Mercurial is very flexible so some ground rules need to be established in order for the team to stay organised.

2.1. General Points

2.1.1. Master Repository

The '*master*' repository is located on BitBucket here: http://bitbucket.org/sinbad/ogre/ . I recommend that developers use the SSH link (with SSH keys if you like) rather than https, because it's more reliable when doing large transactions because the timeout is far larger.

2.1.2. Official Branches

The master repository will use branches *only* for official development streams, which in practice means one branch per major stable version, plus the default branch (which is considered "unstable"). For example, this might be the list of branches in the master repository:

```
default
v1-7
v1-6
```

Note

When we were using Subversion, we used branches for other purposes too such as experimental work by a core team member which was uncertain for official inclusion, or a student project on Google Summer of Code - we will no longer use branches for these things. Instead all other divergences in the code will be handled by making *clones* of the repository and not by branches. If these clones need to be public, then a fork should be created on BitBucket, and all those interested can use that. Later on if the work in these forks is to be reintegrated into the master repository, the changes can be either submitted as a patch or (preferably) the changes pulled across with the full commit history, if that makes sense.

Tip

If you have some long-running experimental changes in your own private repository, do them in a local clone to keep them separate from changes you will want to push upstream to the master. This is similar Git's local branches (which are not pushed unless you ask), and in fact there is a non-official Mercurial extension to support this, but I do not recommend it. A clone is simple to maintain and you’ll want to be testing it anyway so having a separate working copy is useful.

2.2. Committing

2.2.1. General Principles

Testing
Mercurial allows you to commit locally, which means you have a faster workflow. However, as a core developer who will be pushing changes to the master, you should still try to ensure that all your commits are consistent in themselves, that is that every point in the history of the repository builds and runs. It *is* possible to re-shuffle the commits you’ve made later using Mercurial Queues, *but only* if you haven’t pushed the commits anywhere else beforehand. Usually it’s better to try to keep your commits consistent all the time. So, **do** commit often, because a rich history is always best, but do at least check that everything compiles and runs before creating a new commit.

Cohesive commits
Your commits should be as focussed as possible on a single subject. Try not to commit changes for multiple purposes at once, it makes it much more difficult to pick apart those changes later if needed. And most importantly, never *ever* commit a bugfix and a feature change in one commit, unless changing the feature was the only way to fix the bug (in which case it can only be fixed in the unstable default branch).

Unnecessary changes
Although it can be tempting to "tidy" code, fix indenting, standardise etc, you should generally avoid doing this for its own sake. If you’re changing the code anyway, by all means fix any formatting and standardisation issues in the area in question while you’re at it. But purely aesthetic changes should never be committed on their own, because all of them can potentially make merging *real* changes - across branches and accepting patches - far more difficult because of conflicts caused simply by trivial formatting changes.

2.2.2. Committing Bugfixes

Bugfixes should always be committed to the *current stable branch* (assuming they affect it) first. So for example at the time of writing bugfixes should be committed to the v1-7 branch. These bugfixes will be merged forwards into the default branch periodically. The reasons for committing to the stable branch first and merging forward, as opposed to committing to the default branch and picking commits to transplant back to the stable branch, are many:

- It encourages primary testing and early committing on the stable branch, which is where fixes are needed most urgently anyway.
- Merge conflicts always happen in the unstable branch, not the stable branch. This is better again because the stable branch is the most important to keep as clean as possible
- Merging is automatic. You always want to merge ALL changes from the stable branch into the default branch, there is no human error involved in selecting what changes to merge. If you cherry-pick fixes to port to the stable branch, you have to make sure you don’t miss anything, make sure you don’t accidentally include an interface change, etc. And if someone has committed a combined fix and an API change in one commit (see below) it’s a mess.

2.2.3. Committing New Features and Breaking Changes

Any new feature, or a change which changes either the API or the behaviour within the existing API, must always be committed to the default branch (often referred to as *unstable*).

2.3. Pushing / Pulling

Pushing and pulling is how you get commits to and from remote repositories, respectively. This is mostly covered in the Mercurial tutorials & documentation. This section is concerned with the recommended approach to common issues.

2.3.1. Pushing Changesets

This is very easy to do, you can use *hg outgoing* and *hg push* (or the GUI equivalent) to send the changesets (commits) you've already made to the master.

Tip

By default the target of push and pull is where you cloned the repository from. If you make a **local clone** of your repository, remember that the default push/pull location will be the local origin, not the master. Edit .hg/hgrc to change where the default points to (or to add aliases).

However, a common occurrence is that you try to push your commits to the master, but find that you get an error saying:

```
abort: push creates new remote heads!
(did you forget to merge? use push -f to force)
```

This is a bit of a confusing message, but the most common reason is that someone else pushed their commits before you did, and you're not allowed to push your changes until you've resolved this. The *new remote heads* refers to the fact that you would effectively create 2 different parallel streams of development on the branch in question, and you can't (or rather, shouldn't) push that, although you might do it locally.

The advice "(did you forget to merge? use push -f to force)" is not a great tip either. You almost never want to use push -f, and actually in the most common case of being out of date with upstream changes, there's a better option than merging - see Section 2.3.2, “Pulling Changesets” below for details.

If you've actually created multiple heads for a branch locally - which you may have done deliberately by committing 2 sets of parallel changes from the same base, or because you've pulled or imported someone elses changes which were based on an earlier revision, your choices are how to unify those before pushing. See Parallel Development for more details.

2.3.2. Pulling Changesets

Pulling changesets from the master (usually the default) or other repositories is easy to do (*hg incoming* to preview, *hg pull* to do it), but it's important to understand the effects of parallel development.

For now, let's just say that you always want to use the rebase extension when you pull from the master:

```
hg pull --rebase
```

If you forget to use --rebase, it's no big deal because you can rebase later too, but definitely try to remember to use it because it's simpler. In TortoiseHg you can set is as the default option, see TortoiseHg config options for details.

Note

There are special occasions where you would not want to use --rebase (or the GUI option equivalent), and this is mainly when pulling changes from very large, long-running forks of the project for re-integration (for example, Google Summer of Code or other long-running experimental forks). These are the minority case so it's still worth setting rebase as your default behaviour, but just bear in mind that very occasionally you might want to change this.

For more on rebasing and merging, see Parallel Development.

2.4. Dealing With Patches

Although you can pull changes directly from other people's repositories too, and this may the best way to handle very large external contributions such as reintegrating forks, generally external contributions are still handled via patches. There's a difference between patches in Mercurial and patches for Subversion or CVS though, because patches in centralised systems are always a one-shot deal of a working copy compared against a single upstream version. In Mercurial, because contributors can commit locally, patches are actually exported changesets, with a full commit history. This fine grained change information can make patches easier to integrate, but it also comes with some added considerations. We'll talk about generating and consuming patches in this section.

2.4.1. Generating Patches

2.4.1.1. Exporting Changesets

This is the recommended approach to generating patches. As mentioned above, in Mercurial you can generate patches from your local commits. This is quite simple:

```
hg export --git -o %b_patch%nof%N_%h.patch REVS
```

Replace REVS with a list of revisions, or a range such as REV1:REV2. It will create one file per changeset named something like ogre_patch1of2_f0c47360a828.diff. You should zip them up and send them to the patch list. If you've configured Mercurial the way we recommended in the Mercurial configuration section, the --git is unecessary because it's the default, but it's included for completeness.

2.4.1.2. Other Ways

There are alternative ways to generate patches, but they're not recommended unless you know what you're doing.

Piped output

It’s possible to just pipe the output from "hg export" to a single file for multiple changesets, instead of saving each changeset to its own file. However with a single file, a failure to import one of the changesets will cause the entire file to fail, so it’s more fragile when importing. Separate patch files are easier to process.

Working copy diff

You can also generate patches from your working copy if you want, using "hg diff" but these patches lack context so are not a good substitute for exporting actual commits which specify their parents. You might want to use this to post something quickly on a forum as a test, or to exchange work-in-progress information, but not for submitting patches.

PatchBomb extension

The PatchBomb extension can be used to email many patches at once, but its reliance on email (and thus mailing lists) makes it not that useful to us, since we use a dedicated patch list.

Mercurial Queues (MQ)

MQ allows vastly more complex handling of patches, particularly for those wanting to keep track of non-core alterations and to consolidate sets of commits into a smaller list for submission upstream. You can read about them in the Mercurial Manual [http://hgbook.red-bean.com/read/managing-change-with-mercurial-queues.html]. We won’t talk about MQ here because it isn’t needed unless your requirements are fairly complex.

2.4.2. Applying Patches

Applying patches is also fairly simple in itself, but there are a couple of nuances. Here’s the standard command:

```
hg import PATCH_FILES...
```

It doesn’t matter whether the patch file is in GNU style or Git style, it will work either way. But it’s important to understand what this does - each patch file (usually) contains a single commit, and those commits are applied on top of your current repository, on to your current branch. Unlike a patch for Subversion, which applies to your working copy which you then need to commit, this actually transfers commits directly, including the original author names & emails.

Note

Well, at least it tries to preserve the author & commit message, if the patch was generated from a commit using hg export. If alternatively the patch was generated from hg diff in a working copy, or the header was chopped off by someone before submission, then actually hg import will bring up your editor and asks you to provide a new commit message (and will commit it under your own name).

Tip

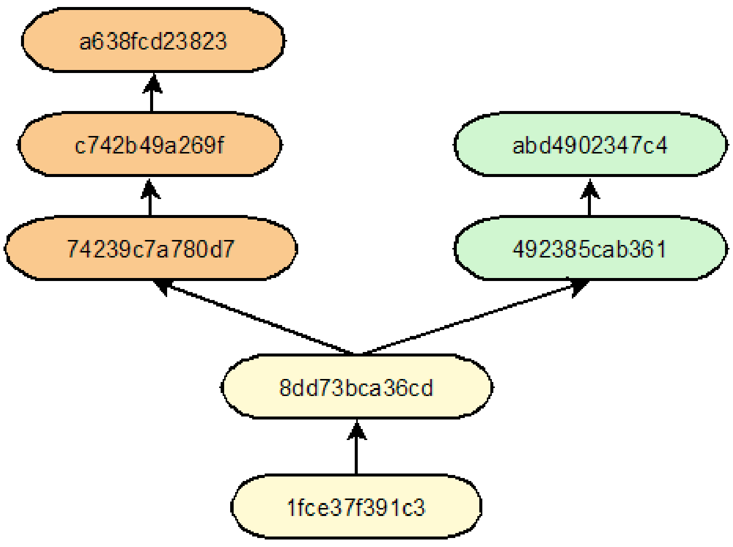
If you would prefer to just import the patch into your working copy rather than as a commit, use "hg import --no-commit PATCH_FILES…"

Because it tries to apply the commits in the patch to the current base, failures are possible if conflicting changes have occurred. In this case, hg import just aborts. A good way to deal with this is to use the --exact parameter, which then applies the changes to the revision that is listed as the base in the patch file (which should therefore always succeed unless the patch is broken). This will create one or more new heads on the branch in question which you should then merge or rebase into your own branch, resolving the conflicts as appropriate. See Parallel Development for more information.

2.5. Parallel Development

2.5.1. Deviations in development within a branch

In centralised systems like Subversion, the only way to record a commit is at the master repository, so all parallel development must be resolved into one stream (per branch) at commit time. Mercurial lets everyone commit locally, so committed deviations can build up in the distributed repositories, and have to be resolved differently. It’s actually possible in Mercurial to have more than one "HEAD" revision on a single branch in one repository, although by default you’re not allowed to push that ambiguitiy to anyone else. This may occur because you have pulled changesets into your repository when you have other changes of your own on that branch, or that you’ve created deviating paths locally (which you can do by manually updating to a previous revision, then committing from that point, creating a split in the branch:

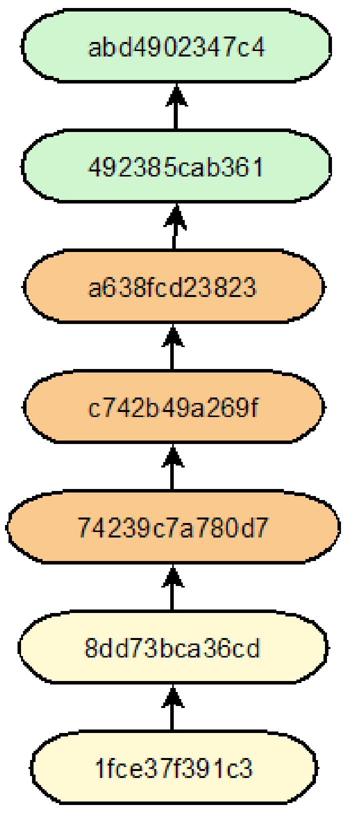


This situation could have been caused by making your own changes (green), and pulling someone elses changes from the master (orange) which were based on the same ancestor. Or, maybe the orange revisions came from a patch applied as "hg import --exact", or pulling changes from a fork.

2.5.2. Resolving multiple heads: Rebase

The most common case of deviations like this is simple, short-term parallel development within the team (for more long-term deviations, see the next section). The HG Init [http://hginit.com] tutorials suggest resolving this by merging the changes with your own (covered next), which is indeed the standard no-extension way to handle this, but it has one major drawback - every time it happens, in the history you have a permanent record of this divergence in the form of a split and merge of the 2 developers changes. Because this is literally an everyday occurrence of short-term deviation, this is extremely distracting when trying to decipher the history, and therefore not recommended.

Instead, by using the Rebase extension, after pulling someone elses changes you *rebase* your own commits so that they are re-applied on top of the head you pulled from the master, essentially flattening the history and looking like there was never any deviation, like this:



You can do this during the pull by appending the --rebase command (or selecting it in TortoiseHg):

```
hg pull --rebase
```

However if you’ve already got the divergence in your repository, you can rebase specifically:

```
hg rebase --continue
```

The "--continue" means that any conflicts will be resolved in your working copy, which will then need committing.

Rebase is usually automatic in the common case where the divergence has been caused by pulling from the master, but if they’ve occurred for another reason (e.g. hg import --exact) then you may need to explictly specify the rebase options. For example, in the example image, the specific command for performing the rebase is:

```
hg rebase -s492385cab361 -da638fcd23823 --continue
```

You could use the Mercurial short revision numbers instead of the "short" hashes there, but the principle is that you’re moving the base of the green revisions to the top of the orange ones. You very rarely need to use this explicit form, but that’s how you do it if you need to.

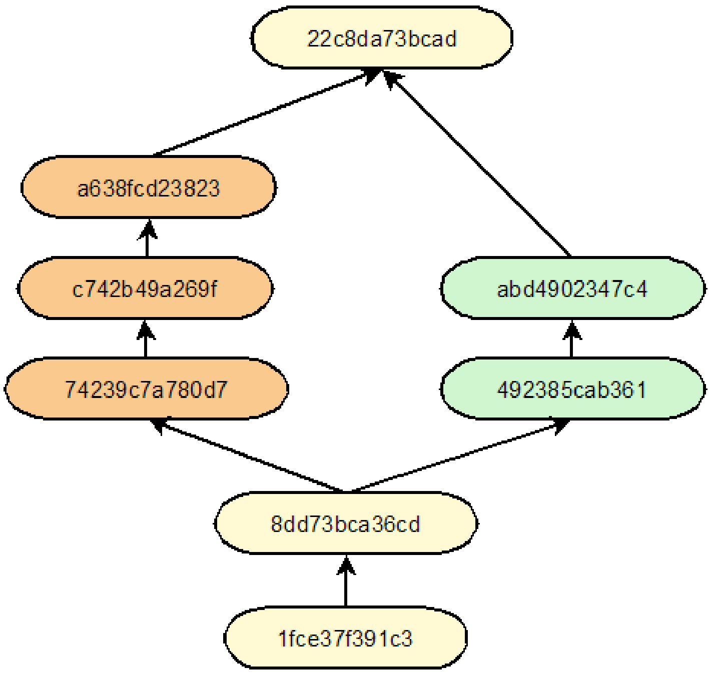
2.5.3. Resolving multiple heads: Merge

The other way of resolving 2 heads on a single branch is to merge one into the other. This preserves the history of the original deviation and then unifies the changes into a single head. As mentioned in the previouis section, you don’t want to use this approach for everyday parallel development, because it just creates a spaghetti history. However, there are less common cases where it **is** the best way to record the outcome of parallel development - specifically the cases where the parallel development has deviated for a reasonable time, such as Google Summer of Code and other longerrunning external forks that you then want to reintegrate. Rebasing history for a period of several months isn’t appropriate because it’s highly artificial - rebase is appropriate for smoothing out the misleading splitting and merging for everyday development, but if the development really did diverge for a long time, it’s worth preserving that.

So, assuming you’ve just pulled a set of changesets from a long-running fork and you now have 2 heads, you simply want to update your working copy to your own HEAD of the branch, and issue a simple command:

```
hg merge
```

Mercurial automatically knows that you want to merge the 2 heads on this branch and will do it for you. You’ll need to resolve any conflicts if they occur, then commit the merge, unifying the heads (and being able to push them if you want), like so:



2.6. Merging bugfixes across branches

The most common case where you will want to use the merge command is when merging bugfixes in the stable branch (currently v1-7) into the default (unstable) branch. This is very simple in Mercurial; if we assume we’re in a working copy which is on the head of the default branch:

```
hg merge v1-7
```

Changes will be merged into your working copy along with parent metadata tracking the merge. If there are conflicts, you’ll have to resolve them in your working copy before you commit. Once you’ve done this and tested the merge, you should commit it. You don’t need to specify much detail in the merge commit (just *Merged from v1-7* or similar) because the changeset metadata gives enough information to track which changesets are included in the merge.

Note

Any team member can perform the merge (sinbad used to do it all the time under Subversion), but it’s important that you merge the entire branch as shown above and not just specifically merge your own changes (which is possible, but not covered here because it’s the wrong thing to do). Also, it’s advisable to push your changes promptly after committing the merge to minimise the chance of overlapping with anyone else doing the same thing.

Tip

Even on Windows, I strongly recommend using KDiff3 (included with TortoiseHg) to resolve conflicts and not WinMerge or TortoiseMerge. The reason is that when using a graphical diff tool, conflicts are handled in Mercurial by keeping separate files rather than embedding the "chevron markers" inside the main file. When Mercurial delegates the merge to the GUI tool, it assumes you resolved the problem when you exit - and the problem with WinMerge is that it doesn’t give you any warning if you exit without resolving the problem (often leaving you with a file which has not merged either of the conflicting changes into it, so you lose changes!). KDiff3 however warns you if you try to exit without resolving every conflict.

Tip

I **strongly recommend** merging on the command line as above, and not using TortoiseHg which can be extremely unintuitive for doing merges. In TortoiseHg you have to left-click to select the revision you want to merge changes into (so in this case, the tip of the default branch), then **right-click** the revision you want to merge changes from (in this case, the tip of the v1-7 branch), then select "Merge With…". It’s very manual and very error prone - if you get it backwards you will effectively switch your working copy to v1-7 and merge the default changes into it!! Very bad. The command-line is far simpler and less prone to manual error.

2.7. Transplanting Individual Changes

Let’s say someone committed a bugfix to the default branch when in fact it should have been committed to the stable branch. How do you handle that? Well, in the target branch, you do this:

```
hg transplant --log REV
```

Where REV is the revision you want to transplant. The "--log" option is there to record the transplant information in the new commit.

Tip

It’s also possible to use the "-s REPOSITORY" option to transplant specific changes from another repository, as a specific alternative to pull.

2.8. Backing out a change

Mistakes happen. When they do, you need to correct them, and there are 2 ways to do this:

2.8.1. Local Rollback

This is a classic "oops" case, maybe you realised just after you committed that you forgot to add a file or forgot to remove some test code, or similar. Provided you haven’t pushed the change to anyone else, you can just undo it:

```
hg rollback
```

This removes the last commit and puts the changes from it back in your working directory, ready to be committed again. You can actually keep calling this to roll back multiple commits, but bear in mind that it will just merge the changes into your working copy, meaning you'll probably create a single commit to replace it.

2.8.2. Reverse Commit

If a change has already left your local repository, or if it's buried a few levels down in the history, then you will want to undo it in a new commit that reverses the changes. You do this using the "backout" command:

```
hg backout -m "Backed out silly mistake in X" REV
```

Notice how this commits the reversal automatically (which is why a message is included).

2.9. Using Bisect to Track Down Breakages

Oh no! Something is broken, and you don't know why, or when it broke, and don't know where to start looking. A common approach is to check out the repository at different revisions to figure out when the problem started happening, and to narrow it down by jumping around the range where the problem started happening, halving the range every time depending on whether the problem is there or not. Mercurial provides a command to make this easier: *hg bisect*.

You start the process, usually at the tip where the problem occurs, by telling Mercurial that it doesn't work (after first resetting any previous bisect state):

```
hg bisect --reset
hg bisect --bad
```

Next, we need to either jump to, or define a revision we know the problem didn't occur, some time in history. For example if we know already that it didn't occur in revision 1300, we could do this:

```
hg bisect --good 1300
```

Otherwise, we could just jump to various revisions ourselves until we found one that worked, then just call hg bisect --good with no revision to tell hg that the current revision is ok. Either way, once you've told Mercurial the extent of the good/bad range, the working copy will be updated to a revision in between the known good & bad revisions for you to test again, and to call "hg bisect --good" or "hg bisect --bad" depending on the test results. Each time you do this, the working copy will update again to the next revision for testing, bisecting the range between good and bad. If you can script this, great! If not, you'll be manually testing at each step.

Eventually you will find the case of a bad revision directly after a good revision, and hg will tell you. In this case, let's assume we've just tested revision 1519 and found that it's ok, but we'd previously tested 1520 and it wasn't:

```
c:\myproject>hg bisect --good
The first bad revision is:
changeset:   1520:898fcb8708fc48
user:        Fred Bloggs <fred@bloggs.com>
date:        Wed Jan 11 07:25:04 2010 +0000
```

```
summary:     Some commit message
```