Mercurial by Example

v0.36

by

Thorbjörn Jemander

1. Table of Contents

1 Introduction.....5
1.1 Key to reading.....5
2 Installing Mercurial.....6
3 Basics.....7
3.1 Creating a repository.....7
3.2 Adding, removing and checking in files.....7
4 History.....9
4.1 Navigation and tagging.....9
4.2 Examining: diff and cat.....10
4.3 Undo: revert, forget and rollback.....11
4.4 Searching: grep and locate.....12
4.4.1 Bisect.....13
5 Diverging history.....15
5.1 Heads.....16
5.2 Merge.....17
5.3 Branching.....17
6 Distributed development.....20
6.1 Clone.....20
6.2 Pull.....21
6.2.1 “Outstanding uncommitted changes".....22
6.3 Push.....22
6.3.1 Restrictions.....24
6.4 Merge conflicts.....26
6.5 Fetch.....28
6.6 Unrelated repositories.....29
7 Import/Export.....32
7.1 Archive.....32
7.2 Export.....32
7.3 Import.....32
7.4 Bundle/Unbundle.....33
8 Working with patches.....35
8.1 UNIX patches.....35
8.2 Patch queues.....38
8.2.1 qinit.....38
8.2.2 qnew.....38
8.2.3 qseries and qapplied.....38
8.2.4 qpop.....39
8.2.5 qpush.....40
8.2.6 qrefresh and qdiff.....41
8.2.7 Multiple patches.....42
8.2.8 Moving between patches: qgoto.....42
8.2.9 Committing patches: qfold and qfinish.....43
8.3 Rejected patches.....45
9 Extras.....47

9.1 Rebase...47
9.2 More undo..50
    9.2.1 Backout...50
    9.2.2 Clone..51
    9.2.3 Strip..52
9.3 Transplant aka Cherry picking...54
9.4 Setting up a server...56
9.5 Tracking external software..56
    9.5.1 Using two repositories...56
    9.5.2 Using an import branch..58

1 Introduction

I wanted an examples based Mercurial tutorial, which was quick and to-the-point, without a lot of theory or principles of operation. Here it is, without prating.

Do worry if it seems long to you, you'll walk through it rather rapidly.

1.1 Key to reading

The table below shows the appearing boxes in this document.

Item	Description

`$ ls --flag` `# this command`
`foo.txt`
`bar.txt       # this file`
	Terminal commands, $ represents the prompt. Input is in **bold,** output is not bold.
	`Red text` is interesting input, the point of the example.
	`Blue text` is interesting output.
	`#Green hashed text` are comments, not to be typed in.

Tip: if you write...	Comment, tip, further reading or reference.

`Foo`
`bar`
The file Myfile.sh
	Contents of a file, in this case Myfile.sh

It is not recommended to skip examples, as later examples may depend on the results of the previous.

2 Installing Mercurial

If you're using a Debian-based Linux PC, which has the apt program manager, you can do the following:

```
$ sudo apt-get install mercurial
$ hg version
Mercurial Distributed SCM (version 1.3.1)

Copyright (C) 2005-2009 Matt Mackall <mpm@selenic.com> and others
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

For other OS:es, see http://mercurial.selenic.com/wiki/Download.

3 Basics

3.1 Creating a repository

Create a mercurial repository in an empty directory by using the `init` command:

```
$ mkdir hg1
$ cd hg1
$ hg init
```

Now there is an empty repository created, locally in your directory. The repository contains both a *working copy*, and the *repository data* (the history information). This is unlike non-distributed SCMs, like CVS or SVN, where the the repository contains only the revision history and the working copy only contains the checked out code.

3.2 Adding, removing and checking in files

Time to start make some history:

```
$ echo A > x1
$ hg add x1
$ hg ci -m"Added feature A to module x1."
```

This is very similar to SVN. Make some more history and learn how to remove:

```
$ echo A > x2
$ echo B > y1
$ hg add x2 y1
$ hg ci -m"Added A to x2 and B to y1."
$ hg rm x1
$ hg ci -m"Removed x1"
```

So now we're added and removed files explicitly. However, there's a command called `addremove` that detects added and removed files automatically for us:

```
$ echo C > z1
$ rm x2                         # Note: not “hg rm”
$ hg addremove
removing x2
adding z1
$ hg status
A z1
R x2
$ hg ci -m"Adding C to z1 and removing x2"
```

Status is a command to check what the pre-commit status is. It tells you whether files are (A)dded, (R)emoved, (M)odified among other things. Check out the **manifest** command too.

So the status command gives some nice information before commit. Let's look at our log now:

```
$ hg log
changeset:   3:c9d133e1c054
tag:         tip
user:        Thorbjorn Jemander
date:        Sat Dec 12 10:15:49 2009 +0100
summary:     Adding C to z1 and removing x2

changeset:   2:3d8afaf1ac30
user:        Thorbjorn Jemander
date:        Sat Dec 12 10:13:09 2009 +0100
summary:     Removed x1

changeset:   1:01e9f6a4aae5
user:        Thorbjorn Jemander
date:        Sat Dec 12 10:12:09 2009 +0100
summary:     Added A to x2 and B to y1.

changeset:   0:593b00eec7b1
user:        Thorbjorn Jemander
date:        Sat Dec 12 09:49:22 2009 +0100
summary:     Added feature A to module x1
```

We can see that there is a straight history, starting with revision zero and ending with revision three. Revision three is also the so-called *tip,* which marks the most recent point in the revision history.

If you wish, you can look at parts of the history by specify revision ranges:

```
$ hg log -r 1:2        # From 1 to 2
changeset:   2:3d8afaf1ac30
user:        Thorbjorn Jemander
date:        Sat Dec 12 10:13:09 2009 +0100
summary:     Removed x1

changeset:   1:01e9f6a4aae5
user:        Thorbjorn Jemander
date:        Sat Dec 12 10:12:09 2009 +0100
summary:     Added A to x2 and B to y1.
$ hg log -r :1         # Up to, and including 1.
changeset:   1:01e9f6a4aae5
user:        Thorbjorn Jemander
date:        Sat Dec 12 10:12:09 2009 +0100
summary:     Added A to x2 and B to y1.

changeset:   0:593b00eec7b1
user:        Thorbjorn Jemander
date:        Sat Dec 12 09:49:22 2009 +0100
summary:     Added feature A to module x1
$ hg log -r 2:         # From, and including 2.
changeset:   3:c9d133e1c054
tag:         tip
user:        Thorbjorn Jemander
date:        Sat Dec 12 10:15:49 2009 +0100
summary:     Adding C to z1 and removing x2

changeset:   2:3d8afaf1ac30
user:        Thorbjorn Jemander
date:        Sat Dec 12 10:13:09 2009 +0100
summary:     Removed x1
```

4 History

4.1 Navigation and tagging

Before we start moving – where are we right now? Use the `identity` command to check:

```
$ hg identify -n
3                              # We're at revision 3.
```

Currently we're at changeset 3. To move in the repository history, we use the `update` command:

```
$ hg update -r 0               # Move to revision 0.
1 files updated, 0 files merged, 2 files removed, 0 files unresolved
$ ls
x1
```

There is a convention to use `-r` to specify revisions, but in `update`, the `-r` is actually optional. Most commands can be abbreviated:

```
$ hg up                        # No revision specified → we go to the latest.
1 files updated, 0 files merged, 2 files removed, 0 files unresolved
$ ls
y1 z1
```

With no argument to `update`, it moves to the the *tip*. As mentioned above, the tip is a symbolic name for the latest revision. You can assign your own symbolic names to specific revisions. These are known as tags:

```
$ hg tag -r 2 my-x1-removal     # Associate a name to revision 2
$ hg log
changeset:   4:4207f3b7f4ee
tag:         tip
user:        Thorbjorn Jemander
date:        Sat Dec 12 16:41:03 2009 +0100
summary:     Added tag my-x1-removal for changeset 3d8afaf1ac30

changeset:   3:c9d133e1c054
user:        Thorbjorn Jemander
date:        Sat Dec 12 10:15:49 2009 +0100
summary:     Adding C to z1 and removing x2

changeset:   2:3d8afaf1ac30
tag:         my-x1-removal
user:        Thorbjorn Jemander
date:        Sat Dec 12 10:13:09 2009 +0100
summary:     Removed x1

changeset:   1:01e9f6a4aae5
user:        Thorbjorn Jemander
date:        Sat Dec 12 10:12:09 2009 +0100
summary:     Added A to x2 and B to y1.

changeset:   0:593b00eec7b1
user:        Thorbjorn Jemander
```

```
date:         Sat Dec 12 09:49:22 2009 +0100
summary:      Added feature A to module x1
$ hg tags
tip                                  4:4207f3b7f4ee
my-x1-removal                        2:3d8afaf1ac30
```

Thus, the tags can be explicitly listed using the `tags` command, but they also show up in the revision log. You can use tag names instead of changeset numbers in most commands.

4.2 Examining: diff and cat

We can rename files and directories:

```
$ hg rename y1 y2
$ hg ci -m"y1 renamed to y2"
$ ls
y2  z1
$ echo C > y2
$ hg ci -m"Updated module y2 with feature C."
```

New contents, new name, but the identity of a file is tracked. Note that the `-f` flag is needed to log across renames, or the `log` command will stop at the rename:

```
$ hg log y2
changeset:    6:47445946964f
tag:          tip
user:         Thorbjorn Jemander
date:         Sat Dec 12 17:08:21 2009 +0100
summary:      Updated module y2 with feature C.

changeset:    5:538cc1eb311d
user:         Thorbjorn Jemander
date:         Sat Dec 12 17:05:38 2009 +0100
summary:      y1 renamed to y2

$ hg log -f y2
changeset:    6:47445946964f
tag:          tip
user:         Thorbjorn Jemander
date:         Sat Dec 12 17:08:21 2009 +0100
summary:      Updated module y2 with feature C.

changeset:    5:538cc1eb311d
user:         Thorbjorn Jemander
date:         Sat Dec 12 17:05:38 2009 +0100
summary:      y1 renamed to y2

changeset:    1:01e9f6a4aae5
user:         Thorbjorn Jemander
date:         Sat Dec 12 10:12:09 2009 +0100
summary:      Added A to x2 and B to y1.
```

We don't need to update the entire repository to another revision to look at the contents of individual files. We can check the difference (`diff`) and print the whole file (`cat`):

```
$ echo D > y2
$ hg diff y2    # What's the difference between the repo. and the working copy?
diff -r 47445946964f y2
--- a/y2  Sat Dec 12 17:08:21 2009 +0100
+++ b/y2  Sat Dec 12 17:22:41 2009 +0100
@@ -1,1 +1,1 @@
-C
+D
$ hg diff -r 5 y2   # How is the working copy different form revision 5?
diff -r 538cc1eb311d y2
--- a/y2  Sat Dec 12 17:05:38 2009 +0100
+++ b/y2  Sat Dec 12 17:23:41 2009 +0100
@@ -1,1 +1,1 @@
-B
+D
$ hg cat -r 5 y2     # Print (cat) y2 as it were in revision 5.
B
$ hg ci -m"Let advance to experimental D version"
```

This last example, with `cat`, can be used to restore files from history, by redirecting the contents to the local file:

```
$ hg tip
changeset:   7:5090879191a2
tag:         tip
user:        Thorbjorn Jemander
date:        Sat Dec 12 17:31:47 2009 +0100
summary:     Let advance to experimental D version
$ hg cat -r 6 y2 > y2     # Print revision 6 and redirect to local file y2
$ hg diff
diff -r 5090879191a2 y2
--- a/y2  Sat Dec 12 17:31:47 2009 +0100
+++ b/y2  Sat Dec 12 17:34:01 2009 +0100
@@ -1,1 +1,1 @@
-D
+C
$ hg ci -m"D caused a thermonuclear explosion. Backed to C."
```

That's a way to undo a previous change, but a bit primitive. Let's move on to the to topic of undoing things.

4.3 Undo: revert, forget and rollback

The last example above showed a way to restore from files previous states in the next revision by using `cat`. However, there is a dedicated command – `revert` – that does precisely that (which also works for directories):

```
$ hg revert -r 5 y2      # Restores the local copy to the revision 5 state
$ cat y2
B
$ hg ci -m"y2/C caused a chemical explosion. Going back to y2/B."
```

But what if we've accidentally added files, and want to remove them before we check in? There's an easy way to tell Mercurial to forget about them:

```
$ touch a y2.o z1.o
$ hg addremove
adding a
adding y2.o
adding z1.o
$ hg forget y2.o z1.o
$ hg ci -m"Adding new module a"
$ rm *.o
```

Do you always want to ignore .o files? You can specify file patterns to ignore in a file called .hgignore in the root of the repository. Read more in the man page for hgignore: man hgignore.

Assume we have committed something very stupid, that we really regret. Fortunately, we can rollback the last transaction. Think of `rollback` as “undo the last check-in":

```
$ mv * /tmp
...

$ hg addremove                     # Maybe you get distracted...
removing a
removing y2
removing z1
$ hg ci -m"quick check-in"          # ... and check in something really bad.
$ ls                               # Nothings left. And it's checked in!
$ hg rollback                      # Don't worry, rollback!
rolling back last transaction
$ hg up -C                         # We need to update the working copy.
3 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ ls                               # Phew, we're back.
a y2 z1
```

Note that you need to update after the rollback. Note also that you have not just moved back in history one step, you have completely erased the last check-in from history.

You cannot perform several consecutive rollbacks as Mercurial only stores information for one rollback. If you stay with us until chapter 9.2 , we'll discuss more powerful undo techniques.

4.4 Searching: grep and locate

We now know how to commit, undo and how to move back and forth in the history. Next topic is about locating files and finding contents in the history. There are two commands available for this: `grep` and `locate`. They work similar to the UNIX shell commands with the same name, but with the addition of history information. In the example below, remember that *y1* was renamed to *y2* early on, and that *y2* has changed contents throughout history (B → C → D → C → B).

```
$ hg locate y1               # There's no y1 in the current version
$ hg locate -r 1 y1          # There's a y1 in revision 1
y1
$ hg grep B y2
y2:9:B                       # y2 in rev.9 contained B
$ hg grep --all B y2
y2:9:+:B                     # B reappeared in 9
y2:6:-:B                     # B disappeared in 6
y2:5:+:B                     # B first appeared in 5
$ hg grep -f --all B y2      # With -f it goes across
y2:9:+:B                     # renames.
y2:6:-:B
y2:5:+:B
y1:1:+:B
```

Do you want to know who is responsible for a particular line in a file? You can find out by using the “blame” command, or by its proper name “annotate”. Use “hg help annotate” to find out more.

The flag `--all` means print all matches, while `-f` means (just as above) “follow renames”.

4.4.1 Bisect

Suppose you have received a bug report from a customer, and you want to know when this bug was introduced to deduce how many releases that are affected. Mercurial can help you find the origin of the bug, if you can provide it with a *test command*. The test command must return 0 if the bug is not present and 1 if the bug is present. Mercurial uses the test command while it jumps back and forth in history to deduce whether revisions are *good* or *bad*. It stops when it has found the earliest version that is bad.

For example, let say we have a bug in our system that is represented by that the *y* and *z* files have different numbers: thus having the files *y2* ans *z1* simultaneously represents a bug condition, while *y1* and *z1* would not. Then a (very simple) test function could look like the one to the right, implemented as a shell script.

```
#!/bin/sh
if [ z* = z1 -a y* = y1 ]; then
   echo No bug!
   exit 0
else
   echo Bug!
   exit 1
fi
```

The file test.sh

Copy this script to a file named **test.sh** and make it executable (using chmod). Run it and you'll see that it tells you that the bug is present in the current working copy.

For Mercurial to find the bug we need to tell it three things:

1. A version that has the bug/is bad (`hg bisect -b`),
2. A version that doesn't have the bug/is good (`hg bisect -g`),
3. The test command (`hg bisect -c`)

We know for a fact that changeset 3 didn't contain the bug and that the current revision (10) does.

Enter the following and Mercurial will run your test command on an intelligent selection of revisions to deduce where the bug was first introduced:

```
$ hg bisect -r      # Reset the bisect state
$ hg bisect -b 10   # Revision 10 (the tip) has the bug
$ hg bisect -g 3    # Revision 3 does not.
Testing changeset 6:1f6db2bf99f3 (7 changesets remaining, ~2 tests)
1 files updated, 0 files merged, 1 files removed, 0 files unresolved
$ hg bisect -c ./test.sh
Bug!
Changeset 6:47445946964f: bad
No bug!
Changeset 4:4207f3b7f4ee: good
Bug!
Changeset 5:538cc1eb311d: bad
The first bad revision is:
changeset:   5:538cc1eb311d
user:        Thorbjorn Jemander
date:        Sat Dec 12 17:05:38 2009 +0100
summary:     y1 renamed to y2
$ rm test.sh
```

The bug was introduced in changeset 5, where we renamed *y1* to *y2*. In this example, we could have seen that by looking at the log, but it is usually not so simple to spot bugs.

5 Diverging history

Before we get into the next subject, let's add an extension to Mercurial that makes it easier to view history: the graphlog extension. Open (or create) the *.hgrc* file in your home directory.

In the file to the right, a user name is given and the graphlog extension is enabled. Save the file and test the `glog` command:

```
[ui]
username = Thorbjorn Jemander

[extensions]
graphlog=
```

The file ~/.hgrc

```
$ hg glog
o  changeset:   10:07c3c9ade9e5
|  tag:         tip
|  user:        Thorbjorn Jemander
|  date:        Sun Dec 13 13:15:37 2009 +0100
|  summary:     Adding new module a
|
o  changeset:   9:1c03fc41fe77
|  user:        Thorbjorn Jemander
|  date:        Sat Dec 12 17:46:52 2009 +0100
|  summary:     y2/C caused a chemical explosion. Going back to y2/B.
|
o  changeset:   8:6ff8f0a0aa98
|  user:        Thorbjorn Jemander
|  date:        Sat Dec 12 17:35:29 2009 +0100
|  summary:     D caused a thermonuclear explosion. Backed to C.
|
o  changeset:   7:5090879191a2
|  user:        Thorbjorn Jemander
|  date:        Sat Dec 12 17:31:47 2009 +0100
|  summary:     Let advance to experimental D version
...
```

If you see the ring-dotted line to the left you have successfully enabled the graphlog extension. There are several other extensions that one can enable, I'll give you some tips later on.

Consider the following scenario. You have released R1.0 to the customer a while back (at revision 3), and now are working towards R2.0, and have done a lot of changes. You suddenly get a bug report from the customer on R1.0 and there's an urgent need for a correction. You're nowhere near releasing R2.0, so you need to work on R1.0.

Let's see what happens if we go back in history and start working on R1.0 (changeset 3):

```
$ hg update -r 3   # Go to revision 3
1 files updated, 0 files merged, 2 files removed, 0 files unresolved
$ ls
y1 z1
$ cat y1
B
$ echo C > y1      # We've fixed the bug by changing y1
$ hg ci -m"Fixed the R1.0 bug."
created new head
```

What? What does “created new head" mean? To cite the movie “Back to the future", you have just created an “alternative future" or “alternative time line". Instead of having one straight line from the first revision to the last, you now have two parallel lines starting at revision 3. To see what I mean, use graphlog:

```
$ hg glog
@  changeset:   11:47da354e2d4d
|  tag:         tip
|  parent:      3:c9d133e1c054
|  user:        Thorbjorn Jemander
|  date:        Tue Dec 15 22:14:59 2009 +0100
|  summary:     Fixed the R1.0 bug.
|
| o  changeset:   10:07c3c9ade9e5
| |  user:        Thorbjorn Jemander
| |  date:        Sun Dec 13 13:15:37 2009 +0100
| |  summary:     Adding new module a
| |
| o  changeset:   9:1c03fc41fe77
| |  user:        Thorbjorn Jemander
| |  date:        Sat Dec 12 17:46:52 2009 +0100
| |  summary:     y2/C caused a chemical explosion. Going back to y2/B.
| |
...
| o  changeset:   5:538cc1eb311d
| |  user:        Thorbjorn Jemander
| |  date:        Sat Dec 12 17:05:38 2009 +0100
| |  summary:     y1 renamed to y2
| |
| o  changeset:   4:4207f3b7f4ee
|/   user:        Thorbjorn Jemander
|    date:        Sat Dec 12 16:41:03 2009 +0100
|    summary:     Added tag my-x1-removal for changeset 3d8afaf1ac30
|
o  changeset:   3:c9d133e1c054
|  user:        Thorbjorn Jemander
|  date:        Sat Dec 12 10:15:49 2009 +0100
|  summary:     Adding C to z1 and removing x2
...
```

You now have two *heads,* changeset 10 and 11. The *parent* of changeset 10 is 9, while the parent of revision 11 is 3. You have implicitly created a branch, a branch without a name. Their *common ancestor* is changeset 3.

5.1 Heads

If you want to display the heads of your repository, use the `heads` command:

```
$ hg heads
changeset:   11:47da354e2d4d
tag:         tip
parent:      3:c9d133e1c054
user:        Thorbjorn Jemander
date:        Tue Dec 15 22:14:59 2009 +0100
summary:     Fixed the R1.0 bug.

changeset:   10:07c3c9ade9e5
user:        Thorbjorn Jemander
date:        Sun Dec 13 13:15:37 2009 +0100
summary:     Adding new module a
```

5.2 Merge

You may have one, two or more heads. However, for practical reasons one should always try to have only one head. To merge the two branches, use the `merge` command:

```
$ hg merge
merging y1 and y2 to y2
2 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg ci -m"Merged R1.0-fix with pre-2.0"
$ hg glog
@    changeset:   12:90cf941b9def
|\   tag:         tip
| |  parent:      11:47da354e2d4d
| |  parent:      10:07c3c9ade9e5
| |  user:        Thorbjorn Jemander
| |  date:        Tue Dec 15 22:31:43 2009 +0100
| |  summary:     Merged R1.0-fix with pre-2.0
| |
| o  changeset:   11:47da354e2d4d
| |  parent:      3:c9d133e1c054
| |  user:        Thorbjorn Jemander
| |  date:        Tue Dec 15 22:14:59 2009 +0100
| |  summary:     Fixed the R1.0 bug.
| |
o |  changeset:   10:07c3c9ade9e5
| |  user:        Thorbjorn Jemander
| |  date:        Sun Dec 13 13:15:37 2009 +0100
| |  summary:     Adding new module a
```

Now we're back at only having a single head again. Changeset 12 has two parents, 11 and 10.

5.3 Branching

The proper way to do branching is to use the `branch` command, and then you get symbolic names for them, as well. The following sequence creates a R1.5 branch, originating from revision 7:

```
$ hg up 7                      # Go to revision 7...
1 files updated, 0 files merged, 1 files removed, 0 files unresolved
$ hg branch R1.5               # and create branch R1.5
marked working directory as branch R1.5
$ hg ci -m"Created R1.5 branch"
created new head
$ hg branches
R1.5                           13:a977411494e5
default                        12:90cf941b9def
$ hg glog
@  changeset:   13:a977411494e5
|  branch:      R1.5
|  tag:         tip
|  parent:      7:5090879191a2
|  user:        Thorbjorn Jemander
|  date:        Tue Dec 15 22:39:21 2009 +0100
|  summary:     Created R1.5 branch
|
| o    changeset:   12:90cf941b9def
| |\   parent:      11:47da354e2d4d
```

Note that these branches are not local, i.e. they will be exported to other repositories when you start to move changes between repositories. So don't use them for private development.

```
| | |   parent:      10:07c3c9ade9e5
| | |   user:        Thorbjorn Jemander
| | |   date:        Tue Dec 15 22:31:43 2009 +0100
| | |   summary:     Merged R1.0-fix with pre-2.0
| | |
....
| | |
| o |   changeset:   9:1c03fc41fe77
| | |   user:        Thorbjorn Jemander
| | |   date:        Sat Dec 12 17:46:52 2009 +0100
| | |   summary:     y2/C caused a chemical explosion. Going back to y2/B.
| | |
| o |   changeset:   8:6ff8f0a0aa98
|/ /    user:        Thorbjorn Jemander
| |     date:        Sat Dec 12 17:35:29 2009 +0100
| |     summary:     D caused a thermonuclear explosion. Backed to C.
| |
o |   changeset:   7:5090879191a2
| |   user:        Thorbjorn Jemander
| |   date:        Sat Dec 12 17:31:47 2009 +0100
| |   summary:     Let advance to experimental D version
```

So, now again, we have two *heads*: 13 and 12. We can jump between branches by using branch names as arguments to the `update` command (just as tags and revision numbers):

```
$ hg up default
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg up R1.5
1 files updated, 0 files merged, 1 files removed, 0 files unresolved
$ hg branch                            # No argument → gives us the branch name
R1.5
```

Before we continue, let's close and merge the R1.5 branch so that we don't have multiple heads:

```
$ hg ci --close-branch -m"Branch R1.5 closed."
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg up default
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg merge R1.5
0 files updated, 0 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg ci -m"R1.5 merged into default."
$ hg branches
default                       15:25ce584f89d6
$ hg glog
@    changeset:   15:25ce584f89d6
|\   tag:         tip
| |  parent:      12:4b9ce0577295
| |  parent:      14:c0ba01e32db9
| |  user:        Thorbjorn Jemander
| |  date:        Sat Dec 19 11:05:54 2009 +0100
| |  summary:     R1.5 merged into default.
| |
| o  changeset:   14:c0ba01e32db9
| |  branch:      R1.5
| |  user:        Thorbjorn Jemander
| |  date:        Sat Dec 19 11:04:35 2009 +0100
| |  summary:     Branch R1.5 closed.
| |
| o  changeset:   13:cdfb8c39e915
| |  branch:      R1.5
| |  parent:      7:bf7c4059b77a
| |  user:        Thorbjorn Jemander
| |  date:        Sat Dec 19 10:51:06 2009 +0100
| |  summary:     Created R1.5 branch
...
```

Sometimes, you don't want to merge in a whole branch, just pick some specific changes. This is referred to as “cherry-picking” and can be done by using the “transplant” command. See the Extras chapter.

Note that the only thing the flag `--close-branch` does is to hide the branch name and mark it inactive. If you know the name, you can continue to use the branch name in commands, as it remains a valid identifier. It will also be visible in the change history, but it will not show up when you issue `hg branches`.

6 Distributed development

Let us move on to subject of the distributed development, i.e. working with several repositories.

6.1 Clone

One can create an exact replica of a repository by using the *clone* command:

```
$ cd ..
$ hg clone hg1 hg2
updating working directory
4 files updated, 0 files merged, 0 files removed,
 0 files unresolved
$ cd hg2
$ hg glog
@    changeset:   15:25ce584f89d6
|\   tag:         tip
| |  parent:      12:4b9ce0577295
| |  parent:      14:c0ba01e32db9
| |  user:        Thorbjorn Jemander
| |  date:        Sat Dec 19 11:05:54 2009 +0100
| |  summary:     R1.5 merged into default.
| |
| o  changeset:   14:c0ba01e32db9
| |  branch:      R1.5
| |  user:        Thorbjorn Jemander
| |  date:        Sat Dec 19 11:04:35 2009 +0100
| |  summary:     Branch R1.5 closed.
| |
| o  changeset:   13:cdfb8c39e915
| |  branch:      R1.5
| |  parent:      7:bf7c4059b77a
| |  user:        Thorbjorn Jemander
| |  date:        Sat Dec 19 10:51:06 2009 +0100
| |  summary:     Created R1.5 branch
| |
o |    changeset:   12:4b9ce0577295
|\ \   parent:      11:a53cffb1a107
| | |  parent:      10:185eb7291e1d
| | |  user:        Thorbjorn Jemander
| | |  date:        Sat Dec 19 10:50:19 2009 +0100
| | |  summary:     Merged R1.0-fix with pre-2.0
| | |
| o |  changeset:   11:a53cffb1a107
| | |  parent:      3:9277041236ca
...
```

Here we clone a local copy, but you can clone a remote repository as well, by specifying an URL, .e.g. http://, https:// or ssh://

The clone has the same history as the original repository, including the branch and tag information. One thing separates the clone from the original repository: the clone knows which repository it was cloned from. It has a *default repository*:

```
$ cat .hg/hgrc
.hg/hgrc:default = /home/<user>/hg-tutorial/hg1
```

The `.hg` directory contains all the bookkeeping information about the repository, and the hgrc file points out (among other things) the default repository. It is possible to pull and push changes between repositories and if no explicit argument is given, the default repository is assumed:

```
$ hg incoming                       # Checks the default repository.
comparing with /home/thorman/hg-tutorial/hg1
searching for changes
no changes found
$ cd ../hg1
$ hg incoming                       # hg1 has no default repository.
abort: repository default not found!
```

6.2 Pull

Lets start to move changes between the two repositories:

```
$ cd ../hg1
$ echo “new feature" > b
$ hg addremove
adding b
$ hg ci -m"added new feature to b"
$ hg glog
@  changeset:   16:cfa13befd88e
|  tag:         tip
|  user:        Thorbjorn Jemander
|  date:        Sat Dec 19 11:16:17 2009 +0100
|  summary:     added new feature to b
|
o    changeset:   15:25ce584f89d6
|\   parent:      12:4b9ce0577295
| |  parent:      14:c0ba01e32db9
| |  user:        Thorbjorn Jemander
| |  date:        Sat Dec 19 11:05:54 2009 +0100
| |  summary:     R1.5 merged into default.
| |
| o  changeset:   14:c0ba01e32db9
| |  branch:      R1.5
| |  user:        Thorbjorn Jemander
| |  date:        Sat Dec 19 11:04:35 2009 +0100
| |  summary:     Branch R1.5 closed.
...
$ cd ../hg2
$ hg incoming                # Let's check if there are any changes for us..?
comparing with /home/thorman/hg-tutorial/hg1
searching for changes
changeset:   16:cfa13befd88e
tag:         tip
user:        Thorbjorn Jemander
date:        Sat Dec 19 11:16:17 2009 +0100
summary:     added new feature to b
$ hg pull                    # Yes, there are. Pull them into our repository
pulling from /home/thorman/hg-tutorial/hg1
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
(run 'hg update' to get a working copy)
$ ls
a  y2  z1                    # What? No b?
$ hg up                      # Ah, we need to update.
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ ls
a  b  y2  z1
```

6.2.1 “Outstanding uncommitted changes"

Sometimes one is in the middle of a big rewrite, and don't want to commit it (for whatever reasons) but still want pull changes from other repositories. Usually it works to pull changes, merge and perform an update, but under certain conditions you will encounter the following error message:

```
abort: outstanding uncommitted changes
```

This diagnostic means that Mercurial had problems merging with the uncommitted changes. This can cause some headache from time to time. There are three ways (which I know of) to handle this:

1. Always commit before pulling in changes. This is easier said than done, however. Sometimes one would not want the current changes to be committed. It could be that it's not compiling or one may want to submit the changes in a single commit (e.g. to make the review the change easy).
2. Use a separate repository for importing changes, like it is described here: http://blogs.sun.com/tor/entry/mercurial_tip_checking_in_regularly It works, but I find it impractical to have another repository just for that reason.
3. The third way (and the one *I* prefer) is to temporarily lift my local changes out, pull in and merge the external changesets and then reapply my local changes.

How and why this happens, is described in detail in chapter 35. There I also present some practical ways to solve the problem.

Next is the opposite to pull: push.

6.3 Push

We can push back changes to the original repository by using the command `push`:

```
$ echo "New module: c" > c      # NB. we're in the hg2 directory
$ hg add c
$ hg ci -m"Added the new module c"
$ hg outgoing                   # Check what's going out.
comparing with /home/thorman/hg-tutorial/hg1
searching for changes
changeset:   17:8e48e69496ec
tag:         tip
user:        Thorbjorn Jemander
date:        Sat Dec 19 11:19:02 2009 +0100
summary:     Added the new module c
$ hg push
pushing to /home/thorman/hg-tutorial/hg1
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
$ cd ../hg1
$ ls
a b y2 z1                      # What? No module c?
$ hg up                        # Ah, we need to update.
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ ls
```

```
a b c y2 z1
```

When we push, the recipient repository will not automatically see the changes in the source code until an update is made (which is identical to the default behavior of pull).

That was the easy case. Assume now that two developers are working in parallel on the repositories hg1 and hg2 and want to exchange code:

```
$ echo “A new nifty feature" > b      # We're in the hg1 directory
$ hg ci -m"New feature added to b."
$ cd ../hg2
$ echo “Bugfix" > c
$ hg ci -m"Fixed a bug in c."
$ hg incoming
comparing with /home/thorman/hg-tutorial/hg1
searching for changes
changeset:   18:ef562bd14249
tag:         tip
user:        Thorbjorn Jemander
date:        Sat Dec 19 11:24:18 2009 +0100
summary:     New feature added to b.
$ hg pull
pulling from /home/thorman/hg-tutorial/hg1
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
```

Hmm... why did Mercurial tell us “`+1 heads`"? The reason is that both the hg1 and hg2 changes were based on changeset 17, so when we pulled in the hg1 changes, we suddenly have two heads, 18 and 19:

```
$ hg glog
o  changeset:   19:ef562bd14249
|  tag:         tip
|  parent:      17:8e48e69496ec
|  user:        Thorbjorn Jemander
|  date:        Sat Dec 19 11:24:18 2009 +0100
|  summary:     New feature added to b.
|
| @  changeset:   18:a7ba278d6de2
|/   user:        Thorbjorn Jemander
|    date:        Sat Dec 19 11:24:53 2009 +0100
|    summary:     Fixed a bug in c.
|
o  changeset:   17:8e48e69496ec
|  user:        Thorbjorn Jemander
|  date:        Sat Dec 19 11:19:02 2009 +0100
|  summary:     Added the new module c
...
$ hg heads
changeset:   19:ef562bd14249
tag:         tip
parent:      17:8e48e69496ec
user:        Thorbjorn Jemander
date:        Sat Dec 19 11:24:18 2009 +0100
summary:     New feature added to b.
```

```
changeset:   18:a7ba278d6de2
user:        Thorbjorn Jemander
date:        Sat Dec 19 11:24:53 2009 +0100
summary:     Fixed a bug in c.
```

`hg heads` confirms that we have two heads, changeset 18 and 19.

We need to merge the two heads, and only after that we can push the changes to hg1:

```
$ hg merge
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg ci -m"Merged in hg1 changes."
$ hg push
pushing to /home/thorman/hg-tutorial/hg1
searching for changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 1 changes to 1 files
$ cd ../hg1
$ hg up
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

We need to merge every time we pull changes from another repository and have (committed) local changes in our repository. This is a necessity because there are parallel lines of development in the two repositories. Other SCM tools do not reveal this fact, but Mercurial makes it clear what is going on. Does this mean that we need to run the merge command manually every time we pull? No, this can be automated, as we will see later. But first we will address couple of other issues.

6.3.1 Restrictions

Now we're in hg2, and have merged in the new “b" feature and shared the bug fix we did in “c" with the hg1 developer. In the mean time, the hg1 developer has been at work and added another feature to b:

```
$ cd ../hg1
$ echo "another nice feature" >> b        # Note the double >!
$ hg ci -m"Added another feature to b."
```

And the hg2 developer has been busy too:

```
$ cd ../hg2
$ echo "another bugfix in c" >> c         # Note the double >!
$ hg ci -m"Another bugfix to c."
```

The hg2 developer now tries to share its bug fix to the hg1 developer, by pushing the changes:

```
$ hg push
pushing to /home/thorman/hg-tutorial/hg1
searching for changes
abort: push creates new remote heads!
```

```
(did you forget to merge? use push -f to force)
```

Mercurial refused because the push would create new heads in the remote repository. Why? Let's look at the two logs:

```
$ cd ../hg1
$ hg glog
@  changeset:   21:7c33c9969d45
|  tag:         tip
|  user:        Thorbjorn Jemander
|  date:        Mon Dec 28 14:00:43 2009 +0100
|  summary:     Added another feature to b.
|
o    changeset:   20:68ee78138e47
|\   parent:      19:bf134b53f0f0
| |  parent:      18:c22bd5b3858a
| |  user:        Thorbjorn Jemander
| |  date:        Mon Dec 28 13:44:23 2009 +0100
| |  summary:     Merged in hg1 changes.

...
```

Here we see that changeset 20 (“Merged in hg1 changes") has a child, 21 (“Added another feature to b."), which is specific to hg1. If we look at what the hg2 tree looks like,

```
$ cd ../hg2
$ hg glog
@  changeset:   21:d8ab303e592a
|  tag:         tip
|  user:        Thorbjorn Jemander
|  date:        Mon Dec 28 14:02:13 2009 +0100
|  summary:     Another bugfix to c.
|
o    changeset:   20:68ee78138e47
|\   parent:      18:bf134b53f0f0
| |  parent:      19:c22bd5b3858a
| |  user:        Thorbjorn Jemander
| |  date:        Mon Dec 28 13:44:23 2009 +0100
| |  summary:     Merged in hg1 changes.
...
```

we can see that hg2 has the same changeset 20 (“Merged in hg1 changes"). However, it has produced a local child, 21 (“Another bugfix to c."). If we would force the push, changeset 20 would get two children, and therefore we would get an extra head.

Mercurial thinks it is impolite (or impractical) to create extra heads in remote repositories, so it only allows creating heads when you *pull* changes:

```
$ cd ../hg1
$ hg pull ../hg2
pulling from ../hg2
searching for changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 1 changes to 2 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
```

```
$ hg merge
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg ci -m"Merged in hg2 changes."
```

Alternatively, you pull in the hg1 changes to hg2, merge and then push back the result to hg1, but don't do that now as it will mess things up for the next example.

6.4 Merge conflicts

First, install a merge tool. I'll use kdiff3 here.

```
$ sudo apt-get install kdiff3
```

Then, update the ~/.hgrc file to contain the new entries shown in the file below (we tell Mercurial to use kdiff3 as a merge tool).

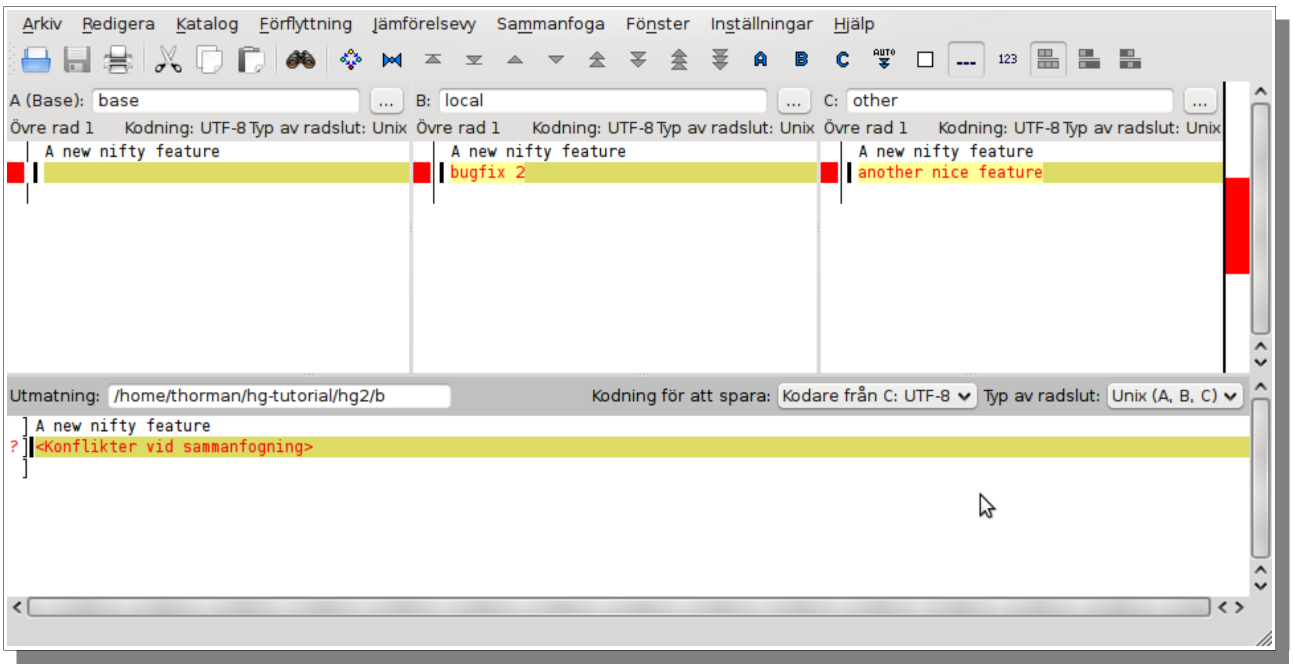
```
[ui]
username = Thorbjorn Jemander

[extensions]
graphlog=
hgext.extdiff =

[extdiff]
cmd.kdiff3 =

[merge-tools]
kdiff3.args = $base $local $other -o $output
```

The file ~/.hgrc

Let's try kdiff3 on a conflict that we will create between hg1 and hg2, by changing the same file on the same line. Hg2 will add a “bugfix no 2" to *b*, which will conflict with the “another feature" changeset of hg1:

```
$ cd ../hg2
$ echo “bugfix 2" >> b
$ hg ci -m"Bug no 2 fixed in b."
$ hg pull
...
$ hg merge
merging b
```

Now Mercurial sees that both hg1 and hg2 has changed *b* in a way not possible to merge automatically. It therefore invokes the manual merge tool kdiff3:


Arkiv Redigera Katalog Förflyttning Jämförelsevy Sammanfoga Fönster Inställningar Hjälp
A (Base): base
B: local
C: other
Övre rad 1 Kodning: UTF-8 Typ av radslut: Unix
Utmatning: /home/thorman/hg-tutorial/hg2/b
Kodning för att spara: Kodare från C: UTF-8
Typ av radslut: Unix (A, B, C)

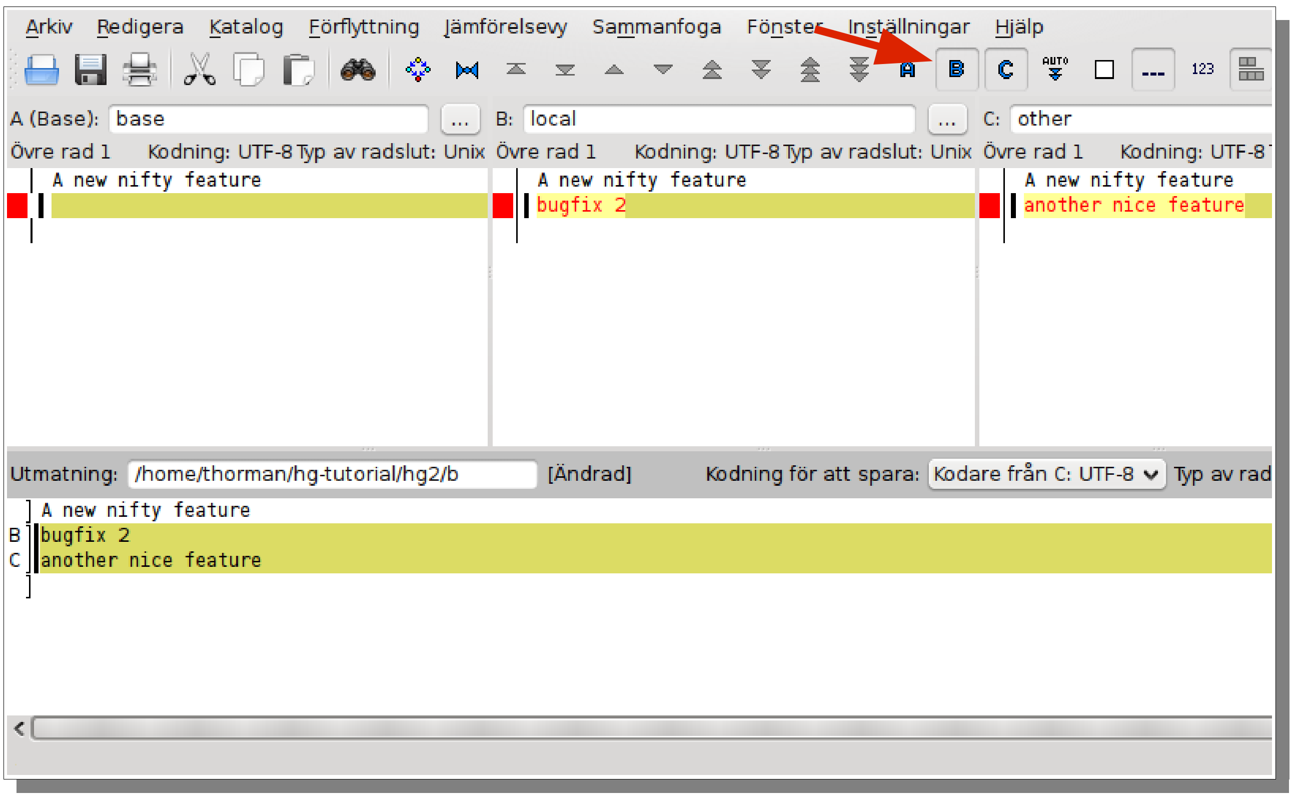
```
A new nifty feature
```

```
A new nifty feature
bugfix 2
```

```
A new nifty feature
another nice feature
```

```
 ] A new nifty feature
?]|<Konflikter vid sammanfogning>
 ]
```

It shows you four panes, three at the top and one at the bottom. The pane to the left (A) shows you the contents of the *base*, the latest common ancestor. Pane B shows you the local changes and C shows you the other, remote, version that you want to merge with. Conflicts are marked with red. B reads “bugfix 2" while C reads “another nice feature". The bottom pane shows you the result of the merge. Say that you would want both the B and C changes. Just press the blue B and the C in the tool bar:


Arkiv Redigera Katalog Förflyttning Jämförelsevy Sammanfoga Fönster Inställningar Hjälp
A (Base): base
B: local
C: other
Övre rad 1 Kodning: UTF-8 Typ av radslut: Unix
Utmatning: /home/thorman/hg-tutorial/hg2/b [Ändrad]
Kodning för att spara: Kodare från C: UTF-8
Typ av rad

```
A new nifty feature
```

```
A new nifty feature
bugfix 2
```

```
A new nifty feature
another nice feature
```

```
 ] A new nifty feature
B |bugfix 2
C |another nice feature
 ]
```

Now we have both the nice feature and the bug fix, as we can see in the bottom pane.

Save, close and commit:

```
...
merging b
1 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg ci -m"Merged bugfix 2 and another nice feature."
```

In real life, the conflicts are more complicated, but the principle is the same.

6.5 Fetch

Do we always have to do `hg pull`, then `hg merge` and finally `hg commit`? No. There's an extension called `fetch` that will do this for us. Edit the ~/.hgrc file and add the line “`fetch=`" to enable this handy extension:

```
[ui]
username = Thorbjorn Jemander

[extensions]
graphlog=
hgext.extdiff =
fetch=

[extdiff]
cmd.kdiff3 =

[merge-tools]
kdiff3.args = $base $local $other -o $output
```

The file ~/.hgrc

Let's try it out:

```
$ cd ../hg1
$ echo "cleanup" > a
$ hg ci -m"Cleaned up module a."
$ hg incoming
comparing with ../hg2
searching for changes
changeset:   22:917ec356d233
user:        Thorbjorn Jemander
date:        Mon Dec 28 14:18:23 2009 +0100
summary:     Bug no 2 fixed in b.

changeset:   25:d7c7adab510e
tag:         tip
parent:      22:917ec356d233
parent:      24:58f714a8f24c
user:        Thorbjorn Jemander
date:        Mon Dec 28 14:22:15 2009 +0100
summary:     Merged bugfix 2 and another nice feature.
$ hg fetch ../hg2
```

```
pulling from ../hg2
searching for changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 2 changes to 1 files (+1 heads)
updating to 26:d7c7adab510e
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
merging with 24:a836f48b2edb
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
new changeset 27:eb831a2a7cef merges remote changes with local
$ cat b
A new nifty feature
bugfix 2
another nice feature
$ hg glog
@    changeset:   27:eb831a2a7cef
|\   tag:         tip
| |  parent:      26:d7c7adab510e
| |  parent:      24:a836f48b2edb
| |  user:        Thorbjorn Jemander
| |  date:        Mon Dec 28 14:29:17 2009 +0100
| |  summary:     Automated merge with file:///home/thorman/hg-tutorial/hg2
| |
| o    changeset:   26:d7c7adab510e
| |\   parent:      25:917ec356d233
| | |  parent:      23:58f714a8f24c
| | |  user:        Thorbjorn Jemander
| | |  date:        Mon Dec 28 14:22:15 2009 +0100
| | |  summary:     Merged bugfix 2 and another nice feature.
| | |
| | o  changeset:   25:917ec356d233
| | |  parent:      22:d8ab303e592a
| | |  user:        Thorbjorn Jemander
| | |  date:        Mon Dec 28 14:18:23 2009 +0100
| | |  summary:     Bug no 2 fixed in b.
...
```

Ugly, hard-to-read history? Make history a straight line, and have your changes at the end of history by *rebasing*. See the Extras chapter.

What we can see above is that the fetch command pulls, merges (if necessary) and updates the local copy with the changes. Very handy. Note the automatic commit message starting with "Automated merge...".

6.6 Unrelated repositories

Mercurial don't allow fetching from unrelated repositories by default, but you can force that behavior. Let's create two unrelated repositories and see what happens:

```
$ mkdir hg3 hg4
$ cd hg3
$ hg init
$ echo a  > a; hg add a; hg ci -m"adding a"
$ echo c1 > c; hg add c; hg ci -m"adding c"
$ cd ../hg4
$ hg init
$ echo b  > b; hg add b; hg ci -m"adding b"
$ echo c2 > c; hg add b; hg ci -m"adding c"
$ hg pull ../hg3
```

```
pulling from ../hg3
searching for changes
abort: repository is unrelated
$ hg pull -f ../hg3            # Use -f to force, Luke.
pulling from ../hg3
searching for changes
warning: repository is unrelated
adding changesets
adding manifests
adding file changes
added 2 changesets with 2 changes to 2 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
$ hg glog
o  changeset:   3:3e236b6d437f
|  tag:         tip
|  user:        Thorbjorn Jemander
|  date:        Mon Dec 28 15:05:15 2009 +0100
|  summary:     adding c
|
o  changeset:   2:1f36da6d72b9
   parent:      -1:000000000000
   user:        Thorbjorn Jemander
   date:        Mon Dec 28 15:03:21 2009 +0100
   summary:     adding a

@  changeset:   1:3d6cdd4bfea2
|  user:        Thorbjorn Jemander
|  date:        Mon Dec 28 15:05:52 2009 +0100
|  summary:     adding c
|
o  changeset:   0:6bb7f0241f12
   user:        Thorbjorn Jemander
   date:        Mon Dec 28 15:05:33 2009 +0100
   summary:     adding b
```

After we forced pull to bring over the changes, we have two heads, one from each repository, but also two changesets without parents (0 and 2). We have two *roots,* which is unorthodox.

Merge:

```
$ hg merge                    # kdiff will appear → merge, save and close.
merging c
1 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg ci -m"merged"
$ glog
@    changeset:   4:b36b10f6d2c8
|\   tag:         tip
| |  parent:      1:3d6cdd4bfea2
| |  parent:      3:3e236b6d437f
| |  user:        Thorbjorn Jemander
| |  date:        Mon Dec 28 15:07:10 2009 +0100
| |  summary:     merged
| |
| o  changeset:   3:3e236b6d437f
| |  user:        Thorbjorn Jemander
| |  date:        Mon Dec 28 15:05:15 2009 +0100
| |  summary:     adding c
| |
| o  changeset:   2:1f36da6d72b9
|    parent:      -1:000000000000
|    user:        Thorbjorn Jemander
|    date:        Mon Dec 28 15:03:21 2009 +0100
|    summary:     adding a
|
o  changeset:   1:3d6cdd4bfea2
|  user:        Thorbjorn Jemander
|  date:        Mon Dec 28 15:05:52 2009 +0100
|  summary:     adding c
|
o  changeset:   0:6bb7f0241f12
   user:        Thorbjorn Jemander
   date:        Mon Dec 28 15:05:33 2009 +0100
   summary:     adding b
```

For the most part you should not need to merge unrelated repositories as it can be tricky to get it right if you have a lot of conflicts. The two trees does not have a common ancestor, and thus no base revision to compare with.

7 Import/Export

In this section we'll see what we can do to interact with the world outside Mercurial.

7.1 Archive

Now and then you need to send source code to someone else, who is not interested in the revision history. The tool for this job is `archive`:

```
$ cd hg1
$ hg archive -t zip ../hg1.zip
$ cd ..
$ ls -l hg1.zip
-rw-r--r--  1 thorman thorman  890 2009-12-28 15:24 hg1.zip
```

You'll get an archive which only contains the source code if the current revision. You can specify several archive types (“files", “tar", “tgz", “zip" etc).

7.2 Export

At other occasions, you don't want to export the full source code, just a patch. A patch is a file describing the changes made to one or more files. To make a patch out of a changeset, use the `export` command:

```
$ cd hg1
$ echo “testing patch-gen" > p
$ hg addr
adding p
$ hg ci -m"added p"
$ hg head
changeset:   28:0d8301efd4b4
tag:         tip
user:        Thorbjorn Jemander
date:        Mon Dec 28 15:39:30 2009 +0100
summary:     adding p
$ hg export -o ../hg1-rev28.diff 28
$ cat ../hg1-rev28.diff
# HG changeset patch
# User Thorbjorn Jemander
# Date 1262011170 -3600
# Node ID 0d8301efd4b47bc4dbed24353a8f21632fe0876a
# Parent  eb831a2a7cefd421aca032af6ccabc7fbefc4e71
adding p

diff -r eb831a2a7cef -r 0d8301efd4b4 p
--- /dev/null  Thu Jan 01 00:00:00 1970 +0000
+++ b/p   Mon Dec 28 15:39:30 2009 +0100
@@ -0,0 +1,1 @@
+testing patch-gen
```

7.3 Import

You can incorporate patches into your repository by using the `import` command:

```
$ cd hg2
$ hg import ../hg1-rev28.diff
applying ../hg1-rev28.diff
$ hg head
changeset:   26:21fdaa10b77d
tag:         tip
user:        Thorbjorn Jemander
date:        Mon Dec 28 15:39:30 2009 +0100
summary:     adding p
$ cat p
testing patch-gen
```

When imported, the patch became changeset 26 in hg2. Do not be confused about that it is different from 28. 28 is the revision the changeset had in hg1. The numbers will be different, in general .

When a patch is imported the local copy is updated and the change is committed automatically (unless the `—no-commit` flag is given).

You cannot import to unrelated repositories.

The import/export feature can be used when you cannot reach the other repository directly. You can for instance send patches over email.

Note that these patches contain information about the repository, the summary, user etc, which is more than the traditional UNIX patches, which we'll discuss in section 8.1 .

7.4 Bundle/Unbundle

The `bundle` and `unbundle` commands are similar to the export/import commands. The power of bundle lies in the ability to specify common ancestors (the `—base` flag) that one knows to exist in the other repository. Mercurial then deduces which changesets that are needed to safely bring over the patch to the other repository.

```
$ cd ../hg1
$ hg log -r 24
changeset:   24:a836f48b2edb
user:        Thorbjorn Jemander
date:        Mon Dec 28 14:28:39 2009 +0100
summary:     Cleaned up module a.
$ hg bundle -r 24 --base 23 ../hg1-bundle
1 changesets found
$ cd ../hg2
$ hg unbundle ../hg1-bundle
adding changesets
adding manifests
adding file changes
added 1 changesets with 2 changes to 2 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
$ hg glog
o  changeset:   27:a836f48b2edb
|  tag:         tip
|  parent:      24:58f714a8f24c
|  user:        Thorbjorn Jemander
|  date:        Mon Dec 28 14:28:39 2009 +0100
|  summary:     Cleaned up module a.
|
| @  changeset:   26:21fdaa10b77d
| |  user:        Thorbjorn Jemander
| |  date:        Mon Dec 28 15:39:30 2009 +0100
| |  summary:     adding p
```

```
...
$ hg merge
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg ci -m"Merged in bundle from hg1."
```

8 Working with patches

In this section, we'll discuss different ways to work with patches. One reason for this discussion is the *outstanding uncommitted changes* (*OUC*) problem mentioned earlier. Patching is a handy way to work around that problem.

The *OUC* problem arises when you try to merge, and have three sources of changes in a single file: remote repository changes, local committed changes and local uncommitted changes.

It is the combination of these three that causes the problem. If you can reduce it down to two sources, you're safe. Either

1. Commit **before** the merge (and thus remove the local changes)
2. Do not commit **any** changes before merging (and thus have no local committed changes)

However, sometimes it is not obvious that there will be a problem when you merge and you may need to resolve it *after* the situation has occurred. The solution to this is lift off your local changes into a patch, merge, and then reapply the local changes.

I'll mention two ways to work with patches: UNIX *patches* and *patch queues*. *UNIX patches* is not a Mercurial concept, but still handy and integrates well with Mercurial.

8.1 UNIX patches

Let's create two repositories and create the *OUC* situation.

```
$ mkdir hg5
$ cd hg5
$ hg init
$ for i in {1..13}; do echo "row $i" >> a; done
$ cat a
row 1
row 2
row 3
row 4
row 5
row 6
row 7
row 8
row 9
row 10
row 11
row 12
row 13
$ hg addr
adding a
$ hg ci -m"added a"
$ cd ..
$ hg clone hg5 hg6
updating working directory
1 files updated, 0 files merged, 0 files removed,
0 files unresolved
$ cd hg5
$ gedit a
```

```
row 1
row 2
row 3 foo
row 4
row 5
row 6
row 7
row 8
row 9
row 10
row 11
row 12
row 13
```

The file hg5/a

In the editor, add the string “foo" to row 3 of the file a.

Commit the change and move to hg6:

```
$ hg ci -m"added foo to row 3"
$ cd ../hg6
$ gedit a
```

```
row 1
row 2
row 3
row 4
row 5
row 6
row 7
row 8 bar
row 9
row 10
row 11
row 12
row 13
```

The file hg6/a (I)

Add the string “bar" to row 8 of *a* in hg6 and commit the change.

```
$ hg ci -m"added bar to row 8"
$ gedit a
```

Edit the file *a* again and add “local change" to row 12, save, but do not commit.

```
row 1
row 2
row 3
row 4
row 5
row 6
row 7
row 8 bar
row 9
row 10
row 11
row 12 local change
row 13
```

The file hg6/a (II)

Now, try to pull in the hg5 change, and merge:

```
$ hg fetch
abort: outstanding uncommitted changes
```

Remove your local changes by generating a patch and revert:

```
$ hg diff > /tmp/mypatch.diff
$ hg revert -a      # -a reverts the whole repository
reverting a
$ hg fetch
pulling from /home/thorman/hg-tutorial/hg5
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
updating to 2:655659cc91c8
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
merging with 1:66fffbe5c693
merging a
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
new changeset 3:ab4970978198 merges remote changes with local
$ cat a
row 1
row 2
row 3 foo
row 4
row 5
row 6
row 7
row 8 bar
row 9
row 10
row 11
row 12
row 13
```

Bring back our changes:

```
$ patch < /tmp/mypatch.diff
$ cat a          # Let's look at a now
row 1
row 2
row 3 foo
row 4
row 5
row 6
row 7
row 8 bar
row 9
row 10
row 11
row 12 local change
row 13
$ hg diff     # Check our local changes...
diff -r ab4970978198 a
--- a/a   Wed Dec 30 08:32:05 2009 +0100
+++ b/a   Wed Dec 30 08:35:45 2009 +0100
@@ -9,5 +9,5 @@
 row 9
 row 10
 row 11
-row 12
+row 12 local change      # The changes are not committed, good.
 row 13
```

Done. What we did was to generate a diff by using `hg diff` and then use the UNIX command `patch` to bring back the changes. Next up is *patch queues*, a more systematic way to work with patches.

8.2 Patch queues

To work with patch queues, first enable the mq extension:

```
[ui]
username = Thorbjorn Jemander

[extensions]
graphlog=
hgext.extdiff =
fetch=
hgext.mq=

[extdiff]
cmd.kdiff3 =

[merge-tools]
kdiff3.args = $base $local $other -o $output
```

The file ~/.hgrc

In the following, we'll show how to use patch queues to solve the above problem. Trigger the same situation as above by changing the *a* file in hg5, and try to pull in the changes to hg6:

```
$ cd ../hg5
$ echo “row 14" >> a
$ hg ci -m"added row 14"
$ cd ../hg6
$ hg fetch
abort: outstanding uncommitted changes
```

8.2.1 qinit

To work with patch queues, we need to initialize the repository first:

```
$ hg qinit
```

8.2.2 qnew

We can create a new patch by using qnew. The `-f` flag takes the local changes, and turns them into a patch. We use this to lift the local, working copy changes to into a patch with name “myfeature".

```
$ hg qnew -f myfeature
```

8.2.3 qseries and qapplied

There are two sets of patches that we work with: the set of all patches and the set of the applied patches. `Qseries` shows you the set of all patches and `qapplied` the set of applied patches:

```
$ hg qseries
myfeature                           # Underline means applied.
$ hg qapplied
myfeature
$ hg st
$ hg log
changeset:   4:5fbc699a1bb1
tag:         qtip
tag:         tip
tag:         myfeature
tag:         qbase
user:        Thorbjorn Jemander
date:        Thu Dec 31 11:03:05 2009 +0100
summary:     [mq]: myfeature

changeset:   3:ab4970978198
tag:         qparent
parent:      2:655659cc91c8
parent:      1:66fffbe5c693
user:        Thorbjorn Jemander
date:        Wed Dec 30 08:32:05 2009 +0100
summary:     Automated merge with file:///home/thorman/hg-tutorial/hg5
...
```

So, the `qnew` patch command apparently took the local changes and turned it into a patch called 'myfeature', which is now applied as changeset 4.

8.2.4 qpop

Weren't we supposed to remove the patch? Yes, we do that now, by using the `qpop` command:

```
$ hg qpop
patch queue now empty
```

```
$ hg qseries
myfeature
$ hg qapplied
$ hg log
changeset:   3:ab4970978198
tag:         qparent
parent:      2:655659cc91c8
parent:      1:66fffbe5c693
user:        Thorbjorn Jemander
date:        Wed Dec 30 08:32:05 2009 +0100
summary:     Automated merge with file:///home/thorman/hg-tutorial/hg5
...
```

Now we have “popped the patch", i.e. it is not applied anymore, so that we can fetch the other stuff:

```
$ hg fetch ../hg5
pulling from ../hg5
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
updating to 4:05b27b8704b0
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
merging with 3:ab4970978198
merging a
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
new changeset 5:6209c396169f merges remote changes with local
```

Now we can import our patch again:

```
$ hg import --no-commit .hg/patches/myfeature
applying .hg/patches/myfeature
$ hg diff
diff -r 6209c396169f a
--- a/a   Thu Dec 31 11:05:33 2009 +0100
+++ b/a   Thu Dec 31 11:08:25 2009 +0100
@@ -9,6 +9,6 @@
 row 9
 row 10
 row 11
-row 12
+row 12 local change
 row 13
 row 14
```

We're in the same situation as after using the UNIX patches above: our local changes are not committed and thus reside in working copy only (because of the `--no-commit` flag).

However, this not not precisely how one is supposed to work with patch queues. The way we use the `import` command above is not exactly elegant, since it requires knowledge that the patches are stored in .hg/patches (which may change) and we are kind of by-passing the patch queue.

We will now use the patch queues to solve the above problem, in a more proper way.

8.2.5 qpush

First, create the *OUC* problem again and delete the old patch:

```
$ cd ../hg5
$ echo “row 15" >> a
$ hg ci -m"added row 15"
$ cd ../hg6
$ hg qdelete myfeature            # delete the patch from the example above.
$ hg fetch
abort: outstanding uncommitted changes
```

Extract the local changes into a patch, pop it off the stack, fetch the changes and push back the patch:

```
$ hg qnew -f myfeature2
$ hg qpop
patch queue now empty
$ hg fetch
pulling from /home/thorman/hg-tutorial/hg5
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
updating to 6:868c5031ed0b
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
merging with 5:6209c396169f
merging a
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
new changeset 7:d44050773789 merges remote changes with local
$ hg qpush
applying myfeature2
now at: myfeature2
$ hg qapplied
myfeature2
$ hg log
changeset:   8:58197e6ac8e3
tag:         qtip
tag:         myfeature2
tag:         tip
tag:         qbase
user:        Thorbjorn Jemander
date:        Thu Dec 31 11:25:00 2009 +0100
summary:     imported patch myfeature2
...
```

So the proper solution to *OUC* is: `qnew`, `qpop`, `fetch` and `qpush`.

The difference between `import` method we did above, and the `qpush` method we did here, is that now is the patch back and is applied (so that it's at the top of the log). One could argue that this is not what one would want, someone might think that the local changes are not ready for check-in. Don't worry, let's continue and see how we can work with patches to make a nice commit in the end.

8.2.6 qrefresh and qdiff

Say that we're not ready with our local changes. No problem. We can add changes to the a patch:

```
$ echo “row 16" >> a
$ hg qrefresh                         # Local changes added to the patch.
$ hg diff                             # No outstanding changes,
$ hg qdiff                            # they're all in the patch:
diff -r d44050773789 a
--- a/a   Thu Dec 31 11:24:44 2009 +0100
+++ b/a   Thu Dec 31 12:16:58 2009 +0100
@@ -9,7 +9,8 @@
 row 9
 row 10
 row 11
-row 12
+row 12 local change
 row 13
 row 14
 row 15
+row 16
```

Qdiff shows the changes of the current patch. If there are local changes too, qdiff shows the local changes AND the changes in the current patch (i.e. showing what the current patch would become after a qrefresh).

The qrefresh command absorbs the local changes into the current patch. However, you have not committed the patch to history for real yet. We'll do that in a moment, first we'll create some more patches.

8.2.7 Multiple patches

You can continue to work by absorbing changes into the current patch, but you can also choose to create new patches. It could be that the current patch is relatively stable, and you want to make some experimental changes and you're not sure if you want to keep them or not:

```
$ hg qnew experimental
$ echo "row 17: experimental" >> a
$ hg diff
diff -r 72ed90212e43 a
--- a/a   Thu Dec 31 11:52:53 2009 +0100
+++ b/a   Thu Dec 31 11:53:02 2009 +0100
@@ -14,3 +14,4 @@
 row 14
 row 15
 row 16
+row 17: experimental
$ hg qrefresh                                # Absorb local changes.
$ hg diff                                    # No working copy changes.
$ hg qdiff                                   # The changes are in the patch:
diff -r ab51ff1f1b56 a
--- a/a   Thu Dec 31 11:34:13 2009 +0100
+++ b/a   Thu Dec 31 11:53:05 2009 +0100
@@ -14,3 +14,4 @@
 row 14
 row 15
 row 16
+row 17: experimental
```

8.2.8 Moving between patches: qgoto

So far, we've moved between patches by using `qpop` and `qpush`. There's also `qgoto`:

```
$ hg qseries
myfeature2
experimental
$ hg qapplied
myfeature2
experimental
$ hg qpop                                 # Remove the top-most applied patch
now at: myfeature2
$ hg qapplied
myfeature2
$ cat a
...
row 11
row 12 local change
row 13
row 14
row 15
row 16                                    # No experimental 17 row
$ hg qpop
patch queue now empty
$ cat a
...
row 11
row 12                                    # No “local changes"
row 13
row 14
row 15                                    # No row 16 (in myfeature2).
$ hg qgoto experimental                   # qgoto applies/removes patches as
applying myfeature2                       # necessary to reach the destination.
applying experimental
now at: experimental
```

8.2.9 Committing patches: qfold and qfinish

If you look at the commit log, you can see that the applied patches are visible as regular changesets at the top of the history, with some extra tagging, `qbase`, `qtip` and the patch names, and generic commit messages starting with “imported patch":

```
$ hg log
changeset:    9:b3a51e5dab2e
tag:          qtip
tag:          tip
tag:          experimental
user:         Thorbjorn Jemander
date:         Thu Dec 31 12:00:02 2009 +0100
summary:      imported patch experimental

changeset:    8:4a97313129b4
tag:          qbase
tag:          myfeature2
user:         Thorbjorn Jemander
date:         Thu Dec 31 12:00:02 2009 +0100
summary:      imported patch myfeature2

changeset:    7:d44050773789
tag:          qparent
```

```
parent:      6:868c5031ed0b
parent:      5:6209c396169f
user:        Thorbjorn Jemander
date:        Thu Dec 31 11:24:44 2009 +0100
summary:     Automated merge with file:///home/thorman/hg-tutorial/hg5
...
```

This is not how we want it to look in the product repository. We would like to have proper commit messages, and maybe we want a single changeset and also have possibility review all the changes that we done before we commit – even if we want to commit several patches as one changeset.

Before we do that, let's create a third patch, *experimental2*, and then move to patch *experimental*:

```
$ hg qnew experimental2
$ echo b > b
$ hg add b                  # Yes, add/remove/addremove/rename work with patches
$ hg qrefresh
$ hg diff
$ hg qgoto myfeature2
popping experimental2
popping experimental
now at: myfeature2
$ hg qseries
myfeature2
experimental
experimental2
```

Now, the time has come to submit *myfeature2* and *experimental* to the product repository, and they should be committed as a single changeset. First, fold the two patches together:

```
$ hg qfold experimental
$ hg qseries
myfeature2                  # Patch experimental gone, merged into myfeature2.
experimental2
$ hg log
changeset:   8:cff35e37e52a
tag:         qtip
tag:         myfeature2
tag:         tip
tag:         qbase
user:        Thorbjorn Jemander
date:        Thu Dec 31 12:46:26 2009 +0100
summary:     [mq]: myfeature2
...
```

Bah. We forgot to use the `-m` flag to `qfold` so that we still have this boring "[mq]: myfeature2" summary. Do not despair, we can update the commit message, using `qrefresh`:

```
$ hg qrefresh -m"Implemented my feature no 2"
$ hg head
changeset:   8:42905ba00dba
tag:         qtip
tag:         myfeature2
tag:         tip
tag:         qbase
user:        Thorbjorn Jemander
```

```
date:          Thu Dec 31 12:52:49 2009 +0100
summary:       Implemented my feature no 2
```

That's better. Before we commit, we may want to review the changes:

```
$ hg qdiff
diff -r d44050773789 a
--- a/a   Thu Dec 31 11:24:44 2009 +0100
+++ b/a   Thu Dec 31 12:53:45 2009 +0100
@@ -9,7 +9,9 @@
 row 9
 row 10
 row 11
-row 12
+row 12 local change
 row 13
 row 14
 row 15
+row 16
+row 17: experimental
```

Good, it contains the changes from both the *myfeature2* patch and the old *experimental* patch. Commit the patch by using `qfinish`:

```
$ hg qfinish 8                          # 8 = revision 8.
$ hg log
changeset:     8:42905ba00dba
tag:           tip
user:          Thorbjorn Jemander
date:          Thu Dec 31 12:52:49 2009 +0100
summary:       Implemented my feature no 2
...
```

You have to specify a revision to qfinish. There's an error in the help text – it says that the revision is optional, but it isn't. You can use "tip" most of the time, or -a if you want to finish all appled patches.

Now we're finished: a single commit containing the all the changes we've done.

I find patch queues very handy to work with. However, when you work with patches and encounter conflicts, things are a little bit different from the conflict management described above. You may experience that patch is *rejected*.

8.3 Rejected patches

Lets create situation where the patch fails to apply:

```
$ gedit a
```

```
row 1
row 2 change #1
row 3 foo
row 4
row 5
row 6
row 7
row 8 bar
row 9
row 10
row 11
row 12 local change
row 13
row 14 change #2
row 15
row 16
row 17: experimental
```

The file hg6/a (I)

Make two changes to the file *a*, one at row 2 and the other at row 14, as the file to the right shows. Generate a UNIX patch:

```
$ hg diff > /tmp/mypatch2.diff
```

Now, revert the file and edit a again, and change row 4 as shown in the file. Save the file and apply our `mypatch2`:

```
$ hg revert a
$ gedit a
$ patch < /tmp/mypatch2.diff
patching file a
Hunk #2 FAILED at 11.
1 out of 2 hunks FAILED -- saving rejects to file a.rej
```

```
row 1
row 2
row 3 foo
row 4
row 5
row 6
row 7
row 8 bar
row 9
row 10
row 11
row 12 local change
row 13
row 14 other change
row 15
row 16
row 17: experimental
```

The file hg6/a (II)

The patch failed for some reason, and what are those *hunks* it is referring to? Take a look at the patch file:

```
$ cat /tmp/mypatch2.diff
diff -r 6b75f218fcc1 a
--- a/a Fri Jan 08 17:38:15 2010 +0100
+++ b/a Fri Jan 08 17:52:03 2010 +0100
@@ -1,5 +1,5 @@           ← Line number information
 row 1
-row 2
+row 2 change #1              Hunk #1
 row 3 foo
 row 4
 row 5
@@ -11,7 +11,7 @@         ← Line number information
 row 11
 row 12 local change
 row 13
-row 14                       Hunk #2
+row 14 change #2
 row 15
 row 16
 row 17: experimental
```

The patch file is a sequence of hunks. Each hunk in the above example has line number and context information. The context information is a couple of lines before and after the change itself to describe what the file looked like when the diff was generated.

When patch command applies the patch it finds the place for making the change by using both the context and line number information. That allows code to move around and one still can patch files which are not identical to the one it was originally “diffed” against.

However, when the patch command cannot find a place in the file matching the context of the hunk, it doesn't know what to do. In our case, we have changed row 14 locally, so patch gives up and *rejects the hunk*. The rejects are dumped as “<file>.rej” files, as you can see in the error message above, we have a “a.rej” file. Take a look at it:

```
$ cat a.rej
***************
*** 11,17 ****
  row 11
  row 12 local change
  row 13
- row 14
  row 15
  row 16
  row 17: experimental
--- 11,17 ----
  row 11
  row 12 local change
  row 13
+ row 14 change #2
  row 15
  row 16
  row 17: experimental
```

Here we can see what it tried to do: remove “row 14” and add “row 14 change #2”. Given this information, it makes it easy for us to analyze the situation and decide what to do. In this particular case, it comes down to: what should row 14 look like, should it contain “change #2” as the patch says, or should it contain “other change” that someone else wants. It requires some manual assistance.

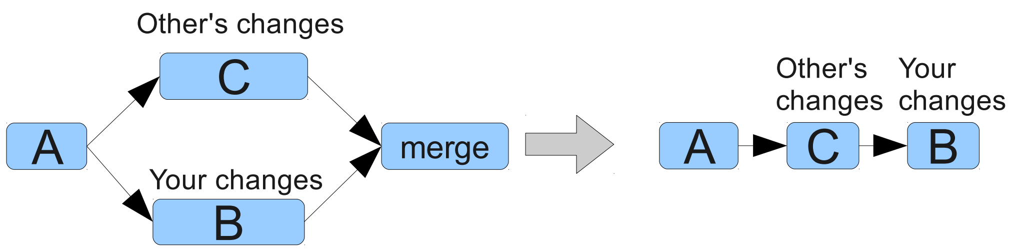
If you encounter failed patches, just open the “<file>.rej” files and look what patch tried to do, and then change the corresponding <file> file.

Patch rejection can occur not only when you work with UNIX patches, but also if you work with other patch-based tools. For example, if an extension to Mercurial is patch-based, you can expect patch rejections instead of regular conflicts. Unfortunately, there's no nice graphical tool for managing patch rejection as there is for regular conflicts.

9 Extras

9.1 Rebase

This is about straighten out the history and putting your changes last.



Usually, when you work in parallel with other people, you would get the situation to the left in the figure: other people's changes (C) would appear in parallel with your changes (B). With rebasing, you can transform this to the situation to the right in the figure and remove the need for merging.

To enable this extension, add `rebase=` to your ~/.hgrc file:

```
[ui]
username = Thorbjorn Jemander

[extensions]
graphlog=
hgext.extdiff =
fetch=
hgext.mq=
rebase=

[extdiff]
cmd.kdiff3 =

[merge-tools]
kdiff3.args = $base $local $other -o $output
```

The file ~/.hgrc

Make some history, clone the repository, make local changes, commit, and pull in more changes:

```
$ mkdir hg7
$ cd hg7
$ hg init
$ echo a > a; hg add a; hg ci -ma
$ cd ..
$ hg clone hg7 hg8
updating to branch default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd hg8
$ echo b > b; hg add b; hg ci -mb
$ cd ../hg7
$ echo c > c; hg add c; hg ci -mc
$ cd ../hg8
$ hg pull
pulling from /home/thorman/hg-tutorial/hg7
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
$ hg merge
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg ci -m"merged"
$ hg glog
@    changeset:   3:ab66109a53ab
|\   tag:         tip
| |  parent:      1:9fe6f317436f
| |  parent:      2:875c5bcfa0a2
| |  user:        Thorbjorn Jemander
| |  date:        Fri Jan 01 09:46:14 2010 +0100
| |  summary:     merged b c
| |
| o  changeset:   2:875c5bcfa0a2
| |  parent:      0:c054b2eb6a95
| |  user:        Thorbjorn Jemander
| |  date:        Fri Jan 01 09:43:14 2010 +0100
| |  summary:     c
| |
o |  changeset:   1:9fe6f317436f
|/   user:        Thorbjorn Jemander
|    date:        Fri Jan 01 09:42:58 2010 +0100
|    summary:     b
|
o  changeset:   0:c054b2eb6a95
   user:        Thorbjorn Jemander
   date:        Fri Jan 01 09:42:19 2010 +0100
   summary:     a
```

Now we have a parallel history, requiring a merge. I want my changes to float on top, to be the latest. With *rebasing*, I can move changes in the tree. There is a *source* and a *destination*. In this case, I want to move my change b (changeset 1), which is the source, on top of the remote change *c* (changeset 2), which is the destination:

```
$ rebase -s 1 -d 2                       # -s = --source, -d = --dest
saving bundle to /home/thorman/hg-tutorial/hg8/.hg/strip-backup/9fe6f317436f-
temp
adding branch
adding changesets
adding manifests
adding file changes
added 2 changesets with 2 changes to 2 files
rebase completed
$ hg glog
@  changeset:   2:35252c967029
|  tag:         tip
|  user:        Thorbjorn Jemander
|  date:        Fri Jan 01 09:42:58 2010 +0100
|  summary:     b
|
o  changeset:   1:875c5bcfa0a2
|  user:        Thorbjorn Jemander
|  date:        Fri Jan 01 09:43:14 2010 +0100
|  summary:     c
|
o  changeset:   0:c054b2eb6a95
   user:        Thorbjorn Jemander
   date:        Fri Jan 01 09:42:19 2010 +0100
   summary:     a
```

Isn't that beautiful? Note how the merge magically disappeared. The need for a merge disappeared when we based *b* on *c*, instead of *a*.

This extension also provides an extra option to pull, `--rebase`, that is handy. Let's try it out:

```
$ cd ../hg7
$ echo d > d
$ hg add d
$ hg ci -md
$ cd ../hg8
$ hg pull --rebase
pulling from /home/thorman/hg-tutorial/hg7
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
saving bundle to /home/thorman/hg-tutorial/hg8/.hg/strip-backup/35252c967029-
temp
adding branch
adding changesets
adding manifests
adding file changes
added 2 changesets with 2 changes to 2 files
rebase completed
$ hg glog
@  changeset:   3:3591d86d1dfd
|  tag:         tip
|  user:        Thorbjorn Jemander
|  date:        Fri Jan 01 09:42:58 2010 +0100
|  summary:     b
|
o  changeset:   2:06b50be3693a
|  user:        Thorbjorn Jemander
|  date:        Fri Jan 01 09:54:24 2010 +0100
|  summary:     d
|
o  changeset:   1:875c5bcfa0a2
|  user:        Thorbjorn Jemander
|  date:        Fri Jan 01 09:43:14 2010 +0100
|  summary:     c
|
o  changeset:   0:c054b2eb6a95
   user:        Thorbjorn Jemander
   date:        Fri Jan 01 09:42:19 2010 +0100
   summary:     a
```

Now it automatically moved *d before* my change *b*, so *b* is still last in history.

You can read more about rebasing here: http://mercurial.selenic.com/wiki/RebaseProject

9.2 More undo

9.2.1 Backout

If we take the hg8 repository from above and want to undo, "back out", a changeset which does not reside at the tip, e.g. changeset 1 (c), we can do:

```
$ cd hg8
$ hg backout -r 1 -m"Backed out c"
removing c
created new head
changeset 4:5d9c49b4e432 backs out changeset 1:875c5bcfa0a2
the backout changeset is a new head - do not forget to merge
(use "backout --merge" if you want to auto-merge)
$ hg glog
@  changeset:   4:894e428a2855
|  tag:         tip
|  parent:      1:875c5bcfa0a2
|  user:        Thorbjorn Jemander
|  date:        Fri Jan 01 22:53:23 2010 +0100
|  summary:     Backed out c
|
| o  changeset:   3:3591d86d1dfd
| |  user:        Thorbjorn Jemander
| |  date:        Fri Jan 01 09:42:58 2010 +0100
| |  summary:     b
| |
| o  changeset:   2:06b50be3693a
|/   user:        Thorbjorn Jemander
|    date:        Fri Jan 01 09:54:24 2010 +0100
|    summary:     d
|
o  changeset:   1:875c5bcfa0a2
|  user:        Thorbjorn Jemander
|  date:        Fri Jan 01 09:43:14 2010 +0100
|  summary:     c
|
o  changeset:   0:c054b2eb6a95
   user:        Thorbjorn Jemander
   date:        Fri Jan 01 09:42:19 2010 +0100
   summary:     a
$ hg merge
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg ci -m"merged in c removal""
$ ls
a b d                          # No c.
```

We removed *c* by applying a "negative patch" just after that changeset to be removed. (That's why we got an extra head.)

If you work with patch queues, you can do this without getting extra heads, if the changeset you intend to remove is actually a patch. Just un-apply it by popping patches, delete it, and push back the patches you popped. This is left as an exercise for the reader.

9.2.2 Clone

Sometimes, when things go very wrong, revert, rollback and backout are just not enough. We can remove mistakes by cloning our repository from a specific point in time, and replace the old repository with the new. Assume everything after changeset 2 *(d)* is really bad:

```
$ hg glog
@    changeset:   5:607d22c7c3c8
|\   tag:         tip
| |  parent:      4:894e428a2855
| |  parent:      3:3591d86d1dfd
| |  user:        Thorbjorn Jemander
| |  date:        Fri Jan 01 22:55:32 2010 +0100
| |  summary:     merged in c removal
| |
| o  changeset:   4:894e428a2855
| |  parent:      1:875c5bcfa0a2
| |  user:        Thorbjorn Jemander
| |  date:        Fri Jan 01 22:53:23 2010 +0100
| |  summary:     Backed out c
| |
o |  changeset:   3:3591d86d1dfd
| |  user:        Thorbjorn Jemander
| |  date:        Fri Jan 01 09:42:58 2010 +0100
| |  summary:     b
| |
o |  changeset:   2:06b50be3693a
|/   user:        Thorbjorn Jemander
|    date:        Fri Jan 01 09:54:24 2010 +0100
|    summary:     d
|
o  changeset:   1:875c5bcfa0a2
|  user:        Thorbjorn Jemander
|  date:        Fri Jan 01 09:43:14 2010 +0100
|  summary:     c
|
o  changeset:   0:c054b2eb6a95
   user:        Thorbjorn Jemander
   date:        Fri Jan 01 09:42:19 2010 +0100
   summary:     a
$ cd ..
$ hg clone -r 2 hg8 hg8-new
requesting all changes
adding changesets
adding manifests
adding file changes
added 3 changesets with 3 changes to 3 files
updating to branch default
3 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ mv hg8 hg8-old
$ mv hg8-new hg8
$ cd hg8
$ hg glog
@  changeset:   2:06b50be3693a
|  tag:         tip
|  user:        Thorbjorn Jemander
|  date:        Fri Jan 01 09:54:24 2010 +0100
|  summary:     d
|
o  changeset:   1:875c5bcfa0a2
```

```
|   user:         Thorbjorn Jemander
|   date:         Fri Jan 01 09:43:14 2010 +0100
|   summary:      c
|
o   changeset:    0:c054b2eb6a95
    user:         Thorbjorn Jemander
    date:         Fri Jan 01 09:42:19 2010 +0100
    summary:      a
$ cat .hg/hgrc
[paths]
default = /home/thorman/hg-tutorial/hg8
```

Note that since we have made a clone, the new copy has a reference to its parent, which point to itself after the rename.

You may need to update the default path to correspond to the repository you want.

A word of warning: when you clone a repository, only the recorded history is cloned. Changes that are in the working copy only are not kept, and the same goes for the patch queue information. The patch queue information is lost during the cloning process and the applied patches are turned into regular changesets. The unapplied patches are completely forgotten and have to be moved manually over to the new repository if one wants to keep them.

9.2.3 Strip

If you have installed the mq extension, you have access to the command `strip`, which is a quick way to cut away any portion of the history. Lets take the hg3 repository, used above in the “unrelated repositories" example:

```
$ cd hg3
$ hg up -C                            # Just cleaning local changes..
$ echo c3 >> C
$ hg ci -m”version 3 of c.”
$ hg glog
@   changeset:    2:e66fd480701c
|   tag:          tip
|   user:         Thorbjorn Jemander
|   date:         Mon Dec 28 17:10:32 2009 +0100
|   summary:      version 3 of c.
|
o   changeset:    1:3e236b6d437f
|   user:         Thorbjorn Jemander
|   date:         Mon Dec 28 15:05:15 2009 +0100
|   summary:      adding c
|
o   changeset:    0:1f36da6d72b9
    user:         Thorbjorn Jemander
    date:         Mon Dec 28 15:03:21 2009 +0100
    summary:      adding a
$ hg strip 1
0 files updated, 0 files merged, 1 files removed, 0 files unresolved
saving bundle to /home/thorman/hg-tutorial/hg3/.hg/strip-backup/3e236b6d437f-
backup
$ hg glog
@   changeset:    0:1f36da6d72b9
    tag:          tip
    user:         Thorbjorn Jemander
    date:         Mon Dec 28 15:03:21 2009 +0100
    summary:      adding a
```

Strip removed the specified revision and all descendants. Note that this does not mean that it removes all later revisions, just the revision that has the specified revision as an ancestor. It can be used to cut away unwanted heads, for example. Let's try that by making two heads;

```
$ echo a2 >> a; hg ci -ma2
$ echo a3 >> a; hg ci -ma3
$ echo a4 >> a; hg ci -ma4
$ hg up 1
$ echo A1 >> a; hg ci -mA1
created new head
$ echo A2 >> a; hg ci -mA2
$ hg glog
@  changeset:   5:6464ba3236fe
|  tag:         tip
|  user:        Thorbjorn Jemander
|  date:        Sat Jan 09 07:36:47 2010 +0100
|  summary:     A2
|
o  changeset:   4:4a63d6f5a351
|  parent:      1:990b75b71265
|  user:        Thorbjorn Jemander
|  date:        Sat Jan 09 07:36:29 2010 +0100
|  summary:     A1
|
| o  changeset:   3:07003b668c27
| |  user:        Thorbjorn Jemander
| |  date:        Sat Jan 09 07:36:21 2010 +0100
| |  summary:     a4
| |
| o  changeset:   2:a9ea0e42df43
|/   user:        Thorbjorn Jemander
|    date:        Sat Jan 09 07:36:17 2010 +0100
|    summary:     a3
|
o  changeset:   1:990b75b71265
|  user:        Thorbjorn Jemander
|  date:        Sat Jan 09 07:36:13 2010 +0100
|  summary:     a2
|
o  changeset:   0:1f36da6d72b9
   user:        Thorbjorn Jemander
   date:        Mon Dec 28 15:03:21 2009 +0100
   summary:     adding a
```

Assume now you change your mind and don't want the a3 and a4 changes, so you don't want to merge. On the other hand, you don't want to keep them in there either, because there are two heads. You could use *backout* to erase them and then merge, but that's not very practical. Strip comes to the rescue. Just say that you want to strip revision 2 and its decendands:

```
$ hg strip 2
saving bundle to /home/thorman/hg-tutorial/chapter-repos/hg3-9.2.3/.hg/strip-
backup/a9ea0e42df43-backup
saving bundle to /home/thorman/hg-tutorial/chapter-repos/hg3-9.2.3/.hg/strip-
backup/a9ea0e42df43-temp
adding branch
adding changesets
adding manifests
adding file changes
added 2 changesets with 2 changes to 1 files
$ hg glog
@  changeset:   3:6464ba3236fe
|  tag:         tip
|  user:        Thorbjorn Jemander
|  date:        Sat Jan 09 07:36:47 2010 +0100
|  summary:     A2
|
o  changeset:   2:4a63d6f5a351
|  user:        Thorbjorn Jemander
|  date:        Sat Jan 09 07:36:29 2010 +0100
|  summary:     A1
|
o  changeset:   1:990b75b71265
|  user:        Thorbjorn Jemander
|  date:        Sat Jan 09 07:36:13 2010 +0100
|  summary:     a2
|
o  changeset:   0:1f36da6d72b9
   user:        Thorbjorn Jemander
   date:        Mon Dec 28 15:03:21 2009 +0100
   summary:     adding a
```

Those changes are now gone, but the later changes A1 and A2 are kept, and we have just one head again.

9.3 Transplant aka Cherry picking

Enable this extension by adding transplant= to your ~/.hgrc file:

```
[ui]
username = Thorbjorn Jemander

[extensions]
graphlog=
hgext.extdiff =
fetch=
hgext.mq=
rebase=
transplant=

[extdiff]
cmd.kdiff3 =

[merge-tools]
kdiff3.args = $base $local $other -o $output
```

The file ~/.hgrc

Construct a branch b1, containing changes *b* and *c*:

```
$ mkdir hg9; cd hg9; hg init
$ echo a > a; hg add a; hg ci -ma
$ hg branch b1
marked working directory as branch b1
```

```
$ echo b > b; hg add b; hg ci -mb
$ echo c > c; hg add c; hg ci -mc
$ hg up default
$ echo d > d; hg add d; hg ci -md
created new head
$ hg glog
@  changeset:   3:8119abef4e08
|  tag:         tip
|  parent:      0:b1c552afb583
|  user:        Thorbjorn Jemander
|  date:        Sat Jan 02 06:36:58 2010 +0100
|  summary:     d
|
| o  changeset:   2:67cada58a4ba
| |  branch:      b1
| |  user:        Thorbjorn Jemander
| |  date:        Sat Jan 02 06:36:49 2010 +0100
| |  summary:     c
| |
| o  changeset:   1:7f057f35689a
|/   branch:      b1
|    user:        Thorbjorn Jemander
|    date:        Sat Jan 02 06:36:45 2010 +0100
|    summary:     b
|
o  changeset:   0:b1c552afb583
   user:        Thorbjorn Jemander
   date:        Sat Jan 02 06:33:12 2010 +0100
   summary:     a
```

How would one go about to bring in the changes from the *b1* branch to the default branch? One would merge. But what if we only want changeset 1 *(b)?* Use `transplant`:

```
$ hg transplant -b b1 1
applying 7f057f35689a
7f057f35689a transplanted to 35ec31cb6706
$ hg glog
@  changeset:   4:35ec31cb6706
|  tag:         tip
|  user:        Thorbjorn Jemander
|  date:        Sat Jan 02 06:36:45 2010 +0100
|  summary:     b
|
o  changeset:   3:8119abef4e08
|  parent:      0:b1c552afb583
|  user:        Thorbjorn Jemander
|  date:        Sat Jan 02 06:36:58 2010 +0100
|  summary:     d
|
| o  changeset:   2:67cada58a4ba
| |  branch:      b1
| |  user:        Thorbjorn Jemander
| |  date:        Sat Jan 02 06:36:49 2010 +0100
| |  summary:     c
| |
| o  changeset:   1:7f057f35689a
|/   branch:      b1
|    user:        Thorbjorn Jemander
|    date:        Sat Jan 02 06:36:45 2010 +0100
|    summary:      b
|
o  changeset:   0:b1c552afb583
   user:        Thorbjorn Jemander
   date:        Sat Jan 02 06:33:12 2010 +0100
   summary:     a
```

You can also pick specific revisions from other repositories by using the `-s` switch instead of the branch switch `-b`.

9.4 Setting up a server

Go to the repository you want to share. Start a server:

```
$ cd hg1
$ hg serve                   # Use the -d flag to daemonize
```

Open the URL http://localhost:8000 in a web browser and you'll get a web interface for examining history, tags, branches etc.

You can access this repository from hg by using the same URL:

```
$ cd ..
$ hg clone http://localhost:8000 my_cloned_repo
```

9.5 Tracking external software

Sometimes third party components are used, and they may require some customization to suit local needs. How do you track external software and still keep the local customization? We want to:

1. import new releases when we wish, and
2. let our changes be easily applied for the newly imported release.

You can do this with little effort by using Mercurial. Either you use two repositories or a single, with an import branch.

9.5.1 Using two repositories

Expat, which we use in these examples, is an XML parsing library.

Create a repository and import an external release, clone it and customize the clone:

```
$ mkdir expat-import
$ wget http://sourceforge.net/projects/expat/files/expat/1.95.0/expat-
1.95.0.tar.gz/download
$ tar xvfz expat-1.95.0.tar.gz
...
$ cp -ar expat-1.95.0/* expat-import
$ cd expat-import
$ hg init
$ hg addremove
...
```

```
$ hg ci -m"First check-in"
$ hg tag 1.95.0
$ cd ..
$ hg clone expat-import myexpat
updating working directory
56 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd myexpat
$ gedit lib/expat.h
```

Change the file slightly (add an #ifdef statement, such as the diff below shows), save and commit:

```
$ hg diff
diff -r ed48f733d885 lib/expat.h
--- a/lib/expat.h       Fri Jan 01 18:54:59 2010 +0100
+++ b/lib/expat.h       Fri Jan 01 19:00:54 2010 +0100
@@ -8,6 +8,10 @@

 #include <stdlib.h>

+#ifdef MY_PATCH
+int MyPatchVar;
+#endif
+
 #ifndef XMLPARSEAPI
 #  ifdef __declspec
 #    define XMLPARSEAPI __declspec(dllimport)
$ hg ci -m"My patch"
```

Get the next release, and update the import repository:

```
$ cd ..
$ wget http://sourceforge.net/projects/expat/files/expat/1.95.1/expat-
1.95.1.tar.gz/download
$ tar xvfz expat-1.95.1.tar.gz
...
$ cp -ar expat-1.95.1/* expat-import
$ cd expat-import
$ hg ci -m"Imported 1.95.1"
$ hg tag 1.95.1
```

Pull over the changes and rebase:

```
$ cd ../myexpat
$ hg pull --rebase                    # Use rebase to keep your patch on top
pulling from /home/thorman/expat-import
searching for changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 14 changes to 14 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
merging lib/expat.h
saving bundle to /home/thorman/myexpat/.hg/strip-backup/9451a841b310-temp
adding branch
adding changesets
adding manifests
adding file changes
```

```
added 3 changesets with 15 changes to 14 files
rebase completed
$ hg glog
@  changeset:   3:653e4db40e6b
|  tag:         tip
|  user:        Thorbjorn Jemander <thorbjorn@jemander.se>
|  date:        Fri Jan 01 19:01:52 2010 +0100
|  summary:     My patch
|
o  changeset:   2:36287c5314a4
|  user:        Thorbjorn Jemander <thorbjorn@jemander.se>
|  date:        Fri Jan 01 19:05:59 2010 +0100
|  summary:     Added tag 1.95.1 for changeset 15d033d27d73
|
o  changeset:   1:15d033d27d73
|  tag:         1.95.1
|  user:        Thorbjorn Jemander <thorbjorn@jemander.se>
|  date:        Fri Jan 01 19:05:50 2010 +0100
|  summary:     Imported 1.95.1
|
o  changeset:   0:ed48f733d885
   tag:         1.95.0
   user:        Thorbjorn Jemander <thorbjorn@jemander.se>
   date:        Fri Jan 01 18:54:59 2010 +0100
   summary:     First check-in
```

It is not necessary to rebase, but it makes it easy to change your patch as time passes: just edit the tip and commit.

9.5.2 Using an import branch

Initiate a repository and create a branch called import, and add the imported contents into that branch.

```
$ mkdir expat; cd expat; hg init; cd ..
$ cp -ar expat-1.95.0/* expat
$ cd expat
$ hg addremove
...
$ hg ci -m"First check-in"
$ hg branch import
marked working directory as branch import
$ hg tag 1.95.0
$ hg update default
0 files updated, 0 files merged, 1 files removed, 0 files unresolved
$ gedit lib/expat.h
```

Change the file, commit, change to import branch, import next release and merge:

```
$ hg ci -m"My patch"
created new head
$ hg up import                     # Move to import branch
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cp -ar ../expat-1.95.1/* .
$ hg ci -m"Imported 1.95.1"
$ hg tag 1.95.1
$ hg up default                    # Move to default branch
```

```
13 files updated, 0 files merged, 1 files removed, 0 files unresolved
$ hg merge -r 4
merging lib/expat.h
13 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg glog
@    changeset:   5:a60f4583b4f3
|\   tag:         tip
| |  parent:      2:b75f6b9f470f
| |  parent:      4:1e367bc40b34
| |  user:        Thorbjorn Jemander <thorbjorn@jemander.se>
| |  date:        Fri Jan 01 18:39:37 2010 +0100
| |  summary:     Merged in new import
| |
| o  changeset:   4:1e367bc40b34
| |  branch:      import
| |  user:        Thorbjorn Jemander <thorbjorn@jemander.se>
| |  date:        Fri Jan 01 18:37:31 2010 +0100
| |  summary:     Added tag 1.95.1 for changeset 26c2874fe905
| |
| o  changeset:   3:26c2874fe905
| |  branch:      import
| |  tag:         1.95.1
| |  parent:      1:d126eef1462c
| |  user:        Thorbjorn Jemander <thorbjorn@jemander.se>
| |  date:        Fri Jan 01 18:37:21 2010 +0100
| |  summary:     Imported 1.95.1
| |
o |  changeset:   2:b75f6b9f470f
|/   user:        Thorbjorn Jemander <thorbjorn@jemander.se>
|    date:        Fri Jan 01 18:35:46 2010 +0100
|    summary:     Implemented my patch
|
o  changeset:   1:d126eef1462c
|  branch:      import
|  user:        Thorbjorn Jemander <thorbjorn@jemander.se>
|  date:        Fri Jan 01 18:32:20 2010 +0100
|  summary:     Added tag 1.95.0 for changeset ade3e08739f3
|
o  changeset:   0:ade3e08739f3
   branch:      import
   tag:         1.95.0
   user:        Thorbjorn Jemander <thorbjorn@jemander.se>
   date:        Fri Jan 01 18:32:11 2010 +0100
   summary:     First import
```

Personally I prefer the method of using two repositories combined with rebasing.