OGD Workshops 2009

Pragmatic Source Code Management

A practical guide to 42

Jeroen van Meeuwen, RHCE

OGD Workshops 2009 Pragmatic Source Code Management
A practical guide to 42
Edition 0

Author Jeroen van Meeuwen, RHCE *j.van.meeuwen@ogd.nl*

A practical guide to Pragmatic Source Code Management, or 42, otherwise known as the answer to the question of "Life, Everything and the Universe", from the Hitchhikers Guide to the Galaxy.


DRAFT

Preface

1. Terminology

Some terminology used throughout this guide, to clarify what it is we are talking about.

1.1. Changeset

A changeset is a single change, preferably consistent, to a source tree, such as the addition of a feature, or a bug being fixed. A changeset therefor may consist of a single *commit*, or multiple *commits* to a *development path*.

Changeset with a Single Commit

An example of a single changeset to a program's source tree using a single commit:

A "fix typo" type of bugfix:

```
commit 87788a87185cf7cb6c235a20d3a51d304f7b6516
Author: Jeroen van Meeuwen (OGD) <j.van.meeuwen@ogd.nl>
Date:   Mon Dec 28 17:01:24 2009 +0100

    Fix typo (ticket #122)

diff --git a/hello_world.sh b/hello_world.sh
index 9436431..cd16289 100644
--- a/hello_world.sh
+++ b/hello_world.sh
@@ -1,3 +1,3 @@
 #!/bin/bash

-echo "Helo World!"
+echo "Hello World!"
```

Changeset with Multiple Commits Example

An example of a single changeset to a program's source tree with multiple commits:

1. The first commit makes a non-intrusive change to the application's code such as an if/else statement with the default being the exact same behaviour the application had previously. Whatever the actual functional change may be given changing the configuration setting doesn't matter in this example.

2. The addition of some configuration file logic that enables the application to read that setting from the configuration file, parse it, check it, and then change the behaviour described in step *1*.

3. The addition of a command-line parameter to the application enabling the user to specify the desired behaviour of the application given the change described in list item *1*.

The addition of the feature (being able to change the application through a configuration file option or with help of a command-line option), is a single changeset. This changeset could be reverted in order to remove the functionality, in case (for example), the feature is not sustainable for a released version of the application.

1.2. Commit

A commit (or *revision*), is a single patch or modification to the source repository.

1.3. Branch

A branch is one continuous development path. A single source code repository can have multiple branches, allowing various development paths for the application. Common use of branches include version specific branches, platform specific branches, or operating system (version) specific branches.

1.3.1. (Feature) Development Branches

Suppose a web application requires refactoring it's Javascripts and CSS Stylesheets. A development team agrees that a couple of people will be working on the refactoring of those scripts and stylesheets, but doing so might break the mainstream development version of the web application, as pieces of content might all of a sudden dissappear and all kinds of events might render the development version of the web application unuseable, this unfit for other developers to continue working on.

The team decides to have the refactoring be done in a different branch, as to not interfere with regular development. Such a *branch* is a development branch, to be *merged* with the *mainstream development branch* once the refactoring is done.

1.3.2. Version Specific Branches

Imagine a program reaches the point where version 1.0 can be released. Maybe the roadmap for the program says development continues towards 2.0, and 1.0 can only contain fixes from this point forward. Imagine version 1.0 might have to be supported for a period of 7 years (the support- and life cycle of an Enterprise Linux distribution).

In such cases, branching off to **`program-1.0`** makes sense, since that the program probably needs a continuous development path in order to be able to apply any fixes to the source code of program version 1.0.

1.3.3. Platform Specific Branches

Imagine a program is released for Linux as well as Windows. In such cases, it can be reasonable to give both platforms their own continuous development path (although not necessarily adviseable), so that development for Windows can continue separately from development for Linux.

Even though this might not be adviseable, one could chose to have those platform specific branches anyway to allow faster development for each platform (only to merge changes and make them interoperable later on).

1.3.4. Operating System or Distribution (Version) Specific Branches

Think of a program that is specifically designed to work on one type of distributions, say Fedora (F in short), and it's Enterprise Linux grade derivatives, Red Hat Enterprise Linux and CentOS (short; EL).

At a given point (say Fedora Core 6 or FC-6), an Enterprise Linux (major version 5) grade distribution is created from what is in the Fedora Core distribution. This would be a reasonable time for the distribution specific application to also branch off and create it's EL-5 version of the program, since the Enterprise Linux variant of the application will need to be supported for 7 years.

1.4. Tag

A tag is a point in a source code repository's history. You would use a tag to be able to retrieve the code at a specific point in time, relatively easy. Example tags are versioned tags, where you tag the point in a source tree development path whenever you release a specific version.

1.5. Merge Conflict

When multiple commits or multiple changesets need to be merged into one continuous development path, a merge conflict may arise.

1.6. Merge Strategy

When multiple commits or multiple changesets need to be merged into one continuous development path, a merge strategy helps you determine what commits and what changesets overrule other commits and changesets.

1.7. Branch Merge

A branch merge happens when two seperate development paths are merged onto one branch. This latter branch can be one of the two development paths that is merged, but can also be a third branch.

Suppose you have a version specific branch **`program-1.0`** for continued development on the 1.0 series of products of program. Suppose someone fixes bug #123, while someone else fixes bug #435, each of them using a separate branch to do so, but also changing the same code. Getting the changes back into mainstream requires a branch merge between the two branches used to fix the bugs.

1.8. Branch Rebase

A branch rebase happens when two seperate development paths need to be merged onto the current branch in a very specific way.

You are on one branch (say, **`master-df-444`**), and you want to rebase your changes on another branch (say, **`master`**).

You will want to rebase the **`master-df-444`** branch using the **`master`** branch, meaning that your changes are going to be applied to the codebase currently in **`master`**. In order to do so, you will need to reset your branch to what is currently in the **`master`** branch, only to then apply each change you made in the **`master-df-444`** branch one by one.

This is different from branch merging, where changes are merged. With rebasing, you are re-applying your changes based on a different HEAD. With merging, you are applying all changes in different branches given a *merge strategy*. You could think of branch rebasing as merging though, with a very specific merge strategy.

1.9. Operating System

Examples of operating systems are win32, Linux, Darwin. This does not include the version of the operating system, such as Linux 2.4 or Darwin 10.4.

2. Feedback

You should over ride this by creating your own local Feedback.xml file.


DRAFT

Introduction

This book is a guide to pragmatic Source Code Management (SCM). With a few simple, example software development projects and a few scenarios for those software development projects, we hope to give you some more insight on why source code management should be done properly, pragmatic, and how it could be done so.

1.1. About this Document

The creation and maintenance of this book is (and has been) a collaborative effort, hopefully introducing you to pragmatic SCM all the way from the very basics onto the more advanced topics.

It has originally been developed as a guide to SCM for developers unfamiliar with various SCM topics such as continuous development paths, branching and tagging, or how to use these features in a software development project.

For those of you readers already familiar with one or the other type of SCM system(s), please realize this book is not a discussion on which SCM system is best suited for any given type of software development project, but a workshop of the type creating awareness of applying (best?) practices using the proper kind of SCM system, whichever you find most appropriate.

1.2. Source Code Management Systems

Well-known, mature Source Code Management systems include:

- Bazaar
- Subversion
- GIT
- Mercurial
- CVS
- RCS

Of course, each of these SCM systems has its features, advantages and disadvantages, and in this document, we'll merely just emphasize some of them. However, we'll not do a full-blown comparison. The authors hope you can make up your own mind.


DRAFT

Revision Control

Revision control (also known as *version control*, *source control* or *(source) code management*) is about the ability to keep two or more revisions of a given file or directory tree apart. It allows you to track changes made between different revisions. In it's simplest form, the set of changes results in only one final *working copy*.

One example of revision control in it's simplest form is the revision control often available in a text document processor such as Microsoft Word or OpenOffice.org Writer.

This type of *embedded* revision control allows you to do only either one of two things:

1. Create a new revision of the document.

2. Checkout a previously created revision.

You can compare this type of revision control with the ability to Undo or Redo changes in most applications. Once you Undo something, and then do something other then Redo, your Redo list is lost.

Another example of simplistic revision control is a Wiki page. A Wiki page has one current version (the latest revision), and a set of changes. All changes put together form the latest revision of the page. This latest revision is what you see when you visit the Wiki page.

To be able to put revision control on a Wiki page, the database stores the following items:

- Changesets to a page, including date/time, author information, and a commit message.

- The latest complete revision of the page, built from all current changesets.

The current version of a Wiki page is built from a chronological set of changes to the page, and the final page (the *working copy*) is stored in the database for efficiency; It would require a lot more processing power to build (and rebuild, and rebuild) the entire page from its related changesets every time the page was requested.

If a user or administrator removes one or more sequential changes from the stack of changes, the page can often be rebuilt by stacking all remaining changes on top of one another. However, removing one changeset could mean the following changeset could not be applied any longer. Suppose there is change #11 changing a line, and change #12 changing that very same line. When you pop change #11 off the stack of changes, change #12 no longer applies cleanly. You are being forced to *merge* change #12 on top of change #10.

A Wiki page or text document rarely needs two or more separate continuous development paths. There is practically only one way to go from a former version of the document to a newer version of the document, on to a future version of the document.

2.1. Problems with Simplistic Revision Control

Sometimes, removing one change from a stack of changes results in (parts of) other changes to not be applicable anymore. You can imagine how a tower of bricks can only have so many bricks removed half-way up the tower. See the following example, where we edit a README file three times;

Revision 1

Our example "program" has a simple README file, which reads:

```
This is the original README file
```

This is revision 1, and the changeset (to create revision 1), looks as follows:

```
--- /dev/null   2009-09-01 15:26:52.811115027 +0200
+++ README.rev1 2009-09-03 15:04:48.781391924 +0200
@@ -0,0 +1 @@
+This is the original README file
```

Revision 2

Someone edits revision 1 and creates revision 2. The README file now reads:

```
This is the 2nd revision of the README file
```

The change reads as follows:

```
--- README.rev1 2009-09-03 15:04:48.781391924 +0200
+++ README.rev2 2009-09-03 15:05:00.734386713 +0200
@@ -1 +1 @@
-This is the original README file
+This is the 2nd revision of the README file
```

What this *diff*, or *changeset*, describes, is that a line (somewhere around line number 1) containing

`This is the original README file`

is to be removed, and instead a line containing

`This is the 2nd revision of the README file`

is to be added.

Revision 3

Someone edits revision 2 and creates revision 3. The file now reads:

```
This is the 3rd revision of the README file
```

The differences between revision 2 and revision 3 read:

```
--- README.rev2 2009-09-03 15:05:00.734386713 +0200
+++ README.rev3 2009-09-03 15:05:09.060422417 +0200
@@ -1 +1 @@
-This is the 2nd revision of the README file
+This is the 3rd revision of the README file
```

2.1.1. Removing a Single Revision

Now imagine revision 2 is being pulled from the stack of changes. Changeset #3 can no longer be applied, because there is no line "**`This is the 2nd revision of the README file`**" to be removed.

This creates a conflict, since the remaining changesets can no longer be stacked (applied in chronological order, one on top of the other). As such, in the most simplistic form of revision control, one can only pop the latest revision off the stack, in order to roll back (a set of) changes. One cannot just simply remove any random changeset from the middle of the stack, unless of course the resulting *merge* succeeds;

Imagine the README file had three lines, and we edited three seperate lines in each revision like in the previous example. Removing changeset #2 would still be compatible with applying changeset #3 to revision #1.

2.2. One Development Path

When you do use simplistic revision control (such as what you do in the case of Wiki pages or Office documents), *Figure 2.1, "One continuous development path"* would be illustrating the development path and history of the document.

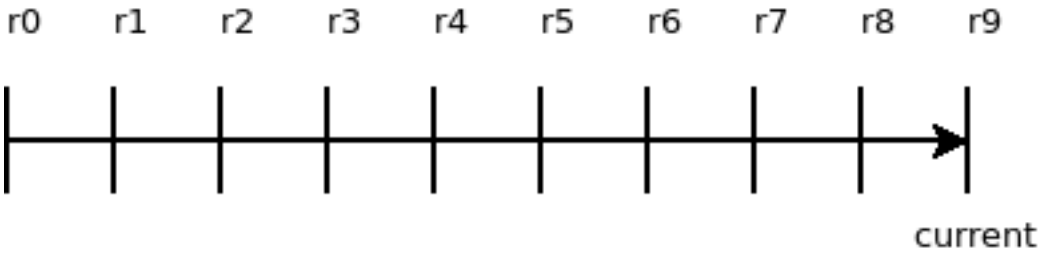


Figure 2.1. One continuous development path

One change is stacked onto a series of other changes. There is no way to derive from that one path the document is on, and the changes that other people are going to make to the document. You cannot simply branch off and say you are going to give the document a different layout, only to merge in changes other people have made, unless you have a merge strategy. The more changes applied to the document by other people, the harder the merge is going to be.

In the end, simplistic revision control is going to give you one development path and one development path only. See *Figure 2.2, "Folding changes into the development path"* for an illustration on how changes aside from the usual revision-to-revision changes are still going to need to be folded into that very one development path.

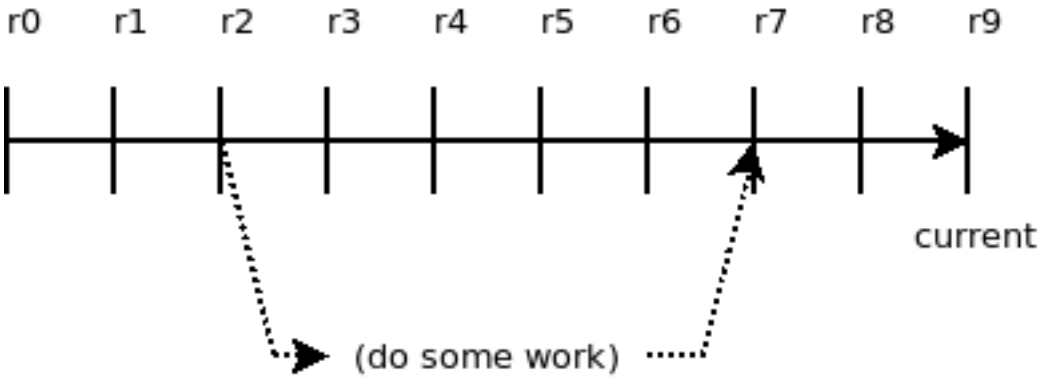


Figure 2.2. Folding changes into the development path


DRAFT

Software Development

While the most basic form of Revision Control might work for Wiki pages and Office documents, you can imagine that in more complex scenarios, such as *Software Development*, you may want more then single stream revision control with just one continuous development path (for example, multi-stream revision control). An SCM system with just simple revision control does not suffice for software development.

One or more of the following conditions may apply when using a SCM system in software development:

- The software may need to be compatible with more then one operating system distribution and/or version.
- The software may need to be supported for a while after the product is released.
- The software may be Free and Open Source software.
- You may have more then one person working on the software.
- You may need different versions of the software source tree to be available in various locations and control which version is available in what location very granularly.
- You may need different authentication and authorization for internal and external development versions of the program.

Now imagine you had to work on your programs source code like if it were a Wiki page. There would be only 1 working copy, and that one working copy would have to work on all supported platforms. Everyone involved with the project can only work on that one working copy. All changes, including compatibility changes within a piece of software would have to happen within that working copy. Like we suggested before, simplistic single stream revision control would not suffice.

This chapter briefly discusses some of the constraints you may face in software development, that justify choosing a more complex SCM system, and managing your source code pragmatically.

3.1. Software Dependencies

Software is most likely to have *dependencies* on other software. There is no all-inclusive piece of software, that runs entirely on it's own, without a single dependency on some other piece of software. For one, all software needs an Operating System, and one Operating System might behave differently then another Operating System.

These kind of dependencies are called *platform dependencies*. Linux behaves differently then Windows, and Windows NT 4.0 in turn behaves differently then Windows 2008. Each of these *platforms* requires the software to be altered.

Then, each version of the platform ships its own set of software. For example; Red Hat Enterprise Linux 5 ships with **gcc-4.3** and **ruby-1.8.5**, whereas Red Hat Enterprise Linux 6 ships (is going to ship) **gcc-4.4** and **ruby-1.8.6**. Advanced Programming Interface of the software available on the platform might change, as well as the requirements to your software.

gcc-4.3 vs. gcc-4.4

In order for a program that needs to be compiled with **gcc** to be compatible with both version **`4.3`** and **`4.4`**, one may need two separate, continuous development paths. Luckily in the case of **gcc**, **`4.4`** code can often be compiled with version **`4.3`** of **gcc** as well, and **`4.3`** code needs minor adjustments to be compiled with **gcc-4.4**, which is basically just more strict then its predecessor.

ruby-1.8.5 vs. ruby-1.8.6

In the case of **ruby-1.8.5** vs. **ruby-1.8.6**, changes to the software program dependent on **ruby** that would make the software compatible with both versions of **ruby** are often not as easy to implement. There's API changes (to for example **`DateTime.parse()`** syntax), and maybe some required applications are only available with **ruby-1.8.6** (such as the Ruby Gem sanitize).

More on how to work out these differences between platforms and software dependencies later on in this document.

3.2. Software Development in Teams

Most software is developed as part of some kind of team effort. When multiple developers work on the same project, the chances of (unintentionally) creating a *merge conflict* rises.

Merge conflicts arise when changes between different versions of the program conflict with one another. Maybe you and your colleague both edited the same line in the same file, but edited them differently.

In order to address merge conflicts, one needs a *merge strategy*. While different SCM systems feature different kinds of utilities facilitating different merge strategies, making sure you have a merge strategy for any given kind of conflict defines the usefulness of pragmatic source code management.

3.3. Community and Contributors

If a project is Free Software, and many of them are, then one source code repository in one location may not suffice to satisfy the need of all your contributors.

A single source code repository has standards like what is conform a roadmap for a certain branch, and when you branch and what the branch name is like, or what to tag and what the tag name is like. If these standards would not be applied to the source code repository, tracking what is what exactly becomes more difficult over time not to say it quickly becomes a giant mess. With only one source code repository and with multiple contributors in your community, the demand for tags and branches other then those relevant to the upstream project quickly grows, and it becomes harder to control the source code repository as there will be more exceptions to the standards as work-flows change and differ between various people.

Some examples of very basic standards include:

- Only complete changesets can be pushed upstream. This means that no single commit can break the program, or, put more plainly, no "Fix typo" commits are allowed.
- Only clean code is committed. Code that conforms to coding standards and best practices, including indentation, function names, corresponding unit and functional tests, proper changelog entries.
- The changeset fixes a bug, adds a feature on the current roadmap or enhances the program within its current functionality in order to go to the master branch.

- The changeset has been reviewed and OK'ed by at least one or two other valued contributors.
- The changesets commit message is clear and distinctive. In other words, no "Let's try this instead", or "Updating program".

3.3.1. The Demand for Tags and Branches

Imagine the **foo** program mentioned earlier. Upstream releases version 1.0, but in order for the program to work on Fedora, it needs a patch or two. One patch may be to make sure all files end up in the right location according to Fedora standards (useless to upstream), while another patch is to make sure **foo-1.0** compiles with gcc-4.4. These patches can be applied to the RPM package (on top of the released tarball of version 1.0), but the package maintainer also needs a way to develop and track his changes, and make them into nice patches. Those patches relevant to upstream will only be released by upstream in either 1.1 or 1.0.1, consituting a new release and so for the time being they are going to need to be patches shipped with the RPM package.

Long story short, in this scenario there are two development forces at work; 1) The upstream development team, and 2) the downstream packager(s).

If both were to operate in a single source code repository, the release of version **foo-1.0** would probably constitute the following branches: **`foo-1.0`** for upstream support, and then **`foo-1.0-rawhide`**, **`foo-1.0-f11`**, **`foo-1.0-f10`**, **`foo-1.0-el4`** and **`foo-1.0-el5`** for downstream packaging for the RPM based distributions, not to mention openSUSE, SUSE, Debian, *buntu, Gentoo or other Linux distributions.

3.4. Long Term Source Code Management

SCM systems, for software source code with an actual future, need to be very efficient in their storage footprint, as well as in the availability of continuous development paths (branches) and specific versions (tags). The cost of these operations, branching, tagging, and branching off from a certain tag whenever the need arises, cannot increase too much over time. The cost effectiveness of a source code management system determines the long term availability of all that a software development team needs to optimally use the features of said source code management system.

If tagging a specific version entails you copy the entire source code tree you tag onto a different location, then imagine what the software tree looks like in 5 years after having tagged a couple of dozen times.

If branching off entails you copy the entire source code tree as well, then also think about this new continuous development path being tagged to indicate you have released a specific version. Again you copy the entire source code tree, which isn't exactly efficient or cost effective.

3.5. Changeset Consistency

Within every development path, you would want to create the possibility to make changesets applied to developing, say, a specific feature to not include "Fix typo" commits, and thus consistent changesets.

3.6. Ever Growing Projects

para


DRAFT

Pragmatic SCM

para

4.1. Tags and Branches

para


DRAFT

Example Project: VMime

VMime is a powerful C++ class library for working with MIME messages and Internet Messaging services like IMAP, POP and SMTP. The website of the VMime project is at *http://www.vmime.org*.

The VMime project uses Subversion as the Source Code Management system, at *https://vmime.svn.sourceforge.net/svnroot/vmime/*. As you can see in the **`tags/`** sub-directory, the project does not always create points of reference for each released version of the program, but that is of later concern.

5.1. Consumer of VMime

A consumer of VMime is Zarafa, a Groupware environment. You can find Zarafa at *http://www.zarafa.com*.

Zarafa has created a number of patches against upstream versions **`0.7.0`** and **`0.7.1`**, creating it's own fork. While reasonable when looked at from the perspective of Zarafa, imagine every other application depending on a library plus a few patches did this. Without the patches going to upstream veryone would be running in circles.

Zarafa now strongly depends on their own 0.7.1 version, because of the patches integrated, and the API compatibility of VMime with Zarafa. One could argue that Zarafa is frozen to a forked libvmime-0.7.1.

5.2. Different Versions of VMime

Right now, there are different versions of VMime:

5.2.1. Upstream Versions

Upstream has released the following versions:

1. 0.7.0 (unmarked in SVN)
2. 0.8.0 (unmarked in SVN)
3. 0.8.1 (unmarked in SVN)
4. 0.9.0 (SVN tag)

5.2.2. Zarafa Versions

1. 0.7.1 (no SCM available)

5.3. Working Pragmatic SCM Magic

- **`libvmime-0.7.0`**, upstream release of version 0.7.0
- **`libvmime-0.7.1`**, upstream release of version 0.7.1
  - **`upstream-libvmime-0.7.1`**, version 0.7.1, with backports of patches accepted upstream but not released in any 0.7.x API version.

    - **`zarafa-libvmime-0.7.1`**, version 0.7.1, with all patches from Zarafa (including patches not yet proposed upstream, or not yet accepted by upstream).

- **`libvmime-0.8.0`**, upstream release of 0.8.0
- **`libvmime-0.8.1`**, upstream release of 0.8.1
- **`libvmime-0.9.0`**, upstream release of 0.9.0

Appendix A. SCM Conflict Demonstration

The following is a step-by-step demonstation of a set of changes applied to one end, conflicting with a set of changes applied on another end. Follow the steps to (re-)create the conflict and to get an impression of the difficulties that come with merging changes.

In this demonstration, we use Subversion because it is the easiest Source Code Management system to create a conflict with, that needs immediate resolving. Also, while the problem introduced in this example is practically inherent to applying full translations to the actual codebase rather then allowing translations to be loaded through appropriate mechanisms, please realize it is not about the translations, and it's not about whether this conflict could have been avoided.

1. On your system, create a SVN repository:

```
$ sudo svnadmin create /srv/hello.svn
$ sudo chown -R `id -u` /srv/hello.svn
```

2. While in your home directory, clone this repository for developer 1:

```
$ svn co file:///srv/hello.svn ~/hello.svn.dev1
Checked out revision 0.
```

3. Create the initial **`hello.sh`**:

```
$ cd ~/hello.svn.dev1
$ cat >hello.sh<<EOF
> #!/bin/bash
>
> # This is probably the most complex program you've ever seen.
>
> echo "Hello World!"
> EOF
$ svn add hello.sh
A         hello.sh
$ svn ci -m "First version of the hello program"
Adding         hello.sh
Transmitting file data .
Committed revision 1.
```

4. Checkout this repository for developer 2:

```
$ cd ~
$ svn co file:///srv/hello.svn ~/hello.svn.dev2
Checked out revision 1.
```

5. Translate **`hello.sh`** to Dutch for developer 1:

```
$ cd ~/hello.svn.dev1
$ cat >hello.sh<<EOF
> #!/bin/bash
>
> # Dit is waarschijnlijk het meest complexe programma dat
```

```
> # je ooit hebt gezien.
>
> echo "Hallo Wereld!"
> EOF
```

6. Optionally, examine the changes you made:

```
$ svn diff
```

7. Checkin your changes:

```
$ svn ci -m "Dit is de Nederlandse versie van hello.sh"
Sending        hello.sh
Transmitting file data .
Committed revision 2.
```

8. Translate revision #1 for developer 2 to French:

```
$ cd ~/hello.svn.dev2
$ cat >hello.sh<<EOF
> #!/bin/bash
>
> # Je ne parles pas Francais du tout.
>
> echo "Salut le monde!"
> EOF
$ svn ci -m "Very bad translation to French"
Sending        hello.sh
svn: Commit failed (details follow):
svn: File '/hello.sh' is out of date
```

To resolve the conflict, developer 2 needs to do the following:

1. Pull the updates:

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

   Let's see what options we have:

   - **(p) postpone**
     Postpone merging the changes, leaving the current working copy in conflict (unmerged).
   - **(df) diff-full**
     Show exactly what this conflict is all about, in a full diff, allowing the developer to examine the impact of the merge conflict. In this demonstration, the output would be:

```
Select: (p) postpone, (df) diff-full, (e) edit, (r) resolved,
        (mc) mine-conflict, (tc) theirs-conflict,
        (s) show all options: df
--- .svn/text-base/hello.sh.svn-base    Thu Sep  3 16:08:00 2009
+++ .svn/tmp/hello.sh.tmp   Thu Sep  3 16:31:05 2009
```

```
@@ -1,5 +1,14 @@
 #!/bin/bash

-# This is probably the most complex program you've ever seen.
+<<<<<<< .mine
+# Je ne parles pas Francais du tout.
+=======
+# Dit is waarschijnlijk het meest complexe programma dat
+# je ooit hebt gezien.
+>>>>>>> .r2

-echo "Hello World!"
+<<<<<<< .mine
+echo "Salut le monde!"
+=======
+echo "Hallo Wereld!"
+>>>>>>> .r2
Select: (p) postpone, (df) diff-full, (e) edit, (r) resolved,
        (mc) mine-conflict, (tc) theirs-conflict,
        (s) show all options:
```

Taking the execution of **echo** as an example, you can see that both changes agree on the removal of the line that says:

```
echo "Hello World!"
```

But then disagree about the line that should replace it, where the changes that developer 2 is trying to commit (indicated with **.mine**, between **<<<<<<< .mine** and **=======**) says it should be replaced with:

```
echo "Salut le monde!"
```

and revision #2 (already committed and pushed upstream, between **=======** and **>>>>>>> .r2**) says it should be replaced with:

```
echo "Hallo Wereld!"
```

More obvious then **df** is **dc**

- **(dc) display-conflicts**
  Show the actual conflict, rather then the full diff between my changes and the ones I'm pulling in.

  In our example scenario, this shows us the following:

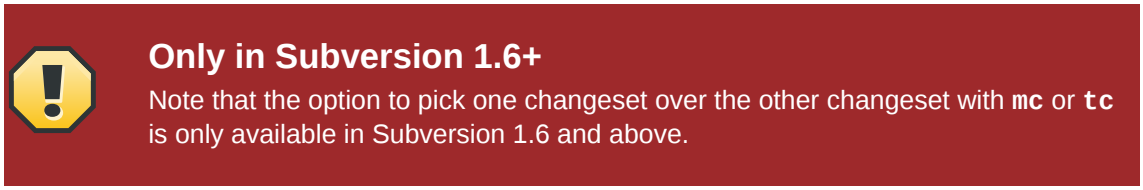
```
Select: (p) postpone, (df) diff-full, (e) edit, (r) resolved,
        (mc) mine-conflict, (tc) theirs-conflict,
        (s) show all options: dc
#!/bin/bash

<<<<<<< MINE (select with 'mc') (3)
# Je ne parles pas Francais du tout.
||||||| ORIGINAL (3)
# This is probably the most complex program you've ever seen.
=======
# Dit is waarschijnlijk het meest complexe programma dat
# je ooit hebt gezien.
>>>>>>> THEIRS (select with 'tc') (3,2)

<<<<<<< MINE (select with 'mc') (5)
echo "Salut le monde!"
```

```
||||||| ORIGINAL (5)
echo "Hello World!"
=======
echo "Hallo Wereld!"
>>>>>>> THEIRS (select with 'tc') (6)
Select: (p) postpone, (df) diff-full, (e) edit, (r) resolved,
        (mc) mine-conflict, (tc) theirs-conflict,
        (s) show all options:
```

immediately indicating that you can pick your changes over anyone else's with **mc**, or go with "their" changes (the changes pulled in by **svn up**) with **tc**.



- **(mc) mine-conflict**
  The acronym's meaning doesn't really speak, but you (having made the **.mine** changes) think that "they" (with the pulled in changes) conflict with you and so you want to pick your changes over their changes. Entirely reasonable, but not always appropriate.
- **(tc) theirs-conflict**
  Again the acronym's meaning doesn't really speak for itself but in this case you conflict with them and you think they know better; You accept their changes and throw away your own.
- **(e) edit**
  Edit the changes (not just the conflicts). In our example, this would open up $SVN_EDITOR or $EDITOR and show you the following, in an editor, for you to edit (and write):

```
#!/bin/bash

<<<<<<< .mine
# Je ne parles pas Francais du tout.
=======
# Dit is waarschijnlijk het meest complexe programma dat
# je ooit hebt gezien.
>>>>>>> .r2

<<<<<<< .mine
echo "Salut le monde!"
=======
echo "Hallo Wereld!"
>>>>>>> .r2
```

Choose and pick the lines you want to keep, write out the file's new contents, quit the editor and mark the conflict as resolved. Then, commit the changes (you have actually rebased onto revision #2, as **svn status** and **svn diff** will show you).

Appendix B. Revision History

Revision 1.0


DRAFT

Index


DRAFT