CHAPTER TWENTY-ONE

Teams and Tools

Karl Fogel

[PROD: "*Creative Commons Attribution 3.0 license*" in Word had an underlying link to http://creativecommons.org/licenses/by/3.0/ but was NOT char-tagged Hyperlink. Underlying hyperlinks are NOT something we currently handle in Word → Frame conversions (although Adam is working on having conversion houses deal with this for Word → DocBook conversions); I just happened to notice it during a post-conversion checkup. Please talk to Adam about how to handle this one in Frame, with an eye to both print and downstream use. I've left the text above as it got semiconverted (note that it's tagged URL), along with the bogus Hypertext Marker it generated, so you'll need to reword/retag the text and move/recreate/edit the marker, according to whatever Adam decides. Please also ask Adam to follow up with the CE about this for future books: if CE removes a Hyperlink char tag, CE should remove any underlying hyperlink too; conversely, if text has an underlying hyperlink in Word that's meant to be retained, the text needs to be char-tagged Hyperlink too.—Tools]

This chapter is about the transformative effect that good tools can have on a team's ability to collaborate.

Consider that for years the World Wide Web consisted mainly of static pages that required technical expertise to write, and that readers could not influence except by sending email to the appropriate *webmaster@* address. Then a few visionary souls started making software that would allow anyone with basic computer skills to cause text to appear on the Web, and other software to allow readers to comment on or even edit those pages themselves. Nothing about blogs or Wikis was *technically* revolutionary; like the fax machine, they could have been invented years earlier, if only someone had thought of them. Yet once they appeared, they greatly increased people's ability to organize themselves into productive networks.

This chapter tells three stories about how good tools (or the lack of them) made a difference to a team. The tools discussed here are much narrower in scope than blogs and Wikis, but their specificity makes them well suited to teasing out some principles of collaboration tools. As my experience has mostly been with open source software projects, that's what I'll draw on, but the same principles are probably applicable to any collaborative endeavor.

How Open Source Projects Work

If you've already taken part in an open source project, you can skip this section. If not, let me introduce the basics of how such projects work so that the tools discussed in this chapter will make sense.

Open source software[*] is software released under a free copyright, allowing anyone to copy, modify, use, and redistribute the code (in either modified or unmodified form). Much of the software that runs the Internet and the World Wide Web is open source, and an increasing number of desktop applications are as well.

Open source programs are usually maintained by loosely organized coalitions of software developers. Some volunteer their time, others are paid by corporations in whose interests it is that the software be maintained. Because the participants are often spread across many time zones, and may never have met each other, the projects rely to an unusual degree on highly sophisticated collaboration tools: bug trackers, email lists and archives, network-based chat rooms, version control repositories, Wikis, and more. You don't have to be familiar with all these tools; I'll explain the ones that figure in the discussions that follow as they come up.

Each open source project must decide how to organize and govern itself. Usually this is resolved by informal means: projects tend to be started by a small group of people anyway (sometimes one person), and over time, as interested volunteers show up and get involved, the original core group realizes it would be to the project's advantage to invite

---

[*] The older term *free software* is synonymous. There is not space here to explain why there are two terms for the same thing; see *http://producingoss.com/en/introduction.html#free-vs-open-source* if you'd like to know more about this.

some of those newcomers to become core maintainers, too.* But once a project has many contributors, it can be difficult to keep track of them all and to identify which ones to cultivate as potential core maintainers. The first tool we'll look at grew out of one project's need to solve this problem.

The Contribulyzer

Like most open source projects, Subversion† is continually trying to identify potential new core maintainers. Indeed, one of the primary jobs of the current core group is to watch incoming code contributions from new people and figure who should be invited to take on the responsibilities of core maintainership. In order to honestly discuss the strengths and weaknesses of candidates, we (the core maintainers) set up a private mailing list, one of the few non-public lists in the project. When someone thinks a contributor is ready, she proposes the candidate on this mailing list, and sees what others' reactions are. We give each other enough time to do some background checking, since we want a comfortable consensus before we extend the offer; revoking maintainership would be awkward, and we try to avoid ever being in the position of having to do it.

This behind-the-scenes background checking is harder than it sounds. Often, patches‡ from the same contributor have been handled by different maintainers on different occasions, meaning that no one maintainer has a good overview of that contributor's activities. Even when the same maintainer tends to handle patches from the same contributor (which can happen either deliberately or by accident), the contributor's patches may have come in irregularly over a period of months or years, making it hard for the maintainer to monitor the overall quality of the contributor's code, bug reports, design suggestions, and so forth.

I first began to think we had a problem when I noticed that names floated on the private mailing list were getting either an extremely delayed reaction or, sometimes, *no* reaction at all. That didn't seem right: after all, the candidates being proposed had been actively

---

* You may be wondering what it means to have a "core group" of maintainers when open source means that anyone has the right to change the code. Doesn't that mean that anyyone who wants to be is a maintainer? Not quite. In open source, anyone is free to make a copy of the code and do what she wants with that copy. But when a group of people get together and agree to maintain one particular copy *collectively*, they obviously have control over who is and isn't in that group. Similarly, anyone who wants to translate the Bible is free to do so, since it's in the public domain, but no one can force one group of translators to work with another; and if two groups don't work together, the result will be two independent translations. In open source software, this situation is known as a *fork*, as in "fork in the road": two independent copies with increasingly diverging sets of modifications, growing increasingly apart over time. In practice, however, forks are rare: more often, programmers coalesce around one particular copy in order to pool their efforts.

† Subversion is an open source version control system; see *http://subversion.tigris.org/*.

‡ A *patch* is code contribution, such as a bugfix, sent in a special format known as "patch format." The details of that format don't matter here; just think of a patch as being a proposed modification to the software, submitted in extremely detailed form—right down to which precise lines of code to change and how.

involved in the project, usually quite recently, and generally had had several of their patches accepted, often after several iterations of review and discussion on the public lists. However, it soon became clear what was going on: the maintainers were hesitant to solely rely on their memories of what that candidate had done (no one wants to champion someone who later turns out to be a dud), but at the same time were daunted by the sheer effort of digging back through the list archives and the code change history to jog their memories. I sensed that many of us were falling into a classic wishful postponement pattern when it came to evaluations: “Oh, so-and-so is being proposed as a new maintainer. Well, I’ll save my response for this weekend, when I’ll have a couple of hours to go through the archives and see what they’ve done.” Of course, for whatever reason the “couple of hours” don’t materialize that weekend, so the task is put off again, and again... Meanwhile, the candidate has no idea any of this is going on, and just continues posting patches instead of committing[*] directly. This means continued extra work for the maintainers, who have to process those patches, whereas if the candidate could be made a maintainer himself, it would be a double win: he wouldn’t require assistance to get his patches into the code, and he’d be available to help process other people’s patches.

My own self-observation was consistent with this hypothesis: a familiar sense of mild dread would come over me whenever a new name came up for consideration—not because I didn’t want a new maintainer, but because I didn’t know where I’d find the time to do the research needed to reply responsibly to the proposal.

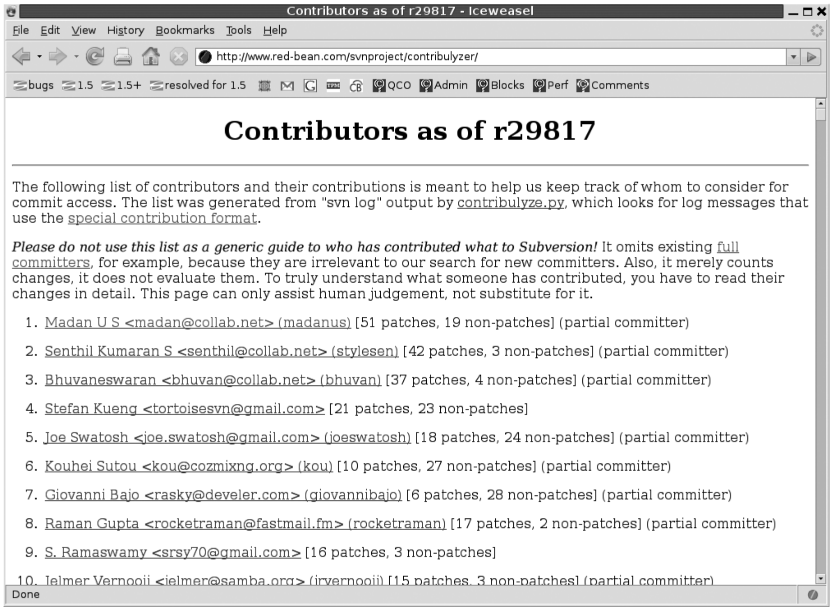
Finally, one night I set aside my regular work to look for a solution. What I came up with was far from ideal, and does not completely automate the task of gathering the information we need to evaluate a contributor. But even a partial automation greatly reduced the time it takes to evaluate someone, and that was enough to get the wheels out of the mud, so to speak. Since the system has been up, proposals of candidates are almost always met with timely responses that draw on the information in the new system, because people don’t feel bogged down by time-consuming digging around in archives. The new system took identical chores that until then had been redundantly performed by each evaluator individually, and instead performed them *once*, storing the results for everyone to use forever after.

The system is called the Contribulyzer (*http://www.red-bean.com/svnproject/contribulyzer/*): it keeps track of what contributors are doing, and records each contributor’s activity on one web page. When the maintainers want to know whether a given contributor is ready for the keys to the car, they just look at the relevant Contribulyzer page for that contributor, first scanning an overview of his activities and then focusing in on details as necessary.

---

[*] To *commit* means to send a code change directly into the project’s repository, which is where the central copy of the project’s code lives (see *http://en.wikipedia.org/wiki/Revision_Control* for more). In general, only core maintainers are able to commit directly; all others find a core maintainer to shepherd their changes into the repository. See *http://producingoss.com/en/committers.html#ftn.id304827* for more on the concept of “commit access.”

But how does a computer program "keep track of" what a contributor is doing? That sounds suspiciously like magic, you might be thinking to yourself. It isn't magic: it requires some human assistance, and we'll look more closely at exactly how in a moment. First, though, let's see the results. shows the front page of our Contribulyzer site.


Contributors as of r29817 - Iceweasel
File Edit View History Bookmarks Tools Help
http://www.red-bean.com/svnproject/contribulyzer/
bugs
1.5
1.5+
resolved for 1.5
QCO
Admin
Blocks
Perf
Comments
Done

Contributors as of r29817

The following list of contributors and their contributions is meant to help us keep track of whom to consider for commit access. The list was generated from "svn log" output by contribulyze.py, which looks for log messages that use the special contribution format.

Please do not use this list as a generic guide to who has contributed what to Subversion! It omits existing full committers, for example, because they are irrelevant to our search for new committers. Also, it merely counts changes, it does not evaluate them. To truly understand what someone has contributed, you have to read their changes in detail. This page can only assist human judgement, not substitute for it.

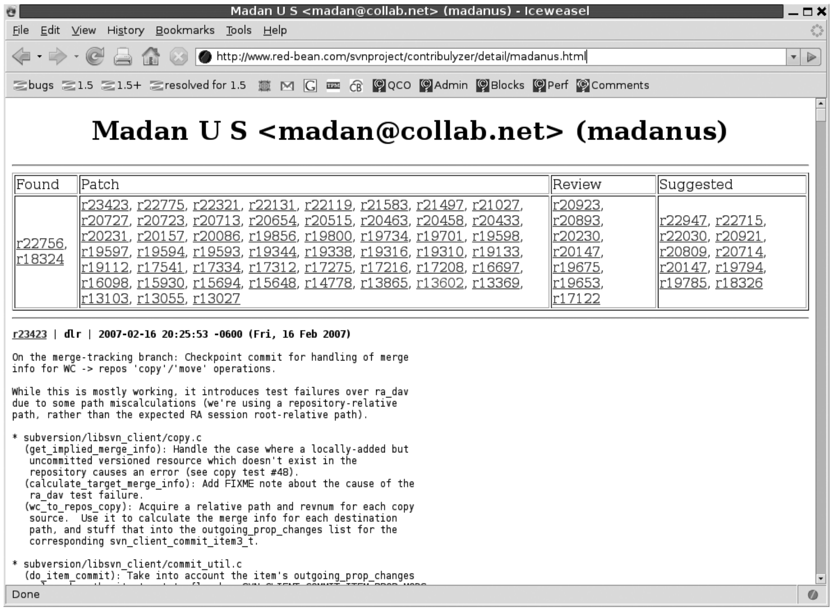
1. Madan U S <madan@collab.net> (madanus) [51 patches, 19 non-patches] (partial committer)
2. Senthil Kumaran S <senthil@collab.net> (stylesen) [42 patches, 3 non-patches] (partial committer)
3. Bhuvaneswaran <bhuvan@collab.net> (bhuvan) [37 patches, 4 non-patches] (partial committer)
4. Stefan Kueng <tortoisesvn@gmail.com> [21 patches, 23 non-patches]
5. Joe Swatosh <joe.swatosh@gmail.com> (joeswatosh) [18 patches, 24 non-patches] (partial committer)
6. Kouhei Sutou <kou@cozmixng.org> (kou) [10 patches, 27 non-patches] (partial committer)
7. Giovanni Bajo <rasky@develer.com> (giovannibajo) [6 patches, 28 non-patches] (partial committer)
8. Raman Gupta <rocketraman@fastmail.fm> (rocketraman) [17 patches, 2 non-patches] (partial committer)
9. S. Ramaswamy <srsy70@gmail.com> [16 patches, 3 non-patches]
10. Jelmer Vernooij <jelmer@samba.org> (jrvernooij) [15 patches, 3 non-patches] (partial committer)

The Contribulyzer (main page)

If you click on a contributor's name, it takes you to a page showing the details of what that contributor has done. shows the top of the detail page for Madan U S.

The four categories across the top indicate the kinds of contributions Madan has had a role in. Each individual contribution is represented by a *revision number*—a number (prefixed with *r*) that uniquely identifies that particular change. Given a revision number, one can ask the central repository to show the details (the exact lines changed and how they changed) for that contribution. For r22756 and r18324, Madan found the bugs that were fixed in those revisions. For the largest block of revision numbers, the "Patches," he wrote the change that some maintainer eventually committed. For the remaining revisions, he either reviewed a patch that someone else committed, or suggested the fix but (for whatever reason) was not the one who implemented it.

Those four sections at the top of the page already give a high-level overview of Madan's activity. Furthermore, each revision number links to a brief description of the corresponding change. This description is known as a *log message*: a short bit of prose submitted along with a code change, explaining what the change does. The repository records this message along with the change; it is a crucial resource for anyone who comes along later wanting to understand the change.


Madan U S <madan@collab.net> (madanus) - Iceweasel
File Edit View History Bookmarks Tools Help
http://www.red-bean.com/svnproject/contribulyzer/detail/madanus.html
bugs 1.5 1.5+ resolved for 1.5 QCO Admin Blocks Perf Comments
Done

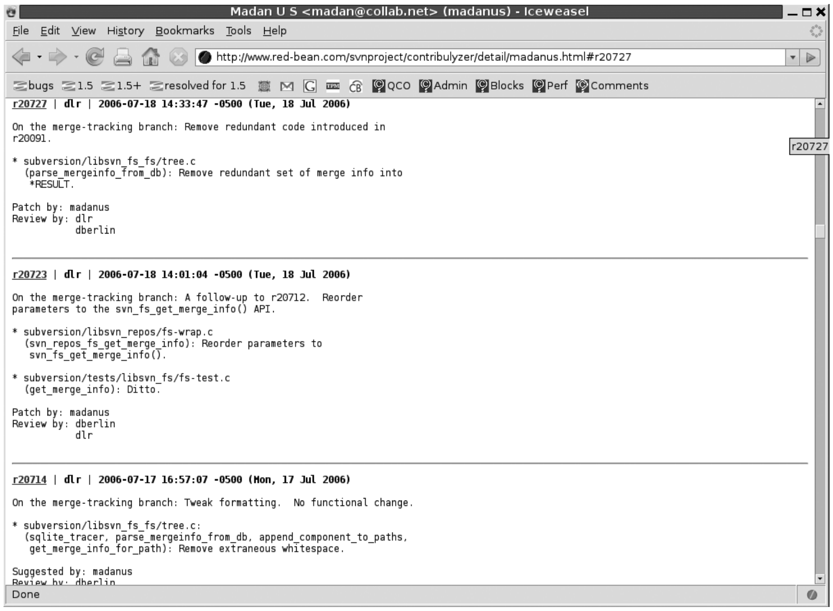
Madan U S <madan@collab.net> (madanus)

Found	Patch	Review	Suggested
r22756, r18324	r23423, r22775, r22321, r22131, r22119, r21583, r21497, r21027, r20727, r20723, r20713, r20654, r20515, r20463, r20458, r20433, r20231, r20157, r20086, r19856, r19800, r19734, r19701, r19598, r19597, r19594, r19593, r19344, r19338, r19316, r19310, r19133, r19112, r17541, r17334, r17312, r17275, r17216, r17208, r16697, r16098, r15930, r15694, r15648, r14778, r13865, r13602, r13369, r13103, r13055, r13027	r20923, r20893, r20230, r20147, r19675, r19653, r17122	r22947, r22715, r22030, r20921, r20809, r20714, r20147, r19794, r19785, r18326

```
r23423 | dlr | 2007-02-16 20:25:53 -0600 (Fri, 16 Feb 2007)

On the merge-tracking branch: Checkpoint commit for handling of merge
info for WC -> repos 'copy'/'move' operations.

While this is mostly working, it introduces test failures over ra_dav
due to some path miscalculations (we're using a repository-relative
path, rather than the expected RA session root-relative path).

* subversion/libsvn_client/copy.c
  (get_implied_merge_info): Handle the case where a locally-added but
   uncommitted versioned resource which doesn't exist in the
   repository causes an error (see copy test #48).
  (calculate_target_merge_info): Add FIXME note about the cause of the
   ra_dav test failure.
  (wc_to_repos_copy): Acquire a relative path and revnum for each copy
   source.  Use it to calculate the merge info for each destination
   path, and stuff that into the outgoing_prop_changes list for the
   corresponding svn_client_commit_item3_t.

* subversion/libsvn_client/commit_util.c
  (do_item_commit): Take into account the item's outgoing_prop_changes
```

The Contribulyzer (contributor page)

If you click on “r20727”, the top of the next screen will show the log message for that revision, as shown in .


Madan U S <madan@collab.net> (madanus) - Iceweasel
File Edit View History Bookmarks Tools Help
http://www.red-bean.com/svnproject/contribulyzer/detail/madanus.html#r20727
bugs 1.5 1.5+ resolved for 1.5 QCO Admin Blocks Perf Comments
r20727
Done

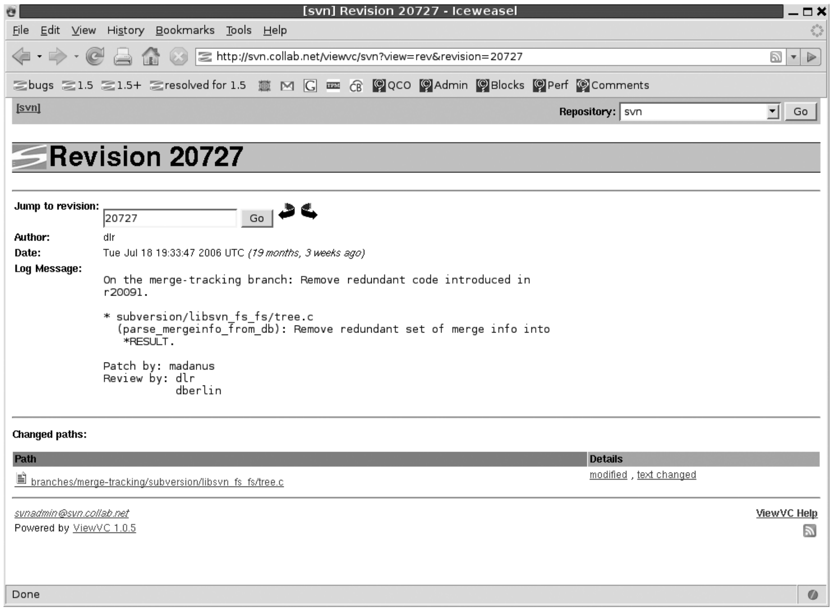
```
r20727 | dlr | 2006-07-18 14:33:47 -0500 (Tue, 18 Jul 2006)

On the merge-tracking branch: Remove redundant code introduced in
r20091.

* subversion/libsvn_fs_fs/tree.c
  (parse_mergeinfo_from_db): Remove redundant set of merge info into
   *RESULT.

Patch by: madanus
Review by: dlr
           dberlin
```

```
r20723 | dlr | 2006-07-18 14:01:04 -0500 (Tue, 18 Jul 2006)

On the merge-tracking branch: A follow-up to r20712.  Reorder
parameters to the svn_fs_get_merge_info() API.

* subversion/libsvn_repos/fs-wrap.c
  (svn_repos_fs_get_merge_info): Reorder parameters to
   svn_fs_get_merge_info().

* subversion/tests/libsvn_fs/fs-test.c
  (get_merge_info): Ditto.

Patch by: madanus
Review by: dberlin
           dlr
```

```
r20714 | dlr | 2006-07-17 16:57:07 -0500 (Mon, 17 Jul 2006)

On the merge-tracking branch: Tweak formatting.  No functional change.

* subversion/libsvn_fs_fs/tree.c:
  (sqlite_tracer, parse_mergeinfo_from_db, append_component_to_paths,
   get_merge_info_for_path): Remove extraneous whitespace.

Suggested by: madanus
Review by: dberlin
```

The Contribulyzer (revision entry)

The revision number here is a link, too, but this time to a page showing in detail what changed in that revision, using the repository browser ViewVC (*http://www.viewvc.org/*); see .


[svn] Revision 20727 - Iceweasel
File Edit View History Bookmarks Tools Help
http://svn.collab.net/viewvc/svn?view=rev&revision=20727
bugs
1.5
1.5+
resolved for 1.5
QCO
Admin
Blocks
Perf
Comments
[svn]
Repository: svn
Go
Go
Done

Revision 20727

Jump to revision: 20727

Author: dlr

Date: Tue Jul 18 19:33:47 2006 UTC *(19 months, 3 weeks ago)*

Log Message:

```
On the merge-tracking branch: Remove redundant code introduced in
r20091.

* subversion/libsvn_fs_fs/tree.c
  (parse_mergeinfo_from_db): Remove redundant set of merge info into
   *RESULT.

Patch by: madanus
Review by: dlr
           dberlin
```

Changed paths:

Path	Details
branches/merge-tracking/subversion/libsvn_fs_fs/tree.c	modified , text changed

svnadmin@svn.collab.net

Powered by ViewVC 1.0.5

ViewVC Help

ViewVC revision page

From here, you can see the exact files that changed, and if you click on “modified”, you can see the code diff itself, as shown in .

As you can see, the layout of information in the Contribulyzer matches what we’d need to jog our memories of a contributor’s work. There’s a broad overview of what kinds of contributions that person has made, then a high-level summary of each contribution, and finally, detailed descriptions for those who want to go all the way to the code level.

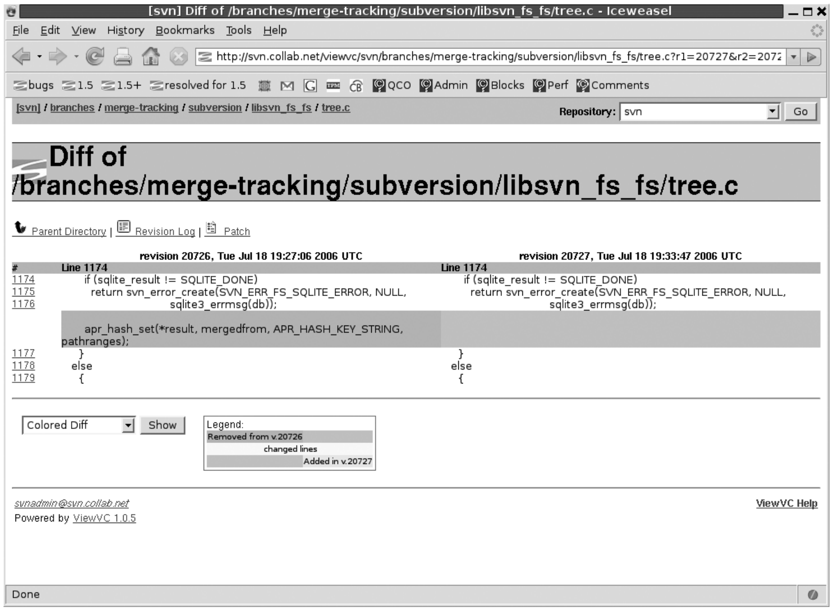
But how did all this information get into the Contribulyzer?

The Catch

Unfortunately, the Contribulyzer is not some miraculous artificial intelligence program. The only reason it knows who has made what types of contributions is because *we tell it*. And the trick to getting everyone to tell it faithfully is twofold:

- Make the overhead as low as possible.
- Give people concrete evidence that the overhead will be worth it.

Meeting the first conditon was easy. The Contribulyzer takes its data from Subversion’s per-revision log messages. We’ve always had certain conventions for writing these, such as naming every code symbol affected by a change. Supporting the Contribulyzer merely


[svn] Diff of /branches/merge-tracking/subversion/libsvn_fs_fs/tree.c - Iceweasel
File Edit View History Bookmarks Tools Help
http://svn.collab.net/viewvc/svn/branches/merge-tracking/subversion/libsvn_fs_fs/tree.c?r1=20727&r2=2072
bugs 1.5 1.5+ resolved for 1.5 QCO Admin Blocks Perf Comments
[svn] / branches / merge-tracking / subversion / libsvn_fs_fs / tree.c
Repository: svn
Go
Parent Directory | Revision Log | Patch
Colored Diff
Show
ViewVC Help
Done

Diff of /branches/merge-tracking/subversion/libsvn_fs_fs/tree.c

```
                revision 20726, Tue Jul 18 19:27:06 2006 UTC                 revision 20727, Tue Jul 18 19:33:47 2006 UTC
#       Line 1174                                                     Line 1174
1174        if (sqlite_result != SQLITE_DONE)                             if (sqlite_result != SQLITE_DONE)
1175          return svn_error_create(SVN_ERR_FS_SQLITE_ERROR, NULL,        return svn_error_create(SVN_ERR_FS_SQLITE_ERROR, NULL,
1176                                  sqlite3_errmsg(db));                                          sqlite3_errmsg(db));

            apr_hash_set(*result, mergedfrom, APR_HASH_KEY_STRING,
        pathranges);
1177      }                                                             }
1178    else                                                          else
1179      {                                                             {
```

Legend:
Removed from v.20726
changed lines
Added in v.20727

svnadmin@svn.collab.net
Powered by ViewVC 1.0.5

ViewVC file diff page

meant adding one new convention: a standard way of attributing changes that came from a source other than the maintainer who shepherded the change into the repository.

The standard is simple. We use one of four verbs (for the expected types of contributions), followed by the word *by:*, and then the names of the contributors who made that type of contribution to that change. Most changes have only one contributor, but if there are multiple contributors, they can be listed on continuation lines:

```
Patch by: name_1_maybe_with_email_address
          name_2_maybe_with_email_address
Found by: name_3_maybe_with_email_address
Review by: etc...
Suggested by: etc...
```

(These conventions are described in detail at *http://subversion.tigris.org/hacking.html#crediting*.)

One reason it was easy to persuade people to abide by the new standard is that, in a way, it actually made writing log messages easier. We'd been crediting people before, but in various ad hoc manners, which meant that each time we committed a contributor's code, we had to think about how to express the contribution. One time it might be like this:

```
Remove redundant code introduced in r20091.  This came from a patch by
name_1_maybe_with_email_address.
```

And another time like this:

```
Fix bug in baton handoff.  (Thanks to so-and-so for sending in the patch.)
```

While the new convention was one more thing for people to learn, *once learned* it actually saved effort: now no one had to spend time thinking of how to phrase things, because we'd all agreed on One Standard Way to do it.

Still, introducing a new standard into a project isn't always easy. The path will be greatly smoothed if you can meet the second condition as well, that is, show the benefits *before* asking people to make the sacrifices. Fortunately, we were able to do so. Subversion's log messages are editable (unlike some version control systems in which they are effectively immutable). This meant that, after writing the Contribulyzer code to process log messages formatted according to the new standard, we could go back and fix up all of the project's existing logs to conform to that standard, and then generate a post-facto Contribulyzer page covering the entire history of the project. This we did in two steps: first we found all the "@" signs in the log messages, to detect places where we mentioned someone's email address (since we often used people's email addresses when crediting them), and then we searched again for just the names—without the email addresses—harvested from the first search. The resultant list of log messages numbered about one thousand, and with the help of a few volunteers (plus some rather labyrinthine editing macros), we were able to get them all into the new format in about one night.

Thus, the proposal of the new standard coincided with a demonstration of what it could do for us: we had the full Contribulyzer pages up and running from the moment the Contribulyzer was announced to the team. This made its benefits immediately apparent, and made the new log message formatting requirements seem like a small price to pay by comparison.

The Limits of the Contribulyzer

There is a famous saying, much used in open source projects but surely long predating them:

> Do not let the perfect be the enemy of the good.

The Contribulyzer could do much more than it currently does. It's really the beginnings of a complete activity-tracking system. In an ideal world, it would gather information from the mailing list archives and bug tracker as well as from the revision control system. We would be able to jump from a log message that mentions a contributor to the mailing list thread where that contributor discusses the change with other developers—and vice versa, that is, jump from the mailing list thread to the commit. Similarly, we would be able to gather statistics on what percentage of a given person's tickets in the bug tracker resulted in commits or in non-trivial discussion threads (thus telling us that this person's bug reports should get comparatively more weight in the future, since she seems to be an effective reporter).

The point would not be to make a ratings system; that would be useless, and maybe even destructive, since it would suffer from inflationary pressures and tempt people into reductively quantitative comparisons of participants. Rather, the point would be to make it easy to find out more about a person *once you already know you're interested in her*.

Everyone who participates in an open source project leaves a trail. Even asking a question on a mailing list leaves a trail of at least one message, and possibly more if a thread develops. But right now these trails are implicit: one must trawl through archives and databases and revision control histories by hand in order to put together a reasonably complete picture of a given person's activity.

The Contribulyzer is a small step in the direction of automating the discovery of these trails. I included it in this chapter as an example of how even a minor bit of automation can make a noticeable difference in a team's ability to collaborate. Although the Contribulyzer covers only revision control logs, it still saves us a lot of time and mental energy, especially because the log messages often contain links to the relevant bug-tracker tickets and mailing list threads—so if we can just get to the right log messages quickly, the battle is already half-won.

I don't want to claim too much for the Contribulyzer; certainly there are many aspects of running an open source project that it doesn't touch. But it significantly reduces our workload when evaluating potential new maintainers, and therefore makes us more likely to do such evaluations in the first place. For one day's investment in coding, that's not a bad payoff.

Writing metacode rarely feels as productive as writing code, but it's usually worth it. If you've correctly identified a problem and see a clear technical solution, then a one-time effort now can bring steady returns over the life of the project.

Commit Emails and Gumption Sinks

This next example shows what can happen when a team doesn't pay enough attention to tool usage. It's about how a seemingly trivial interface decision can have large effects on how people behave. First, some background.

Most open source projects have a *commit email list*. The list receives an email every time a change enters the master repository, and the mail is generated automatically by the repository. Typically each one shows the author of the change, the time the change was made, the associated log message, and the line-by-line change itself (expressed in the "patch" format mentioned earlier, except that for historical reasons, in this context the patch is called a "diff"). The email may also include URLs, to provide a permanent reference for the change or for some of its subparts.

Here's a commit email from the Subversion project:

```
From: dionisos@tigris.org
Subject: svn commit: r30009 - trunk/subversion/libsvn_wc
To: svn@subversion.tigris.org
Date: Sat, 22 Mar 2008 13:54:38 -0700

Author: dionisos
Date: Sat Mar 22 13:54:37 2008
New Revision: 30009
```

```
Log:
Fix issue #3135 (property update on locally deleted file breaks WC).

* subversion/libsvn_wc/update_editor.c (merge_file): Only fill WC file
   related entry cache-fields if the cache will serve any use (ie
   when the entry is schedule-normal).

Modified:
   trunk/subversion/libsvn_wc/update_editor.c

Modified: trunk/subversion/libsvn_wc/update_editor.c
URL: http://svn.collab.net/viewvc/svn/trunk/subversion/libsvn_wc/update_editor.
c?pathrev=30009&r1=30008&r2=30009
==============================================================================
--- trunk/subversion/libsvn_wc/update_editor.c Sat Mar 22 07:33:07 2008 (r30008)
+++ trunk/subversion/libsvn_wc/update_editor.c Sat Mar 22 13:54:37 2008 (r30009)
@@ -2621,8 +2621,10 @@ merge_file(svn_wc_notify_state_t *conten
   SVN_ERR(svn_wc__loggy_entry_modify(&log_accum, adm_access,
                                      fb->path, &tmp_entry, flags, pool));

-  /* Log commands to handle text-timestamp and working-size */
-  if (!is_locally_modified)
+  /* Log commands to handle text-timestamp and working-size,
+     if the file is - or will be - unmodified and schedule-normal */
+  if (!is_locally_modified &&
+      (fb->added || entry->schedule == svn_wc_schedule_normal))
     {
       /* Adjust working copy file unless this file is an allowed
          obstruction. */
```

When done right, commit emails are a powerful collaboration tool for software projects. They're a perfect marriage of automated information flow and human participation. Each change arrives in the developer's mailbox packaged as a well-understood, discrete unit: an email. The developer can view the change using a comfortable and familiar interface (her mailreader), and if she sees something that looks questionable, she can reply, quoting just the parts of the change that interest her, and her reply will automatically be put into a thread that connects the change to hers and everyone else's comments on it. Thus, by taking advantage of the data management conventions already in place for email, people (and other programs) can conveniently track the fallout from any given change.[*]

However, another project I'm a member of, GNU Emacs,[†] does things a little differently. Partly for historical reasons, and partly because of the way its version control system[‡] works, each commit to GNU Emacs generates *two* commit emails: one showing the log message, and the other containing the diff itself.

---

[*] For more on this practice, see *http://producingoss.com/en/vc.html#commit-emails* and *http://producingoss.com/en/setting-tone.html#code-review*.

[†] GNU Emacs is a text editing tool favored by many programmers, and is one of the oldest continuously maintained free software programs around. See *http://www.gnu.org/software/emacs/* for more.

[‡] CVS (*http://www.nongnu.org/cvs/*).

The log message email looks like this:

```
From: Juanma Barranquero <lekkktu@gmail.com>
Subject: [Emacs-commit] emacs/lisp info.el
To: emacs-commit@gnu.org
Date: Sat, 08 Mar 2008 00:09:29 +0000

CVSROOT:    /cvsroot/emacs
Module name:    emacs
Changes by:    Juanma Barranquero <lektu>    08/03/08 00:09:29

Modified files:
    lisp           : info.el

Log message:
    (bookmark-make-name-function, bookmark-get-bookmark-record):
    Pacify byte-compiler.

CVSWeb URLs:
http://cvs.savannah.gnu.org/viewcvs/emacs/lisp/info.el?cvsroot=emacs&r1=1.519&r2=1.520
```

And the diff email looks like this:

```
From: Juanma Barranquero <lekkktu@gmail.com>
Subject: [Emacs-diffs] Changes to emacs/lisp/info.el,v
To: emacs-diffs@gnu.org
Date: Sat, 08 Mar 2008 00:09:29 +0000

CVSROOT:    /cvsroot/emacs
Module name:    emacs
Changes by:    Juanma Barranquero <lektu>    08/03/08 00:09:29

Index: info.el
===================================================================
RCS file: /cvsroot/emacs/emacs/lisp/info.el,v
retrieving revision 1.519
retrieving revision 1.520
diff -u -b -r1.519 -r1.520
--- info.el    7 Mar 2008 19:31:59 -0000    1.519
+++ info.el    8 Mar 2008 00:09:29 -0000    1.520
@@ -3375,6 +3375,8 @@

 (defvar tool-bar-map)
 (defvar bookmark-make-record-function)
+(defvar bookmark-make-name-function)
+(declare-function bookmark-get-bookmark-record "bookmark" (bookmark))

 ;; Autoload cookie needed by desktop.el
 ;;;###autoload
```

Together, those two emails contain the same information as a single one like the one I showed earlier from the Subversion project. But the key word is *together*. They are not together; they are separate. Although the programmer committed a single change,[*] there is no single email containing everything a reviewer would need to understand and review that change. To review a change, you need the log message so that you can understand the general intent of the change, and the diff so that you can see whether the actual code edits match that intent.

It turns out that if people don't have both of these things in one place, they're much less likely to review changes.

Or so it seems from my highly rigorous* survey comparing the two projects. In February 2008, there were 207 unique threads (from 908 messages) on the Subversion development list. Of these, 50 were follow-up threads to commit emails. So by one reasonable measurement, 24% of development list attention goes to commit review (or if you want to count messages instead of threads, then a little over 5%). Note that follow-ups to Subversion commit emails are automatically directed to the main development list via a Reply-to header, so the development list is the right data set to use.

Meanwhile, in February 2008, the Emacs development list had 491 unique threads (from 3,158 messages), of which, apparently, zero were review mails.

Stunned at this result, I loosened my filters for detecting review mails and scanned again. This time I came up with, at most, 49 emails. But spot-checking those 49 showed that most seemed to be reviews of patches posted to the list from elsewhere, rather than reviews based on commit emails; only two were definite review mails. However, even if we count all 49 (which is almost certainly overly permissive), that's at most 10% of the development list traffic being commit reviews (or 1.5%, if we count messages instead of threads). Because Emacs does not automatically redirect commit email follow-ups to the main development list, I also searched in the commits list and the diffs list archives. I found no review follow-ups there in February, and exactly two in March, both of which were from me.

Now, the two projects have different commit rates and different traffic levels on their development lists. But we can partially control for this by approaching the question from the other side: what percentage of commits gets reviewed? In February 2008, Subversion had 274 unique commits, and Emacs had 807.† Thus, the ratio of review threads to commits for Subversion is about 18%, and for Emacs is somewhere between 0% and 6% (but probably tending low, around .2%, if there really were only two true review mails).

Something is happening here, something that makes one project much more likely than the other to do peer review. What is it?

---

* Some people use the word *changeset* for what I'm talking about here: the situation where perhaps multiple files were modified, but the modifications are all part of a single logical group. We can call that overall group a "change" or a "changeset"; the two words mean the same thing in this context.

* Read: "extremely anecdotal."

† If you're checking these numbers against primary sources, note that when counting Emacs commits, you shouldn't count commits that affect only `ChangeLog` files, because (due to certain oddities of the way the Emacs project uses its version control system) those are duplicates of other commits.

I cannot prove it, of course, but I think it's simply the fact that each Emacs commit arrives separated into two emails: one for the log message and another for the diff. There is no way, using a normal mail-reading interface with no extensive customizations, to view the log message and the diff at the same time. Thus, *there is no convenient way to review a change*. It's not that review is impossible, or even hard. It's neither: if I wanted to, I could review all the Emacs commits, and so could the other developers. But each review would require shuffling back and forth between two emails, or clicking on the URL in the shorter email and waiting for a page to load. In practice, it's too much trouble, and I can never be bothered. Apparently, neither can anyone else.

There's a sobering lesson here: adding a few seconds of overhead to a common task is enough to make that task uncommon. Your team isn't lazy, just human. Put light switches at roughly shoulder level and people will happily turn off lights when they leave a room; put the switches at knee level, and your electricity bill will skyrocket.

What is the cost to a project of not getting code review? Pretty high, I think. Looking over the Subversion commits for that month, 55 indicate that they follow up to a previous specific commit, and 35 bear a special marker (see *http://subversion.tigris.org/hacking.html#crediting*) indicating that they fix problems found by someone else. In my experience with the project, such suggestions are usually the result of commit review. Thus, probably somewhere between 12% and 20% of the commits made in Subversion result from review of previous commits. I cannot easily come up with the comparable number for Emacs, because Emacs does not have standardized attribution conventions the way Subversion does. But I've been watching Emacs development for a long time, and while it's clear that some percentage of commits there results from reviews of previous commits (or from just stumbling across problematic code in the course of other work), I do not believe it reaches 12% to 20%.

Besides, the benefits of timely commit review cannot be measured solely in further code changes. Commit review sustains morale, sharpens people's skills (because they're all learning from each other), reinforces a team's ability to work together (because everyone gets accustomed to receiving constructive criticism from everyone else), and spurs participation (because review happens in public, thus encouraging others to do the same). To deprive a team of these benefits because of a trivial user interface choice is a costly mistake.

They're Staying Away in Droves: A Tale of Two Translation Interfaces

In 2005, I wrote a book on managing open source projects,[*] much of which is about exactly what this chapter is about: setting up tools to make teams more effective.

Awhile after the book was published, and its full contents placed online, volunteers started showing up to translate it into other languages. Naturally, I was extremely pleased

---

[*] *Producing Open Source Software* (O'Reilly, *http://producingoss.com/*).

by this, and wanted to help them as much as possible, so I set up some technical infrastructure to enable the translators to collaborate. Each target language generally had more than one volunteer working on it, and having the translators for a given language work together was simply common sense. In addition, certain aspects of the translation processes were common across all the languages, so it made sense even for translators of different languages to be able to see each other's work.

It all sounds good on paper, doesn't it? But what actually happened was a deep lesson in the consequences of failing to understand a team's tool needs.

What I Gave Them

The infrastructure I set up for the translators was basically an extension of what I had used myself to write the book: a collection of XML master files stored in a Subversion repository, written to the DocBook DTD, and converted to output formats using a collection of command-line programs driven by some Makefiles.

Let me try saying that in English:

I wrote the book in a format (XML) that, while a bit awkward to work in, is very easy to produce both web pages and printable books from. For example, here's what the beginning of Chapter 3 looks like in the original XML:

```
<chapter id="technical-infrastructure">

<title>Technical Infrastructure</title>

<simplesect>

<para>Free software projects rely on technologies that support the
selective capture and integration of information.  The more skilled
you are at using these technologies, and at persuading others to use
them, the more successful your project will be.  This only becomes
more true as the project grows.  Good information management is what
prevents open source projects from collapsing under the weight of
Brooks' Law,<footnote><para>From his book <citetitle>The Mythical Man
Month</citetitle>, 1975.  See <ulink
url="http://en.wikipedia.org/wiki/The_Mythical_Man-Month"/> and <ulink
url="http://en.wikipedia.org/wiki/Brooks_Law"/>.</para></footnote>
which states that adding manpower to a late software project makes it
later.  Fred Brooks observed that the complexity of a project
increases as the <emphasis>square</emphasis> of the number of ...
```

Here is what that becomes after being run through the converter to produce web output:

> Chapter 3**: Technical Infrastructure**
> Free software projects rely on technologies that support the selective capture and integration of information. The more skilled you are at using these technologies, and at persuading others to use them, the more successful your project will be. This only becomes more true as the project grows. Good information management is what prevents open source projects from collapsing under the weight of Brooks' Law,[8] which states that adding manpower to a late software project makes it later. Fred

> Brooks observed that the complexity of a project increases as the *square* of the number of…

One can certainly see the similarity between the two texts, but there's no doubt that the XML master files are harder to read. For me, that was OK: the inconvenience of working in XML was outweighed by the ability to produce different types of output conveniently.

I managed the collection of XML files using a version control system called Subversion, which is very familiar to programmers but less widely known outside the programming community. Similarly, the converters—one for web output and another for printed output—were powerful and flexible tools, but somewhat hard to use, especially for non-programmers. Just to set them up you must first have various other supporting programs installed on your computer. The overall process is difficult under any circumstances, but it's a bit less difficult if your computer runs on a Unix-like operating system (again, common among programmers but not other demographics), and a bit more difficult on Microsoft Windows.

All of this was fine when I was the only author. I'm a programmer anyway, I was somewhat familiar with all those tools already, and since I was putting in a lot of effort to write the book in the first place, adding a little extra overhead to learn some tools was well worth it. I knew it would pay off in the long run, and it did—for me.

But the situation for the volunteer translators was completely different. They were *not* all programmers, most had no prior experience with these tools, and they therefore faced a steep learning curve right at the start (always the worst place to face a steep learning curve!). Furthermore, since any given translation was being done by several translators, for each individual the ratio of tool-learning to real work was worse than it had been for me: each translator paid the same tool-learning cost I paid, but each translator was not writing an entire book, and therefore had less benefit across which to amortize that cost. The translators showed up with one particular kind of expertise—the ability and desire to take English text and rewrite it in another language—and found themselves asked instead to take on a task completely unrelated to that expertise: that of learning a whole new set of tools, tools that they might or might not use elsewhere in their lives.

The wonder is not that some of them rebelled, but rather that we got any useful translations at all. We did get some: as of this writing, the German translation is draft-complete, the Japanese is about 80% done, the Spanish about 60%, and so forth. (The French are a more interesting case; we'll get to them in a moment.) But seeing what the translators had to go through to get this far makes me wonder how much *more* could have been done if I hadn't imposed such a high tool overhead on the process. Take a look at the translator guidelines I wrote up at *http://producingoss.com/translations.html#guidelines* to get a sense of how much engineering was required of the participants. (I'd just show those guidelines here, but they'd take up too much room, which already tells you something.)

What I Should Have Given Them

Unfortunately, for a long time I didn’t realize how punishing the aforementioned system was. I knew it was less than ideal, but it’s very hard to tell why any particular translator isn’t doing as much work as he said he would—and it’s impossible to notice when a translator doesn’t volunteer in the first place because he’s daunted by the requirements.

Eventually, though, we got a wake-up call from France:

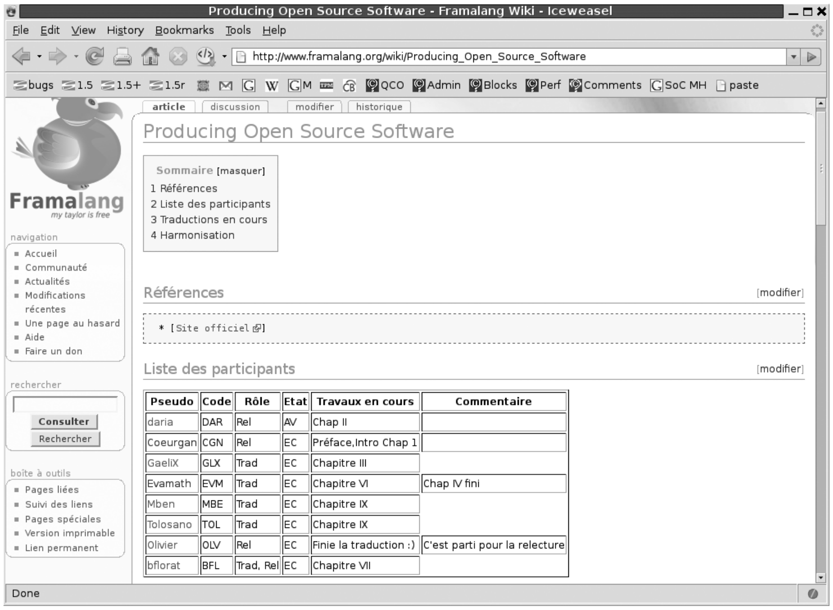
```
From: Bertrand Florat <...>
Subject: French translation
To: ProducingOSS Translators Mailing List
Date: Sun, 17 Feb 2008 16:54:32 +0100

Hi Karl,

Just to keep you in touch, Etienne and myself moved to the Framalang
wiki to finish the book translation. We plan to port all its content to
docbook format once done (it's about 90% done now).

I think it could be a good idea to link the French translation from
http://www.producingoss.com/fr/ to
http://www.framalang.org/wiki/Producing_Open_Source_Software so people
can benefit from the almost-done translation right now, and it could
bring new translators.

What do you think ?

Cheers,

Bertrand
```

In other words, the French translators decided to *route around* the technical infrastructure that I had set up to help them translate the book. Instead, they copied all the English data over to a Wiki, where they could do their editing in a convenient, familiar environment. When they’re done,* they plan to port the translation back to XML.

That’s a pretty searing indictment of my infrastructure, I’d say. And looking at the Framalang Wiki site (*http://www.framalang.org/wiki/Producing_Open_Source_Software*), I can’t really blame them. It is much more suited to the task of coordinating translators than the ad hoc system I’d set up. At Framalang, the original text and translation are displayed next to each other, in different colors. There are special functions for tracking who is responsible for what portions, for giving feedback, for providing a centralized list of agreed-on translations for common terms, for labeling what work remains to be done, and so on. All of these things can be accomplished in my ad hoc system, too, of course, but the difference is that they aren’t handed to the translators for free: the team is forced to reinvent the wheel over and over, whereas Framalang just gives out the keys to the car ().

---

* In fact, they finished their first draft of the translation while I was writing this chapter.



Producing Open Source Software

Sommaire [masquer]
1 Références
2 Liste des participants
3 Traductions en cours
4 Harmonisation

Références [modifier]

* [Site officiel]

Liste des participants [modifier]

Pseudo	Code	Rôle	Etat	Travaux en cours	Commentaire
daria	DAR	Rel	AV	Chap II	
Coeurgan	CGN	Rel	EC	Préface,Intro Chap 1	
GaeliX	GLX	Trad	EC	Chapitre III	
Evamath	EVM	Trad	EC	Chapitre VI	Chap IV fini
Mben	MBE	Trad	EC	Chapitre IX	
Tolosano	TOL	Trad	EC	Chapitre IX	
Olivier	OLV	Rel	EC	Finie la traduction :)	C'est parti pour la relecture
bflorat	BFL	Trad, Rel	EC	Chapitre VII	

Framalang translation Wiki

Bertrand's announcement prompted another translator to voice some dissatisfaction with the homegrown infrastructure:

```
From: "Manuel Barkhau" <mbarkhau@googlemail.com>
Subject: Re: French translation
To: bertrand@florat.net
Cc: producingoss-translators@red-bean.com
Date: Sun, 17 Feb 2008 17:21:35 +0100
Reply-To: mbarkhau@gmail.com

Bonjour mon amis,

on a different note, maybe you would like to share you're experience
with the wiki format vs. subversion. How much participation have you
received apart from the main translators. How many translators joined
your effort, quality of their work etc.
Something I especially miss with the current format, are reader
statistics and feedback.

ciao,
Manuel
```

Seeing this on-list correspondence made me think back to private conversations I'd had with some of the other translators. Many of them had had problems working in XML. Even though the translators themselves generally understood the format well enough to work with it, their editing tools often did not. Particularly on Microsoft Windows, their word processors would sometimes mess up the XML. This was something I'd never experienced myself, but that's only to be expected: I'd chosen the format in the first place, so naturally it worked well with *my* tools.

Looking back, I can correlate the conversations I had with frustrated individual translators with dropoffs in their contributions; a few of them actually stopped doing translation altogether. Of course, some turnover is to be expected in any volunteer group, but the important thing is to notice when dropoffs happen for a specific reason. In this case, they were, but because I was personally comfortable with the tools, it didn't occur to me for a long time that they were a significant gumption sink for others. The result is that I effectively turned down an unknown amount of volunteer energy: many of the translations that were started have still not been finished, and some of the responsibility for that has to lie with the awkward tool requirements I imposed.

Conclusion

From the preceding examples, we can distill a few principles of collaboration tools:

- Good tools get the fundamental units of information right.

  For example, with commit emails the fundamental unit is the change (or "changeset"). When the tool failed to treat each change as a single logical entity, and instead divided it across multiple interface access points (in this case, emails), people became less inclined to inspect the change.

  Getting the fundamental units right does not merely make individual tool usage better. It improves collaboration, because it gives the team members a common vocabulary for talking about the things they're working with.

- When you see necessary tasks being repeatedly postponed ("I'll try to get to it this weekend..."), it's often a sign that the tools are forcing people into a higher per-task commitment level than they're comfortable with. Fix the tools and the postponements may go away.

  The Contribulyzer didn't help with any of the interesting parts of evaluating a contributor's work: one still had to review his actual code changes, after all. But it did remove the *uninteresting* part, which was the manual search through the revision control logs for that contributor's changes. That dismaying prospect, although less work than the actual evaluation in most cases, was significant enough to be a gumption sink, in part because it's simply boring.

  Removing it, and replacing it with a pleasant, lightweight interface, meant that team members no longer had to make an emotional commitment to a major effort when deciding to evaluate a contributor. Instead, it became like a decision to sharpen a pencil in an electric sharpener: there's no effort involved in starting up the task, you can stop in the middle and resume later if you want, and you can do as many pencils as you want without multiplying the start-up overhead.

- When your team starts routing around the tools you've offered them, pay attention: it may be a sign that you're offering the wrong tools.

  Not always, of course: there can be cases where people don't initially appreciate the benefits a tool will bring, and need to be taught. But even then, the tool is probably at least partly to blame, in that it didn't make the benefits clear enough from the outset. At the very least, grass-roots workarounds should make you sit up and investigate what's motivating them. When the French translators of my book defected to a Wiki-based interface, at considerable start-up cost to themselves, that was a clear sign that something was wrong with the interface I'd been providing.

- A change in a team's demographics may require a change in tools.

  With those XML book chapters, as long as the "team" consisted of programmers who had prior experience working with version-controlled XML files—which was the case for as long as the team consisted only of me—the tools I'd set up were fine. But when new people came on board, simply extending the existing framework was not the right answer. Instead, the tool environment needed to be rethought from scratch.

- Partial automation is often enough to make a difference.

  Designers of collaboration tools love 100% solutions: if something *can* be fully automated, they feel, then it should be. But in practice, it may not always be worth the effort; a hybrid model is often the better choice. If you can get 90% of the benefit with 10% of the effort, then do just that much, and remember that the team is willing to make up the difference as long as they'll get clear benefits from it. The Contribulyzer, for example, completely depends on humans following certain crediting conventions when they write log messages. It takes time and effort to teach people these conventions. But the alternative—a fully automated intelligent project watcher that infers connections between mailing list messages and commits—would require a huge amount of effort with no guarantee of perfect reliability anyway.

  Actually, that's one of the often overlooked benefits of hybrid tools: because they cause humans to stay engaged with the data at more points, such tools can prove *more* reliable than allegedly fully automated ones. The point of good tools is not to make humans unnecessary, but to make them happy. (Some might say "…to make them more efficient," but an inefficient team is rarely happy anyway, and you surely wouldn't want a team that is efficient but unhappy.)

Despite the benefits to be had from good tools, my experience is that most teams' actual tool usage lags behind their potential tool usage. That is, most teams are failing to take advantage of potentially large multiplier effects. I think this is because of two built-in biases shared by most humans.

One bias is that trying new things costs too much. Since most new things are likely to bring little benefit, groups tend to be selective about adopting new tools, for the overhead of changing habits can be a drag on both productivity and enjoyment. (I do not mean to imply that this bias is unreasonable. Indeed, it is usually well founded, and I share it myself!)

The other bias is subtler. Because successful tools quickly become second nature—blending into the mental landscape until they are considered just part of "the way things are done"—people can easily forget the effect a tool had *at the time it was introduced*. Examples of this phenomenon are not hard to find, starting with the text editor in which I'm writing these words. Fifty years ago, the idea of editing and reshaping a text while simultaneously writing its first draft would have been a writer's wishful dream;* now it is so commonplace that it's probably hard for schoolchildren in the developed world to imagine writing any other way. And even computer text-editing starts to look like a mere incremental improvement when compared to a truly transformative tool like universal literacy. Imagine trying to get a team to cooperate without depending on its members being able to read and write!

And yet, consider how your team would react if someone came along and advocated the following:

> I know a way to make you many times more productive. Do what I tell you, and your group's ability to collaborate will increase beyond your most optimistic projections. Here's the plan: first, everyone's going to learn this set of symbols, called an *alphabet*. Each symbol is called a *letter*, and the letters map, more or less, to sounds in a language that is native to some of you and non-native to others. Once you've learned the basic sounds of the letters, I'll show you how to arrange them in groups to form written words. Once you've memorized between 5,000 and 10,000 of those words, you'll be able to transmit arbitrary sentences, and then your productivity will increase by *a whole lot*!

Literacy is a classic example of a high-investment, high-payoff tool. Everyone spends a lot of time in training, but after he makes it through that part, it pays off for years to come.

Usually, the tools that get people most excited are low-investment, high-payoff. But it would be a mistake to consider only such tools: literacy did pay off in the end, after all. A dedicated team can make non-trivial tool investments when necessary, especially if everyone bears the burden together and thus is able to strengthen his sense of community by learning the new tool as a group. A tool that makes obstacles go away and makes new things possible will justify a lot of investment. For people working together on a technical project, few things have as direct an effect on daily experience as the tools they work with. The more you understand their experience, the better tool choices you can make.

---

* Whether it results in *better* text is, of course, open to question.