Beautiful Code

(日本語)

In 2007, O'Reilly Media published the book *Beautiful Code: Leading Programmers Explain How They Think*, of which this is Chapter 2. Get the entire book if you can, though: it's 600 pages long, with 33 chapters, so I haven't read all of it yet, but the parts I've read so far have been terrific. All royalties from O'Reilly's sales of the treeware go to Amnesty International.

Chapter 2. Subversion's Delta Editor: Interface As Ontology

Karl Fogel

Examples of beautiful code tend to be local solutions to well-bounded, easily comprehensible problems, such as Duff's Device (http://en.wikipedia.org/wiki/Duff's_device) or *rsync's* rolling checksum algorithm (http://en.wikipedia.org/wiki/Rsync#Algorithm). This is not because small, simple solutions are the only beautiful kind, but because appreciating complex code requires more context than can be given on the back of a napkin.

Here, with the luxury of several pages to work in, I'd like to talk about a larger sort of beauty — not necessarily the kind that would strike a passing reader immediately, but the kind that programmers who work with the code on a regular basis would come to appreciate as they accumulate experience with the problem domain. My example is not an algorithm, but an interface: the programming interface used by the open source version control system Subversion (http://subversion.tigris.org) to express the difference between two directory trees, which is also the interface used to transform one tree into the other. In Subversion, its formal name is the C type `svn_delta_editor_t`, but it is known colloquially as the *delta editor*.

Subversion's delta editor demonstrates the properties that programmers look for in good design. It breaks down the problem along boundaries so natural that anyone designing a new feature for Subversion can easily tell when to call each function, and for what purpose. It presents the programmer with uncontrived opportunities to maximize efficiency (such as by eliminating unnecessary data transfers over the network) and allows for easy integration of auxiliary tasks (such as progress reporting). Perhaps most important, the design has proved very resilient during enhancements and updates.

And as if to confirm suspicions about the origins of good design, the delta editor was created by a single person over the course of a few hours (although that person was very familiar with the problem and the code base).

To understand what makes the delta editor beautiful, we must start by examining the problem it solves.

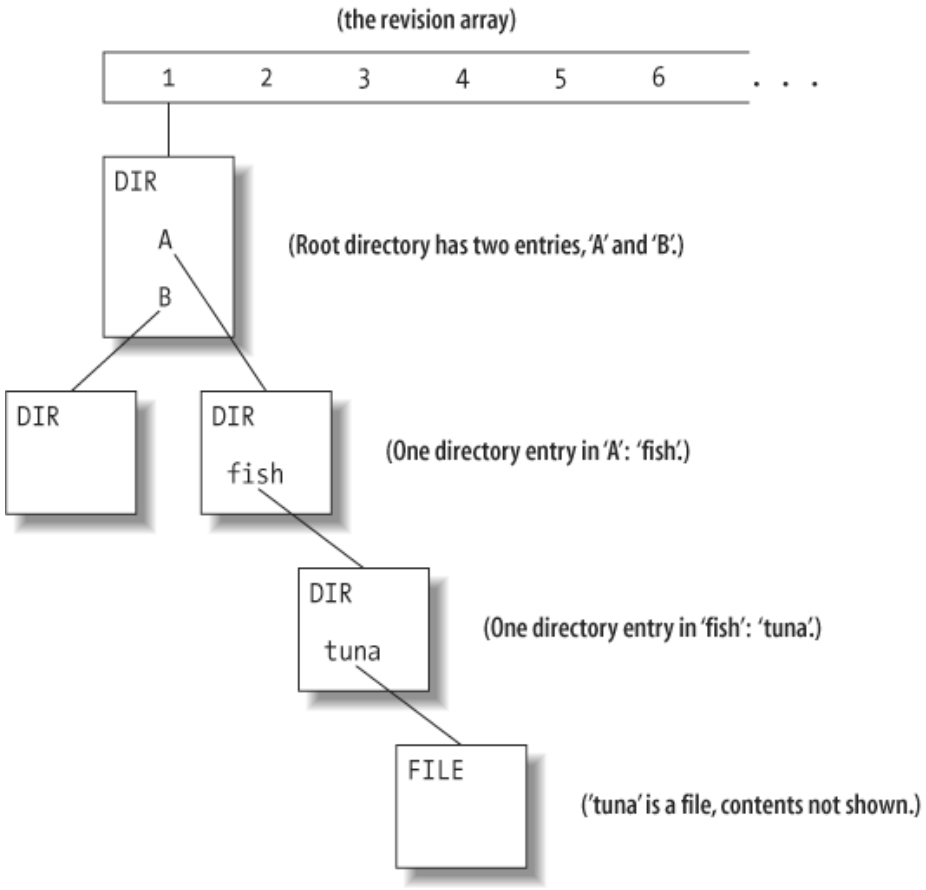
Version Control and Tree Transformation

Very early in the Subversion project, the team realized we had a general task that would be performed over and over: that of minimally expressing the difference between two similar (usually related) directory trees. As a version control system, one of Subversion's goals is to track revisions to directory structures as well as individual file contents. In fact, Subversion's server-side repository is fundamentally designed around directory versioning. A repository is simply a series of snapshots of a directory tree as that tree transforms over time. For each changeset committed to the repository, a new tree is created, differing from the preceding tree exactly where the changes are located and nowhere else. The unchanged

portions of the new tree share storage with the preceding tree, and so on back into time. Each successive version of the tree is labeled with a monotonically increasing integer; this unique identifier is called a *revision number*.

Think of the repository as an array of revision numbers, stretching off into infinity. By convention, revision 0 is always an empty directory. In the example below, revision 1 has a tree hanging off it (typically the initial import of content into the repository), and no other revisions have been committed yet. The boxes represent nodes in this virtual filesystem: each node is either a directory (labeled DIR in the upper-left corner) or a file (labeled FILE):

Figure 2-1. Conceptual view of revision numbers

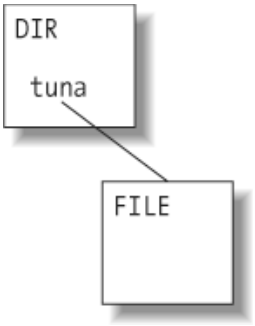


What happens when we modify *tuna?* First, we make a new file node, containing the latest text. The new node is not connected to anything yet, it's just hanging out there in space, with no name:

Figure 2-2. New node when just created

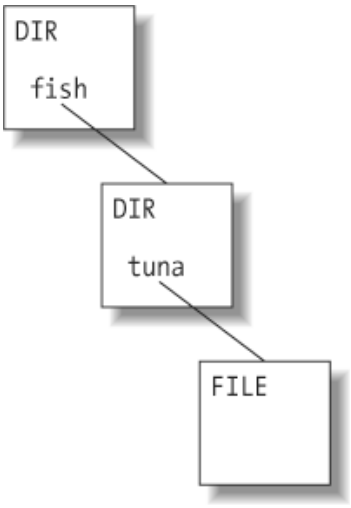
Next, we create a new revision of its parent directory. The subgraph is still not connected to the revision array:

Figure 2-3. Creation of new parent directory



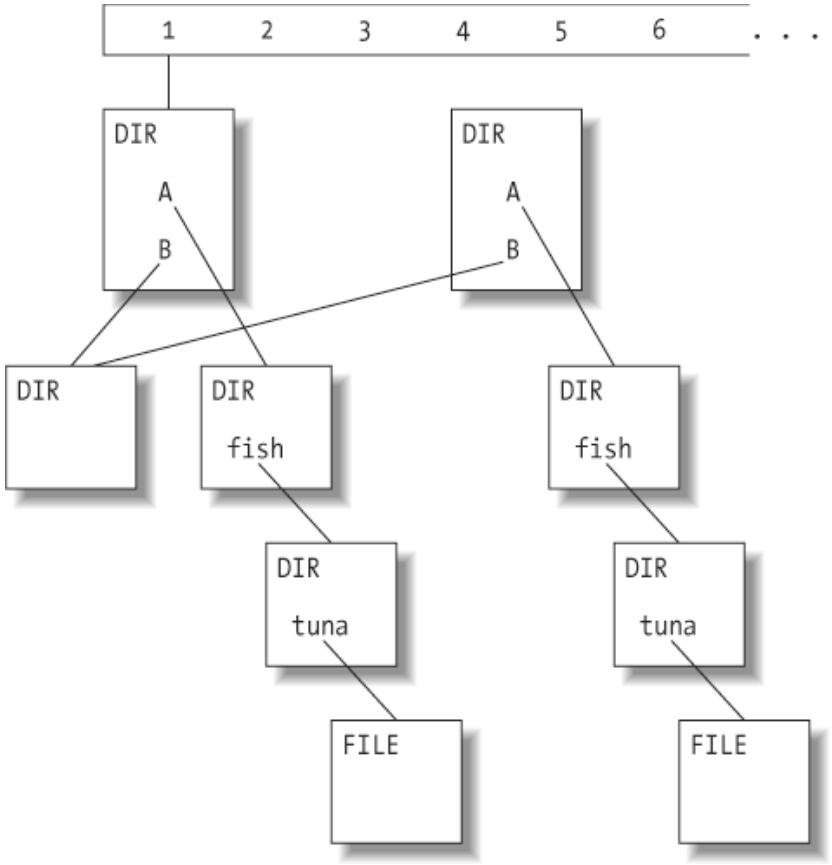
We continue up the line, creating a new revision of the next parent directory:

Figure 2-4. Continuing to move up, creating parent directories



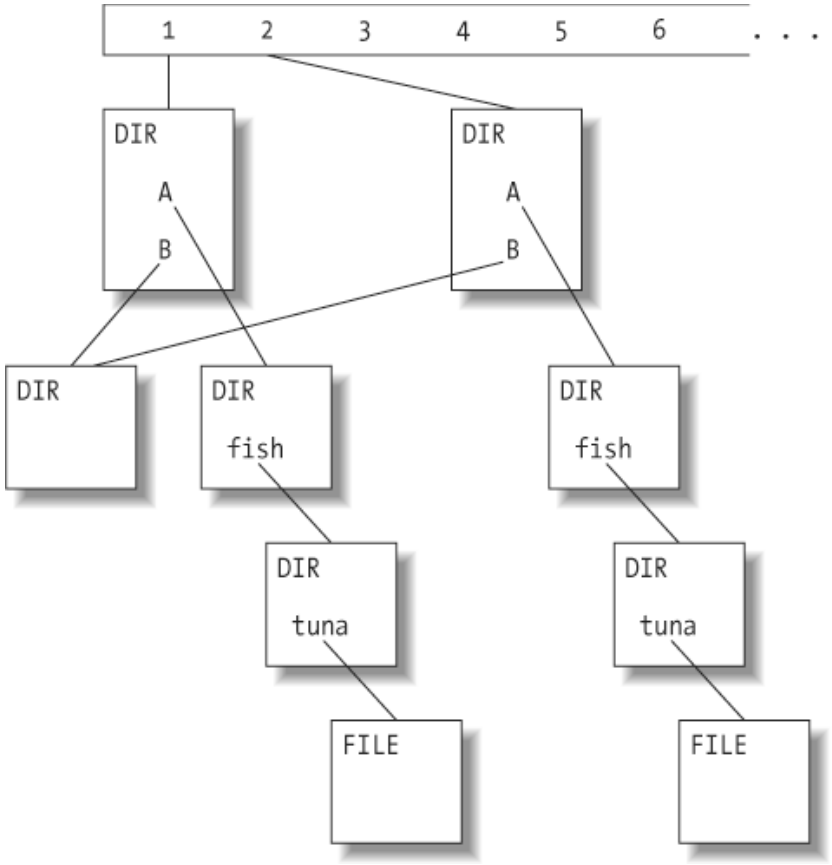
At the top, we create a new revision of the root directory. This new directory needs an entry to point to the "new" directory A, but since directory B hasn't changed at all, the new root directory also has an entry still pointing to the *old* directory B's node:

Figure 2-5. Complete new directory tree



Now that all the new nodes are written, we finish the "bubble up" process by linking the new tree to the next available revision in the history array, thus making it visible to repository users. In this case, the new tree becomes revision 2:

Figure 2-6. Finished revision: link to new tree



Thus each revision in the repository points to the root node of a unique tree, and the difference between that tree and the preceding one is the change that was committed in the new revision. To trace the changes, a program walks down both trees simultaneously, noting where entries point to different places. (For brevity, I've left out some details, such as saving storage space by compressing older nodes as differences against their newer versions.)

Although this tree-versioning model is all background to the main point of this chapter (the delta editor, which we'll come to soon), it has such nice properties that I considered making it the subject of its own chapter, as an example of beautiful code. Some of its attractive features are:

Easy reads

> To locate revision *n* of file */path/to/foo.txt*, one jumps to revision *n*, then walks down the tree to `/path/to/foo.txt`.

Writers don't interfere with readers

> As writers create new nodes, bubbling their way up to the top, concurrent readers cannot see the work in progress. A new tree becomes visible to readers only after the writer makes its final "link" to a revision number in the repository.

Tree structure is versioned

> The very structure of each tree is being saved from revision to revision. File and directory renames, additions, and deletions become an intrinsic part of the repository's history.

If Subversion were only a repository, this would be the end of the story. However, there's a client side, too: the *working copy*, which is a user's checked-out copy of some revision tree plus whatever local edits the user has made but not yet committed. (Actually, working copies do not always reflect a single revision tree; they often contain mixtures of nodes from different revisions. This turns out not to make much of a difference as far as tree transformation is concerned. So, for the purposes of this chapter, just assume that a working copy represents some revision tree, though not necessarily that of the latest revision.)

Expressing Tree Differences

The most common action in Subversion is to transmit changes between the two sides: from the repository to the working copy when doing an update to receive others' changes, and from the working copy to the repository when committing one's own changes. Expressing the difference between two trees is also key to many other common operations — e.g., diffing, switching to a branch, merging changes from one branch to another, and so on.

Clearly it would be silly to have two different interfaces, one for server→client and another for client→server. The underlying task is the same in both cases. A tree difference is a tree difference, regardless of which direction it's traveling over the network or what its consumer intends to do with it. But finding a natural way to express tree differences proved surprisingly challenging. Complicating matters further, Subversion supports multiple network protocols and multiple backend storage mechanisms; we needed an interface that would look the same across all of those.

Our initial attempts to come up with an interface ranged from unsatisfying to downright awkward. I won't describe them all here, but what they had in common was that they tended to leave open questions for which there were no persuasive answers.

For example, many of the solutions involved transmitting the changed paths as strings, either as full paths or path components. Well, what order should the paths be transmitted in? Depth first? Breadth first? Random order? Alphabetical? Should there be different commands for directories than for files? Most importantly, how would each individual command expressing a difference know that it was part of a larger operation grouping all the changes into a unified set? In Subversion, the concept of the overall tree operation is quite user-visible, and if the programmatic interface didn't intrinsically match that concept, we'd surely need to write lots of brittle glue code to compensate.

In frustration, I drove with another developer, Ben Collins-Sussman, from Chicago down to Bloomington, Indiana, to seek the advice of Jim Blandy, who had invented Subversion's repository model in the first place, and who has, shall we say, strong opinions about design. Jim listened quietly as we described the various avenues we'd explored for transmitting tree differences, his expression growing grimmer and grimmer as we talked. When we reached the end of the list, he sat for a moment and then politely asked us to scram for a while so he could think. I put on my jogging shoes and went running; Ben stayed behind and read a book in another room or something. So much for collaborative development.

After I returned from my run and showered, Ben and I went back into Jim's den, and he showed us what he'd come up with. It is essentially what's in Subversion today; there have been various changes over the years, but none to its fundamental structure.

The Delta Editor Interface

Following is a mildly abridged version of the delta editor interface. I've left out the parts that deal with copying and renaming, the parts related to Subversion properties (properties are versioned metadata, and are not important here), and parts that handle some other Subversion-specific bookkeeping. However, you can always see the latest version of the delta editor by visiting http://svn.collab.net/repos/svn/trunk/subversion/include/svn_delta.h. This chapter is based on r21731 (that is, revision 21731) at http://svn.collab.net/viewvc/svn/trunk/subversion/include/svn_delta.h?revision=21731.

To understand the interface, even in its abridged form, you'll need to know some Subversion jargon:

pools

> The `pool` arguments are memory pools — allocation buffers that allow a large number of objects to be freed simultaneously.

`svn_error_t`

> The return type `svn_error_t` simply means that the function returns a pointer to a Subversion

error object; a successful call returns a null pointer.

text delta

A text delta is the difference between two different versions of a file; you can apply a text delta as a patch to one version of the file to produce the other version. In Subversion, the "text" of a file is considered binary data — it doesn't matter whether the file is plain text, audio data, an image, or something else. Text deltas are expressed as streams of fixed-sized windows, each window containing a chunk of binary diff data. This way, peak memory usage is proportional to the size of a single window, rather than to the total size of the patch (which might be quite large in the case of, say, an image file).

window handler

This is the function prototype for applying one window of text-delta data to a target file.

baton

This is a `void *` data structure that provides context to a callback function. In other APIs, these are sometimes called `void *ctx`, `void *userdata`, or `void *closure`. Subversion calls them "batons" because they're passed around a lot, like batons in a relay race.

The interface starts with an introduction, to put a reader of the code in the right frame of mind. This text is almost unchanged since Jim Blandy wrote it in August of 2000, so the general concept has clearly weathered well:

```
/** Traversing tree deltas.
 *
 * In Subversion, we've got various producers and consumers of tree
 * deltas.
 *
 * In processing a `commit' command:
 * - The client examines its working copy data, and produces a tree
 *   delta describing the changes to be committed.
 * - The client networking library consumes that delta, and sends them
 *   across the wire as an equivalent series of network requests.
 * - The server receives those requests and produces a tree delta --
 *   hopefully equivalent to the one the client produced above.
 * - The Subversion server module consumes that delta and commits an
 *   appropriate transaction to the filesystem.
 *
 * In processing an `update' command, the process is reversed:
 * - The Subversion server module talks to the filesystem and produces
 *   a tree delta describing the changes necessary to bring the
 *   client's working copy up to date.
 * - The server consumes this delta, and assembles a reply
 *   representing the appropriate changes.
 * - The client networking library receives that reply, and produces a
 *   tree delta --- hopefully equivalent to the one the Subversion
 *   server produced above.
 * - The working copy library consumes that delta, and makes the
 *   appropriate changes to the working copy.
 *
 * The simplest approach would be to represent tree deltas using the
 * obvious data structure. To do an update, the server would
 * construct a delta structure, and the working copy library would
 * apply that structure to the working copy; the network layer's job
 * would simply be to get the structure across the net intact.
 *
 * However, we expect that these deltas will occasionally be too large
 * to fit in a typical workstation's swap area. For example, in
 * checking out a 200Mb source tree, the entire source tree is
 * represented by a single tree delta. So it's important to handle
 * deltas that are too large to fit in swap all at once.
 *
 * So instead of representing the tree delta explicitly, we define a
```

```
 * standard way for a consumer to process each piece of a tree delta
 * as soon as the producer creates it. The svn_delta_editor_t
 * structure is a set of callback functions to be defined by a delta
 * consumer, and invoked by a delta producer. Each invocation of a
 * callback function describes a piece of the delta --- a file's
 * contents changing, something being renamed, etc.
 */
```

Then comes a long, formal documentation comment, followed by the interface itself, which is a callback table whose invocation order is partially constrained:

```
/** A structure full of callback functions the delta source will invoke
 * as it produces the delta.
 *
 * Function Usage
 * ==============
 *
 * Here's how to use these functions to express a tree delta.
 *
 * The delta consumer implements the callback functions described in
 * this structure, and the delta producer invokes them. So the
 * caller (producer) is pushing tree delta data at the callee
 * (consumer).
 *
 * At the start of traversal, the consumer provides edit_baton, a
 * baton global to the entire delta edit.
 *
 * Next, if there are any tree deltas to express, the producer should
 * pass the edit_baton to the open_root function, to get a baton
 * representing root of the tree being edited.
 *
 * Most of the callbacks work in the obvious way:
 *
 * delete_entry
 *  add_file
 * add_directory
 * open_file
 * open_directory
 *
 * Each of these takes a directory baton, indicating the directory
 * in which the change takes place, and a path argument, giving the
 * path (relative to the root of the edit) of the file,
 * subdirectory, or directory entry to change. Editors will usually
 * want to join this relative path with some base stored in the edit
 * baton (e.g. a URL, a location in the OS filesystem).
 *
 * Since every call requires a parent directory baton, including
 * add_directory and open_directory, where do we ever get our
 * initial directory baton, to get things started? The open_root
 * function returns a baton for the top directory of the change. In
 * general, the producer needs to invoke the editor's open_root
 * function before it can get anything of interest done.
 *
 * While open_root provides a directory baton for the root of
 * the tree being changed, the add_directory and open_directory
 * callbacks provide batons for other directories. Like the
 * callbacks above, they take a parent_baton and a relative path
 * path, and then return a new baton for the subdirectory being
 * created / modified --- child_baton. The producer can then use
 * child_baton to make further changes in that subdirectory.
 *
 * So, if we already have subdirectories named `foo' and `foo/bar',
 * then the producer can create a new file named `foo/bar/baz.c' by
 * calling:
 *
 *     - open_root () --- yielding a baton root for the top directory
 *
 *     - open_directory (root, "foo")
```

```
 *
 *     - open_directory (f, "foo/bar") --- yielding a baton b for `foo/bar'
 *
 *     - add_file (b, "foo/bar/baz.c")
 *
 * When the producer is finished making changes to a directory, it
 * should call close_directory. This lets the consumer do any
 * necessary cleanup, and free the baton's storage.
 *
 * The add_file and open_file callbacks each ret urn a baton
 * for the file being created or changed. This baton can then be
 * passed to apply_textdelta to change the file's contents.
 * When the producer is finished making changes to a file, it should
 * call close_file, to let the consumer clean up and free the baton.
 *
 * Function Call Ordering
 * ======================
 *
 * There are five restrictions on the order in which the producer
 * may use the batons:
 *
 * 1. The producer may call open_directory, add_directory,
 *    open_file, add_file at most once on any given directory
 *        entry. delete_entry may be called at most once on any given
 *    directory entry and may later be followed by add_directory or
 *       add_file on the same directory entry. delete_entry may
 *    not be called on any directory entry after open_directory,
 *   add_directory,open_file or add_file has been called on
 *    that directory entry.
 *
 * 2. The producer may not close a directory baton until it has
 *    closed all batons for its subdirectories.
 *
 * 3. When a producer calls open_directory or add_directory,
 *    it must specify the most recently opened of the currently open
 *    directory batons. Put another way, the producer cannot have
 *    two sibling directory batons open at the same time.
 *
 * 4. When the producer calls open_file or add_file, it must
 *    follow with any changes to the file (using apply_textdelta),
 *    followed by a close_file call, before issuing any other
 *    file or directory calls.
 *
 * 5. When the producer calls apply_textdelta, it must make all of
 *    the window handler calls (including the NULL window at the
 *    end) before issuing any other svn_delta_editor_t calls.
 *
 * So, the producer needs to use directory and file batons as if it
 * is doing a single depth-first traversal of the tree.
 *
 * Pool Usage
 * ==========
 *
 * Many editor functions are invoked multiple times, in a sequence
 * determined by the editor "driver". The driver is responsible for
 * creating a pool for use on each iteration of the editor function,
 * and clearing that pool between each iteration. The driver passes
 * the appropriate pool on each function invocation.
 *
 * Based on the requirement of calling the editor functions in a
 * depth-first style, it is usually customary for the driver to similarly
 * nest the pools. However, this is only a safety feature to ensure
 * that pools associated with deeper items are always cleared when the
 * top-level items are also cleared. The interface does not assume, nor
 * require, any particular organization of the pools passed to these
 * functions.
 */
typedef struct svn_delta_editor_t
{
```

```
/** Set *root_baton to a baton for the top directory of the change.
 * (This is the top of the subtree being changed, not necessarily
 * the root of the filesystem.) Like any other directory baton, the
 * producer should call close_directory on root_baton when they're
 * done.
 */
svn_error_t *(*open_root)(void *edit_baton,
                          apr_pool_t *dir_pool,
                          void **root_baton);



/** Remove the directory entry named path, a child of the directory
 * represented by parent_baton.
 */
svn_error_t *(*delete_entry)(const char *path,
                             void *parent_baton,
                             apr_pool_t *pool);



/** We are going to add a new subdirectory named path. We will use
 * the value this callback stores in *child_baton as the
 * parent_baton for further changes in the new subdirectory.
 */
svn_error_t *(*add_directory)(const char *path,
                              void *parent_baton,
                              apr_pool_t *dir_pool,
                              void **child_baton);

/** We are going to make changes in a subdirectory (of the directory
 * identified by parent_baton). The subdirectory is specified by
 * path. The callback must store a value in *child_baton that
 * should be used as the parent_baton for subsequent changes in this
 * subdirectory.
 */
svn_error_t *(*open_directory)(const char *path,
                               void *parent_baton,
                               apr_pool_t *dir_pool,
                               void **child_baton);

/** We are done processing a subdirectory, whose baton is dir_baton
 * (set by add_directory or open_directory). We won't be using
 * the baton any more, so whatever resources it refers to may now be
 * freed.
 */
svn_error_t *(*close_directory)(void *dir_baton,
                                apr_pool_t *pool);
/** We are going to add a new file named path. The callback can
 * store a baton for this new file in **file_baton; whatever value
 * it stores there should be passed through to apply_textdelta.
 */
svn_error_t *(*add_file)(const char *path,
                         void *parent_baton,
                         apr_pool_t *file_pool,
                         void **file_baton);

/** We are going to make change to a file named path, which resides
 * in the directory identified by parent_baton.
 *
 * The callback can store a baton for this new file in **file_baton;
 * whatever value it stores there should be passed through to
 * apply_textdelta.
 */
svn_error_t *(*open_file)(const char *path,
                          void *parent_baton,
                          apr_pool_t *file_pool,
                           void **file_baton);
```

```
  /** Apply a text delta, yielding the new revision of a file.
   *
   * file_baton indicates the file we're creating or updating, and the
   * ancestor file on which it is based; it is the baton set by some
   * prior add_file or open_file callback.
   *
   * The callback should set *handle to a text delta window
   * handler; we will then call *handle on successive text
   * delta windows as we receive them. The callback should set
   * * handler_baton to the value we should pass as the baton
   * argument to *handler.
   */
  svn_error_t *(*apply_textdelta)(void *file_baton,
                                  apr_pool_t *pool,
                                  svn_txdelta_window_handler_t *handler,
                                  void **handler_baton);

  /** We are done processing a file, whose baton is file_baton (set by
   * add_file or open_file). We won't be using the baton any
   * more, so whatever resources it refers to may now be freed.
   */
  svn_error_t *(*close_file)(void *file_baton,
                             apr_pool_t *pool)

  /** All delta processing is done. Call this, with the edit_baton for
   * the entire edit.
   */
  svn_error_t *(*close_edit)(void *edit_baton,
                             apr_pool_t *pool);]

  /** The editor-driver has decided to bail out. Allow the editor to
   * gracefully clean up things if it needs to.
   */
  svn_error_t *(*abort_edit)(void *edit_baton,
                             apr_pool_t *pool);
} svn_delta_editor_t;
```

But Is It Art?

I cannot claim that the beauty of this interface was immediately obvious to me. I'm not sure it was obvious to Jim either; he was probably just trying to get Ben and me out of his house. But he'd been pondering the problem for a long time, too, and he followed his instincts about how tree structures behave.

The first thing that strikes one about the delta editor is that it *chooses* constraint: even though there is no philosophical requirement that tree edits be done in depth-first order (or indeed in any order at all), the interface enforces depth-firstness anyway, by means of the baton relationships. This makes the interface's usage and behavior more predictable.

The second thing is that an entire edit operation unobtrusively carries its context with it, again by means of the batons. A file baton can contain a pointer to its parent directory baton, a directory baton can contain a pointer to *its* parent directory baton (with a null parent for the root of the edit), and everyone can contain a pointer to the global edit baton. Although an individual baton may be a disposable object — for example, when a file is closed, its baton is destroyed — any given baton allows access to the global edit context, which may contain, for example, the revision number the client side is being updated to. Thus, batons are overloaded: they provide scope (i.e., lifetime, because a baton only lasts as long as the pool in which it is allocated) to portions of the edit, but they also carry global context.

The third important feature is that the interface provides clear boundaries between the various suboperations involved in expressing a tree change. For example, opening a file merely indicates that something changed in that file between the two trees, but doesn't give details; calling `apply_textdelta` gives the details, but you don't have to call `apply_ textdelta` if you don't want to. Similarly, opening a directory indicates that something changed in or under that directory, but if you don't need to say any more than that, you can just close the directory and move on. These boundaries

are a consequence of the interface's dedication to *streaminess*, as expressed in its introductory comment: "*...instead of representing the tree delta explicitly, we define a standard way for a consumer to process each piece of a tree delta as soon as the producer creates it*." It would have been tempting to stream only the largest data chunks (that is, the file diffs), but the delta editor interface goes the whole way and streams the entire tree delta, thus giving both producer and consumer fine-grained control over memory usage, progress reporting, and interruptibility.

It was only after we began throwing the new delta editor at various problems that these features began to show their value. For example, one thing we wanted to implement was change summarization: a way to show an overview of the difference between two trees without giving the details. This is useful when someone wants to know which files in her working copy have changed in the repository since she checked them out, but doesn't need to know exactly what the changes were.

Here's a slightly simplified version of how it works: the client tells the server what revision tree the working copy is based on, and then the server tells the client the difference between that revision tree and the latest one, using the delta editor. The server is the producer, the client is the consumer.

Using the repository from earlier in the chapter, in which we built up a change to */A/fish/tuna* to create revision 2, let's see how this would look as a series of editor calls, sent by the server to a client whose tree is still at revision 1. The if block about two-thirds of the way through is where we decide whether this is a summarization edit or a "give me everything" edit:

```
svn_delta_editor_t *editor
void *edit_baton;

/* In real life, this would be a passed-in parameter, of course. */
int summarize_only = TRUE;

/* In real life, these variables would be declared in subroutines,
   so that their lifetimes would be bound to the stack frame just
   as the objects they point to are bound by the tree edit frame. */
void *root_baton;
void *dir_baton;
void *subdir_baton;
void *file_baton;

/* Similarly, there would be subpools, not just one top-level pool. */
apr_pool_t *pool = svn_pool_create();

/* Each use of the delta editor interface starts by requesting the
   particular editor that implements whatever you need, e.g.,
   streaming the edit across the network, applying it to a working
   copy, etc. */
Get_Update_Editor(&editor, &eb,
                  some_repository,
                  1, /* source revision number */
                  2, /* target revision number */
                  pool);

/* Now we drive the editor. In real life, this sequence of calls
   would be dynamically generated, by code that walks the two
   repository trees and invokes editor->foo() as appropriate. */

editor->open_root(edit_baton, pool, &root_baton);
editor->open_directory("A", root_baton, pool, &dir_baton);
editor->open_directory("A/fish", dir_baton,pool, &subdir_baton);
editor->open_file("A/fish/tuna", subdir_baton, pool, &file_baton);

if (! summarize_only)
  {
    svn_txdelta_window_handler_t window_handler;
    void *window_handler_baton;
    svn_txdelta_window_t *window;

    editor->apply_textdelta(file_baton, pool
                                apr_pool_t *pool,
```

```
                                    &window_handler,
                                    &window_handler_baton);
  do {
    window = Get_Next_TextDelta_Window(...);
    window_handler(window, window_handler_baton);
  } while (window);
}

editor->close_file(file_baton, pool);
editor->close_directory(subdir_baton, pool);
editor->close_directory(dir_baton, pool);
editor->close_directory(root_baton, pool);
editor->close_edit(edit_baton, pool);
```

As this example shows, the distinction between a summary of a change and the full version of the change falls naturally along the boundaries of the delta editor interface, allowing us to use the same code path for both purposes. While it happens that the two revision trees in this example were adjacent (revision 1 and revision 2), they didn't have to be. The same method would work for any two trees, even with many revisions between them, as is the case when a working copy hasn't been updated for a long time. And it would work when the two trees are in reverse order — that is, with the newer revision first. This is useful for reverting a change.

Abstraction As a Spectator Sport

Our next indication of the delta editor's flexibility came when we needed to do two or more distinct things in the same tree edit. One of the earliest such situations was the need to handle cancellations. When the user interrupted an update, a signal handler trapped the request and set a flag; then at various points during the operation, we checked the flag and exited cleanly if it was set. It turned out that in most cases, the safest place to exit the operation was simply the next entry or exit boundary of an editor function call. This was trivially true for operations that performed no I/O on the client side (such as change summarizations and diffs), but it was also true of many operations that did touch files. After all, most of the work in an update is simply writing out the data, and even if the user interrupts the overall update, it usually still makes sense to either finish writing or cleanly cancel whatever file was in progress when the interrupt was detected.

But where to implement the flag checks? We could hardcode them into the delta editor, the one returned (by reference) from `Get_Update_Editor()`. But that's obviously a poor choice: the delta editor is a library function that might be called from code that wants a totally different style of cancellation checking, or none at all.

A slightly better solution would be to pass a cancellation-checking callback function and associated baton to `Get_Update_Editor()`. The returned editor would periodically invoke the callback on the baton and, depending on the return value, either continue as normal or return early (if the callback is null, it is never invoked). But that arrangement isn't ideal either. Checking cancellation is really a parasitic goal: you might want to do it when updating, or you might not, but in any case it has no effect on the way the update process itself works. Ideally, the two shouldn't be tangled together in the code, especially as we had concluded that, for the most part, operations didn't need fine-grained control over cancellation checking, anyway — the editor call boundaries would do just fine.

Cancellation is just one example of an auxiliary task associated with tree delta edits. We faced, or thought we faced, similar problems in keeping track of committed targets while transmitting changes from the client to the server, in reporting update or commit progress to the user, and in various other situations. Naturally, we looked for a way to abstract out these adjunct behaviors, so the core code wouldn't be cluttered with them. In fact, we looked so hard that we initially *over*-abstracted:

```
/** Compose editor_1 and its baton with editor_2 and its baton.
 *
 * Return a new editor in new_editor (allocated in pool), in which
 * each function fun calls editor_1->fun and then editor_2->fun,
 * with the corresponding batons.
 *
 * If editor_1->fun returns error, that error is returned from
```

```
 * new_editor->fun and editor_2->fun is never called; otherwise
 * new_editor->fun's return value is the same as editor_2->fun's.
 *
 * If an editor function is null, it is simply never called, and this
 * is not an error.
 */
void
svn_delta_compose_editors(const svn_delta_editor_t **new_editor,
                          void **new_edit_baton,
                          const svn_delta_editor_t *editor_1,
                          void *edit_baton_1,
                          const svn_delta_editor_t *editor_2,
                          void *edit_baton_2,
                          apr_pool_t *pool);
```

Although this turned out to go a bit too far — we'll look at why in a moment — I still find it a testament to the beauty of the editor interface. The composed editors behaved predictably, they kept the code clean (because no individual editor function had to worry about the details of some parallel editor invoked before or after it), and they passed the associativity test: you could take an editor that was itself the result of a composition and compose it with other editors, and everything would *just work*. It worked because the editors all agreed on the basic shape of the operation they were performing, even though they might do totally different things with the data.

As you can tell, I still miss editor composition for its sheer elegance. But in the end it was more abstraction than we needed. Much of the functionality that we initially implemented using composed editors, we later rewrote to use custom callbacks passed to the editor-creation routines. Although the adjunct behaviors did usually line up with editor call boundaries, they often weren't appropriate at *all* call boundaries, or even at most of them. The result was an overly high infrastructure-to-work ratio: by setting up an entire parallel editor, we were misleadingly implying to readers of the code that the adjunct behaviors would be invoked more often than they actually were.

Having gone as far as we could with editor composition and then retreated, we were still free to implement it by hand when we really wanted it, however. Today in Subversion, cancellation is done with manual composition. The cancellation-checking editor's constructor takes another editor — the core operation editor — as a parameter:

```
/** Set *editor and *edit_baton to a cancellation editor that
 * wraps wrapped_editor and wrapped_baton.
 *
 * The editor will call cancel_func with cancel_baton when each of
 * its functions is called, continuing on to call the corresponding wrapped
 * function if cancel_func returns SVN_NO_ERROR.
 *
 * If cancel_func is NULL, set *editor to wrapped_editor and
 * *edit_baton to wrapped_baton.
 */
svn_error_t *
svn_delta_get_cancellation_editor(svn_cancel_func_t cancel_func,
                                  void *cancel_baton,
                                  const svn_delta_editor_t *wrapped_editor,
                                  void *wrapped_baton,
                                  const svn_delta_editor_t **editor,
                                  void **edit_baton,
                                  apr_pool_t *pool);
```

We also implement some conditional debugging traces using a similar process of manual composition. The other adjunct behaviors — primarily progress reporting, event notification, and target counting — are implemented via callbacks that are passed to the editor constructors and (if nonnull) invoked by the editor at the few places where they are needed.

The editor interface continues to provide a strong unifying force across Subversion's code. It may seem strange to praise an API that first tempted its users into over-abstraction, but that temptation was mainly a side effect of suiting the problem of streamy tree delta transmission exceptionally well — it made the problem look so tractable that we wanted other problems to become that problem! When they didn't fit,

we backed off, but the editor constructors still provided a canonical place to inject callbacks, and the editor's internal operation boundaries helped guide our thinking about when to invoke those callbacks.

Conclusions

The real strength of this API, and, I suspect, of any good API, is that it guides one's thinking. All operations involving tree modification in Subversion are now shaped roughly the same way. This not only reduces the amount of time newcomers must spend learning existing code, it gives new code a clear model to follow, and developers have taken the hint. For example, the *svnsync* feature, which mirrors one repository's activity directly into another repository — and was added to Subversion in 2006, six years after the advent of the delta editor — uses the delta editor interface to transmit the activity. The developer of that feature was not only spared the need to design a change-transmission mechanism, he was spared the need to even *consider* whether he needed to design a change-transmission mechanism. And others who now hack on the new code find it feels mostly familiar the first time they see it.

These are significant benefits. Not only does having the right API reduce learning time, it also relieves the development community of the need to have certain debates: design discussions that would have spawned long and contentious mailing list threads simply do not come up. That may not be quite the same thing as pure technical or aesthetic beauty, but in a project with many participants and a constant turnover rate, it's a beauty you can use.