Practical Subversion with 4D

By Josh Fletcher, Technical Account Manager, 4D Inc.

Technical Note 12-08

Table of Contents

Abstract

Revision control is a fundamental part of software development and yet, because of 4D's propensity to use binary files, it remains an unsolved problem for many 4D developers.

Subversion (SVN) is a well-established, easy-to-use revision control solution. It is particularly attractive for 4D use with the addition of the Source Toolkit feature in 4D v13.

This Technical Note briefly covers revision control in general, SVN, and practical tips for using SVN with 4D. The goal is to make revision control an attractive target for **any** 4D developer (not just large teams, even solo developers can reap the rewards). A component is included (CodeExport.4dbase) that can automatically export all methods from a 4D database for the purpose of placing those methods under revision control.

Introduction

4D has never made revision control easy. The fully-integrated nature of 4D's platform, while extremely efficient for application development, is also somewhat closed and proprietary. Revision control software tends to cast a wider net to the point the most implementations are built around the assumption that the files under revision control are plain text files or maybe images, but not large complex binary files. To be clear, revision control software generally supports **any** file type for archival purposes but 9 times out of 10 the most important features only work on text files.

4D v13 changes the game with the Source Toolkit feature. From teams to solo developers, any 4D developer can reap the benefits of adding revision control into their environment.

Please note: this Technical Note does not intend to teach SVN, in general. Basic tasks like installing SVN software, creating repositories, checking out, etc. are not covered here simply because there is a wealth of getting started material already available on the Internet. Instead this Technical Note offers practical information about using 4D with SVN. It is not expected that the reader necessarily be familiar with SVN already, but it is expected that the reader will be willing to install and configure SVN independent of this Technical Note.

Revision Control

What is revision control? There are many ways to answer this question of course, but at the heart the answer is in the name:

- Revision – software naturally moves through time with various finite states. As changes are made, new revisions are introduced. Take 4D as an example: 4D v12 or 4D v13 are simply revisions of a master set of source code.
- Control – the point of revision control is to literally take control of the evolution of the software.

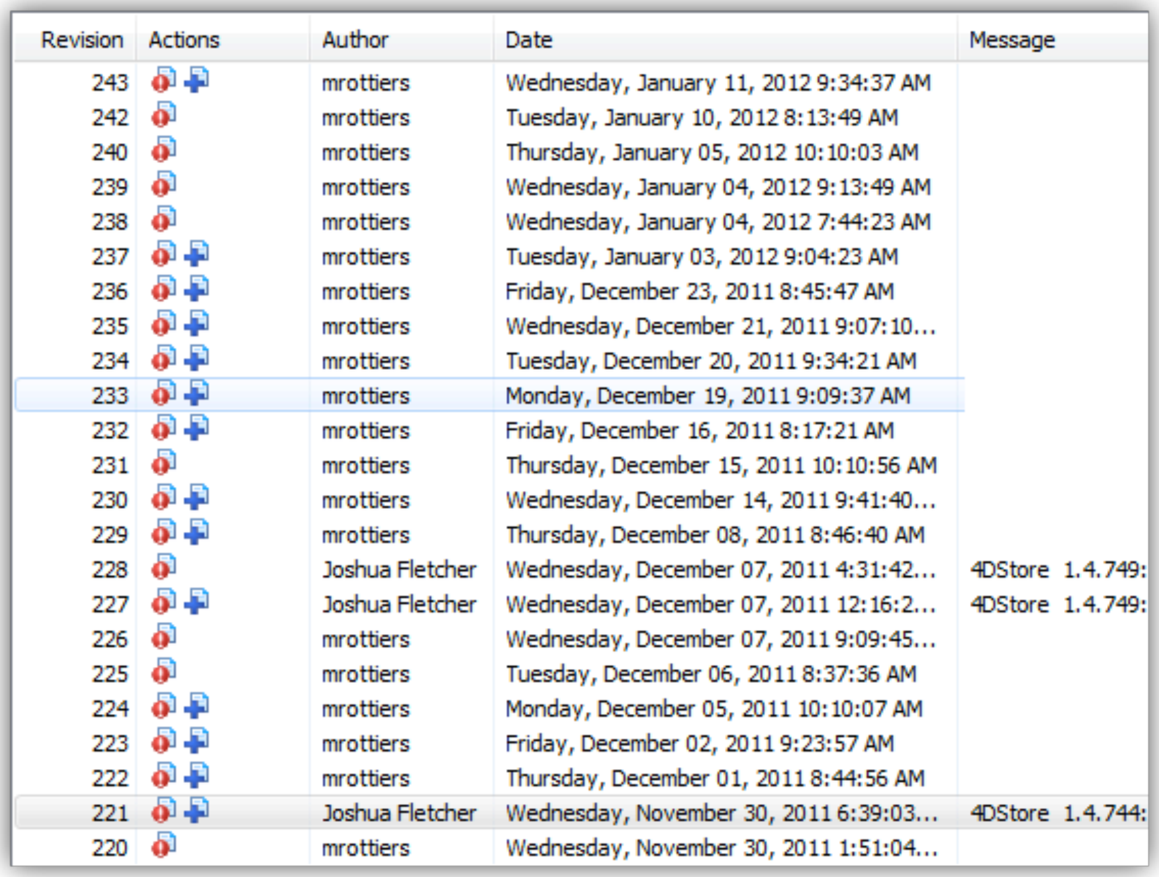
Revision control software attempts to add **structure** and **automation** to something that is already happening. If you open 4D database and change a method, you have created a new revision of the database. Can the change be easily reverted? How can the developer keep track of which version is which? What if more than one person changes the same method? Which version is currently deployed to the customer? These are the kinds of questions that, while answerable without revision control software, become trivial to manage with the addition of such.

Structure generally comes in the form of **revision numbers**. SVN assigns a unique, automatically incremented number to each revision of the project, for example. Thus determining which version of the software is being developed, or which versions have been deployed is just a matter of identifying the revision number.

To some extent, revision control allows a measure of “time travel”. Here is an example.

Time Travel

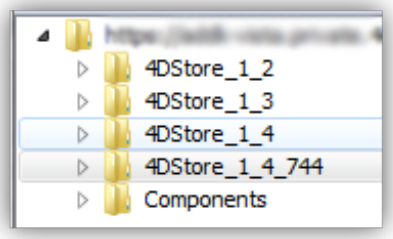
Several internal 4D systems are built from a common codebase called “4DStore”. This is a segment of the SVN revision log for this project:



Revision	Actions	Author	Date	Message
243		mrottiers	Wednesday, January 11, 2012 9:34:37 AM	
242		mrottiers	Tuesday, January 10, 2012 8:13:49 AM	
240		mrottiers	Thursday, January 05, 2012 10:10:03 AM	
239		mrottiers	Wednesday, January 04, 2012 9:13:49 AM	
238		mrottiers	Wednesday, January 04, 2012 7:44:23 AM	
237		mrottiers	Tuesday, January 03, 2012 9:04:23 AM	
236		mrottiers	Friday, December 23, 2011 8:45:47 AM	
235		mrottiers	Wednesday, December 21, 2011 9:07:10...	
234		mrottiers	Tuesday, December 20, 2011 9:34:21 AM	
233		mrottiers	Monday, December 19, 2011 9:09:37 AM	
232		mrottiers	Friday, December 16, 2011 8:17:21 AM	
231		mrottiers	Thursday, December 15, 2011 10:10:56 AM	
230		mrottiers	Wednesday, December 14, 2011 9:41:40...	
229		mrottiers	Thursday, December 08, 2011 8:46:40 AM	
228		Joshua Fletcher	Wednesday, December 07, 2011 4:31:42...	4DStore 1.4.749:
227		Joshua Fletcher	Wednesday, December 07, 2011 12:16:2...	4DStore 1.4.749:
226		mrottiers	Wednesday, December 07, 2011 9:09:45...	
225		mrottiers	Tuesday, December 06, 2011 8:37:36 AM	
224		mrottiers	Monday, December 05, 2011 10:10:07 AM	
223		mrottiers	Friday, December 02, 2011 9:23:57 AM	
222		mrottiers	Thursday, December 01, 2011 8:44:56 AM	
221		Joshua Fletcher	Wednesday, November 30, 2011 6:39:03...	4DStore 1.4.744:
220		mrottiers	Wednesday, November 30, 2011 1:51:04...	

One of these systems currently deployed in the US office is based on revision 221 (based on 4DStore 1.4.744). Of course many changes have been made to the project since then (new features, refactoring, bug fixes, etc.)

A critical bug was identified in revision 221. It was critical enough that it warranted a fix and redeployment of that revision and could not wait for the full testing cycle that would be necessary to deploy the current revision of the source code. Luckily with revision control software this is trivial to fix. SVN was used to create a "branch" of the project at revision 221. Here is what the project looked like after the branch:



```
4DStore_1_2
4DStore_1_3
4DStore_1_4
4DStore_1_4_744
Components
```

The branch is just a copy of the original project, but taken at a specific point in time. The bug fix was made, tested, and deployed against revision 221 instead of the current version of the project (of course the bug fix was made in the current version as well).

This type of task is certainly achievable without revision control, but is far more difficult to manage as the developer would need to be very diligent about keeping track of different copies of the application and ensuring the change was made to the correct version(s).

Revision Control Concepts and Terminology

There are 10's if not 100's of different revision control solutions available today. While the implementations vary, they often share the same concepts. Thus it is useful to have an understanding of these concepts and the terms used to refer to them. This information can be applied to any revision control system.

Please note this list is not exhaustive. Revision control is an extensive area of software development. In larger organizations it is even possible to find one or more engineers devoted only to maintaining the revision control system (these are sometimes called "integration engineers"). Instead the list presented covers the basics of revision control and additionally focuses on the concepts that are important for 4D.

Secondly the terms used to describe these revision control concepts are SVN-specific. Other revision control solutions may use different terms, but the concepts are the same. For the purposes of this Technical Note it is better to stick to SVN-specific terms to avoid confusion.

Repository

If revision control software is thought of in a client-server context, the repository is the server side of the connection. The repository is the master copy of the project. The most important thing to understand about the repository is that the developer never manipulates the repository assets directly. The files in the repository are protected from direct access. This is how the "control" in revision control is achieved. In this sense the repository is an archive.

Naturally the repository also contains the full revision history of the project. SVN achieves this via differential copies (only the differences are tracked, not full copies of each revision) so the repository storage is often quiet efficient.

Note that the client-server model is not the only revision control model available. There are distributed revision control solutions where there is no single master repository. Nonetheless the repository is still protected from direct access and thus the "control" is maintained.

Working Copy

The working copy is the client portion of the project, using the preceding client-server analogy. This is the copy of the project that the developer directly manipulates. Changes made to the working copy are **not** reflected in the repository without specific action. When changes are ready for inclusion in the repository, they are pushed back to the server (thus creating a new revision).

Check Out

If the developer cannot directly modify the repository and must therefore make a copy (working copy), a check out is how this occurs. In other words the action of checking out creates a working copy.

Also implied here is that the developer must have permission to check out the project, so an additional measure of control is added via permissions in any revision control solution.

Update

Revision control often implies multi-user development (though this is **not** a requirement for the use of revision control). If more than one user is capable of updating the repository, the update action is what allows the other users to get the latest changes.

Note that, like the differential storage of the repository, an update only processes any assets that have been changed, not the entire project. In this way the update process is also very efficient.

Commit

When changes made to the working copy need to be reflected in the repository, it is time to commit those changes. The commit action creates a new revision of the project in the repository. Additionally the working copy is considered updated; that is to say there is no need to do an update immediately after a commit. See "revert" for a better understanding of why this is important.

Revert

The revert action discards any changes made to the working copy, returning it to the last known revision from the repository. When a check out or update is completed, the working copy is considered to be in a known good state; i.e. the working copy reflects exactly some revision in the repository. As soon as a change is made, the working copy no longer reflects any known revision (until those changes are committed).

The power of this is that the developer is free to experiment on the working copy and, if things go horribly wrong, they can simply revert back to the last known good state.

This is **especially** useful in a 4D context because once a change is made there is no way to revert it in many cases. Even with undo, the undo history is lost as soon as the editor (method or form) window is closed.

Note: *Revert is one of the most important actions for a 4D developer to understand. More on this later but pay special attention to this action.*

Conflict Resolution

Since no developer can work on the repository directly and must therefore work on their own working copy, it is possible for more than one user to modify the same asset at the same time. Remember that changes in the working copy are not reflected in the repository unless they are committed. If two users have the same revision checked out and change the same asset, when they commit that asset will be in conflict.

Typically in revision control software the first user to commit will not run into any issue, it is only when the second user commits that the conflict occurs. Note that this is not always the case, but is true for SVN.

Conflict resolution is simply the task of deciding which change will persist in the repository. The value of revision control is that it can help automate conflict resolution. If the conflicting asset is a text file, for example, the software may present the user with a copy of each file and highlight the differences. The user can then decide which changes to keep and which to reject.

Conflict resolution in 4D databases is, unfortunately, more complex. Since 4D databases are generally made of binary files the revision control software will not have any meaningful understanding of the differences. Nonetheless the software will at least be able to detect that a conflict has occurred and warn the user. More in this later.

Diff(erence)

Diff'ing refers to the ability of revision control software to show the user the difference between different revisions of an asset in the repository. Most of the time (in most other programming languages) the source code is housed in plain text files so any revision control solution is quite good at diff'ing text files. Some go a step further: SVN for example supports diff’ing Word documents using Word’s built in diff’ing feature. SVN will obtain a copy of each revision of the file and open it in Word.

As mentioned previously, there are no revision control solutions that can diff 4D’s binary files. This is one issue this Technical Note attempts to address by exporting the 4D code to text files as described later.

Tagging

Tagging refers to the act of marking a specific revision as a “milestone”. I.e. if a specific revision is released to customers, it can be useful to tag that revision so that it is easier to locate later.

Tagged revisions are not necessary, but they make it easier to identify milestone revisions.

Branching

An example of branching was already presented in the “Time Travel” section.

Branching is the act of creating an additional, independent copy of the project in the repository. The previous example created a branch from a tagged revision of the project for the purposes of making a single bug fix.

4D’s software line-up is another excellent example of branching. All current 4D products (including Wakanda) are branches off a common set of source code in the repository. These branches are maintained independently for the purpose of managing each unique product (v12, v13, Wakanda).

The advantage of branching, in addition to the already mentioned benefits of revision control, is the ability to merge important changes (see next section).

Merging

Merging refers to the ability of revision control software to make a single change to multiple copies of the project. For example if a bug is fixed in the main branch of 4D’s source code, and that code is shared among different branches, then the fix can be merged into each unique branch. This is **extremely** efficient when maintaining multiple copies/versions of software.

Subversion (SVN)

Subversion is an open source version control system. Founded in 2000 by CollabNet, Inc., the Subversion project and software have seen incredible success over the past decade. Subversion has enjoyed and continues to enjoy widespread adoption in both the open source arena and the corporate world.

(From http://subversion.apache.org/)

Subversion is a true open source project in the sense that visiting Apache's SVN website provides access to download source code, not binary packages. Various third-parties have implemented solutions based on this source code for both SVN servers and clients. Said another way, Apache does not provide the server or the client for SVN, only the source code. Some of the client and server solutions are discusses in the following sections.

Why SVN?

Here are some of the reasons SVN works well for 4D.

Well-Established

A 2007 report by Forrester Research recognized Subversion as the sole leader in the Standalone Software Configuration Management (SCM) category.

SVN was first released in 2000 and to this day it is still under active development. It supports 4D's target platforms, Mac OS X and Windows (as well as *nix platforms).

What this means for 4D developers is that it is easy to get started using SVN and there is a wealth of resources available when help is needed.

Easy

Above all one of the goals of this Technical Note is for **every** 4D developer to seriously consider adding revision control software into their development environment. Thus an overriding theme is that this be easy to accomplish.

Being a well-established solution with a long history means, for one thing, that SVN benefits from client and server packages that are easy to install and easy to use. Make no mistake, revision control is a complex topic, but getting started with SVN is easy so it is a good place to start.

Additionally, being and open source project, many of the best SVN distributions are free (or at least offer a free version) even for commercial use. The cost of entry here is quite low.

Optimistic

Keeping with the theme of being easy to use, one advantage that SVN offers is that its file management mechanisms are completely optimistic. A contrasting example helps explain this:

Another revision control solution, Perforce, uses pessimistic file locking for files that are under revision control. Checking files out to the working copy does not mean they can be edited because all files in the working copy are read-only. An additional Perforce command must be issued in order to make a file read/write. Think about what this means for a moment:

- If the 4D structure file is under Perforce's control, the developer must issue a command to make it read/write.
- If the methods in the 4D database have been exported to text files (as with the Source Toolkit feature), the same Perforce command must be issued from within 4D in order to export a new copy of the file.

Perforce's pessimistic file locking means that the revision control software must be **integrated** into 4D. This is by no means "easy".

SVN by contrast is completely optimistic. Any file in the working copy can be modified at any time and SVN keeps track of what has been changed. This means that SVN requires **no integration** with the 4D database at all. The developer is free to continue developing as before and only need concern themselves with SVN when it comes time to update or commit.

TortoiseSVN

This is a slight digression because TortoiseSVN is a Windows-specific SVN client but it is nonetheless one of the reasons SVN is so attractive. Put simply, TortoiseSVN is perhaps the best revision control client there is (though equivalent clients exist for other revision control solutions like TortoiseHG for Mercurial). The power of TortoiseSVN is that it integrates directly with the Windows shell (that is, Windows Explorer). If you know how to use Windows Explorer, you already know how to use most of TortoiseSVN.

All features are driven through the context menu in Windows Explorer. Want to check out? Right click on a folder and choose "SVN Checkout…" Need to update? Right-click on the folder or file and choose "SVN Update…" For complex tasks like diff'ing, TortoiseSVN includes a built in diff viewer that is automatically launched when a diff is requested via the shell.

The overall point is that it is not necessary for a new user to learn a new, separate application to get started with SVN on Windows.

(Mac OS X solutions will be discussed later)

SVN Software

To get started using SVN, two pieces of software are needed:

- SVN Server
- SVN Client

This section talks about each piece and what tasks they are used for.

Note: *It is not necessary to run the client and server on two different machines, though this is convenient even if you are solo developer. Imagine being able to work on the project at the office, commit the changes, head home and simply update from the repository to get the latest changes. You never need to worry about whether or not you have the latest version nor which specific machine you're developing on.*

Server

In a 4D context the SVN server software is really only needed for two tasks:

- Repository management; that is creating, moving, and/or deleting repositories.
- Permissions; adding and removing user access to repositories.

The investment for the developer is quite small, especially if the SVN server integrates with a domain login system. Direct SVN server interaction will be minimal.

At 4D, Inc. we are using VisualSVN server for the following reasons:

- Easy setup via a Windows installer package.
- Authentication via the Windows domain, no need to manually create users.
- Free version for commercial use.

You are by no means required to use VisualSVN server, this is just one example of an easy-to-use SVN server package.

Client

The SVN client is used for everything else:

- Check out, update, commit
- Adding, renaming, and deleting files
- Branching, tagging, merging
- Diff'ing
- Auditing (viewing logs, revision graphs, etc.)
- Build automation

Note that last point: SVN clients come in both command-line and GUI variants, with the command-line clients being helpful for script-based build automation.

As already implied, the only client that really needs to be considered on Windows is TortoiseSVN and all examples throughout this Technical Note will use it.

There is, unfortunately, no equivalent client on Mac OS X. SVN clients that integrate with Finder do exist, but the feature sets are severely limited. Thus the best Mac OS X SVN clients are standalone GUI applications. In particular two clients should be considered:

- Cornerstone
- Versions

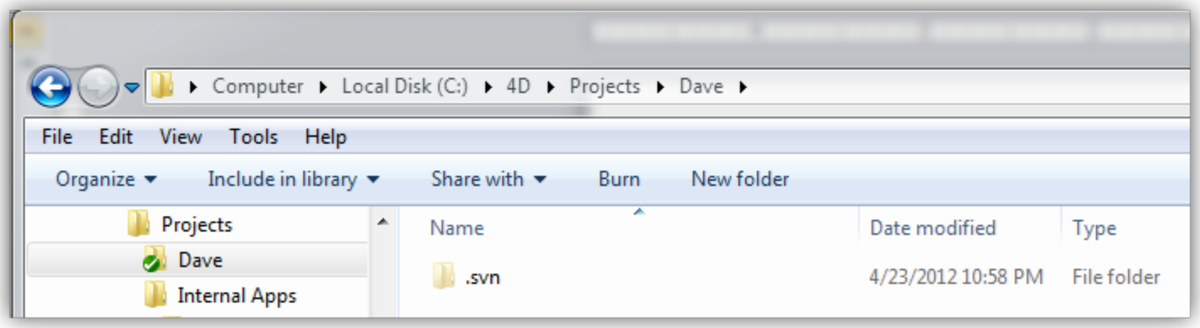
Note that both of these products are paid software. They do both offer trial modes to feel free to try them. No satisfactory free SVN clients exist for Mac unfortunately.

Getting Started

Briefly, here is how to get started with SVN:

- Install the server.
- Create a repository.
- Set permissions for the repository.
- Install the client.
- Check out the repository to create a working copy.

At this point the working copy is probably empty as shown here:


Computer ▸ Local Disk (C:) ▸ 4D ▸ Projects ▸ Dave ▸
File Edit View Tools Help
Organize ▾ Include in library ▾ Share with ▾ Burn New folder
Projects
Dave
Internal Apps
Name
Date modified
Type

.svn 4/23/2012 10:58 PM File folder

Pay attention to the ".svn" folder. This is where SVN stores the necessary metadata to keep track of the working copy, for example:

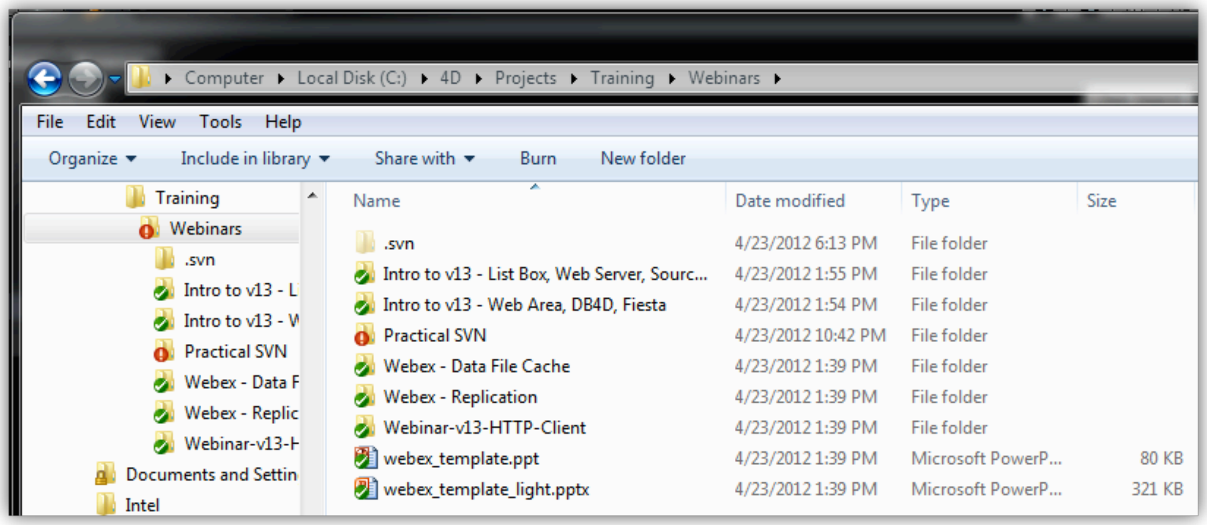
- Where is the repository? (the server address)

- Credentials to access the repository.
- Which files have changed?
- What is the correct revision to rollback to in case of a revert?

Want to remove the working copy, but keep the local files? Simply delete the .svn folder. The .svn folder is really the key that lets the SVN client know what is going on in the working copy and, without it, the working copy is just a normal set of files like any other.

Note also that the root folder of the working copy ("Dave" in the above example) need not have any specific name and can be placed anywhere on the machine.

Here is a more fleshed out example of a working copy with several projects. This is a working copy of all of the 4D Webinar materials:


Computer ▸ Local Disk (C:) ▸ 4D ▸ Projects ▸ Training ▸ Webinars ▸
File Edit View Tools Help
Organize ▾ Include in library ▾ Share with ▾ Burn New folder
Training
Webinars
.svn
Intro to v13 - L
Intro to v13 - W
Practical SVN
Webex - Data F
Webex - Replic
Webinar-v13-H
Documents and Settin
Intel

Name	Date modified	Type	Size
.svn	4/23/2012 6:13 PM	File folder	
Intro to v13 - List Box, Web Server, Sourc...	4/23/2012 1:55 PM	File folder	
Intro to v13 - Web Area, DB4D, Fiesta	4/23/2012 1:54 PM	File folder	
Practical SVN	4/23/2012 10:42 PM	File folder	
Webex - Data File Cache	4/23/2012 1:39 PM	File folder	
Webex - Replication	4/23/2012 1:39 PM	File folder	
Webinar-v13-HTTP-Client	4/23/2012 1:39 PM	File folder	
webex_template.ppt	4/23/2012 1:39 PM	Microsoft PowerP...	80 KB
webex_template_light.pptx	4/23/2012 1:39 PM	Microsoft PowerP...	321 KB

Note in this screenshot that TortoiseSVN applies shell icon overlays to the folders. The green check mark means no files have been changed in that folder (including subfolders) and the red exclamation means some files have been changed.

SVN 1.6 vs SVN 1.7

SVN clients and servers come in two major flavors, supporting either SVN 1.6 or SVN 1.7 (often both). It is important to note that both versions are being maintained in parallel so it is worth examining the major difference between the two.

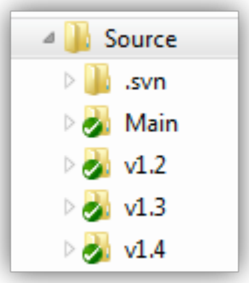
The major difference between 1.6 and 1.7 is the handling of the .svn folder:

- In 1.6 every folder in the working copy (that is, **every** subfolder) contains a .svn folder. Said another way, every folder in the working copy is itself another working copy.
- In 1.7 the working copy has only a single .svn folder at the root.

For SVN's developers, this change is largely about technical debt and performance. The 1.6 metadata implementation is fairly old and difficult to maintain. The 1.7 implementation is fresher and performs better.

For SVN users this change has important ramifications:

Under SVN 1.6 it is extremely easy to restructure the working copy. For example imagine a working copy contains several branches of a project, but the old branches are no longer being maintained:



In 1.6 each folder is itself a working copy. To remove the other branches, just delete those folders as well as the parent .svn folder. In the above example the "Main" branch could remain a fully functional working copy with no further steps but to delete the other folders, including the .svn folder at the same level.

Under 1.7 the above is not possible. To reduce the working copy to only one branch the developer would need to delete the current working copy and check out the desired branch. This can be **highly** undesirable for one simple reason: files not under revision control. If the working copy contains files that are not checked into the repository, for example 4DLINK files (which have machine-specific paths), the developer needs to manually preserve those files before deleting the working copy.

On the other hand 1.7 offers a nice advantage. To make a copy of the project, perhaps to send to someone else, simply copy all of the files except the .svn folder at the root. Under 1.6 the copy would contain all of the .svn folders from any subfolders. This is both messy and a potential risk because, as previously mentioned, the .svn folder contains repository location and credential information.

In the end the move from 1.6 to 1.7 is a trade-off for the developers of SVN as well as its users.

Note: *A clearly defined upgrade path from 1.6 to 1.7 exists so it is not necessary to re-check-out when switching versions, the 1.6 working copy can be upgraded to 1.7 (though not the other direction).*

SVN and 4D

With a working copy in place, it is time to look at using 4D with SVN. This section is divided into three "stages", looking at the simplest case, a more advanced usage, and looking to the future.

Stage 1 – Commit the Database

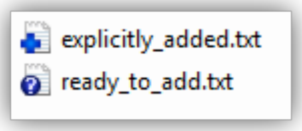
The simplest way to get started using 4D with SVN is to commit the database to the repository. After creating the working copy, copy (or move) the 4D database files and commit them to the repository.

Adding Files

Note there are two ways to add files with most SVN clients:

- There is a specific SVN "add" command. This explicitly adds the file to the next commit.
- By contrast creating a file in the working copy does not mean it will be committed to the repository. This can be convenient if, for the example, the file is something machine-specific and does not belong in the repository. In fact the SVN add command is not necessary. Generally new files can be added at commit time as well.

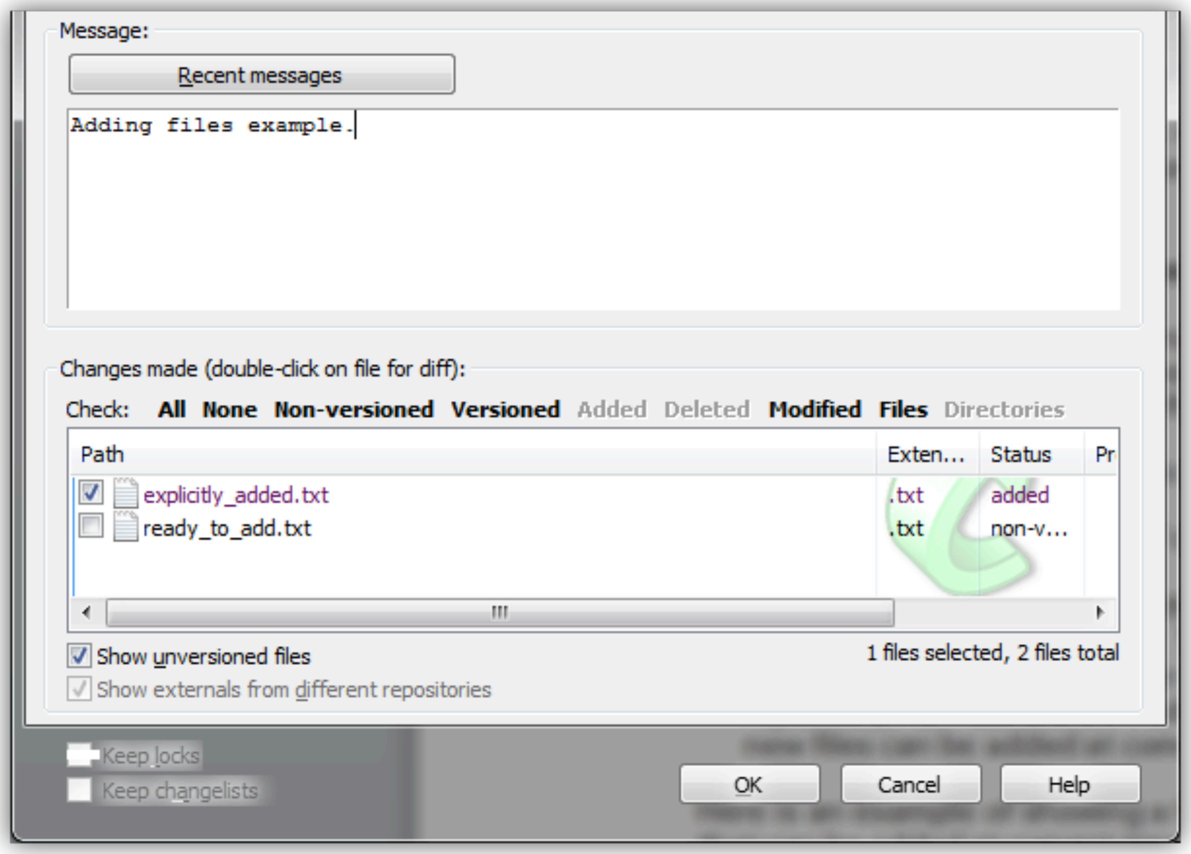
Here is an example of showing a file that has been explicitly added and one that can be added at commit time:



The TortoiseSVN icons indicate the file state:

- The file "explicitly_added.txt" has a + icon because an SVN add command was issued for that file.
- The file "ready_to_add.txt" has a ? icon because TortoiseSVN does not yet know about this file.

Here is the commit dialog for these files:



Adding files example.

Changes made (double-click on file for diff):
Check: All None Non-versioned Versioned Added Deleted Modified Files Directories

Path	Exten...	Status
✓ explicitly_added.txt	.txt	added
ready_to_add.txt	.txt	non-v...

✓ Show unversioned files
1 files selected, 2 files total
✓ Show externals from different repositories
Keep locks
Keep changelists
OK
Cancel
Help

Note the "Status" of "explicitly_added.txt" is "added" and "non-versioned" for the other file. Note also that there is a check-box next to each file. Checking the box next to "ready_to_add.txt" will also commit it to the repository.

Thus when committing a 4D database to an SVN repository it is not necessary to issue the add command for each individual file. Simply mark the files to be added at commit time.

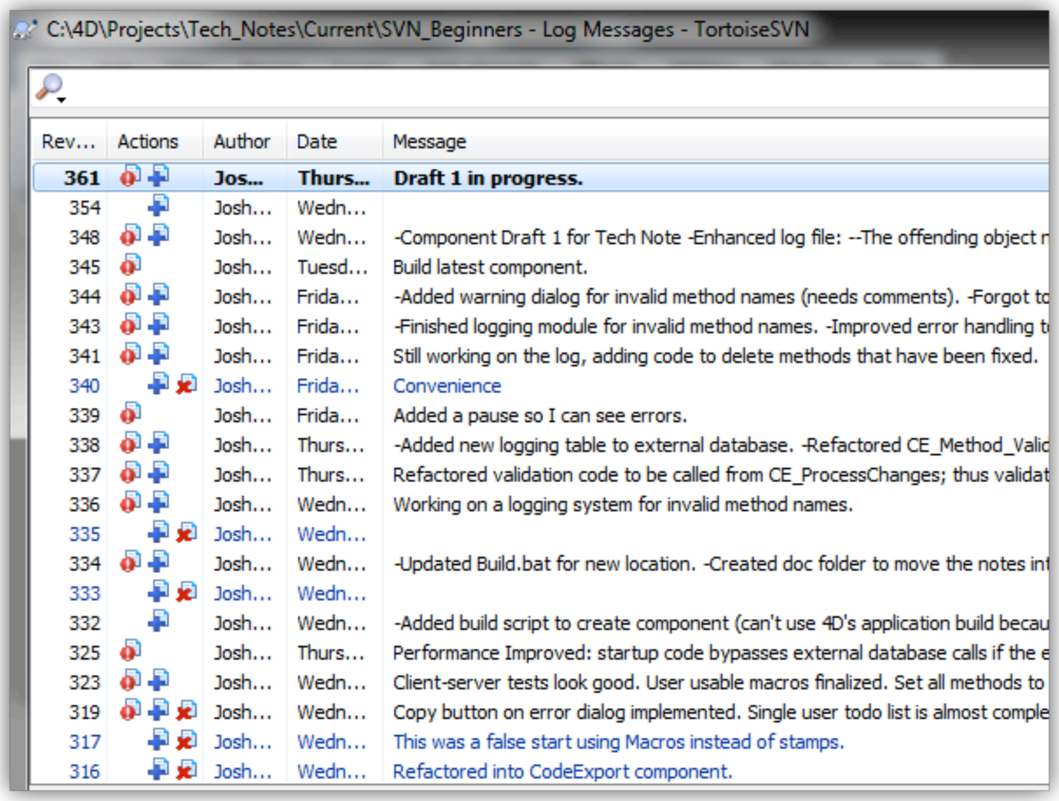
Benefits

Even with the simplest step of adding the database to SVN the developer can reap many rewards:

- **Differential Backup** – A tremendous advantage to using SVN is that it performs differential copies of the revisions, not full copies. It is an extremely efficient way to store multiple copies of the database.
- **Revert** – as explained previously, once changes are made to a 4D database there is no way to revert them. In some cases there is an undo feature, but the undo history is easily lost as well. A database

under SVN's control can easily be reverted to the last known revision (in fact to **any** revision) with the SVN revert command.

- **Branching and Tagging** – the full benefits of branching and tagging are realized at stage 1. A branching example was already given in the "Revision Control" section. This powerful feature makes it trivial to move back and forth through the database's life and make changes as needed.
- **Auditing (sort of)** – looking at the preceding commit dialog screenshot, note there is an area to add messages with each commit. In this way the developer is able to start building an audit trail for the changes that are made to the database. These messages can be viewed via the SVN commit log. Here is an example log:


C:\4D\Projects\Tech_Notes\Current\SVN_Beginners - Log Messages - TortoiseSVN

Rev...	Actions	Author	Date	Message
361		**Jos...**	**Thurs...**	**Draft 1 in progress.**
354		Josh...	Wedn...	
348		Josh...	Wedn...	-Component Draft 1 for Tech Note -Enhanced log file: --The offending object r
345		Josh...	Tuesd...	Build latest component.
344		Josh...	Frida...	-Added warning dialog for invalid method names (needs comments). -Forgot to
343		Josh...	Frida...	-Finished logging module for invalid method names. -Improved error handling t
341		Josh...	Frida...	Still working on the log, adding code to delete methods that have been fixed.
340		Josh...	Frida...	Convenience
339		Josh...	Frida...	Added a pause so I can see errors.
338		Josh...	Thurs...	-Added new logging table to external database. -Refactored CE_Method_Valid
337		Josh...	Thurs...	Refactored validation code to be called from CE_ProcessChanges; thus validat
336		Josh...	Wedn...	Working on a logging system for invalid method names.
335		Josh...	Wedn...	
334		Josh...	Wedn...	-Updated Build.bat for new location. -Created doc folder to move the notes int
333		Josh...	Wedn...	
332		Josh...	Wedn...	-Added build script to create component (can't use 4D's application build becau
325		Josh...	Thurs...	Performance Improved: startup code bypasses external database calls if the e
323		Josh...	Wedn...	Client-server tests look good. User usable macros finalized. Set all methods to
319		Josh...	Wedn...	Copy button on error dialog implemented. Single user todo list is almost comple
317		Josh...	Wedn...	This was a false start using Macros instead of stamps.
316		Josh...	Wedn...	Refactored into CodeExport component.

Risks

There are some downsides to only checking the database into SVN.

- There is no explicit way to see the difference between revisions of the database's binary files (structure, data, resources, etc.).
- While the commit log offers the potential for an audit trail, only diligent use of the log by the developer will make the feature useful.

    - Conflict resolution is a manual process. If one user overwrites the changes of another user, each revision of the database needs to be checked out and compared manually.

Stage 2 – Get the Code Out

To tap into the real potential of SVN and 4D, content inside the 4D database needs to be exported into a format that SVN can understand. The Source Toolkit feature, added in 4D v13, greatly assists this task. The Source Toolkit provides access to the code of the database, in particular read and write access to every method; project, form, database, object, and trigger methods can all be accessed and, therefore, exported.

There are two main commands to consider from the Source Toolkit:

- **METHOD GET PATHS** – this command provides an array of paths to methods in the database. Each method in a 4D database is identified by a unique path. Furthermore, these paths are compliant with the file system so they can be used to directly export the methods to disk.
    - Additionally this command accepts and returns a "stamp" value. Thus it is possible to get the paths to those methods that have changed since a particular stamp. This makes the Source Toolkit extremely efficient as there is no need to dump the entire database with each export.
- **METHOD GET CODE** – given a method path, this command returns a copy of the code in the method. Additionally the method attributes are also included in this copy; i.e. not only can the code be exported but the attributes as well.

Benefits

With only these two commands the Source Toolkit exponentially increases the value of SVN with 4D. By exporting the methods of the database to text files, and committing them to the repository, a wealth of SVN features is realized. Now changes to individual methods can be tracked using the built-in diff tools. If a method is renamed, the newly named method can automatically be exported. If a Global Find and Replace changes a whole series of methods, those can be exported as well. A final important benefit to using the Source Toolkit is this feature represents the future of revision control with 4D. The time to start using it is now.

The CodeExport component included with this Technical Note represents a full implementation of this "Stage 2" integration of SVN and 4D. Please refer to the "CodeExport Component" section for further details.

Risks

There are nonetheless some unsolved challenges even here at Stage 2.

- **Methods Only** – the Source Toolkit only supports the exporting of methods currently. Other structure objects (forms, lists, users, tables, etc.) cannot be exported.
    - For example there is still the risk that one user might overwrite the changes of another user to a form. There is no obvious or automatic way to resolve this except to get a copy of each revision, open them in 4D, and compare manually.
    - By contrast if one user overwrites another user's changes and the only thing that was changed was a method, there is no reason to compare the databases manually. Instead:
        - Get the latest revision of the database.
        - Get the **previous** revision of the exported method in question.
        - Paste the content of the exported method text file into the method in the database.
        - Finally commit the change.
- **Deletions** – if a method is deleted, there is no obvious indication via the Source Toolkit. Any exported text file version of that method will become "orphaned".
    - On the other hand, this may not be important. The fact is the database is still the master copy of the code. The presence of orphaned text versions of the methods, while perhaps a bit sloppy, has **no effect** on the database. Consider:
        - Investigating the difference between two methods means locating that method file and comparing the revisions. The presence of **other** method files is not important.
        - If a new method is created with the same name as a deleted method, the new version will simply overwrite the old one.
        - If the methods on disk must be cleaned up, simply delete all of the exported methods and re-export them from the database.
            - Note that the method files would need to be explicitly deleted via SVN (there is a "delete" command), not just via deleting the files, otherwise the next time an update occurs the deleted files would just be restored from the repository. In other words files deleted at the file system level are considered "missing", not deleted.
- **Rename** – the problem with renaming methods is the same as deletions. The newly named method will be exported of course, but the old version of the method on disk would remain.
- **Technical Debt** – 4D v13 is more stringent about method names than previous versions of 4D. In particular method names need to contain characters that are legal in the file system in order to be exported. As such the possibility exists that converted databases contain methods with invalid names. These methods cannot be exported to disk until they are fixed.

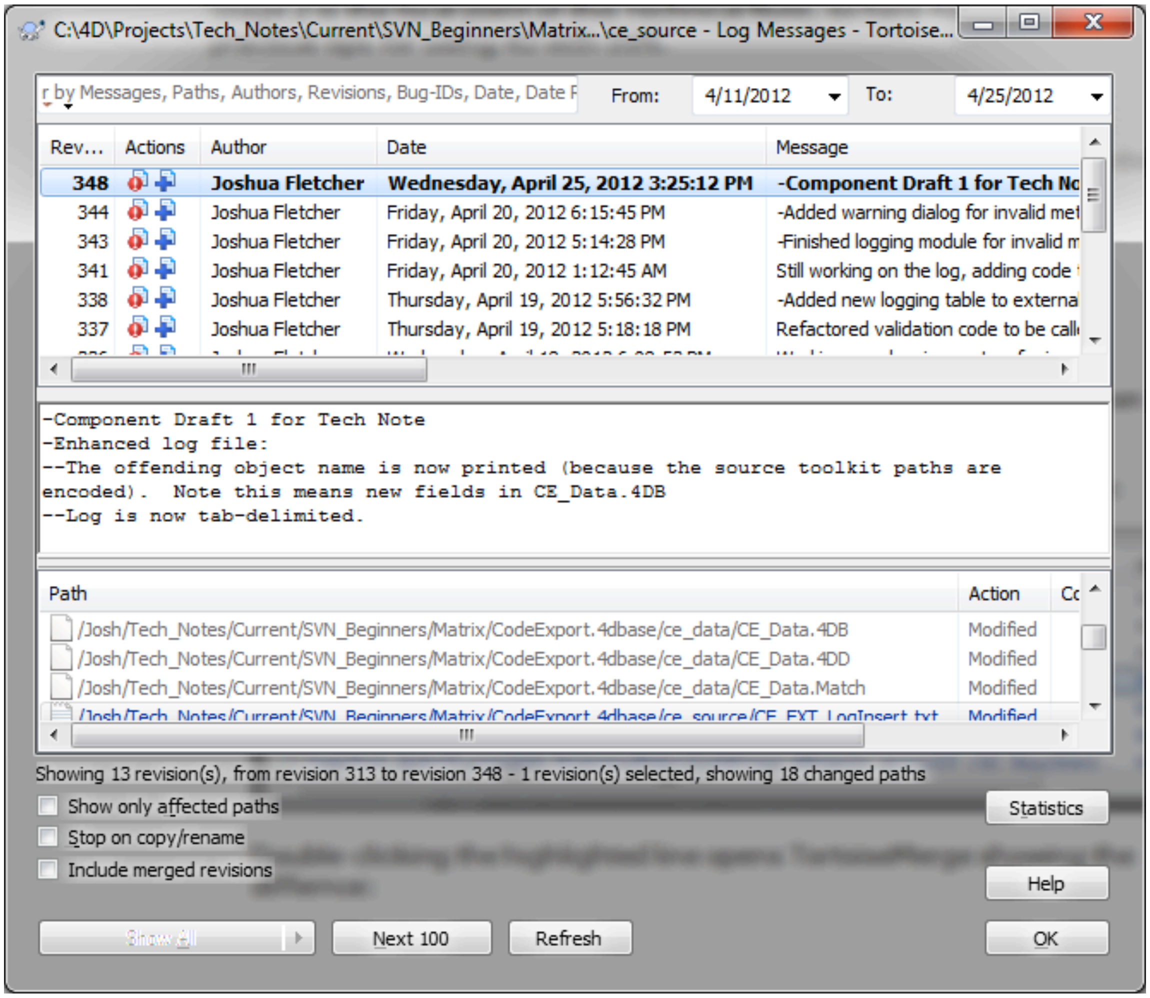
- **Conflict Resolution** – conflict resolution remains a largely manual process. SVN tools exist to merge the changes in two different text files, for example, but remember that in all cases the database is still the master copy of the source code so even if the conflict in the exported method text files is resolved, the code would need to be pasted back into 4D.

Tips

Stage 2 is the focal point of this Technical Note, so here follows some practical tips for using 4D with SVN.

Log Data

The SVN revision log contains a wealth of information. Already mentioned was the message feature, but here are some more (as presented in TortoiseSVN):


C:\4D\Projects\Tech_Notes\Current\SVN_Beginners\Matrix...\ce_source - Log Messages - Tortoise...
r by Messages, Paths, Authors, Revisions, Bug-IDs, Date, Date F
From:
4/11/2012
To:
4/25/2012

Rev...	Actions	Author	Date	Message
348		Joshua Fletcher	Wednesday, April 25, 2012 3:25:12 PM	-Component Draft 1 for Tech No
344		Joshua Fletcher	Friday, April 20, 2012 6:15:45 PM	-Added warning dialog for invalid met
343		Joshua Fletcher	Friday, April 20, 2012 5:14:28 PM	-Finished logging module for invalid m
341		Joshua Fletcher	Friday, April 20, 2012 1:12:45 AM	Still working on the log, adding code
338		Joshua Fletcher	Thursday, April 19, 2012 5:56:32 PM	-Added new logging table to externa
337		Joshua Fletcher	Thursday, April 19, 2012 5:18:18 PM	Refactored validation code to be calli

```
-Component Draft 1 for Tech Note
-Enhanced log file:
--The offending object name is now printed (because the source toolkit paths are
encoded). Note this means new fields in CE_Data.4DB
--Log is now tab-delimited.
```

Path	Action	Cc
/Josh/Tech_Notes/Current/SVN_Beginners/Matrix/CodeExport.4dbase/ce_data/CE_Data.4DB	Modified	
/Josh/Tech_Notes/Current/SVN_Beginners/Matrix/CodeExport.4dbase/ce_data/CE_Data.4DD	Modified	
/Josh/Tech_Notes/Current/SVN_Beginners/Matrix/CodeExport.4dbase/ce_data/CE_Data.Match	Modified	
/Josh/Tech_Notes/Current/SVN_Beginners/Matrix/CodeExport.4dbase/ce_source/CE_EXT_LogInsert.txt	Modified	

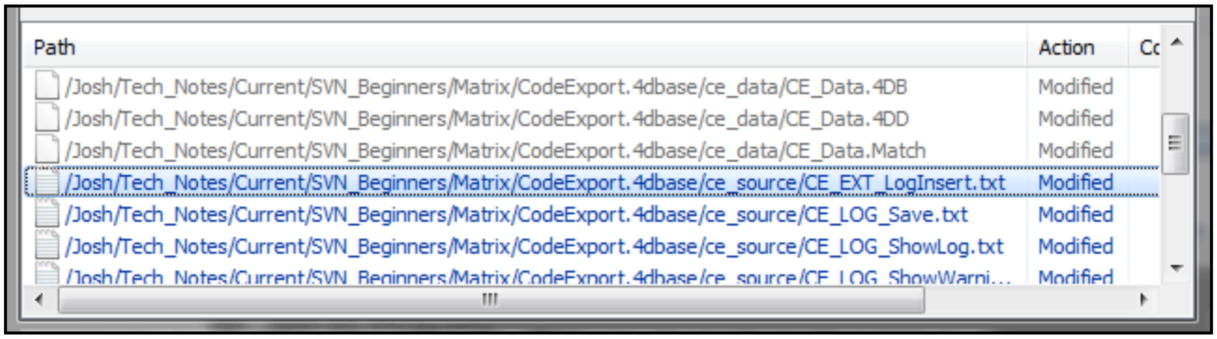
Showing 13 revision(s), from revision 313 to revision 348 - 1 revision(s) selected, showing 18 changed paths

- The automatically assigned revision numbers are in the left column.
- Additionally the actions for each commit are summarrized via icons.
- The user who commited the changes and the date and time are available.
- The aformentioned messages are shown in the last column and the message for the currently highlighted revision is displayed underneath the list as well.
- Finally the list below the message contains information about each file that was modified for each and every revision (including new files, deleted files, and changed files).

Diff

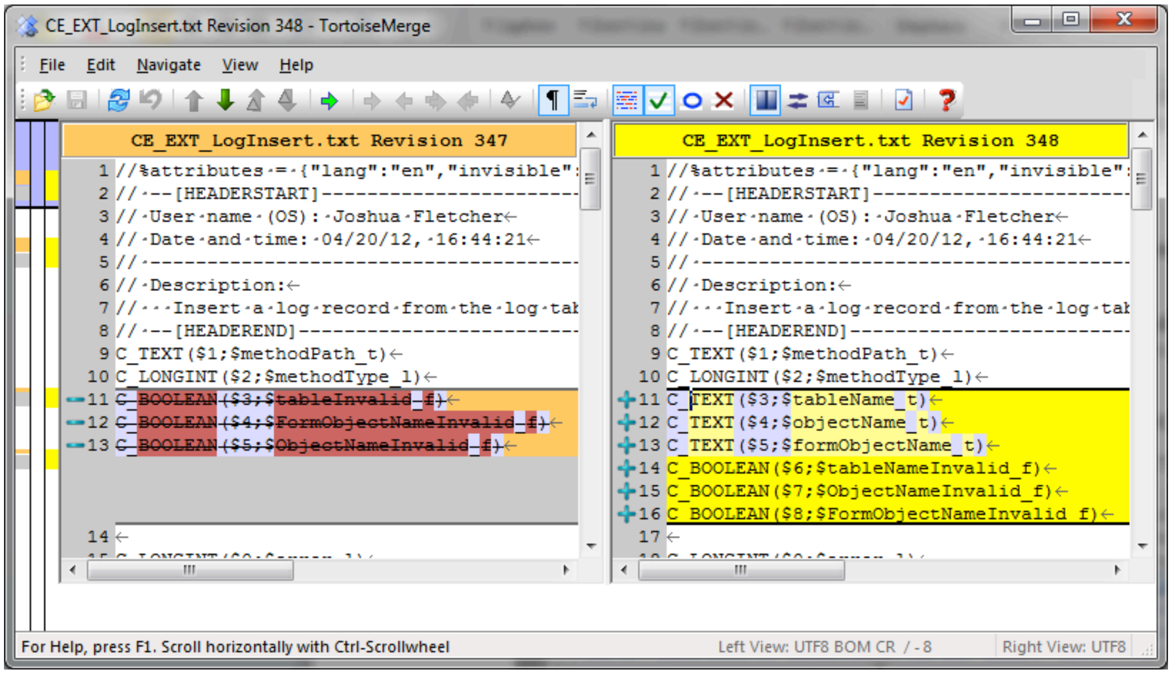
Use the SVN client's diff tools to track the changes to methods. In TortoiseSVN this is as simple as double-clicking the method in question when viewing the revision log, for example:

The method "CE_EXT_LogInsert" was modified in the commit below:



Path	Action	Cc
/Josh/Tech_Notes/Current/SVN_Beginners/Matrix/CodeExport.4dbase/ce_data/CE_Data.4DB	Modified	
/Josh/Tech_Notes/Current/SVN_Beginners/Matrix/CodeExport.4dbase/ce_data/CE_Data.4DD	Modified	
/Josh/Tech_Notes/Current/SVN_Beginners/Matrix/CodeExport.4dbase/ce_data/CE_Data.Match	Modified	
/Josh/Tech_Notes/Current/SVN_Beginners/Matrix/CodeExport.4dbase/ce_source/CE_EXT_LogInsert.txt	Modified	
/Josh/Tech_Notes/Current/SVN_Beginners/Matrix/CodeExport.4dbase/ce_source/CE_LOG_Save.txt	Modified	
/Josh/Tech_Notes/Current/SVN_Beginners/Matrix/CodeExport.4dbase/ce_source/CE_LOG_ShowLog.txt	Modified	
/Josh/Tech_Notes/Current/SVN_Beginners/Matrix/CodeExport.4dbase/ce_source/CE_LOG_ShowWarni...	Modified	

Double-clicking the highlighted line opens TortoiseMerge showing the difference:


CE_EXT_LogInsert.txt Revision 348 - TortoiseMerge
File Edit Navigate View Help

CE_EXT_LogInsert.txt Revision 347

```
//%attributes = {"lang":"en","invisible":
// --[HEADERSTART]-------------------------
// User name (OS): Joshua Fletcher
// Date and time: 04/20/12, 16:44:21
// ----------------------------------------
// Description:
//   Insert a log record from the log tab
// --[HEADEREND]---------------------------
C_TEXT($1;$methodPath_t)
C_LONGINT($2;$methodType_l)
~~C_BOOLEAN($3;$tableInvalid_f)~~
~~C_BOOLEAN($4;$FormObjectNameInvalid_f)~~
~~C_BOOLEAN($5;$ObjectNameInvalid_f)~~

```

CE_EXT_LogInsert.txt Revision 348

```
//%attributes = {"lang":"en","invisible":
// --[HEADERSTART]-------------------------
// User name (OS): Joshua Fletcher
// Date and time: 04/20/12, 16:44:21
// ----------------------------------------
// Description:
//   Insert a log record from the log tab
// --[HEADEREND]---------------------------
C_TEXT($1;$methodPath_t)
C_LONGINT($2;$methodType_l)
C_TEXT($3;$tableName_t)
C_TEXT($4;$objectName_t)
C_TEXT($5;$formObjectName_t)
C_BOOLEAN($6;$tableNameInvalid_f)
C_BOOLEAN($7;$ObjectNameInvalid_f)
C_BOOLEAN($8;$FormObjectNameInvalid_f)

```

For Help, press F1. Scroll horizontally with Ctrl-Scrollwheel — Left View: UTF8 BOM CR / - 8 — Right View: UTF8

The previous revision is on the left and the current one on the right. In this case the parameter list for the method was altered to accept 8 parameters instead of 5. The lines highlighted in red are considered removed, and the lines highlighted in yellow are new.

Revert, Revert, Revert!

It has been mentioned a few times that every time a 4D database is launched, the binary files (structure, data) are changed. SVN will recognize these files as changed, even if no deliberate changes were made.

When using 4D with SVN, it is important to get into the habit of reverting these files if no changes are made. Adopt this pattern:

- Launch the database.
- Quit the database.
- If no changes were made, **perform a revert.**

To revert using ToroiseSVN just right-click on the database folder and select SVN -> Revert. A list of changed files is presented, select them all and confirm the action.

The reason this is important reveals itself when it is time to update. If someone else has committed the structure file, for example, and the structure file in the working copy was opened in 4D since the last update, it will not be possible to complete the latest update. SVN will recognize that the structure file in the working copy has been changed and prevent it from being overwritten by the copy in the repository.

Note: *A quick way to check and see what has been changed is to "soft" commit; open the commit dialog without accepting the commit and look at the file list.*

Make a habit of reverting when necessary to avoid this issue.

Note: *Make note of what your SVN client does when reverting. TortoiseSVN for example places the reverted files in the Recycle Bin. This can be convenient if they need to later be retrieved but on the other hand they do occupy disk space.*

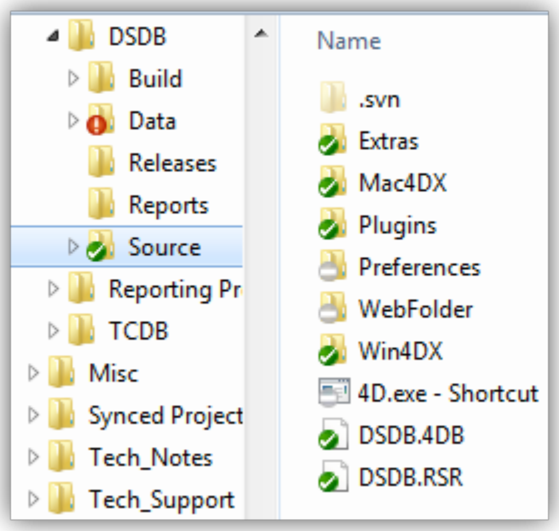
Data File Management

Committing changes to the repository involves sending each modified file to the server. Simply launching a 4D database modifies the data file, even if no data was changed. Thus every commit involves copying the entire data file to the server by default, which can take a significant amount of time with large data files.

Note: *The data file is still **stored** differentially, but the entire file must be copied to the server so that the server can figure out which parts have changed.*

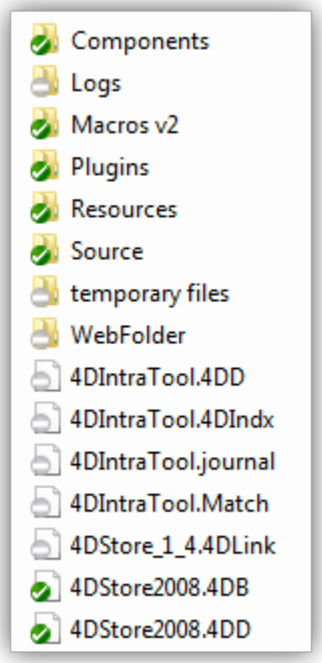
Consider two solutions to avoid this:

- Store the data file in a separate working copy. Only commit this working copy when absolutely necessary. In this way the commits for the structure file (and related files) will be much faster. For example, here is a project that has the source and data file partitioned in this way:



The “Source” folder (which is committed and up to date) contains all of the database files **except** the data file. The data file is stored in the “Data” folder (itself a working copy) and has **not** been committed since the changes are not currently important. The Data working copy can persist in its non-committed state and therefore not slow down commits for the Source.

- Add the data file to the ignore list. This is especially handy for development-only/test data files that really have no business being in the repository. Here is an example of a project where a test data file is used for development instead of the matching data file:



Components
Logs
Macros v2
Plugins
Resources
Source
temporary files
WebFolder
4DIntraTool.4DD
4DIntraTool.4DIndx
4DIntraTool.journal
4DIntraTool.Match
4DStore_1_4.4DLink
4DStore2008.4DB
4DStore2008.4DD

The minus sign icon indicates that a file is ignored. In this case all of the “4DIntraTool” files have been ignored because they represent test-only data. The “real” data file (4DStore2008.4DD) is rarely accessed and therefore rarely affects the commit speed.

Ignore List

SVN allows the developer to ignore files (either by name, or by extension). There are definitely certain 4D files that are important to have in the ignore list:

- **Script Files** – working with SVN naturally lends itself to automation via scripting (build automation for example). These script files tend to be computer specific and, if so, should be placed on the ignore list. In this way other users do not get files they cannot use.

- **4DLINK Files** – 4DLINK files (aka Database Access Files) do not support relative paths for the Structure and Date file. Thus a 4DLINK file that contains these paths is only useful on that machine. Consider adding these to the ignore list.
  - On the other hand it can be useful to provide a template 4DLINK file in the repository for other uses to copy as needed. The file can contain the database credentials, for example, to bypass the login during development. Other users can make a copy of this file and add their copy to the ignore list.
- **Logs Folder** – files in the database "Logs" folder tend to be machine-specific. Consider ignoring this folder.
- **Preferences Folder** – as with the Logs folder, the "Preferences" folder may contain machine-specific files (especially build project files). Be careful to ignore as appropriate.

Note that adding a file to the ignore list is a change that must be committed. Said another way, all users benefit from the same ignore list.

Differential Storage

It is worth repeating that SVN offers and extremely efficient, multi-version backup of the database since it only stores the differences between different revisions.

Tracking Changes

It is important to understand that SVN is quite good at detecting when a file is changed. It does not, for example, use the date- nor time-modified of the file. SVN uses a hashing scheme to track changes. This can be quite powerful.

The CodeExport component includes a feature to re-export all methods in the database. This means recreating every method file on disk. But SVN is smart enough to know whether or not the methods have really been changed. If no methods have been modified, re-exporting them has no effect on the working copy.

Export

SVN has a feature to create a copy of a revision, but without the necessary data to maintain a working copy. This command is called "export". This command is useful for creating copies of the project for deployment, for example. A working copy should never be deployed to a customer. Instead export the desired revision and deploy that.

Stage 3 (And Beyond)

Exporting the database code is not the final step in using 4D with SVN. This section looks towards future issues that may be tackled.

Method Renames/Deletes

In fact it is possible to handle method renames and deletes as long as they are treated as the same thing. The Source Toolkit command METHOD GET PATHS is the key. Renames and deletes can be detected by comparing the list returned by METHOD GET PATHS and looking for any missing paths.

The complication is what to do when a missing path is detected. Is it a deleted method or a renamed method? The only way to really know is to ask the user, but this can be disruptive during development.

A second complication is that in order to delete the method text file from SVN, a command must be explicitly issued. Deleting the file from disk is not sufficient. Thus working copy management must be coupled into 4D. This makes any solution (like the CodeExport component) far more complex. What if the SVN command fails? What if the server is not reachable?

The CodeExport component does not address this issue for these reasons as the intention is for the component to have no impact on the developer (or as little as possible).

Team Development

One of the foundations of revision control software is team development. Ironically perhaps, this topic has only briefly been mentioned in this Technical Note. But this is for a good reason. 4D already has a team development solution: 4D Team Developer.

4D already allows multiple users to work on the same database using 4D Server and 4D Remote. If team development is necessary, even in an SVN context, the recommended technique is to use the feature 4D already has. When the team is done working (e.g. at the end of the work day), commit the changes to SVN at that time.

By contrast team development in the style of revision control software is very difficult with 4D. There is no effective way that more than one person can work on the same structure file at the same time without 4D Team Developer.

Note that the CodeExport component supports client-server development (the code runs on the server) so it fits perfectly with 4D Team Developer.

Other Structure Objects

Though the Source Toolkit only provides access to methods, it is nonetheless possible to export other types of 4D objects.

- The database structure can be exported in one of several ways.
    - Copying objects in the Structure editor places an XML representation of the structure in the pasteboard. This can be exported to a text file.
    - The SQL System Tables can be used to generate a text version of the structure as well.
- Forms can be exported as images via FORM SCREENSHOT command. While limited in usefulness it does provide a quick way to visually detect the differences between revisions of a form.
    - An additional 4D v13 command, Equal pictures, can be used to detect if two pictures are different.
- Users and groups can be exported as a BLOB or text via built-in commands.
- Lists, menus, and other Tool Box items can be exported in the same way.

So, even without direct support via the Source Toolkit, it is possible to build an even more robust audit trail of a 4D database.

CodeExport Component

The CodeExport component facilitates automatic export of all methods in the host database to text files on disk. If the host database is under revision control with SVN, these text files are suitable for committing to the repository, thus giving the developer to ability to track changes to methods over time.

Documentation for the CodeExport component is provided in a separate file.

Note: *CodeExport requires 4D v13.*

Conclusion

More than anything else the goal of this Technical Note is to convince every single 4D developer to strongly consider trying revision control with 4D. From solo developers to large teams, the benefits of revision control are immediately apparent. SVN provides an easy way to get started because it is flexible and adaptable to 4D. The CodeExport component makes the first step even easier by automatically exporting all of the database methods. There is nothing to lose, and so much to gain.

Related Resources

- VisualSVN Server - http://www.visualsvn.com/server/
- TortoiseSVN - http://tortoisesvn.tigris.org/
- Cornerstone - http://www.zennaware.com/cornerstone/
- Versions - http://versionsapp.com/