

# Versioning and the Repository

## 1 Introduction and Warning

This is user documentation for the repository and versioning features of Sheets. Unfortunately, the code that implements this is not fully debugged, and has significant holes in its functionality. The purpose of this documentation is to describe what the repository code is trying to do. We cannot recommend that anyone use this code in its current state, which is why this documentation is split off from the reference manual.

## 2 Repository Overview

The repository provides version management capabilities along the lines of those provided by CVS and other source control systems. However, since Sheets is not file oriented, the resulting system ends up being rather different than file base systems.

The traditional reason for using such systems is that it greatly aids concurrent development by multiple programmers. This reason also applies in Sheets. Although it is possible to use Sheets with an external file-oriented versioning systems such as CVS, it is painful, because merging must be done on sheets dump files, which are only vaguely human readable. In addition, we have found that when a dump file becomes very large, diff-based text merging starts giving bad results.

We also feel that an environment which makes history browsing truly easy can significantly aid program comprehension. The rationale behind a particular program structure may become much more obvious when you can see how it started out, before many incremental changes were layered on top.

## 3 Basic Versioning Model

- The smallest unit of change is a fragment. Whenever a fragment changes in any way, a new version of that fragment is created, and this version supersedes the old version.
- Changes are organized into *deltas*. A Delta is a conceptually related group of changes to a project. Each delta has change documentation associated with it. In implementation, a delta is a set of new definitions for all the changed components plus this documentation.
- When determining what changes have been made, sheets and list attributes are compared

by comparing the abstract identity of their contents (object ID), and not any representation of the content value. So if you simply edit a method, but don't move it, then the method is changed but not its sheet.

- A fragment is considered changed if either its text contents or some attribute changes. Unlike in file-based systems, the history of a fragment is maintained entirely independently from where the fragment lives. This means that reorganizing code is much less painful and antisocial than in other source control systems, since a move is not confused with an actual change to the moved code.

## 4 Update/Open/Check-In model

To begin working on a project, you must check it out into your database. Currently the only way to check out a project is to specify `-pproject-name` on the command line, such as:

```
sheets -pSheets
```

This checks out the latest version of the project. There is currently no way to check out any version other than the latest. This command line argument is analogous to specifying the `.sheets` file for a project: it adds the contents of the project to the database.

In order to get any changes that may have been checked in since your checkout, you must explicitly update your database. Updating does a three-way merge of the base version in the repository, the new repository version, and the local version. In some cases there will be merge conflicts which must be manually resolved.

In order to store your changes in the repository you must check them in. This is actually a two-step process. First you create an *open delta*, then you check in that open delta. The intent of the two-step process is to allow you to group unrelated changes in your database into distinct deltas which can then be checked in separately.

Your database must be up to date w.r.t. the repository before you can check in a delta. If you got to check in and someone else has gotten there first, the checkin will fail, and you must update.

### **repository-update**

Update all of the checked out projects. This will reconcile the repository changes with any local changes, generating a conflict report when overlapping changes have been made to the same fragment.

### **resolve-merge-conflict**

The conflict report for a project contains *conflict sheets* containing the information concerning a particular conflict. Each conflict sheet has the base version, the local version, the new repository version and a merged result created from the text of those

## Versioning and the Repository

fragments using `diff3`.

Most often you will edit the merged version to be correct. Once you have done this, you can run this command to tell sheets that the conflict has been resolved.

You can also resolve a merge conflict by selecting one of the other versions of the fragment in the conflict sheet and doing `resolve-merge-conflict`. Usually you would select either the current local version or the new repository version, however it is also permitted to select the base version.

### **show-merge-conflicts**

Regenerate the merge conflict reports for all checked out projects. The conflict report sheets are generated automatically when you do an update, but the report sheets are not persistent, so if you exit from sheets they will be lost. The conflict information **is** persistent, so the reports can be regenerated when you restart.

### **open-delta**

This command creates an open delta for each project in the local database which has changes that haven't been checked in. The open delta is just a sheet containing all of the modified fragments. Modified fragments that are already in some open delta won't be added to the new one. You can move fragments between open deltas just as you would move them between any two ordinary sheets.

You can (and should) also add documentation to the open delta describing what changes were made and why. This documentation becomes the change log information for the delta when it is committed.

Simple text summary information should be entered into the sheet description (the gray bar at the top of the sheet.) You may also add any other sort of documentation fragments to the contents of the open delta, and you can freely reorder the fragments in the delta.

It is currently required that all fragments in the delta must appear directly in the open delta sheet (not in some subsheet.)

Once you are satisfied with the delta, you can check it in to the repository using [commit-delta](#).

### **commit-delta**

This command checks in to the repository the changes associated with the selected *open delta*.

## 5 History Browsing

The biggest improvement in versioning support that Sheets offers in comparison to file-based

systems is easy history browsing. Sheets stores history both per-project and per-fragment.

## 5.1 The Log Sheet

The history of each project is stored in the log sheet. The log sheet is kept somewhere within the project that it is the log for, usually at the end of the project's root sheet. The log is simply a listing of all the deltas that have been checked out into the local database. The most recent delta is at the head of the log, with successively older deltas following it. If there are any open deltas, they will appear at the head of the log.

A *delta* is a fragment describing some change to a project. Deltas are created by [open-delta/commit-delta](#). A delta looks basically the same as the open delta that it was created from, but is read-only. When you open the log sheet, the deltas will be in header view, so you will see the summary text for each delta.

At first, the log sheet looks just like an ordinary change log, but you can show the contents of a delta and see all of the fragments in that delta. Furthermore, for any fragment in the delta, you can see diffs associated with the changes to that fragment, and can also do most of the fragment operations that you could do in a normal non-historic context.

## 5.2 Historic Fragments

In order for you to be able to easily browse historic code, special *historic fragments* are used to represent past versions of each fragment. Historic fragments have some peculiar properties and behavior.

Most obviously, historic fragments are read-only. You can't change the past, only make it obsolete.

With historic fragments, there is also some concern that either the user or the sheets semantic analysis engine might become confused about what the current version of the program is.

To avoid complicating semantic analysis, historic fragments simply do not appear in the indices used by the general query mechanism and the various shortcut commands. This means that if you go to the references of some method, you will never be presented with historic fragments, only with fragments currently in the program. However, this also means that you can't do queries on historic fragments. Queries on historic fragments will either simply fail, or will return the result with respect to the current program, not the historic one.

To avoid confusing the user about whether a fragment is current or not, historic fragments are displayed on a yellowish off-white background instead of white. This background cue is suppressed when browsing the delta itself, as it is redundant in that context, and also looks bad.

## 5.3 History Attributes

### Delta Attribute

Every fragment that has ever been checked into the repository has a `delta` attribute. This attribute is a one element list containing the delta that introduced this version of the fragment. This is just a back-pointer from each fragment to the delta that contains it. The delta of a non-historic fragment is simply the last change that affected this fragment.

### Changes Attribute

Every fragment that has more than one version also has a `changes` attribute. The `changes` attribute is a list of all the versions of the fragment, with the current version at the head of the list. This attribute is effectively a per-fragment change log. To facilitate this sort of use, fragments in the `changes` attribute by default come up in the [changes view](#).

## 5.4 The Changes View

The `changes` view is a `Sheets` view which is an alternative to the `full` view for textual fragments. When a fragment is in the `changes` view, it is displayed as a text `diff` between the displayed fragment and the preceding version of that fragment.

Fragments displayed via the [changes attribute](#) are placed in the `changes` view by default. You can switch to the normal `full` view by `right-click/Show As Full`. In other cases, you may want to explicitly place fragments in the `changes` view:

- When browsing a delta, you can put a fragment in the `changes` view to see exactly how the fragment was changed.
- When preparing an open delta to be checked in, you can put a fragment in the `changes` view to see how a particular fragment was changed.

### 5.4.1 Limitations

There are some problems with the `changes` view that come from running a `diff` subprocess. One problem is that you must install a `diff` utility like Gnu `diff`, and must [configure sheets](#) to use it. Another serious problem is that if you place a sheet containing hundreds or thousands of fragments into the `changes` view, this operation may take many minutes to complete.

Another class of problem is with actually displaying the changes. The `changes` view only compares the text contents of the fragment. It does not compare attribute values or container contents.

If a fragment attribute has changed but the text hasn't, then the changes view will report no changes. The actual underlying object comparison used by the repository is aware of attributes, so if a fragment appears in a delta or open delta, it did change somehow, but you may need to look at the attributes to see how.object

There is also no way to compare the contents of two sheets, which is a serious problem. If the fragments contained in a sheet changes, the sheet will be put in corresponding delta, but there is no way to tell how the sheet changed other than side-by-side comparison.

## 6 Configuring Sheets for the Repository

**diff3-command : string = "diff3 -m "**

This is the command which sheets runs to merge fragments when there is a conflict during an [update](#)

**diff-command : string = "diff -u "**

This is the command which sheets runs to compute the [changes view](#) for the text contents of a fragment.

**repository-hosts : string = ""**

A comma-separated list of the host names where repository servers are to be found. See [Client/Server Structure](#)

## 7 Using Both Dump Files and the Repository

You can use both normal project dump files and the repository. If a project is in the repository and you create a dump file, the dump file will contain the entire history of the project, and will be segmented into deltas. Stuff that has not been checked in yet is also dumped, but is not in any delta. Loading this file will restore the database state as of the time that the file was dumped.

You can migrate a non-versioned dump file into the repository by loading the dump file into sheets and doing a [check in to create a new project](#). As far as the versioning facility is concerned, loading a dump file containing modified fragments is just the same as modifying those fragments by hand.

## 8 Client/Server Structure

The repository itself is managed by a fairly simple server process. A repository server can serve several different projects. The state of each project is stored as files in the server filesystem. The format of the server files is basically the same as the dump file format.

## Versioning and the Repository

Each user sheets process can be a client of several repository servers. All of the projects on all of the known repository servers are registered in the project namespace, so project names must be unique at this level. However, a project only actually becomes part of the database when it is *checked out*.

### 8.1 Basics

To set up a repository server, create an empty directory for the repository data to be stored in. `cd` to this directory and run the server:

```
java org.browsecode.repository.Server Project1 Project2 ...
```

When running the server, the directory where sheets is installed must be in the classpath. If you have not globally set `CLASSPATH` this way, then you must specify `-classpath` as well.

The command line arguments to the `Server` class are the names of the projects which are to be served. A subdirectory with that name will be created for each project. When deltas are checked into the project, they are stored in files in the project directory with names `1.delta`, `2.delta`, etc.

When the server is started, it checks for each project directory's existence. If there are any existing deltas, it will *recover* them, which just means that the server reacquaints itself with the existence of those changes so that it can serve the deltas to clients.

The server is designed to be failure tolerant. A crash during some server operation will at worst result in the most recent checkin being discarded. There is no special mechanism for shutting down or restarting the server. Just terminate it and run it again.

The server logs stuff to standard output for each command that it processes. This may be useful for determining that the server is alive or that someone is currently accessing it.

### 8.2 Adding a New Project

To create a new project and check in initial stuff for that project, you must first restart the repository server with the new project on its command line. Then start up sheets with the [repository-host](#) specified in the `.sheetsrc`.

The fragments that you want to check in must of course have their `project` attribute set to the correct project name. For Java fragments, you need only set the project name of the package.

After you've got some fragments in the project to be initially checked in, just do [open-delta](#) and [commit-delta](#) as you would normally do. `open-delta` will create an open delta for any project that has modified fragments, even if it has never been checked out (e.g. because it

didn't previously exist.)

### 8.3 Database Administration

It is intended that the directory structure and delta files be directly manipulated via file commands and text editors. All the server does is serve out existing deltas and record new ones. Other operations that you might possibly want to do to change the content of the repository can be done by directly hacking the files. You should kill the server while making changes and restart it afterward, since the server caches some information at startup time and reads other stuff as needed.

Each file `1.delta`, etc., is the representation of a single delta. In a direct reflection of the accumulative versioning model, the `.delta` files are never modified after creation, only read. The numbering of the `.delta` files is the order in which they are checked in, which is also the order they appear in the log sheet, and the order in which they supersede one another.

You can trivially back up the state of a project repository simply by removing the files for the last N deltas. Note that server only considers files with the type `.delta` to be deltas, and that other files may exist in the project directory. This means that you can cause a delta "`42.delta`" to not exist without actually deleting it by renaming it to e.g. "`42.bad`".

Note that the server reads the delta files in order when recovering, and will assume it has reached the last checked in delta when it fails to find the next file in sequence. In order to support the possibly useful operation of dropping deltas out of the middle of the log, the server will look for "`nnn.void`" when it fails to find `nnn.delta`. If a `.void` file is found, then the server simply goes on to the next number.

The idea is that you can rename a file from `nnn.delta` to `nnn.void` to cause that delta to be dropped. Note that it is actually the mere presence of a `.void` file which causes this behavior, no file contents are actually required.

The format of the `.delta` files themselves is nominally human readable. You can in principle change anything in a delta by editing the file on a server. Note that the client repository update assumes that deltas don't change, so if you do modify something, clients must rebuild their databases rather than just doing an update.

### 8.4 Limitations

The current repository server implementation is single-threaded. This means that only one client can access the server at a time. In the case of checkin, this is a useful form of mutual exclusion, but for update it is more of an annoyance.

## *Versioning and the Repository*

There is no security. This is not quite as bad as it sounds, since all that anyone can do is can do is add spurious new deltas, and those deltas can be deleted as described below. And of course, anyone can read what has been checked in.