Technical Report

Shrew – A Prototype for Subversion Analysis

Philipp Bunge
Supervised by: Tudor Gîrba
University of Bern, Switzerland
Software Composition Group

February 2007

Abstract

With the growth of the World Wide Web, version control systems have become an essential component in collaborative software development. One such version control system that has found generous adoption in recent years is Subversion, a centralized system that was designed explicitly to match the requirements of the open-source community. Equally, specialized web based tools have emerged to browse and inspect version control systems such as Subversion and have proven themselves to be valuable instruments for the developers of software projects. As projects become larger and more complex however, these tools have often reached their limitations on the level of introspecting they can provide. To solve this problem we present Shrew, an approach to analyze Subversion repositories that builds upon a specialized meta-model and makes use of the Moose object-orientated reengineering environment to facilitate information extraction and that presents its results with a convenient web interface.

1. Introduction

The history of web based version control systems can be traced back to 1996 when Bill Fenner released a script for browsing CVS repositories called CVSweb

[CVSweb, 2007]. Since then, CVSweb has come a long way and inspired the application to be ported to Python which resulted in the well known ViewCVS project [ViewVC, 2007]. Today, ViewCVS has been extended to support the Subversion version control system as well and has been renamed to ViewVC. Further products such as Chora [Chora, 2007], FishEye [FisheEye, 2007] and Trac [Trac, 2007] have been developed to support version control systems unsupported by ViewVC or to provide functionality that the system does not cover.

Despite of the existing abundance of such web based tools, few exist that facilitate more complex historic analysis that could be used for reverse and reengineering purposes while adhering to a straightforward web based interface. Shrew is intended as a suggestion how this gap can be filled. For this purpose, Shrew makes use of the Smalltalk programming language, the reengineering environment Moose [Ducasse *et al.*, 2000] and of the Seaside web application framework [Seaside, 2007] to analyze and browse Subversion repositories.

Subversion was chosen as the preferred version control system for Shrew as it has a wide and growing acceptance in the development community, particularly in open-source circles. It has a more modern design than and does not suffer from the inherent flaws of CVS. Finally, Subversion has a less complex architecture than distributed version control systems such as *darcs* [darcs, 2007] or *Git* [git, 2007].

Shrew is designed as a layered architecture as displayed in Figure 1 on the following page to support reuse of its components. At the bottom is the *repository access* layer, followed by a *data analysis* layer and finally the *presentation* layer. These are discussed in detail in the following sections.

2. Repository Access

Before any analysis can take place, it is essential that we can access the data stored in a Subversion repository. Subversion provides multiple mechanisms to access remote repositories of which the most popular uses WebDAV/DeltaV which is an extension of HTTP. Two other possibilities are *svnserve* – which uses a custom protocol – and direct file access if the Subversion server and client are on the same host and permissions are set accordingly. To make

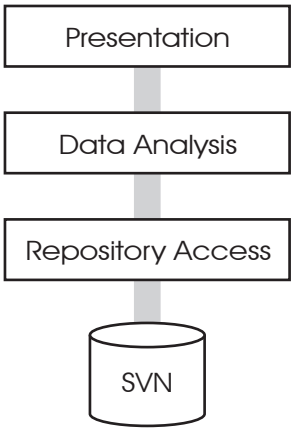

Figure 1: Shrew layered architecture

matters more complicated, WebDAV data can be tunneled via SSL, svnserve over SSH and both may provide authentication mechanisms to restrict access.

Essentially, the Subversion client implemented in Shrew should provide all of these methods so as to support all projects that use Subversion as a version control system. Furthermore, the initial design of the client attempted to provide a full-fledged Subversion client as developers are used to using on the command line with exception that the working copy of a checked out repository would not be file based but object based. This means, that a user would be able to check out a Subversion repository into Smalltalk and manipulate all files and directories as objects. Once manipulated, running a *Subversion commit* on the objects would allow the developer to upload the data back onto the repository server. The goal was to provide all of the common Subversion functions such as *checkout*, *update*, *commit*, *add*, *delete*, *merge*, *lock* and so forth with the confidence that a code-repository such as VisualWorks' Store or other applications could be built on top of them.

2.1. Discussion of Possible Implementations

With these considerations being made, there are a total of five solutions to implementing a client in VisualWorks Smalltalk that were considered: (1) using Subversion's *SWIG bindings*, (2) importing the Subversion API using *DLLCC*,

(3) writing a C wrapper around the API, (4) implementing the protocols natively and (5) calling the Subversion console client as a separate process.

I present these solutions in the order of the "elegance" of the solution. In principle, the items are sorted by an estimate of their maintainability and efficiency after having put a decent amount of thought into them.

SWIG bindings Subversion is written in C with a strong modular design that allows developers in C or C++ to simply link to or dynamically load Subversion libraries into their applications. This has been allowed by Subversion being designed from the beginning to have a well defined interface – an issue that has plagued CVS extensively. For languages other than C and C++, Subversion provides support using the *Simplified Wrapper and Interface Generator* [SWIG, 2007]. SWIG is an *interface compiler* that takes an interface definition file and the C or C++ header files from an application to generate the wrapper code required to access the underlying code with the higher level language.

Subversion already provides the interface definition file so SWIG would provide an optimal solution for implementing a client. Unfortunately, SWIG does not provide a language definition for any Smalltalk dialects natively and a related project [Upright, 2007] attempting to provide such a definition is still quite experimental and has not shown any progress in close to a year at the time of this writing. Writing such a language definition, although not difficult, would require an amount of time that was not available during the period of the project.

DLLCC VisualWorks Smalltalk comes with a suite of tools named *DLL and C Connect* (DLLCC) that can be used to generate and use interfaces between Smalltalk and C. DLLCC can automatically parse the header files of a C application and construct the necessary bind code in Smalltalk or the developer can create the binding code manually. Unfortunately, DLLCC was unable to parse the header files of Subversion correctly and a manual implementation would have been a tedious process. Additionally, the primitive data types would have needed to be converted to higher level objects adding more complexity to this solution.

C wrapper A third solution is to write a wrapper around Subversion as a C library and only exposing the interface as needed by Shrew. We could then use DLLCC to construct the binding code automatically or even consider a manual implementation as the header files would be a lot smaller than for the entire Subversion code base. However, the issue with the data type conversion persists, rendering this an unfeasible solution as well.

Native protocols Alternatively, one could implement the protocols Subversion uses to communicate over a network or to the file systems directly in Smalltalk. However, the protocols are all rather complex and insufficiently documented which made it difficult to estimate the time required to implement even one protocol. Furthermore, the protocols are subject to change as Subversion develops, so we could not have guaranteed interoperability in the future and maintenance would have been difficult.

Console client The last proposal for a client solution would be to call the Subversion console client, passing any options along the command line and parsing the text output that is returned. Obviously this comes with an expensive overhead: every time information is retrieved from a remote location we start a new process, fulfill a TCP handshake, possibly an SSL handshake, exchange Subversion data including authentication, tear down the connection and finally return a string result along the command line. If the result is in XML format, we will additionally fire up an XML parser to interpret the data.

Needless to say, this last solution is far from elegant, efficient or maintainable but it permits for the quick prototype implementation that was needed to be able to continue the development of Shrew.

2.2. A Brief Look at the Client

The implementation of a full-fledged client as discussed above was thwarted by the use of the console client rather than a different back-end mechanism and it soon became obvious that such a client was not the ideal structure for the analysis tools. For this reason, Shrew only implements a partial set of the

Subversion client commands and not all exhibit the same behavior as their console counterpart.

Listing 1 shows an example how the client is used to check out a project. The revision: part of the message is optional if the user wants to checkout the newest version.

```
root := SVN
    checkout: 'https://example.org/svn/repos'
    revision: (Revision number: 12)
```

Listing 1: Checking out a repository

Revisions can be specified by a number of messages of Revision. These are #head, #number: and #date: which correspond to the revision arguments as specified for the console client. #number: takes an integer value and #date: takes an instance of Timestamp as argument.

The SVN class as used above is not actually the client implementation but acts as a wrapper. It overrides #doesNotUnderstand: on the class side to delegate messages to the concrete implementation of the Subversion client. Which client this is, is specified in SVN class>>#defaultClient.

Since the client is incomplete, most users will not be interested in the client as such, but will only use it in the context of the data analysis as will be presented in the next session. The data analysis layer relies solely on a handful of the client functions such as *info*, *list* and *log*. Furthermore, it uses a caching wrapper around the default (console based) client to avoid retrieving the same information from the server more than once. This implementation is called CachedConsoleClient and its implementation is rather straightforward.

3. Data Analsis

Oncee the data from a Subversion repository can be accessed within Smalltalk, the next step is to analyze this data. This is the responsibility of the model as discussed below.

3.1. Model import

Shrew's meta-model is tightly integrated with Moose [Ducasse *et al.*, 2000], a language independent reengineering environment developed at the university of Bern, Switzerland. To analyze a Subversion repository, Moose provides a menu entry "Import from Subversion repository into new model" that prompts the user for the URL of the repository to analyze and then imports the data.

Behind the scenes, Moose delegates the responsibility of creating the meta-model to ProjectBuilder which defines four public methods: buildProject:from:to:, updateProject:to:, buildProject: and updateProject:. The latter two are convenience methods which build all possible revisions or update to the newest revision respectively. The ability to provide a range of revisions allows the user to analyze only a subset of an entire Subversion repository.

The build methods expect a string or a URL pointing to the repository location and return an instance of ProjectHistory that is the base of the meta-model (see Figure 2 on the following page). To later update the project to a newer revision, the instance is simply passed to updateProject:.

The functionality behind ProjectBuilder is complex and will not be discussed here. The responsibility is separated within small methods however, which allow for easy maintainability.

3.2. Meta-model Design

The design of the meta-model builds on top of Hismo [Gîrba, 2005], an approach that models history as a first class entity. A history is an ordered set of versions where history and version are generic concepts that can be applied to a file or a directory for example. This scheme is shown in Figure 2 on the next page.

Every file and every directory will have a multitude of versions throughout the development of a software project. Collectively, the versions constitute the history of the respective file or directory.

Simultaneously, the entire Subversion repository has a history which is made

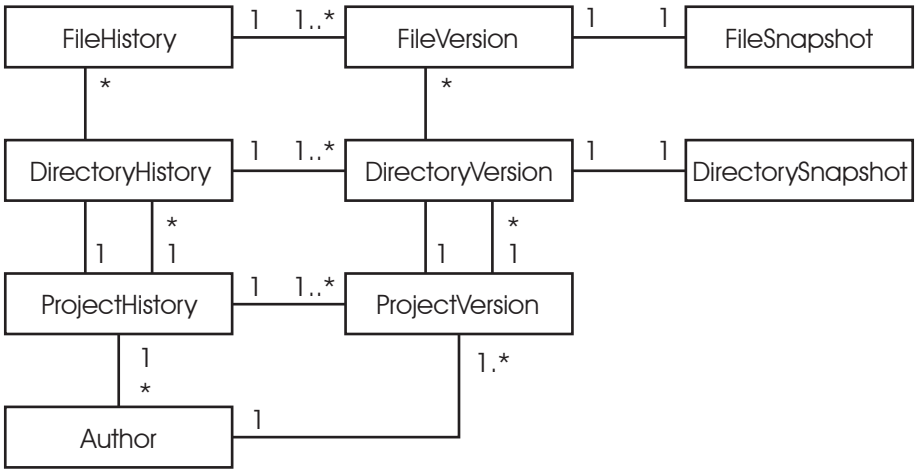

Figure 2: meta-model UML

up by the versions of the repository. These entities are named ProjectHistory and ProjectVersion respectively. In Subversion, a new version is created every time a user commits to the repository. The revision number is incremented and a possible commit message is attached to that version, rather than to the modified files and directories.

Now that we have entities for concepts such as a FileHistory and a ProjectVersion we can map relationships between them. As mentioned above, Histories consist of their respective Versions but there is also navigation between versions and between histories. Whenever a user commits to a repository, he or she commits changes to files and directories that are part of this repository. This means that a number of file and directory versions are correlated with a specific project version. Parallel to this, the history of a repository will consist of a number of file and directory histories.

One entity in Figure 2 is Author. Each Author encapsulates information about a user that has at least once committed to the repository. An Author can be understood as consisting both of a history and of a version simultaneously as a user of Subversion repository does not have attributes that change over time and is therefore modeled as a single entity.

Finally, the entities FileSnapshot and DirectorySnapshot simply encapsulate the content of the respective file or directory and are used to delegate this responsi-

bility from the file and directory versions. A snapshot is associated with exactly one version.

3.3. History Properties

An advantage of structuring the meta-model as presented above is that it facilitates the measurement of historic properties. Using measurements in historic analysis is of particular use as we can quantify the changes within the history without having to take a detailed look at each version.

Some examples of measurements are the *age* of a history – that is the number of versions it contains – or the deviation of a property P over time. Such a property P could be the number of lines of a file or the number of modified files in a commit, i.e. in a project version.

A more complex example of a measurement is the concept of "code ownership" by Mauricio Seeberger [Seeberger, 2006]. He suggests that a trivial approach would be to retrieve all versions of a file and applying a *diff* algorithm to determining which lines were last changed by which author and thus determining the percentile code ownership of each author for that particular file. Seeberger correctly points out that this solution is not feasible for large histories or large files and suggests an approximative approach. He uses CVS logs that contain the number of lines a particular author modified during each commit and he assumes that the changes are evenly distributed over a file. He then defines the code ownership for an author α of a file version f_n as follows, where s_{f_n} is the size of file f in version n, α_{f_n} is the author that committed the change and l_{f_n} is the number of lines he or she added.

$$
\begin{aligned}
\text{own}^{\alpha}_{f_0} &:= \begin{cases} 1, & \alpha = \alpha_{f_0} \\ 0, & \text{else} \end{cases} \\
\text{own}^{\alpha}_{f_n} &:= \text{own}^{\alpha}_{f_{n-1}} \cdot \frac{s_{f_n} - l_{f_n}}{s_{f_n}} + \begin{cases} \frac{l_{f_n}}{s_{f_n}}, & \alpha = \alpha_{f_n} \\ 0, & \text{else} \end{cases}
\end{aligned}
$$

Although a very efficient way of measuring code ownership, the implementation

of this method fails in she current version of Shrew because Subversion does not provide information about the number of line changes in its log. If, as we had previously discussed, the implementation of the client did not rely on the console client but called the Subversion API directly, this information could be easily obtained and such a measurement would be feasible.

As of the moment, Shrew is therefore limited in its measurement possibilities due to the restrictions imposed by the client implementation.

3.4. Further Research

In comparison to version control systems such as CVS, Subversion has the great advantage that it tracks information about files and directories such as copies, moves and deletes. This means that if a file is moved or renamed, it remains part of the same history. Subversion therefore does not implement separate functionality for tagging and branching as one is accustomed from CVS. Instead, a user simply copies the directory tree to a new location. If it remains unchanged it can be interpreted as a tag and if a lot of development work occurs on it later, the copy was probably intended to be a branch from the original.

Now the meta-model as implemented and described in section 3.2 does not currently allow for such copies and moves to be modeled properly. Instead, if a file is copied or renamed a new history is created and no information about the relationship to the old history is stored.

This calls for an extension of our current meta-model which should not, however, break the current meta-model. Rather, the current meta-model should be able to coexist as a *degraded* interpretation of this extended meta-model.

Our proposed solution is to implement relationships between the existing history entities. This idea is displayed in Figure 3 on the next page. In the diagram, the small squares represent the versions and the surrounding rectangles the history of a file. Shaded squares indicate that the file was modified in that version. The numbers at the bottom correspond to the revision of the repository and therefore to a particular project version.

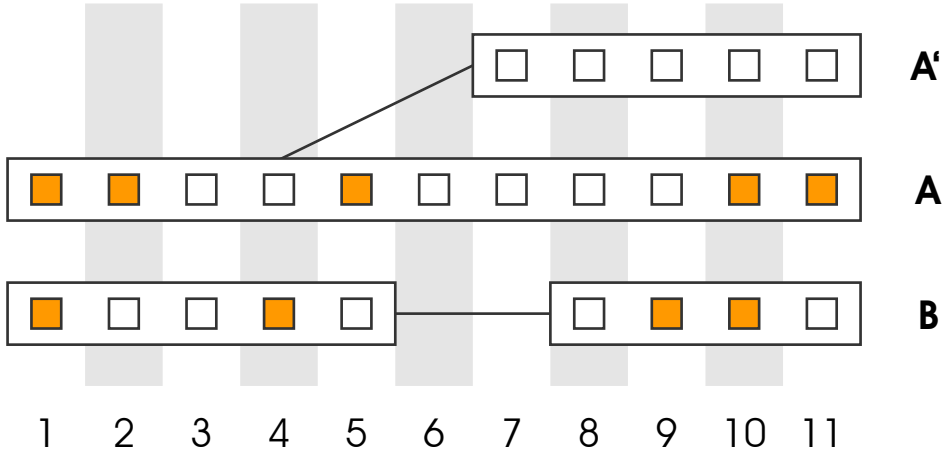

Figure 3: Rename allowing meta-model

The diagram displays three files A, B and A'. File A' was created in revision 7 and is a copy of file A as it was during revision 4. As file A' remains unchanged we can interpret that it was meant as a *tag* of file A during revision 4. File B was deleted in revision 6 and later resurrected in revision 8.

Now the edges displayed in the diagram are the proposed extension to the meta-model. They represent what we call a *history relationship* and encapsulate a reference to the version copied and the first version of the new history. Histories are extended that they have an optional reference to such a history relationship.

Using these *history relationships* now permits to model moves and copies as discussed above. Optionally, all histories connected by history relationships could be collected within a *co-history* entity for analysis.

4. Presentation

The uppermost layer of Shrew is the presentation layer. It has the responsibility of using the data as provided by the data analysis layer and displaying it in a manner that is accessible to the end user.

In Shrew, the presentation layer is twofold. First of all, the Moose reengineering

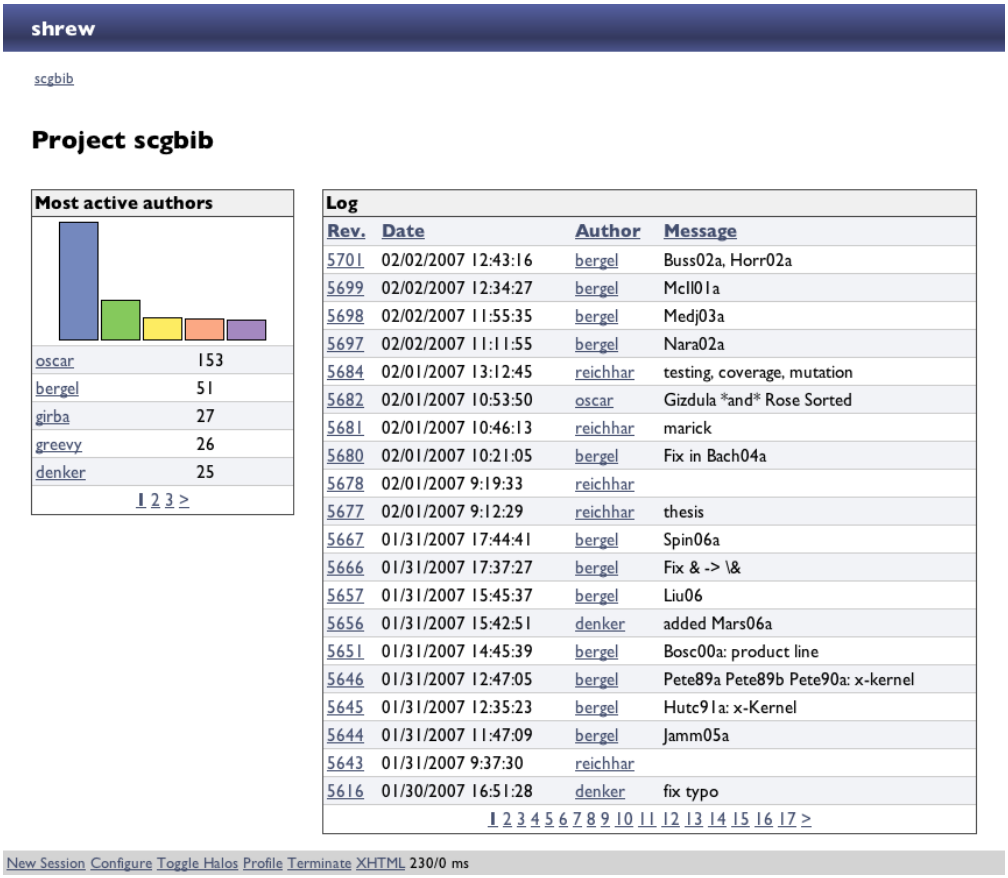

Figure 4: Shrew web frontend

environment provides easy access to all information within the meta-model. The structure can be navigated and various visual analysis can be performed with tools such as Mondrian [Meyer, 2006].

Shrew provides a web based interface that is built on top of the Seaside application framework [Seaside, 2007] which is considered the main interface to Shrew. A screenshot of this interface in action can be seen in figure 4.

Shrews web interface currently supports four different types of views:

Project details shows information on the entire project, that is the ProjectHistory it is associated with. This can include such information as the most active authors, the last recent changes and the age of the project.

Changelog shows detailed information on a particular version of the project.

It provides navigation to the author that committed the change, to the files modified displays the commit message if the author specified any.

Author details displays measurements and information on the history entity Author. This currently includes the number of versions by the author and a list of recent changes. It could be extended to include a ranking of the ownership of the files in the newest version and a possibly collaboration metrics with other authors.

Browsing allows the user of the web interface to navigate through the files and folders of a particular version in a tree like manner. It also provides navigation to reach a previous or later version of a particular file and is able to display file contents and other detailed information.

With these views, Shrew allows the user to navigate efficiently through the history entities of the meta-model but retains an approach that resembles strongly that of systems such as ViewVC and others.

5. Conclusion

Shrew has proven itself successful at providing a new approach to version control system analysis. Nonetheless, there are three areas which hinder the current implementation of Shrew to become successful in productive use:

1. The Subversion client is more than inefficient at the moment. Every time the client retrieves information from a repository, a system process is initialized that connects to the remote location and retrieves the datum from the server. The connection is then torn down and the process stopped, requiring the client to repeat this process and causing quite an expensive overhead.

   The author of this thesis strongly suggests that a new implementation would make use of the SWIG interface as provided by Subversion and discussed in section 2.1. This would require writing a language definition for SWIG but the final result would be more maintainable, more versatile

and less prone to changes in the Subversion protocol as say an implementation of the client using the WebDAV protocol. Furthermore, the SWIG language definition could be used for creating entirely different interfaces to other C or C++ based applications.

2. The data analysis currently does not make use of all the information available from a Subversion repository. In particular, the meta-model does not understand file moves and copies and start a new history whenever a file or a directory is renamed.

3. The visualization layer only provides limited information. This is probably the smallest issue and quickly fixed. The only reason the web interface is at a basic level at the moment is because the requirements where not fully apparent. As soon as the users of Shrew request more features, these can be easily added.

All in all, these momentary limitations are encouraging to continue the work on Shrew where convincing enough to keep the word "prototype" in the title of this text.

A. SeasideSCGComponents

As a byproduct of the development efforts of the Shrew web front-end a collection of generic Seaside components was created that allow for quick page construction with reusable elements.

These elements can be separated into two distinct groups: *views* and *panes*. Views define the page layout of a website and come in one, two or three column versions. They provide some convenience methods for displaying various information such as a location (bread-crumb trail) and a title. Intended to be used with these views are the panes that provide a generic page component. Panes come with a title and a content. Specialized types of panes display tabular data as the content or images.

Developers wishing to implement their own panes can write a new class that inherits from Seaside.AbstractPane and override #renderContentOn:. The method

takes one argument which is the WARenderCanvas that Seaside uses to render the HTML components.

If the pane only needs to display simple content, it may be more convenient to use the predefined class Seaside.SimplePane as shown in listing 2.

```
exampleSimplePane
    ^ SimplePane new
        title: 'Example SimplePane';
        contents: [ :html |
            html paragraph with: [
                html strong: 'This is an example SimplePane!
                    '.
                html text: 'It really is, really simple.'
            ].
        ]
```

Listing 2: SimplePane example

In the above listing, a block with one argument is passed to contents: which behaves in an equal manner as overriding renderContentsOn: as described above. Alternatively, contents: also accepts a Seaside WAComponent or anything else that Seaside can directly render such as a String.

A more complex predefined pane is the TabularPane. It provides a convenient interface for displaying tabular data. A simple example is shown in listing 3.

```
exampleTabularPane
    ^ TabularPane new
        parent: self;
        title: 'Numbers';
        rows: #(1 2 3 4 5 6);
        columns: (OrderedCollection new
            add: (TableColumn new valueBlock: #yourself;
               title: 'original'; clickBlock: [ :number |
               self inform: 'You clicked on ', number
               printString, '!' ]);
            add: (TableColumn new valueBlock: #factorial;
               title: 'factorial');
            add: (TableColumn new valueBlock: [ :number |
```

```
                number even ifTrue: [ 'even' ] ifFalse: [ '
                odd' ] ]; title: 'even/odd');
            yourself
        );
        yourself
```

Listing 3: TabularPane example

The important methods in this example are rows: and columns:. The first takes an ordered collection of the data that is supposed to be displayed. In the above example we are going to display a table with some integers and some information about them. The second method columns: takes an ordered collection of TableColumn instances which define what information is displayed and the behavior of the table columns. The column title is defined with title:, the value for each row is calculated using valueBlock: which can either be a symbol which will be performed as a message on each row or a block with one argument. Furthermore, clickBlock: receives a block that causes the value for that row to be displayed as a link and calls the block when it is clicked.

There are two more methods that are not used in the example above. sortBlock: overrides the default sorting mechanism for the particular column and formatBlock: alters the way the value is displayed. The interested reader should take a look at how these are initialized in TableColum>>#initialize to understand how to use them.

References

[Chora, 2007] Chora repository viewer, 2007. http://www.horde.org/chora/.

[CVSweb, 2007] CVSweb, a web interface for CVS repositories with which you can browse a file hierarchy on your browser to view each file's revision history in a very handy manner., 2007. http://www.freebsd.org/projects/cvsweb.html.

[darcs, 2007] darcs, a free, open source source code management system., 2007. http://darcs.net/.

[Ducasse *et al.*, 2000] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In *Proceedings of CoSET '00 (2nd International Symposium on Constructing Software Engineering Tools)*, June 2000.

[FisheEye, 2007] FishEye, analyze, search, share and monitor CVS and Subversion repositories, 2007. http://www.cenqua.com/fisheye/.

[Gîrba, 2005] Tudor Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, Berne, November 2005.

[git, 2007] Git, fast version control system, 2007. http://git.or.cz/.

[Meyer, 2006] Michael Meyer. Scripting interactive visualizations. Master's thesis, University of Bern, November 2006.

[Seaside, 2007] Seaside, developing sophisticated web applications in Smalltalk, 2007. http://www.seaside.st.

[Seeberger, 2006] Mauricio Seeberger. How developers drive software evolution. Master's thesis, University of Bern, January 2006.

[SWIG, 2007] SWIG, a software development tool that connects programs written in C and C++ with a variety of high-level programming languages., 2007. http://www.swig.org/.

[Trac, 2007] Trac, an enhanced wiki and issue tracking system for software development projects., 2007. http://trac.edgewall.org/.

[Upright, 2007] Ian Upright. SWIG language bindings for smalltalk, 2007. http://commonsmalltalk.wikispaces.com/SWIG.

[ViewVC, 2007] ViewVC, web-based version control repository browsing, 2007. http://www.viewvc.org/.