Automating Subversion
An Open Object Rexx Approach

Bachelor Course Paper

Vienna University of Economics and Business Administration
Department of Business Information Systems
Prof. Dr. Rony G. Flatscher

23 July 2005
Contents

1 Introduction......................................................................................................................................................... 9
  1.1 About this paper.................................................................................................................................................. 9
  1.2 Research questions.............................................................................................................................................. 10
  1.3 Software automation and code reuse.................................................................................................................. 10
  1.4 Big picture......................................................................................................................................................... 11
2 Technical requirements........................................................................................................................................... 13
  2.1 Overall concept.................................................................................................................................................... 13
  2.2 Subversion........................................................................................................................................................ 14
    2.2.1 General information and architecture....................................................................................................... 14
    2.2.2 Using Subversion........................................................................................................................................... 16
    2.2.3 Problem of file-sharing.................................................................................................................................. 16
  2.3 Java.................................................................................................................................................................... 18
    2.3.1 JavaSVN......................................................................................................................................................... 19
  2.4 ooRexx.............................................................................................................................................................. 20
  2.5 BSF4Rexx.......................................................................................................................................................... 21
  2.6 Apache and Mod_Rexx....................................................................................................................................... 22
    2.6.1 Mod_Rexx..................................................................................................................................................... 22
  2.7 Tomcat............................................................................................................................................................. 23
3 Installation............................................................................................................................................................ 24
  3.1 Installation guide................................................................................................................................................... 24
    3.1.1 Subversion................................................................................................................................................... 24
    3.1.2 Java and JavaSVN......................................................................................................................................... 26
    3.1.3 ooRexx with BSF4Rexx support.................................................................................................................... 27
    3.1.4 Apache with Mod_Rexx................................................................................................................................ 27
    3.1.5 Tomcat......................................................................................................................................................... 29
  3.2 Common pitfalls................................................................................................................................................... 30
4 Concepts of automating Subversion................................................................................................................... 32
  4.1 General information............................................................................................................................................. 32
  4.2 Use of JavaSVN.................................................................................................................................................... 33
  4.3 Java approach..................................................................................................................................................... 34
    4.3.1 Working with workspaces.......................................................................................................................... 34
4.3.2 Working with repositories .................................................. 35
4.4 ooRexx approach ........................................................................ 36
  4.4.1 Loading needed classes ..................................................... 36
  4.4.2 Setting environmental variables ........................................ 37
  4.4.3 Working with workspaces .................................................. 39
  4.4.4 Working with repositories .................................................. 40
5 Developed examples ........................................................................ 41
  5.1 Standalone examples ............................................................... 41
    5.1.1 Example 1 – Simple command-line wrapper ......................... 41
    5.1.2 Example 2 – Variation of the command-line wrapper ............ 44
    5.1.3 Example 3 – Last modification property ............................. 46
    5.1.4 Example 4 – Self defined keywords .................................... 47
    5.1.5 Example 5 – Listing repositories ....................................... 49
    5.1.6 Example 6 – Charting file activity ..................................... 51
    5.1.7 Example 7 – Repository listing standalone server ................. 53
    5.1.8 Example 8 – HTML repository log information ...................... 57
    5.1.9 Example 9 – XML repository log information ....................... 59
    5.1.10 Example 10 – Editing revision properties ......................... 61
    5.1.11 Example 11 – File information ......................................... 63
    5.1.12 Example 12 – Repository information ............................... 64
    5.1.13 Example 13 – Checkout revisions ..................................... 66
    5.1.14 Example 14 – Shut down script ....................................... 67
    5.1.15 Example 15 – Start up script .......................................... 69
    5.1.16 Example 16 – Virus check ............................................. 70
    5.1.17 Example 17 – Working with different repositories ............... 72
  5.2 Hook scripts ............................................................................ 74
    5.2.1 General information .......................................................... 74
    5.2.2 Example 18 – post-revprop-change logging modifications ....... 75
    5.2.3 Example 19 – post-commit sending email ............................. 77
  5.3 Example using Apache with Mod_Rexx ..................................... 80
    5.3.1 Example 20 – Repository listing ........................................ 80
  5.4 Example using Tomcat .............................................................. 84
    5.4.1 Example 21 – Subversion HTTP manager ............................. 84
6 Conclusion........................................................................................................................................89
  6.1 Further work................................................................................................................................89
List of Figures

Figure 1: Overall concept of Subversion automation.......................................................... 13
Figure 2: Subversion's architecture, source [Coll05].......................................................... 15
Figure 3: The file-sharing problem to avoid, source [Coll05]........................................... 17
Figure 4: Architecture of BSF4Rexx, source [Flat05b]...................................................... 21
Figure 5: Process phases of Apache, source [Ashl05a]...................................................... 22
Figure 6: Possible output of example 5............................................................................. 51
Figure 7: Chart showing creation date of files.................................................................. 53
Figure 8: ooRexx standalone server.................................................................................. 56
Figure 9: Repository listing using standalone server......................................................... 57
Figure 10: Detailed log entries example.......................................................................... 59
Figure 11: Generated XML structure out of repository's log information.......................... 61
Figure 12: Example output of script 12_repospository-info.rex........................................ 66
Figure 13: Example output of several revision checkouts............................................... 67
Figure 14: Possible log file for a repository..................................................................... 76
Figure 15: Communication schema of example 20............................................................ 81
Figure 16: Communication schema of Subversion HTTP manager.................................... 85
List of Source codes

Source code 1: A simple Java program................................................................. 18
Source code 2: A simple Object Rexx script.................................................... 20
Source code 3: Working with workspaces (Java).............................................. 35
Source code 4: Working with repositories (Java)............................................. 36
Source code 5: File 99_utils.rex..................................................................... 37
Source code 6: File 00_set-paths-minimalistic.bat........................................ 38
Source code 7: File 00_set-paths.rex................................................................. 39
Source code 8: Working with workspaces (ooRexx)....................................... 40
Source code 9: Working with repositories (ooRexx)....................................... 40
Source code 10: File 01_simple-cmd-line.rex............................................... 43
Source code 11: File 02_simple-cmd-line-variation.rex.................................. 46
Source code 12: File 03_last-modified-property.rex..................................... 47
Source code 13: File 04_self-defined-keywords.rex..................................... 49
Source code 14: File 05_repository-listing.rex............................................... 50
Source code 15: File 06_charting-file-activity.rex........................................ 52
Source code 16: File 07_repository-listing-standalone-server.rex.................. 56
Source code 17: File 08_repository-log-information-html.rex....................... 59
Source code 18: File 09_repository-log-information-xml.rex.......................... 61
Source code 19: File 10_editing-revision-properties.rex.............................. 62
Source code 20: File 11_file-info.rex............................................................... 64
Source code 21: File 12 Repository-info.rex..................................................... 65
Source code 22: File 13_checkout-revisions.rex............................................. 67
Source code 23: File 14_shutdown.rex............................................................... 69
Source code 24: File 15_startup.rex................................................................. 70
Source code 25: File 16_virus-check.rex......................................................... 72
Source code 26: File 17_working-with-different-repositories.rex................... 74
Source code 27: File <path to repository>\hooks\post-revprop-change.bat........ 76
Source code 28: File <path to repository>\hooks\post-revprop-change-rexx.rex... 76
Source code 29: File <path to repository>\hooks\post-commit.bat.................... 77
Source code 30: File <path to repository>\hooks\post-commit-rexx.rex............. 78
Source code 31: File <path to repository>\hooks\post-commit-rexx2.rex............ 80
Source code 32: File `<path to Apache>\htdocs\<any sub-directory>\index.rsp`....................82
Source code 33: File `<path to Apache>\htdocs\<any sub-directory>\server.rex`....................84
Source code 34: File `<path to Tomcat>\webapps\<application name>\subversion\WEB-INF\classes\Update.java`..........................................................87
Source code 35: File `<path to Tomcat>\webapps\<application name>\subversion\WEB-INF\classes\update.rex`..........................................................88
1 Introduction

1.1 About this paper

This bachelor course paper has been written for the last course of the specializing field e-commerce at the Vienna University of Economics and Business Administration. Course lecturer is Prof. Dr. Rony G. Flatscher with assistants. The paper is the second of two bachelor papers which have to be written in the bachelor's program Information Systems.

The document in hand is about the automation of Subversion using Open Object Rexx (ooRexx) as scripting language\(^1\). Chapter 1 gives a brief introduction to the topic along with some general information.

The technical requirements needed for realizing my approach are described in chapter 2. I will give an overview of the version control system Subversion and the programming languages Java and ooRexx. Needed Java libraries, like JavaSVN (for communicating with Subversion) or dom4j (for working with XML\(^2\)) are explained. The use of the Bean Scripting Framework (BSF) is described, especially BSF4Rexx, the BSF for Object Rexx. Furthermore, the two web-servers Apache and Tomcat, which are also involved in my later introduced “nutshell-examples” and their interaction with ooRexx, are explained.

Chapter 3 is a how-to guide for installing all required programs and components. Common pitfalls are depicted as well as their solutions.

Concepts of automating Subversion using Java and ooRexx are described in chapter 4. Differences between the two languages will be shown and their close interaction using BSF4Rexx. After this chapter it should be clear how one can write a Java or ooRexx program which communicates with Subversion. This is necessary for understanding the nut-

---

\(^1\) “A scripting language differentiates itself from other typical languages in that they are usually simpler to learn and use as well as not needing to be compiled. The language is interpreted at run-time so you can execute instructions immediately.” [see Goul05]

\(^2\) “XML (Extensible Markup Language) is a standard for creating markup languages which describe the structure of data. It is not a fixed set of elements like HTML, but rather, it is like SGML (Standard Generalized Markup Language) in that it is a metalanguage, or a language for describing languages. XML enables authors to define their own tags. XML is a formal specification of the World Wide Web Consortium.” [see Tere05]
This chapter shows several examples of useful programs for automating Subversion using ooRexx. Source codes with a detailed description of the scripts are serving as proof of concept. Screen shots of possible program outputs are made available so you get an idea of the results without executing a script yourself.

The last chapter (number 6) gives a conclusion of the work described in this paper as well as an outlook of further work. Some maybe interesting ideas are described whose realization could be worth noting.

1.2 Research questions

I have defined two main research questions, which will be answered in this paper:

- Is it possible to automate Subversion with ooRexx?
- Can the automation be used to generate an advantage in comparison to Subversion's build-in functions?

Question two needs question one to be answered positively.

The first main goal will be to check if Subversion has the capability to be automated by third party components. If it is possible, this will raise the question about the usefulness of such automation. Therefore the second main goal is the creation of helpful programs generating a visible benefit regarding the use of Subversion as versioning system.

Readers who are familiar with the technical details described in chapter 2 on page 13 and who want the first research question to be answered should read chapter 4 starting on page 32. If you are interested in the answer of research question two and want to see some examples, just go to chapter 5 beginning on page 41.

1.3 Software automation and code reuse

Two of the main goals when writing software are automation and code reuse. You will not want to do the same routine over and over again if it is possible to hook up things by writ-
ing an automation script. In this context automation is understood as the ability to perform standardized operations without human interaction. It is just a way to save time and manpower and therefore money.

You also speak of automation if you extend program's build-in functions with your own customized operations. That can be grouping or rearranging already existing functions in a way that makes sense or building completely new ones. Every time you miss some functionality (maybe something the author has not thought about) and implement your own solution for the problem by using third-party tools, you automate the original software.

There is a connection between software automation and code reuse. If you have written a software program (maybe a long time ago) which works fine and you need similar behaviour in another context, it would be great to reuse the already written code of the old program. In other words there is no use of *reinventing the wheel*. Code reuse means that you integrate an already working code into a new program. As already mentioned, this is another way to save time and money.

Object oriented programming languages like Java or ooRexx are making allowance for the paradigm of code reuse by implementing the permission to split up programs to many small pieces. So there are many small software pieces, each representing another functionality which can be implemented as often as needed.

### 1.4 Big picture

Subversion is a version control system for any type of data. Subversion manages files and directories over time. It is designed for several operating systems and handles multiple users, as well as network connections. Therefore it is perfectly fitted to handle projects where many developers work together.

Java is a platform independent, object oriented programming language which has become one of the most used languages worldwide. There are huge libraries for nearly every functionality wanted. For interacting with Subversion there is a library called JavaSVN which I used for generating the examples described in chapter 4 and 5.
The object oriented Version of Rexx – Object Rexx – is a scripting language with a human-oriented syntax which is easy to learn. The Bean Scripting Framework (BSF) is a set of Java classes which provides scripting language support within Java applications. BSF4Rexx is the BSF implementation for Object Rexx. Therefore it is possible to use any Java Object from Object Rexx and even vice versa.

As for now it is only important to know that programs written in Object Rexx can automate Subversion using Java as an interface. A detailed view of the topics described here is given in the next chapter.

In chapter 5 I make use of the very well known HTTP server (so called web-server) Apache. “Apache has been the most popular web server on the Internet since April of 1996.” [see Apac05a] There are many modules which allow software to integrate into Apache – also for Subversion and Object Rexx.

Another web- and application-server⁵ is Tomcat, which acts as a servlet container for Java Servlets and Java Server Pages. There is also an example of a Java Servlet application interacting with Subversion.

All used and described software have something in common: The various programs are distributed under any well known open source license, therefore free of charge and can be extended by anybody.

---

³ “An application server is a server program in a computer within a distributed network that provides the business logic for an application program. The application server is frequently viewed as part of a three-tier application, consisting of a graphical user interface (GUI) server, an application (business logic) server, and a database and transaction server.” [see Lany05]
2 Technical requirements

In this chapter I am going to give a short overview of the used software programs, programming languages and their components. It is important to say that this section cannot be understood as a tutorial for a programming language or a handbook/reference for any software described. In every sub-chapter there will be references to further material dealing with the topic. If you are not familiar with any techniques, you may have a look at the references mentioned.

If you want to try out anything of the software described in this chapter, I recommend reading chapter 3 starting on page 24, which deals with the installation of the components and the problems likely to arise during installation.

2.1 Overall concept

Figure 1 below shows the main interaction between used components for automating Subversion.

Subversion libraries → Python, Perl, Java etc. → BSF4Rexx → ooRexx

Figure 1: Overall concept of Subversion automation

“Subversion has a modular design, implemented as a collection of C libraries. Each library has a well-defined purpose and interface, and most modules are said to exist in one of three main layers – the Repository Layer, the Repository Access (RA) Layer, or the Client Layer.” [see Coll05, 143]

If you are interested in using the Subversion libraries in conjunction with something different than a C program – say a Python script or a Java application – Subversion has some initial support for this via the Simplified Wrapper and Interface Generator (SWIG). These bindings allow you to call Subversion API functions indirectly, using wrappers that trans-
late the data types native to your scripting language into the data types needed by Subversion's C libraries.

There is an obvious benefit of accessing the Subversion APIs via a language binding – simplicity. Generally speaking, languages such as Python and Perl (and ooRexx) are much more flexible and easy to use than C or C++. The sort of high-level data types and context-driven type checking provided by these languages are often better at handling information that comes from users. As you know, humans are proficient at botching up input to a program, and scripting languages tend to handle that misinformation more gracefully. Of course, often that flexibility comes at the cost of performance. That is why using a tightly-optimized, C-based interface and library suite combined with a powerful, flexible binding language, is so appealing. [cp. Coll05, 152]

With the use of BSF4Rexx any Java application can use (Object) Rexx as a scripting language. Hence, (Object) Rexx can use Java as a huge operating system independent function library. The Object Rexx support enabled with BSF.CLS makes Java look like a huge Object Rexx class library. [cp. Flat05a].

2.2 Subversion

2.2.1 General information and architecture

Subversion is a free/open-source version control system. That is, Subversion manages files and directories over time. A tree of files is placed into a central repository. The repository is much like an ordinary file server, except that it remembers every change ever made to your files and directories. This allows you to recover older versions of your data or examine the history of how your data changed.

Subversion can access its repository across networks, which allows it to be used by people on different computers. At some level, the ability for various people to modify and manage the same set of data from their respective locations fosters collaboration. Progress can occur more quickly without a single channel through which all modifications must occur. And because the work is versioned, you need not fear that quality is the trade-off for losing that
conduit – if some incorrect change is made to the data, just undo that change. [cp. Coll05, 1]

Figure 2 illustrates Subversion's architecture as a diagram. If you obtain Subversion, you have to handle everything through a command-line because there is no such thing like a Graphical User Interface (GUI). But as mentioned before you can easily write your own software – for example a GUI – which was done many times in the past (just have a look at [Coll05, 274 ff]).

On the – let us say – server side there is the Subversion repository – a container for all files for a specific project. The files can be stored in a Berkeley DB or as FSFS\(^5\). The repository can be accessed over the local file system if repository and client are on the same computer

---

5 FSFS (Fairly Secure File System) is a versioned file system implementation that uses the native operating system file system to store data (ordinary flat files using a custom format).
or over a network. There are two protocols for which Subversion has standard implementations: DAV and SVN. The SVN (for Subversion) protocol communicates with Subversion's own server svnserv, “a lightweight server, capable of speaking to clients over TCP/IP using a custom, stateful protocol” [see Coll05, 96]. The WebDAV/DeltaV protocol is an extension to HTTP 1.1. Via a custom module Apache's web-server makes Subversion repositories available to clients.

2.2.2 Using Subversion

Along with starting a new project an administrator has to create a repository and eventually has to import already existing files. Not everybody who knows of the address of the repository should have read and/or write privileges, so there have to be set up user authentications. Using the SVN protocol the administrator has to set up read and write rights through svnserve. With the integration in Apache user rights can be managed using Apache's httpd.conf and/or .htaccess files.

At first every client has to do a checkout once. This means to transfer all files in the repository to a local copy. After doing that users can add, move, delete, edit etc. files – whatever they want to do with them. To copy the modified files back to the repository a commit is needed along with a commit message describing changes made. With each commit a new revision is created (starting at 0; new revision = old revision + 1). If the user wants to edit the files at a later point in time, he has to do an update which copies only needed files (=modified files since his workspace's revision) into his local workspace. The procedure is the same for every user working with the repository.

As Subversion is a version control system, everybody has the ability to revert changes. Every modification made is logged and can be reverted at any time. By means of Subversion it is also comprehensible which files and which content of files where altered at which point in time and by whom.

2.2.3 Problem of file-sharing

All version control systems have to solve the same fundamental problem: how will the sys-

---

6 You can even write your own protocol and make it available through Subversion.
tem allow users to share information, but prevent them from accidentally stepping on each other's feet? It is all too easy for users to accidentally overwrite each other's changes in the repository.

Consider the scenario shown in figure 3: Suppose we have two co-workers, Harry and Sally. They each decide to edit the same repository file at the same time. If Harry saves his changes to the repository first, it is possible that (a few moments later) Sally could accidentally overwrite them with her own new version of the file. While Harry's version of the file will not be lost forever (because the system remembers every change), any changes Harry made will not be present in Sally's newer version of the file because she never saw Harry's changes to begin with. Harry's work is still effectively lost – or at least missing from the latest version of the file. [cp. Coll05, 9]

Subversion, like other version control systems (e.g. CVS⁷), use a copy-modify-merge model. In this model every user creates a personal working copy – local reflection of the repository's files and directories. Users then work in parallel, modifying their private copies. Finally the private copies are merged together into a new, final version. The version control system often assists with the merging, but ultimately a human being is responsible for making it happen correctly. [cp. Coll05, 11]

---

⁷ “Concurrent Versioning System. CVS is an open source version control and collaboration system.” [see Naud05]
The reason why I wanted to explain briefly the problem of file-sharing is not only because it is a very fundamental one. It is also because some of my examples and ideas described in later chapters have the goal to provide information about a repository and its status to avoid problems of file-sharing and loss of data.

Subversion itself and lots of information with further references and links can be found at [Fitz05]. If your are looking for a good introduction to Subversion, I suggest reading [Coll05].

### 2.3 Java

Java is an object-oriented programming language primarily developed by James Gosling and colleagues at Sun Microsystems. The language, initially called Oak (named after the oak trees outside Gosling's office), was intended to replace C++, although the feature set rather resembles that of Objective C. Java should not be confused with JavaScript, which shares only the name and a similar C-like syntax. Sun Microsystems currently maintains and updates Java regularly. [cp. Wiki05a]

Java is platform independent and has Internet-ability – summarized in the slogan: *Write once – run everywhere.* From the source code the Java-Compiler generates a so-called byte code which is independent of the operating system used and can be executed, wherever a Java Virtual Machine (JVM) is installed. Today a JVM is available for every well known operating system. [cp. Abts04, 3 f] and [cp. Ulle05, 52]

Writing programs for the Internet means developing Java Servlets or Java Server Pages (JSP). Servlets are the basis for JSPs – at runtime every JSP is translated into a Servlet.

To have an idea what the simplest Java program looks like see source code 1 below:

```
class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World");
    }
}
```

Source code 1: A simple Java program
If you want to save these five lines to a file, it has to be named after the class – Hello-
World.java. Then you have to compile the code making use of javac HelloWorld.java
at the command-line. Execution of the program is done by typing java HelloWorld. If
everything worked fine, then there must be the output Hello World at the command-line.

Java can be obtained from [SunM05a]. If you are searching for tutorials, books, links, user
groups, forums etc., Sun's website is the place to go to. An absolute must for Java program-
mers is the Java API at [SunM05b] (latest version 5). For a quick overview and a lot of
links I recommend [Wiki05a]. Books which (I think) can be useful are [Abts04] and
[Ulle05] (both in German). If you are new to Java and want to have a look at some code
samples, go to [SunM05c].

2.3.1 JavaSVN

JavaSVN is a pure Java Subversion client library (see also figure 2 on page 15). This means
that users of the library, e.g. Java applications do not have to include Subversion native bin-
aries or javahl bindings\(^8\) to work with Subversion repositories. JavaSVN library is not only
a 100% Java replacement for javahl bindings, but also a library that provides high level of
control over Subversion repository operations.

Major features of JavaSVN:
- No external binaries or libraries are needed.
- Supports HTTP, HTTPS\(^9\), SVN and SVN+SSH\(^10\) connection protocols.
- Default implementation provides support for default Subversion working copy files.
- Low level API allows to work directly with repository, without file system overhead.
- Extensible design – every part of implementation could be extended or replaced.
- May be used as a transparent javahl replacement. [cp. Tmat05a]

JavaSVN can be obtained from [Tmat05a]. The JavaDoc\(^{11}\) is located here [Tmat05b] and a
couple of example programs are published at the following address [Tmat05c].

---

8 Javahl is the language binding library (like JavaSVN) contributed from the same authors as Subversion.
9 Secure HTTP – Protocol enabling the secured transmission of web pages.
10 Secure Shell - A command-line interface used to securely access a remote computer.
11 Java documentation of classes, interfaces and their methods which are made available (=API).
2.4 ooRexx

Rexx (Restructured Extended Executor) is a programming language which was developed at IBM, and several implementations are available under open source licenses. It is a structured high-level programming language which was designed to be both easy to learn and easy to read. Both commercial and open source Interpreters for Rexx are available on a wide range of computing platforms. [cp. Wiki05b]

Object Rexx is the object-oriented version of Rexx developed in the early 90s. Object Rexx is fully compatible to Rexx, but has a complete internal object oriented structure and a powerful object model (meta-classes, multiple inheritance). [cp. Flat05b, 8]

“Open Object Rexx (ooRexx) is an Open Source project managed by Rexx Language Association (RexxLA) providing a free implementation of Object Rexx.” [see Rexx05]

Source code 2 shows the Object Rexx variation of the same example as it was realized in Java in source code 1.

```
1 say "Hello World"
```

Source code 2: A simple Object Rexx script

Saved to hello_world.rex the script can be executed using the command-line rexx hello_world.rex. If you are using Microsoft's Windows, you can alternatively run the script by doing a double click on the specific file in the Windows Explorer12.

Downloading ooRexx can be done from the official RexxLA website [Rexx05]. IBM's programming guide and reference to Object Rexx can be found on [Ibm01a] and on[Ibm01b]. The course slides by Prof. Dr. Rony G. Flatscher (in German) provide a good introduction to Object Rexx, as well as a tutorial and sample scripts for automating Windows and Java applications (download available on [Flat05b] and on [Flat05c]).

---

12 Attention: Execution of scripts shown in this chapter requires needed software to be installed (see chapter 3).
2.5 BSF4Rexx

The Bean Scripting Framework (BSF) was an open source project developed at IBM. In 2003 the code was handed out to Apache's Jakarta project. BSF is a framework that allows Java to execute scripting languages very easily. Defined interfaces give scripting languages the possibility to interact with Java objects.

BSF4Rexx is the Bean Scripting Framework for Object Rexx. Developed by Prof. Dr. Rony G. Flatscher and a former student of his, Peter Kalender, in 2000/01. Up to now there have been three major versions: the Essener, the Augsburger and the Wiener version\(^\text{13}\). The current Wiener version is in an ongoing development status.

Figure 4 demonstrates the architecture of the latest BSF4Rexx (Wiener version).

---
\(^{13}\) Version names are the cities where Prof. Dr. Rony G. Flatscher worked as a university professor and at that point in time developed the version.

Further information on BSF can be found on [Apac05b]. Any material concerning BSF4Rexx is available at [Flat05a].
2.6 Apache and Mod_Rexx

The Apache HTTP Server Project is an attempt to develop and maintain an open-source HTTP server\textsuperscript{14} for modern operating systems including UNIX and Windows NT. The goal of this project is to provide a secure, efficient and extensible server which provides HTTP services that match current HTTP standards. [cp. Apac05a]

The Apache HTTP Server Project is a collaborative effort aimed at creating a robust, commercial-grade and freely-available source code implementation of an HTTP (web) server. The project is jointly managed by a group of volunteers located around the world, using the Internet and the Web to communicate, plan, and develop the server and its related documentation. This project is part of the Apache Software Foundation. [cp. Apac05c]

Apache is commonly used in combination with Linux or Windows as an operating system, MySQL as the database system and PHP as the scripting language\textsuperscript{15}.

Apache's official site is accessible at [Apac05a].

2.6.1 Mod_Rexx

Mod_Rexx is an Apache loadable module for integrating Object Rexx into Apache's web-server. All phases of an Apache request can be processed with Mod_Rexx (see figure 5). That means that each of these phases can have an Object Rexx script assigned to it. In most cases you will only assign a script to the response phase (when data is sent back to the client).

HTTP requests are handled through Apache's web-server. If a client requests an Object Rexx script (noticeable, for example through the file extension .rex), the Object Rexx script is executed (if Apache's \texttt{httpd.conf} is configured

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.png}
\caption{Process phases of Apache, source [Ashl05a]}
\end{figure}

\textsuperscript{14} If you know nothing about TCP/IP networks, the Domain Name System (DNS), HTTP-client-server communication etc., you may read the very short introduction at [Hois05, 27 ff] (in German) or do an Internet search on these topics – there are many good resources available.

\textsuperscript{15} LAMP – Linux, Apache, MySQL, PHP or WAMP – Windows, Apache, MySQL, PHP
correctly, see chapter 3) and the returned data is sent to the client. With Mod_Rexx it is also possible to write Rexx Server Pages (RSP)\(^{16}\), which means writing Object Rexx code inside a HTML\(^{17}\) file (like JSP or PHP files)\(^{18}\). This allows HTML pages to be created dynamically at runtime.

The version of Mod_Rexx used works only with Apache 2.

The Homepage of Mod_Rexx containing all information is available on [Ashl05b] – download is possible at SourceForge.net [Sour05].

### 2.7 Tomcat

As described before Tomcat (in fact it is called Apache Tomcat) “is the servlet container that is used for the official Reference Implementation of the Java Servlets and JSPs technology” [see Apac05d]. That means writing Java programs which are called through an HTTP request and executed with Tomcat. Generated HTML code is sent back to the client (probably a web-browser).

[Apac05d] comprises a link to Tomcat's official website.

\(^{16}\) An allusion to Java Server Pages.

\(^{17}\) HyperText Markup Language – The document format language used on the World Wide Web.

\(^{18}\) An example is given in chapter 5.
3 Installation

The following chapter contains an installation guide for the software described in chapter 2. As I used Microsoft Windows for developing my examples, this guide is written for that operating system. If you are using another operating system, the installation will certainly more or less differ from this guide, but correspond in principle.

This chapter assumes that you have already downloaded the software to install. For links and information concerning the download of needed components see the specific sub-chapters in chapter 2.

3.1 Installation guide

3.1.1 Subversion

Installation of Subversion should be straight forward. After execution of the Windows Installer program you should have installed Subversion to a directory like

C:\Program Files\Subversion

For creating a new repository open a command-line shell by clicking on the left sided button Start and then Execute. In the text field enter cmd and then press Ok. A Windows command-line window will appear. Enter the following:

svnadmin create <path to repository>

svnadmin is Subversion's administration program for creating, dumping, recovering etc. repositories. The command create creates a new repository under the path specified in <path to repository> (e.g. C:\_repository which I used in my examples).

Now that you have created a new repository at the path declared at <path to repository> let us have a closer look at the directory structure:

- <path to repository>\conf\ – Containing repository configuration files.
- <path to repository>\dav\ – Provided to Apache and mod_dav_svn\(^{19}\) for the private housekeeping data.

\(^{19}\) Subversion's Apache module for authorization. Subversion users are authorised by Apache's user configuration files.
• `<path to repository>\db\` – Where all the versioned data resides (stored either as Berkeley DB or FSFS).

• `<path to repository>\hooks\` – Directory for hook scripts\(^20\). Important to know because there are examples of hook scripts in chapter 5.

• `<path to repository>\locks\` – For locking data, used for tracking accessors to the repository.

I suggest creating another directory as the workspace – the data to work with. In my examples I use `C:\_checkouts`. The sub-directories are the different working copies of repositories.

Once changed to `<path to workspace>` (e.g. by typing `cd <path to workspace>`) you can do an initial checkout by typing

```
svn checkout <url to repository>
```

`<url to repository>` can be accessed locally (with `file:///<path to repository>`\(^21\)), via the HTTP protocol (with `http://<url to repository>`\(^22\)) or via the SVN protocol (with `svn://<url to repository>`\(^23\)).

If the repository is accessed over HTTP or SVN protocol, `<url to repository>` must be either the IP address or the full qualified domain name (FQDN) followed by the local path to the repository.

If the checkout was successful, there is a message like `Checked out, Revision 0`. Now your repository and workspace are set up and you are free to work with Subversion, which is best described in [Coll05, 19 ff].

For using the `svn` command\(^24\) not only in Subversion's binary directory (where `svn.exe` resides) but also in any other directory, Subversion on its own sets following environmental variable:

```
PATH=%PATH%;<path to Subversion>\bin
```

---

\(^20\) Hook scripts are programs triggered by some repository event, such as the creation of a new revision or the modification of an unversioned property.

\(^21\) Notice the forward slashes, e.g. `file:///c:/_repository`

\(^22\) Needs Apache to be set up and running.

\(^23\) Needs `svnserve` to be set up and running.

\(^24\) And `svnadmin` and `svnlook` and `svnserve` and so on...
Command `%PATH%` inserts the value of the environmental variable `PATH` itself, so that Sub-
version's path to its binary directory is appended.

You can have a look at all environmental variables in the system control centre or by typing
the following at the command-line:

```
echo %<environmental variable>%(e.g. echo %PATH%)
```

Setting of environmental variables is done either in the system control centre or by typing:

```
set <environmental variable>=<value>
```

Latest Subversion version for Windows is 1.2.0, which I used for my examples.

### 3.1.2 Java and JavaSVN

Install, for example, the Java 2 Standard Edition (J2SE) Development Kit to a directory
similar to C:\Program Files\j2sdk.

Make sure that the following environmental variables are set:

```
PATH=%PATH%;<path to Java>\bin
PATH=%PATH%;<path to Java>\jre\bin\client
```

Test your installation by typing `java -version`, which prints out used Java version.

After that install JavaSVN to any directory you want, for example C:\Program
Files\javasvn.

You have to set the following environmental variable to make JavaSVN work:

```
set CLASSPATH=%CLASSPATH%;<path to JavaSVN>\javasvn.jar
```

For my examples I used Java in version 1.4.2_01 and JavaSVN in version 0.8.8.1.

---

25 You have to set environmental variables to be able to access resources located elsewhere than in the
current directory. The resources are made available system wide.
3.1.3 ooRexx with BSF4Rexx support

Open Object Rexx installation is as easy as installing Subversion or Java – just follow the instructions on the screen. After exiting the installation wizard you should have installed ooRexx to a directory like C:\Program Files\ooRexx.

The environmental variables REXX_HOME and PATH are set to <path to ooRexx>.

Test your ooRexx installation by typing rexx -v at the command-line prompt to see ooRexx' version number and license.

After finishing the installation of ooRexx unzip the downloaded bsf4rexx.zip to a directory of your choice, e.g. C:\Program Files\bsf4rexx.

You have to set the following environmental variables:

```
set CLASSPATH=%CLASSPATH%;<path to BSF4Rexx>
set PATH=%PATH%;<path to BSF4Rexx>\bin
```

The used ooRexx version for my examples is 3.0.0 in connection with BSF4Rexx in the version of 2005-07-09.

3.1.4 Apache with Mod_Rexx

You only have to install Apache with Mod_Rexx if you want to try out examples requiring this web-server and the module (see chapter 5.3).

Installing Apache 2 (And only Apache 2 if you want to use Mod_Rexx!), for example, with the Windows Installer to a directory similar to C:\Program Files\Apache Group\Apache226.

Test your installation by starting Apache's web-server and typing http://localhost in the address field of your preferred web-browser. You should see a confirmation page that the web-server has been installed successfully.

---

26 Generally speaking it is better to use directories not containing white spaces.
In Apache's home directory you will find a sub-directory called `conf` – the central configuration directory. If you want to configure Apache to work with Subversion, edit the file `httpd.conf` and do the following:

If necessary uncomment the following line:

```
LoadModule dav_module modules/mod_dav.so
```

Insert this line in the `LoadModule` section:

```
LoadModule dav_svn_module <path to Subversion>\bin\mod_dav_svn.so
```

This will load the needed Subversion support module when Apache starts.

The only thing left is to tell Apache where to find repositories. This is done by inserting the following lines:

```
<Location /<name of repository>>
   DAV svn
   SVNPath <path to repository>
</Location>
```

You can access your repository by typing `http://<url to host>/<name of repository>` in the address field of your web-browser.

If you need authorization, the following lines may be interesting:

```
<Location /<name of repository>>
   DAV svn
   SVNPath <path to repository>
   AuthType Basic
   AuthName <name of repository>
   AuthUserFile <path to .htpasswd>
   require valid-user
</Location>
```

After an Apache restart your repository is accessible via HTTP. For a more detailed instruction read [Coll05, 102 ff].

---

27 Or you copy the `mod_dav_svn.so` to the directory `<path to Apache>\modules` where all modules are located.

28 Look at [Coll05, 105] for information about Apache user authorization.
If you want to install Mod_Rexx, do the following after downloading it:

Copy <path to Mod_Rexx>\bin\mod_rexx.dll to <path to Apache>\modules
Copy <path to Mod_Rexx>\rspcomp\rspcomp.rex to <path to Apache>\bin
Copy <path to Mod_Rexx>\rexscripts\Apache.cls to <path to Apache>

Edit httpd.conf again and add the following:

LoadModule rexx_module modules/mod_rexx.dll

In the AddType section add:

AddType application/x-httpd-rexx-script .rex .rexx^{29}
AddType application/x-httpd-rexx-rsp .rsp^{30}

and

RexxTempFileNameTemplate <path to any temp directory>execrsp?????.rex
RexxRspCompiler <path to Apache>\bin\rspcomp.rex

As Mod_Rexx transfers every RSP into an Object Rexx Script at runtime^{31}, the last two lines specify (1) a temporary Object Rexx file to write the transformed code to and (2) the executed transformation script.

At last restart Apache again – now Object Rexx support is enabled. For further information have a look at [Ashl05b].

I am using Apache 2.0.54 with Mod_Rexx 2.1.0.

### 3.1.5 Tomcat

You only have to install Tomcat if you want to try out examples requiring this application-server (see chapter 5.4).

Install Tomcat in a directory like C:\Program Files\Tomcat.

---

^{29} Tells Apache to forward files with specified extension to Object Rexx for execution.
^{30} HTML files with Object Rexx code inside must have the extension specified here. That is the only way for Apache to recognize that files not only have HTML but also Object Rexx code inside (which must be executed).
^{31} Think of JSPs translated to Java Servlets.
Be sure that you configure Tomcat running on a different port than Apache. Only one of the two web-servers can be invoked to handle a request made on a specific port. The value of the standard HTTP port is 80. I configured my system as follows: Apache on port 80 and Tomcat on port 8080\textsuperscript{32}. Configuration can be made using Apache's \texttt{httpd.conf} and Tomcat's \texttt{server.xml}.

Test your installation by browsing to \url{http://localhost:8080/}\textsuperscript{33}. You should see Tomcat's standardized welcome page.

I am using Tomcat's version 5.0.19 (scripts were also tested on Tomcat 5.5.9).

\section*{3.2 Common pitfalls}

Here I am going to give some hints on common pitfalls during installation and configuration.

\begin{itemize}
\item Be sure to download the latest software release available.
\item Check if all paths described in chapter 3.1 are set correctly. This is one of the main problems because you must not have any typing errors.
\item If a directory contains any spaces, they must be between two quotation marks. Usually it does not matter if you use `"` or `."
\item Subversion URLs accept only forward slashes. For example only \texttt{file:///c:/_repository} is valid.
\item On windows machines Subversion paths can be written with both forward and backward slashes. Just pay attention if you mix them up.
\item If you set environmental variables on the command-line prompt (with \texttt{set <name>=<value>}), these settings only apply to the window opened and only last as long as the window stays open. If you want to modify the variables for every session, you have to set them in the system control centre.
\item Every \texttt{.jar} file is a compressed container that comprises Java classes. If you do not want to set any environmental variables for accessing these classes, just unpack them in the same folder as your script is located.
\end{itemize}

\textsuperscript{32} Be sure to specify a port greater than 1023 because these well known ports are reserved for other usage.
\textsuperscript{33} Or whatever you set the port to.
• You cannot set Apache's directory index to any RSP. You have to do a workaround like putting an index.html file forwarding to the RSP in the same directory.

• You cannot include BSF4Rexx' BSF.CLS into an Object Rexx Script or RSP\(^34\) and let it execute through an Apache process. If anyone manages this, please let me know.

• If you are using Tomcat 5.5.x with Java 1.4.x, you have to download a compatibility package from the following address http://apache.mirror.netmonic.com/jakarta/tomcat-5/v5.5.9/bin/jakarta-tomcat-5.5.9-compat.zip.

• You have to ensure that Tomcat's process has access to BSF4Rexx classes. The simplest way is to copy the `<path to BSF4Rexx>\org directory to `<path to Tomcat>\webapps\<application name>\WEB-INF\classes`.

• Also JavaSVN libraries must be accessible. You have to copy `<path to JavaSVN>\javasvn.jar` either to `<path to Tomcat>\common\lib` or to `<path to Tomcat>\webapps\<application name>\WEB-INF\lib` depending if you want to grant access to the JavaSVN classes for all web-applications or only a specific one.

\(^34\) And therefore you cannot use any provided BSF4Rexx functions.
4 Concepts of automating Subversion

This chapter deals with concepts of automating Subversion. Where are the weak points and how can they be improved? Furthermore, there is a description of the Java library JavaSVN, which makes automating Subversion possible. The last two sub-chapters are dealing with the basic Java and ooRexx implementation concepts of Subversion automation.

4.1 General information

By means of Subversion one can version control any sort of data. You do not even have to share files to benefit from using Subversion. If you work on any written document (an article, a book etc.), source code, images and so on, it could be helpful to version any changes made. You for yourself can realize that you have done something wrong and then you will praise Subversion to be able to revert easily to a former version. Subversion is also an excellent tool for backing up data (if the repository is located at another computer and accessed over a network). At any time you can retrieve any revision ever made. Subversion only stores the differences between revisions, so it uses the smallest amount of disc space.

When people work together, mistakes can occur very easily and following Murphy's law mistakes certainly will occur. The use of Subversion cannot prevent errors to happen, but they are so much easier to solve. Subversion generates a benefit when many people work on a project sharing any data. With a central repository any person has access to the latest revision. There is no need to send modified documents to every person involved in the project by email. This circumstance reduces network traffic as well as Subversion's ability to update only needed data on the client's workspace.

As good as Subversion is at managing data and revisions of data over time, as bad it is at managing collaboration between members in a project – it is just not designed for that. Working only with the Subversion standard command-line interface can easily lead to a lack of information. That is because the command-line is limited to show only very small text information with no accentuation available. It is not a good idea to display such a great amount of information – as it is the case in Subversion – on such a limited medium like the command-line. For sure, when more than twenty, fifty or hundred people are working on a
project at the same time, there is certainly so much information that no one has a clear overview anymore.

Another negative aspect is that nobody knows who is currently working on a specific file. By means of Subversion you can see who has modified what and you also have a log message with the user's comments, but you have no chance to see modifications right on time. So if anybody forget to commit their changes, nobody will ever know. Then it can happen that another user modifies the same (work was done twice) or that an idea has already been discarded (work was in vain). So it would be a benefit if there was a central contact point where current information about modifications done by users is collected.

Some of my examples are targeting at these negative behavioural patterns of Subversion by generating clear reports, better summaries and user notifications. I will also introduce an idea for a better solution for collaborative work. Further examples will show how to improve functions by a new behaviour, which I think can be useful. Furthermore, there are implementations of missing functionality, which can be advantageous as well.

4.2 Use of JavaSVN

The main packages used from JavaSVN are org.tmatesoft.svn.core (for working with workspaces) and org.tmatesoft.svn.core.io (for working with repositories) [cp. Tmat05b]. That separation allows the user to work only with a repository or only with workspaces, for example.

In the org.tmatesoft.svn.core.io package the main class to work with is called SVNRepository. It declares all the basic interface methods as well as implements commonly used ones to work with a Subversion repository. It is the skeleton of the low-level mechanism of accessing a repository. In the model of the Subversion distributed system of versioning and sharing data, this mechanism corresponds to the Repository Access (RA) Layer. This low-level API itself knows nothing about working copies.

The high-level library API rests upon this basis: for example, manipulations with a working copy (which need an access to a repository), say, committing it, uses an appropriate imple-
mentation (depending on the protocol that is chosen to access the repository) of SVNRepos-
itory as an engine that carries out the commit itself. [cp. Tmat05b]

In the org.tmatesoft.svn.core package there is an interface called ISVNWorkspace
providing more or less all methods for working with a workspace. This package relies on
the low-level API org.tmatesoft.svn.core.io when an access to a repository is needed.

4.3 Java approach

My first approach to automating Subversion was using Java. For those who are familiar
with the language it will be easier to understand the ooRexx examples if the basic concepts
are also shown in Java. Exactly the same examples will be shown in chapter 4.4 using
ooRexx.

4.3.1 Working with workspaces

Source code 3 shows the Java setup for working with Subversion workspaces. Assuming
that you have already checked out a revision to a workspace defined in path, this program
prints out the corresponding URL of the repository. There is no need to access the reposit-
ory. Used packages for working with workspaces are imported in line 1 and 2. With the
called method in line 10 the working copy storage is initialized and configured (default is
file system). Then a new workspace is created using the SVNWorkspaceManager.create-
Workspace method. Using file type to create workspace, results in creating workspaces
that use file system to store working copy files and administrative info. The only implemen-
ted and therefore possible type is file.workspace.getLocation() returns a SVNReposit-
oryLocation object which is printed out. Code for accessing workspaces must be
between a try-catch block or declared to be thrown.

```
1 import org.tmatesoft.svn.core.internal.ws.fs.*;
2 import org.tmatesoft.svn.core.*;
```

35 If you have a look at line 7 where variable path is defined, you will notice forward slashes in place of
backward slashes. The reason is that Java notices a backward slash as an escape character which itself must
be escaped. Therefore I use forward slashes to bypass that problem. Windows does not care about using
forward or backward slashes.

36 There is no need to apply the toString() method to the SVNRepositoryLocation object because
System.out.println() does that for us.
public class Workspace {
    public static void main(String[] args) {
        String path = "C:/_checkouts/repos";
        try {
            FSEntryFactory.setup();
            ISVNWorkspace workspace = SVNWorkspaceManager.createWorkspace("file", path);
            System.out.println(workspace.getLocation());
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}

Source code 3: Working with workspaces (Java)

Output will be something like http://localhost/repos.

4.3.2 Working with repositories

You can see by means of source code 4 a sample implementation of accessing a repository over HTTP and printing out repository's latest revision. If you want to use SVN as protocol, you have to replace the first line with import org.tmatesoft.svn.core.internal.io.svn.*; for importing the needed package. Also line 10 must be substituted for SVNRepositoryFactoryImpl.setup(); which sets up connection protocol support. To get a repository instance having a repository URL is shown on lines 11 and 12. The behaviour of method getLatestRevision() should be obvious – the result is printed out. As in source code 3, code for accessing repositories must be between a try-catch block or declared to be thrown.

import org.tmatesoft.svn.core.internal.io.dav.*;
import org.tmatesoft.svn.core.io.*;
public class Repository {
    public static void main(String[] args) {
        String url = "http://localhost/repos";
        try {
            DAVRepositoryFactory.setup();
            SVNRepositoryLocation location = SVNRepositoryLocation.parseURL(url);
            SVNRepository repository = SVNRepositoryFactory.create(location);
            System.out.println(repository.getLatestRevision());
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
4.4 ooRexx approach

All examples shown in chapter 5 are written in ooRexx. At this point I will display the differences between Java and ooRexx on the one hand and demonstrate the basic syntax for working with Subversion workspaces and repositories using ooRexx on the other hand. Compared to Java you will see that you need less ooRexx code for the same functionality. As ooRexx is a scripting language you also do not have to compile the written source code. Variables used in ooRexx need not be declared – there is no strict typing. These all are reasons for using ooRexx and not Java as programming language in my examples.

4.4.1 Loading needed classes

As I use Subversion classes in every example there is a routine\(^{37}\) (starting at line 13 in source code 5) which loads all needed classes. In my examples in chapter 5 I import the file 99_utils.rex and call the defined routine loadClasses to load the Java classes. Therefore I do not have to copy the import function to every single of my example files. For some examples I also need dom4j (for working with XML files) and jfee.chart (for generating graphical charts) packages and some Java specific classes.

In source code 5 a variable classes is defined and filled up with classes to load (line 14).Delimiter between the text strings is a single blank character. The comma treats the following line as if all was written in one single line. After that there is a loop over the number of words (returned by words(classes)) in variable classes (line 49 to 53). At each pass the variable class contains the word (=package) at position i. The variable rexxClass is filled up with only those characters found after the last “.”, e.g. at the first cycle rexxClass has the value DAVRepositoryFactory. The last thing to do is to import the Java classes using BSF4Rexx (BSF4Rexx support is loaded in line 10 using the requires directive). The Java class is now accessible through the name defined in rexxClass.

---

\(^{37}\) The routine is declared public to be accessible from everywhere.
4.4.2 Setting environmental variables

As described in chapter 3 you have to set a lot of environmental variables for the installed software. Some of them are set automatically along with the installation, some you have to set on your own. Source code 6 shows an executable BAT-file\(^{38}\) which adapts my environmental variable `CLASSPATH` according to my system settings\(^ {39}\). By executing the file variable

\(^{38}\) A batch file for DOS.

\(^{39}\) Caution: Not every environmental variable needed by the installed software described in chapter 3 is set by the batch file, only those required according to my system configuration.
CLASSPATH is set to Java archives (.jar-files) from JavaSVN, JfreeChart\(^\text{40}\) and dom4j\(^\text{41}\). If you want to use the script, just adopt the paths to your system settings.

If you have not set environmental paths (or not all of them) in the system control centre, you can use this script by running it every time you open a new command-line window. The variables are available as long as the window stays open. You can easily extend the script by setting more variables you need. It is also possible to call batch files from ooRexx scripts using the address cmd directive. Therefore you can set environmental variables every time you execute an ooRexx script needing those paths to be set.

```
rem 00_set-paths-minimalistic.bat, 2005-06-12
rem Adapts CLASSPATH according to my environmental settings.
set CLASSPATH=%CLASSPATH%;C:\Dokumente und Einstellungen\Berni\Eigene Dateien\Temp\subversion\javasvn-0.8.8.1\javasvn.jar
set CLASSPATH=%CLASSPATH%;C:\Dokumente und Einstellungen\Berni\Eigene Dateien\Temp\subversion\jfreechart-1.0.0-rc1\lib\jfreechart-1.0.0-rc1.jar
set CLASSPATH=%CLASSPATH%;C:\Dokumente und Einstellungen\Berni\Eigene Dateien\Temp\subversion\dom4j-1.6.1\dom4j-1.6.1.jar
set CLASSPATH=%CLASSPATH%;C:\Dokumente und Einstellungen\Berni\Eigene Dateien\Temp\subversion\dom4j-1.6.1\lib\jaxen-1.1-beta-6.jar
```

Source code 6: File 00_set-paths-minimalistic.bat

Source code 7 below is an ooRexx script which sets more or less every environmental variable needed for executing my examples shown in the next chapter\(^\text{42}\). Therefore a new variable paths of type directory\(^\text{43}\) is created (line 14) and filled up with the name of environmental variables as keys and the corresponding paths as values. Lines 41 to 45 loop over the variable and execute the DOS command for setting environmental variables (using address cmd). The next line prints out the directly set variable (DOS command echo) and a blank row (statement say without a value).

It is possible to load this script using the requires directive to set environmental variables.

\(^{40}\) JfreeChart is a free Java class library for generating charts and can be obtained from [Gilb05].

\(^{41}\) “dom4j is an easy to use, open source library for working with XML […] on the Java platform” [see Meta05] and can be downloaded from [Meta05].

\(^{42}\) Here environmental variables are available as long as the execution of the script lasts.

\(^{43}\) “A directory is a collection with unique indexes that are character strings representing names.” [see Ibm01b, 127]
before execution of other parts of an ooRexx program is taking place. Hook scripts (described in chapter 5.2) are executed by the system where no environmental variables are set and therefore need the file $00_set-paths.rex$ to be loaded first.

Source code 7: File $00_set-paths.rex$

4.4.3 Working with workspaces

Source code 8 does exactly the same as source code 3 described earlier in chapter 4.3.1, but this time it is implemented in ooRexx.

On line 10 and 11 BSF4Rexx support and the file $99_utils.rex$ (whose method loadClasses is called in the first line of the program) are loaded. Lines 3 to 6 are setting up the workspace while in line 8 the repository location is printed out. This time the method to-
String is invoked because otherwise we would print out an object and not a text string.

```plaintext
1  call loadClasses
2  path = "C:\_checkouts\repos"
3  .FSEntryFactory~setup
4  workspace = .SVNWorkspaceManager~createWorkspace("file", path)
5  say workspace~getLocation~toString
6  ::requires BSF.CLS
7  ::requires 99_utils.rex
```

Source code 8: Working with workspaces (ooRexx)

### 4.4.4 Working with repositories

Source code 9 below is an ooRexx copy of Source code 4 in chapter 4.3.2 and itself prints out the latest revision retrieved from a repository. I think the example is self-explanatory and does not need any further explanations.

```plaintext
1  call loadClasses
2  url = "http://localhost/repos"
3  .DAVRepositoryFactory~setup
4  location = .SVNRepositoryLocation~parseURL(url)
5  repository = .SVNRepositoryFactory~create(location)
6  say repository~getLatestRevision
7  ::requires BSF.CLS
8  ::requires 99_utils.rex
```

Source code 9: Working with repositories (ooRexx)

---

44 As you can see there is no need for brackets like in Java when executing a method without having parameter values.
Developed examples

The following chapter shows the source code of my developed examples together with a detailed description. All examples are made available with the publication of this paper.

5.1 Standalone examples

5.1.1 Example 1 – Simple command-line wrapper

The first example is an implementation of the Subversion command-line client with only the checkout and commit commands implemented. The script has nearly the same behaviour as Subversion's build-in client. Executing the script you have to specify the connection protocol (HTTP or SVN) and after that the URL of the repository to communicate with. There is an inquiry to be sure that the given FQDN is correct. After specifying the workspace directory you can work using `svn checkout` or `svn commit` as statements. By checking out a revision you must submit user name and password. Matching user name with password will only be checked if there is authentication needed. After the checkout you can edit files you want using your preferred editor. Committing changes is done by entering `svn commit` with the optional parameter `-m [commit message]`. Again you have to specify user name and password. This is a wanted behaviour because you can commit revisions with user names other than the standard user chosen by Subversion (e.g. currently logged in user in Windows). Exiting program is done by typing `svn exit`.

If you have a look at source code 10, you will see that on line 61 file `BSF.CLS` is loaded (BSF4Rexx support)\(^{45}\). This example is the only one having routine `loadClasses` implemented by itself\(^{46}\). The first lines calling some routines defined later in the script for setting connection protocol, repository location and workspace directory. Then a loop is defined which waits for user input (line 22). Possible input is `svn checkout`, `svn commit`, `svn help` and `svn exit`. On checkout and commit the user has to authenticate himself. Besides, there is an inquiry if checkout was already done because you can checkout a repository only once into one specific directory. The `help` routine prints out only one single line (line 13) and is actually an example of how to extend the program with more functionality.

---

\(^{45}\) After invoking an ooRexx script `requires` directives are executed first.
\(^{46}\) As mentioned earlier all other scripts require file `99_utils.rex`. 
If you look at the routine `setup` (for connection protocol configuration; line 79), you will see that there are two different methods called depending on the protocol you specified.

```plaintext
/*
* 01_simple-cmd-line.rex, 2005-06-12
* Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
* Command-line tool for simple interaction. Protocol, repository and
* workspace must be defined. Simple "checkout" and "commit" functions are
* implemented including authentication. Authentication has to be done
* on each "Checkout" or "commit" event.
*/
call loadClasses
protocol = setup()
location = setLocation(protocol)
workspace = setWorkspace()
say ">> Setup completed"
quit = false
checkedout = false
do while quit = false
parse pull cmd
parse var cmd svn command option
if (svn! = "svn") then call help(cmd)
else do
select
    when command = "checkout" then
        do
            if (checkedout = false) then
                -- Authentication works only if user must be authenticated against repository
                auth = authentication(workspace)
                say ">> Checking out"
                -- For getting latest revision
                head = .bsf~bsf.getStaticValue("org.tmatesoft.svn.core.ISVNWorkspace", "HEAD")
                revision = workspace~checkout(location,head,.false)
                checkedout = true
                say ">> Checked out revision" revision
                say ">> Edit needed files now..."
            end
        else say ">> Already checked out, use 'svn update' instead"
    end
    when command = "commit" then
        do
            -- Authenticate again for submit changes as different user
            parse var option "-m " msg
            revision = workspace~commit(msg)
            say ">> Committed revision" revision
        end
        when command = "help" then call help
        when command = "exit" then quit = true
    otherwise call error(cmd)
end
end
do while quit = false
quit = true
end
end
say ">> Quitting"
exit
::requires BSF.CLS
::routine loadClasses public
classes = "org.tmatesoft.svn.core.internal.io.dav.DAVRepositoryFactory",
        "org.tmatesoft.svn.core.internal.io.svn.SVNRepositoryFactoryImpl",
        "org.tmatesoft.svn.core.internal.ws.fs.FSEntryFactory",
        "org.tmatesoft.svn.core.io.SVNRepositoryLocation",
        "org.tmatesoft.svn.core.ISVNWorkspace",
        "org.tmatesoft.svn.core.SVNWorkspaceManager",
        "org.tigris.subversion.javahl.SVNClients",
        "java.io.File"
```
do i = 1 to words(classes)
    class = word(classes,i)
    rexxClass = substr(class,lastpos('.',class)+1)
    .bsf-bsf.import(class,rexxClass)
end

::routine setup
quit = false
do while (quit = false)
    say ">> connection over 'http' or 'svn': ('file' not yet implemented)"
    parse pull conn
    if (conn = "http") then
        .DAVRepositoryFactory-setup
        quit = true
    end
    else if (conn = "svn") then
        .SVNRepositoryFactoryImpl-setup();
        quit = true
    end
    else call error(conn)
end
.FSEntryFactory~setup();
say ">> Connection type set"
return conn

::routine setLocation
parse arg prot
quit = false
do while (quit = false)
    say ">> set repository location:"
    parse pull loc
    if (loc = ") then call error(loc)
    else
        do
            parse var loc protocol "://" domain "." toplevel_plus_port "/" path
            if (prot = protocol & domain <> ") & toplevel_plus_port <> ") then
                location = .SVNRepositoryLocation~parseURL(loc)
                quit = true
            end
        end
        else call error(loc)
    end
end
say ">> Location of repository set"
return location

::routine setWorkspace
say ">> set workspace directory: (blank for ")
parse pull ws
if (ws = "") then ws = directory()
workspace = .SVNWorkspaceManager~createWorkspace("file",ws)
say ">> workspace set"
return workspace

::routine authentication
use arg workspace
say ">> User:"
parse pull user
say ">> Password:"
parse pull pw
workspace-setCredentials(user,pw);
return 1

::routine help
say ">> help yourself :)

::routine error
parse arg cmd
say ">> unknown command: ")
say ">> type 'svn help' for help"
5.1.2 Example 2 – Variation of the command-line wrapper

Example 2 is a variation of the first example with some minor changes and improvements in functionality. Now one can also update existing checkouts using `svn update` and every functionality is implemented as a single routine. On lines 59 to 62 you can see that the connection protocol must not explicitly be specified, but it is detected by entering only repository's URL. On line 20 the static variable `head` is specified which consists of an integer (here $-2^{47}$) referring to the revision number to be checked out or updated. A routine which is also implemented in example 1 showed earlier is named `error` (line 17). When a user types a command the program does not recognize, this routine is called with the unknown command as an argument to be printed out. If one tries to commit a new revision without having changed anything, the returned revision number is $-1$ (lines 103 and 104). This indicates that nothing has changed and no new revision was committed (else the new revision number is returned).

```plaintext
/*
 * 02_simple-cmd-line-variation.rex, 2005-06-12
 * Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
 * *
 * Command-line tool for simple interaction. Protocol, repository and
 * workspace must be defined. Simple "checkout", "commit" and "update"
 * functions are implemented including authentication. Authentication
 * has only to be done once.
 * Improvement towards 01_simple-cmd-line.rex: Protocol is detected
 * through repository URL and must not be defined separately. Each
 * functionality is implemented as a routine. Also there are some minor
 * improvements...
 * */

call loadClasses
location = setLocation()
workspace = setWorkspace()
auth = authentication(workspace)
head = .bsf-bsf.getStaticValue("org.tmatesoft.svn.core.ISVNWorkspace", "HEAD")
say ">> Setup completed"
quit = false
checkedout = false
do while quit = false
parse pull cmd
parse var cmd svn command option
if (svn! = "svn") then call help(cmd)
else do
select
when command = "checkout" then do
if checkedout = false then checkedout = checkout(workspace,location,head)
else say ">> Already checked out, use 'svn update' instead"
end
when command = "commit" then committed = commit(workspace,option)
when command = "update" then updated = update(workspace,head)
when command = "help" then call help
when command = "exit" then quit = true
```

47 A negative integer indicates that the latest revision is going to be checked out.
otherwise call error(cmd)
end
end
end
exit
::requires BSF.CLS
::requires 99_utils.rex
::routine setLocation
quit = false
do while (quit = false)
say ">> set repository location:
parse pull loc
if (loc = "") then call error(loc)
else do
parse var loc protocol "://" domain "," toplevel_plus_port ":/" path
if ((protocol = "http" | protocol = "svn") & domain <> "" & toplevel_plus_port <> "") then do
if (protocol = "http") then .DAVRepositoryFactory~setup
else if (protocol = "svn") then .SVNRepositoryFactoryImpl~setup();
else call error(conn)
.FSEntryFactory~setup();
location = .SVNRepositoryLocation~parseURL(loc)
quit = true
end
else call error(loc)
end
do while (quit = false)
say ">> Location of repository set"
return location
do while (quit = false)
::routine setWorkspace
say ">> set workspace directory: (blank for "||directory()||")"
parse pull ws
if (ws = "") then ws = directory()
workspace = .SVNWorkspaceManager~createWorkspace("file",ws)
say ">> workspace set"
return workspace
do while (quit = false)
::routine authentication
use arg workspace
say "User:"
parse pull user
say "Password:"
parse pull pw
workspace~setCredentials(user,pw);
return 1
do while (quit = false)
::routine checkout
use arg workspace,location,head
say ">> Checking out"
revision = workspace~checkout(location,head,.false)
say ">> Checked out revision" revision
say ">> Edit needed files now..."
return true
do while (quit = false)
::routine commit
use arg workspace,option
-- Only message argument is implemented
parse var option "-m " msg
revision = workspace~commit(msg)
if (revision = -1) then say ">> Nothing to commit..."
else say ">> Committed revision" revision
return 1
do while (quit = false)
::routine update
use arg workspace,head
revision = workspace~update(head)
say ">> Updated to revision" revision
return 1
}
::routine help
say ">> help yourself :)

5.1.3 Example 3 – Last modification property

The following example inserts a last-modified timestamp property (retrieved from the operating system\(^{48}\)) for every directory and file found in a given workspace. Again routine `loadClasses` is called and the repository and workspace location are defined. Then there are the common configuration lines (lines 18 to 22) for setting up repository and workspace. On lines 25 and 26 RexxUtilities functions are loaded for getting a file-tree and the last-modified timestamp from a local file. The workspace's file-tree is retrieved on line 30 – returned are file and directory names (including any sub-directories). There are some inquiries regarding system's memory status and file system and Subversion's `.svn` configuration directory (which is excluded). For every directory and file found the last-modified timestamp is added as the property defined in variable `property` (here “Last-Modified”). The functionality is implemented using JavaSVN's `setPropertyValue` method on the workspace (line 43). As a comment there is also an alternative solution getting help from Subversion's command-line client. As editing properties is also a change recognized by Subversion a commit has to be done to generate a new revision (line 53).

---

48 This cannot be done on FAT file systems.
5.1.4 Example 4 – Self defined keywords

With the example shown below it is possible to place self defined keywords inside files, which will be replaced with a certain text specified by a property value with the same name as the keyword. Executing program assumes that Subversion's file and directory properties which will be replaced have already been set.

On line 24 to 27 stem variables are defined containing the keywords to replace (and the number of keywords). If the script is invoked specifying `update` as an argument, at first the workspace is updated to the latest revision (`-1`). Every file found in the workspace is opened and it is searched for properties specified in variable `property` delimited by characters defined in variable `delimiter`. If a text string equals that configuration, it is replaced by the value of the equally named file property. A variable `show` of type boolean has to be set to indicate if the content of the file has to be printed out (line 32).

If the script is invoked with `cleanup` or `commit` as arguments, the former replaced text string is reverted (that is, replacing the string a second time with the value specified in the
corresponding property variable). This behaviour has the effect that one can specify a certain text string which will be inserted at checkout time. Therefore a user has the ability to checkout the same files with different property values inserted (e.g. if he wants to set different copyright texts). If a commit is done, the replaced text should be reverted, so that the next user can do the same. For the reversion each file is opened and text strings delimited by specified delimiters are replaced. The difference between cleanup and commit is that on commit the changed files are committed to the repository.

```plaintext
/*
 * 04_self-defined-keywords.rex, 2005-06-20
 * Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
 * Running "update" script replaces self defined keywords inside files with values of properties.
 * "cleanup" reverts all keywords in files.
 * "commit" does the same as "cleanup" but commit changes at the end.
 */
call loadClasses
path = "C:\_checkouts\repos"
-- Repository and working directory configuration
.DAVRepositoryFactory~setup
.FSEntryFactory~setup
workspace = .SVNWorkspaceManager~createWorkspace("file",path)
-- If authorization is needed
workspace~setCredentials("alex","alex");
property.0 = 3
property.1 = "Last-Modified"
property.2 = "Author"
property.3 = "Date"
delimiter = "$"
-- Show content of files?
show = .false
if arg(1) = "update" then do
   -- Latest revision = any negative integer
   revision = workspace~update(-1)
   say "Updated to revision" revision
   filetree = SysFileTree(path || "\.*",file,"FSO")
do i = 1 to file.0
   if i = 1 to file.0
      if file.i~lastpos('.svn') = 0 then do
         relative = .SVNUtil~getWorkspacePath(workspace,file.i)
         do j = 1 to property.0
            if workspace~getPropertyValue(relative,property.j) <> .NIL then do
               say "-----------------------------"
               say file.i property.j workspace~getPropertyValue(relative,property.j)
               say "-----------------------------"
            end
            f = .stream~new(file.i)
f~open
            text = f~charin(1,f~chars)
            text = text~changestr(delimiter||property.j||delimiter,delimiter||property.j||": "||workspace~getPropertyValue(relative,property.j)||delimiter)
f~charout(text,1)
         end
      end
   end
end
*/
```
else if arg(1) = "cleanup" | arg(1) = "commit" then do
   filetree = SysFileTree(path || "\.*",file,"FSO")
   do i = 1 to file.0
      if file.i~lastpos('.svn') = 0 then do
         relative = .SVNUtil~getWorkspacePath(workspace,file.i)
         do j = 1 to property.0
            if workspace~getPropertyValue(relative,property.j) <> .NIL then do
               say "-----------------------------------------------"
               say file.i property.j workspace~getPropertyValue(relative,property.j)
               say "----------------"
               f = .stream-new(file.i)
               f-open
               text = f-charin(1,f-chars)
               text_clean = text~changestr(delimiter||property.j||": "||workspace~getPropertyValue(relative,property.j)||delimiter,delimiter||property.j||delimiter)
               f-charout(text_clean,1)
               if (text-length-text_clean-length)>0 then do
                  text_append = " "
                  do l = text_clean-length to text-length
                     text_append = text_append||" "
                  end
                  f-charout(text_append,text_clean-length+1)
               end
            end
            if show = .true then say text_clean
            f-close
         end
      end
      if arg(1) = "commit" then do
         revision = workspace~commit(""
         say "Commited revision" revision
      end
      else say "Please submit 'update', 'cleanup' or 'commit' as an argument."
end
::requires BSF.CLS
::requires 99_utils.rex

5.1.5 Example 5 – Listing repositories

Example 5 generates an HTML file containing interesting information about a repository. There is a routine called getTree (starting at line 46) which digs into the whole repository file tree. By means of the method repository~getDir (line 51) a Java collection is returned containing all directory entries found in a specific directory. Therefore for getting all files and directories (even in sub-directories) the method getTree has to call itself recursively (line 73). From each directory entry the creation date, the author, the revision of the last modification and the size in bytes are retrieved (lines 58 to 61) and an HTML string is created (lines 66 to 70) which is returned. The returned HTML string is then embedded in another string representing an HTML framework for a web-site (lines 25 to 31). The complete HTML site is written to a file (lines 34 to 37) which is finally opened (with your standard web-browser) using the command on line 40.
Developed examples

```rex
/*
* 05_repository-listing.rex, 2005-06-22
* Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
* Generates an HTML file for a specified repository
* containing information which may be interesting.
*/
call loadClasses
repos = "http://localhost/repos"
outputFile = "05_repository.html"
-- Repository configuration
.DAVRepositoryFactory~setup
location = .SVNRepositoryLocation~parseURL(repos)
repository = .SVNRepositoryFactory~create(location)
revision = repository~getLatestRevision
-- Recursive call in routine for building directory tree
tree = getTree(repository,"","",revision,"")
-- Creating html output
out = "<html><head><title>repos - revision " revision "</title></head>"
out = out "<body><h2>repos - revision " revision "</h2>
out = out "<table width = 100%><tr><th align = left>Path</th><th align = left>CreationDate</th><th align = left>Author</th><th align = left>LastModifiedRevision</th><th align = left>Size (byte)</th></tr>
out = out tree
out = out "</table></body></html>"
-- Writing html file
f = .stream~new(outputFile)
f~open("replace")
f~charout(out,1)
f~close
-- Opening html file
address cmd outputFile
::requires BSF.CLS
::requires 99_utils.rex
-- Routine for building directory tree
::routine getTree
use arg repository,path,revision,tree
m = .HashMap~new
v = .Vector~new
-- Don't need 'm' and 'v' but have to be declared
c = repository~getDir(path,revision,m,v);
i = c~iterator
do while i~hasNext
dir = i~next
dirName = dir~getName
dirKind = dir~getKind~toString
dirPath = path || dirName
dirCreationDate = dir~getDate~toString
dirAuthor = dir~getAuthor
dirLastModifiedRevision = dir~getRevision
dirSize = dir~size
if dirAuthor = .Nil then dirAuthor = "[no author]"
-- Generates html output string
tree = tree "<tr><td>
dirPath"</td>"
tree = tree "<td>
dirCreationDate"</td>"
tree = tree "<td>
dirAuthor"</td>"
tree = tree "<td>
dirLastModifiedRevision"</td>"
tree = tree "<td>
dirSize"</td>"
end
return tree
```

Source code 14: File 05_repository-listing.rex
Figure 6 shows an example HTML site created with source code 14.

http://localhost/repos2 - revision 202

<table>
<thead>
<tr>
<th>Path</th>
<th>CreatedDate</th>
<th>Author</th>
<th>LastModifiedRevision</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ps</td>
<td>Sun 30 10:12:31</td>
<td>[no author]</td>
<td>2005</td>
<td>0</td>
</tr>
<tr>
<td>ps/lib1</td>
<td>Sun 30 10:12:31</td>
<td>[no author]</td>
<td>2005</td>
<td>0</td>
</tr>
<tr>
<td>ps/sample2</td>
<td>Sun 30 10:12:31</td>
<td>[no author]</td>
<td>2005</td>
<td>133</td>
</tr>
<tr>
<td>ps/sample2/sample2</td>
<td>Sun 30 10:12:31</td>
<td>[no author]</td>
<td>2005</td>
<td>0</td>
</tr>
<tr>
<td>ps/sample2d</td>
<td>Wed 30 12:05:30</td>
<td>[no author]</td>
<td>2005</td>
<td>13109</td>
</tr>
</tbody>
</table>

5.1.6 Example 6 – Charting file activity

The next example generates a graphical chart which is built up on repository file activity data using JFreeChart libraries. Again there is the routine `getTree` (starting at line 51) which gets all directory entries for the whole repository. This time the creation date of a directory entry is retrieved (line 61), parsed (line 62), transformed in a moderate format (line 68) and saved in a variable of type directory. The creation date of a directory entry is the key for the directory variable `d` while the value is set to 1. If another file was created on the same date, the variable `d` on the position of the date is increased by 1. So routine `getTree` counts the number of directory entries created on a specific date and saves the data into variable `d`. Another example is the revision number a directory entry was last modified. If you want to display a chart representing that kind of information, uncomment line 67 (and comment the following line). On line 29 a JFreeChart chart type is chosen whose dataset values are filled on lines 32 to 34 (with the numbers count in routine `getTree` and saved to variable `d`). After that on line 38 the title of the chart is set. In lines 41 to 43 the chart is generated and saved to a JPG file named after `outputFile`. Some parameters are set including charts dimensions (here: 800 pixel width and 500 pixel height). At last the chart is opened using the default picture viewer for JPG files (line 46).
Developed examples

```plaintext
* showing creation date of files or revision number of last modification.
*/
call loadClasses
repos = "http://localhost/repos"
outputFile = "06_chart.jpg"

-- Repository configuration
.DAVRepositoryFactory~setup
location = .SVNRepositoryLocation~parseURL(repos)
repository = .SVNRepositoryFactory~create(location)
revision = repository~getLatestRevision
d = .directory~new

-- Recursive call in routine for building directory tree
tree = getTree(repository,"",revision,"",d)

-- JFreeChart chart type
dataset = .DefaultPieDataset~new

-- Save data in dataset
do item over d
    dataset~setValue(item,d[item])
end

-- Whatever you set 'option' to
title = "Creation date of files"

-- Generate chart and write outputFile
chart = .ChartFactory~createPieChart3D(title,dataset,.true,.true,.false)
chartFile = .File~new(outputFile)
.ChartUtilities~saveChartAsJPEG(chartFile,chart,800,500)

-- Have a look at the chart
address cmd outputFile
::requires BSF.CLS
::requires 99_utils.rex

::routine getTree
use arg repository,path,revision,tree,d
m = .HashMap~new
v = .Vector~new
c = repository~getDir(path,revision,m,v);
i = c~iterator
do while i~hasNext
    dir = i~next
dirName = dir~getName
dirKind = dir~getKind~toString
dirCreationDate = dir~getDate~toString
parse var dirCreationDate nameOfDay month day time timeZone year
dirPath = path || dirName
dirLastModifiedRevision = dir~getRevision
dirPath = path || dirName

-- Revision number or date?
option = "revision" dirLastModifiedRevision
option = day||"-"||month||"-"||year
time
if d[option] = .NIL then d[option] = 1
else d[option] = d[option]+1

if dirKind = "<dir>" then tree = getTree(repository,path||dirName|
|"";revision,tree,d)
end
return tree
```

Source code 15: File 06_charting-file-activity.rex
Figure 7 below displays a possible chart generated with JfreeChart and data retrieved from a repository.

![Chart showing creation date of files](image_url)

**5.1.7 Example 7 – Repository listing standalone server**

Example 7 is a minimalistic HTTP standalone server written in ooRexx. The program listens on a specific host and port waiting for a client to connect. On connection the script contacts the defined Subversion repository and retrieves the same data as in example 5 (chapter 5.1.5), creates an HTML table and sends an HTTP answer back to the client containing this information. As the implementation of HTTP is a little lazy, I recommend using Mozilla's Firefox, Opera or Netscape's Navigator 7.x as web-browser\(^49\).

First on line 17 to 20 the ooRexx socket functions are loaded which are needed to handle any connection type. On line 24 the host address is set to the IP address which the method `SockGetHostId()` returns\(^50\). If you know what you are doing, you can set the host IP address on your own (see next line). On the IP address specified, the created socket will be bound. That means the program will be listening on this IP address and the port defined on

---

\(^{49}\) The Script does not really work with Microsoft's Internet Explorer or Netscape's Navigator 4.x.

\(^{50}\) The `SockGetHostId()` method returns the host ID.

---
Developed examples

line 28 (here 8081) for incoming connections. There is an endless loop waiting for clients to connect (line 33). If a client establishes a connection, the string transmitted from the client is saved to the variable `InString` (line 39). After that the string is parsed and the repository with an optional defined revision is saved. The HTML output is generated and saved in a variable called `out`. First an HTML form is defined (lines 54 to 60) enabling a user to submit a repository location and a revision number. If the user has already requested a resource from this server by submitting this form (checked in line 62), the routine `listFiles` is called which generates the same information as example 5 and returns an HTML formatted string (lines 103 to 109). The response from the server is saved in a variable called `OutString`. The first three lines (lines 66 to 68) containing HTTP header information (HTTP status code, content-type and content-length) are followed by an empty line. Then the HTTP body is attached – the variable `out`, containing the HTML string which will be displayed in the user's web-browser. At last `OutString` is sent back to the client (line 72) and the client's connection is closed. After that the server is able to handle another client connection (synchronous web-server).

```/*
* 07_repository-listing-standalone-server.rex, 2005-06-22
* Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
* 
* A HTTP standalone server written in ooRexx. Program listens
* on a specific host and port and waits for a client connection.
* On connection the program contacts the specified Subversion
* repository and retrieves some data, creates an HTML table and
* sends an HTTP answer back to the client.
* Do not really work with Microsoft's Internet Explorer and
* Netscape's Navigator 4.x because of a very lazy interpretation
* of the HTTP. I recommend using Mozilla's Firefox, Opera or
* Netscape's Navigator 7.x.
* 
*/
if RxFuncQuery("SockLoadFuncs") then do
  call RxFuncAdd "SockLoadFuncs","RXSOCK","SockLoadFuncs"
  call SockLoadFuncs
end
socket = SockSocket("AF_INET","SOCK_STREAM","0")
host.!addr = SockGetHostId()
  -- host.!addr = 127.0.0.1
host.!family = "AF_INET"
host.!port = 8081
call sockbind socket,"host.!
call SockListen socket, 1
```

50 If you are not connected to the Internet or to an Ethernet, the method will return 127.0.0.1 representing your local computer. If your computer is in a network, it is likely to happen that the specified network IP address is returned (maybe something like 192.168.0.x). If you are connected to the Internet without being in a local network, your Internet provider will assign you an IP address which then is returned using `SockGetHostId()`.
Developed examples

```plaintext
do forever
say "------------------------"
say "Waiting at " host.!addr ||":"|| host.!port "for a client to connect..."
ClientSocket = SockAccept(socket)
say "Client has established connection."
call SockRecv ClientSocket, "InString", 256
say "String read from client: " || InString || ""
repository = ""
revision = ""
pause var InString "GET /" path "?repository = " repository "&revision = " revision "
" else
repository = repository~changestr("%3A",":")
repository = repository~changestr("%2F","/")
out = ""
out = out||"<html><head><title>List repository</title></head><body>"
out = out||"<form action = / method = get>"
out = out||"Repository (URL): <input type = text name = repository value = " 
repository ">
"&nbsp;&nbsp;&nbsp;&nbsp;
" Revision: <input type = text name = revision value = " revision 
"&nbsp;&nbsp;&nbsp;&nbsp;
" submit value = 'List repository >>'>"
out = out||"</form>"
if repository <> "" then out = out||listFiles(repository,revision)
-- out = out||"<br><br><small><i>"InString"</i></small>
out = out||"</body></html>"
OutString = "HTTP/1.0 200 OK" || "0D0A"x -- Represents '
'
OutString = OutString||"Content-Type: text/html" || "0D0A"x
OutString = OutString||"Content-Length:" out~length || "0D0A"x
OutString = OutString || "0D0A"x
OutString = OutString || out
call SockShutDown ClientSocket, 2
call SockClose ClientSocket
say "An answer was send back to the client (" || OutString~length "bytes)...
" end
::requires BSF.CLS
::requires 99_utils.rex
::routine listFiles
use arg repos,revision

call loadClasses
-- Repository configuration
.DAVRepositoryFactory~setup
location = .SVNRepositoryLocation~parseURL(repos)
repository = .SVNRepositoryFactory~create(location)
rev = repository~getLatestRevision
if revision = "" | revision>rev then revision = rev
-- Recursive call in routine for building directory tree
tree = getTree(repository,"",revision,""
-- Creating html output
out = "<h2>repos - revision" revision "</h2>
out = out||"<table width = 100%><tr><th align = left>Path</th><th align = left>Author</th><th align = left>CreationDate</th><th align = left>LastModifiedRevision</th><th align = left>Size (byte)</th></tr>"
out = out||tree
out = out"</table>"
```

Page 55
Developed examples

```
return out
-- Routine for building directory tree
::routine getTree
  use arg repository,path,revision,tree
  m = .HashMap~new
  v = .Vector~new
  -- Don't need 'm' and 'v' but have to be declared
  c = repository~getDir(path,revision,m,v);
  i = c~iterator
  do while i~hasNext
    dir = i~next
    dirName = dir~getName
    dirKind = dir~getKind~toString
    dirPath = path || dirName
    dirCreationDate = dir~getDate~toString
    dirAuthor = dir~getAuthor
    dirLastModifiedRevision = dir~getRevision
    dirSize = dir~size
    if dirAuthor = .Nil then dirAuthor = "[no author]"
    -- Generates html output string
    tree = tree "<tr><td>"dirPath"</td>
    tree = tree "<td>"dirCreationDate"</td>
    tree = tree "<td>"dirAuthor"</td>
    tree = tree "<td>"dirLastModifiedRevision"</td>
    tree = tree "<td>"dirSize"</td></tr>
    -- Recursive call
    if dirKind = "<dir>" then tree = getTree(repository,path||dirName|"/",revision,tree)
  end
  return tree
```

Source code 16: File 07_repository-listing-standalone-server.rex

Figure 8 displays a running server with some clients connected.

![Figure 8: ooRexx standalone server](image)

User's view is shown in figure 9. There you can see the HTML form on top and beneath the corresponding repository entries listed.
5.1.8 Example 8 – HTML repository log information

The next example generates an HTML site out of repository’s log information. You can specify a range of revisions within several information like date, author, commit message or changed paths are printed out. After the typical repository configuration lines, a Java array of type string (line 24) and a vector (line 27) are created because later called method needs parameters of these types. Then you can define from which revision to start and where to end (lines 30 to 31). All log entries of a certain revision range are returned (as a Java collection) using method log with parameters shown in line 36. After that the creation of the HTML site begins. First the HTML framework and head are defined (lines 39 to 42), then HTML links pointing to the detailed log entry of a certain revision which is created later (because file can become extremely large). There is a loop over all found log entries generating detailed revision information (lines 58 to 96). The changed paths of a certain log entry are returned as a Java map using getChangedPaths. There must be another loop (lines 81 to 92) printing out changed paths (if exists). In the end the HTML code (variable out) is written to a file (lines 101 to 104) named after variable outputFile defined at the start of the script. Then the saved file is opened using address cmd.

```java
/*
 * 08_repository-log-information-html.rex, 2005-07-07
 * Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
 * Generates an HTML file for a repository containing several log information (revision, date, author, commit-message and changed paths) of a specified revision range (Attention: File could be huge).
 */
call loadClasses
repos = "http://localhost/repos"
outputFile = "08_repository.html"
-- Repository configuration
.DAVRepositoryFactory~setup
location = .SVNRepositoryLocation~parseURL(repos)
repository = .SVNRepositoryFactory~create(location)
revision = repository~getLatestRevision
```
-- creating a JAVA array with 1 element (and leave it null because we don't need it)
s = .bsf->bsf.createArray(.bsf|rexx|string.class,1)

-- Method 'log' needs a collection as parameter that's why we define a vector here
v = .vector->new

-- Checks all revisions from 0 to revision
startRevision = 0
endRevision = revision

-- Gets a collection with all the log entries
logEntries = repository->bsf.invokeStrict

-- Creates html links over all entries
out = "<html><head><title>Repository "repos"</title></head><body>"
out = out||"<a name = top></a><h2>Repository "repos"
Revision "startRevision" to "endRevision"</h2>
out = out||"<center>Revision "
i = 0
do while entries->hasNext
  logEntry = entries->next
  out = out||"<a href = ">#"||logEntry->getRevision||""||logEntry->getRevision |
|"</a>"
  if i <> endRevision then out = out||" | "
i = i+1
end
out = out||"</center>"

-- Creates detailed revision information over all entries
entries = logEntries->iterator
out = out||"<a name = top></a><h2>Repository "repos"
Revision "startRevision" to "endRevision"</h2>
out = out||"<center>Revision "
i = 0
do while entries->hasNext
  logEntry = entries->next
  date = logEntry->getDate
  author = logEntry->getAuthor
  message = logEntry->getMessage
  if date = .nil then date = "[no date]"
  else date = date->toString
  if author = .nil then author = "[no author]"
  if message = .nil then message = "[no message]"
  out = out||"<a href = ">"||logEntry->getRevision||""||logEntry->getRevision |
|"</a>"
  out = out||"<a name = ""||logEntry->getRevision||""</a>"
  date = logEntry->getDate
  author = logEntry->getAuthor
  message = logEntry->getMessage
  if date = .nil then date = "[no date]"
  else date = date->toString
  if author = .nil then author = "[no author]"
  if message = .nil then message = "[no message]"
  out = out||"<a href = ">
|"||logEntry->getRevision||""
  date = logEntry->getDate
  author = logEntry->getAuthor
  message = logEntry->getMessage
  if date = .nil then date = "[no date]"
  else date = date->toString
  if author = .nil then author = "[no author]"
  if message = .nil then message = "[no message]"
  out = out||"<a href = ">
|"||logEntry->getRevision||""
  date = logEntry->getDate
  author = logEntry->getAuthor
  message = logEntry->getMessage
  if date = .nil then date = "[no date]"
  else date = date->toString
  if author = .nil then author = "[no author]"
  if message = .nil then message = "[no message]"
  out = out||"<a href = ">
|"||logEntry->getRevision||""

if logEntry->getChangedPaths->size>0 then do
  out = out||"<br>Changed paths:<br>"
  changedPathSet = logEntry->getChangedPaths->keySet
  changedPaths = changedPathSet->iterator
  do while changedPaths->hasNext
    -- invokeStrict here because method need a string as argument
    entryPath = logEntry->getChangedPaths->bsf.invokeStrict("get","st",
    changedPaths->next)
    out = out||"<font style = 'width:15pt'>"||entryPath->getType
    out = out||""||entryPath->getPath
    if entryPath->getCopyPath <> .nil then do
      out = out||"(from"||entryPath->getCopyPath||", revision"
      entryPath->getCopyRevision||")"
    end
    out = out||""||entryPath->getCopyPath||")"
    out = out||""||entryPath->getCopyRevision||")"
  end
  out = out||"<hr>|"""||entryPath->getCopyPath||")"
  end
out = out||"</body></html>"
The following figure 10 shows examples of detailed log entries. They are written beneath each other separated by a horizontal line. That fact and the large number of revisions are reasons why the generated file tends to be extremely large. This is why there is a linked list containing all revision numbers at the top of the file, where a user can hop easily to one revision and back to the beginning.

### Revision 40

Date: Mon Jun 20 21:41:27 CEST 2005  
Author: [no author]  
Message:  

Changed paths:  
- M /ip4/001_html.html  
- M /ip4/002_html.html

### Revision 41

Date: Tue Jun 21 17:17:25 CEST 2005  
Author: [no author]  
Message: edited something

Changed paths:  
- M /ip4/008_css.css

Figure 10: Detailed log entries example

### 5.1.9 Example 9 – XML repository log information

Example 9 is a copy of example 8 showing repository's log information as an XML structure. If you compare examples 8 and 9, you will find many matching lines. The difference starts on line 38, where a new XML document named `document` is generated (using dom4j...
packages). In the following four lines the root element is created and added to the XML document. In line 46 a loop over all log entries is defined, information about a certain log entry is saved to variables (lines 49 to 51), inquires if a variable contains no value are set and the elements are added to the XML document. For not getting a too big file, information about changed paths are not retrieved. After the loop and therefore after the generation of the XML document, it is saved to a file (lines 72 to 74) using dom4j’s XMLWriter class. Just before that the method createPrettyPrint (for generating a decent output format) is invoked and the encoding-type of the document is set. Finally the XML document is opened (probably in your standard web-browser).

```java
/*
 * 09_repository-log-information-xml.rex, 2005-07-06
 * Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
 * *
 * Generates an XML file with some log information (revision, date, author and commit-message) from a specified range of revisions. Uses dom4j (http://www.dom4j.org) for the XML structure and dom4j's XMLWriter class for writing the file.
 * *
 */
call loadClasses
repos = "http://localhost/repos"
outputFile = "09_repository.xml"
-- Repository configuration
.DAVRepositoryFactory~setup
location = .SVNRepositoryLocation~parseURL(repos)
repository = .SVNRepositoryFactory~create(location)
revision = repository~getLatestRevision
-- Creates a JAVA array with 1 element (and leave it null because we don't need it)
a = .bsf-bsf.createArray(.bsf4rexx-string.class,1)
-- Method 'log' needs a collection as parameter that's why we define a vector here
v = .vector~new
-- Checks all revisions from 0 to head
startRevision = 0
endRevision = revision
-- Gets a collection with all the log entries
logEntries = repository-log(a,v,startRevision,endRevision,.true,.true)
entries = logEntries-iterator
-- Creates an xml-document and a root element
document = .DocumentHelper~createDocument
root = document~addElement("repository"),
~addAttribute("xmlns","http://yourdomain.com/whatever/namespace"),
~addAttribute("xmlns:xsi","http://www.w3.org/2001/XMLSchema-instance"),
~addAttribute("xsi:schemaLocation","http://yourdomain.com/whatever/schema.xsd")
-- Creates xml structure
entries = logEntries-iterator
do while entries-hasNext
logEntry = entries-next
date = logEntry~getDate
author = logEntry~getAuthor
```

51 The object `.nil` does not contain any data. It usually represents the absence of an object, as a null string represents a string with no characters (equivalent to Java’s `null` value).
message = logEntry~getMessage
if date = .nil then date = "[no date]"
else date = date~toString
if author = .nil then author = "[no author]"
if message = .nil then message = "[no message]"

element_revision = root~addElement("revision")
element_revision_nr = element_revision~addElement("revision_nr"),
    ~addText(logEntry~getRevision)
element_date = element_revision~addElement("date"),
    ~addText(date)
element_author = element_revision~addElement("author"),
    ~addText(author)
element_message = element_revision~addElement("message"),
    ~addText(message)
end

-- Writing xml file
format = .OutputFormat~createPrettyPrint
format~setEncoding("ISO-8859-1")
writer = .XMLWriter~new(.FileWriter~new(outputFile), format)
writer~write(document)
writer~close
address cmd outputFile
::requires BSF.CLS
::requires 99_utils.rex

Look at figure 11 if you want to have an idea of the generated XML document.

```xml
<repo version="1.0" encoding="ISO-8859-1">
    <revision revision_nr="0">
      <date>Mon Jun 06 00:00:00 CEST 2005</date>
      <author>[no author]</author>
      <message>[no message]</message>
    </revision>
    <revision revision_nr="1">
      <date>Mon Jun 06 00:02:07 CEST 2005</date>
      <author>team/author</author>
      <message>Initial import</message>
    </revision>
    <revision revision_nr="2">
      <date>Mon Jun 06 01:03:31 CEST 2005</date>
      <author>[no author]</author>
      <message>commit</message>
    </revision>
    <revision revision_nr="3">
      <date>Mon Jun 06 01:04:54 CEST 2005</date>
      <author>[no author]</author>
      <message>commit</message>
    </revision>
  </repository>
</repo>
```

Figure 11: Generated XML structure out of repository's log information

5.1.10 Example 10 – Editing revision properties

This example allows the user to add and edit new properties for a specified revision of a repository. Standard settings of Subversion prevent editing revision properties because there is no logging of the modifications made. You have to enable the hook script called pre-revprop-change in repository's hooks directory, which is executed before a change to a revi-
sion property is made (for more information on hook scripts see chapter 5.2).

On line 30 a Java map is returned containing unversioned properties associated with the
given revision. In the next few lines all found properties and their values are printed out.
Then the user has to enter a property name for editing and the new value. If a property does
not exist, a new one is created. In line 46 the property is set using method setRevision-
PropertyValue with the revision, the property name and value as parameters. At last the
new or edited property with its value is printed out.

```plaintext
/*
 * 10_editing-revision-properties.rex, 2005-07-07
 * Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
 * 
 * Script allows to edit (and even add) new properties
 * for a specified revision of a repository.
 * Attention: Hook script 'pre-revprop-change' must be
 * activated because by default Subversion doesn't
 * allow modifications of revision properties.
 * If the hook scripts did something at runtime of this
 * script, then you should have a look at the 'hooks'
 * directory of the specific repository.
 */

call loadClasses
repos = "http://localhost/repos"
-- Repository configuration
.DAVRepositoryFactory~setup
location = .SVNRepositoryLocation~parseURL(repos)
repository = .SVNRepositoryFactory~create(location)
revision = repository~getLatestRevision
say "Properties of repository " || repos || " revision " || revision || "":"
say "--------------------------"
m = .HashMap~new
c = repository~getRevisionProperties(revision,m)
e = c~keySet~iterator
do while e~hasNext
    property = e~next
    say property "=" repository~getRevisionPropertyValue(revision,property)
end
say "--------------------------"
say "Enter property name for editing:"
pull prop
say "New value for property " || prop || "":"
pull value
repository~setRevisionPropertyValue(revision,prop,value)
say "--------------------------"
say "Property " || prop || " has now value " || repository~getRevisionPropertyValue(revision,prop) || "".
:requires BSF.CLS
:requires 99_utils.rex
```

Source code 19: File 10_editing-revision-properties.rex
5.1.11 Example 11 – File information

Example 11 displays information about a text file and its content at a certain revision of a repository. Method `checkPath` on line 25 returns the kind of a node defined by `file` at a `revision`. If it is a file, the script fetches the properties and content of the file (lines 27 and 28). In the next lines there is an inquiry if the file fetched is a text file. As the script prints out the content of the file using the command-line only text characters can be shown. There is no use of printing out binary data. All file properties are printed out in lines 43 to 47. The content of the file was written to a variable `baos` of type `ByteArrayOutputStream`\(^{52}\) (with method `getFile`) which is printed out using the `toString` method on line 52.

```java
/*
11_file-info.rex, 2005-07-08
* Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
*/
call loadClasses
repos = "http://localhost/repos"
file = "anyDirectory/anyFile.txt"
-- Repository configuration
.DAVRepositoryFactory~setup
location = .SVNRepositoryLocation~parseURL(repos)
repository = .SVNRepositoryFactory~create(location)
revision = repository~getLatestRevision
fileProperties = .HashMap~new
baos = .ByteArrayOutputStream~new
nodeKind = repository~checkPath(file,revision)
if nodeKind~toString = "<file>" then do
    repository~getFile(file,revision,fileProperties,baos)
mT = .bsf~bsf.getStaticValue("org.tmatesoft.svn.core.SVNProperty", "MIME_TYPE")
mimeType = fileProperties~bsf.invokeStrict("get","st",mT)
isTextType = .SVNProperty~isTextType(mimeType);
if isTextType = 1 then do
    say
    say file
    say "---------------------"
    say "File properties:"
    say "---------------------"
    iterator = fileProperties~keySet~iterator
do while iterator~hasNext
    propertyName = iterator~next
    propertyValue = fileProperties~bsf.invokeStrict("get","st",propertyName)
say propertyName " = " propertyValue
end
say
say "File contents:"
```

\(^{52}\) This class implements an output stream in which the data is written into a byte array.
Example 12 – Repository information

Example 12 uses some already mentioned routines for generating a complete list of repository's files, their properties, and values. Saving of data is done by creating an HTML file (which can be extremely large – depending on repository's number of files and content).

The well known getTree method is used for digging into the complete directory tree of a repository. For each file entry (inquiry on line 51) the same information is retrieved as in example 11. This time an HTML formatted string (variable tree) is created saving all data. We want to display the information in an HTML document, so characters < and > of files content must be replaced (line 76), so that they are not interpreted by the web-browser. On lines 26 to 29 the HTML framework is generated and saved to a file in lines 32 to 35.
Developed examples

```ex
f = .stream-new(file)
f-open("replace")
f-charout(out,1)
f-close
::requires BSF.CLS
::requires 99_utils.rex
::routine getTree
  use arg repository,path,revision,tree
  m = .HashMap-new
  v = .Vector-new
  c = repository-getDir(path,revision,m,v);
  i = c~iterator
  do while i~hasNext
    dir = i~next
    dirName = dir~getName
    dirKind = dir~getKind~toString
    if dirKind = "<file>" then do
      fileProperties = .HashMap-new
      baos = .ByteArrayOutputStream-new
      repository-getFile(path||dirName,revision,fileProperties,baos)
      mT = .bsf-bsf.getStaticValue("org.tmatesoft.svn.core.SVNProperty", "MIME_TYPE")
      mimeType = fileProperties~bsf.invokeStrict("get","st",mT)
      isTextType = .SVNProperty~isTextType(mimeType);
      if isTextType = 1 then do
        tree = tree "<h3>" path||dirName "</h3>"
        tree = tree||"<b>Property list:</b><br>
        iterator = fileProperties~keySet~iterator
        do while iterator~hasNext
          propertyName = iterator~next
          propertyValue = fileProperties~bsf.invokeStrict("get","st",propertyName)
          tree = tree||propertyName " = " propertyValue "<br>"
        end
        tree = tree||"<br><b>File contents:</b><br>
        -- Need to replace '<' and '>' so that they aren't interpreted because we're
        -- writing a HTML file
        tree = tree||baos~toString~changestr("<","&lt;")~changestr(">","&gt;") "<hr>"
        end
    end
    if dirKind = "<dir>" then tree = getTree(repository,path||dirName|
    end
  end
  return tree
```

Source code 21: File 12_repository-info.rex

Figure 12 below displays a small part of an example output generated by the script. As you can see the content of the file is not interpreted even though there is HTML code, as well.
5.1.13 Example 13 – Checkout revisions

This example checks out a defined revision range and measures needed time. The range is set using variables `startRevision` and `endRevision` on lines 15 and 16. In line 25 a start point of time keeping is set. There is an inquiry to check if

- `startRevision` is smaller than 0,
- `endRevision` is smaller than 0,
- `endRevision` is greater than revision (which is repository's latest revision) and
- `startRevision` is greater than `endRevision`.

If one or more of the statements above are true, there will be an error message, otherwise the script starts with the checkout for every revision in range. On line 36 there is also an inquiry if a workspace already exists in a directory. The revisions are checked out to the path specified by `path` and its subdirectory containing the current revision number. After the checkout needed execution time is saved in variable `endTime`. On the first call to `time` ("E"), the elapsed time clock is started and returns 0. From then on, calls to `time` ("E") return the elapsed time since that first call. At last a message including needed time is printed out.
Developed examples

```ruby
/*
 * 13_checkout-revisions.rex, 2005-07-08
 * Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
 * Script checks out repositories from a defined
 * start to a defined end revision and measures
 * needed time.
 */
call loadClasses
repos = "http://localhost/repos"
path = "C:\_checkouts\repos"
startRevision = 1
endRevision = 2
-- Repository and workspace configuration
.DAVRepositoryFactory~setup
.FSEntryFactory~setup
location = .SVNRepositoryLocation~parseURL(repos)
repository = .SVNRepositoryFactory~create(location)
revision = repository~getLatestRevision
startTime = time("E")
say "Checking out" repos || "."
-- Checks if start and end revision are set correct
if startRevision<0 | endRevision<0 | endRevision>revision | startRevision>endRevision
then say "Revision start and/or end are set incorrect."
else do
  do while startRevision< endRevision
    pathToWS = path||"\"||startRevision
    -- Checks if workspace already exists
    if SysGetFileDateTime(pathToWS) = -1 then do
      workspace = .SVNWorkspaceManager~createWorkspace("file",pathToWS)
      revision = workspace~checkout(location,startRevision,.false)
      say "Checked out revision" revision "to" pathToWS || "."
    end
    else say "Workspace" pathToWS "already exists."
    startRevision = startRevision+1
  end
  endTime = time("E")
-- Only endTime is needed (time between the two time("E") calls)
say "Finished checkout in" endTime "seconds."
::requires BSF.CLS
::requires 99_utils.rex
```

Source code 22: File 13_checkout-revisions.rex

Figure 13: Example output of several revision checkouts

5.1.14 Example 14 – Shut down script

As I have already mentioned there is the problem that people often forget to commit changes they have made. This example which is an approach for Microsoft Windows (but
can easily be transformed for any other platform) substitutes the normal shut down process. Before the computer is turned off, it is checked if the local workspace revision is smaller than the revision of the repository. If so, the workspace will be updated. Also a commit statement is executed (with an automated commit message) if the user has local modifications which are not yet submitted. After updating workspace and repository the computer is shut down either by the user (normal shut down screen) or automatically. The script has to be invoked as a replacement of the normal shut down process.

On line 18 a new OLE-object is instantiated (Wscript.Network) from which the name of the computer and the logged-in user are retrieved and saved (next two lines). This information is inserted in a variable generating the automated commit message in lines 23 and 24. In lines 24 and 39 both workspace's and repository's revisions are saved. In line 41 there is a check if the revision of the repository is smaller than the revision of the workspace. If this is true, the workspace is being updated. The next thing is the commit of eventually done modifications containing a status message. Commented line 60 would shut down the computer without human interaction. In the example an instance of the Shell.Application OLE-object is created and the method ShutdownWindows is invoked which shuts down Windows using normal shut down screen (on Windows XP, for example, with choices Standby, Turn off and Reboot).

If you copy the script, e.g. on your desktop for easier execution, some required paths have to be set absolutely. Have a look at lines 66 to 69 where full paths to 00_set-paths.rex (for setting environmental variables) and 99_utils.rex (for loading Java classes) are set. Path to BSF.CLS need not contain the full path because an environmental variable is pointing to that directory.

```plaintext
/*
  14_shutdown.rex, 2005-07-09
  Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
*/
```

53 For example, clicking on the **Start** button and choosing **Shutdown**.
54 “Object Linking and Embedding developed by Microsoft. Allows objects from one application to be embedded within another” [see Envi05].
55 “The WScript.Network object provides access to Windows networking methods to easily control functions such as adding and displaying network shares and printers. It also exposes several networking properties including the current user name, domain and computer name.” [see Guid05]
56 The shell automation service for managing operating system wide functions, e.g. starting/stopping services, search for files and folders, get system information and so on.
Developed examples

* Script is a substitute for the normal Windows shut down process. It checks if a working copy is up to date and commits changes made to the working copy. You can choose if you want no user interaction or the normal Windows shut down screen.

```
call loadClasses
path = "C:\_checkouts\repos"
WshNetObj = .OLEObject~new("WScript.Network")
computer = WshNetObj~ComputerName
user = WshNetObj~UserName
-- Generates commit message
commit_message = user "committed this revision on" date() time() "from the computer"
computer || "."
-- Repository and workspace configuration
.DAVRepositoryFactory-setUp
.FSEntryFactory-setUp
workspace = .SVNWorkspaceManager-createWorkspace("file",path)
repos = workspace-getLocation-toString
-- Workspace's revision
ws_status = workspace-status("",.true)
ws_revision = ws_status-getWorkingCopyRevision
-- Repository's revision
location = .SVNRepositoryLocation-parseURL(repos)
repository = .SVNRepositoryFactory-create(location)
repos_revision = repository-getLatestRevision
if ws_revision<repos_revision then do
  say "Workspace revision" ws_revision "is not up to date (newest" repos_revision || ")!
  "
  update_revision = workspace-update(repos_revision)
  say "Updated to revision" update_revision || "."
end
else say "Workspace is up to date."
commit_revision = workspace-commit(commit_message)
if commit_revision <> "-1" then do
  say "Committed revision" commit_revision || "."
  update_revision = workspace-update(commit_revision)
  say "Updated workspace to revision" update_revision || "."
end
else say "Nothing to commit."
say "Shutting down computer ..."
-- Shut down for Windows XP - no user interaction
-- address cmd "shutdown -s -t 0"
```

If you do not want to update your working copy (or copies) by hand, you can handle the

Source code 23: File 14_shutdown.rex

5.1.15 Example 15 – Start up script
Developed examples

script shown below to be executed automatically at system start up\(^57\). As in example 14 the script checks if the local workspace revision is smaller than the revision of the repository. If that is the case, the workspace will be updated. The location of the repository is retrieved by using `getLocation` on the workspace object (line 22).

If you copy the script, e.g. to the Autostart folder, again full paths have to be set for the same reason as in example 14 (lines 41 to 44).

```
/*
 * 15_startup.rex, 2005-07-09
 * Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
 * ...
 * Script for automated update of a working copy.
 * Checks if a newer revision exists and if so
 * updates the working copy.
 * For Windows user: Put this in your 'Autostart'
 * directory for automatic execution every time
 * Windows is starting up.
 */
call loadClasses
path = "C:\_checkouts\repos"
-- Repository and workspace configuration
.DAVRepositoryFactory~setup
.FSEntryFactory~setup
workspace = .SVNWorkspaceManager~createWorkspace("file",path)
repos = workspace~getLocation~toString
-- Workspace's revision
ws_status = workspace~status("",.true)
wsv_revision = ws_status~getWorkingCopyRevision
-- Repository's revision
location = .SVNRepositoryLocation~parseURL(repos)
repository = .SVNRepositoryFactory~create(location)
repos_revision = repository~getLatestRevision
if ws_revision<repos_revision then do
say "Workspace revision" ws_revision "is not up to date (newest" repos_revision || ")!
" update_revision = workspace~update(repos_revision)
say "Updated to revision" update_revision || "."
end
else say "Workspace is up to date."
::requires "C:\Dokumente und Einstellungen\Berni\Eigen Dateien\Temp\subversion\_examples\00_set-paths.rex"
::requires "C:\Dokumente und Einstellungen\Berni\ Eigen Dateien\Temp\subversion\_examples\99_utils.rex"
::requires BSF.CLS
```

Source code 24: File 15_startup.rex

### 5.1.16 Example 16 – Virus check

Example 16 does a virus check before committing a new revision using a command-line...
virus scanner\textsuperscript{58}. This can be very useful because a central repository, where many users share files, is an enormous threat for spreading viruses. The path which the virus scanner has to scan is translated in 8.3\textsuperscript{59} format (line 20) because the program is executed through the DOS command-line. On lines 31 to 33 the path to the virus scanner with some parameters\textsuperscript{60} is generated and executed on the following line. A log file is written which contains information about found viruses. The directly generated log file is examined for a certain line (line 42) which tells the program that no viruses were found. If no virus was found, a message is printed out, the log file is deleted and the revision is committed. Else the program tells the user that a virus was found (and that the revision was not committed) and that he has to have a look at the log message for further information.

```plaintext
/*
 * 16_virus-check.rex, 2005-07-09
 * Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
 *
 * Script checks for viruses before committing a new revision. DOS command-line scanner F-Prot from FRISK Software International is used here as an example. After scanning, the log message is investigated for reports about viruses. If none is found, the revision is committed, else there is a warning message and no files are being transferred.
 */
call loadClasses
path = "C:\_checkouts\repos"
-- Scanner needs 8.3 file format
path_8.3 = "C:\_check~1\repos"
logfile = "log.txt"
commitMessage = "A message"
-- Repository and workspace configuration
.DAVRepositoryFactory~setup
.FSEntryFactory~setup
workspace = .SVNWorkspaceManager~createWorkspace("file",path)
repos = workspace~getLocation~toString
-- Scan for viruses and create log file
f_prot = "C:\Programme\F-Prot"
scan = "" || f_prot || " F-PROT.EXE "path_8.3" /ARCHIVE = 10 /AUTO /DELETE /DUMB /
NOBOOT /NOFLOPPY /NOMEM /PACKED /SILENT /REPORT="logfile
address cmd scan
-- Checking log file
f = .stream~new(logfile)
f-open("read")
lines = f-lines
virus = .true
do i = 1 to lines
  if f-linein(i) = "No viruses or suspicious files/boot sectors were found." then virus= .false
```

\textsuperscript{58} You can use any virus scanner you want. I automate the program using the command-line because there is no good implementation of OLE automation.

\textsuperscript{59} That means eight characters for the file name and three for the extension, including the dot as delimiter there is a maximum of twelve characters.

\textsuperscript{60} For example /DELETE deletes viruses found automatically or /REPORT specifies the file to write the log message to.
5.1.17 Example 17 – Working with different repositories

By means of example 17 it is possible to define several different repositories (and revisions) and checkout only needed files in one directory. After editing files a commit statement updates the files in exactly the repositories they belong to.

First the submitted arguments while executing the script are parsed. The variable `working_path` acts as the main directory for working with the different repositories – there the needed files will be checked out to. The different repositories and revisions to work with and their checkout-paths relative to `working_path` are defined from line 31 to 36 in variables `repositories` and `paths`. The program can handle as many repositories as you want – just extend the list like it is shown in lines 39 and 40. Then the files of each repository are defined (variable `files`) which will be checked out to `working_path` (again there is no limit). If you submit the arguments `checkout` or `update` on script's execution, the repositories (which are saved as keys in variable `paths`) will be checked out (or updated) to the corresponding path defined in `paths`. After that the files from each repository specified in variable `files` will be copied from the directly created (or updated) workspace to the `working_path` directory. There files from different repositories are saved and you can do the modifications you want. After editing the files, running the script with the argument `commit [commit message]` will copy edited files back to the workspace they belong to. Then the changes made to the workspaces are committed (lines 100 to 113). The commit statement used by JavaSVN does not update the local workspace to the latest revision, so eventually

---

61 Checkout of repositories will be done using following path: `<working_path>\_data\<paths [repository_url]>`. 
the working copy has to be updated (only if a new revision was committed).

```rexx
/*
* 17_working-with-different-repositories.rex, 2005-07-09
* Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
*
* Here it is possible to define several different
* repositories (and revisions) and checkout only
* needed files in one directory. After editing
* files a commit statement updates edited files
* in the right repositories they belong to.
* Attention: All files are copied in a specified
* directory with no sub-directories. Therefore
* all files must have different names (otherwise
* files could be committed to wrong repositories).
*
*/
args = arg(1)
parse var args command commit_message

call loadClasses

working_path = "C:\_checkouts\repos"

repositories = .directory-new
paths = .directory-new
files = .directory-new

repositories["http://localhost/repos"] = -1
paths["http://localhost/repos"] = "repos"

repositories["http://localhost/repos2"] = -1
paths["http://localhost/repos2"] = "repos2"

repositories["" ] = -1
paths["" ] = ""

files["http://localhost/repos"] = "anyDirectory\anyFile.txt"
files["http://localhost/repos2"] = "otherDirectory\otherFile.txt"
files["" ] = "" -- and so on

repositories and workspace configuration
.DAVRepositoryFactory~setup
.FSEntryFactory~setup

if command = "checkout" | command = "update" then do
  -- Do a checkout or update for all repositories specified
  say "Begin checkout or update."
  say "---------
  do repos over paths
    path = working_path || "\_data\" || paths[repos]
    workspace = .SVNWorkspaceManager~createWorkspace("file",path)
    location = .SVNRepositoryLocation~parseURL(repos)
    revision = workspace~checkout(location,repositories[repos],.false)
    say "Checked out" repos "revision" revision "to" path || "."
  end
else do
  revision = workspace~update(repositories[repos])
  say "Updated workspace at" path "to revision" revision || "."
end
end
end
-- Copy specified files to working directory ('working_path')
say "Begin copying of files."
```
say "-----------------"
do repos over files  
file = working_path || "\_data" || paths[repos] || "\" || files[repos]  
address cmd "copy " || file || " " || working_path  
say "Copied" file "to" working_path || "."  
end

say "-----------------"
say "Feel free to edit files now!"
say "After editing execute script with arguments 'commit [message]' ."
end
else if command = "commit" then do  
say "Begin committing revisions."
say "-----------------"
do repos over paths  
path = working_path || "\_data" || paths[repos]  
workspace = .SVNWorkspaceManager~createWorkspace("file",path)  
revision = workspace~commit(commit_message)  
update = .false  
if revision = -1 then say "Nothing to commit."
else do  
say "Committed repository" repos "revision" revision || "."  
update = .true  
end
end

if update = .true then do  
say "Updating working copies."
say "-----------------"
do repos over paths  
path = working_path || "\_data" || paths[repos]  
workspace = .SVNWorkspaceManager~createWorkspace("file",path)  
revision = workspace~update(repositories[repos])  
say "Updated workspace at" path "to revision" revision || "."  
end
end
end
else say "Please submit 'checkout', 'update' or 'commit' [message] as an argument."
::requires BSF.CLS  
::requires 99_utils.rex

5.2 Hook scripts

5.2.1 General information

Hook scripts are programs executed by some repository event. Each hook is handed enough information to tell what that event is, what target(s) it is operating on, and the user name of the person who triggered the event. Depending on the hook's output or return status, the
hook program may continue the action, stop it, or suspend it in some way.

Subversion implements the following hooks (excerpt):

- **start-commit** – This is run before the commit transaction is even created.
- **pre-commit** – This is run when the transaction is complete, but before it is committed.
- **post-commit** – This is run after the transaction is committed, and a new revision is created.
- **pre-revprop-change** – This is run before a revision (=unversioned) property is changed.
- **post-revprop-change** – This is run after a revision property has been changed. This script will not run unless the **pre-revprop-change** hook exists. [cp. Coll05, 70 f]

Hook scripts have to be placed in the directory `<path to repository>\hooks` and must be executable by the operating system. For Windows that means the hook must have an extension like `.bat` or `.exe`. Enabling a hook is done by putting an executable file named after the hook into the `hooks` folder (e.g. `pre-commit.bat`). Unfortunately I could not manage Subversion to execute an ooRexx script directly. That is why there is always a batch file (.bat) named after the hook which calls the ooRexx script and submits the arguments. For security reason, the Subversion repository executes a hook script with an empty environment – that means no environmental variables are set at all. For that reason there will be full paths to required and loaded ooRexx scripts.

### 5.2.2 Example 18 – `post-revprop-change` logging modifications

The batch file shown in source code 27 executes an ooRexx script (with four parameters) where the functionality is implemented. In batch files comments are written using `rem` as start of line. The four parameters are: the path to the repository, the revision, the property being set on the revision and the action (property is being 'A'dded, 'M'odified, or 'D'eleted).

```bash
rem post-revprop-change.bat, 2005-07-07
rem Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
```

62 Remember that for the execution `pre-revprop-change` hook must exist (and even if it is only a blank executable file).
Developed examples

rem Wrapper file calling an ooRexx script and passing some arguments
rem because direct execution of .rex seems not working.
rexx C:\_repository\hooks\post-revprop-change-rexx.rex %1 %2 %3 %4

Source code 27: File <path to repository>\hooks\post-revprop-change.bat

The ooRexx script called is shown in source code 28. For each repository the modifications are written line-by-line in a text file named after the repository. An administrator has for each repository the ability to see when a property was modified (or added or deleted). As mentioned earlier paths must always be set in an absolute way (line 19).

Source code 28: File <path to repository>\hooks\post-revprop-change-rexx.rex

Figure 14 shows what a possible log file for a repository having hook post-revprop-change-rexx.rex activated may look like.

Figure 14: Possible log file for a repository
5.2.3 Example 19 – post-commit sending email

Again there is a batch file executing an ooRexx script with the following parameters: the path to the repository and the number of the revision just committed.

```
rem post-commit.bat, 2005-07-09
rem Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
rem Wrapper file calling an ooRexx script and passing some arguments
rem because direct execution of .rex seems not working.
rem rexx C:\_repository\hooks\post-commit-rexx.rex %1 %2
```

The hook script for sending a notification email after a successful commit was made is shown in source code 30. The recipients are set on lines 14 and 15. There are two implementations of sending email: the first using CDO and the second using Microsoft's Outlook (in comments). If we have a look at the first method an OLE-object (a CDO.Message) is instantiated. Then a SMTP server, the port of the server and the server's time out are configured (lines 24 to 36). The message sender, subject and body are defined on lines 39 to 41. After that a loop sends an email to all specified recipients.

The Outlook sending method uses the standard account specified in Outlook for sending emails. A new OLE-object is instantiated (an Outlook.Application) on line 55. A loop is creating the message to send (recipient, subject and body) and sends it at last. In the end Outlook is being closed as shown in line 67.

There are some commented lines, e.g. for attaching a file to the email or sending a (blind) carbon-copy. If you want to use these functions, just uncomment the lines.

```
/*
 * post-commit-rexx.rex, 2005-07-10
 * Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
 * Hook script for sending notification mail after
 * a successful commit was made. Showing implementation
 * of sending mail through CDO and Outlook. Recipients
 * are defined in a stem variable.
 */
```

63 Collaboration Data Objects – The library provides developers with an easy way to create, manipulate, and send Internet messages.
64 There is no sender defined because the default Outlook account is being used, where the sender's name and email address are saved.
Developed examples

```
* */

-- Setting recipients
recipient.0 = 2
recipient.1 = "bernhard.hoisl@wu-wien.ac.at"
recipient.2 = "h0252748@wu-wien.ac.at"
args = arg(1)
repository = args~word(1)
revision = args~word(2)

-- Sending mail using CDO
msg = .OLEObject~new("CDO.Message")
smtpServer = "smtp.chello.at"
smtpServerport = 25
smtpTimeout = 60

-- Configuration
f = msg~configuration~fields
f~Item("http://schemas.microsoft.com/cdo/configuration/sendusing")~value = 2
f~Item("http://schemas.microsoft.com/cdo/configuration/smtpserver")~value = smtpServer
f~Item("http://schemas.microsoft.com/cdo/configuration/smtpserverport")~value = smtpServerport
f~Item("http://schemas.microsoft.com/cdo/configuration/smtpconnectiontimeout")~value = smtpTimeout
f~update

-- Generate message
msg~from = "Bernhard Hoisl" "<bernhard.hoisl@wu-wien.ac.at>"
msg~subject = "Commit notification from" repository
msg~textbody = "There is a new revision" revision "on repository" repository "from" date () time() || "."

-- Loop over all recipients
do i = 1 to recipient.0
   msg~to = recipient.i
   -- msg~cc = "" -- If needed
   -- msg~bcc = "" -- If needed
   msg~send
end

-- Sending mail using Outlook
/*
outlook = .OLEObject~new('Outlook.Application')
do i = 1 to recipient.0
   mail = outlook~CreateItem(0)
   mail-To = recipient.i
   mail-Subject = "Commit notification from" repository
   mail-Body = "There is a new revision" revision "on repository" repository "from" date () time() || "."
   -- mail-Attachments~Add(); -- If needed
   mail-Send
end
outlook~Quit
*/
```

Source code 30: File <path to repository>\hooks\post-commit-rexx.rex

A small modification of the script is shown in source code 31. This time the script contacts a repository to find out to whom sending a notification email is necessary. In the root directory of the repository properties beginning with notification\(^65\) (like notification1, notification2 etc.) have to be set. The value is the email address to which a notification has

\(^{65}\) Or whatever you set variable property to (line 25).
to be sent if a new commit occurs. The sending method of the email uses CDO. The loop over the properties found is shown on lines 63 to 74. If the trimmed property name found equals the value of variable property, an email is sent to the recipient defined as the property’s value. Again no environmental variables are set and full paths have to be used (lines 77 to 83).

```plaintext
/*
 * post-commit-rexx2.rex, 2005-07-10
 * Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
 * 
 * Hook script for sending notification mail after
 * a successful commit was made. This is another
 * implementation which slightly differs from
 * 'post-commit-rexx.rex'. This time the script
 * contacts the repository to find out to whom
 * sending a notification email is necessary. In
 * the root directory of the repository there has
 * to be set properties beginning with
 * 'notification' (like 'notification1',
 * 'notification2' etc.). The value is the email
 * address to which a notification has to be sent
 * if a new commit occurs.
 * 
 */

call loadClasses
repos = "http://localhost/repos"
path = ""
-- Beginning chars to find property
property = "notification"
-- Repository configuration
.DAVRepositoryFactory~setup
location = .SVNRepositoryLocation~parseURL(repos)
repository = .SVNRepositoryFactory~create(location)
revision = repository~getLatestRevision
-- Mail configuration
msg = .OLEObject~new("CDO.Message")
smtpServer = "smtp.chello.at"
smtpServerport = 25
smtpTimeout = 60
f = msg~configuration~fields
f~Item("http://schemas.microsoft.com/cdo/configuration/sendusing")~value = 2
f~Item("http://schemas.microsoft.com/cdo/configuration/smtpserver")~value = smtpServer
f~item("http://schemas.microsoft.com/cdo/configuration/smtpserverport")~value = smtpServerport
f~Item("http://schemas.microsoft.com/cdo/configuration/smtpconnectiontimeout")~value = smtpTimeout
f~update
-- Generate message
msg~from = "Bernhard Hoisl" "bernhard.hoisl@wu-wien.ac.at>"
msg~subject = "Commit notification from" repos
msg~textbody = "There is a new revision" revision "on repository" repos "from" date()
time() || " "."
-- msg~addAttachment("") -- If attaching files is needed
m = .HashMap~new
v = .Vector~new
-- Get property list from repository's root directory
c = repository~getDir(path,revision,m,v);
i = m~keySet~iterator
do while i~hasNext
```
5.3 Example using Apache with Mod_Rexx

5.3.1 Example 20 – Repository listing

Example 20 is doing the same as example 7 – showing a list of files and directories of a repository at a specific revision. This time I am making use of the web-server Apache for handling HTTP requests. The module Mod_Rexx integrates ooRexx support for Apache.

As you can see in source code 32 there is an RSP printing an HTML form (lines 23 to 39). If a user submits this form, transmitted variables are saved in the stem variable `WWWARGS` (lines 28 to 31). If the form has been submitted and there has been an entry for the input field `repository`, an ooRexx server is called (with the parameters `repository` and `revision`) which handles the connection to the Subversion repository (see source code 33). An HTML formatted text string will return from the `server.rex` script which is printed out (line 65). So this is just another way of printing the directory entries of a repository, graphically shown in figure 15.
The user contacts the web-server using a web-browser. By submitting the HTML form the RSP contacts the ooRexx server (both must be configured for working together – IP address and port number). The running ooRexx server contacts Subversion's repository, retrieves the information and generates an HTML formatted string which is returned to the RSP. There the string is sent to the client and displayed in his web-browser.

The reason why we do not contact Subversion's repository out of an RSP is that in my tests it was simply not possible. As I mentioned before in chapter 3.2 BSF4Rexx cannot be loaded from an RSP or an ooRexx script executed within an Apache process.

```rexx
<?rexx
/*
* index.rsp, 2005-06-22
* Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
* *
* This script has two functions: (1) Handle HTTP requests and presents data in HTML in a browser (GUI). (2) Perform as a client, which connects to the server instantiated by 'server.rex'. Submits repository's URL and revision (optional) as arguments. Expects a string as return result.
* */
<?
<html>
<head>
<title>List repository</title>
</head>
<body>
<form action=index.rsp method=post>
<?rexx
repository = ""
```
If we take a closer look at source code 33, we will not see anything new. The ooRexx socket functions are loaded (lines 14 to 17), the program listens on a defined IP address and port 1508 (configured on lines 21 and 25) and the server waits for a client to connect (line 33). If a connection has been established, the parameters are parsed (line 39) and a routine list-Files is called. This routine we already knew generates a list consisting of every directory entry in the defined repository. The produced string is then returned to the client (the RSP) in line 43.
if RxFuncQuery("SockLoadFuncs") then do
  call RxFuncAdd "SockLoadFuncs","RXSOCK","SockLoadFuncs"
call SockLoadFuncs
end
socket = SockSocket("AF_INET", "SOCK_STREAM", "0")
host.!addr = SockGetHostId()
host.!family = "AF_INET"
host.!port = 1508
call SockBind Socket, "host."
call SockListen Socket, 1
do forever
  say "------------------------"
say "Waiting at" host.!addr || ":" || host.!port "for a client to connect..."
call SockAccept(Socket)
say "Client has established connection."
call SockRecv ClientSocket, "InpString", 256
say "String read from client: " || InpString || ""
parse var InpString repository "," revision
OutString = listFiles(repository,revision)
call SockSend ClientSocket, OutString
say "An answer was send back to the client (" || OutString~-length "bytes)..."
call SockShutDown ClientSocket, 2
call SockClose ClientSocket
say "Client connection closed."
end
::requires BSF.CLS
::routine listFiles
use arg repos,revision
call loadClasses
-- Repository configuration
.DAVRepositoryFactory~setup
location = .SVNRepositoryLocation~parseURL(repos)
repository = .SVNRepositoryFactory~create(location)
rev = repository~getLatestRevision
if revision = "" | revision>rev then revision = rev
-- Recursive call in routine for building directory tree
tree = getTree(repository,"",revision,"")
-- Creating html output
out = "<h2>
out = "<table width = 100%><tr><th align = left>Path</th><th align = left>CreationDate</th><th align = left>Author</th><th align = left>LastModifiedRevision</th><th align = left>Size (byte)</th></tr>
out = "</table>"
return out
-- Routine for building directory tree
::routine getTree
use arg repository,path,revision,tree
m = .HashMap~new
v = Vector~new
-- Don't need 'm' and 'v' but have to be declared
c = repository~getDir(path,revision,m,v);
i = c~iterator
do while i~hasNext
  dir = i~next
  dirName = dir~getName
  dirKind = dir~getKind~toString
  dirPath = path || dirName
  dirCreationDate = dir~getDate~toString
  dirAuthor = dir~getAuthor
  dirLastModifiedRevision = dir~getRevision
  dirSize = dir~size
  if dirAuthor = .Nil then dirAuthor = "[no author]"
  -- Generates html output string
  tree = tree "<tr><td>"dirPath"</td>
  tree = tree "<td>"dirCreationDate"</td>
  tree = tree "<td>"dirAuthor"</td>
  tree = tree "<td>"dirLastModifiedRevision"</td>
  tree = tree "<td>"dirSize"</td></tr>"
  -- Recursive call
  if dirKind = "<dir>" then tree = getTree(repository,path||dirName||"/",revision,tree)
end
return tree
::routine loadClasses public
  classes = "org.tmatesoft.svn.core.internal.io.dav.DAVRepositoryFactory" ,
            "org.tmatesoft.svn.core.io.svn.SVNRepositoryFactoryImpl" ,
            "org.tmatesoft.svn.core.io.svn.SVNRepositoryLocation" ,
            "org.tmatesoft.svn.core.io.SVNRepositoryFactory" ,
            "org.tmatesoft.svn.core.io.SVNDirEntry" ,
            "org.tmatesoft.svn.core.io.SVNNodeKind" ,
            "org.tmatesoft.svn.core.io.SVNLogEntry" ,
            "org.tmatesoft.svn.core.IsvnWorkspace" ,
            "org.tmatesoft.svn.core.I svnWorkspaceManager" ,
            "org.tmatesoft.svn.core.ISVNRepositoryFactory" ,
            "org.tigris.subversion.javahl.SVNClient" ,
            "java.util.HashMap" ,
            "java.util.Vector" ,
            "java.io.File"
do i = 1 to words(classes)
  class = word(classes,i)
  rexxClass = substr(class,1,length(class)-1)
  .bsf-bsf.import(class, rexxClass)
end

Source code 33: File <path to Apache>htdocs<any sub-directory>\server.rex

5.4 Example using Tomcat

5.4.1 Example 21 – Subversion HTTP manager

The Subversion HTTP manager is a substitute for Subversion's build-in client. As a user you do not have to install Subversion because all of the communication is managed by using only a web-browser.

By means of the Subversion HTTP manager you can checkout, update, commit and edit
files using your preferred web-browser. First you have to specify a repository and (if re-
quired) authenticate yourself by submitting your user name and password. By doing a
checkout a workspace is created in a defined directory on the server side. Then it is possible
to list and edit files. The file to edit is loaded and displayed in an HTML textarea whose
content is saved back to the file. After modifications are made you can commit your revi-
sion and add a commit message\textsuperscript{66}. Update of the server sided workspace is done using the
update link.

Using Tomcat as web-server a Java Servlet calls an ooRexx script which interacts with Sub-
version (using Java). The ooRexx script returns a code which is implemented in the Java
Servlet. Hence generated HTML code is sent to the client's web-browser where it is dis-
played.

This method can be a little confusing because we do not really need ooRexx here. There is
the qualified question why not interact with Subversion using Java only. The reason is that I
wanted to prove that Java can use ooRexx and ooRexx Java. Java Servlets are calling
ooRexx scripts and ooRexx scripts are using Java to interact with Subversion. The de-
scribed communication is best shown in figure 16 below.

\textbf{Figure 16: Communication schema of Subversion HTTP
manager}

\textsuperscript{66} As I presented the Subversion HTTP manager in a course at university, there was also a function that
notifies a student by sending a SMS that a new revision was committed. That function used the in-house
SMS gateway and therefore I do not distribute it here with my examples.
As there is too much source code for displaying all interacting files, I will only show two examples which represent the communication between a Java Servlet and an ooRexx script.

Source code 34 below displays a Java Servlet importing needed classes and generating HTML output. The call to the ooRexx script is done on line 63 and returned string is printed out in the next line.

```
/*
* Update.java, 2005-06-21
* Bernhard Hoisl (bernhard.hoisl@wu-wien.ac.at)
* * Handles HTTP-requests and generates the HTML output.
* * Calls 'update.rex' with repository URL, user and
* * password as arguments. Gets back the status code.
* */

import org.tmatesoft.svn.core.internal.io.dav.DAVRepositoryFactory;
import org.tmatesoft.svn.core.internal.io.svn.SVNRepositoryFactoryImpl;
import org.tmatesoft.svn.core.internal.ws.fs.FSEntryFactory;
import org.tmatesoft.svn.core.io.*;
import org.tmatesoft.svn.core.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import org.apache.bsf.*;
import org.rexxla.bsf.engines.rexx.*;
import org.apache.bsf.util.BSFEngineImpl;

public class Update extends HttpServlet {
  public void processRequest(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html; charset=ISO-8859-1");
    PrintWriter out = response.getWriter();
    HttpSession session = request.getSession(true);
    String repository = session.getAttribute("repository").toString();
    String user = session.getAttribute("user").toString();
    String pw = session.getAttribute("pw").toString();
    try {
      out.println("<html>\n      <head>");
      out.println("<title>Subversion HTTP Manager</title>\n      </head>\n      <body>\n      <table width=100% height=100% cellspacing=0 cellpadding=0
      border=0>\n      <tr><td valign=center align=center>
      String pathToHere = "C:/Programme/Apache Group/Tomcat/webapps/subversion/WEB-INF/classes/";
      BufferedReader in = new BufferedReader(new FileReader(pathToHere+"update.rex"));
      String rexxScript = "";
      String thisLine = "";
      while (((thisLine = in.readLine()) != null) {
        rexxScript += thisLine::*
      }
      BSFManager mgr = new BSFManager();
      Vector v1 = new Vector();
      Vector v2 = new Vector();
      v2.add(0,repository);
      v2.add(1,user);
      v2.add(2,pw);
```
The corresponding ooRexx script is shown below. Submitted arguments are retrieved (lines 13 to 15). If the workspace exists (line 23) the normal Subversion configuration is done and the workspace is updated (line 30). A string defined in line 31 is returned to the Java Servlet (where it is displayed). As you can see in lines 17 and 18 you have to set variable path in an absolute way.

```rexx
repos = arg(1)
user = arg(2)
pw = arg(3)
path = "C:/Programme/Apache Group/Tomcat/webapps/subversion/WEB-INF/classes/data/"||repos~substr(8,repos~length-7)~reverse~changestr("/",".")
check = SysGetFileDateTime(path,"C")
if check = -1 then return repos "doesn't exist"
else do
   .DAVRepositoryFactory~setup
   .FSEntryFactory~setup
   workspace = .SVNWorkspaceManager~createWorkspace("file",path)
   location = .SVNRepositoryLocation~parseURL(repos)
   workspace~setCredentials(user,pw);
   head = .bsf~bsf.getStaticValue("org.tmatesoft.svn.core.ISVNWorkspace", "HEAD")
   revision = workspace~update(head)
   return "Updated to revision" revision
```

Source code 34: File <path to Tomcat>/webapps/<application name>/subversion/WEB-INF/classes/Update.java
```java
end
::requires BSF.CLS
::routine loadClasses public
  classes = "org.tmatesoft.svn.core.internal.io.dav.DAVRepositoryFactory",
           "org.tmatesoft.svn.core.internal.io.svn.SVNRepositoryFactoryImpl",
           "org.tmatesoft.svn.core.internal.io.svn.SVNRepositoryFactoryImpl",
           "org.tmatesoft.svn.core.SVNRepositoryLocation",
           "org.tmatesoft.svn.core.SVNRepositoryFactory",
           "org.tmatesoft.svn.core.ISVNWorkspace",
           "org.tmatesoft.svn.core.SVNWorkspaceManager",
           "org.tmatesoft.svn.core.SVNWorkspaceManager",
           "org.tmatesoft.svn.util.SVNUtil",
           "org.tigris.subversion.javahl.SVNClient",
           "java.io.File"
  do i = 1 to words(classes)
    class = word(classes,i)
    rexxClass = substr(class, lastpos('.', class) + 1)
    .bsf-.bsf.import(class, rexxClass)
end
```

Source code 35: File `<path to Tomcat>\webapps\<application name>\subversion\WEB-INF\classes\update.rex`
6 Conclusion

In this paper I wanted to give a detailed view of how to automate Subversion using ooRexx. Beginning with the technical requirements and an installation guide I discussed in-depth the concepts of Subversion automation. The creation of useful examples were substantial for the understanding of my work.

As to be responsive to my research questions defined at the beginning of this paper I can declare question one and two to be answered positively.

It is definitely possible to automate Subversion using ooRexx – this question was answered in chapter 4. From my point of view this automation can certainly be used to generate an advantage in comparison to Subversion's build-in functions. Chapter 5 shows many examples how the automation can be used in different ways. In the end every reader has to decide on his own if the examples provided are useful or not. Anyway, my examples are declared to be nutshell-programs, which means that not every imaginable functionality could be implemented. Much more important is that the reader understands the concept because then he has the opportunity to implement or improve functions by himself.

I had not used Subversion or any version control system before starting with the work on this paper. Therefore it was a great learning experience for me to get to know such a type of software. My Subversion automation examples were definitely influenced by my interest in the development of Internet applications.

If anybody has questions regarding my work, contact me by using the email address on the cover.

6.1 Further work

As mentioned before, in this last chapter I would like to give you a brief overview of further work whose realization could be worth noting.

In chapter 4.1 I claimed that Subversion is very good at managing data, but not designed to
handle collaborative work. The weak point can be diminished by providing on-time information about a user's currently done work. If we think of displaying such information using an ooRexx script, I suggest running a server which can handle multiple clients. Every project member has to invoke an ooRexx script when he begins to work with his local files. The script controls the working directory and notices the server of every change made. Not only changes should be monitored, also, for example, if a file is opened and for how long. The server has to collect the information and display it, for example, as a web-page or send data back to the client (where a script has to handle it).

Concerning the use of Subversion collaboratively there must be a central server. That is why probably the directly described method should be the best. Another solution might be not to run a central information server, but to have all clients acting as agents. That means that every agent has the ability to find other agents in a defined namespace over a network connection and communicates with them interactively. Every agent has the same methods implemented and can respond to a request as well as request a resource by itself. Then there will be no need for a central server anymore.

Both implementation methods can have several extensions, e.g. a communication module for users to speak to each other to solve problems. If a user modifies a file and another one wants to do the same, there should be a warning message and the user should have the possibility to communicate with the other one. To a certain level it is also possible that such a process of interaction can be done automatically. For example, if it is defined that one user has stronger privileges on a file than another one. Then the user with the stronger privileges should have access to the file all the time (maybe with exceptions).

Subversion uses a copy-modify-merge model, that is why there should be no lock status for a file as described in the previous paragraph. Of course you are able to lock a file but that is not the developer's philosophy. So maybe an information broker as characterised above would be a really interesting and useful program. This idea could be realized in the near future or it might be an impulse for developing your own examples.
Bibliography


