Projekt AN.ON
Hannes Federrath, Universität Regensburg
Stefan Köpsell, TU Dresden

Part I

Development Environment

Development Environment
Kuno G. Grün
Rolf Wendolsky
Derek Daniel
Elmar Schraml
Georg Koppen
Stefan Köpsell
Version: 4.10
from: 28. August 2013

1 Development Environment

1.1 Summary

This document describes how to set up and use the recommended development environment for the AN.ON project.

Table of Contents

I Development Environment 3

1 Development Environment **7**

1.1 Summary . . . 7
1.2 Introduction . . . 11
1.3 Components . . . 12
1.3.1 Overview . . . 12
1.3.2 Applications . . . 12
1.3.3 Core Components . . . 14
1.3.4 External libraries . . . 14
1.3.5 Java library installation . . . 15
1.3.6 C++ library installation on Linux . . . 15
1.3.7 C++ library installation on Windows . . . 15
1.4 Compiler . . . 19
1.4.1 C++ . . . 19
1.4.2 Java . . . 19
1.5 IDEs . . . 19
1.5.1 Specific development requirements on Microsoft Windows . . . 19
1.5.2 Eclipse IDE . . . 21
1.5.3 Borland JBuilder IDE . . . 24
1.6 Doxygen . . . 26
1.6.1 Download . . . 26
1.6.2 Installation . . . 26
1.7 Unittest environments . . . 26
1.7.1 JUnit . . . 26
1.7.2 CppUnit . . . 26
1.8 Running the tests . . . 26
1.8.1 Architecture . . . 26
1.8.2 Configuration . . . 27
1.8.3 Start . . . 28
1.8.4 Test . . . 29
1.9 Payment Instance Setup . . . 29
1.9.1 System requirements: . . . 29
1.9.2 You will need: . . . 29
1.9.3 Creating the Jar file . . . 29
1.9.4 Creating the key file and certificate . . . 30
1.9.5 Creating the Postgres database . . . 30
1.9.6 Creating the JPI configuration . . . 30
1.9.7 Starting the JPI . . . 31
1.10 Accounting Instance Setup . . . 31
1.10.1 System Requirements . . . 31
1.10.2 You will need . . . 32
1.10.3 Compiling the first Mix . . . 32
1.10.4 Setting up the database . . . 32
1.10.5 Payment Configuration . . . 33
1.10.6 Price Certificates . . . 33

1.11 Payment Instance GUI (PIG) setup . . . 34
1.11.1 Ruby . . . 34
1.11.2 Rails . . . 34
1.11.3 Database configuration . . . 34
1.11.4 Plugins . . . 34
1.11.5 Java Bridge . . . 35
1.11.6 Server . . . 35
1.12 Bibliography . . . 35

2 Release Procedures **37**
2.1 JAP . . . 37
2.1.1 Overview . . . 37
2.1.2 Libraries . . . 37
2.1.3 Source preprocessing . . . 38
2.1.4 Build JAP.jar . . . 38
2.1.5 Deploy JAP.jar . . . 38
2.1.6 Build Windows installers . . . 38
2.1.7 Deploy Windows installers . . . 38
2.1.8 Build OSX install package . . . 38
2.1.9 Deploy OSX install package . . . 39
2.1.10 Sign installer packages . . . 39
2.1.11 Deploy to SourceFroge . . . 39
2.2 JonDo und JonDoPortable . . . 39
2.2.1 JonDo . . . 39
2.2.2 JonDoPortable . . . 39
2.3 JonDoFox Extension . . . 40
2.4 JonDoFox Profile . . . 40
2.5 JonDoBrowser . . . 42
2.5.1 Build-Setup . . . 42
2.5.2 Release . . . 43

3 Document history **45**

II Programmierrichtlinien AN.ON 47
3.1 Introduction . . . 52
3.2 Primary Goals . . . 53
3.2.1 Security . . . 53
3.2.2 Compatibility . . . 53
3.2.3 Performance . . . 54
3.2.4 Quality . . . 54
3.3 Structure . . . 55
3.3.1 Package Structure . . . 55
3.3.2 Makefile – Adding Classes (C++) . . . 56
3.3.3 Code Structure . . . 56
3.3.4 Classes- / Interface Structure . . . 57
3.3.5 Methods . . . 58
3.4 Libraries . . . 59
3.4.1 Internationalization (JAP/MixConfig) . . . 59
3.4.2 Logging . . . 59
3.4.3 XML (Java) . . . 60
3.4.4 Images (JAP/MixConfig) . . . 60
3.4.5 Other Resources (JAP/InfoService/MixConfig) . . . 60
3.4.6 External Libraries . . . 60
3.5 Naming Conventions . . . 61
3.5.1 Constants . . . 61

3.5.2 Variables . . . 61
3.5.3 Methods . . . 61
3.5.4 Classes / Interfaces . . . 62
3.5.5 Message Properties (JAP/MixConfig) . . . 62
3.5.6 Image File Names (JAP/MixConfig) . . . 62
3.5.7 Test Classes . . . 63
3.5.8 Class File Name Extensions . . . 63
3.6 Type Conventions . . . 64
3.6.1 Simple Data Types (C++) . . . 64
3.6.2 Constants (Java) . . . 64
3.6.3 Variables . . . 64
3.6.4 Methods . . . 65
3.6.5 Classes / Interfaces . . . 65
3.6.6 Threads . . . 65
3.6.7 Exceptions . . . 66
3.6.8 HTML-Formated GUI Elements (JAP/MixConfig) . . . 66
3.7 Documentation . . . 67
3.7.1 Documentation Tool . . . 67
3.7.2 Procedure . . . 67
3.7.3 Classes . . . 67
3.7.4 Attribute (@param Tag) . . . 67
3.7.5 Return Values (@return Tag) . . . 67
3.7.6 Methods . . . 68
3.7.7 Algorithms . . . 68
3.7.8 @todo Tag . . . 68
3.8 Other Conventions . . . 69
3.8.1 Code Style . . . 69
3.8.2 Coding Conventions . . . 70
3.9 Testing . . . 71
3.9.1 Introduction to Unittests . . . 71
3.9.2 JUnit . . . 72
3.9.3 CppUnit . . . 74
3.9.4 Dummy and Mock Objects . . . 75
3.10 Literature Cited . . . 76
3.11 Appendix (Summary) . . . 77

1.2 Introduction

This document is an introduction to the development tools and code base of the AN.ON project. We recommend some IDEs and give some help on their installation and usage. Configuration files for the IDEs are kept up to date by our project maintainer. The proposed IDEs are platform neutral and can be used with Windows, Linux and Mac OS X. AN.ON itself is split into Java and C++ components that can be compiled and run on any of the mentioned systems.

The Java builds are based on Maven. We use our own maven repository which can be accessed here: https://anon.inf.tu-dresden.de/artifactory/. In general Eclipse is recommended. The project settings for C++ are based on the Unix-native GNU make in combination with CDT 9.2.0. Moreover we use the autoconf tools for C++, so in principle every editor would do. Nevertheless for development of the C++ components under Microsoft Windows we recommend MS Visual Studio (the Express version is sufficient). If you use Mac OS X we recommend XCode for the C++ development.

Once again: you are welcome to develop with any IDE you like. Just keep in mind that you may have to invest some time for generating the project settings.

We use subversion as version control system. Each component has its own subversion repository. The most recent code can be checked out using the following general URL scheme, where COMPONENT has to be substituted by the name of the component you want to check out:

https://anon.inf.tu-dresden.de/svn/COMPONENT/COMPONENT/trunk

The use Jenkins as continuous integration tool. The results of the continuous integration can be seen here: https://anon.inf.tu-dresden.de/jenkins/. New builds will be automatically triggered as soon as new source code appears in the corresponding repository. If you are responsible for the latest changes you should ensure that the compilation and tests will not fail.

1.3 Components

1.3.1 Overview

The components can be distinguished into: *applications*, i.e programs run by (end) users, *core components* and (external) *libraries*. Figure 1.1 depicts the various Java applications and components and their dependency. Figure 1.2 depicts the dependency for the Mix written in C++.

1.3.2 Applications

Mix (Subversion-module: proxytest)

Mix is the virtual core of AN.ON. With a Mix, the process of anonymisation is done. Although the Subverion component's name is proxytest (due to historic reasons), the executable program is named `mix`. Since this is a development documentation, we will normally refer to it as proxytest. Due to performance reasons, proxytest is written in C++.

AN.ON VPN Server (Subversion-module: ANONVPNServer)

The AN.ON VPN Server is the server-side part (proxy) for realising the VPN-like anonymisation solutions utilised especially in the area of mobile devices (smart phones). As it interacts quite closely with the operating system the VPN server is written in C++.

JAP client (Subverison-module: Jap)

The *JAP* client is used as local proxy that manages the communication with Mix cascades and InfoServices. The client has to run on as many platforms as possible, so it is written as a Java application. It uses an old 1.1.8 Java runtime environment for this reason as well. This runtime version is also available on older operating systems and makes porting to new environments (e.g. of SmartPhones, SetTop boxes etc.) more easy.

Android VPN-based client (Subverison-module: ANONdroidVPN)

The *ANONdroidVPN* client is a client for the Android operating system. It acts as a VPN client and is thus able to anonymise all Apps installed on the smart phone.

JondoConsole client (Subverison-module: JondoConsole)

This is a command line client for the AN.ON system, i.e. it offers the same functionality as JAP – but without a GUI.

InfoService (Subversion-module: InfoService)

The *InfoService* is needed to register new Mixes and to connect JAP to existing Mix cascades. It also enables JAP to find the IP addresses of active Mix cascades. InfoService is also written completely in Java.

Payment Instance (CVS-module: Jap)

The *Payment Instance* administrates user accounts for the AN.ON payment system.

MixConfig tool (Subversion-module: MixConfig)

With this application, a configuration file for a Mix can be created and maintained. It is also used to configure the Mix-on-CD live-CD. Again, this tool is written as a Java application.

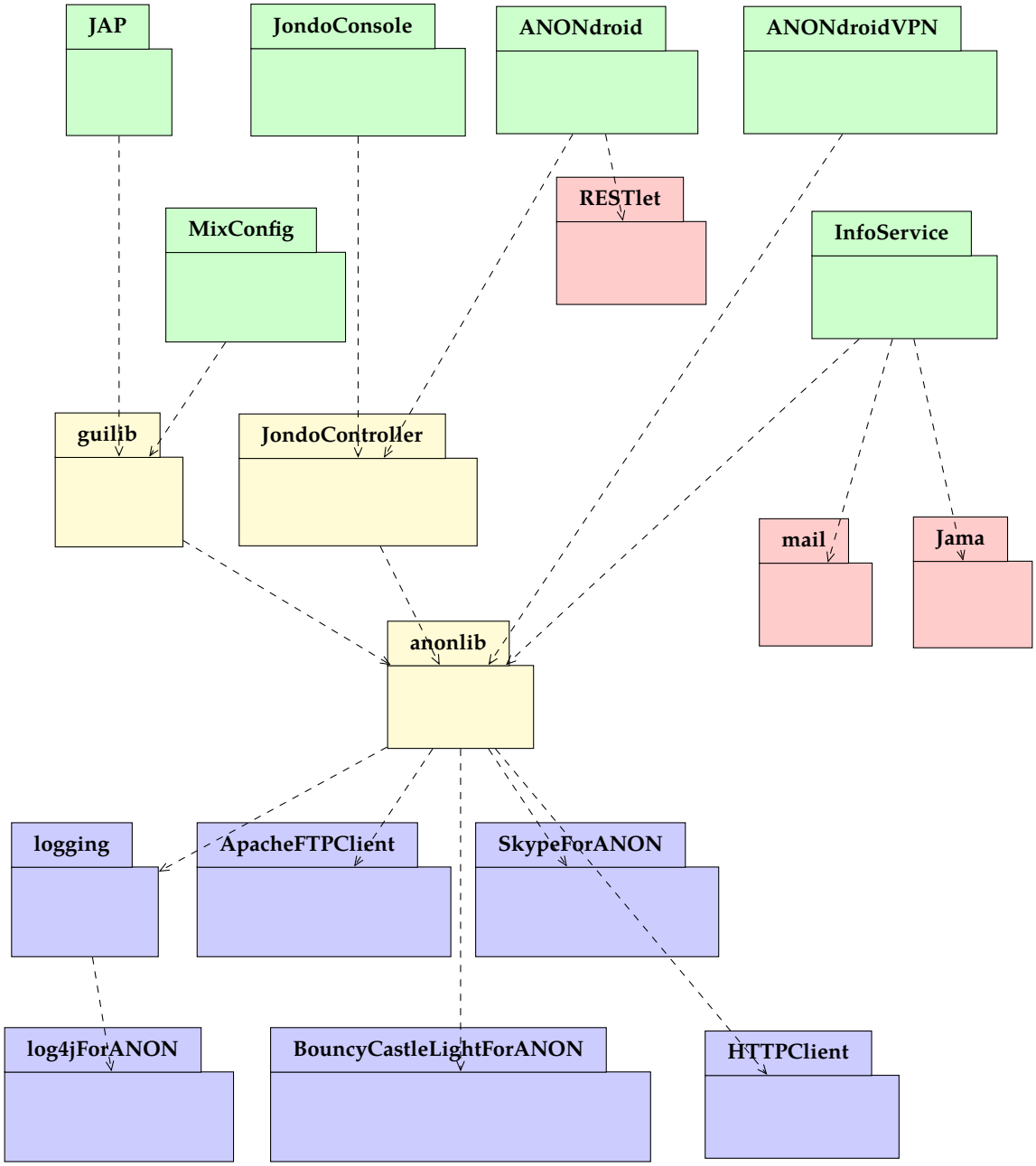

Figure 1.1: Diagram showing Java applications, core components, unmodified external and tailored "external" libraries.

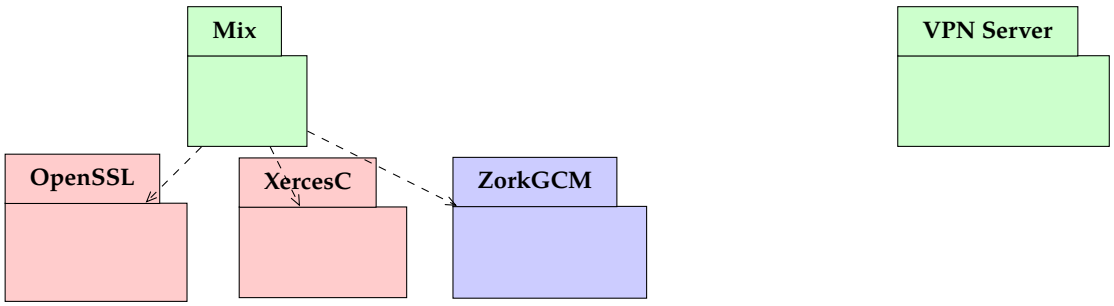

Figure 1.2: Diagram showing C++ applications, core components, unmodified external and tailored "external" libraries.

Java Mix (CVS-module: JavaMix)

An experimental Mix implementation.

1.3.3 Core Components

AN.ON library (Subversion-module: anonlib)

This Java library contains all classes that are needed by JAP, Infoservice and MixConfig. It implements the core of the AN.ON protocols.

Changes to this central library potentially affect the other components and have to be tested with each of them.

JonDo Controller library (Subversion-module: JonDoController)

This library builds on top of the anonlib and abstracts the functionality of the anonlib to give programmers a more easy way of accessing the functionality of the anonlib.

GUI library (Subversion-module: guilib)

These utility library contains helper components and classes used by JAP and MixConfig for the graphical user interface.

1.3.4 External libraries

Each of the mentioned components needs some "external" libraries. The term "external" reflects the fact, that some of these libraries are implemented by third parties. Nevertheless the AN.ON project uses customised versions tailored to the needs of the AN.ON project. Therefore most of these "external" libraries have their own AN.ON based subversion repository. This allows to build the whole AN.ON system from a defined set of source code.

Name	**S**ubversion module	**u**sed for
Java:		
ApacheFTPClient	ApacheFTPClient.jar	FTP-client libraries
ApacheBZIP2	ApacheBzip2.jar	Bzip2 libraries
Apache XML-RPC	apache-xmlrpc-1.1.jar	Apache XML-RPC libraries
BouncyCastleLightForJAP	BouncyCastleLightForJAP.jar	crypto libraries
BouncyCastleLightForMixConfig	BouncyCastleLightForMixConfig.jar	crypto libraries
http	http.jar	HTTP-Client libraries
Jama	Jama.jar	matrices libraries
JavaBeans Activation Framework	activation.jar	needed by JavaMail
JavaMail	mail.jar	blocking resistance
JAI 1.1.2	jai_core.jar	blocking resistance
junitx	junitx-5.1.jar	Unittests for protected and private members
Log4j	log4j.jar	Logging library
mac	MRJClasses.zip	MRJ library for Macintosh
Swing	swingall.jar	exclusively for Java 1.1; installed in Java 1.2
XML-1.0	xml.jarXML related functions	XML related functions
XML-1.1	xml-1.1.jar	XML related functions
XML-2.4	xml-2.4_min.jar	XML related functions
C++:		
cppunit	>= cppunit-1.8.0	Unittests
openssl	openssl-0.9.7e	Data encryption
pthreads	pthreads	Thread programming
xerces	Xerces-c2_5_0	XML-Parser

1.3.5 Java library installation

You may download the Java libraries from the [ANON] homepage and copy them to any directory you want. We recommend the directory name `ext_lib_jap`.

Since Java is platform independent, you can use the same libraries in both Windows and Linux. You should download the libraries needed by JAP from the [ANON] homepage. If the libraries are not available at the [ANON] homepage or you need newer ones, you must take care of project settings and library names when updating a library. We recommend to put all needed Java libraries together into one directory, for example `libs_java`. You will need to add this directory and the libraries, respective, to the classpath variable of your IDE (described later).

1.3.6 C++ library installation on Linux

Please use the package manager of your Linux distribution to install the needed C++ libraries. You will need the following C++ libraries in the above mentioned versions. Be sure to get the right ones for your distribution. (The following are the package names from SuSE 9.3. Other distributions may differ in the package names.)

- cppunit
- cppunit-devel
- openssl
- openssl-devel
- xerces-c
- xerces-c-devel

Also be aware that your distributor may deliver some buggy packages. If the package does not work, you will have to get the sources and compile it on your own. To do so, you can normally follow the instructions included with the source code. We mention this because we experienced some problems with OpenSSL and SuSE 9.1.

1.3.7 C++ library installation on Windows

Using Windows, we recommend Eclipse running on Cygwin as the development platform for C++. Microsoft Visual Studio .NET 2003 will also do. If you decide to use Eclipse for C++ development, the following steps do not apply to you, because you can use the Cygwin package manager to install them.

Xerces-C

Due to some changes with the namespaces in Xerces-C 2.6.0, a new proxytest version needs to be compiled with a Xerces-C source version lower than 2.6.0. A configuration with Xerces-C 2.5.0 has been tested and works properly. To get a running version of the Xerces-C 2.5.0 libraries you have to follow these steps:

- Since Xerces-C 2.5.0 is no longer the current version, you will need to go to:

http://archive.apache.org/dist/xml/xerces-c/Xerces-C_2_5_0/
and download
xerces-c-src_2_5_0.zip

- Unpack the downloaded archive to any directory; we recommend for example: `d:\coding\libsc\xerces-c-src_2_5_0`

This chosen directory will be referenced as:
[xerces_home]

- Start Visual Studio 2003 .NET and, by selecting

File -> Open -> Project, open the file:
[xerces_home]\Projects\Win32\VC7\xerces-all\xerces-all.sln
If asked to convert the files, you should answer with '**Yes**'.

- Make sure that '**Debug**' is chosen as the '**Solution Configuration**'
- Start building the binaries with: **Build -> Build Solution** (or: **Ctrl+Shift+B**)

Compilation and building will take a while.

- Now choose a '**Release**' build in the '**Solution Configuration**'
- Once again:

Build -> Build Solution (or: **Ctrl+Shift+B**)
Compilation and building will again take a while.

- Depending on the chosen build, you will find the executables in the subdirectories: `[xerces_home]\Build\Win32\` `[xerces_home]\Build\Win32\VC7\Release`
- Add the following environment variables to your system:

`INCLUDE_XERCES_C = [xerces_home]\src`
LIB_XERCES_C_DEBUG = [xerces_home]\Build\Win32\VC7\Debug
LIB_XERCES_C_RELEASE = [xerces_home]\Build\Win32\VC7\Release

OpenSSL

Proxytest needs an OpenSSL version 0.9.7e. Testing caused some problems with other versions. To get a running version of the OpenSSL 0.9.7e, you will have to compile OpenSSL yourself.

Since the config files are generated by a Perl script, you first have to get a running Perl environment onto your Windows system.

There is a free distribution from http://www.activestate.com/, which is simple to set up and will suffice for our purposes. To get Perl running, just go to the above mentioned website and look for a download link to **ActivePerl 5.8.7** or newer. Activestate asks you to register but registration is not mandatory. Proceed to the download with the '**NEXT**' button and then choose a binary package with either the MSI installer or a zipped package. You can keep the default options given by the installation procedure. After installation, do not forget to add the path of the `\bin` subdirectory of your Perl installation to your `PATH` environment variable! Unfortunately, this is not done automatically during the installation. Indeed, it is necessary to be able to use the Perl interpreter from anywhere.

- To get a source distribution of OpenSSL, go to:
- Unpack this archive into any directory, for example: `d:\coding\libsc\openssl-0.9.7e`

The chosen directory will be referenced as: `[openssl_home]`

- Open a command prompt window ("DOS box") and change into this directory
- Automatic generation of the config file is initiated by executing:

Perl Configure VC-WIN32

- For the following step, it is necessary that you have the ml.exe Microsoft Macro Assembler in your path. Although this should have occurred automatically upon installation of Visual Studio 2003, it sometimes causes problems. So, if the following commandcauses error messages, you can fix the problem by executing the batch file:

C:\Programs\Microsoft Visual Studio .NET 2003 \Common7\Tools\vcvars32.bat
This batch file will set all environment variables properly so that VS2003 works again. Be sure to call this batch file from the same command prompt window! Environment variables set from a command prompt window are only set for that
command prompt instance!

- To finish the configuration process of OpenSSL, you need to run the following from the command prompt:

ms\do_masm

- This will take some seconds. After the script has run, you can start the make process:

nmake -f ms\ntdll.mak
This will start the command line compiler of Visual Studio 2003 and generate the libraries.

- These libraries will be placed in:

[openssl_home]\out32dll

- Add the following environment variables to your system:

```
INCLUDE_OPENSSL = [openssl_home]\inc32
   LIB_OPENSSL = [openssl_home]\out32dll
```

CppUnit

For compilation of CppUnit, the source code of version 1.8.0 or higher is needed. For this document, version 1.10.2 was used. You can get CppUnit as a zipped archive from:
Documentation for CppUnit needs to be generated using Doxygen. A short explanation of how to do this is given in chapter 1.6.
Using Visual Studio .NET 2003
To compile CppUnit using Visual Studio .NET 2003, you should unzip to any directory (referenced as [cppunit_home]). Compilation with VS2003 is quite simple:

- Start VS2003 and open:

```
[cppunit_home]\examples\examples.sln
```

- If asked if you want to convert you should answer '**Yes to all**'.

- Make HostApp the 'StartUp Project' and compile all. You can do so by pressing <F5>, which will compile/build all.

* ToDo * missing more detailed instructions
* how to run and test
--- snip (from the CppUnit docu) ---
- in VC++, Tools/Customize.../Add-ins and macro files/Browse...
- select the file lib/TestRunnerDSPlugIn.dll and press ok to register the add-ins (double-click on failure = open file in VC++).
--- snip (from the CppUnit docu) ---

-

-

zLib (Windows)

Proxytest only needs the static libraries of zlib-1.2.3. Compiling zlib-1.2.3 from source code is not necessary, and we explicitly recommend against doing so, since it is a bit tricky. There is a binary distribution for VS2003 that completely suffices for our purposes. However, you will need to get the zlib-1.2.3 sources, as you need the header files. To hook this distribution into proxytest you have to follow these steps:

- Download the binaries from

```
http://www.winimage.com/zLibDll/zlib123dll.zip
```

- To get the sources of zlib-1.2.3, unpack this zip archive to any directory, for example to

d:\coding\libsc\zlib-1.2.3 This directory is referenced in this document as [zlib_home]
http://www.zlib.net/zlib-1.2.3.tar.gz
Unpack these sources into [zlib_home]\src

- Change into

[zlib_home]\static32

- Rename the file

zlibstat.lib into zlib.lib This is necessary because the file is referenced from VS2003 as zlib.lib and nothing else. Alternatively, you could change the reference in VS2003, but each new checkout of the proxytest sources will overwrite your changes.

- Add the following environment variables to your system:

INCLUDE_ZLIB = [zlib_home]\src
LIB_ZLIB = [zlib_home]\static32 As an alternative to this procedure, you can instead download an archive that has already been set up from: http://www.anon-online.de/zlib/zlib.zip

The archive at that location has already been prepared according to the above instructions, and placed in a zipped directory. This archive can be unpacked into any directory. The chosen directory is referred as [zlib_mix]. It is not necessary to rename anything. You simply have to add the following to your environment variables:

INCLUDE_ZLIB = [zlib_mix]
LIB_ZLIB = [zlib_mix]\static32

+++ ToDo – provide the mentioned zip-file / upload it to the webserver +++

In Unix or Linux, the zLib library is normally already included. If it is not, you should get it with your distribution's package manager.

If you want or need to build the zlib-1.2.3 libraries completely on your own, you will run into problems with the dependencies of proxytest. As mentioned, building zlib-1.2.3 on your own is tricky and sources from various different sites are necessary.

pthreads (Windows)

Proxytest needs both the static and dynamic linked libraries and the header files of pthreads-win32. It's not necessary that you compile the libraries yourself - there are precompiled win-32 versions that work perfectly. We tested both 1.5.0 and 2.7.0 and they both run well. So versions in between are also likely to run well.

- To get the binaries and header files, go to:

ftp://sources.redhat.com/pub/pthreads-win32/ Choose one of the pthreads-w32-x-xx-x-release versions

- Run the sfx and choose an empty directory by using the 'Browse' button. Then click on '**Extract**'.

- We recommend, for example

d:\coding\libsc\pthreads-2-7-0
The chosen directory will be referenced in this document as [pthreads_home]

- Change into the directory

[pthreads_home]\Pre-built.2\lib
and rename the file pthreadvc2.lib to pthreadvc.lib
Again, this is a problem caused by the references from the VS2003 project file. As previously mentioned for zlib, you can change the VS2003 references, but they will be overwritten each time you checkout new versions of *proxytest* from CVS.

- add the following environment variables to your system:

INCLUDE_PTHREADS = [pthreads_home]\Pre-built.2\include
LIB_PTHREADS = [pthreads_home]\Pre-built.2\lib

Setting the path variables

If you have followed the instructions above on compiling the libraries for Windows and you added all of the environment variables as described, most of the configuration is already done.

You just have to add as a global variable:

```
LIB_PROXYTEST = %LIB_XERCES_C_RELEASE%;%LIB_XERCES_C_DEBUG%;                    %LIB_OPENSSL%;%LIB_ZLIB%;%LI
```

and finally add to your path variable:

PATH = %PATH%;%LIB_PROXYTEST%

1.4 Compiler

1.4.1 C++

Using Windows, it is not necessary to install an extra compiler since it is already included in VS2003 or Cygwin. Using Linux or MacOS, you need g++ from the GNU compiler collection in version 2.95 or higher. This compiler is usually shipped with every installation of Unix or Linux.

1.4.2 Java

For the purpose of compatibility on many operating systems, the JAP client has to be compiled with a Java compiler no newer than JDK 1.1.8. This version is the last available Java version for MacOS 9. Since there are some bugs and difficulties in the 1.1.8 JDK version for Linux, we recommend using version 1.4.0 or higher for execution and testing.

As a developer, you are restricted to use classes that are also available in the 1.1.8 JDK version of Java despite the bugs. Thus, the 1.1.8 version from the Blackdown Java port is recommended for compilation.

system	*version*	*source*
Windows	1.1.8_10	http://java.sun.com/products/archive/jdk/1.1.8_010/ http://java.sun.com/products/archive/jdk/1.1.8_010/jre/
Linux	1.1.8_v3	http://www.ibiblio.org/pub/mirrors/blackdown/JDK-1.1.8/i386/v3/
MacOS	1.1.8	http://developer.apple.com/java/classic.html

For installation, follow the automated scripts or install procedures. If automatic installation is not available, you will need to copy both the compiler and the runtime into one directory. We recommend adding the version number of your JDK to the directory name, e.g. `jdk1.1.8` Newer JDKs are recommended for testing and debugging purposes. As mentioned previously, it is also possible to install more recent JDKs/JREs on your system for testing purposes. Just be sure not to use libraries / classes / methods that are not available in Java 1.1.8.

1.5 IDEs

The Eclipse IDE is written in Java and available for all relevant platforms (Linux, Windows, MacOS). As it supports both Java and C++, we recommend it for developing of all AN.ON sub-projects. For Windows, we also use Microsoft Visual Studio 2003 for Mix development, but the installation of the needed libraries is tricky and we only recommend to use this IDE if you are a very experienced programmer. Our personal favourite for Java development is Borland JBuilder - if you do not mind using two different IDEs for Java and C++, and if you are not against closed-source programs, this will be your best choice for the Java sub-projects.

1.5.1 Specific development requirements on Microsoft Windows

For Java development, either Eclipse or JBuilder are a good choice and both easy to install. If you use Eclipse for C++ development, too, you have to run it under the Unix emulator Cygwin, which prodives the tools and libraries needed to checkout and compile the C++ code. Instead, you may use Microsoft Visual Studio for C++ development and an arbitrary extra tool for checking out the code from the source repository.

Installation of Cygwin

Download and install Cygwin from http://www.cygwin.com/ first. Execute `setup.exe` and confirm the default settings (Install from Internet, installation path etc) until you see the package selection window. Keep in mind that this installation might easily consume 500MB, so try to exclude unwanted packages. You will, however, need all the compiler tools (especially `gcc` and `make`) AND the C++ libraries mentioned in section 3.

Setting up a CVS client

If you use Microsoft Visual Studio for development, you will first need to manually download a CVS client. A really easy-to-use client that integrates completely in the Windows Explorer is *TortoiseCVS* (http://www.tortoisecvs.org/).

In the following, we provide short insturctions for the use of another tool, *WinCVS*, (http://www.wincvs.org/), which is distributed under the LGPL license:

Choose a current version (see the 'Latest Recommended Release' on the website). Download a release with an installer. Choose a mirror near your location. Start the installer and follow its instructions. Nothing additional is needed.

Now, to get the source code from CVS, follow these instructions:

- Start WinCVS and select a directory in which to put your copy of the code
- Enter this directory in the upper dropdown field, which also allows you to browse through your system directory
- To checkout a project / component choose **'Remote'** -> **'Checkout module'**

WinCVS – Settings for proxytest

- In the 'Checkout settings' window that appears, enter:

  Module... : proxytest *(or any other module name – see above)*

  Local folder... : any directory you want to put the sourcecode to

  CVSROOT : :pserver:anonymous@cvs.inf.tu-dresden.de:/home/sk13/cvssource
- Be especially careful about the colon, which has to be the first character before "pserver"
- Clicking the 'OK' button will start downloading the sources
- This procedure will work for each of AN.ON's modules (ie. JAP, MixConfig, InfoService)

C++ development using Visual Studio 2003 .NET

The project files with working settings for Visual Studio 2003 .NET are shipped with the *proxytest* code.

Visual Studio 2003 .NET – Opening proxytest and setting the 'Startup project'

- Start Visual Studio 2003 and open the file `proxytest.sln` as shown above
- Choose `proxytest` as your startup project and start a compile / make
- This should lead to a binary in the subdirectory `\windows\Debug_Build_Test`
- On errors, you can change settings and the paths to header / library files by manipulating the project settings by choosing: **Project** -> **Properties** from the upper menu bar

- For example, the paths to header files are set at: **C/C++** -> **General** -> **Additional Include Directories**
- And libraries are set at:

  Linker -> Input -> Additional Dependencies

Visual Studio 2003 .NET – Project settings

1.5.2 Eclipse IDE

Java Runtime Environment

Eclipse 3.1 requires the Java 1.5 (J2SE 5.0) runtime environment. If Java 5.0 is not yet provided by your distribution as a package, you can download it from:

`http://java.sun.com/j2se/downloads/`

Please follow the installation instructions shipped with the archive. If an older version is already installed on your system, this is not a problem. You may install as many JDKs simultaneously as you want in different directories on your system. Normally they will not interfere with each other.

IDE installation

After having set the basic runtime environment, you may proceed with the installation of the Eclipse IDE. To get Eclipse, go to:

`http://www.eclipse.org/downloads/index.php`

To accelerate your download, choose a mirror near your location. After having finished the download, unpack the archive. For Linux, you may do this like this:

tar xfz eclipse-SDK-3.1-linux.gtk.tar.gz

Then, move the new folder to your programs directory, for example to C:\Program Files\eclipse, `/opt/eclipse` or maybe `/usr/share/eclipse`

On Linux, you may need root access to move the directory:

su
mv ./eclipse /opt/eclipse (for example)
exit

Open a shell to create a link `/usr/bin/eclipse` to the Eclipse program file to get a quick access to it. If you use Windows, you have to open the Cygwin shell. A link may be created by

ln -s /cygdrive/c/eclipse/eclipse.exe /usr/bin/eclipse (for example)

if Eclipse was copied to c:\eclipse on a Windows system. Now simply type "eclipse" anywhere in a shell to start Eclipse. You may also create a graphical link for your desktop or start menu: On Windows, create a text file /usr/bin/eclipse.sh with the content

```
/usr/bin/eclipse
```

Make it executable by typing

```
chmod a+x /usr/bin/eclipse.sh
```

Now, create a desktop link with the following call (replace %cygdir% with your Cygwin installation path)%:

```
%cygdir%\bin\bash %cygdir%\bin\eclipse.sh
```

For Linux, please refer to your window manager help on how to generate desktop links.

Configuration for Java sub-projects

To get the current development version of JAP, there is a CVS with anonymous checkout permissions. Eclipse 3.1 provides a built-in CVS client that is easy to use.

- Start Eclipse, open the 'File' menu, choose 'New' and select 'Project'
- In the 'New Project' window, choose the 'CVS' and 'Checkout Projects from CVS' wizard
- Proceed by clicking the '**Next >**' button
- Enter the following values and proceed by clicking '**Next >**' again

Host: cvs.inf.tu-dresden.de
Repository path: /home/sk13/cvssource
User: anonymous

Connection type: pserver

- Finally, choose the module that you want to import (e.g. 'JAP', 'MixConfig',)

Eclipse – CVS Checkout settings for JAP

- It may take a few seconds until the code is downloaded and imported into Eclipse 3.1

Since there are no configuration files for Eclipse 3.1 in the CVS source yet, you have to adapt JAP to your systems settings. For running and compiling JAP, the following libraries are needed:

ApacheBZIP2	ApacheXMLrpc	HttpClient	JunitX 5.1	Mail (??)
ApacheFTPClient	BoucyCastleLightForJAP	Jama	Log4J	MRJClasses

As mentioned previously, you can download all of these libraries in the proper versions from the AN.ON website. Choose the packages for your desired component.

Please save these libraries into one common directory. Then change back to Eclipse and open the Project -> Properties and navigate to 'Java Build Path' and choose the 'Libraries' tab. Get rid of all the 'unbound' references by selecting each of them and pressing the 'Remove' button on the right. After this again add the external libraries which you have downloaded before by pressing the 'Add External JARs...' button and choosing the folder where you saved the libraries. Select all of them and press 'Open' in the file dialog.

Do not forget to add the 'JRE System Library' you want to execute the program on. In the screenshot below this is for example 'Java-1.5.0.sun-1.5.0_03' (the last entry). Do so by pressing the 'Add Library...' button, selecting 'JRE System Library' and normally choosing your 'Workspace default JRE'. Confirm with the 'Finish' button.

This JRE settings would allow you to test the proper execution of changed / improved JAP versions on older Java runtimes.

Eclipse – Properties - Project / Libraries for JAP

After having done this, the project can be compiled / run. The tests should run correctly as well. To start JAP on your machine, open the '**src**' and then the '(default package)' from the '*Package explorer*' on the left side. Choose `JAP.java` or `JAPMacintosh.java` and press the right mouse button. Then choose: '**Run as**' -> '**Java application**' or use the key combination **<Shift> + <Alt> + <J> <X>** to start it.

Installing FatJar Plugin

Eclipse doesn't automatically integrate external libraries into your .jar when exporting your Project. This functionality is provided by an plugin called "FatJar". To install this plugin, go to `http://sourceforge.net/projects/fjep` and download it. Unzip the "plugins" folder and move it to the plugins folder of Eclipse. You will have to start Eclipse with the "clean" option ("eclipse -clean"), otherwise the plugin won't be found.

Right click on your project and you will see the new entry "Build fat jar". This is also available under File -> Export. Use this instead of the standard jar.

For more information on this plugin, visit `http://fjep.sourceforge.net/` (description) and `http://fjep.sourceforge.net/fjeptutorial.html` (tutorial, just have a look).

Configuration for C++ sub-project 'proxytest'

Installing CDT

For C/C++ development, you will need to download CDT, a plugin for Eclipse that provides a fully functional C and C++ IDE for the Eclipse platform. More information about CDT can be found at:

`http://www.eclipse.org/cdt/`

Downloading and integrating CDT into Eclipse is simple. You can use the built-in update manager.

- Start Eclipse, open the 'Help' menu, choose 'Software Updates' and select 'Find and Install...'
- In the 'Install / Update' window, choose the second point, 'Search for new features to install' and click '**Next >**'
- Click the first button 'New Remote Site...' and enter the following into the window that appears:

Name: CDT 3.0.0 for Eclipse 3.1
then proceed by clicking the 'OK' button

Eclipse – Installation of CDT 3.0.0

- In the new window, be sure that the checkbox '*CDT 3.0.0 for Eclipse 3.1*' is selected or nothing will be installed
- Proceed by clicking the '**Finish**' button
- In the new window, select the checkbox '*CDT 3.0.0 for Eclipse 3.1*' again,
- The branches will be automatically selected. To proceed, click '**Next >**' again
- Accept the license agreement and click '**Next >**'
- Finally, you need to confirm where you want to install CDT and click '**Finish**'
- It takes a while until all the components have been downloaded
- In the 'Verification' window that may appear, you can proceed by clicking '**Install All**' even though CDT has no digital signature yet
- This verification may not be necessary in the future when the package has been signed
- Restart Eclipse so that the configuration is updated

Getting the code from CVS

Having set up the IDE for C++, you can now get the proxytest source code from the CVS server. If you have checked out the 'JAP' code before, there should already be an existing profile for the repository settings. If you have not done this yet, you can follow the instructions given above. You only have to exchange the module name with 'proxytest'.

Eclipse – Checkout proxytest from CVS

Be sure to put the source code in a new directory! It cannot be placed in your Eclipse workspace! To put the source code in a new directory, in the last window at 'Select the project location', be sure to **deselect** 'Use default workspace location' and enter another directory within your /home that you have already created.

This step is very important because otherwise it is not possible to create a new 'Standard C/C++ Project'.

After these steps, close the project by selecting it in the left 'Navigator' perspective and choosing 'Close Project' from the context menu. To remove the entry too, select 'Delete' from the context menu. Be sure that 'Do not delete contents' is selected when you click 'Yes' to confirm it.

Checking out the code on a windows platform you will have to use dos2unix to convert *config.h.in* from windows linebreaks to unix linebreaks. To do so, open cygwin and call

dos2unix /path/to/config.h.in

Because the dependencies on libraries and code for the Eclipse C++ project are based on the 'proxytest' make file, you have to run the configure script before you import the project into Eclipse.

To do so, open a console and change into the directory where you put the sources from CVS.

```
cd proxytest    (change into the directory)
./configure    (run the configure script)
```

It will take some time until the make file is generated.

Now you can import the source code as a 'Standard C/C++ Project'.

Eclipse – Create new 'C++ Make Project'

- Be sure to **deselect** 'Use default' and enter the directory where you put the code that you checked out from CVS
- Proceed by clicking the '**Finish**' button
- If you are asked whether you want to open the C/C++ perspective, you can answer with '**Yes**'

Compiling and Running proxytest

If everything worked, you should now be able to open each class and the syntax should be highlighted. As you start to program, you will realize that code completion is enabled from the Java perspective of Eclipse.

Although it should be disabled by default, when changing the perspective, the 'Build automatically' option could be enabled. This setting won't work with C++ because its build process is quite different from the Java build process. To change the settings, open the 'Project' menu from the upper menu bar and disable this option manually.

If the right versions of the necessary libraries are installed on your system in the correct manner, you should now be able to build the project.

To build a binary, select the project in the left-side 'Navigator' and choose 'Build Project' from the context menu.

Eclipse – Building 'proxytest'

If all went well, you should now be able to run the MIX server. Before starting, you have to set the configuration files. You can choose to run MIX as the first or last MIX in a cascade, with or without logging functionality. Additional or alternative configuration files can easily be generated with the MixConfigTool. The shipped configuration files can be found in: `/proxytest/documentation/SampleConfiguration/`

Starting the MIX with a specific configuration from within Eclipse 3.1 is done as follows:

- Set the focus to the 'Navigator' perspective and click 'mix'. A small input box will appear at the bottom of that window and the cursor will jump to the file name that most closeley matches your input (this is a built-in search function)
- You should find an entry like

```
>mix(Binary)
```

- Select this and choose from the context menu '**Run as**' and '**Run...**' (the last menu point)
- In the 'Run Manager' window that opens, navigate to the 'Arguments' tab and, for example, to run the MIX as last Mix, enter: --config=/documentation/SampleConfiguration/LastMix.xml
- Click the '**Run**' button
- In the '*Console*' perspective, you should now see the starting messages from the MIX

Eclipse – Running 'proxytest'

-

1.5.3 Borland JBuilder IDE

The *JBuilder* IDE from Borland is a good choice for Java development, as it is a platform independent, well-engineered piece of software and avaliable both in free and commercial versions.

Download and Installation

Download the program and a registration file from the following address:

Tool	*Version*	*Quelle*
JBuilder	X, Foundation	http://www.borland.com/products/downloads/download_jbuilder.html

You will have to register at Borland to start the download. We recommend that you also install the JBuilder help files, that are available in an extra package.

If you have problems to run the installer under Linux, you should set execution rights via `chmod +x`.

Programmstart

When first starting Jbuilder, it needs to be registered by reading the registration file. In the registration wizard, click "Have Activation File", then "next", and then enter the path to the registration file. Clicking on "Finish" then completes registration.

Links to the IDEs are automatically created during the installation. For Jbuilder on Linux, the link needs to be modified in order to insure that the JDK 1.1.8 is found correctly.: `JAVA_HOME="" /usr/JBuilderX/bin/jbuilder` (The exact path to JBuilder depends on your system/Linux distribution and may therefor vary)

CVS Checkout

CVS stands for "Concurrent Versioning System. It is a system for version control of code., intended to help programming teams coordinate their work.. All code is saved on a central CVS-server, and downloaded (checked out) by IDEs or other clients.

Note for JBuilder on Linux: The CVS-Integration for JBuilder contains a Bug, which may lead to CVS not being found by JBuilder. This is most likely caused by a version conflict between several versions of CVS installed on the same Linux system. This problem can be resolved by replacing the CVS executable used by Jbuilder with a symbolic link:

```
su
cd [JBUILDER_HOME]\bin
rm cvs
ln -s usr/bin/cvs cvs
```

Before a component can be edited or compiled, you need to create a project and check out the source code from CVS. AN.ON consists of the following modules (Name is case-sensitive!):

```
Jap
MixConfig
proxytest
```

The corresponding Jbuilder project will be created by checking out the code like this:

The module "Jap" will create several projects. To be able to update the each project via CVS, activate it (by double clicking it in the explorer window on the left-hand side) and register it for CVS by choosing "Select Project VCS" in the "Team" menu.

Formatting Options

Most formatting guidelines contained in the programming standards can be automated by the IDE. [PRICHTLINIEN]. Simply import the file "[IDE].codestyle", and import it like this:

(Caution: Autoformatting in the C++-Builder is buggy, do not use it yet!)

Integrating the libraries

Each Component depends on several external libraries, which first need to be set up in the IDE:

Compilation

If everything worked so far, you can now compile the project:

Jbuilder will put the compiled files into the “classes“ directory, C++Builder into the directory Linux\[Debug]Build or Windows\[Debug]Build.

Tip: If you should enconter an error during compilation, it is recommended to delete the contents of those directories before compiling again or starting “Rebuild“.

Ausführen

You can now run the project by choosing *Run->Run Projects* .

1.6 Doxygen

Doxygen is an open source documentation tool for different programming languages.

For graphical output it needs *Graphviz*, which is also open source and needs to be installed after *Doxygen*.

1.6.1 Download

tool	*version*	*source*
Doxygen	1.3.6	http://www.doxygen.org
GraphViz	1.1x	http://www.graphviz.org

1.6.2 Installation

Using Linux, we recommend installing Doxygen and Graphviz from the pre-compiled packages for your distribution.

Using Windows, you can download the binaries from the websites listed above. Convenient auto-installers are shipped with the Windows versions.

Run the main program by starting '*Doxywizard*'. In the GUI, select '*Load*' and choose the [proxytest_home] folder. From there, select the *.doxy file from which you want to generate the documentation.

In the source folder of each AN.ON component, there is a file called

<components-name>.doxy

This file has to be opened with the Doxywizard.

Finally, click the '*Start*' button to generate the documentation. This may take a while. You can follow the output of Doxygen in the lower 'Output' window. If errors occur, they will also appear there.

1.7 Unittest environments

1.7.1 JUnit

The unittests can be found in the "test" directory and can be started by right clicking on the desired class:

Tip: Unfortunately, the JUnit support built into JBuilder cannot be used with JDK 1.1.8 instead of the previously described process. Use of a newer JDK would be necessary for that.

1.7.2 CppUnit

In C++Builder, double click on the "AllTests" project. and then execute the Unittests as you would a normal project:

1.8 Running the tests

1.8.1 Architecture

On the local machine, we can simulate a Mix-cascade by running two Mixproxy instances in parallel. You will also

have to start the JAP client and set it as proxy for your browser.

If your web browser can access the data distributed by the Apache web server (e.g. the Apache documentation), the configuration is working properly.

There is no access possible from the internet.

The testing environment consists of the following elements:

- Infoservice – instance
- 2 Mixproxy instances (cascade)
- Apache Web server
- JAP – instance
- any Web browser (recommendation: Mozilla)

1.8.2 Configuration

Config files

type filter text

The necessary config files do not need to be created by hand but are shipped with the CVS module. You can find them at: `[proxytest_home]/docs/SampleConfiguration`

The following configurations and directories can be found there:

- logFirstMixWihth (d)
- logLastMix (d)
- FirstMix.b64.cer
- FirstMix.xml
- FirstMixWithLog.xml
- InfoService.properties
- jap.conf
- LastMix.b64.cer
- LastMix.xml
- LastMixWithLog.xml
- private.pfx
- public.cer

InfoService

The following certificates and config files can be found at: `[proxytest_home]/documentation/SampleConfiguration/`

- FirstMix.b64.cer
- LastMix.b64.cer
- private.pfx
- public.cer
- InfoService.properties

There, you will also find the logfiles for the InfoService component.

Mix cascade

The following config files for parts of a Mixcascade can be found at:
[proxytest_home]/documentation/SampleConfiguration/

- FirstMix.xml
- FirstMixWithLog.xml
- LastMix.xml
- LastMixWithLog.xml

Depending on the function of the MIX as first or last mix, the logfiles of the MIX server can be found in the `/logFirstMix` or `/logLastMix` subdirectory.

Apache HTTP Server (version 1.3.x oder 2.x)

Normally on Linux systems, the Apache server is running in the default configuration.
If it is not running yet on your system, it is quite simple to install with the help of your distribution's package manager.
Using a Windows system, you first have to get Apache from http://httpd.apache.org and install it.
You can proceed directly to this URL to get the 1.3.29 version for Windows.
During the installation process, you should keep the preselected default options.
Starting and stopping the server is done as follows.
(Linux)
Open a console and enter:
`su` (to get root) `/etc/init.d/apache start` (starting Apache) `/etc/init.d/apache stop` (stopping Apache)
(Windows)
There are shortcuts in your 'Start' menu for both starting and stopping.

JAP

The configuration file jap.xml (also in the CVS of the JAP project) has to be placed in the home directory of the user. If non-existent, the file will be generated automatically with default values upon the first use of JAP.
For running the tests, you will have to overwrite it with the values you need.
Debug- or logging output from JAP will be printed on an optional console, which has to be activated explicitly, or in the debug-view of your IDE. To turn on the debug output from JAP, activate the checkbox in 'Misc' in the JAP window.
The connection to your Mixcascade is set by activating the 'Anonymity' checkbox within the main window of JAP.

Web browser

Your browser has to be configured so that web access done exclusively through JAP. To do so, add JAP as http-proxy on address 127.0.0.1 at port 4001 (default).
Furthermore, you have to ensure that only this proxy is used for accessing the web (no other proxies should be set). You can ensure this by turning JAP off and trying to connect to any website that is not in your cache.
In order to avoid any external errors, we also advise turning off any caching in your browser.

1.8.3 Start

For the same reason of avoiding external errors, you should turn off any running firewall.
Needless to say that tests should never be run on a production system!
Now the time has come to open and run the components in their native development environment with each component in its own instance if the IDE.
Apache can run the entire time as a daemon / background service.

- InfoService (JBuilder)
- First Mix (VC7)

- Last Mix (VC7)
- JAP (JBuilder)

Finally, you have to activate 'Anonymity' in the JAP client. JAP should not generate any errors in the console.

1.8.4 Test

The correct configuration of the test environment can be tested with a simple trick. First, start JAP and make sure that it is running as a proxy (activate 'Anonymity'). Now start your browser (after configuration as described in 8.2.6) and open the URL:

`http://localhost`

You should see a website (normally the documentation of Apache) generated by your locally running Apache.

Now uncheck the 'Anonymity' checkbox in JAP and in your browser reload the same address. An error message generated by your browser should appear. There is no longer a proxy running in the background, so your browser cannot find any websites.

If this does not happen, you either did not set JAP as a proxy in your browser (see 8.2.6), or there is still a way your browser has access to the internet. Another problem may be that the site is already cached by your browser. Clear the browser cache and try again.

1.9 Payment Instance Setup

1.9.1 System requirements:

- a working Java environment
- a running Postgresql server
- the JDBC driver for postgresql
- the OpenSSL tool (unless you already have a key file and certificate)

1.9.2 You will need:

- BI.jar (contains all Java code for the payment instance)
- the jar files of all necessary libraries
- a key file and certificate for the JPI
- a jpi configuration file
- a postgresql database and server

1.9.3 Creating the Jar file

If you just want to run the JPI without making changes to the code, simply download it from the project homepage / ask a project representative for the code.

To build a new jar file after making changes in Jbuilder:

Do a rebuild

Menu "Project", choose "Rebuild Project Bezahlinstanz.jpx")

1. Set up the jarfile

In the Project Pane, pick the "Project" tab. Find the file "BI.jar", right-click on it, and choose "Properties". Under "Content", either pick "Include all classes and resources", or include at least the filters "anon/crypto/*.*" and "jpi.*.*". IMPORTANT: Under "Dependencies", pick "Include All" for the Postgres and BouncyCastle packages.

2. Rebuild jar file

In the jar file's context menu, click on "Rebuild".

1.9.4 Creating the key file and certificate

Use the following commands in a shell:

```
openssl dsaparam -genkey 1024 -out bi.key
openssl req -x509 -new -out bi.cer -key bi.key -subj "/CN=BI/"
openssl pkcs12 -export -in bi.cer -inkey bi.key -name BI -nodes -noiter -out bi.pfx
```

1.9.5 Creating the Postgres database

Use the following commands:

```
su postgres
createuser -A -D biuser
createdb -O biuser bidb
```

To confirm the creation of the database, log into your newly created database by typing `psql -U biuser -d bidb`

The database will be empty, the necessary tables will be created by the JPI itself.

If conecting to the database does not work, try editing the `hba.conf` file (under linux, usually in `/etc/postgresql`, or in your postgres data directory) to allow connecting with a password. Example:

```
# All other connections by UNIX sockets
local all all password
# All IPv4 connections from localhost
host all all 127.0.0.1 255.255.255.255 ident password
```

Note that Postgres looks only for the first line matching a connection in pg_hba.conf

After editing hba_conf, you need to restart postgres using `pg_ctl reload` (usally found in `/usr/lib/postgresql/bin`)

1.9.6 Creating the JPI configuration

The JPI configuration can be named anything you like and is a simple textfile with all lines in the format of option = value.A sample configuration named config.example is contained in the CVS project and can be used as a starting point.

Note that options regarding the volume plans and payment options offered are intended to be edited using the PIG (payment instance GUI) and will only be read from the configuration file if you start the JPI using the parameter new.

You will need to adjust the following options:

id	official identifier within the AN.ON system, used by JAP, AI and InfoService
name	human-readable name of the JPI
infoservices	Hostname and port of the InfoServices that the JPI will register itself with (Format: host:port, host2:port2,...)
japlisteners	How the JPI can be contacted by JAPs (Format: host:port, host:port,...).
ailistener	connection interface towards the AI (Format: host:port)
mclistener	connection interface towards the MixConfig tool
paypallistener	connection interface for Paypal to send IPN notifications – make sure the same value is set in your Paypal merchant account
virtualXXXlistener	Every listener interface has a corresponding virtual listener interface. If you do NOT use a cluster setup for the JPI, you can leave out the virtual listeners, or set it to the same as the real listener For cluster setups, the real (non-virtual) listener is what the JPI itslef uses to listen to; the virtual listener is used by the outside world to contact it, and what is sent to the InfoService
Database options	
dbhost	hostname or IP of the JPI's postgres database (most likely localhost)
dbport	Postgres port (most likely the default 5432)
dbname	Name of the dababase to be used by the JPI (should be bidb)
dbusername	Owner of <dbname>, should be biuser
dbpassword	<dbusername>'s password
Keyfile options	

keyfile	Path to the .pfx key file (just the filename if it's in the same directory as the JPI jarfile)
keyfilepassword	Password for <keyfile>, as entered when <keyfile> was created using OpenSSL. For extra security, leave blank to be asked for the password on startup.
Logging options	
logfilename	self-explanatory
logfilethreshold	accepted values: 1 up to 7 (7 = maximum detail)
logstderrthreshold	just like logfilethreshold, for messages to be written to stderr
nr_of_logfiles	The jpi keeps rotating log files, i.e. BI.log is the newest, older files get names like BI.log.1, BI.log.2,... Nr_of_logfiles determines how many old log files you want to keep, 10 is a good idea
size_of_logfiles	Size of a single logfile in bytes, suggested value: 5000000
Textfiles options	
termsfile	Name of the html file(s) containing the terms and conditions. Needs to be valid xhtml! Will be read once on startup, changes require a restart The JPI will load files with names of <termsfile>_<language>.<termsfile-extension> e.g. termsfile=terms.html, termslanguages=de,en => will load terms_en.html and terms_de.html
termslanguages	comma-separated 2-letter codes, e.g. de,en
policyfile	html file(s) containing the cancellations policy, ananlogous to termsfile
policylanguages	comma-separated 2-letter codes, e.g. de,en
Misc options	
maxcascadelength	Integer. Maximum no of mixes that will be paid in one cascade
log_payment_stats_enabled	True or false: set true if you want to log statistics about traffic per Mix or Jap and month.

For Details on how to set other options regarding prices and payment, see [DIPLOBAIER]. However, these options should really be set using the JPI-GUI. Additional Options needed for configuring the paysafecard or Call2pay payment options are described in [DIPLOSCHRAML].

1.9.7 Starting the JPI

It is recommended to run the JPI inside a screen. To create a screen, type `screen -R jpi` (or any other name). Then start the jpi inside the screen, and return to the normal shell using Ctrl+A+D. To return to the shell, use `screen -list` for an overview over existing screens, and `screen -r <name>` to go to an existing screen.

Your Java classpath needs to include the jar files of the necessary libraries (Every single .jar-file, not just the directory of the libraries!). Alternatively, you can integrate the libraries into the single BI.jar file by using the BI.jar file's context menu in Jbuilder, going to "Properties", then "Dependencies", and choosing "Include all" for all libraries.

To start the JPI, simply use `java -jar BI.jar config.example`

If you use another configuration file, replace config.example with the path and filename to your custom configurations file.

IMPORTANT: When starting the JPI for the first time, add a command line parameter `new` (e.g. `java -jar BI.jar config.example new`). This will create the database tables, and populate the paymentoption tables in the database from the configuration file.

1.10 Accounting Instance Setup

1.10.1 System Requirements

- A working, running Postgresql Server

- Ability to recompile the Mix code

1.10.2 You will need

- The mix code (CVS module proxytest)
- the postgres-devel packages (should be OK if you have Postgresql installed)

1.10.3 Compiling the first Mix

1. Configure the Mix using ./configure –enable-payment
2. Recompile

1.10.4 Setting up the database

1. Create the User

In a shell, use `createuser -A -D aiuser`

1. Create the database

In a shell, use `createdb -O aiuser aidb`

1. Create the necessary tables

Log into the database by typing `psql -U aiuser -d aidb`
Then run the following commands:

```
create table costconfirmations (
  accountnumber bigserial,
  bytes bigint,
  xmlcc varchar(2000),
  settled integer,
  cascade varchar(200),
  primary key(accountnumber,cascade)
);
create table prepaidamounts(
  accountnumber bigint unique not null,
  prepaidbytes integer,
  cascade varchar(200),
  primary key(accountnumber, cascade)
);
create table accountstatus(
  accountnumber bigint,
  statuscode integer
);
```

or use the prepared script: psql -U aiuser -d aidb < mixtables.sql
To confirm the creation of the database, log into your newly created database by typing `psql -U aiuser -d aidb`. If it worked, log out again using `\q`.
If connecting to the database does not work, try editing the `hba.conf` file (under linux, usually in `/etc/postgresql`, or in your postgres data directory) to allow connecting with a password. Example:

```
# All other connections by UNIX sockets
local all all password
# All IPv4 connections from localhost
host all all 127.0.0.1 255.255.255.255 ident password
```

Note that Postgres looks only for the first line matching a connection in pg_hba.conf
After editing hba_conf, you need to restart postgres using `pg_ctl reload` (usally found in `/usr/lib/postgresql/bin`)

1.10.5 Payment Configuration

Use the Mixconfig tool. When editing the xml configuration file by hand, the relevant sections can be found under <Accounting>

General Options	
SoftLimit	Minimum amount of prepaid bytes at which a cost confirmation is demanded from the JAP
HardLimit	The JAP will be kicked out if the number of prepaid bytes drops to this level. Obviously, needs to be set LOWER than SoftLimit. WARNING: A distance between SoftLimit and HardLimit of at least 400 000 is recommended to give Jap enough time to reply with a CC even when surfing fast
SettleInterval	Interval in seconds: How often you want to contact the payment instance to cash in the cost confirmations received from JAPs
PrepaidIntervalKbytes	How many kilobytes are paid in advance, i.e. whenever softlimit is reached, AI will request a CC for (current transferred bytes + prepaidInterval)
Database options	
host, port, dbname, username,password	self-explanatory
Pricing	
Price (shown in Mixconfig tool)	NOT to be set manually; instead, use the Mixconfig tool to use a valid, signed price certificate. If editing manually, be sure to insert the complete xml node for a price certificate, including the JPI's signature

1.10.6 Price Certificates

In order to set a price, you will need to:

1. Request the price

Use the MixConfigTool. Load a Mix configuration that contains your operator certificate, which needs to be registered with the payment instance (i.e. stored in the JPI's database).

To to the payment Tab, find the "PriceCertificate" panel, click on "Change", and enter your new price.

1. Have the JPI's operator confirm your new price

This will be done by logging into the PIG, and signing your new price.

1. Store the signed price certificate in your Mix's configuration

In the MixConfigTool, load your Mix's configuration, go to the PriceCertificate panel on the Payment tab, click "Update". Select your newly signed price certificate, and click on "Use".

Alternatively, you can edit the mixconfig-file manually. In that case, the <PriceCertificate> node goes under <Accounting>, and the JPI's certificate needs to be written as a block (no spaces before or after each line of the certificate).

1.11 Payment Instance GUI (PIG) setup

1.11.1 Ruby

Check if ruby is already installed by running `ruby -v`. If not, go to `ruby-lang.org` to download and install the correct version for your operating system. Note that the version of Ruby shipping with Mac OS X 10.4 does NOT run Rails well, installing a newer version is required. On Debian, `apt-get install ruby ri irb ruby1.8-dev libzlib-ruby zlib1g rdoc` will install Ruby 1.8

On RedHat/Fedora/CentOS, `yum install ruby ruby-libs ruby-devel ruby-rdoc ruby-irb ruby-ri ruby-docs` should work.

Of course, you can also compile from the source code. Excellent instructions on how to install Ruby and Rails from source can be found at http://hivelogic.com/narrative/articles/ruby-rails-mongrel-mysql-osx (meant for MacOS X, but also applicable to Linux)

1.11.2 Rails

Rails is available as a ruby gem (Gem is the package management system for Ruby).

To install Gem, download the source code from `rubygems.rubyforge.org`. On Linux, you can use `wget` http://rubyforge.org/frs/download.php/17190/rubygems-0.9.2.tgz Unzip, and install with `sudo ruby setup.rb`

To install Rails, simply type `sudo gem install rails --include-dependencies`

In case you should get an error like "Could not find rails in any repository", do gem list rails –remote first. This updates the list of avaiilable packages, afterwards the regular gem install rails command should work.

If you should have multiple versions of the rails gem installed, you have to tell your application which one to use in `config/environment.rb`, e.g. RAILS_GEM_VERSION = '1.2.1'

1.11.3 Database configuration

The ruby adapter for postgresql is available under http://ruby.scripting.ca/postgres.

It is also available as a gem, to install use `gem install postgres`.

Note that in order to compile, postgresql needs to be installed on the same machine. If necessary, set non-standard installation directories with e.g. `gem install postgres -- --with-pgsql-include-dir=/usr/local/pgsql/include -with-pgsql-lib-dir=/usr/local/pgsql/lib`

Alternatively, a pure-ruby implementation called postgres-pr offers lower performance, but easier installation: simply type `sudo gem install postgres-pr`

To configure Rails for using the correct database, edit the file database.yml in the /config directory under the rails application's root directory.

Use the following values:

adapter: postgresql (or postgres-pr, if you went with the pure-ruby version)

database: bidb

username: biuser

password: yourpassword

host: localhost

The adapter is called "postgresql", no matter if you use postgres or postgres-pr

Pay attention to YAML syntax! (e.g. use two spaces for indentation instead of Tabs, no superfluous newlines at the end of the file)

1.11.4 Plugins

Run the following command to install the graphing library Scruffy:

gem install scruffy

The project's website can be found at scruffy.rubyforge.org

The other plugins should work just by copying the contents of your application folder over to the server. To install from scratch, use

gem install login_generator

and

script/plugin install http://svn.cardboardrocket.com/paginating_find

1.11.5 Java Bridge

Download YAJB from http://raa.ruby-lang.org/project/yajb.

Unzip and run `ruby setup.rb` to install.

Find the global variable JBRIDGE_OPTIONS in the bigui rails application (should be in `config/environment.rb`, `controllers/prices_controller.rb`), and check that :`classpath` points to the Java libraries you intend to use (at least `Jap.jar` and `BouncyCastleLightForJAP.jar`).

Set the constant PATH_TO_JPI_CERT in controllers/prices_controller.rb to the absolute path, including the file name, to the private key file of the JPI

1.11.6 Server

Go to the application's root directory, and start the integrated Webrick server with `script/server`. Open your web browser and find your application at http://localhost:3000. Use option -p to user a different port, or -e for a different environment, e.g. `script/server -p 8080 -e production`

If you have problems using Webrick, try the alternative Mongrel server.

To install, type `gem install mongrel` (needs ruby version 1.8.4 or higher)

To start, go to the pig root directory, and type `mongrel_rails start`

mongrel_rails -h will output all startup options, use `mongrel_rails stop` to shut the server down.

For production use (better performance, using SSL etc) , a "proper server" (i.e. Apache or Lighttpd) is recommended.

Apache + CGI is VERY slow, a fallback option only.

Apache + FastCGI is possible, but seems to never work properly (random errors, escessive database connection,...)

Lighttpd + FastCGI has not been tried yet.

Current setup is Apache for SSL with mod_proxy redirecting to Mongrel, works well.

1.12 Bibliography

textreference	link
[ANON]	http://www.anon-online.de
[CPPUNIT]	http://cppunit.sourceforge.net
[JUNIT]	http://www.junit.org
[PRICHTLINIEN]	http://anon.inf.tu-dresden.de/develop/codingstyle_de.html
[UNITTESTS]	Johannes Link: Unit Tests mit Java, dpunkt.verlag, 1.Auflage 2002 http://www.dpunkt.de/utmj/
[DIPLOBAIER]	Tobias Baier: Fertigstellung und Inbetriebnahme eines Bezahlsystems für den AN.ON-Anonymisierungsdienst (Diplomarbeit)
[DIPLOSCHRAML]	Elmar Schraml: Technische und organisatorische Fertisgstellung eines Bezahlsystems für den Internet-Anonymisiserungsdienst AN.ON und Überführung in die Produktivphase (Diplomarbeit)

2 Release Procedures

2.1 JAP

2.1.1 Overview

Various steps are involved in the overall release of JAP. As visible results we have at the end:

- a new JAP.jar, which can be downloaded from the Web-site (currently: http://anon.inf.tu-dresden.de/jap/JAP.jar)
- new setup files for Windows and OSX
- a new released version of JAP for Java Web Start
- a new released version of JAP used by the internal update function of JAP
- a new release of JAP on SourceForge (sf.net/projects/anon)

Below is an overview of the steps executed during a release build. In the following sections these steps are explained in more detail.

1. **L**ibraries: compilation and deployment of the libraries JAP depends on
2. **S**ource preprocessing: pre-process the sources of JAP, e.g. setting the version number
3. **B**uild JAP.jar: Create the signed JAP.jar
4. **D**eploy JAP.jar: Deploy JAP.jar to the Web-site, Java Web Start, the internal update system.
5. **B**uild Windows installers: Build the Windows installers (online and offline version).
6. **D**eploy Windows installers: Deploy the Windows installers to the Web-site
7. **B**uild OSX install package: Build the install package for OSX.
8. **D**eploy OSX install package: Deploy the OSX install package to the Web-site
9. **S**ign installer packages: Create the signatures for the installers for the Web-site.
10. **D**eploy to SourceFroge: Deploy all the relevant artefacts to the SourceForge projet Web-site.

All the steps mentioned above are executed by running the Apache Ant script ant_build_maven_anon_release.xml on the release build machine.[1] Note that this script needs a few properties/parameters to work correctly. More specific the following properties have to be given as command line parameters:

- japReleaseVersion: the release version, using the format: xx.xx.xxx
- japReleaseDate: the release date, using the format: YYYY/MM/DD hh:mm:ss

2.1.2 Libraries

In this step the libraries JAP depends on are compiled as release version. The resulting artefacts are deployed to the the anon-release artifactory repository (https://anon.inf.tu-dresden.de/artifactory/artifacts-release/).

"Compiling as release version" basically means, that no debug information is generated and that the pom.xml file is preprocessed removing the -SNAPSHOT part of the version identifiers (i.e. from the version identifier of the library build as well as from the version identifiers of libraries this library depends on). Note the for consistence reasons not only the desktop (aka PC) version of each library is built but also the Android version.

[1] The execution can be started on Jenkins by executing the JAP-release project.

2.1.3 Source preprocessing

During this step the files JAPConstants.java and pom.xml are changed.
JAPConstants.java:

- set aktVersion to the release version number, given as Ant parameter (japReleaseVersion)
- set RELEASE_DATE to the release date, which is given as Ant parameter (japReleaseDate)
- set m_bReleasedVersion=true, indicating that this version is a release version

pom.xml:

- set the version number to the release version number, given as Ant parameter (japReleaseVersion)
- remove the -SNAPSHOT part of the version identifiers of the dependencies

2.1.4 Build JAP.jar

Call Apache Maven to build JAP.jar. During this step the resulting JAP.jar is also digitally signed.

2.1.5 Deploy JAP.jar

During this step the release version of JAP.jar is copied to the AN.ON Web-site. Additionally the release version number will be written to the text file (japversionstable.txt), which is part of the Web-site. This text file is used by the Web-site to show the version number of the current JAP release version. JAP.jar is also copied to the Java Web Start deployment directory. Finally the deployment descriptor (japrelease.jnlp) used for Java Web Start as well as the internal JAP update system is updated, i.e. the version number and release date is set.

2.1.6 Build Windows installers

During this step the current version of the installer build scripts are checked out from the related subversion repository (JAPWinInstall). Afterwards the file japversion.nsh is updated according to the release version number. Note that the release version number is take from the Web-site, i.e. from the file japversionstable.txt! Moreover all the needed JAP related files are downloaded from the Web-site and copied to the installer build input directories. Finally the installers are build.

2.1.7 Deploy Windows installers

During this step the previously created installers are copied to the Web-site.

2.1.8 Build OSX install package

During this step the OSX install package (which is actually just a disk image) is created. This happens on a OSX server machine, which is accessed using ssh. On that machine the script updateDmg.sh is executed. This script expects the release version number as parameter and does the following:

- Download of the current OSX disk image from the Web-site.
- Download of the release version of JAP.jar from the Web-site.
- Mount that disk image and copy the content to a temporary directory (build directory).
- Update Info.plist (found in the build directory) according to the version number given on the command line.
- Copy the downloaded JAP.jar to the right place within the build directory.
- Create a new disk image from the build directory.
- Copy the newly created disk image to a public Web-directory on the OSX server machine.

2.1.9 Deploy OSX install package

After the script above is executed the resulting newly created disk image is downloaded from the OSX server machine Web-site and copied to the AN.ON Web-site.

2.1.10 Sign installer packages

During this step the authenticode signatures of the Windows *.exe installers are created. Moreover the GPG signatures for all the relevant artefacts are created.

2.1.11 Deploy to SourceFroge

During this step all the relevant artefacts are copied to the SourceForge project Web-site. Therefore they are first copied to a temporary local directory named according to the release version number. Afterwards this directory is copied to the SourceForge Web-site using `rsync` over `ssh`.

2.2 JonDo und JonDoPortable

2.2.1 JonDo

Für den Windows-Installer von JonDo wird NSIS (Version 2.46-7) verwendet, welches z.B. in Debian Wheezy mitgeliefert wird. Um den Installer zu bauen, muss nun Folgendes gemacht werden:

- Der aktuelle Installer-Code ist auszuchecken. (https://svn.jondos.de/svn/JAPWinInstall/JAPWinInstall/trunk/)
- Die nötigen Plugins und .nsh-Dateien befinden sich im NSIS-Verzeichnis und müssen in die entprechenden Ordner in /usr/share/nsis kopiert werden.
- Die aktuelle JAP.jar muss in den JAP-Ordner kopiert werden.
- Wenn dem Installer eine neue Java-Version beigelegt werden soll, dann ist diese in den Java-Order zu kopieren und in der JonDo.nsi einzutragen (Section secJava).
- In der JonDo.nsi müssen noch `VERSION` und `VIProductVersion` an die Versionsnummer des Releases angepasst werden.
- im trunk-Verzeichnis ist die JonDo.paf.exe mit

  ```
  makensis JonDo.nsi
  ```

  zu erzeugen.
- Das Codesigning-Zertifikat muss in den Explorer importiert werden. Das geht einfach über einen Doppelklick auf die .pfx-Datei.
- Als nächstes öffnet man ein MS-DOS-Fenster in dem Verzeichnis, in dem die JonDoSetup.paf.exe liegt.
- Nun kann man die Programmdatei mithilfe des folgenden Befehls signieren:

  ```
  signtool.exe sign /a \
            /t http://timestamp.comodoca.com/authenticode \
            JonDoSetup.paf.exe
  ```

2.2.2 JonDoPortable

Für eine Aktualisierung von JonDoPortable muss nichts extra gemacht werden, es sei denn, eine neue JRE ist nötig. In diesem Fall ist die aktuelle JRE für Windows herunterzuladen und zu entpacken/installieren, um das jre-Verzeichnis nach trunk/JonDoPortable/Files/App/JonDo kopieren zu können.

2.3 JonDoFox Extension

Für ein Release der JonDoFox-Erweiterung sind folgende Schritte notwendig:

- Zuerst muss der Code ausgecheckt werden: https://svn.jondos.de/svnpub/JonDoFox_Extension/trunk
- Das aktuelle Datum muss im CHANGELOG eingetragen werden.
- In der Datei install.rdf muss die neue Version eingetragen werden.
- Es muss in der Datei jondofox-manager.js der neue Versionsstring des Profils eingetragen werden (in checkProfileUpdate()). Sonst bekommt der Nutzer des neuen Profils eine Warnung, dass dieses nicht aktuell ist.
- Es müssen die Fingerabdrücke aus der X509ChainWhitelist.js von HTTPS Everywhere in die ssl-observatory-white.jsm von JonDoFox übernommen werden und die aus der Root-CAs.js von HTTPS Everywhere in die ssl-observatory-cas.jsm von JonDoFox.
- Die Standard-Einstellungen des JonDoFox sind gegebenenfalls anzupassen (das betrifft im Augenblick den User Agent für den JonDo- und den Tor-Modus und Die 'app version' bei den Safebrowsing-Einstellungen. Dort ist der „appver"-Wert so abzuändern, dass er mit dem des JonDo User Agents übereinstimmt) (siehe die preferences.js in trunk/src/defaults/preferences/). Das gilt auch für die Einstellung „extensions.jondofox.jdb.version", die die Update-Warnung des JonDoBrowsers regelt (für weitere Dinge, die es beim Release des JonDoBrowsers zu beachten gibt, siehe Abschnitt 2.5 auf Seite 42).
- Dann muss die neue .xpi mit dem makexpi.sh-Skript gebaut werden.
- Anschließend gilt es, die .xpi zu signieren. Ich verwende im Augenblick `signtool`, was sich im Paket `libnss3-tools` befindet. Zuerst muss das Code-Signing-Zertifikat in den Browser importiert werden. Dann sollte es reichen das signxpi.sh-Skript zu starten. Wenn Fehler auftreten, dann liegt das vermutlich am falschen Pfad des zu verwendenden Firefox-Profils oder an fehlenden Intermediate-Zertifikaten. Das erstere lässt sich direkt in der signxpi.sh beheben, das letztere durch Import der fehlenden Zertifikate (gegenwärtig von StartCom Ltd. https://www.startssl.com/certs/sub.class2.code.ca.pem).
- Die signierte jondofox.xpi muss nach /var/www/website/htdocs/downloads kopiert und die dortige updateInfo.xhtml sowie die update.rdf angepasst werden. In letzterer sind lediglich die Versionsnummer zu erhöhen und die maxVersion-Werte aus der install.rdf (im trunk/src-Verzeichnis) zu übernehmen. In die updateInfo.xhtml kommen die Eintragungen aus der CHANGELOG-Datei sowie die neue Versionnummer.
- Im Wiki müssen zum Schluss noch die Changelogs aktualisiert werden.
- Dann gilt es die geänderten Dateien mit

  ```
  svn ci
  ```

  ins SVN einzupflegen und einen entsprechenden Release-Tag zu erstellen:

  ```
  svn copy https://svn.jondos.de/svnpub/JonDoFox_Extension/trunk \
  https://svn.jondos.de/svnpub/JonDoFox_Extension/tags/new_version
  ```

  „new_version" ist dabei die neue Version der JonDoFox-Erweiterung.

2.4 JonDoFox Profile

Zunächst einmal gibt es im Bugtracker immer einen aktuellen Eintrag der Form „Users want to get an updated JDF version (X.X.XX) and profile (X.X.X)" und dort stehen all die Kleinigkeiten drin, die noch für das Release zu machen sind. Es sind nicht immer alle Punkte relevant, aber auf jeden Fall die folgenden:

- **Notwendige Punkte**

  - update Firefox portable: Dazu müssen sowohl die englische (en-US) als auch die deutsche (de) aktuelle Firefox-Version heruntergeladen und deren Signatur überprüft werden. Die Versionen und Signaturen findet man unter http://releases.mozilla.org/pub/mozilla.org/firefox/releases/latest. Wenn alles stimmt, müssen die Veränderungen ins SVN übernommen werden. Dazu ist die aktuelle Version auszuchecken (https://svn.jondos.de/svnpub/JonDoFox_Profile/trunk) und die Firefox-Versionen sind nacheinander auf einem Windows-Rechner (oder in einer VM etc.) zu installieren und zu mergen. Der gemeinsame Code kommt nach trunk/Firefox/App/firefox und die übrigen, sprachspezifischen Teile nach FirefoxByLanguage/de|enFirefoxPortablePatch/App/firefox.
  - update Profile if necessary, NSIS script: Die angegebenen Werte in allen prefs-Dateien müssen aktualisiert werden und im NSIS-Skript JonDoFox.nsi unter trunk/Firefox/Other/Source sind `FF-VERSION` und `VERSION` anzupassen. Daüber hinaus ist in der Datei JonDoFoxPortable.nsi `VER` entsprechend zu erhöhen und die JonDoFoxPortable.exe neu zu bauen, so dass beim näheren Betrachten der .exe im Explorer die korrekte Version angezeigt wird.
  - include latest extension versions: Die aktuellen Versionen der Erweiterungen sind in das Profil-Verzeichnis (trunk/full/profile/extensions) einzuchecken.
  - update profile version in appinfo.ini files: In den jeweiligen appinfo.ini-Dateien unter trunk/de|enFirefoxPortablePatch/
    sind die Werte für `PackageVersion` und `DisplayVersion` anzupassen.
  - disable Crashreporter in application.ini: In der Datei application.ini, die sich in trunk/Firefox/App/firefox befindet, ist `Enabled` unter `[Crash Reporter]` auf `0` zu setzen. Gleiches gilt für die webapprt.ini in /trunk/Firefox/App/firefox/webapprt.
  - tagging: Wie schon beim JonDoFox-Release gilt es, die noch nicht eingebrachten Änderungen ins SVN einzupflegen und einen Release-Tag zu erstellen.
  - changelogs in our wiki: Ganz zum Schluss, wenn alle neuen Versionen auf dem Server sind, sind wiederum die jeweiligen Changelogs in unserem Wiki anzupassen.
  - upload of new jondofox.xpi: Wenn das Profil online ist, dann kann der Upload der XPI beginnen (siehe dazu den entsprechenden Abschnitt im JonDoFox-Release-Teil).

- **Übrige Punkte**:
  - update check JDF profile: Wurde bereits beim JonDoFox-Release (Erweiterung) besprochen.
  - update SSL Observatory cert whitelist + CAs: Wurde bereits beim JonDoFox-Release (Erweiterung) besprochen.
  - default preferences of the JDF extension: Wurde bereits beim JonDoFox-Release (Erweiterung) besprochen.
  - update patterns.ini (Adblock Plus): Die aktuellen Filterregeln müssen nach /trunk/full/profile/adblockplus kopiert werden.
  - JonDoFox' Defenses (see our Wiki) and help pages update: Wenn etwas zu den Verteidigungen, die JonDoFox liefert, hinzukommt oder die Hilfeseiten, die im portablen JonDoFox mitgeliefert werden, aktualisiert werden müssen, dann muss das erstere im Wiki geschehen und das letztere in trunk/FirefoxByLanguage/de|enFirefoxPo

Bevor man die JonDoFox.paf.exe in trunk/Firefox/Other/Source nun mit

```
makensis JonDoFox.nsi
```

baut, müssen wiederum eventuell fehlende NSIS-Plugins bzw. -Headerdateien nach /usr/share/nsis in das jeweilige Verzeichnis kopiert werden. Die Dateien befinden sich unter /trunk/Firefox/Other/Source/Include|Plugins. Die Authenticode-Signatur für die .exe wird wieder auf einem Windows-Rechner gemäß der Prozedur, die schon beim JonDo-Release erläutert wurde, erstellt. Hochgeladen wird die mit unserem Schlüssel signierte .paf.exe wiederum nach /var/www/website/htdocs/downloads.

Für die **Mac-Versionen** des Profils muss man sich im Prinzip nur die Datei jondofox_pkg.sh aus dem trunk-Verzeichnis auschecken und diese starten. Nachdem die Mac-Images gebaut worden sind, kann man das Skript abbrechen. Nun müssen die .dmg-Dateien nacheinander in `JonDoFox_OS_X.dmg` umgewandelt, signiert und auf den Server (nach /var/www/website/htdocs/de|en/downloads) hochgeladen werden.

Für die **Linux-Versionen** muss ebenfalls das Skript jondofox_pkg.sh benutzt werden. Die signierten Dateien kommen dabei gleichfalls nach /var/www/website/htdocs/de|en/downloads.

Die **Debian-Version** kann nach dem Build der Linux-Version aus den Linux-Archiven gebaut werden. Es ist das Verzeichnis *debian* auszuchecken. Danach ist die Datei *debian/changelog* anzupassen und die Changelogs aus dem Wiki zu übernehmen (ohne Windows- und MacOS-spezifische Details). Die Debian-Pakete werden mit

```
fakeroot make -f debian/rules binary
```

gebaut. Und mit

```
fakeroot make -f debian/rules clean
```

wird wieder aufgerämut. Dann sind die Debian-Pakete auf den Webserver nach */root/debian/all* hochzuladen, wo die architekturunabhängigen Pakete für die Nutzer liegen.

Das Debian-Repository ist mit dem Skript */root/debian/create-archiv-pub* zu aktualisieren, was die Repositories für die unrerschiedlichen Distributionen in dem Verzeichnis */var/www/debian/dists* erstellt. Danach ist die jeweilige „Release"-Datei aus */var/www/debian/dists/wheezy|squezze|sid|raring...* herunterzuladen und auf dem eigenen Rechner mit dem JonDos-Signaturschlüssel zu signieren und die Signaturen wieder hochzuladen. Die Signaturen der Release-Dateien müssen *Release.gpg* heißen. Die Signatur kann mit folgendem Kommando erstellt werden:

```
gpg -b --armor --default-key=support@jondos.de -o Release.gpg Release
```

Anschließend werden die lokalen Pakete umbenannt und die Version aus dem Namen wird entfernt. So wird z.B. aus „jondofox-de_2.6.14.1_all.deb" „jondofox-de_all.deb". Die Pakete werden signiert und zusammen mit der Signatur (die nun die übliche Endung „.asc" besitzt) auf den Webserver in die Download-Verzeichnisse nach */var/www/website/htdoc* hochgeladen.

Wenn alle Pakete und Archive auf dem Webserver bereit stehen sind im Verzeichnis */var/www* auf dem Webserver die Versionsnummern in der Datei *jondofox-version.txt* anzupassen und das Skript *calcmd5.sh* zu starten, das die Webseite aktualisiert.

2.5 JonDoBrowser

2.5.1 Build-Setup

- Linux: Da verwenden wir die eigenen Buildserver und es muss nur das Release-Buildskript (build_jdb_stable_linux.sh) aus dem build Verzeichnis des JonDoBrowser-Repositories (`https://svn.jondos.de/svnpub/JonDoBrowser/trunk`) ausgecheckt und gestartet werden.

- Mac OS X: Gegenwärtig wird Mac OS X 10.6.8 zum Bauen der 32bit- und 64bit-Binaries verwendet, da Mozilla das ebenfalls so macht. Um alles Einzurichten, ist zunächst `https://developer.mozilla.org/en-US/docs/Developer_Guide/Build_Instructions/Mac_OS_X_Prerequisites` zu folgen. Zum Kompilieren wird `clang` verwendet, was aber in einer Version $>$ 2.9 verfügbar sein muss (3.1 funktioniert z.B.). Ich habe das selbst kompiliert und dann installiert, was den Pfad „/usr/local/clang" im Buildskript ergibt. Das muss eventuell an die lokalen Gegebenheiten angepasst werden.

  Darüber hinaus muss das Verschlüsselungswerkzeug `gpg` installiert und der Mozilla-Schlüssel importiert werden (den findet man z.B. hier: `http://releases.mozilla.org/pub/mozilla.org/firefox/releases/latest/KEY)`.

  Wenn mehr oder weniger Kerne beim Bauen verfügbar sind, so muss der Parameter „j4" in .mozconfig_mac in /trunk/build angepasst werden. Als Faustregel gilt, dass man „jn" verwendet, wobei n = 2 x Kernanzahl gilt.

  Dann muss nur noch das Build-Skript (build_jdb_stable_mac.sh) aus dem Repository geholt und gestartet werden.

- Windows: Gegenwärtig wird Windows 7 und Visual C++ 2010 verwendet. Die Einrichtung der vollständigen Buildumgebung wird auf `https://developer.mozilla.org/en-US/docs/Developer_Guide/Build_Instructions/Windows_Prerequisites` beschrieben. Darüber hinaus gilt für Windows ebenfalls, dass zusätzlich `gpg` installiert (z.B. `gpg4win`) und der Mozilla-Schlüssel importiert werden muss.

  Wenn man mehr oder weniger Kerne als 4 zur Verfügung hat, dann muss der Wert in .mozconfig_win32 wie oben beschrieben geändert werden.

Schließlich ist das Build-Skript (build_jdb_win.sh) in einem Terminal auszuchecken (siehe dazu den Abschnitt „Opening a Build Command Prompt“ unter dem oben angegebenen Link). Die Variable „gpg“, die den Pfad zum lokalen GPG-Programm enthält, muss nun gegebenenfalls angepasst werden. Danach ist das Build-Skript zu starten (siehe dazu den Link zu den Build-Vorausstzungen oben).

2.5.2 Release

Wie beim JonDoFox-Profile gibt es im Bugtracker einen Eintrag, der alle noch zu erledigenden Dinge für ein Release enthält: „JonDoBrowser users want to have a X.X release“.

- proper jondofox.xpi: Wenn die jondofox.xpi gemäß den oben angegebenen Schritten erstellt wurde, ist sie für den JonDoBrowser noch einmal mit

  ```
  ./makexpi.sh -b && ./signxpi.sh
  ```

  neu zu bauen, da Anpassungen an die JonDoBrowser-Umgebung nötig sind. Danach muss die Datei noch ins JonDoBrowser-Repository kopiert werden (nach /trunk/build/patches/xpi).

- CHANGELOG update: Die CHANGELOG-Datei in /trunk ist zu aktualisieren.

- JDF profile extensions update: Solange das JonDoFox-Profil parallel von uns herausgegeben wird, muss beim Release des JonDoBrowsers geprüft werden, ob in der Zwischenzeit Aktualisierungen von Erweiterungen vorhanden sind. Wenn ja, gilt es das JonDoFox-Profil entsprechend zu aktualisieren (die JonDoFox-Erweiterung ist davon nicht betroffen, da das jeweilige Buildskript JonDoFox fürs Profil mit JonDoFox für den Browser ersetzt).

- Adblock patterns.ini: Siehe vorherigen Punkt. Gegebenenfalls sind die Adblock-Filter zu aktualisieren (siehe dazu auch den Eintrag beim JonDoFox-Profil-Release).

- JDB profile prefs update: Die prefs-Dateien in /trunk/build/langPatches müssen gemäß der JonDoBrowser-Version und den Erweiterungs-Versionen (z.B. von Adblock Plus und NoScript) angepasst werden. Siehe dazu auch den entsprechenden Schritt beim JonDoFox-Profile-Update.

- JDB version bump in build scripts: „jdbVersion“ in den Release-Build-Skripten für Linux und Mac OS X anzupassen. Unter Windows regelt das „JDB_VERSION“ im JonDoBrowser.nsi-Skript. Zu beachten ist, dass auch die JonDoBrowserExe.nsi angepasst werden muss, so dass die richtige Version angezeigt wird, wenn man sich die .exe im Explorer näher anschaut.

- Info.plist (version) (für Mac OS X): Wenn die minimal unterstützte Mac OS X-Version anzupassen ist, dann muss das in der Datei Info.plist in /trunk/build/mac geschehen.

- windows profile update: Da wir zur Zeit zwei verschiedene Windows-Installer (einen für das JonDoFox-Profil und einen für den JonDoBrowser) bauen, muss das Profil noch speziell für den JonDoBrowser angepasst werden. Dafür gibt es das Verzeichnis /trunk/build/win im JonDoBrowser-Repository. Die prefs-Dateien müssen nicht angepasst werden, die werden direkt wie unter Linux und OS X beim Bauen aus dem JonDoBrowser-Repository geholt. Wichtig für die deutsche Version ist es, das entsprechende Sprachpaket im Profil zu aktualisieren. Das ist auf ftp://ftp.mozilla.org/pub/firefox/releases/latest-esr/win32/xpi/de.xpi zu finden (SHA512-Summen sind unter ftp://ftp.mozilla.org/pub/firefox/releases/latest-esr/SHA512SUMS zu finden; Signatur prüfen!). Um es zu aktualisieren, geht man wie folgt vor (Annahme: man befindet sich im Verzeichnis trunk des JonDoBrowser):

  ```
  cd build/win/full/profile/extensions
  unzip -d test de.xpi
  cd test
  patch -tp1 < ../../../../../patches/xpi/XPI-Branding.patch
  zip -r9 ../langpack-de@firefox.mozilla.org.xpi *
  cd ..
  rm -rf test
  ```

Alle anderen Änderungen sind im Prinzip analog zu den im JonDoFox-Profil-Update beschriebenen durchzuführen, wobei natürlich das Mergen des aktuellen Firefox-Codes entfällt, da wir diesen selbst Kompilieren. Die Skripte, die für den Installer relevant sind, sind „JonDoBrowser.nsi“ und „JonDoBrowserExe.nsi“

- wiki update: Die Changelogs auf der Wiki-Seite sind zu aktualisieren.
- releasing: Die Binaries sind alle mit dem JonDos-Schlüssel zu signieren und in die entsprechenden Verzeichnisse hochzuladen.
- tagging: Der aktuelle Code im trunk-Verzeichnis kommt noch in einen SVN-Tag.
- extension update: Solange es noch keinen Updater für den JonDoBrowser gibt, wird ein Hinweis über die JonDoFox-Erweiterung eingeblendet. Wenn alle JonDoBrowser-Binaries hochgeladen sind und die Webseite angepasst wurde, dann ist die JonDoFox-Erweiterung für den Browser noch hochzuladen. Dazu nimmt man am besten die Datei jondofox.xpi aus dem JonDoBrowser-Repository und kopiert sie als jondofoxBrowser.xpi in das entsprechende Verzeichnis auf dem Webserver. Danach sind noch die Dateien updateBrowser.rdf und updateInfoBrowser.xhtml anzupassen, so wie das beim JonDoFox-Profil-Update im Prinzip beschrieben wurde.

3 Document history

Version	Status	Date	Authors	Description
0.01	in progress	21.03.04	Rolf Wendolsky	First version
0.02	in progress	23.03.04	Rolf Wendolsky	Inserted chapter “IDEs“
0.03	in progress	24.03.04	Rolf Wendolsky	Finished “IDEs“
0.04	in progress	25.03.04	Rolf Wendolsky	Added ’Compilation and installation of external libraries’ Index for ’Test environment’
0.05	in progress	30.03.04	Rolf Wendolsky	Added ’CppUnit’ configuration chapter
0.06	in progress	31.03.04	Rolf Wendolsky	Corrections in ’CppUnit’
0.07	in progress	08.04.04	Rolf Wendolsky	Added chapter about CVS
0.08	in progress	08.04.04	Rolf Wendolsky	Added ’Testconfiguration mixproxy’
0.09	in progress	15.04.04	Rolf Wendolsky	Different corrections in ’Test environment’ and ’Testconfiguration’
0.10	in progress	09.09.04	Rolf Wendolsky	Different corrections
1.00	released	25.11.04	Rolf Wendolsky	New libraries added
1.01	released	01.08.05	Rolf Wendolsky	JDKs updated
2.0alpha	in progress	04.08.05	Kuno G. Gruen	First translation into English
2.0beta	in progress	15.09.05	Kuno G. Gruen	Added configurations for Visual Studio .NET 2003 Eclipse 3.1; different updates and additions
2.01	in progress	18.09.05	Rolf Wendolsky	Fixed some errors
2.02	in progress	18.09.05	Kuno G. Gruen	Fixes some errors
2.03	in progress	17.10.05	Derek Daniel	Corrected english version
3.00	released	17.10.05	Rolf Wendolsky	Final review
3.01	released	22.11.05	Rolf Wendolsky	Version corrected
3.02	released	22.12.05	Kuno G. Gruen	Version corrected / updated
3.03	released	16.01.06	Rolf Wendolsky	Corrected MixConfig infos
3.04	released	26.01.06	Jonas Schießl	Added solution for compiling proxytest under Windows using Eclipse and Cygwin
3.05	in progress	09.02.06	Jonas Schießl	Description of FatJar plugin.
3.06	released	06.04.06	Rolf Wendolsky	Major corrections and reintegration of old german JBuilder instructions
3.07	released	09.05.06	Jonas Schießl	How to convert linebreaks

Version	Status	Date	Authors	Description
3.08	released	28.02.07	Elmar Schraml	translated Jbuilder instructions, added payment instande installation and configuration
3.09	released	14.03.07	Elmar Schraml	updated database installation; added AI configuration for prepaid system
3.10	released	06.11.07	Elmar Schraml	merged previous additions and fixes to AI/PIG installation
4.00	in progress	24.05.12	Georg Koppen	added release procedures for JonDo/JonDoFox
4.10	in progress	28.08.13	Stefan Köpsell	added release procedures for JAP

Teil II

Programmierrichtlinien AN.ON

Project AN.ON

Projekt AN.ON
Hannes Federrath, Universität Regensburg
Stefan Köpsell, TU Dresden

http://www.anon-online.de

Programming Standards

Stefan Köpsell
Rolf Wendolsky
Derek Daniel

Version: 3.04

from: 2. February 2017

Summary

This document contains the obligatory programming standards for developing the AN.ON project.

History

Version	Datum	Author(s)	Description
0.01	05.02.04	Rolf Wendolsky	First version
0.02	17.02.04	Rolf Wendolsky	Added Variable Names
0.03	17.02.04	Rolf Wendolsky	New document structure, Split "Code Structure and Documentation"; Naming Conventions, Threads, Primary Goals, Libraries

Version	Datum	Author(s)	Description
0.04	18.02.04	Rolf Wendolsky	Partially new structure, Added: Simple Data Types, Other Conventions, Testing, more details for Libraries; Version Management removed
0.05	18.02.04	Stefan Köpsell	A few notes
0.06	18.02.04	Rolf Wendolsky	Notes taken into account; small addition to "Testing" chapter
0.07	20.02.04	Rolf Wendolsky	Appendix
0.08	26.02.04	Stefan Köpsell	Minimal changes, first public version
1.00	26.02.04	Rolf Wendolsky	Changed format, made public
1.01	26.02.04	Rolf Wendolsky	Grammatic errors removed, minor corrections, formating
1.02	26.02.04	Rolf Wendolsky	Minor corrections
1.03	27.02.04	Rolf Wendolsky	Minor corrections
1.04	04.03.04	Rolf Wendolsky	2.1.1 changed
1.05	05.03.04	Rolf Wendolsky	Minor corrections
1.06	11.03.04	Stefan Köpsell	Minor corrections, Version number obtained from properties field
1.07	01.04.04	Rolf Wendolsky	Line length change, additional documentation, added to Naming Conventions for variables, Makefile, CppUnit chapter corrected
1.08	08.04.04	Rolf Wendolsky	Minor corrections
1.09			
1.10	10.09.04	Rolf Wendolsky	Testing of private members
1.11	22.09.04	Rolf Wendolsky	Module tests are now required; Changes to Testing of private members
1.12	23.09.04	Rolf Wendolsky	Minor corrections
1.13	01.10.04	Rolf Wendolsky	Static Code/Singletons
1.14	14.10.04	Rolf Wendolsky	Interface Immutable..., various corrections
1.15	18.10.04	Rolf Wendolsky	Update to Unittests and to Constants
1.16	07.11.04	Rolf Wendolsky	Corrections to JUnit
2.00	09.06.05	Rolf Wendolsky	General re-working: Removed "Status" from the History, Libraries->XML, Type Conventions->Threads, Goals, Internationalization, ...
3.00	26.10.05	Derek Daniel	Partially translated into english
3.01	22.11.05	Rolf Wendolsky	Nomenclature and editing of Java messages
3.02	16.01.06	Rolf Wendolsky	Nomenclature of images in Java, Coding conventions
3.03	08.10.06	Rolf Wendolsky	Coding conventions for if-statements
3.04	11.10.06	Rolf Wendolsky	StringBuffer replacement

Table of Contents

3.1 Introduction

This document describes the programming standards for the AN.ON project. It provides a framework for writing project-specific code. The project goals in Chapter 2 should be achieved with the help of this framework. The Sun Coding Standards are the general basis for the project programming standards; Differences and additional guidelines are given. Free external sources may deviate from these standards. Since AN.ON is essentially split among the Mixproxy, written in C++, and the JAP/InfoService/MixConfig, written in Java, the programming standards for the two parts differ slightly. However, unless otherwise noted, the following guidelines are identical for both the Mixproxy/C++ and JAP/InfoService/Mixconfig/Java.

The central AN.ON management will ensure that the standards are followed.

3.2 Primary Goals

The Programming guidelines support the primary goals listed in this section. The goals are listed in order of significance with the higher-ranked goal taking priority over the lower-ranked goal in case of a conflict.

3.2.1 Security

JAP/InfoService/MixConfig

In anonymity mode, only encrypted data may be sent and the data may only be sent to the first mix in a cascade. Other than that data, the client may communicate only with a certified Infoservice to obtain mix cascade IP addresses or to obtain public keys from mixes, or with the central update server in order to obtain automatic updates. This restriction may change in the future with the planned introduction of a payment system, which would require communication with the account instance.

The JAP client cannot be used as a backdoor or similar point of attack in the user's system.

Mixproxy

Exploits must be prevented at all costs. Denial of service attacks (DoS) must be prevented as much as possible. Code should be written so that all data sent to the mix can be examined in detail. The mix may only be controlled by the user who has administrator access rights to the Mixproxy.

3.2.2 Compatibility

JAP/InfoService/MixConfig

The portability of the complete program to as many other platforms as possible must be ensured.

Mixproxy

The portability of the source code to as many other platforms as possible must be ensured.

3.2.3 Performance

JAP/InfoService/MixConfig

Good performance is not a mission-critical goal for the JAP/InfoService/MixConfig part, but is always welcome. The quality of the code is more important in this case.

Mixproxy

Good performance in the Mix, however, is important and is a critical project goal. Therefore, speed is more important than robustness for this part of the code. Additional code tests may certainly be built into the debug mode, however.

3.2.4 Quality

The following points are listed in order of importance:

- Readability (...for other programmers)
- Maintainability (short turn-around for changes)
- Correctness (the correct output results from an appropriate input)
- Error tolerance (no crashes, useful error messages)
- Re-usability (...in other contexts / in other projects)

3.3 Structure

3.3.1 Package Structure

A specific directory and storage structure must be followed for the JAP/InfoService files and the Mix. Files that are no longer used should be removed immediately.

JAP/InfoService

The following directories are important for development:

- certificates ("trusted" root certificates)
- doc (Javadocs)
- documentation (automatically or manually created code documentation)
- help (documentation for the JAP user interface)
- images (pictures / graphics / icons for the JAP client)
- JapDll (libraries for special functions in Windows)
- src (source code)
- test (source code for test cases; alternative source code directory)

The source code uses the following main packet paths:

- anon (everything necessary for the connection to the Mix)
- forward (forwarder functionality and blocking resistance function)
- gui (GUI classes)
- jap (main functionality of JAP)
- infoservice (main functionality of the InfoService)
- jarify (classes for automatically creating jar files)
- logging (logging classes)
- misc (everything else)
- proxy (local proxy functionality)
- update (classes for Automatic Update)

The subpackages are arranged by their implemented functions.

Mixproxy

There are no packages in the first mix. For compatibility reasons, namespaces may not be used. However, classes may be organized by component name. Thus, the names of classes related to the Mixproxy all begin with CA.

Central settings, error codes, and includes that are used in every class can be found in the file StdAfx.h.

Test Cases – Java

Test cases and test classes must be placed in the test directory, which must also have the same packet structure as src. In the IDE, it will then appear that the production code and test code are in the same directory. In reality, the tests are separate from production code and are not allowed to be required for running the program.

For each package, there is a corresponding test package, for example jap.test, in the test directory, where the module tests for the package can be found. The module tests are thus visually separated (in the IDE) from production code, but correspond to each package.

Test Cases – C++

The test cases are in the test directory.

3.3.2 Makefile – Adding Classes (C++)

If classes are to be added to or removed from the project, it is not sufficient to simply do so within the IDE. You must also change the Makefile.am file (in the project's main directory) with the help of a text editor.

Be sure to save the file in UNIX format.

3.3.3 Code Structure

Each source file in the project has a required structure that will be described in the following sections.

Java

Java source files contain the following elements in exactly this order:

- disclaimer
- package statements
- <empty line>
- import of Java sources (zero, one or more)
- import of external library sources (zero, one or more)
- import of project sources (zero, one or more)
- <empty line>
- class and interface descriptions
- class and interface definitions

Imports with a * (for example, java.lang.*) are not permitted. Instead, each imported class must be listed individually. Imports that are no longer used must be removed immediately. Modern development environments can handle such removal automatically, for example, "optimize imports" in JBuilder.

C++ Header Files

Header files must contain the following components in exactly this order:

- disclaimer
- #ifndef __CLASSNAME_IN_UPPERCASE__
- #define __CLASSNAME_IN_UPPERCASE__
- <empty line>
- #include <project sources> (zero, one or more)
- <empty line>
- class description
- <empty line>
- #endif

There should be no methods implemented in header files (inline) unless the function is trivial (for example, get and set methods).

Macros (for example, #ifdef, #define) are only permitted to turn a specific functionality on or off (for example, Mix with or without a payment method). An #ifdef _DEBUG (together with #endif) is allowed for debug purposes.

C++ Program Files

Program files must contain the following elements:

- disclaimer
- <empty line>
- #include "StdAfx.h"
- #include "classname.hpp"
- #include <project sources> (zero, one or more)
- <empty line>
- class definition

3.3.4 Classes- / Interface Structure

Java

A class or interface should be constructed as follows, although not all parts are necessarily present, especially for interfaces. Also, some parts are only allowed under special circumstances.

- final public static constants
- final protected static constants
- final private static constants

- private static attributes
- private attributes
- constructor(s), including logical create-constructors
- public class functions (static)
- public member functions
- protected class functions (static)
- protected member functions
- package (without a label) class functions (static)
- package (without a label) member functions
- private class functions (static)
- private member functions

Anonymous classes should be avoided because they would adversely affect readability, maintainability and conformity.

C++

Class elements must be strictly ordered in the following order of visibility:

- public
- protected
- private

The elements within each visibility area (scope) must be ordered as follows:

- static const constants
- const constants
- attributes (static)
- attributes
- constructor(s), including logical create-constructors (for example create(), getInstance())
- destructor
- class functions (static)
- member functions

3.3.5 Methods

Normally, local variables should be declared at the beginning of a method. Variables that are only needed briefly (for example, in loops) should be declared later at the time they are used.

Tip: If a group of variables appears somewhere in the middle of a method, it's a good indicator that the method should be split into two methods.

3.4 Libraries

3.4.1 Internationalization (JAP/MixConfig)

All character chains in JAP/MixConfig can easily be internationalized, in other words, expressed in multiple languages. The property files JAPMessages and MixConfigMessages serve this purpose. All strings that could appear in dialogs for the user must be deposited in these files. These strings can then be called in the program through the gui.JAPMessages class. The class also allows the use of placeholders for variables.

The property files must be edited with the JRC editor which is available at
http://www.zaval.org/products/jrc-editor
It is based on Java and should run on all currently available systems.

3.4.2 Logging

The JAP/InfoService/MixConfig and Mixproxy use their own logging components. For this reason, System.out.println() and printStackTrace() (in Java) and cout (in C++) are forbidden when code is checked in!

JAP/InfoService/MixConfig

The class logging.LogHolder is responsible for logging. Output occurs through the static methods
log(int logLevel, int logType, String message)
log(int logLevel, int logType, Exception exception)
The logLevel is entered as a static constant in the LogLevel class and is accessible as LogLevel.LOGLEVEL. There is a total of eight different log levels that can be used depending on the importance of the output: DEBUG, INFO, NOTICE, WARNING, ERR, EXCEPTION, ALERT and EMERG. Only the outputs whose log level is equal to or higher than the configured log level will be written to the log file.
The logType is a static constant in the LogTypes class and is accessible as LogTypes.LOGTYPE. There are four types of logging, depending on which part of the JAP client is outputting the log: GUI, NET, THREAD and MISC. The constants can be added to each other if the output is relevant to more than one part of the JAP client.
The global log setting DETAIL_LEVEL also determines the level of logging detail. On the highest setting, the class, method and even the line of code from which the logging message was sent will be output. Therefore, localization information (classes or method names) in the log message does not make sense and is not allowed.

Mixproxy

Log outputs occur through a method in the CAMsg class:
CAMsg::printMsg(UINT32 type, char* message)
The following log types are available as macros in CAMsg: LOG_DEBUG, LOG_INFO, LOG_CRIT, LOG_ERR. Only the outputs whose log level is equal to or higher than the configured log level will be written to the log file.

3.4.3 XML (Java)

The class anon.util.XMLUtil must be used for XML operations if possible. Objects that can be manipulated in XML must implement the anon.util.IXMLEncodable interface and should also contain the static attribute
public static final String XML_ELEMENT_NAME
which contains that name of the XML element created.

3.4.4 Images (JAP/MixConfig)

Pictures and graphics are centrally located in the images directory. If graphics with only 16 colors exist, they are placed in the lowcolor subdirectory. The filename can then be entered in the JAPConstants class and the graphic can be loaded using the
GUIUtils.loadImageIcon(String strImage, boolean sync)
GUIUtils.loadImageIcon(String strImage)
methods.

3.4.5 Other Resources (JAP/InfoService/MixConfig)

Other resources (texts, binary data, URLs, ...) can be loaded by methods of the
anon.util.ResourceLoader class.

3.4.6 External Libraries

External libraries are programs or source code packages that do not originate from the AN.ON developer community. New external libraries may only be used after prior arrangements with the central AN.ON management. Furthermore, "exotic" libraries that only run on certain platforms may not be used. Code that excludes commercial use, GPL for example, are also not permitted.

JAP/InfoService/MixConfig

The source code is compiled with Java version 1.1.8_010. Libraries, classes and methods from newer Java versions are thus not allowed.

Mixproxy

The STL (Standard Template Library) may not be used. Of course, other proprietary libraries (for example MFC) are also not permitted.

3.5 Naming Conventions

The naming conventions presented here apply to all labels such as constants, attributes, etc. The labels may have to be supplemented with prefixes. All labels should clearly indicate their purpose. This topic is covered by the subchapters that follow.

3.5.1 Constants

Constants are capitalized. Multiple words are connected by underscores.
Example:

```
final int SPECIAL_INT_CONSTANT = 0x34;  // Java
const SINT32 MY_INT_CONSTANT = 0x34;  // C++
```

3.5.2 Variables

Each variable must be named according to the following schema:
<prefix>_<type>Name
Variable names must contain information about their scopes. Thus, a prefix is added to each variable name according to the following table:

Variable Scope	Prefix	Declaration
Local Variable		within a method
Member Variable	m	within a class
Static member Variable	ms	within a class, static Attribute
Argument	a	argument for a method
Return value	r	argument for a method, used as a return value

In addition to scope, the variable type must be integrated into the name. In most cases the programmer can freely choose the label. The following data types are an exception:

Variable Type	Type Label
bool, boolean	b
char*, string, String	str

Elementary types (numbers) do not need a type label. The type label can also be left off if the name length becomes long and unwieldy (for example, m_JapClientWindowEventHandlerTemp). Examples for correct naming:

```
private static String ms_strName;
```

C++: the symbol for reference types (&) and pointers (*) is always directly after the type, not by the variable name. Example: char* m_strName = "Caesar";

3.5.3 Methods

A method name or function name consists of alphabet symbols where the first symbol is a small letter. The first word describes the functionality in the form of a verb. Words are separated by always capitalizing the first letter of each word (with the exception of the leading verb). It should be recognizable from the name whether the method is one that makes changes or one that does not make changes. For example, getNextAndIncrement() is easily recognizable as a method that alters values. On the other hand, one expects that the method getTheMix() does not change the object.

3.5.4 Classes / Interfaces

Class names consist of one or more words written together where each word must begin with a capital letter, for example, JapCascadeMonitorView. The name of a class should indicate an object (for example CASocket) or a handler (for example, CAListCellRenderer) and should describe the actual function of the class.

Abstract classes get the prefix Abstract, for example, AbstractConnection.

Java

Interfaces have the "I" prefix, for example, IConnection. The name should normally indicate a property, for example, IDistributable.

C++

All class names related directly to the Mixproxy must begin with CA.

3.5.5 Message Properties (JAP/MixConfig)

Message properties are always named after the class where they are defined:

packagename.classname_messageLabel = ...

Each property must be defined in one and only one class as a constant defined like this:

static final String MSG_LABEL = ClassName.class.getName() + "_messageLabel";

The message label must not contain other characters than ASCII letters and starts with a lowercase letter. If it consists of more than one word, the words are separated by writing each first letter uppercase. The constant name always starts with "MSG_".

If a property is used in many classes of a package, it is either placed in the class it logically belongs to, or, if this is not clear, in a common class named "Messages".

Messages are then loaded, for example, by writing

String strMessage = JAPMessages.getString(MSG_LABEL);

Messages that are not defined in any class may be deleted.

3.5.6 Image File Names (JAP/MixConfig)

The naming convention for images corresponds to the one of the message properties:

static final String IMG_LABEL = ClassName.class.getName() + "_imageLabel";

The differences are that an image file contants starts with „IMG_" and, if there is no logically best class to place it, it has to be defined in a class named „Images".

Images are then loaded, for example, by writing

ImageIcon icon = GUIUtils.loadImageIcon(IMG_LABEL);

Images that are not defined in any class may be deleted.

3.5.7 Test Classes

The test class names should indicate the name of the tested class or module with "Test" added to the end, for example, CAMsgTest for the CAMsg class.

3.5.8 Class File Name Extensions

C++ header files always end in .hpp and the implementation of the class is always found in the respective .cpp file. The extension .java is always used for Java files.

Only one class definition is allowed per file. The class name must be the same as the file name without the file name extension.

3.6 Type Conventions

3.6.1 Simple Data Types (C++)

To achieve platform independence for elementary data types, only the following data types are used:

Data Type	Bytes	Value Range
SINT8	1	-128..127 or all ASCII symbols up to 127
UINT8	1	0..255 or all ASCII symbols
SINT16	2	-32768..32767
UINT16	2	0..65535
SINT32	4	-2147483648.. 2147483647
UINT32	4	0..4294967295
UINT64	8	0..18446744073709551615 > 1019
float	4	-1038..1038
double	8	-10308..10308

The special forms, SINT and UINT, reflect either the 16- or 32- versions and should never be used due to platform dependence!

3.6.2 Constants (Java)

It should be noted that constant object references defined with final in Java only make the reference constant. The object itself can still be altered. To duplicate const behavior (from C++) in Java, an Immutable<Classname> interface must be created for each class. The interface contains only the non-changing methods that for this class. A constant object can then be created as follows, for example:

final ImmutableMyClass A = new MyClass();

This object is thus a true constant, since the public attribute is not allowed (see below).

3.6.3 Variables

Variables on the class level (member and static variables) must always be declared private. public and protected are not permitted except for constants. Access to these variables should always proceed through get/set<variable name without prefix> methods, for example, getName() for the attribute m_strName. Constants that are also allowed to be accessed directly are an exception to the rule. For boolean attributes, the reading method is is<variable name without prefix and without b>, for example isReady() for the attribute bool m_bReady.

3.6.4 Methods

Arguments that can be given as references (C++: type&, type*; Java: every object) and are not allowed to be changed by the method must be declared as constant arguments if possible.

C++: Methods that do not change the state of their own objects must be defined as const.

3.6.5 Classes / Interfaces

Static Code

Static code should be avoided in classes, since it generally weakens the object orientation and clarity of the code structure.

Singletons

Classes with a singleton design patter should only be implemented if absolutely necessary, since code dependent on this class is usually not testable or is testable only with extreme difficulty (through Unittests). Furthermore, singletons make flexibility or swapping of code more difficult and generally weaken the object oriented structure of the code.

"final" Classes (Java)

Classes should always be declared final by default in order to improve performance. If a class needs to be inheritable later, the final attribute can then be removed.

Templates (C++)

Templates are not allowed for compatibility reasons.

3.6.6 Threads

All classes and structures that are simultaneously used by more than one thread must be implemented thread safe. To create thread safe code, you absolutely must consult current literature because this issue is not trivial!

Java

The Runnable interface must be implemented for threads. Inheriting from Thread is not allowed because it can lead to faulty behavior in some JVMs. The keyword synchronized must be used to ensure thread safety.

Thread.interrupt() works only under certain conditions. In particular, you must be sure that Thread.isInterrupted() is tested for in the thread's loops and that I/O operations contain a timeout or other similar possibility of interrupting a thread. Otherwise it depends on the JVM whether a thread can be ended or not.

C++

The CAThread class is used for instantiating a thread. Thread safety can be achieved by using the lock() and unlock() methods of objects from the CAMutex class.

3.6.7 Exceptions

Java

When exceptions are thrown, they should specify the error as precisely as possible. Whenever possible, no new exceptions should be defined unless they offer real functionality.

C++

C++ exceptions are not allowed due to compatibility issues. Instead, each method must return a value of SINT32 type, which represents an error code. If no error occurs, the constant E_SUCCESS is returned.

The error codes are defined in StdAfx.h and new error codes can be added as necessary. However, new error codes should only be added upon arrangement with the central code management.

The actual return values of a method are generally returned using a pointer or reference in the method arguments. For example:

SINT32 doSomething(UINT32 a_u32Value, Object& r_Object)

A return value other than SINT32 will be tolerated if necessary for trivial methods (for example, get and set methods).

3.6.8 HTML-Formated GUI Elements (JAP/MixConfig)

Some GUI elements (labels, buttons, etc.) can be formatted with the help of HTML tags. When this is done, all tags are to be written in lower case and without spaces, for example:

```
JLabel label=new Jlabel("<html><body><b>A test in bold...</b></body></html>");
```

Otherwise it can not be guaranteed that every JDK will correctly display the formatting.

3.7 Documentation

3.7.1 Documentation Tool

Javadoc is the minimum documentation standard. Javadoc allows the creation of HTML-based documentation based on the source code. However, Doxygen is used for documentation creation because it posses further functionality and also supports Javadocs under C++ [DOXYGEN].

3.7.2 Procedure

The documentation for classes and methods should always be written before the actual implementation. Based on experience, little is documented otherwise. When the code is changed, the documentation must also be changed (before the code is changed if possible). Note: In C++, the documentation for classes, attributes, and member functions must appear only in the header files.

Multi-line comments must be written directly before the code to be documented as follows:

```
/**
* ....(comments)
* @tag ....(comments)
*/
```

One or more so-called tags, which implement special functions, may appear within the comments preceded by an @.

A single-line comment must also be written directly before the code to be documented as follows:

```
/// ....(comments)
```

The documentation language is English (UK).

3.7.3 Classes

The class and interface descriptions should describe the purpose of the class or interface. If a short example is needed for clarity, it can be included as part of the description.

3.7.4 Attribute (@param Tag)

Only attributes whose function is not recognizable by their names need to be documented. Attributes are to be described within the comments of the method using the @param tag: @param attribute-name description

3.7.5 Return Values (@return Tag)

Return values are to be described within the comments of the method using the @return tag: @return description

3.7.6 Methods

The functionality, input parameters, and if necessary the result are to be documented for methods. Furthermore, exceptions and errors must be documented including when they could occur and which values they could return.

Units (for example, whole number percentages) and allowed value ranges must also be commented.

Note:

If several methods interact with each other within a class, it may be advantageous to explain the interactions once in the class/interface documentation and then only describe the actual purpose of the method for the method description itself. If this is done, @see must be used to link the corresponding class and method documentation together:

@see myClass()

@see myMethod() (arguments must not be given here.)

Tip:

When implementing interface methods or overwriting methods, the documentation does not need to be repeated. Special cases can be documented if desired. Here too, an @see tag must indicate the implemented interface, for example: @see java.lang.Comparable.method().

3.7.7 Algorithms

The code within methods, from here on referred to as an algorithm, must also be commented unless it is trivial (for example, if it is only a value being assigned). However, each individual line should not be commented, rather whole code blocks. The comments are not a repeat of the code.

When algorithms are taken from external sources (for example, an implementation of quicksort), the source of the algorithm must also be given in the comment.

If an algorithm requires very detailed comments, consideration should be taken whether the algorithm can be simplified so that it is more easily understood. Experience shows that someone will eventually bring the code into a "clean" form, but adjusting the comments is often forgotten! Furthermore, "hacks", "fixes" and similar are generally not tolerated.

3.7.8 @todo Tag

Generally there are things in source code yet to be done, that have been put off for a later date. These areas should be marked with an @todo tag so that they are also visible in the javadoc documentation: /** @todo reason */

3.8 Other Conventions

3.8.1 Code Style

Die folgenden Codestyle-Regeln können normalerweise von der IDE automatisch auf den Code angewandt werden [ENTWUMG]. Dennoch sollte jeder Programmierer sie berücksichtigen.

Line Length

Die maximale Länge von Programmzeilen beträgt 110 Zeichen. Dadurch ist gewährleistet, dass die Seiten auch auf kleinen Bildschirmen gut darstellbar und noch gut zu drucken sind, was beispielsweise für Code-Review etc. notwendig ist. Abweichungen sind nur zulässig, wenn aufgrund der Übersichtlichkeit des Quelltextes mehr Zeichen erforderlich sind. Ein Beispiel dafür wäre die matrixähnliche Anordnung von Code bei der Definition eines Zustandsgraphen oder Konfigurationen einer Maske.

Wenn die Liste der Funktionsargumente zu lang für eine Zeile ist, wird sie am Zeilenende umgebrochen und in der nächsten Zeile weitergeschrieben. Diese Zeile sollte um mindestens zwei Tabulatoren eingerückt werden. Sind weitere Zeilen nötig, so liegen diese auf der gleichen Höhe.

Tab Size

Die Einrücktiefe bei Verschachtelungen beträgt 4 Leerzeichen, bzw. einen Tabulator. Es werden grundsätzlich Tabulatoren zur Einrückung verwendet.

Brace Setting

Öffnende geschweifte Klammern stehen durch ein Leerzeichen getrennt direkt unter dem auslösenden Code und auf der gleichen Einrücktiefe. Schließende geschweifte Klammern stehen in einer eigenen Zeile ebenfalls auf der gleichen Einrücktiefe wie der auslösende Code. Die Anweisungen innerhalb der Klammern sind einmal eingerückt. Auf Höhe der Klammern stehen keine Anweisungen.

Beispiel:

```
public String getName()
{
  return m_strName;
}
```

Öffnende runde Klammern stehen bei Methoden direkt hinter dem auslösenden Code, schließende runde Klammern direkt hinter dem eingeschlossenen Code. Bei Schlüsselwörtern sind die runden Klammern durch ein Leerzeichen vom Schlüsselwort getrennt.

Beispiele:

```
setGlobals(aParam1, aParam2, aParam3);
while (myObject != enumeration.next());
```

Methodenparameter

Kommas stehen direkt hinter Methodenparametern. Zum nächsten Parameter wird jeweils ein Abstand von einem Leerzeichen gelassen (Beispiel siehe oben).

3.8.2 Coding Conventions

- Pro Zeile darf aus Gründen der Übersichtlichkeit nur ein Statement kodiert werden. Ausnahme ist die Deklaration von Bedingungen für for-Schleifen, die in einer Zeile geschrieben werden darf.
- Es sollte nur ein return pro Methode verwendet werden. Weitere return-Anweisungen sind zulässig, wenn am Anfang einer Methode die Vorbedingungen abgeprüft werden, bzw. wenn der Code sonst sehr unübersichtlich würde.
- Der ?:-Operator sollte nicht verwendet werden, da er das Debuggen unnötig erschwert.
- Grundsätzlich sollten möglichst wenige Objekte erzeugt werden, um die Performance nicht unnötig zu drücken.
- Konstanten sollen nie in Form von konkreten Zahlenwerten bzw. expliziten Characterstrings angegeben werden, sondern immer über den „Umweg“ einer Klassenvariable. Dies erleichtert eine spätere Umbenennung enorm und vermeidet Tippfehler.
- If(...)-chains with only one possible condition must be written as if()... else if()... or as switch() statement (but may not be written as consecutive if...if...if...)
- Boolean values in if-statements must be written as if(boolean) but not as if(boolean == true | false)

Java

- Statt der Stringverkettung mit „+“ sollte in häufig durchlaufenen Schleifen ein anon.util.MyStringBuilder verwendet und mit der append(..) – Methode aufgebaut werden. Dadurch wird die Performance im Allgemeinen gesteigert.
- Referenzen auf Objekte, die nicht mehr benötigt werden, sollten sofort gelöscht werden, um dem Garbage Collector die Möglichkeit zu geben, den Speicher aufzuräumen (objct = null). Dies ist mit finalize() nicht sichergestellt, weshalb finalize() nicht verwendet werden darf.
- Be warned when using Enumerations: In case a second thread alters the number or the order of elements in the instance that generated the Enumeration, the Enumeration is not updated! This means, for example, if you remove an element from a Vector, the Enumeration skips the next element or may even throw an Exception if there is none. The solution is to synchronize on the object whenever there is the possibility of competing threads.
- Manipulation of GUI elements is only allowed in the AWT event thread. Otherwise, it might have no effect on the GUI. To test if your code is executed in the event thread, call SwingUtilities.isEventDispatcherThread(). If the code is not executed in the event thread, you may use SwingUtilites.invokeLater() or SwingUtilities.invokeAndWait() to manupulate the GUI. On the other hand, no time-consuming code may be executed in the event thread, as this would block the GUI.
- If you need to put manual breaks into HTML labels like JLabel, use <p> instead of
, as Java 1.1 ignores
 when computing the preferred width.

C++

- Grundsätzlich sollten eher dynamisch allokierte, das heißt über new instanzierte, Objekte verwendet werden als per Typdefinition instanzierte. Dies stellt sicher, dass das neu erzeugte Objekt so im Speicher ausgerichtet wird, wie es für die jeweilige Zielplattform aus Performancesicht am Günstigsten ist.
- RTTIs (Run Time Type Informations) dürfen aus Kompatibilitätsgründen nicht verwendet werden, sollten also bereits im Compiler abgeschaltet werden.
- Dynamisch angelegte, nicht mehr gebrauchte Zeiger auf Objekte sollten sofort gelöscht werden, da es keine Garbage Collection gibt und der Speicher nicht von selbst freigegebenen wird.

3.9 Testing

3.9.1 Introduction to Unittests

Um die Funktionalität der Software sicherstellen zu können, sind als qualitätssichernde Maßnahmen Entwickler- bzw. Modultests und Integrationstests durchzuführen. Außerdem werden Lasttests benötigt, um die Performance-Anforderungen an die Software zu überprüfen.

Teile dieser Tests sind automatisierbar: nämlich die auf Modulebene, auch Unittests genannt. Ein Modul kann dabei eine einzelne Methode, Klasse, Package oder auch das Zusammenspiel mehrerer dieser Module sein. Bevor ein Entwickler eine neue Version seines Moduls veröffentlicht, d.h. unter die Versionskontrolle stellt, sollte dieses in Form eines Modultests getestet werden. Im Rahmen der Integrationstests, die sämtliche Testfälle umfassen, wird dann sichergestellt, dass der aktuelle Stand der freigegebenen Module fehlerfrei zusammenspielt.

Die Modultests liegen in Form von sogenannten Testsuiten vor, die sich meist auf eine konkrete, zu testende Klasse beziehen und mehrere Testfälle auf die Methoden der Klasse zusammenfassen. Jeder Testfall bzw. Testcase enthält ein oder mehrere Tests. Hier werden, ähnlich dem bekannten ASSERT Makro, Bedingungen auf Gültigkeit geprüft. Die Tests liegen normalerweise in einem vom Produktivcode getrennten Verzeichnis „test“. Der Produktivcode muss völlig unabhängig von den Testfällen sein.

Es sollte mindestens ein Testfall pro Methode vor deren eigentlicher Implementierung (!) geschrieben werden. Pro Klasse sollte mindestens eine Testsuite angelegt werden. Auch für das Zusammenspiel von mehreren Methoden und Klassen können eigene Testsuites erforderlich sein. Während der Entwicklung können mit neuen Anforderungen auch neue Testfälle hinzukommen.

Das Konzept der Unittests wurde ursprünglich von Kent Beck und Erich Gamma im Rahmen des Extreme Programming entwickelt. Für die Durchführung von Unittests existieren frei verfügbare Open-Source Bibliotheken. Wir verwenden für Java JUnit [JUNIT] von Kent Beck und Erich Gamma bzw. für C++ CppUnit [CPPUNIT] von Sourceforge.net. Über dieses Dokument hinausgehende Hilfe zu Unittests bzw. JUnit findet man beispielsweise unter [UNITTESTS]. Weiterführende Informationen zum Testen objektorientierter Software findet man unter [OOTESTEN].

Die Modultests sind für neue Klassen verpflichtend, soweit sie technisch möglich sind.

3.9.2 JUnit

General Information on JUnit

JUnit basiert auf Testfällen, die durch die Klasse TestCase gruppiert werden. Mehrere Gruppen von Testfällen können im Kontext einer TestSuite abgearbeitet werden. Beide Klassen implementieren das Interface Test.

Um JUnit verwenden zu können, muss jeweils junitx.framework.PrivateTestCase importiert werden. Für JAP/InfoService wird die Klasse junitx.framework.extension.XtendedPrivateTestCase vorgeschrieben.

Test Classes

Jede Testklasse ist von PrivateTestCase bzw. XtendedPrivateTestCase abgeleitet. Sie sollte in einem Verzeichnis Packagename.test liegen und sich auf eine konkrete, zu testende Klasse im Package Packagename beziehen. Ihr Name setzt sich aus dem Namen der zu testenden Klasse plus Test zusammen (ClassnameTest). Sie kann folgende Methoden enthalten:

public void setUp() wird vor jedem Test ausgeführt
public void tearDown() wird nach jedem Test ausgeführt
public void testX() ein oder mehrere Testfälle

Der (einzige) Konstruktor sollte einen String name als Argument haben und selber super(name) aufrufen.

Test Cases

Die Methodennamen von Testfällen müssen immer mit dem Präfix test beginnen und die folgende Signatur haben:

public void test<Testfallbezeichnung>()

Dies ist notwendig, damit die Testfälle automatisch über den Java-Reflection-Mechanismus gestartet werden können.

In den Implementierungen der Testfälle können die Tests über assert-Funktionen der TestCase-Klasse ausgeführt werden, z.B. TestCase.assertTrue oder TestCase.assertEquals. Schlägt eine assert-Annahme fehl, wird dies mitprotokolliert.

TestSuite

Im Rahmen einer TestSuite werden beliebig viele TestCase ausgeführt und ausgewertet. JUnit bietet verschiedene Umgebungen, in denen die Tests abgearbeitet und ausgewertet werden können. Wir verwenden den junit.swingui.TestRunner.

Dabei geht man wie folgt vor:

- Erzeuge im Verzeichnis Packagename.test (wenn das zu testende Modul im Package Packagename liegt) eine Klasse mit der Bezeichnung Alltests.
- Implementiere eine Methode static Test suite(), welche ein TestSuite Objekt erzeugt (mit dem Konstruktorparameter „Packagename“), diesem Objekt alle Testfallklassen für das zu testende Paket mittels der Methode addTestSuite(<Testklassenname>.class) und alle Suiten der Unterpackages mittels der Methode addTest(Packagename.AllT hinzufügt und es schließlich an den Aufrufer zurückgibt.
- Implementiere eine main-Methode, welche die Tests mittels der Methode
  junit.swingui.TestRunner.run(AllTests.class)
  durchführt.

Sämtliche Unittests im AN.ON-Projekt befinden sich in einem gesonderten Verzeichnis test. Die Packages in diesem Verzeichnis korrespondieren zu denen im Produktivcode. In den Verzeichnissen des Produktivcodes dürfen keine Testfälle abgelegt werden.

Testing Private, Protected and Package-scoped Members

Das Testen von private, protected und package scoped Members (im Folgenden allgemein als private Members bezeichnet) ist über das Framework JUnitX möglich [JUNITX].

Ein Aufruf der Methode myMethod eines Objektes der Klasse <TestedClass> und die anschließende Prüfung der privaten Variable m_value erfolgt beispielsweise so:

```
Object m_<TestedClass>Object = newInstance("packagename.<TestedClass>", NOARGS);
Object result = invokeWithKey(m_<TestedClass>Object, "myMethod", NOARGS);
assertEquals(asBoolean(result), true);
assertEquals(getInt(m_<TestedClass>Object, "m_value“), 1);
```

Im Paket junitx.framework.extension werden Hilfsklassen bereitgestellt, die speziell das Testen privater Members nochmals vereinfachen:

junitx.framework.extension.TestProxy:

In jedem Package, in dem sich zu testende Klassen mit private Members befinden, muss eine Klasse mit exakt diesem Namen (TestProxy) von TestProxy abgeleitet und in das Verzeichnis test/packagename kopiert werden. Dabei darf diese Klasse nur folgende Zeilen als Inhalt haben (wobei packagename natürlich angepasst werden muss):

```
package packagename;

import java.lang.reflect.*;

public class TestProxy extends junitx.framework.extension.TestProxy {

  protected Object createInstance(Constructor a_Constructor, Object[] a_args)
    throws Exception
  {
    return a_Constructor.newInstance(a_args);
  }
}
```

Da die Klasse TestProxy nirgendwo direkt verwendet wird muss, damit der TestProxy auch automatisch kompiliert wird, mindestens eine Referenz auf diese Klasse in einer anderen Klasse angelegt werden. Dazu bietet sich die Klasse AllTests im jeweiligen Unterpackage test an. In die Klasse AllTests sollten folgende Zeilen aufgenommen werden:

```
...
import java.lang.reflect.*;
...
TestProxy m_proxy;
```

3.9.3 CppUnit

General Information on CppUnit

CppUnit ist ähnlich aufgebaut wie JUnit und basiert auf der Klasse TestFixture, die einzelne Testfälle zusammenfasst. Um CppUnit komfortabel verwenden zu können, müssen folgende includes gesetzt werden:

```
#include <cppunit/extensions/HelperMacros.h>
#include <cppunit/TestFixture.h>
```

Test Classes

Jede Testklasse ist von CppUnit::TestFixture abgeleitet und sollte sich auf eine konkrete, zu testende Klasse beziehen. Sie kann folgende (public) Methoden enthalten:

```
void setUp()      wird vor jedem Test ausgeführt
void tearDown()   wird nach jedem Test ausgeführt
void testX()      ein oder mehrere Testfälle, beschrieben durch X
```

Nach den Methoden sollte immer folgendes Makro folgen, damit die Tests auch ausgeführt werden können:

```
CPPUNIT_TEST_SUITE(ClassnameTest);
CPPUNIT_TEST(testX);
CPPUNIT_TEST(testY);
...
CPPUNIT_TEST_SUITE_END();
```

Dabei muss ClassnameTest dem Namen des TestCases entsprechen, und testX, testY, ... dem Namen der zu testenden Testfälle.

Test Cases

Die Methodennamen von Testfällen sollten immer mit dem Präfix test beginnen und die folgende Signatur haben:

```
public void test<Testfallbezeichnung>()
```

Die Tests können über Makros durchgeführt werden. Das am Häufigsten verwendete Makro ist

```
CPPUNIT_ASSERT_EQUAL(expected, result);
```

wobei expected und result den == Operator zum Vergleich und den << Operator zur Ausgabe überschrieben haben müssen.

Analog kann mit dem Makro

```
CPPUNIT_ASSERT(expression);
```

der Wahrheitswert eines boolschen Ausdrucks ermittelt werden.

Durch Anhängen von MESSAGE kann im Fehlerfall eine zusätzliche Ausgabe erzeugt werden, wobei message eine Zeichenkette sein muss:

```
CPPUNIT_ASSERT_EQUAL_MESSAGE(message, expected, result);
CPPUNIT_ASSERT_MESSAGE(message, expression);
```

TestSuite

Die Tests können in mehrere Packages (TestSuites) aufgeteilt werden. Dazu ist für jedes Package eine Klasse AllTests<Packagename> anzulegen, die sämtliche Testklassen ausführt, und eine Oberklasse AllTests, die alle anderen AllTests<...> ausführt.

Auch AllTests ist von CppUnit::TestFixture abgeleitet und enthält eine nur folgende Methode:

```
static CppUnit::Test* suite()
```

In dieser Methode befinden sich folgende Aufrufe:

```
CppUnit::TestSuite* suiteOfTests = new CppUnit::TestSuite( "AllTests<...>" );
```

```
suiteOfTests->addTest( DummyClass1Test::suite() );
suiteOfTests->addTest( DummyClass2Test::suite() );
...
```

In der ersten Zeile definiert sich die Klasse selbst; es muss exakt der Klassenname eingetragen werden. Die nächsten Zeilen sind die Testklassen, die im Rahmen dieser Suite ausgeführt werden sollen.

Testing Private and Protected Members

Sollen private oder protected Memberfunktionen einer Klasse getestet werden, so müssen die Testklassen in der zu testenden Klasse als friend deklariert werden.

3.9.4 Dummy and Mock Objects

Manchmal müssen Methoden oder Objekte getestet werden, die wiederum ein oder mehrere Objekte als Argument benötigen. Um den Test nicht unnötig zu verkomplizieren oder auch um ihn überhaupt erst möglich zu machen entwirft man speziell für diesen Zweck sogenannte Dummy-Objekte. Diese sind von der Klasse der „richtigen“ Argument-Objekte abgeleitet, besitzen aber keine wirkliche Funktionalität. Eine gute Beschreibung, wie man solche Dummy-Objekte geschickt implementiert, findet man hier:

http://www.dpunkt.de/leseproben/3-89864-150-3/Kapitel_6.pdf

3.10 Literature Cited

Textreferenz	Titel
[CODING STANDARDS]	http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
[CPPUNIT]	http://cppunit.sourceforge.net
[DOXYGEN]	http://www.doxygen.org
[ENTWUMG]	Entwicklungsumgebung.pdf
[JUNIT]	http://www.junit.org
[JUNITX]	http://www.extreme-java.de
[OOTESTEN]	Uwe Vigenschow: Objektorientiertes Testen und Testautomatisierung in der Praxis, dpunkt.verlag, 1.Auflage 2005 http://www.oo-testen.de
[UNITTESTS]	Johannes Link: Unit Tests mit Java, dpunkt.verlag, 1.Auflage 2002 http://www.dpunkt.de/utmj/

3.11 Appendix (Summary)

P

- Sicherheit
- Kompatibilität
- Performance
- Qualität

S
Zum genauen Aufbau von Klassen und Methoden siehe Kapitel 3.3.

L

- Zur Textdefinition-und Ausgabe die Klasse JAPMessages benutzen.
- Bilder laden: JAPUtil.loadImageIcon(String strImage, boolean sync)
- JAP: Logging über logging.LogHolder.log(level, type, message)
- Mix: Logging über CAMsg::printMsg(UINT32 typ, char* format)
- XML (Java): die Klasse anon.util.XMLUtil verwenden
- Keine proprietären Bibliotheken / keine STL verwenden

N

- Konstanten nur mit Großbuchstaben: const int SPECIAL_INT_CONSTANT;
- Variablen nach dem Schema: <Präfix>_<Typbezeichner>Name

Variablenscope	Präfix
Lokale Variable	
Membervariable	m
Statische Variable	ms
Argument	a
Rückgabewert	r

- Referenzsymbole direkt hinter den Typ: UINT8* value; UINT8& value = a;
- Methoden: mehrere Wörter, erstes Zeichen klein, jedes weitere Wort beginnt mit Großbuchstaben: getNextAndIncrement()
- Klassen: ein oder mehrere Wörter, beginnen jeweils mit Großbuchstaben: JapCascadeMonitorView
- Interfaces: IConnection
- Abstrakte Klassen: AbstractConnection
- Mix – Klassen beginnen mit CA: CAMsg
- Testklassen haben angehängtes Test: CAMsgTest
- Dateiendungen: .java, .hpp, .cpp

Type Conventions

- Zu den einfachen Datentypen siehe Tabelle in Kapitel 3.6.
- Java: Konstante Objekte nur über Interface definieren: Immutable<Classname>;
  nicht-konstante Methoden im Interface auslassen;
  Instanzierung über final ImmutableMyClass A = new MyClass();
- Variablen: nur private, public und protected sind verboten (außer in Strukturen)!
- Methoden: so konstant wie möglich definieren
- Klassen: möglichst kein statischer Code und keine Singletons
- Java-Klassen: immer als final deklarieren
- C++-Klassen: Templates sind verboten
- Threads: Runnable implementieren, bzw. die Klasse CAThread verwenden
- C++-Ausnahmen: jede Methode gibt Fehlercode nach folgendem Schema zurück:
  SINT32 doSomething(UINT32 a_u32Value, Object& r_Object)

D

- Standard ist javadoc
- Dokumentation wird vor der Implementierung geschrieben
- Sprache ist Englisch (UK)
- Todo-Tag benutzen, wenn volleFunktionalität nicht implementiert ist:
  /** @todo Begründung */

O

- maximale Zeilenlänge:110 Zeichen
- Funktionsparameter über mehrere Zeilen mit zwei Tabs einrücken
- Einrücktiefe: ein Tabulator (4 Leerzeichen)
- geschweifte Klassern {}: in die nächste Zeile, beide Klammern auf gleicher Höhe
- runde Klammern (): direkt hinter den Auslöser, bei Schlüsselwörtern ein Leerzeichen Abstand
- Komma steht direkt hinter Methodenargumenten, danach ein Leerzeichen
- siehe auch Kapitel 3.8.2 zu C!

Test

- Unittests sind vorgeschrieben
- Testbibliotheken: JUnit bzw. CppUnit
- Tests müssen vor der eigentlichen Implementierung geschrieben werden
- zur genaueren Beschreibung siehe Kapitel 3.9