Project AN.ON

Programming Standards

Stefan Köpsell
Rolf Wendolsky
Derek Daniel

Version: 3.04

from: 11. October 2006

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

http://www.anon-online.de

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
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, ...

Version	Datum	Author(s)	Description
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

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.

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.

2.1 Security

2.1.1 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.

2.1.2 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.

2.2 Compatibility

2.2.1 JAP/InfoService/MixConfig

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

2.2.2 Mixproxy

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

2.3 Performance

2.3.1 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.

2.3.2 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.

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 Structure

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.

3.1.1 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.

3.1.2 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.*

3.1.3 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.

3.1.4 Test Cases – C++

The test cases are in the `test` directory.

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 Code Structure

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

3.3.1 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.

3.3.2 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.

3.3.3 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

3.4 Classes- / Interface Structure

3.4.1 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.

3.4.2 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

3.5 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.

4 Libraries

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.

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!**

4.2.1 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.

4.2.2 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.

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.

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.

4.5 Other Resources (JAP/InfoService/MixConfig)

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

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.

4.6.1 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.

4.6.2 Mixproxy

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

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.

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++
```

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_JapClientWindowEvenHandlerTemp`). 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";`

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.

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*.

5.4.1 Java

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

5.4.2 C++

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

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 i**n 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.

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.

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.

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.

6 Type Conventions

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 > 10^{19}
float	4	$-10^{38}..10^{38}$
double	8	$-10^{308}..10^{308}$

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

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).

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`.

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`.

6.5 Classes / Interfaces

6.5.1 Static Code

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

6.5.2 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.

6.5.3 "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.

6.5.4 Templates (C++)

Templates are not allowed for compatibility reasons.

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!

6.6.1 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.

6.6.2 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.

6.7 Exceptions

6.7.1 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.

6.7.2 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).

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.

7 Documentation

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].

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).

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.

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`

7.5 Return Values (@return Tag)

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

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().*

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.

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 */`

8 Other Conventions

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.

8.1.1 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.

8.1.2 Tab Size

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

8.1.3 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());
```

8.1.4 Methodenparameter

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

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.
- I`f(...)`-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)`

8.2.1 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.

8.2.2 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.

9 Testing

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.

9.2 JUnit

9.2.1 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.

9.2.2 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.

9.2.3 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.

9.2.4 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.AllTests.suite())` 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.

9.2.5 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;
```

9.3 CppUnit

9.3.1 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>
```

9.3.2 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.

9.3.3 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);
```

9.3.4 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.

9.3.5 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.

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

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/

11 Appendix (Summary)

P

- Sicherheit
- Kompatibilität
- Performance
- Qualität

S

Zum genauen Aufbau von Klassen und Methoden siehe Kapitel 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 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 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 9