

EU Project



Deliverable 1.1.1

Specification of Software Development Environment

*Rainer Worst, Ralph Breithaupt, Claus Hoffmann, Hartmut Surmann,
Keyan Zahedi*

Number: MACS/1/1.1

WP: 1.1

Status: Version 2

Created at: November 5, 2004

Revised at: February 8, 2005

FhG/AIS

Fraunhofer Institut für

Autonome Intelligente Systeme, Sankt Augustin, D

JR_DIB

Joanneum Research Graz, A

LiU-IDA

Linköpings Universitet, Linköping, S

METU-KOVAN

Middle East Technical University, Ankara, T

OFAI

Österreichische Studiengesellschaft für Kybernetik,
Vienna, A

This research was partly funded by the European Commission's 6th Framework Programme IST Project MACS under contract/grant number FP6-004381. The Commission's support is gratefully acknowledged.

© FhG/AIS 2004

Author addresses:

Rainer Worst
Fraunhofer Institut für
Autonome Intelligente Systeme
Schloß Birlinghoven
D-53754 Sankt Augustin, Germany



Fraunhofer Institut für
Autonome Intelligente Systeme
Schloß Birlinghoven
D-53754 Sankt Augustin
Germany

Tel.: +49 (0) 2241 14-2683
(Co-ordinator)

Contact:
Dr.-Ing. Erich Rome



Joanneum Research
Institute of Digital Image Processing
Computational Perception (CAPE)
Steyrergasse 9
A-8010 Graz
Austria

Tel.: +43 (0) 316 876-1769

Contact:
Dr. Lucas Paletta



Linköpings Universitet
Dept. of Computer and Info. Science
Linköping 581 83
Sweden

Tel.: +46 13 24 26 28

Contact:
Prof. Dr. Patrick Doherty



Middle East Technical University
Dept. of Computer Engineering
Inonu Bulvari
TR-06531 Ankara
Turkey

Tel.: +90 312 210 5539

Contact:
Prof. Dr. Erol Şahin



Österreichische Studiengesellschaft
für Kybernetik (ÖSGK)
Freyung 6
A-1010 Vienna
Austria

Tel.: +43 1 5336112 0

Contact:
Prof. Dr. Georg Dorffner

Contents

1	Introduction	1
2	Operating System	1
2.1	Version	1
2.2	Naming Conventions	2
3	Programming Languages	3
3.1	C/C++	3
3.2	Java	3
4	Coding Style	3
4.1	Usage	3
4.2	General Recommendations	4
4.3	Naming Conventions	4
4.4	Files	10
4.5	Statements	12
4.6	Layout and Comments	19
5	Tools	25
5.1	Eclipse	25
5.2	CVS	25
5.3	Doxygen	25
6	Middleware (CORBA)	26
7	Summary and Future Action	26

1 Introduction

An important part of the work to be achieved in MACS will be the implementation of the affordance-based architecture on real mobile robots and within simulation. This software will be developed in a distributed manner by all the partners. We have the intention, to set up a reference control system, where all the contributions are being integrated and are always ready for demonstration. The partners have agreed on a common software development environment (SDE), to make the integration task most efficient.

The SDE, which is used for MACS, consists of proven tools, which are available as source code, and which do not imply any commercial licence. An overview is given here.

Topic	Software	Version
Operating System	SuSE Linux	9.0 Professional Edition
C/C++ Language	GCC	3.3.1
Java Language	J2SE	1.4.2 till June, 2005 1.5 (aka 5.0) henceforth
Integrated Development Environment	Eclipse	3.0.1 till June, 2005 3.1 henceforth
Version Control	CVS	1.11.6
Documentation	Doxygen	1.3.9
Middleware (CORBA)	TAO	1.4

The different topics are explained in more detail in the rest of this document. The central section contains the description of a coding style that shall be applied to the software written in MACS. The report concludes with some remarks on further action in the related task of WP1.

2 Operating System

Though it is possible to be independent of a specific operating system by using middleware such like CORBA, it makes sense to agree on a specific operating system, which is used by default within the MACS project. As we know from experience, the integration of different modules is always easier, if they have been developed for the same target system.

2.1 Version

During the project kick-off meeting, all the partners agreed to use Linux as the operating system for the software developed in MACS. To be more precise, we propose a defined variant of Linux for the duration of the whole project. To avoid unnecessary portings of programs, this variant must not be changed, unless there is a decision of the Managerial Board, depending on one of these criteria:

1. Support of necessary hardware
2. Compatibility with necessary libs

Regarding the KURT2 robots that will be used as demonstrators we have to pay attention to the following constraints.

Laser scanner: The laser scanner is connected to the laptop either via an usb device or via a pcmcia card, which convert the data from the laser scanner's rs422 interface. Both variants are directly supported under the 2.4 and 2.6 kernels of Linux.

Serial servo device: The servo is connected with the servo controller and this is connected via rs232 to the laptop. The RS232 device is supported under 2.4 and 2.6 kernels.

Cameras (quickcam pro): The cameras are connected via USB to the laptop, and the camera servos with the servo controller, as described above. USB drivers for the Philips chips of the cameras are available for the 2.4 and 2.6 kernels, but the maintainer has finished his work because of some trouble inside the Linux community. So, further support of newer kernels is still open [6].

Robot: The robot is controlled via a PCMCIA CAN card (MicroControl or Softing). Both vendors only support drivers for the 2.4.18++ kernels. Drivers for the 2.6 kernel are currently not available.

WLAN: The use of WLAN cards and drivers is under development. The best supported card that we know is the Orinoco (aka Agere) card 802.11b (2.4 and 2.6 kernels).

As a result, currently we use SuSE 9.0 with the 2.4.21xxx kernels mainly limited by the PCMCIA CAN card. The Orinoco PCMCIA WLAN cards are directly supported. CAN drivers and camera drivers have to be compiled for that kernel.

As long as these constraints do not change, we will therefore use SuSE Linux 9.0 Professional Edition in MACS. This means in more detail:

Component	Version	Remark
Kernel	2.4.21	modified by SuSE
GCC	3.3.1	
glibc	2.3.2	

The recommended Linux distribution is available from a lot of FTP servers, e.g. [10]. A list of all mirror sites can be found at [11].

2.2 Naming Conventions

We shall use some general naming conventions for file names as described below.

File names are made up of a base name, and an optional period and suffix. The first character of the name should be a letter and all characters (except the period) should be letters and numbers. These rules apply to both program files and default files used and produced by the program.

Some compilers and tools require certain suffix conventions for names of files. The standard suffix rules of the make command are to be used, and it is recommended to name C++-files with the suffix ".cpp".

In addition, it is conventional to use "Makefile" (not "makefile") for the control file for make (for systems that support it) and "README" for a summary of the contents of the directory or directory tree.

3 Programming Languages

3.1 C/C++

GCC 3.3.1 and glibc 2.3.2 are the reference versions for C/C++ programming. Naming conventions and coding standards to be used in MACS are defined in section 4.

3.2 Java

The Java code, which is developed in MACS, is based on Sun's Java 2 Platform Standard Edition: release J2SE 1.4.2 till June, 2005, and release J2SE 1.5 (aka 5.0) henceforth.

The basis for naming conventions and coding standards are the Code Conventions for the Java Programming Language from Sun, see [9]. It is possible to adjust Eclipse according to these conventions.

We propose the following important add-ons to these Code Conventions, to be conform with the recommendations in section 4:

1. All code blocks are put into cambered parenthesis. This regards also code blocks that have only one statement.
2. It is not allowed to use tabulators for indentation. This feature can be adjusted in eclipse, too.

4 Coding Style

4.1 Usage

The content of this section is mainly taken from the GeoSoft website [2]. Some rules are added or modified. It lists C++/Java coding recommendations common in the C++ and Java development community.

The recommendations are based on established standards collected from a number of sources, individual experience, local requirements and needs, as well as suggestions given in [5], [4], [1], and [3].

While a given development environment (IDE) can improve the readability of code by access visibility, color coding, automatic formatting and so on, the programmer should never rely on such features. Source code should always be considered larger than the IDE it is developed within and should be written in a way that maximize its readability independent of any IDE.

The style includes statements specially for C++, but is applicable for C and Java as well.

4.1.1 Layout of the Recommendations

The recommendations are grouped by topic and each recommendation is numbered to make it easier to refer to during reviews. Layout of the recommendations is as follows:

- Guideline, short description
- Example if applicable
- Motivation, background, and additional information

The motivation section is important. Coding standards and guidelines tend to start "religious wars", and it is important to state the background for the recommendation.

4.1.2 Recommendation Importance

Within these recommendations the terms *must*, *should* and *can* have special meaning. A must requirement must be followed, a should is a strong recommendation, and a can is a general guideline.

4.2 General Recommendations

(1) Any violation to the guide is allowed if it enhances readability.

The main goal of the recommendation is to improve readability and thereby the understanding and the maintainability and general quality of the code. It is impossible to cover all the specific cases in a general guide and the programmer should be flexible.

(2) The rules can be violated if there are strong personal objections against them.

The attempt is to make a guideline, not to force a particular coding style onto individuals. Experienced programmers normally want adopt a style like this anyway, but having one, and at least requiring everyone to get familiar with it, usually makes people start thinking about programming style and evaluate their own habits in this area.

On the other hand, new and inexperienced programmers normally use a style guide as a convenience of getting into the programming jargon more easily.

4.3 Naming Conventions

4.3.1 General Naming Conventions

(3) Names representing types must be in mixed case starting with upper case.

`Line, SavingsAccount`

Common practice in the C++ development community.

(4) Variable names must be in mixed case starting with lower case.

`line, savingsAccount`

Common practice in the C++ development community. Makes variables easy to distinguish from types, and effectively resolves potential naming collision as in the declaration `Line line;`

(5) Named constants (including enumeration values) must be all uppercase using underscore to separate words.

`MAX_ITERATIONS, COLOR_RED, PI`

Common practice in the C++ development community. In general, the use of such constants should be minimized. In many cases implementing the value as a method is a better choice:

```
int getMaxIterations() {    // NOT: MAX_ITERATIONS = 25
    return 25;
}
```

This form is both easier to read, and it ensures a unified interface towards class values.

(6) Names representing methods or functions must be verbs and written in mixed case starting with lower case.

```
getName(), computeTotalWidth()
```

Common practice in the C++ development community. This is identical to variable names, but functions in C++ are already distinguishable from variables by their specific form.

(7) Names representing namespaces should be all lowercase.

```
analyzer, iomanager, mainwindow
```

Common practice in the C++ development community.

(8) Names representing template types should be a single uppercase letter.

```
template<class T> ... template<class C, class D> ...
```

Common practice in the C++ development community. This makes template names stand out relative to all other names used.

(9) Abbreviations and acronyms must not be uppercase when used as name.
[3]

```
exportHtmlSource();    // NOT: exportHTMLSource();
openDvdPlayer();      // NOT: openDVDPlayer();
```

Using all uppercase for the base name will give conflicts with the naming conventions given above. A variable of this type would have to be named dVD, hTML etc. which obviously is not very readable. Another problem is illustrated in the examples above; When the name is connected to another, the readability is seriously reduced; the word following the abbreviation does not stand out as it should.

(10) Global variables should always be referred to using the :: operator.

```
::mainwindow.open(), ::applicationContext.getName()
Class::GLOBAL_VAR
```

In general, the use of global variables should be avoided. Consider using singleton objects instead.

(11) Private class variables should have underscore suffix.

```
class SomeClass {
    private:
        int length_;
}
```

Apart from its name and its type, the scope of a variable is its most important feature. Indicating class scope by using underscore makes it easy to distinguish class variables from local scratch variables. This is important because class variables are considered to have higher significance than method variables, and should be treated with special care by the programmer.

A side effect of the underscore naming convention is that it nicely resolves the problem of finding reasonable variable names for setter methods and constructors:

```
void setDepth (int depth) {
    depth_ = depth;
}
```

An issue is whether the underscore should be added as a prefix or as a suffix. Both practices are commonly used, but the latter is recommended because it seems to best preserve the readability of the name.

It should be noted that scope identification in variables has been a controversial issue for quite some time. It seems, though, that this practice now is gaining acceptance and that it is becoming more and more common as a convention in the professional development community.

(12) Generic variables should have the same name as their type.

```
void setTopic (Topic *topic)
// NOT: void setTopic (Topic *value)
// NOT: void setTopic (Topic *aTopic)
// NOT: void setTopic (Topic *x)

void connect (Database *database)
// NOT: void connect (Database *db)
// NOT: void connect (Database *oracleDB)
```

Reduce complexity by reducing the number of terms and names used. Also makes it easy to deduce the type given a variable name only.

If for some reason this convention doesn't seem to fit it is a strong indication that the type name is badly chosen.

Non-generic variables have a role. These variables can often be named by combining role and type:

```
Point startingPoint, centerPoint; Name loginName;
```

(13) All names should be written in English.

```
fileName;    // NOT:  filNavn
```

English is the preferred language for international development.

(14) Variables with a large scope should have long names, variables with a small scope can have short names. [5]

Scratch variables used for temporary storage or indices are best kept short. A programmer reading such variables should be able to assume that its value is not used outside a few lines of code. Common scratch variables for integers are i, j, k, m, n and for characters c and d.

(15) The name of the object is implicit, and should be avoided in a method name.

```
line.getLength();    // NOT:  line.getLineLength();
```

The latter seems natural in the class declaration, but proves superfluous in use, as shown in the example.

4.3.2 Specific Naming Conventions**(16) The terms get/set must be used where an attribute is accessed directly.**

```
employee.getName();    matrix.getElement (2, 4);
employee.setName (name); matrix.setElement (2, 4, value);
```

Common practice in the C++ development community. In Java this convention has become more or less standard.

(17) The term compute can be used in methods where something is computed.

```
valueSet->computeAverage();  matrix->computeInverse()
```

Give the reader the immediate clue that this is a potential time consuming operation, and if used repeatedly, he might consider caching the result. Consistent use of the term enhances readability.

(18) The term find can be used in methods where something is looked up.

```
vertex.findNearestVertex();  matrix.findMinElement();
```

Give the reader the immediate clue that this is a simple look up method with a minimum of computations involved. Consistent use of the term enhances readability.

(19) The term initialize can be used where an object or a concept is established.

```
printer.initializeFontSet();
```

The American initialize should be preferred over the English initialise. Abbreviation init should be avoided.

(20) Variables representing GUI components should be suffixed by the component type name.

```
mainWindow, propertiesDialog, widthScale, loginText,
leftScrollbar, mainForm, fileMenu, minLabel, exitButton, yesToggle
etc.
```

Enhances readability since the name gives the user an immediate clue of the type of the variable and thereby the objects resources.

(21) The suffix List can be used on names representing a list of objects.

```
vertex (one vertex),    vertexList (a list of vertices)
```

Enhances readability since the name gives the user an immediate clue of the type of the variable and the operations that can be performed on the object.

Simply using the plural form of the base class name for a list (`matrixElement` (one matrix element), `matrixElements` (list of matrix elements)) should be avoided since the two only differ in a single character and are thereby difficult to distinguish.

A list in this context is the compound data type that can be traversed backwards, forwards, etc. (typically an STL vector). A plain array is simpler. The suffix `Array` can be used to denote an array of objects.

(22) The prefix n should be used for variables representing a number of objects.

```
nPoints, nLines
```

The notation is taken from mathematics where it is an established convention for indicating a number of objects.

(23) The suffix No should be used for variables representing an entity number.

```
tableNo, employeeNo
```

The notation is taken from mathematics where it is an established convention for indicating an entity number.

An elegant alternative is to prefix such variables with an `i`: `iTable`, `iEmployee`. This effectively makes them named iterators.

(24) Iterator variables should be called i, j, k etc.

```
for (int i = 0; i < nTables); i++) {
    :
}

vector<MyClass>::iterator i; for (i = list.begin(); i !=
list.end(); i++) {
    Element element = *i;
    ...
}
```

The notation is taken from mathematics where it is an established convention for indicating iterators.

(25) The prefix is should be used for boolean variables and methods.

`isSet, isVisible, isFinished, isFound, isOpen`

Common practice in the C++ development community and partially enforced in Java.

Using the `is` prefix solves a common problem of choosing bad boolean names like `status` or `flag`. `isStatus` or `isFlag` simply doesn't fit, and the programmer is forced to choose more meaningful names.

There are a few alternatives to the `is` prefix that fits better in some situations. These are the `has`, `can` and `should` prefixes:

`bool hasLicense(); bool canEvaluate(); bool shouldSort();`

(26) Complement names must be used for complement operations. [5]

`get/set, add/remove, create/destroy, start/stop, insert/delete, increment/decrement, old/new, begin/end, first/last, up/down, min/max, next/previous, old/new, open/close, show/hide, suspend/resume, etc.`

Reduce complexity by symmetry.

(27) Abbreviations in names should be avoided.

`computeAverage(); // NOT: compAvg();`

There are two types of words to consider. First are the common words listed in a language dictionary. These must never be abbreviated. Write:

`command` instead of `cmd` `copy` instead of `cp` `point`
 instead of `pt` `compute` instead of `comp` `initialize` instead of
`init` etc.

Then there are domain specific phrases that are more naturally known through their abbreviations/acronym. These phrases should be kept abbreviated. Write:

`html` instead of `HypertextMarkupLanguage` `cpu` instead of
`CentralProcessingUnit` `pe` instead of `PriceEarningRatio` etc.

(28) Naming pointers specifically should be avoided.

`Line *line; // NOT: Line *pLine; or Line *linePtr; etc.`

Many variables in a C/C++ environment are pointers, so a convention like this is almost impossible to follow. Also objects in C++ are often oblique types where the specific implementation should be ignored by the programmer. Only when the actual type of an object is of special significance, the name should emphasize the type.

(29) Negated boolean variable names must be avoided.

```
bool isError;    // NOT:  isNoError bool isFound;    // NOT:
isNotFound
```

The problem arises when such a name is used in conjunction with the logical negation operator as this results in a double negative. It is not immediately apparent what `isNotFound` means.

(30) Enumeration constants can be prefixed by a common type name.

```
enum Color {
    COLOR_RED,
    COLOR_GREEN,
    COLOR_BLUE
};
```

This gives additional information of where the declaration can be found, which constants belongs together, and what concept the constants represent.

An alternative approach is to always refer to the constants through their common type: `Color::RED`, `Airline::AIR_FRANCE` etc.

(31) Exception classes should be suffixed with Exception.

```
class AccessException {
    :
}
```

Exception classes are really not part of the main design of the program, and naming them like this makes them stand out relative to the other classes.

(32) Functions, i.e. methods returning something, should be named after what they return and procedures, i.e. void methods, after what they do.

Increase readability. Makes it clear what the unit should do and especially all the things it is not supposed to do. This again makes it easier to keep the code clean of side effects.

4.4 Files**4.4.1 Source Files****(33) C++ header files should have the extension .h. Source files can have the extension .cpp (recommended), .C, .cc or .c++.**

```
MyClass.cpp MyClass.h
```

These are all accepted C++ standards for file extension.

(34) A class should be declared in a header file and defined in a source file where the name of the files match the name of the class.

MyClass.h, MyClass.cpp

Makes it easy to find the associated files of a given class. This convention is enforced in Java and has become very successful as such.

(35) All definitions should reside in source files.

```
class MyClass {
public:
    int getValue () {return value_;} // NO!
    ...
private:
    int value_;
}
```

The header files should declare an interface, the source file should implement it. When looking for an implementation, the programmer should always know that it is found in the source file. The obvious exception to this rule is of course inline functions that must be defined in the header file.

(36) File content must be kept within 80 columns.

80 columns is a common dimension for editors, terminal emulators, printers and debuggers, and files that are shared between several people should keep within these constraints. It improves readability when unintentional line breaks are avoided when passing a file between programmers.

(37) Special characters like TAB and page break must be avoided.

These characters are bound to cause problem for editors, printers, terminal emulators or debuggers when used in a multi-programmer, multi-platform environment.

(38) Indenting after of new lines should be 2 white spaces.

This is common within the C++ development community.

(39) The incompleteness of split lines must be made obvious. [5]

```
totalSum = a + b + c +
          d + e;
function (param1, param2,
          param3);
setText ("Long line split"
         "into two parts.");
for (tableNo = 0; tableNo < nTables;
     tableNo += tableStep)
```

Split lines occurs when a statement exceed the 80 column limit given above. It is difficult to give rigid rules for how lines should be split, but the examples above should give a general hint.

In general: Break after a comma. Break after an operator. Align the new line with the beginning of the expression on the previous line.

4.4.2 Include Files and Include Statements

(40) Header files must include a construction that prevents multiple inclusion. The convention is an all uppercase construction of the module name, the file name and the h suffix.

```
#ifndef MOD_FILENAME_H #define MOD_FILENAME_H
:
#endif // MOD_FILENAME_H
```

The construction is to avoid compilation errors. The name convention is common practice. The construction should appear in the top of the file (before the file header) so file parsing is aborted immediately and compilation time is reduced.

(41) Include statements should be sorted and grouped. Sorted by their hierarchical position in the system with low level files included first. Leave an empty line between groups of include statements.

```
#include <fstream> #include <iomanip>

#include <Xm/Xm.h> #include <Xm/ToggleB.h>

#include "ui/PropertiesDialog.h" #include "ui/MainWindow.h"
```

In addition to show the reader the individual include files, it also give an immediate clue about the modules that are involved.

Include file paths must never be absolute. Compiler directives should instead be used to indicate root directories for includes.

(42) Include statements must be located at the top of a file only.

Common practice. Avoid unwanted compilation side effects by "hidden" include statements deep into a source file.

4.5 Statements

4.5.1 Types

(43) Types that are local to one file only can be declared inside that file.

Enforces information hiding.

(44) The parts of a class must be sorted public, protected and private. All sections must be identified explicitly. Not applicable sections should be left out. [4], [1]

The ordering is "most public first" so people who only wish to use the class can stop reading when they reach the protected/private sections.

(45) Type conversions must always be done explicitly. Never rely on implicit type conversion.

```
floatValue = static_cast<float> (intValue);    // YES!
floatValue = intValue;                        // NO!
```

By this, the programmer indicates that he is aware of the different types involved and that the mix is intentional.

4.5.2 Variables

(46) Variables should be initialized where they are declared.

This ensures that variables are valid at any time. Sometimes it is impossible to initialize a variable to a valid value where it is declared:

```
int x, y, z; getCenter (&x, &y, &z);
```

In these cases it should be left uninitialized rather than initialized to some phony value.

(47) Variables must never have dual meaning.

Enhance readability by ensuring all concepts are represented uniquely. Reduce chance of error by side effects.

(48) Use of global variables should be minimized.

In C++ there is no reason global variables need to be used at all. The same is true for global functions or file scope (static) variables.

(49) Class variables should never be declared public.

The concept of C++ information hiding and encapsulation is violated by public variables. Use private variables and access functions instead. One exception to this rule is when the class is essentially a data structure, with no behavior (equivalent to a C struct). In this case it is appropriate to make the class' instance variables public [4].

Note that structs are kept in C++ for compatibility with C only, and avoiding them increases the readability of the code by reducing the number of constructs used. Use a class instead.

(50) Related variables of the same type can be declared in a common statement. [1]

Unrelated variables should not be declared in the same statement.

```
float x, y, z; float revenueJanuary, revenueFebruary,
revenueMarch;
```

The common requirement of having declarations on separate lines is not useful in the situations like the ones above. It enhances readability to group variables like this.

(51) C++ pointers and references should have their reference symbol next to the variable name rather than to the type name. [1]

```
float *x;    // NOT: float* x; int  &y;    // NOT: int&  y;
```

It is debatable whether a pointer is a variable of a pointer type (`float* x`) or a pointer to a given type (`float *x`). Important in the recommendation given though is the fact that it is impossible to declare more than one pointer in a given statement using the first approach. I.e. `float* x, y, z;` is equivalent with `float *x; float y; float z;` The same goes for references.

(52) The `const` keyword should be listed before the type name.

```
void f1 (const Widget *v)    // NOT: void f1 (Widget const *v)
```

Neither is better nor worse, but since the former is more commonly used that should be the convention. It also highly depends on what is intended:

```
const Widget *v; //non-const pointer, const data
Widget const *v; //const pointer, non-const data
```

In most cases the first version is wanted, and should be used.

(53) Implicit test for 0 should not be used other than for boolean variables and pointers.

```
if (nLines != 0)    // NOT:  if (nLines) if (value != 0.0)    //
NOT:  if (value)
```

It is not necessarily defined by the compiler that ints and floats 0 are implemented as binary 0. Also, by using explicit test the statement give immediate clue of the type being tested. It is common also to suggest that pointers shouldn't test implicit for 0 either, i.e. `if (line == 0)` instead of `if (line)`. The latter is regarded as such a common practice in C/C++ however that it can be used.

(54) Variables should be declared in the smallest scope possible. Keeping the operations on a variable within a small scope, it is easier to control the effects and side effects of the variable.

(55) Braces should stand alone.

```
if (...) {
    :
}

for(...) {
    :
}
```

Increases the readability of the code. This emphasizes the distinction between conditionals or loops and the enclosing block. In special cases, such as empty statements opening and closing braces can be written in the same line `{}`. Even if only one line follows a conditional or a loop declaration, braces should be used. This reduces the possibility of bugs when debug output or additional statements are added.

```
if (...) {           // NOT if (...)
    onlyStatement;  //      onlyStatement;
}

for(...) {          // NOT for(...)
    onlyStatement;  //      onlyStatement;
}
```

4.5.3 Loops

(56) Only loop control statements must be included in the `for()` construction.

```
sum = 0; for (i = 0; i < 100; i++) {
    sum += value[i];
}
// NOT: for (i = 0, sum = 0; i < 100; i++)
//      {
//          sum += value[i];
//      }
```

Increase maintainability and readability. Make it crystal clear what controls the loop and what the loop contains.

(57) Loop variables should be initialized immediately before the loop.

```
doSo = true;        // NOT:  bool isDone = false;
while (doSo) {     //      :
    :              //      while (!isDone) {
}                  //      :
                  //      }
```

(58) Loop variables should indicate what is done in the loop.

```
doAnalysis = true; // NOT:  bool running = true;
while (doAnalysis) { //      while (running) {
    :                //      :
}                    //      }
```

Enhances the readability of the code. Good choices of the variables name helps to indicate what is done in the following.

(59) do-while loops can be avoided.

do-while loops are less readable than ordinary while loops and for loops since the conditional is at the bottom of the loop. The reader must scan the entire loop in order to understand the scope of the loop.

In addition, do-while loops are not needed. Any do-while loop can easily be rewritten into a while loop or a for loop. Reducing the number of constructs used enhance readability.

(60) The use of break and continue in loops should be avoided.

These constructs can be compared to goto and they should only be used if they prove to have higher readability than their structured counterpart.

(61) The form while(true) should be used for infinite loops.

```
while (true) {
  :
}

for (;;) { // NO!
  :
}

while (1) { // NO!
  :
}
```

Testing against 1 is neither necessary nor meaningful. The form for (;;) is not very readable, and it is not apparent that this actually is an infinite loop.

4.5.4 Conditionals**(62) Complex conditional expressions must be avoided. Introduce temporary boolean variables instead. [5]**

```
if ((elementNo < 0) || (elementNo > maxElement) ||
    elementNo == lastElement)
{
  :
}
```

should be replaced by:

```
isFinished      = (elementNo < 0) || (elementNo > maxElement);
isRepeatedEntry = elementNo == lastElement; if (isFinished ||
isRepeatedEntry) {
  :
}
```

By assigning boolean variables to expressions, the program gets automatic documentation. The construction will be easier to read and to debug.

(63) The nominal case should be put in the if-part and the exception in the else-part of an if statement. [5]

```
isError = readFile (fileName);
if (isError) {
    :
}
else {
    :
}
```

Makes sure that the exceptions don't obscure the normal path of execution. This is important for both the readability and performance.

(64) The conditional should be put on a separate line.

```
if (isDone) {
    doCleanup();
}
// NOT:  if (isDone) doCleanup();
// NOT:  if (isDone)
//          doCleanup();
```

This is for debugging purposes. When writing on a single line, it is not apparent whether the test is really true or not.

(65) Executable statements in conditionals must be avoided.

```
// Bad!
if (!(fileHandle = open (fileName, "w"))) {
    :
}

// Better!
fileHandle = open (fileName, "w");
if (!fileHandle) {
    :
}
```

Conditionals with executable statements are just very difficult to read. This is especially true for programmers new to C/C++.

(66) Conditionals should be positive, if possible.

```
// DO                DON'T
if (isSomething)     // if (!isDone)
while (doAnalysis)   // while (!isDone)
if (returnValue != 0.0) // if (!(returnValue == 0.0))
```

Enhances the readability as the „!“ is easily overseen.

4.5.5 Miscellaneous

(67) The use of magic numbers in the code should be avoided. Numbers other than 0 and 1 should be considered declared as named constants instead.

If the number does not have an obvious meaning by itself, the readability is enhanced by introducing a named constant instead. A different approach is to introduce a method from which the constant can be accessed.

(68) Floating point constants should always be written with decimal point and at least one decimal.

```
double total = 0.0;    // NOT: double total = 0;
double speed = 3.0e8; // NOT: double speed = 3e8;
```

```
double sum;
:
sum = (a + b) * 10.0;
```

This emphasizes the different nature of integer and floating point numbers even if their values might happen to be the same in a specific case.

Also, as in the last example above, it emphasize the type of the assigned variable (sum) at a point in the code where this might not be evident.

(69) Floating point constants should always be written with a digit before the decimal point.

```
double total = 0.5;    // NOT: double total = .5;
```

The number and expression system in C++ is borrowed from mathematics and one should adhere to mathematical conventions for syntax wherever possible. Also, 0.5 is a lot more readable than .5; There is no way it can be mixed with the integer 5.

(70) Functions must always have the return value explicitly listed.

```
int getValue()    // NOT: getValue() {
:
}
```

If not explicitly listed, C++ implies int return value for functions. A programmer must never rely on this feature, since this might be confusing for programmers not aware of this artifact.

(71) goto should not be used.

Goto statements violates the idea of structured code. Only in some very few cases (for instance breaking out of deeply nested structures) should goto be considered, and only if the alternative structured counterpart is proven to be less readable.

(72) "0" should be used instead of "NULL"

NULL is part of the standard C library, but is made obsolete in C++.

4.6 Layout and Comments

4.6.1 Layout

(73) Basic indentation should be 2.

```
for (i = 0; i < nElements; i++) {
    a[i] = 0;
}
```

Indentation of 1 is too small to emphasize the logical layout of the code. Indentation larger than 4 makes deeply nested code difficult to read and increases the chance that the lines must be split. Choosing between indentation of 2, 3 and 4, 2 and 4 are the more common, and 2 is chosen to reduce the chance of splitting code lines.

(74) Block layout should be as illustrated in example 1 below (recommended), and must not be as shown in example 2. [3]

```
while (do) {
    doSomething();
    done = moreToDo();
}
```

```
// NOT
while (do)
{
    doSomething();
    done = moreToDo();
}
```

Example 2 introduces an extra indentation level which doesn't emphasize the logical structure of the code as clearly as in example 1.

(75) The class declarations should have the following form:

```
class SomeClass : public BaseClass {
    public:
        ...
    protected:
        ...
    private:
        ...
}
```

This follows partly from the general block rule above.

(76) The function declarations should have the following form:

```
void someMethod() {
    ...
}
```

This follows from the general block rule above.

(77) The if-else class of statements should have the following form:

```
if (condition) {  
    statements;  
}
```

```
if (condition) {  
    statements;  
}  
else {  
    statements;  
}
```

```
if (condition) {  
    statements;  
}  
else if (condition) {  
    statements;  
}  
else {  
    statements;  
}
```

This follows partly from the general block rule above. However, it might be discussed if an else clause should be on the same line as the closing bracket of the previous if or else clause:

```
if (condition) {  
    statements;  
} else {  
    statements;  
}
```

This is equivalent to the Sun recommendation. The chosen approach is considered better in the way that each part of the if-else statement is written on separate lines of the file. This should make it easier to manipulate the statement, for instance when moving else clauses around.

(78) A for statement should have the following form:

```
for (initialization; condition; update) {  
    statements;  
}
```

This follows from the general block rule above.

(79) An empty for statement should have the following form:

```
for (initialization; condition; update)
    ;
```

or

```
for (initialization; condition; update) {}
```

This emphasize the fact that the for statement is empty and it makes it obvious for the reader that this is intentional. Empty loops should be avoided however.

(80) A while statement should have the following form:

```
while (condition) {
    statements;
}
```

This follows from the general block rule above.

(81) A do-while statement should have the following form:

```
do {
    statements;
} while (condition);
```

This follows from the general block rule above.

(82) A switch statement should have the following form:

```
switch (condition) {
    case ABC :
        statements;
        // Fallthrough

    case DEF :
        statements;
        break;

    case XYZ :
        statements;
        break;

    default :
        statements;
        break;
}
```

Note that each case keyword is indented relative to the switch statement as a whole. This makes the entire switch statement stand out. Note also the extra space before the `:` character. The explicit `Fallthrough` comment should be included whenever there is a case statement without a `break` statement. Leaving the `break` out is a common error, and it must be made clear that it is intentional when it is not there. The default case should be included, even if empty, so that it is visible, that the default case has been thought of.

(83) A try-catch statement should have the following form:

```
try {
    statements;
} catch (Exception &exception) {
    statements;
}
```

This follows partly from the general block rule above. The discussion about closing brackets for `if-else` statements apply to the `try-catch` statements.

(84) Single statement `if-else`, `for` or `while` statements should not be written without brackets.

```
if (condition) {
    statement;
}

while (condition) {
    statement;
}

for (initialization; condition; update) {
    statement;
}
```

It is a common recommendation (Sun Java recommendation included) that brackets should always be used in all these cases. However, brackets are in general a language construct that groups several statements.

(85) The function return type can be put in the left column immediately above the function name.

```
void MyClass::myMethod (void) {
    :
}
```

This makes it easier to spot function names within a file since one can assume that they all start in the first column.

4.6.2 White Space

(86) General Spacing Rules.

- Conventional operators should be surrounded by a space character.
- C++ reserved words should be followed by a white space.
- Commas should be followed by a white space.
- Colons should be surrounded by white space.
- Semicolons in for statements should be followed by a space character.

```
a = (b + c) * d;           // NOT:  a=(b+c)*d
while (true) {           // NOT:  while(true) ...
doSomething (a, b, c, d); // NOT:  doSomething (a,b,c,d);
case 100 :               // NOT:  case 100:
for (i = 0; i < 10; i++) { // NOT:  for (i=0;i<10;i++){
```

Makes the individual components of the statements stand out. Enhances readability. It is difficult to give a complete list of the suggested use of whitespace in C++ code. The examples above however should give a general idea of the intentions.

(87) Method names should be followed by a white space when it is followed by another name.

```
doSomething (currentFile); // NOT:  doSomething(currentFile);
```

Makes the individual names stand out. Enhances readability. When no name follows, the space can be omitted (`doSomething()`) since there is no doubt about the name in this case.

An alternative to this approach is to require a space after the opening parenthesis. Those that adhere to this standard usually also leave a space before the closing parentheses: `doSomething(currentFile);`. This do make the individual names stand out as is the intention, but the space before the closing parenthesis is rather artificial, and without this space the statement looks rather asymmetrical (`doSomething(currentFile);`).

(88) Logical units within a block should be separated by one blank line.

Enhance readability by introducing white space between logical units of a block.

(89) Methods should be separated by three blank lines.

By making the space larger than space within a method, the methods will stand out within the file.

(90) Variables in declarations should be left aligned.

```
AsciiFile *file; int      nPoints; float      x, y;
```

Enhance readability. The variables are easier to spot from the types by alignment.

(91) Use alignment wherever it enhances readability.

```
value = (potential      * oilDensity) / constant1 +
        (depth          * waterDensity) / constant2 +
        (zCoordinateValue * gasDensity) / constant3;
```

```
minPosition      = computeDistance (min,      x, y, z);
averagePosition = computeDistance (average, x, y, z);
```

There are a number of places in the code where white space can be included to enhance readability even if this violates common guidelines. Many of these cases have to do with code alignment. General guidelines on code alignment are difficult to give, but the examples above should give a general clue.

4.6.3 Comments

(92) Tricky code should not be commented but rewritten. [5]

In general, the use of comments should be minimized by making the code self-documenting by appropriate name choices and an explicit logical structure. Nevertheless comments should be as much as possible.

(93) All comments should be written in English. [4]

In an international environment English is the preferred language.

(94) Use // for all comments, including multi-line comments.

```
// Comment spanning
// more than one line.
```

Since multilevel C-commenting is not supported, using // comments ensure that it is always possible to comment out entire sections of a file using /* */ for debugging purposes etc.

There should be a space between the "//" and the actual comment, and comments should always start with an upper case letter and end with a period.

(95) Comments should be included relative to their position in the code . [5]

```
while (true) {           // NOT:   while (true) {
  // Do something        //         // Do something
  something();           //         something();
}                         //         }
```

This is to avoid that the comments break the logical structure of the program.

(96) Class and method header comments should follow the JavaDoc conventions.

Regarding standardized class and method documentation the Java development community is far more mature than the C++. This is of course because Java includes a tool for extracting such comments and produce high quality hypertext documentation from it.

There have never been a common convention for writing this kind of documentation in C++, so when choosing between inventing your own convention, and using an existing one, the latter option seem natural. Also, there are JavaDoc tools for C++ available, for instance Doxygen (cf. section 5.3).

5 Tools

5.1 Eclipse

It is recommended to use Eclipse as an IDE for the development of code written in Java or C/C++. The current version 3.0.1 of Eclipse does not support J2SE 5.0, but the version 3.1, which is expected to be released at June, 2005, will do.

For Java programming the settings should be adjusted to Sun's code conventions [9].

5.2 CVS

Although the software will be developed distributed at several locations, a central repository is maintained at Fraunhofer AIS, where the contributions of all partners are checked in. This repository is the base for the reference control system to be established in WP 1.1.3.

The repository is controlled by the well-known tool CVS, running on a server at Fraunhofer AIS that is accessible from outside.

The directory structure for each module should look like:

Category	Java	C/C++
source code	src/	src/
header files		include/
libraries	lib/	lib/
object code	obj/	obj/
binaries	bin/	bin/
text files	doc/	doc/
building	build	Makefile
to-do-list	ToDo	ToDo

Only programs that were successfully compiled and tested may be checked in to the repository. As a default rule, updated files should be checked in each evening after work.

5.3 Doxygen

All the source code shall be documented by the means of the tool Doxygen [12], which extracts the documentation directly from the sources (similar to JavaDoc, but not restricted to Java).

The contents and layout of the generated documentation can be configured by means of a configuration file. It is recommended to use the utility "Doxywizard" to create such files until standard configuration files have been established in MACS.

6 Middleware (CORBA)

We will use CORBA as the glue between the different software components that establish the MACS architecture. CORBA is the acronym for Common Object Request Broker Architecture, OMG's open, vendor-independent architecture and infrastructure that computer applications use to work together over networks. CORBA has been specified by the Object Management Group (OMG) [7].

Among the different implementations of CORBA that are available we have chosen TAO [8], which is an open-source CORBA-compliant Object Request Broker born of research at Washington University.

7 Summary and Future Action

This report contains the initial specification of the SDE that is going to be used in MACS. During the remaining duration of Task 1.1 this environment has to be implemented. Deliverable 1.1.3 will include a revised version of this document, where the experiences gathered during the implementation are reflected. That version will give more detailed recommendations for the usage of some tools, in particular regarding standard settings of Eclipse and Doxygen.

The next output of Task 1.1 will be deliverable 1.1.2 "Specification of Module Interfaces", where the use of CORBA as middleware in the MACS project and templates for interface specifications written in IDL will form the main part.

References

- [1] GABRYELSKI, K. <http://www.wildfire.com/~ag/Engineering/Development/C++Style/>.
- [2] GEOSOFT. <http://geosoft.no/development/cppstyle.html>.
- [3] HOFF, T. <http://www.possibility.com/Cpp/CppCodingStandard.htm>.
- [4] M. HENRICSON, E. N. <http://www.doc.ic.ac.uk/lab/cplus/c++.rules/>.
- [5] MCCONNELL, S. *Code Complete*, 2nd ed. Microsoft Press, 2004.
- [6] NEMOSOFT. <http://www.smcc.demon.nl/webcam/>.
- [7] OMG. <http://www.omg.org/gettingstarted/corbafaq.htm>.
- [8] SCHMIDT, D. C. <http://www.cs.wustl.edu/~schmidt/TAO.html>.
- [9] SUN. <http://java.sun.com/docs/codeconv/>.
- [10] SUSE. <ftp://ftp.gwdg.de/pub/linux/suse/ftp.suse.com/suse/i386/9.0/>.

[11] SUSE. http://www.suse.com/us/private/download/ftp/int_mirrors.html.

[12] VAN HEESCH, D. <http://www.stack.nl/~dimitri/doxygen/>.