Coding Rules
für JAVA und C/C++

Versions-Nr.:	1.00
Erstellungsdatum:	2003-02-07
Datei-Name:	CodingRules_1_00.doc
URL:	www.break-it-down.de
Copyright:	© Copyright 2003 Dipl-Informatiker (FH) Stefan Kuhn Augustusstraße 2c 86343 Königsbunn
Verfasser/Review	_____ (Verfasser: Stefan Kuhn) _____ (Review: _____)
Historie	V0.90 Neuerstellung V1.00 Ergänzung nach Review

Inhalt

1 Einleitung 3

1.1 Ausgangssituation 3
1.2 Wozu Coding Rules 3
1.3 Häufige Fehler beim Erstellen von Coding Rules 4
1.4 Verfahrenshinweis 4

2 Coding Rules für JAVA und C++ 5

2.1 Allgemeine Namensgebung 5
2.2 Spezielle Namensgebung 8
2.3 SourceCode Formatierung 11
2.4 Klassen 17
2.5 Wichtige Regeln 20
2.6 Weitere Regeln 24
2.7 C/C++ Coding Rules 28
2.8 C++ Classfile Template 32
2.8 C++ Classfile Template 33
2.9 Java Class Template 34

3 Kommentierung 35

3.1 Verzeichnisse 35
3.2 Copyright Hinweis 35
3.3 Dateikopf / Fileheader 35
3.4 Methoden 37
3.5 Methoden mit vielen Argumenten 38
3.6 Spezielle Keyworte 39

4 Anderes 40

4.1 Konfigurationsmanagement 40
4.2 Bug Tracking System 40
4.2 Bug Tracking System 41

1 Einleitung

1.1 Ausgangssituation

Es gibt heute glücklicherweise einige Quellen für Coding Rules. Dies belegt, dass solche Regeln mehr und mehr eingesetzt werden. Viele dieser Regelwerke entstanden in der OpenSource-Gemeinde. Da dort Projekte von verschiedensten und weltweit verteilten Personen entwickelt werden ist es gerade hier besonders wichtig, dass ein gemeinsamer Standard für den SourceCode beachtet wird.

Trotz der Verfügbarkeit dieser Coding Rules gibt es nur wenige wirklich vollständige Dokumente dazu. Oftmals wird nur ein spezielles Thema, wie z.B. die Einrückung mit Tabulator oder das Verbot für den ? Operator bearbeitet.

Dieses Dokument stellt hingegen eine umfangreiche Sammlung von Regeln dar. Falls wichtige Regeln fehlen, oder die Regeldefinitionen missverständlich sind, so sollten diese noch ergänzt bzw. besser erklärt werden.

Beachten Sie bitte: eine Definition von Coding Rules ist ein lebendes Dokument. Das bedeutet, dass einige der Aussagen mit der Zeit falsch oder unvollständig sein können. Ändern Sie diese und ergänzen Sie diese um Coding Rules für Ihre Firma oder Ihr Projekt zu erhalten.

1.2 Wozu Coding Rules

Großer Aufwand fällt auf die Pflege eines Softwareproduktes, dies sind im Schnitt 80% der entstandenen Kosten. Zudem wird kaum ein Softwareprodukt vom Anfang bis Ende (=Außerbetriebnahme) vom gleichen Entwickler betreut.
Regeln für den SourceCode sind deshalb wichtig um einerseits die Lesbarkeit des Codes zu gewährleisten und dadurch die Verständlichkeit zu verbessern und andererseits um häufig auftretende Fehler zu vermeiden.

Coding Rules sollen:

- häufige Fehler vermeiden helfen;
- schwerverständliche Sprachkonstrukte verhindern;
- die Lesbarkeit des Sources verbessern;
- die Einarbeitungszeit in vorhandenen SourceCode reduzieren und damit die Pflege vereinfachen;

1.3 Häufige Fehler beim Erstellen von Coding Rules

Zu kurz.
Es gibt nur Regeln für die Einrückung, Tabulatorenweite und Ähnliches. Dies ist nicht ausreichend, da nur die Lesbarkeit vereinheitlicht wird, die noch wichtigeren Teile jedoch fehlen.

Zu ausführlich.
Eine 100-seitige Vorgabe für Coding Rules wird vermutlich nicht von jedem Programmierer gerne gelesen und beachtet. Dies führt zu großem Aufwand bei der Einarbeitung und bringt Akzeptanzprobleme mit sich.

Zu streng.
Die oft zitierte künstlerische Freiheit ist vielen Programmieren und vor Allem den hoch qualifizierten Softwareentwicklern heilig. Eine zu weitgehende Einschränkung der Kreativität kann die Motivation beeinträchtigen und führt zu schlechterer Softwarequalität und längeren Entwicklungszyklen. Eine zu bürokratisch durchgeführte Qualitätssicherung kann hier die Qualität verringern, anstatt verbessern.

1.4 Verfahrenshinweis

Um optimale Coding Rules zu erhalten, sollten diese an die Bedürfnisse

- der Firma;
- des Projektes und der Kunden;
- der Entwickler und Programmierer;

angepasst und ggf. je nach Stand der Technik und der gesammelten Erfahrungen (z.B. aufgetretene Fehler) erweitert werden. Dieses Dokument soll als Vorlage dafür dienen, kann jedoch nicht allgemeingültig für alle Bedürfnisse sein und sollte deshalb angepasst werden.

2 Coding Rules für JAVA und C++

2.1 Allgemeine Namensgebung

2.1.1 Namen

Die verwendeten Namen für Variablen, Methoden, Klassen und Dateinamen sind ein wichtiger Teil des SourceCodes. Ein sprechender Name sagt mehr als viele Kommentare und erleichtert die Zuordnung, das Verständnis oder das Auffinden von Code enorm.

Namen resultieren aus einem langen Denkprozess. Werden allgemeine oder sinnlose Namen verwendet, so wurde meist zu wenig Nachgedacht. Eventuell hat der Autor des SourceCodes selbst die Bedeutung des Codes nicht verstanden! Wofür der Code ist und was er macht sollte aus einem Wort erkennbar sein.

Jedes Projekt wird eine Anzahl dieser Namen aufweisen. Diese müssen in einem Glossar dokumentiert werden und einheitlich angewendet werden. Eine widersprüchliche Interpretation oder Verwendung muss dadurch verhindert werden.

2.1.2 Klassennamen

Der Name lässt klar erkennen, was die Klasse ist. Ist dies nicht gegeben wurde meist kein gründliches Design durchgeführt. Der Namen sollte wirklich einen schnellen Einblick in die Funktionalität der Klasse geben.
Zusammengesetze Klassennamen mit mehr als 3 Wörtern lassen auf Designprobleme schließen. (Macht die Klasse zu viel? Kann diese aufgeteilt werden?)
Bei abgeleiteten Klassen sollte die Bezeichnung der Basisklasse NICHT in den Namen aufgenommen werden. Der Name der Klasse soll eigenständig bleiben.

- Großbuchstaben am Wortanfang und zur Wortunterscheidung;
- Rest der Wörter in Kleinbuchstaben;
- erstes Zeichen immer Groß (sollte auch Wortanfang sein!);
- kein Underbar „_“.

Beispiel:

```
class NameOneTwo

class Name
```

2.1.3 Methoden / Funktionsnamen

Normalerweise sollte eine Methode/Funktion eine Aktion ausführen. Diese Aktion muss aus dem Namen klar erkennbar sein. Die Namensgebung sollte analog zu den Klassennamen erfolgen, also sprechende Namen sind zu finden. Ob der erste Buchstabe Groß (C++) oder Klein (Java) geschrieben wird ist letztlich Geschmacksache, sollte jedoch einheitlich festgelegt sein. Dieses Dokument legt sich hier auf Kleinbuchstaben am Anfang von Methoden fest. Dies bietet den Vorteil, dass Klassennamen klar von Methodennamen unterschieden werden können.

- Argumentnamen beginnen immer mit Kleinbuchstaben.
- Statische Methoden (class global) beginnen mit einem Großbuchstaben.
- Methoden welche nur für debug verwendet werden sollen einen Prefix erhalten.

Beispiele:

```
checkForError();

dumpDataToFile();

int startYourEngines(Engine& someEngine, Engine& anotherEngine);

static void SomeStaticMethod(...);

DebugTestAllData();
```

2.1.4 Attributnamen / Klassenvariable

Die Attribute (Variablen der Klasse oder Klassenvariablen) werden mit dem Prefix (m) gekennzeichnet und folgen sonst den gleichen Regeln wie Klassennamen. Sind Klassenattribute im public Bereich der Klasse enthalten, so stellt dies im Allgemeinen eine Verletzung der Kapselregelung aus der objektorientierten Programmierung da. Dies muss explizit begründet werden – oder besser vermieden werden.

```
class CodingRulesChecker
{
   public:
      int      mSpecialCounter;  // this must be public because <reason>

   private:
      int      mVal;
      int      mCounter;
}
```

2.1.5 Variablenamen

Die Variablen innerhalb der Methoden/Funktionen verwenden nur Kleinbuchstaben und den Underbar „_“ zur Worttrennung. Dadurch sind die Klassenattribute klar von den lokalen (Stack) Variablen zu unterscheiden.

```
int NameOneTwo::HandleError(int errorNumber)
{
      int            error= OsErr();
      Time           time_of_error;
      ErrorProcessor error_processor;
}
```

Alternativ dazu können Variablennamen analog zu Methodennamen gebildet werden (erster Buchstabe klein, weitere Wörter mit Großbuchstaben)

```
int             i;
char            c;
float           myWidth;
```

2.2 Spezielle Namensgebung

2.2.1 Großbuchstaben

Vor Allem bei Abkürzungen werden häufig mehrere Grußbuchstaben hintereinander verwendet. Dies erschwert jedoch die Lesbarkeit des Codes.
Beispiele:
Statt `NetworkABCKey` sollte `NetworkAbcKey` verwendet werden.
Statt `UMLRessource` sollte `UmlRessource` verwendet werden.

2.2.2 Suffixes

Dem Namen nachgestellte Bezeichner können, wenn diese einheitlich definiert und verwendet werden, die Lesbarkeit des Codes verbessern.

`Max`	Der Maximalwert einer Variablen.
`Min`	Der Minimalwert einer Variablen
`Cnt`	Ein Counter (Zähler)
`Key`	Eine key value (Schlüsselvariable)
`Type`	Kennzeichnet durch typedef definierte Typen.

Mit diesen Suffixen erstellte Namen sind leichter verständlich:
Beispiele:

```
RetryMax
RetryCnt
UserIdKey
```

2.2.3 Globale Variablen / Defines

- Globalen Variablen wird ein „g“ vorangestellt.
- Der gesamte Name wird in Großbuchstaben geschrieben.
- Trennung der einzelnen Wörter mit Underbar „_“.
- #defines haben globale Funktionalität und folgen den Namensregeln für globale Variablen (alles Groß).

```
static final int MIN_WIDTH = 4;
static final int MAX_WIDTH = 999;
static final int GET_THE_CPU = 1;
#define MIN_HDD_SPACE = 10;
```

2.2.4 Parametername

Den Parameternamen wird ein „a“ oder „an“ vorgestellt. Der Name des Parameters solle den Typ klar wiederspiegeln, außer der Typ ist trivial.

```
public boolean numberIsEven(int aValue, Object anObject) {
  ....
}
```

2.2.5 Konstante

Klassen-Konstanten werden mit dem Prefix „k“ gekennzeichnet.

```
static final int kDaysInWeek = 7;
```

Konstante Zahlen im SourceCode sollten nicht verwendet werden – außer wenn die Bedeutung der Zahl klar ersichtlich ist. Stattdessen sollten defines oder Klassen-Konstante verwendet werden.

2.2.6 Packages

Die Namen der Packages werden in Kleinbuchstaben geschrieben.

```
#package mypackage
```

2.2.7 Prefixes

Analog wie bei Suffixes können einheitliche Prefixes (dem Namen vorgestellte Namenergänzungen) die Lesbarkeit verbessern. Auch hier ist eine einheitliche Verwendung wichtig.

Methoden / Klassen		
`I`	Kennzeichnet eine Interface-Klasse.	`IMyInterface`
`is`	*is* impliziert immer eine Frage, auf welche die Funktion die Antwort geben sollte. Der return-Wert ist immer vom Typ Boolean.	`boolean isReady();`
`get`	Get liefert einen Wert. Get liefert nie einen Boolean zurück, dort muss is verwendet werden.	`int getMaxCount();`
`set`	Set setzt einen Wert.	`void setDate(Date new_date);`
Variablen		
`m`	Kennzeichnet Attributnamen bzw. Membervariablen.	`int mCounter;`
`p`	Kennzeichnet Pointer (Zeiger).	`int* pData;`
`r`	Kennzeichnet Referenzen.	`&rData;`

`g`	Kennzeichnet globale Variablen. (auch global ausgeschrieben ist erlaubt).	`Logger* gpLog;`
		`Logger* globalpLog;`
		`// eine globale Pointervariable`

`s`	Kennzeichnet statische (static) Variablen.	`static StatusInfo msStatus;` `// a static Member variable.`
`k`	Kennzeichnet eine Konstante.	

Verbirgt sich hinter einer is oder get Methode eine Datenmanipulation oder Zustandsänderung der Klasse, so stellt dies ein grobes Fehlverhalten dar. Die Methode macht mehr als der Name angibt. Ist das Auslesen z.B. nur einmal möglich, da die Variable hernach gelöscht wird, so muss dies im Namen ersichtlich sein, z.B. getAndCleanValue().
Prefixes können auch verwendet werden um alle Sourceteile eines Moduls, wie z.B. einer Library, eindeutig benennen zu können.

2.3 SourceCode Formatierung

2.3.1 Zeilenlänge / Statements pro Zeile

Das Ausdrucken von Code ist immer noch sehr hilfreich und zu lange Zeilen am Monitor führen zu Unübersichtlichkeit. Deshalb sollte die Länge einer Zeile 80 Zeichen nicht überschreiten.

Eine Codezeile sollte nur ein Statement enthalten. Wenn zwei Statements eng zusammengehörig sind, so können diese ggf. in eine Zeile geschrieben werden.

Wenn ein Zeilenumbruch notwendig wird, so sollte der Sourcecode lesbar formatiert werden, indem bis zum letzen logischen Element eingerückt wird. Falls dadurch zu viele Zeilen entstehen, so sollte der Code normal eingerückt werden.

Die if Anweisung sollte so formatiert werden, dass das Statement gut erkennbar ist, dazu sollte nicht die oben beschriebene Formatierung angewandt werden, sondern normal mit 8 Leerzeichen eingerückt werden.

Ausnahmen von diesen Regeln sind möglich.

Beispiel:

```
someMethod(longExpression1, longExpression2, longExpression3,
           longExpression4, longExpression5);

var = someMethod1(longExpression1,
                  someMethod2(longExpression2,
                              longExpression3));

if ((condition1 && condition2) || (condition3 && condition4)
                               ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

int level;
int size;
```

und nicht: `int level, size;`

2.3.2 Methoden/Funktionen

Die Länge des Codes einer Funktion/Methode sollte eine Bildschirmseite nicht überschreiten.
Zu komplexe oder zu lange Methoden können oftmals sinnvoll aufgeteilt werden. Kleine Methoden mit sprechenden Namen vereinfachen den Entwicklungsprozess und helfen Fehler zu vermeiden.
Das Performance-Argument bei Funktionsaufrufen gilt für die heutigen Rechner meist nicht mehr. Harte embedded Systeme mögen hier noch Ausnahmen rechtfertigen.
Übrigens kann diese Regel mit gängigen SourceCode-Metriken automatisch verifiziert werden.

2.3.3 Einrückung / Tabulator

- Den Tabulator auf 8 Zeichen einstellen, jedoch für Codeeinrückung nicht verwenden. Die Tabulatoreinstellung erfolgt lokal im Editor und ist somit nicht mit der Sourcecodedatei verknüpft. Deshalb sind Leerzeichen besser, da diese überall gleich erscheinen.
- Stattdessen immer mit Spaces (Leerzeichen) einrücken. Dabei 4 Leerzeichen verwenden (auch 3 oder 8 werden teilweise verwendet). Auch hier gilt, die einmal in dieser Sourcecodedatei verwendete Einrückungsweite bei jeder Codeänderung beibehalten.
- Soweit wie notwendig einrücken – aber nicht weiter.
- Sind 5 Einrückungsstufen erfolgt, so ist oftmals der Code nicht gut modularisiert – die Funktion ist eventuell zu komplex. Diese Regel kann jedoch nicht strikt angewandt werden, oft sind mehr als 5 Stufen gerechtfertigt.

2.3.4 Leerzeichen

Ein Leerzeichen vor einem Schlüsselwort verwenden. Ein Leerzeichen zwischen den Expressions der for-Anweisung.

```
while (true) {
        ...
   }

   for (expr1; expr2; expr3) {
       ...
   }
```

Kein Leerzeichen zwischen Methodennamen und öffnender Klammer. Dadurch kann klar zwischen Schlüsselwort und Methode unterschieden werden.
Ein Leerzeichen nach dem Komma „,“ zwischen den Parametern.

```
   isEmpty();

   loadData(File filename, Table tableame) {
       ...
   }
```

Ein Leerzeichen vor und nach `+ - / *`. Aber kein Leerzeichen bei `++ --` und `*` bei Pointern.

```
   a += c + d;
   a = (a + b) / (c * d);

   while (d++ = s++) {
       n++;
   }
   printSize("size is " + foo + "\n");
```

2.3.5 Ausrichtung

Die Typenbezeichner und Namen sind gleichmäßig auszurichten. Die Pointerzeichen „*“ und „&“ sind am Typ, nicht am Namen zu platzieren.

```
DWORD       mDword
DWORD*      mpDword
mDword  = 0;
mpDword = NULL;
```

2.3.6 Klammerung mit { }

Einer der gebräuchlichen Stile sollte angewandt werden. Wenn eine Sourcecode-Datei bearbeitet wird, so sollte der dort angewandte Stil weiterverwandt werden. Hier hat sich der Zweitautor am Erstautor zu orientieren um eine Vermischung der Stile innerhalb einer Datei zu vermeiden.
Eine firmenweite Festlegung ist nicht zwingend erforderlich, da beide Stile gleich gut lesbar sind und man eventuell nur die Freiheitsgrade der Entwickler unnötig bescheidet. Es gilt jedoch: “Nur ein Stil in einer Datei“.

Möglichkeit 1
Beide Klammern auf gleicher Höhe wie die Schlüsselwörter.

```
if(condition)          while(Bedingung)
{                      {
  ....                   ...
}                      }
```

Möglichkeit 2 (von Unix)
Die erste Klammer in gleichen Zeile wie das Schlüsselwort, die schließende Klammer auf gleicher Höhe wie das Schlüssenwort.

```
if(condition) {         while(Bedingung) {
  ....                    ...
}                       }
```

2.3.7 Klammern mit (und)

Bei if/while wird ein Leerzeichen zwischen Schlüsselwort und Klammer geschrieben. Nach Funktionsnamen erfolgt kein Leerzeichen. Dadurch ist eine klare Unterscheidung zwischen Schlüsselwort und Funktion gegeben.

Beim return Statement werden Klammern nur geschrieben wenn diese notwendig sind, also nicht bei trivialen returns.

```
if (a==1) {          // one space

printf("hallo");     // no space

return true;         // no ( )
```

2.3.8 if – else

Die if Anweisung kann nach folgenden zwei Arten geschrieben werden. Die Klammern { } sollten immer angegeben werden.

1)

```
   if (condition)                  // Comment
   {
     ...
   }
   else if (condition)             // Comment
   {
     ...
   }
   else                            // Comment
   {
     ...
   }
```

2)

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

Das Format ohne Klammerung ist zu vermeiden:

```
if (condition)
      statement;  // NOT GOOD !
```

Mehrere logische Elemente sollten mit Klammern so gruppiert werden, dass diese klar und eindeutig gelesen werden können.

```
   if (a == b && c == d)      // AVOID!
   if ((a == b) && (c == d)) // RIGHT
```

2.3.9 try – catch

Analog zu den anderen Konstrukten erfolgt Schreibweise.

```
   try {
      statement;
   }
   catch (ExceptionClass e1) {
      statement;
   }
   catch (ExceptionClass e2) {
      statement;
   }
   finally {
      statement;
   }
```

2.4 Klassen

2.4.1 Methoden

Jede Klasse sollte die folgenden Methoden implementieren:

`Default Constructor`	Dies ermöglicht die Verwendung in Arrays.
`Virtual Destructor`	Aufgrund dem Garbage-Collector Mechanismus von Java kann hier oft auf den Destructor verzichtet werden. Bei C++ sollte man hier sauber allen Speicher freigeben. Hier entstehen die meisten schwer findbaren Speicherprobleme – „der Speicher geht langsam aus“.
`Copy Constructor`	
`Assigment Operator`	

2.4.2 Methodenreihenfolge

Die Reihenfolge der Methoden in einer Sourcecode Datei ist:

`public`	Das ist das Interface dieser Klasse und somit der wichtigste Teil. Der Datenzugriff erfolgt nur über get.. und set.. Methoden. Interne Methoden sind private und hier nicht enthalten.
`protected`	Je nach Anwendungsfall kann protected gewünscht sein. Oftmals reicht jedoch public und private völlig aus und ist zu bevorzugen.
`private`	Ist die Klasse fertiggestellt, so sind die inneren Abläufe nicht mehr wichtig (encapulation) – außer bei der Fehlersuche. Hinweis: Die ersten Anwendungen von C++ haben private zuoberst in der Datei dargestellt, dies ist jedoch nicht mehr Stand-der-Technik.

2.4.3 Constructor

Im Constructor dürfen keine weitreichenden Aktionen durchgeführt werden. Er darf nur Attributzuweisungen, Initialisierungen und nicht fehlschlagende Aktionen beinhalten. Z.B. ein openFile Kommando ist im Constructor zu vermeiden. Hinweis: Der Constructor kann keine Fehlermeldung zurückgeben, deshalb sollten dort auch keine Aktionen mit potentiellen Fehlermöglichkeiten durchgeführt werden. Exceptions im Constructor sollten vermieden werden.
Sind weitere Initialisierungen für die Klasse notwendig (Database connect, File open), so muss dafür eine eigene Methode implementiert werden (.open() oder .connect()).
Sind mehrere Konstruktoren vorhanden, so sollte eine Init() Methode die Initialisierung der Daten vornehmen. Diese Init() Methode wird dann von jedem Constructor aufgerufen. Dies vermeidet doppelten Code und verhindert das Vergessen von einzelnen Attributen – insbesondere bei nachträglichen Erweiterungen.

```
class Test
{
   public:
      Test() {
         Init();   // Call to common object initializer
      }

      Test(int val) {
          Init();   // Call to common object initializer
      }

   private:
      int      mVal;
      void Init() {
          mVal  = 0;
      }
   }
```

2.4.4 Law of Demeter

Zugriff auf die Memberattribute nach Möglichkeit über zugehörige Methoden, auch bei klasseninternem Zugriff.

```
class SunWorkstation
{
   void          UpVolume(int amount) { mSound.Up(amount); }
   SoundCard     mSound;
}

SunWorksation sun;
sun.UpVolume(1);   // richtig
sun.mSound.Up(1);  // falsch
```

2.4.5 Factory Klassen

Eine Component-Factory ist eine Klasse, welche nur static Methoden implementiert. Diese Methoden erzeugen andere Objekte (darum Factory) und sollten deshalb den Prefix „Make“ besitzen.

```
public class WidgetFactory
{
   static Button MakeButton(int aButtonType);
   static ListBox MakeListBox();
}
```

2.4.6 UnitTest / PackageTest

Jede Klasse sollte eine Testmethode enthalten, welche die Funktionalität der Klasse prüft. Ggf. sollte diese Methode auch die Funktionalität anderer Klassen überprüfen, wenn die Klasse Methoden der anderen Klassen verwendet.
Jedes Package sollte eine Testklasse enthalten, welche die gesamten Package-Klassen einzeln und ggf. verknüpft testet.

2.5 Wichtige Regeln

Diese Regeln sollen häufige Programmierfehler vermeiden helfen, indem besondere Codierungsregeln beachtet werden.

2.5.1 NULL / VOID Statements

Sollten Schleifen oder Ähnliches mit leeren Statements notwendig sein, so ist dies immer zu dokumentieren. Ansonsten stellt sich dem späteren Leser des Codes immer Frage, ob hier ein Code vergessen wurde.

```
while (*dest++ = *src++)
   ;         // VOID, do nothing here

for (u=1; u<10; u++)
   ;         // no statement, no further action required
```

2.5.2 Vergleich mit Konstante

Bei Vergleichen mit Konstanten sollte die Konstante links stehen:

```
if ( 6==errorNumber)
```

Dies erleichtert das Auffinden der Konstanten, da diese links steht und verhindert bei C/C++ das versehentliche vergessen des zweiten „=“. (Die Anweisung 6=errorNumber ist unzulässig.)

2.5.3 switch

Klammerungsformat:

```
switch (expression) {
 }
```

oder

```
switch (expression)
{
}
```

Wird ein break ausgelassen um beim nächsten Fall mit der Programmausführung fortzufahren, so ist dies mit einem Kommentar zu vermerken. Ansonsten gilt dies als vergessenes break und somit als Codierungsfehler.
Der default Fall sollte immer vorhanden sein, wenn er nicht erlaubt scheint so muss ein Fehlerhandling implementiert sein.

```
switch (expression) {
    case x:
        statement;
        break;
    case y:
        statement;
        // continue to default section, because ...
    default:
        statement;
        break;
 }
```

2.5.4 ++ / -- Operator

Diese Operatoren dürfen verwendet werden, jedoch ist darauf zu achten, dass die Übersichtlichkeit gewährleistet bleibt. Problematische Anordnungen, welche von der Ausführungsreihenfolge der ++/-- Operation abhängen sind zu vermeiden.

Vermeiden:

```
if (--xx == 0)
```
// wird erst erniedrigt und dann verglichen ?

Erlaubt:

```
if ( ... ) xx++;
```

Schlechtes Beispiel:

```
int i=0;
String text[] = { „eins“,“zwei“,“drei“ };
String str = „a test „+text[i++] + text[i++] + text[i++];
```

Wird C++ verwendet, so ist dieser Anweisung str="a test einseinseins" und i=3. Bei Java ist str=“a test einszweidrei“. Die Reihenfolge der ++ Operation ist also nicht klar erkennbar und ist deshalb hier verboten.

2.5.5 goto / continue / break

Das goto Statement sollte nicht verwendet werden. (siehe auch Exceptions).

Continue und Break sind getarnte gotos and sollten nicht verwendet werden. Die Sprungziele der break und continue sind oftmals nicht klar erkennbar und es können Probleme entstehen.

```
while (TRUE)
{
   ...
   // A lot of code
   ...
   if (/* some condition */) {
      continue;
   }
   ...
   // A lot of code
   ...
   if ( i++ > STOP_VALUE) break;
}
```

2.5.6 Doppelte Namen vermeiden

Eine doppelte Verwendung eines Namens (z.B. einer Variable) muss vermieden werden. Im Beispiel wird eine lokale Variable angelegt, welche den gleichen Namen trägt wie ein Klassenattribut. Dies führt zu sehr schwer verständlichem Code, Missverständnisse sind wahrscheinlich.

```
int count;
...
myMethod() {
    if (condition) {
        int count = 0;     // AVOID!
        ...
    }
    ...
}
```

2.5.7 Floating Point

Fließkommazahlen nur dort verwenden, wo diese auch notwendig sind. In Schleifen keine Fließkommazahlen einsetzen und Vergleiche nie als konkreten Zahlenvergleich == oder !=, sondern immer mit >= oder <= durchführen. Dies gilt vor Allem beim Einsatz in Schleifen oder bei Abbruchswertabfragen. Die Rundungsdifferenzen können zu schwer auffindbaren Fehlern führen, welche zudem Plattform und Compiler abhängig sind.

```
float value=0;       // in C++
double value=0;      //  in Java

while( value!=2.0)   // this is an endless loop !!
   value=value+0.1;
```

2.6 Weitere Regeln

Diese Regeln sollen insbesondere die Lesbarkeit und damit die Wartbarkeit des Codes verbessern. Ein guter Stil ist, nicht alle Feinheiten der Programmiersprache zu nutzen, sondern einen schnell und klar verständlichen Code zu implementieren.

2.6.1 Reentrant

Methoden/Funktionen sollten keine static Variablen enthalten, welche einen Mehrfachaufruf verhindern. Ausnahme ist hier, wenn dies explizit gewünscht ist, wie z.B. bei Interruptfunktionen häufig angewandt.
Die Verwendung von static Variablen ist zu kommentieren. Bei public Methoden, also dem Klasseninterface, sollte ein Hinweis in der Kommentierung der Methode vorhanden sein.

```
/**
 * Establish a connection to the database
 *
 * @return true if connection is made successfully
 * WARNING: you can call this function only once,
 *          all further calls are skipped.
 **/
```

:

2.6.2 Keine Tests mit Defaultwert

Die Abfrage nach true sollte immer explizit geschrieben werden.
`if (f() != true)  statt  if (f())`

Insbesondre wichtig ist dies bei C/C++ Programmen welche als Rückgabewerte 0 für gut und negative Werte für Fehler liefern. Hier also: `if (f() ==0) oder if (f() !=0)` verwenden.

Durch diese Schriebweise sollte auch das häufige Problem der umgekehrten Logik beim Stringvergleich strcmp() vermieden werden.

2.6.3 Eingebettete Zuweisungen

Zuweisungen innerhalb einer anderen Anweisung sind möglichst zu vermeiden. Wenn durch diese die Übersichtlichkeit des Codes verbessert wird, kann die Verwendung zulässig sein.

```
while ((c = getchar()) != EOF)  { ...
```

Unnötige interne Zuweisungen sind zu vermeiden:

```
d = (a = b + c) + r; // verboten

a = b + c;  // richtig
d = a + r;
```

2.6.4 Die Abkürzung (? :)

Da durch diese Schreibweise ein, für Ungeübte, schwer lesbarer Code entsteht, sollte darauf ganz versichtet werden. Wenn doch verwendet, dann sind die Anweisungen kurz zu halten um eine gute Übersichtlichkeit zu erreichen.

```
(condition) ? funct1() : func2();
```

2.6.5 Exceptions

Nach Auffassung vieler und Aufgrund des Verhaltens muss beachtet werden, dass eine Exception die gleichen Regeln wie ein goto verwendet. Exceptions sind somit mit Vorsicht anzuwenden und sollten nicht für den normalen Programmfluss eingesetzt werden.

2.6.6 is... Methoden

Für Zustandsabfragen sind `is..` Funktionen zu verwenden.
Beispiel:

`isOpen()` ist besser als `if(state()==STATE_OPEN)`

2.6.7 Makros

Makros sind in C++ nicht mehr nötig. Die Verwendung von Inline Funktionen bietet hier die gleiche Performance mit einer verbesserten Übersichtlichkeit.

```
// do not use macros in C++ and Java
  #define  MAX(x,y) (((x) > (y) ? (x) : (y))   // Get the maximum

// use inline methods
   inline int
   max(int x, int y) {
      return (x > y ? x : y);
   }
```

Makros mit Seiteneffekten sind zu vermeiden.

`MAX(f(x),z++);` // !! vermeiden !!

2.6.8 Interfaces

In C++ können „Abstrakte-Basis-Klassen“ für die Definition eines Interfaces verwendet werden,.

```
   class Jumpable
   {
    public:
      virtual void Jump() = 0;  // pure virtual method
   }
```

Und die Implementierung:

```
   class Frog : public Jumpable
   {
    public:
      virtual void Jump() { jump frog jump; }
   }
```

Interfaces oder abstrakte Basisklassen sind jedoch nicht die Lösung aller Probleme. Nicht in jedem Falle sind diese sinnvoll. Der Einsatz muss von Fall zu Fall überprüft werden.

2.6.9 return-values / Rückgabewerte

Die return-values, vor Allem bei Systemcalls sollten immer überprüft werden und bei Meldung eines Fehlers dieser weitergereicht oder ausgegeben werden.
Bei allen Speicheranforderungen (malloc/realloc/...) muss auf jeden Fall auf Erfolg geprüft werden und bei Misserfolg der weitere Programmablauf daran angepasst werden – oder abgebrochen werden.

2.6.10 Operatoren nicht überladen

Außer bei speziellen mathematischen Klassen sollten die Operatoren nicht überladen werden. Der Zusammenhang zwischen der mathematischen Logik des Operators und der Klasse muss klar sein – auch den späteren Lesern des Codes.

2.7 C/C++ Coding Rules

Hier sind weitere Regeln, speziell für C oder C++ aufgeführt. Die Regeln ergänzen die obigen Regeln.

2.7.1 Pointer

Dem Name sollte ein „p“ vorgestellt sein. Das „*“ sollte nahe am Variablenname stehen, kein Space zwichen „*“ und Name.
Bei der Definition steht der „*“ neben der Typbezeichnung, da er den Typ der Definition verändert.
Es ist wichtig, Referenzen von Pointern unterscheiden zu können.

```
String* pName= new String;
```

2.7.2 Referenzen

Um Referenzen erkennen zu können wird „r“ vorangestellt.
Beispiel:

```
StatusInfo&        rStatus():       // liefert Referenz
const StatusInfo&  Status() const;  // liefert nicht änderbare Daten
```

2.7.3 Typedef

Typedef ist ein Vorgänger der Klassen. Somit sind Types „Dinge“ wie Klassen auch und sollten den gleichen Namenskonventionen folgen. Zur Unterscheidung Type-Klasse wird dem Namen ein „Type“ hinten angestellt.

2.7.4 Keine Enums

Als konstant deklarierte Variablen sind der Definition von enum vorzuziehen. Eine Ausnahme kann hier bei extrem kleinen Speicherverhältnissen (embedded systems) gemacht werden, dort sind enums evtl. vorteilhaft.

2.7.5 Header File Guard

Jede Include-Datei sollte gegen mehrfach inkludieren geschützt sein. Dazu benötigt jede Datei einen eindeutigen Define. Als Namen ist der Name der Klasse oder der Name der Datei zu verwenden.

```
#ifndef _ClassName_h_
#define _ClassName_h_
. . .
#endif // _ClassName_h_
```

2.7.6 boolean

Unterschiedliche boolean Definitionen wurden in der Vergangenheit verwendet. Dies führt natürlich zu extremen Problemen bei Logiktests.

Folgender boolean wird ausschließlich verwendet:

```
typedef int     bool;
#define FALSE   0
#define TRUE    1
```

Ein Vergleich ist nur mit den oben definierten Symbolen erlaubt. Ausdrücklich Verboten ist ein Vergleich mit den Zahlenwerten.

```
if (f() == TRUE)   // ist richtig
if (f() == 1)      // ist verboten
```

Viele Funktionen liefern 0 für Fehler und einen positiven Wert für den Gutfall. Hier muss die Abfrage lauten:

```
if (f() != FALSE)
```

2.7.7 Dateinamen

Die Klassendefinition ist in einer eigenen Datei und enthält einen „Header File Guard“

`ClassName.h`

Die Implementation in:

`ClassName.cpp` (oder einer analogen Erweiterung)

2.7.8 Streams

Streams statt stdio.h (printf) verwenden. Streams sind Typen-sicher und arbeiten mit den meisten Typen und den meisten Ausgabegeräten (Bildschirm, Datei, Drucker,...).

Jedoch: printf (etc.) kann bei embedded Systemen mehr Sinn machen als steams.

2.7.9 Fallstricke bei C/C++

Werden die Coding Rules in diesem Dokument beachtet, so sollten folgende Fehlermöglichkeiten verhindert werden. Zur Klarheit sind diese hier nochmals aufgelistet.

Dieser Code will eine Datei kopieren. Aber: != bindet stärker als =. Die Datei enthält somit nur 1 [getc(in) != EOF ist true, also 1].

```
while (c=getc(in) != EOF)     // wrong
      putc(c,out);

while ((c=getc(in)) != EOF)   // right
      putc(c,out);
```

Typischer C Fehler, dass == vergessen.
Wenn dies wirklich gewünscht wird, so sollte dies vermerkt werden. Sonst ist dies als Fehler zu werten.

```
if( x = y)         // wrong

if ( x==y )        // right
if ((x = y) != 0) // right
```

Das Thema mit der Kommentierung /*
Kommentare innerhalb einer Zeile sind verboten, da diese zu extrem unübersichtlichem Code führen. Vor dem * eines Pointers muss immer ein Leerzeichen stehen, damit ist dieser besser sichtbar und die Fehlerquelle vermieden.

```
y = x/*p;                // is not a pointer, ist a comment !!
y = x / *p;              // better
y = x/(*p);              // even better
a + /* strange */ = 1    // this means a+=1 !!
```

Achtung bei Abfragen mit und/oder (& oder |). Ein häufiger Fehler tritt durch die stärkere Bindungskraft von = und != auf.

```
if (flags & FLAG != 0) ..   //  mostly wrong, because this means:
if (flags & (FLAG != 0) ) ..

if( (flags & FLAG) != 0) ..  // this is what the programmer wants
```

Ein weiteres Bindungsproblem tritt bei << / >> auf.

```
r = h<<4 + l;       // mostly wrong, because this means:
r = h<<(4 + l);

r = (h<<4) + l;     // this is what the programmer wants
```

Eine leere if Anweisung kann schwer erkannt werden.

```
if (x[i] > big);
   big = x[i];
```

2.7.10 C++ für Embedded Systems

C++ eignet sich sehr gut für performancekritische Anwendungen. Es hat gegenüber C kaum Laufzeitnachteile, bietet jedoch immense Vorteile bei der Wartung und Wiederverwendung.

Folgende Punkte sollten beachtet werden, um C++ effizient in Applikationen mit kleinem Speicher und langsamer CPU einzusetzen.

- Keine Streams verwenden (<<,>>).
- Keine Templates verwenden (je nach Compiler können Templates großen Code generieren).
- Exceptions vermeiden. Interrupt getriggerte Systeme müssen „multi-thread sicher“ sein. Eine Interrupt Funktion sollte keine Exceptions generieren.
- Wird ein OS verwendet, so sollte eine Klasse die OS Funktionalität abkapseln. Dadurch wird eine Portierbarkeit auf anderes OS möglich.
- Das Speicherhandling, vor Allem das Sharing zwischen Interruptservice-Routine und normalen Tasks muss durchdacht werden.
- Memory-Mapped IP Ports müssen als volantile deklariert werden um eine eventuelle Optimierung durch den Compiler zu verhinern.

C++ Classfile Template

```
// NAME:
//   XX -
//
// SYNOPSIS:
// DESCRIPTION:
// AUTHOR:
// VERSION:
// HISTORY:
// EXAMPLES:
// FILES:
// BUGS:
// SEE ALSO:
// TYPE:
///X//////////////////////X//////////////////X//////////////////////////////

#ifndef _XX_h_         // XX= your new class name
#define _XX_h_

// SYSTEM INCLUDES
//
// PROJECT INCLUDES
//
// LOCAL INCLUDES
//
// FORWARD REFERENCES
//
class XX
{
 public:
    //====================LIFECYCLE=======================
    XX();
    XX(const XX&);                     // copy constructor
   ~XX();

   //=====================OPERATORS=======================
   XX&    operator=(XX&);              // assignment operator

   //=====================OPERATIONS======================
   set...
   compute...

   //=====================ACCESS==========================
   get...

   //=====================INQUIRY=========================
   is...

   ////////////////////// PROTECED ///////////////////////
   protected:

   ////////////////////// PRIVATE ////////////////////////
   private:
};

// INLINE METHODS
//
// EXTERNAL REFERENCES
//
#endif  // _XX_h_
```

2.9 Java Class Template

```
/*
 * @(#)classname.java        Version Date
 *
 * Copyright (c) 2003  company name
 * All rights reserved.
 *
 */
package java.classname;

import java.xyz.xyz;

/**
 * Class description goes here.
 *
 * @version     Version   Date
 * @author      Firstname Lastname
 */
public class classname extends SomeClass {
    /* A class implementation comment can go here. */

    /** classVar1 documentation comment */
    public static int classVar1;

    /**
     * classVar2 documentation comment that happens to be
     * more than one line long
     */
    private static Object classVar2;

    /** instanceVar1 documentation comment */
    public Object instanceVar1;

    /** instanceVar2 documentation comment */
    protected int instanceVar2;

    /** instanceVar3 documentation comment */
    private Object[] instanceVar3;

    /**
     * ...constructor Blah documentation comment...
     */
    public classname () {
        // ...implementation goes here...
    }

    /**
     * ...method doSomething documentation comment...
     */
    public void doSomething() {
        // ...implementation goes here...
    }
}
```

3 Kommentierung

3.1 Verzeichnisse

Jedes Verzeichnis im Sourcecode-Baum sollte ein REAME enthalten. In diesem ist vermerkt:

- Wozu dienen die Dateien in diesem Directory;
- Build Instruktionen, spezielle Compiler-Einstellungen, welches Makefile macht was, etc;
- Welche Dokumentationen sind verfügbar – Verweis auf gedruckte Doku oder andere Dokumente.

3.2 Copyright Hinweis

Jede SourceCode Datei enthält am Beginn einen Hinweis auf den Copyright-Inhaber. Analog gilt dies für andere wichtige Dateien, wie sprachabhängige Textdateien und Makefiles.

```
/* Copyright Notice ====================================*/
/* This file contains proprietary information of        */
/*    <<company name>>                                  */
/* Copying and reproduction without prior written       */
/* approval is prohibited.                              */
/* Copyright (c) 2003                                   */
```

3.3 Dateikopf / Fileheader

Oft werden automatische „Sourcecode zu Dokumentierung“ Umsetzer verwendet. Das von diesen verwendete Format muss dann natürlich eingehalten werden. Ein gebräuchlicher Header ist hier exemplarisch angegeben.

```
// NAME: (=short summary)
//   XX - (=class name, modul name)
//
// SYNOPSIS:
// DESCRIPTION:
// AUTHOR:
// VERSION:   (last change)
// HISTORY:   (who changes what, why and when)
// EXAMPLES:
// FILES:
// BUGS:
// SEE ALSO:
// TYPE:
///X//////////////////////X//////////////////X//////////////////////////
```

Bei C kann das obige Format mit der /* Kommentierung verwendet werden. Bei Skripten wird häufig als Kommentarzeichen „#“ verwendet.

Die Angaben bedeuten:

`Name`	Kurze Beschreibung der Datei (Für was, Was macht die)
`XX`	Der Klassenname (oder der Moduls, etc.)
`SYNOPSIS`	Was wird benötigt (andere Libraries, externe Abhängigkeiten). Welches Interface bietet dieses Modul.
`Description`	Eine ausführliche Beschreibung der Moduls, der Klassen.
`///x/////x`	Die letzte Zeile dient als Seperator und kann gleichzeitig die bevorzugte Einrückung für diese Datei festlegen. Das erste X markiert dabei die Position an der Methoden-Typen beginnen, das zweite X die Position der Methoden-Namen und das dritte X die Position für Inline-Kommentare (Kommentare in gleicher Zeile wie SourceCode).

Versionsangaben, Autorangaben und eine Historie kann entfallen, wenn diese konsequent in einem Versionsverwaltungssystem enthalten sind und dort abgerufen werden können. Da dies jedoch die Weitergabe des Sourcecodes erschwert, sollten diese Angaben in der Datei vermerkt werden.

Alternativer Dateikopf:

```
/*
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

3.4 Methoden

Bei Methoden mit trivialen Kommentaren kann dieser als inline Kommentar hinter den Methodename geschrieben werden. Triviale Kommentare sind jedoch oft wenig hilfreich (z.B. „dies ist ein Konstruktor“).
Der normale Methoden-Kommentar sieht wie folgt aus.

```
// Description
// PRECONDITION:
//   The Open() method MUST be called before this.
// WARNING:
//   LOCK REQUIRED: lock name
// RETURN: OK: success
//         FAIL: a bad thing happened
int       AnyMethod();
```

Oder **JavaDoc** konform:

```
/**
 * <detailed description of the method>
 *
 * @param  <description of each parameter>
 * @return <explain each return type>
 * @exception <explain each exception>
 * @author  <your name>  <mm-dd-yy time>
 * PRECONDITION:
 * WARNING:
 **/
```

Jede public Methode (=das Klasseninterface) sollte einen Java-Doc konformen Kommentar vorgestellt haben. Dieser sollte obiges Format haben, wobei nicht relevante Zeilen ausgelassen werden können.

`Description`	Beschreibt kurz und klar, was die Funktion macht, für was diese verwendet werden kann.
`Precondition / Warning`	Falls die Funktion irgendwelche Vorbedingungen hat (z.B. die Datei muss bereits geöffnet sein), so sind diese zu vermerken. Ist die Funktion sehr laufzeitintensiv, erfordert z.B. besondere Echtzeitanforderungen oder benötigt viel Speicherplatz, so sind diese Aspekte ebenfalls zu dokumentieren. Wird ein Semaphor oder anderes Verfahren benötigt um eine Ununterbrechbarkeit zu gewährleisten, so muss dies zwingend vermerkt werden.
`Return`	Beschreibung der Rückgabewerte (Typen) und deren Bedeutung. Insbesondere der Fehlercodes und ihrer Bedeutung.

Unzutreffende Teile des Kommentars werden auslassen. Alle für die Funktion nicht zutreffenden Teile der obigen Vorlage entfallen, d.h. wenn Leer werden diese nicht als Kommentar niedergeschrieben. (z.B “Precondition: keine“ vermeiden).

Beispiele:

```
 * @param aSource - the input source string.
 *                  Cannot be 0-length.
 * @return Possible values are 1..n
 * @exception ResourceNotFoundExeption. Recoverable,
 *            try another resource
```

3.5 Methoden mit vielen Argumenten

Generell sollte eine Funktion/Methode möglichst wenig Übergabeparameter haben.

Hat die Klasse viele Methoden, welche immer die gleichen Übergabewerte verwenden, so kann dies ein Anzeichen für ein Klassenattribut sein.

Jedes übergebene Attribut sollte dokumentiert sein. Besonders wichtig ist diese Dokumentation für die Klassenschnittstelle, also alle public Methoden.

3.6 Spezielle Keyworte

Um durch Textsuche oder Robots verschiedene Kommentararten finden zu können, sollten stets feste Keywörter verwendet werden.

`TODO: xxxx`	Der Methode/Klasse fehlt noch etwas. Dies ist hier vermerkt und sollte nicht vergessen werden. Ausgelieferte Module sollten keine TODO mehr enthalten.
`BUG: xxxx`	Hier wurde ein Fehlverhalten festgestellt aber noch nicht korrigiert. Dies resultiert in ein TODO.
`TRICKY`	Der folgende Code ist trickreich – also schwer verständlich. Bei einer Änderung ist große Sorgfalt notwendig. Kann in einer Warning enthalten sein.
`WARNING: xxx`	Ein wichtiger Hinweis, welcher vor einer Änderung beachtet werden muss.

Beispiel:

```
// :TODO: tmh 960810: possibile performance problem
// We should really use a hash table here but for we'll
// now use a linear search.

// WARNING: this code is tested and works, do not
//          change them.
```

:

4 Anderes

4.1 Konfigurationsmanagement

Schon vor dem Projektbeginn sollte ein System zur SourceCode-Verwaltung aufgebaut sein. Ein kleines Handbuch mit den zu beachtenden Regeln muss erstellt werden und jedem Programmierer vorliegen.
Ein Master sollte bestimmt werden, der Hilfestellung und Einweisungen geben kann, sowie den Zustand des Archivs stichprobenartig überprüft.

Die Dokumentation sollte enthalten:

- Wie können die Module ausgechecked werden?
- Welche Regeln für makefiles müssen beachtet werden?
- Wie werden neue Module aufgenommen? Welche Dokumentation, Kommentare sind vorzunehmen, wer ist zu benachrichtigen?
- Wie können Dateien gelöscht werden? Wann ist der Master zu benachrichtigen.
- Welche gemeinsamen Libraries gibt es?
- Wie werde Module als final gekennzeichnet? Welche Freigabeprozedur muss eingehalten werden?
- Welche Kommentierung ist Wann bei einem checkin notwendig? Wie sind Änderungen zu kommentieren?
- Wie kann das Build-Environment aufgebaut werden? Welcher Compiler, welches OS, welche SourceCode Verwaltung, etc.
- Gibt es ausgezeichnete Owner von Modulen? (Nur diese dürfen Ändern, oder müssen jede Änderung genemigen.)
- Die Dokumentation sollte jedem Programmierer zur Verfügung stehen. Ab besten auf einer gemeinsamen Intranet-Seite.

Tools dafür sind z.B. CVS oder Clear Case.

4.2

Bug Tracking System

Analog zum Konfigurationsmanagement sollte eine Bug-Tracking Prozedur installiert werden. Diese kann automatisiert oder manuell ablaufen und sollte folgende Punkte abdecken:

- Probleme werden nicht vergessen, sie werden festgehalten.
- Probleme werden (automatisch) zu den Verantwortlichen weitergemeldet.
- Der Lebenszyklus der Probleme kann verfolgt werden (offen, in test, fixed).
- Anzahl der Restfehler kann helfen die Projektrestzeit einzuschätzen.
- Jedes Problem / Bug erhält eine eindeutige ID. Diese soll auch beim SourceCode-Management verwendet werden. (e.g. Bug #123 fixed).
- Das Bug-Tracking-System kann als Kommunikationsplattform von QA und technischem Support zur Entwicklung dienen,

Tools: DDTS