

# Codingstandard

Softwareentwicklung Praktikum Stand: 27.02.2008

## I. Einleitung

Wie in der Vorlesung schon ausgeführt wurde, ist die Lesbarkeit und Wartbarkeit des Sourcecodes ein sehr wichtiges Kriterium für guten Code. Ein Coding Standard (CS) gibt nun Regeln vor, die darauf abzielen, die Lesbarkeit des Source Codes zu verbessern; neben generellen Anleitungen auch genaue Formatierungsregeln. Dies erleichtert die Lesbarkeit des Codes sowohl für die Entwickler selbst als auch für Unbeteiligte. Der für diese LV vorgegebene CS ist eine vereinfachte Version des von K. Schmaranz<sup>1</sup> in seinem Buch "Softwareentwicklung in C++" verwendeten.

## II. Allgemeine Regeln

Die Sprache des Source Codes und der Kommentare ist Englisch.

**Einfachheit:** Im Englischen auch als KISS (Keep it small and simple) bekannt. Es ist eine sinnvolle Aufteilung anzustreben, sodass Funktionen eine bestimmte Aufgabe erfüllen und keine Seiteneffekte haben. Die Aufgabe der Funktion spiegelt sich in deren Namen wieder.

**Verständlichkeit:** Der Source Code sollte von anderen Personen bei einmaligem Durchlesen sofort verstanden werden können. Dazu ist vor allem eine sinnvolle Namenswahl für Funktionen und Variablen nötig (Beispiel: `I` ist kein guter Variablenname und daher nicht erlaubt. Besser `lastname`.)

**Einheitlichkeit:** Ähnliche Codeteile sollen auch gleich aufgebaut sein. Zum Beispiel sollen verwandte Funktionen, wie `append` und `remove`, die Parameter in derselben Reihenfolge haben. Auch lokale Variablen für dieselbe Aufgabe sollen in allen Codeteilen gleich benannt werden.

## III. Detail Regeln

- **Globale Variablen** sind zu vermeiden.
- **Kommentare** sind als Zeilenkommentare anzugeben (Also mit `//` und nicht `/* ... */`).
- **Unnötige Kommentare** sind zu unterlassen. Beispiel: `int number = 5; // hier wird der Wert 5 zugewiesen`
- **Namenskonvention:** Durch diese Namenskonvention wird erreicht, dass schon aufgrund der Schreibweise klar ist, um welches Sprachkonstrukt es sich handelt.

Klassen sowie typedef'ed Enums, Structs und Unions	AllWordsCapitalizedWithoutUnderscore
Methoden und Funktionen	firstWordLowerCaseRestCapitalizedWithoutUnderscores
Lokale Variablen	all_lower_case_with_underscores
Klassen und Instanz Member Variablen	all_lower_case_with_underscores_and_trailing_underscore_
Konstanten (auch in Enum)	ALL_UPPERCASE_WITH_UNDERSCORES
Namespaces	AllWordsCapitalizedWithTrailingUnderscore_
Exceptions (müssen mit <i>Exception</i> enden)	AllWordsCapitalizedWithoutUnderscoreEndsWithException
Nicht typedef'ed Enums, Structs, Unions	_AllWordsCapitalizedWithUnderscoresOnBothEnds_

<sup>1</sup> Klaus Schmaranz. *Softwareentwicklung in C++*, Kapitel Coding-Standard. Springer, 2001.

- Jede Klasse besteht aus einem **Header** (klassenname.h) und dem **Code** (klassenname.cpp). Eine Implementation im Header ist nicht erlaubt *außer*:
  - Sehr kleine Klassen die logisch zusammengehören wie zum Beispiel die Exceptions eines Moduls können in einem Header und cpp Datei Paar zusammengefasst werden (Bsp.: input\_exceptions.h und input\_exceptions.cpp).
  - Bei Templates ist die Trennung Header und Code Datei wegen Compiler Inkompatibilitäten nicht nötig.
- Statt C **Casts** sind nur die entsprechenden Casts aus C++ zu verwenden.
- Jede Datei enthält einen Kommentar **Header**. Dieser beginnt mit dem Dateinamen und einer kurzen Beschreibung und enthält meist noch weitere Angaben. Zumindest erforderlich sind Name und Matrikelnummer aller Autoren.
- Formatierungsregeln: Um die Formatierungsregeln besser zu verstehen, sollte unbedingt das Code-Beispiel genau betrachtet werden.
  - Vor jeder Funktion steht eine Zeile dieser Form (Länge 60-70 Zeichen):  

```
//-----
```
  - Nach dieser **Trennline** ist die Funktion kurz zu beschreiben. Speziell auch die Parameter und Rückgabewerte.
  - Block Klammern stehen in eigenen Zeilen.
  - Jeder Block **muss eingerückt** werden (**2 Leerzeichen**).
  - Zeilen dürfen eine Breite von **72 Zeichen** nicht übersteigen (siehe Code Beispiel für Umbruch)
  - Zwischen Parametern und Operatoren sind Leerzeichen einzufügen
- **Verschachtelungen** sollten nicht zu tief sein (Richtwert maximal 4 Ebenen). Dabei ist besonders auf Übersichtlichkeit und korrekte Einrückung zu achten.
- Funktionen und Methoden sollen eine **Länge** von 70 Zeilen nicht übersteigen.
- **Initialisierung** ist gegenüber der Zuweisung zu bevorzugen:
  - Gut: 

```
Class::Class():counter_(0)
    {
    }
```
  - Schlecht: 

```
Class::Class()
    {
    counter_ = 0;
    }
```
- Using Direktiven wie `using namespace std` sind zu vermeiden. Stattdessen sollten using Deklarationen wie `using std::cout` verwendet werden.
- In **Headerdateien** sind keinerlei using Statements erlaubt! Stattdessen ist immer direkt **Scoping** zu verwenden (z.B.: `printString(std::String text);`)
- **Includes** in Header Dateien sind wenn möglich zu vermeiden und durch forward Deklarationen zu ersetzen.
- Es sind die C++ **Standard Header** zu inkludieren (ohne .h) Bsp.: `#include<stdio>` (eigene Header mit .h). Includes sind zu gruppieren.
- Für eigenen Code ist ein **Namespace** zu verwenden (siehe auch Punkt 3, Namenskonvention)
- Für **Konstanten** ist im Allgemeinen `const` und nicht `#define` zu verwenden. (Hinweis: Es gibt auch `enum`)
- **const** wird linksbindend verwendet. Also `int const counter;` statt `const int counter;`

- **Pointer**-Deklaration: `char *c_string` statt `char* c_string` (Stern immer direkt vor Variablenname)
- **Public Members** in Klassen sind nicht erlaubt außer sie sind `const`. Members dürfen also nur durch Methoden verändert werden.
- `public`, `protected` und `private` Methoden und Members sind zu **gruppieren** (also nicht ein `public`, `private` und wieder ein `public member`).
- Alle Header müssen durch **Makros** gegen mehrfaches Inkludieren geschützt werden. Als Namenskonvention gilt folgendes:

```
#define FILE_NAME_H_
```

Beispiel für die Datei `specialcounter.h`:

```
1 #ifndef SPECIAL_COUNTER_H_
2 #define SPECIAL_COUNTER_H_
3
4 //insert the declarations here
5
6 #endif //SPECIAL_COUNTER_H_
```

- **Exceptions**: Für alle eigenen Exceptions muss es eine von `std::exception` abgeleitet Basisklasse geben. Beispiel:

```
1 class OwnBaseException : public std::exception
2 {
3 };
4
5 class InvalidArgumentException : public OwnBaseException
6 {
7     InvalidArgumentException(string const &message);
8 };
9
10 // throwing an exception . . .
11
12 void testFunction ()
13 {
14     throw InvalidArgumentException ("Function testFunction has been
15         called with invalid arguemnts");
16 }
```

- Exceptions: Im **Destruktor** dürfen keine Exceptions geworfen werden.
- Bei Typedefs auf einen Standarddatentypen wird der definierte Datentyp klein geschrieben, z.B.:

```
typedef double coord;
typedef vector<int> choice;
```

## IV. Kurzes Code Beispiel (Cheat Sheet)

```
1 //-----
2 /// Filename: main.cpp
3 /// Description: main file
4 /// Authors: Andreas Weinberger M# 9XXXXXX
5 /// Group XXX, Tutor XYZ
6 /// Date of Creation: 2007-02-27
7 /// Last Changes: 2007-02-27
8 //-----
9 #include "account.h"
10
11 #include <iostream>
12
13 using std::cout;
14
15 int main(int args, char *argv[])
16 {
17     CodingStandardExample_::Account account = CodingStandardExample_::Account(0);
18     cout << account.getBalance() << std::endl;
19     account.depositMoney(100);
20     cout << account.getBalance() << std::endl;
21     account.withdrawMoney(77);
22     cout << account.getBalance() << std::endl;
23 }

```

```
1 //-----
2 /// Filename: account.h
3 /// Description: money account to deposit and withdraw money
4 /// Authors: Andreas Weinberger M# 9XXXXXX
5 /// Group XXX, Tutor XYZ
6 /// Date of Creation: 2007-02-27
7 /// Last Changes: 2007-02-27
8 //-----
9 #ifndef ACCOUNT_H_
10 #define ACCOUNT_H_
11
12 namespace CodingStandardExample_
13 {
14 {
15     //-----
16     ///
17     /// Account
18     ///
19     /// a simple account for money
20     ///
21     /// @author <Andreas Weinberger sep-tw-andi@iicm.edu>
22     /// @version <1.0.0>
23
24     class Account
25     {
26     public:
27
28         //-----
29         ///
30         /// constructor
31         /// @param start_amount Initial balance for this account
32
33         Account(int start_amount);
34
35         //-----
36         ///
37         /// @return int the current balance
38
39         int getBalance();
40
41         //-----
42         ///
43         /// deposit the amount of money to the account
44         /// @return void No description needed in this case
45         /// @param amount Amount of money to deposit on the account, if
46         /// there are more parameters use a separate param tag for each
47         /// @exception Here would be the exception names and description if
48         /// there would be a throw clause
49
50         void depositMoney(int amount);

```

```

51
52 //-----
53 ///
54 /// decrease the balance by the given amount
55 /// @param amount Amount of money to withdraw, if there are more
56 /// parameters use a seperate param tag for each
57
58 void withdrawMoney(int amount);
59
60 protected:
61     int balance_;
62 };
63
64 }
65
66 #endif // ACCOUNT_H_

1 //-----
2 /// Filename: account.cpp
3 /// Description: money account to deposit and withdraw money
4 /// Authors: Andreas Weinberger M# 9XXXXXX
5 /// Group XXX, Tutor XYZ
6 /// Date of Creation: 2007-02-27
7 /// Last Changes: 2007-02-27
8 //-----
9 #include "account.h"
10
11 //-----
12 CodingStandardExample_::Account::Account(int start_amount):
13     balance_(start_amount)
14 {
15 }
16
17 //-----
18 int CodingStandardExample_::Account::getBalance()
19 {
20     return balance_;
21 }
22
23 //-----
24 void CodingStandardExample_::Account::depositMoney(int amount)
25 {
26     balance_ += amount;
27 }
28
29 //-----
30 void CodingStandardExample_::Account::withdrawMoney(int amount)
31 {
32     balance_ -= amount;
33 }

```