Kommentierung in C

Von Sebastian Rothe

Inhaltsverzeichnis

1. Programmierstil - eine Übersicht

Betrachtet man das Thema *Kommentierung*, sollte man auch auf das Thema *Programmierstil* eingehen, da es sich hierbei um den übergeordneten Begriff handelt. Als Programmierstil werden verschiedene Richtlinien angesehen, nach denen der Code gestaltet werden soll. Er umfasst in der Regel verschiedene Aspekte:

- Quelltextformatierung: Interessant sind hier verschiedene Themen wie Klammerung, Einrückungen, vertikale Anordnung von Variablen, die Nutzung von Tabulatoren und/oder Leerzeichen oder auch Left-hand- vs. Right-hand-comparisons.
- Namenskonvention: Vor allem in Team-Projekten ist es wichtig, Konventionen für Variablenbennung zu nutzen. Konstanten in Großbuchstaben oder Funktionsnamen, Typendefinitionen und Variablen in Kleinbuchstaben sind nur zwei von vielen Konventionen, die oft genutzt werden.
- Wiederverwertbarkeit/Wartung und Modularität der Software: Dies sind zwei Begriffe, die sehr eng miteinander verbunden sind. Der Programmcode sollte so gestaltet werden, dass verschiedene Teile der Software möglichst unabhängig voneinander gebaut werden und daher leicht gewartet, erweitert, geändert oder sogar ausgetauscht werden können.
- Robustheit durch Fehlerbehandlung: Sowohl für den Nutzer einer Software als auch den Entwickler ist es wichtig, dass ein Programm bei einem Absturz ausreichend über den Grund informiert. Außerdem kann Fehlverhalten auch abgefangen werden, sodass der Anwender davon nichts mitbekommt.
- Und natürlich auch die Kommentierung.

2. Motivation - wozu Kommentierung?

Kommentierung wird leider von vielen Programmierern oftmals unterschätzt und als lästig empfunden, da es doch einen gewissen Mehraufwand darstellt. Daher wird es oft auch sehr vernachlässigt. Es sprechen jedoch mehrere wichtige Punkte für eine gute Kommentierung.

Zunächst einmal wird eine Software im Teamprojekt nur selten von ihrem ursprünglichen Autor weitergepflegt. Folglich werden weitere Entwickler an dieser arbeiten. Gute und sinnvolle Kommentare helfen daher diesen, um sich schneller in die Software einarbeiten zu können. Außerdem verbessern Kommentare auch den "Lesefluss" eines Quellcodes, sodass beispielsweise schwierige Passagen nicht Codezeile für Codezeile aufgearbeitet werden müssen, sondern durch einen Kommentar passend zusammengefasst werden können. Es hilft darüber hinaus auch, die Software leichter, schneller und flexibler erweitern zu können.

Als Richtwert wird angesehen, dass ungefähr 80% der Lebenszeit einer Software auf die Wartung entfällt.

"Effiziente Programmierung in C" - der Bezug zum Seminartitel?

Betrachtet man andere Bereiche dieses Seminars, so stellt man sich die Frage, inwiefern die Effizienz durch Kommentierung beeinflusst werden kann. Es ist offensichtlich, dass der Effizienzgewinn hier nicht in der Performance der Software wiederzufinden ist. Die Effizienz liegt vielmehr in der Erstellung des Codes. Verschiedene bereits durchgeführte Arbeitsschritte können schneller und besser nachvollzogen werden, darauf aufbauende Ideen können also effizienter eingepflegt werden. Auch für später folgende Aufgaben, wie beispielsweise die Wartung, hat gute Kommentierung positive Auswirkungen. Denn auch der beste Programmierer erinnert sich sicherlich nach einiger Zeit nicht mehr an alle "Kniffe", die er in einer Software verbaut hat. Ein kurzer Kommentar kann hier also schnell auf die Sprünge helfen.

Kurz gesagt kann Kommentierung präventiv für effizienteren Code sorgen.

Sinnvolle und unsinnige Kommentare

Alle bereits angesprochenen Punkte setzen natürlich voraus, dass Kommentierung sinnvoll verwendet wird. Nutzt man hingegen wenig sinnvolle Kommentare, kann der Effizienzgewinn sehr schnell wieder zunichte gemacht werden.

Ein oft gemachter Fehler ist hier beispielsweise eine Wiederholung des Quelltextes. Der Entwickler gewinnt dadurch keinerlei neue oder hilfreiche Informationen, wird stattdessen im schlimmsten Fall sogar noch im Lesefluss gestört. Auch oft genutzt werden Kommentare, die einen einfachen Zusammenhang erklären sollen, der allerdings ohne Kommentar bereits ersichtlich sein sollte. Ebenfalls unsinnig sind Kommentare, die nur von dem Autor selbst nachvollzogen werden können. Stattdessen sollte sich der Entwickler um sinnvolle Kommentare bemühen. Diese sollten umfangreiche Codepassagen zusammenfassen, sodass weitere Programmierer, die an dieser Software arbeiten, erst bei genauerer Betrachtung den wirklichen Code betrachten müssen und ansonsten für das grobe Verständnis die Zusammenfassung nutzen können. Ebenso sollen komplizierte Anweisungen einfach dargestellt werden. Eine weitere Möglichkeit zur Nutzung von sinnvollen Kommentaren ist eine gute Strukturierung des Codes zu erzielen.

```
8  int functionOne(int arg1, int arg2, int arg3){
9      int result; //result var
10     int sum;    //sum var
11
12     sum = /*arg1 from function call*/ arg1 + arg2; //sum of arg1 and arg2
13     int produkt = doProduct(arg3, sum); //product call for arg3 & sum
14     //DEPRECATED!!!!
15     //doSomethingElseWithSum(sum);
16     //TODO find a better idea of what to do with sum
17
18     if(sum>arg3){
19         //does something with sum
20         doSomethingWithSum(sum);
21     } //if(sum>arg3)
22     else if(arg3>sum){
23         //doSomethingWithArg(arg3);
24         doSomethingElseWithArg(arg3);
25     } //else if(arg3>sum)
26     else doNothing();
27     //do nothing
28     return 0; //return 0 because we can
29 }
```

[Bild 1]: So sollte man es nicht machen...

3. Aspekte der Kommentierung

In der Programmiersprache C gibt es zwei verschiedene Arten von Kommentaren:

Der einzeilige Kommentar wird genutzt um eine gesamte Zeile bzw. das Ende einer Zeile mit hilfreichen Informationen zu versehen. Er ist zwar schon länger verfügbar, wird aber erst seit dem C99-Standard offiziell unterstützt. Zuvor war es möglich, dass Compiler diesen Kommentar falsch interpretierten und ein Fehlverhalten des Programmes hervorriefen.

```
1  // Dies ist ein einzeiliger Kommentar (seit C99 unterstützt)
2
3  int x = 42; //Ab hier wird die restliche Zeile als solcher betrachtet
```

[Bild 2]: Einzeiliger Kommentar

Der Blockkommentar ist ein Kommentar, der dafür genutzt werden kann, mehrere Zeilen im Quelltext mit Informationen zu versehen. Er kann aber auch einzeilig und innerhalb einer Anweisung genutzt werden. Letztere Nutzungsweise sollte allerdings vermieden werden.

```
1  /* Dies ist ein Blockkommentar,
2     der sich über mehrere Zeilen
3     erstrecken kann. */
4
5  int x = 42 + /*Er kann auch innerhalb einer Anweisung genutzt werden */ 13 - 21;
```

[Bild 3]: Blockkommentar

Diese beiden Möglichkeiten können nun sehr vielseitig zur Unterstützung genutzt werden. Im folgenden wird nun auf verschiedene Methoden eingegangen.

Kommentare als Zusammenfassung und Gliederung

Kommentare werden sehr häufig als eine Art "Inhaltsangabe" für einzelne Dateien (die dann oft einen bestimmten Bereich der Funktionalität einer Software bereitstellen) verwendet. Gebräuchliche Informationen dieser Inhaltsangabe sind unter anderem Dateiname, Autor(en), eine kurze

Beschreibung der folgenden Funktionalitäten und meistens auch ein kurzer Hinweis auf Copyright/Lizenzen.

```
1  /* GRegex -- regular expression API wrapper around PCRE.
2   *
3   * Copyright (C) 1999, 2000 Scott Wimer
4   * Copyright (C) 2004, Matthias Clasen <mclasen@redhat.com>
5   * Copyright (C) 2005 - 2007, Marco Barisione <marco@barisione.org>
6   *
7   * This library is free software; you can redistribute it and/or
8   * modify it under the terms of the GNU Lesser General Public
9   * License as published by the Free Software Foundation; either
10  * version 2.1 of the License, or (at your option) any later version.
11  *
12  * This library is distributed in the hope that it will be useful,
13  * but WITHOUT ANY WARRANTY; without even the implied warranty of
14  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
15  * Lesser General Public License for more details.
16  *
17  * You should have received a copy of the GNU Lesser General Public
18  * License along with this library; if not, write to the Free Software
19  * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
20  */
```

[Bild 4]: Kommentar als Zusammenfassung

Für die Gliederung werden oft einzelne Schlagworte oder kurze, prägnante Aussagen in sich wiederholende Zeichen eingebettet. Das lenkt schnell den Blick des Entwicklers auf diese Abtrennung bzw. Unterteilung.

```
210 /* ************************************************************************ */
211 /* mpiInitMatrices: Init Matrix for each Process                            */
212 /* ************************************************************************ */
```

[Bild 5]: Kommentar zur Gliederung

<u>Kommentare im Zusammenhang mit Funktionen</u>

Ebenfalls sehr häufig werden Kommentare genutzt, um einzelne Funktionen zusammenzufassen. Hierbei wird kurz die Funktionalität erläutert, gegebenenfalls gibt es auch noch Hinweise zur Nutzung, beispielsweise wenn es performantere Funktionen gibt, die Spezialfälle abdecken. Außerdem

werden oftmals noch kurz die Parameter aufgegriffen. Auch Erläuterungen oder zusätzliche Anmerkungen zum Rückgabewert findet man häufig.

```
499 /**
500  * g_match_info_get_string:
501  * @match_info: a #GMatchInfo
502  *
503  * Returns the string searched with @match_info. This is the
504  * string passed to g_regex_match() or g_regex_replace() so
505  * you may not free it before calling this function.
506  *
507  * Returns: the string searched with @match_info
508  *
509  * Since: 2.14
510  */
511 const gchar *
512 g_match_info_get_string (const GMatchInfo *match_info)
```

[Bild 6]: Kommentar als Kurzbeschreibung einer Funktion

Kommentare bei Schleifen

Für Schleifen gibt es zwei Nutzungen von Kommentaren. Einmal wird natürlich die Schleife erläutert, wobei hier vor allem interessant ist, wie oft die Schleife durchlaufen wird, aber auch was überhaupt in der Schleife geschieht. Letzteres kann durch weitere Kommentare innerhalb der Schleife ergänzt werden. Die andere Möglichkeit ist die Nutzung von Kommentaren zur "Markierung" von längeren Schleifen am Ende. Dadurch erhält der Quelltext eine bessere Strukturierung gerade bei Verschachtelung des Codes. Der Programmierer erhält dadurch außerdem schnell einen Überblick über größere Passagen. Man sollte allerdings berücksichtigen, dass es eventuell sinnvoller sein kann, zu lange Schleifen anders aufzubauen, indem man beispielsweise Funktionalitäten in Funktionen auslagert.

```
                    while( ( c = fgetc( datei ) ) != EOF ){
                        if( iFirstValues < 3 && ( ( 0 == iFileCounter ) || ( 0 == iFileCounter % iProcess ) ) )
                            //Nur jede iProzess-te Datei darf ein neues bmp angefangen werden
                            //& die Grund-Variablen verändert werden
                            //in der ersten Zeile müssen dann spezielle Daten ausgelesen werden
                            if( ( ';' == c ) || ( ',' == c ) ){
                                //Variable kann "gefüllt" werden
                                if( 0 == iFirstValues ){
                                    sscanf(cFirstRow, "%d", &iProcess);
                                    if( DEBUG ) printf("VARIABLE iProcess = %d\n", iProcess);
                                }
                                else if( 1 == iFirstValues ){
                                    sscanf(cFirstRow, "%d", &iSizeY);
                                    if( DEBUG ) printf("VARIABLE iSizeY = %d\n", iSizeY);
                                }
                                else if( 2 == iFirstValues ){
                                    sscanf(cFirstRow, "%d", &iSizeX);
                                    if( DEBUG ) printf("VARIABLE iSizeX = %d\n", iSizeX);
                                }
                                memset(cFirstRow, '\0', 6);
                                iFirstValues++;
```

[Bild 7]: Kommentar zur Schleifenerläuterung

Kommentare bei Bedingungen

Bei Bedingungen werden Kommentare oftmals genau wie auch bei den Schleifen verwendet. Wie dem folgenden Bild zu entnehmen ist, werden sowohl das Ende einer if-Bedingung markiert, als auch die darauf folgenden else-Blöcke kurz erläutert. Markierungen der einzelnen Blöcke sollten auch hier vor allem aufgrund auftretender Verschachtelung (und weniger wegen zu langen Blöcken) genutzt werden.

```
                                }
                            } //if(zusätzliche Zeilen nach iSizeY)
                            else{
                                //ansonsten normale Fallunterscheidung für iSizeY Zeilen
                                if( ':' == c ){
                                    //Wert folgt
                                    data.isPos = 0;
                                }
                                else if( ',' == c  || ';' == c){
                                    //neuer Datensatz bzw...
                                    if( 0 == iNewRow){
                                        data.isPos = 1;
                                        int iMomentaryProcess = iFileCounter % iProcess;
                                        mySetPixel(bmp, atoi(data.val), atoi(data.posX), iRowCounter,
                                        memset(data.posX, '\0', sizeof(data.posX)/sizeof(char));
                                        memset(data.val, '\0', sizeof(data.val)/sizeof(char));
                                    }
                                    else iNewRow = 0;
                                    if( ';' == c) iRowCounter++;
                                    //... neue Zeile
                                }
```

[Bild 8]: Kommentar zur Erläuterung einer Bedingung

TODO-Kommentare und Auskommentieren

Zwei weitere interessante Nutzungsmöglichkeiten von Kommentaren sind TODO-Kommentare und Kommentare zum Auskommentieren.

TODO-Kommentare werden oft als "provisorische Lückenfüller" genutzt. Hierfür wird an entsprechenden Stellen im Quelltext ein Kommentar eingefügt, der eine kurze Erläuterung enthält, was an dieser Stelle noch zu tun ist. Für diese Kommentare gibt es teilweise auch zusätzlich Unterstützung durch die IDE. So hat Eclipse eine Möglichkeit, aus einem Projekt alle TODO-Kommentare herauszufiltern und geordnet anzuzeigen. Dadurch können die Entwickler sich schnell einen Überblick über noch ausstehende Aufgaben verschaffen und an die entsprechenden Stellen innerhalb des Codes springen.

Oft sieht man Projekt, in denen jene Kommentare, die zum Auskommentieren genutzt werden, die einzigen Kommentare sind. Sie bieten eine sehr schnelle Möglichkeit zum Debuggen, da man interessante Codepassagen in einen Blockkommentar setzen kann und diese Passagen dann nicht mehr als Teil der Software interpretiert werden. Nachdem das neue Verhalten der Software getestet wurde, können die Abschnitte dann wieder einkommentiert werden.

4. GLib als Beispiel

GLib ist eine Bibliothek für C, die verschiedene Funktionen bereitstellt, die der Entwickler nicht noch einmal ausprogrammieren muss. Dementsprechend ist es natürlich wichtig, auf guten Programmierstil zu achten, damit diese Bibliothek auch weiterhin gewartet und vor allem erweitert werden kann. Kommentare sind dementsprechend ein wichtiger Bestandteil dieses Programmierstils. Die GLib-Bibliothek beinhaltet eine Vielzahl von Funktionen aus Bereichen wie Makros, Basistypen, Typumwandlungen, Timern, Datenstrukturen wie Listen oder Bäumen und Threads. Oft werden auch die String-Verarbeitungsmöglichkeiten der GLib genutzt.

In GLib lassen sich viele Aspekte der Kommentierung, die zuvor angesprochen wurden, wiederfinden. So verfügt jede Funktion über eine Funktionsbeschreibung mit einer kurzen Zusammenfassung sowie einigen Hinweisen. Zusätzlich sind immer wieder verschiedene Kommentare innerhalb von einzelnen Abschnitten erkennbar. Ebenfalls auffällig ist die gute und klare Codestruktur. Es werden also neben der Kommentierung auch andere Aspekte eines Programmierstils abgedeckt.

```
1265 /**
1266  * g_source_set_priority:
1267  * @source: a #GSource
1268  * @priority: the new priority.
1269  *
1270  * Sets the priority of a source. While the main loop is being
1271  * run, a source will be dispatched if it is ready to be dispatched and no sources
1272  * at a higher (numerically smaller) priority are ready to be dispatched.
1273  **/
1274 void
1275 g_source_set_priority (GSource  *source,
1276                        gint      priority)
1277 {
1278   GSList *tmp_list;
1279   GMainContext *context;
1280
1281   g_return_if_fail (source != NULL);
1282
1283   context = source->context;
1284
1285   if (context)
1286     LOCK_CONTEXT (context);
1287
1288   source->priority = priority;
1289
1290   if (context)
1291     {
1292       /* Remove the source from the context's source and then
1293        * add it back so it is sorted in the correct plcae
1294        */
1295       g_source_list_remove (source, source->context);
1296       g_source_list_add (source, source->context);
1297
1298       if (!SOURCE_BLOCKED (source))
1299         {
1300           tmp_list = source->poll_fds;
1301           while (tmp_list)
1302             {
1303               g_main_context_remove_poll_unlocked (context, tmp_list->data);
1304               g_main_context_add_poll_unlocked (context, priority, tmp_list->data);
1305
1306               tmp_list = tmp_list->next;
1307             }
1308         }
1309
1310       UNLOCK_CONTEXT (source->context);
1311     }
1312 }
```

[Bild 9]: Auszug aus einem Skript der GLib-Bibliothek

```
2605  #ifdef G_THREADS_ENABLED
2606    if (!context->poll_waiting)
2607      {
2608  #ifndef G_OS_WIN32
2609        gchar a;
2610        read (context->wake_up_pipe[0], &a, 1);
2611  #endif
2612      }
2613    else
2614      context->poll_waiting = FALSE;
2615
2616    /* If the set of poll file descriptors changed, bail out
2617     * and let the main loop rerun
2618     */
2619    if (context->poll_changed)
2620      {
2621        UNLOCK_CONTEXT (context);
2622        return FALSE;
2623      }
2624  #endif /* G_THREADS_ENABLED */
```

[Bild 10]: Nutzung eines Kommentars für Präprozessor-Anweisungen

Im zweiten Bild lässt sich die angesprochene Markierung am Ende einer Bedingung erkennen. Auch hier wird diese eher aufgrund der Schachtelung und weniger wegen einem zu langen Anweisungsblock genutzt. Ein weiterer interessanter Aspekt ist die abschließende Markierung für Präprozessor-Anweisungen.

```
3137                        g_print ("]");
3138                      }
3139                    i++;
3140                  }
3141                pollrec = pollrec->next;
3142              }
3143            g_print ("\n");
3144
3145            UNLOCK_CONTEXT (context);
3146          }
3147  #endif
3148      } /* if (n_fds || timeout != 0) */
```

[Bild 11]: Wiederholung der eigentlichen Bedingung als Abschluss

Hier ist die Abgrenzung für eine Bedingung gut erkennbar. Eine Besonderheit ist allerdings die Wiederholung der eigentlich Bedingung, also der if-

Anweisung. Dies ist eine gern genutzte Alternative zu einem kurzen zusammenfassenden Satz, der eine Bedingung (oder Schleife) beschreiben soll.

```
2711          /* attribute not collected:  could be caused by two things.
2712           *
2713           * 1) it doesn't exist in our list of attributes
2714           * 2) it existed but was matched by a duplicate attribute earlier
2715           *
2716           * find out.
2717           */
```

[Bild 12]: Kommentar im "eigentlichen Sinne"

Kommentare wie auf Bild Nummer 12 sieht man ebenfalls häufig. Hier wird kurz die momentane Situation erläutert, außerdem wird auch kurz darauf eingegangen, wie nun fortgefahren wird.

5. Dokumentationssysteme

Der Begriff "Quelltextdokumentationssystem" lässt zunächst auf ein Programm schließen, dass den Entwickler bei der Kommentierung unterstützt. Dies ist jedoch nicht der Fall. Vielmehr handelt es sich um eine Software zur automatischen Erzeugung von Dokumentationen im Sinne der "Java Platform API Specification" von Oracle für die Programmiersprache *Java*. Quelltextdokumentationssysteme nutzen hierfür den Quelltext, aber auch UML-Diagramme oder Grafiken zur Erzeugung einer solchen Dokumentation. Diese kann oftmals in verschiedenen Ausgabeformaten exportiert werden. html, LaTeX und XML sind sehr geläufig, aber auch PDF wird oft genutzt.

Dokumentationssysteme bringen oftmals einen nicht zu unterschätzenden Mehraufwand mit sich, da speziell für die Dokumentation die Software entsprechend gewartet werden muss (Anpassung/Einfügen der Kommentare in der jeweiligen Notation des Systems).

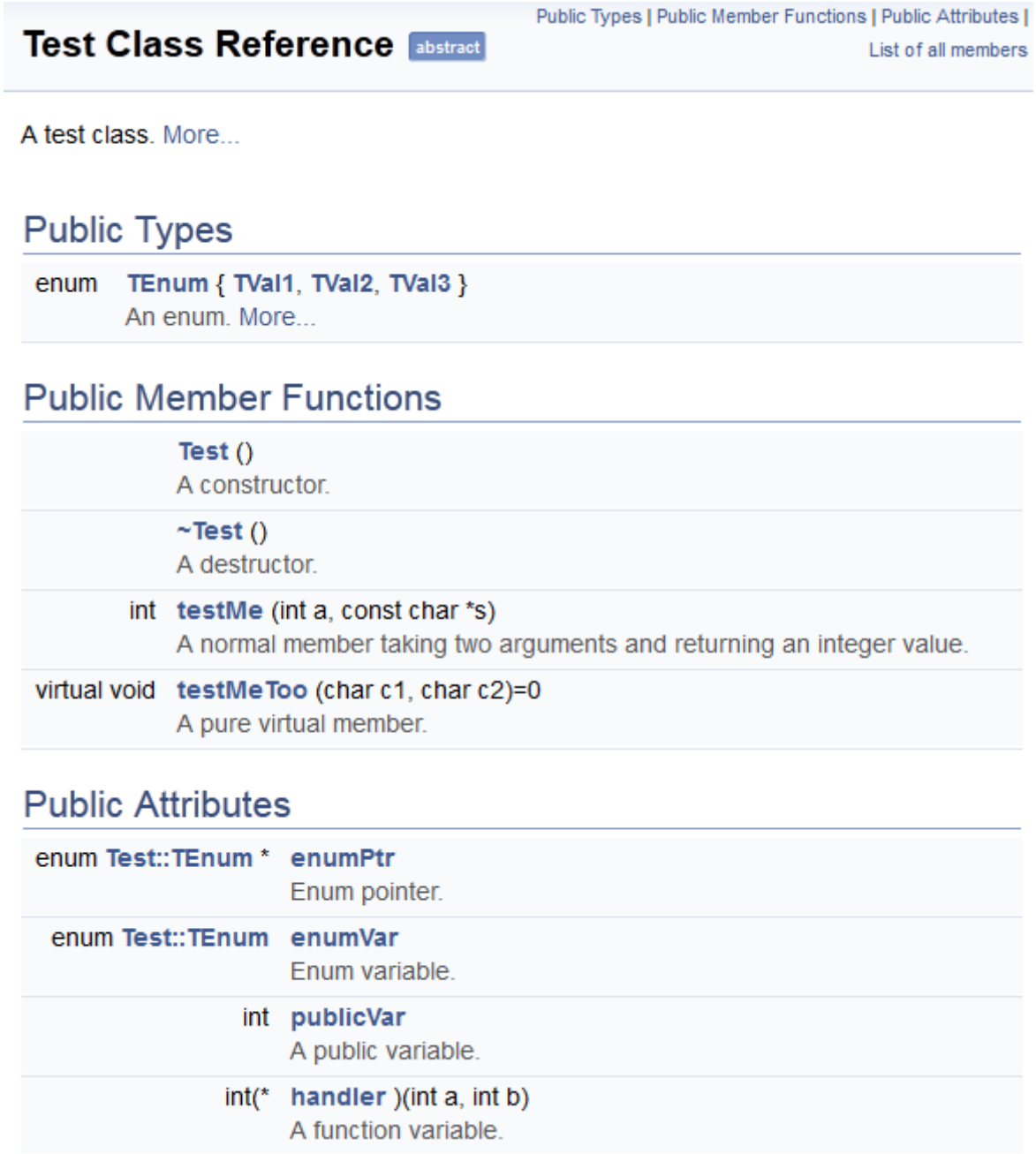
Beispiele für derartige Dokumentationssysteme sind Javadoc (für Java), Doxygen (C, C++, Objective-C, aber auch Python, Fortran und Java) und GTK-Doc (C). Einen interessanten Ansatz wählt Natural Docs (Perl, C# und Actionscript), das keine zusätzlichen Tags für eine Generierung einer

Dokumentation benötigt, sondern versucht aus den "natürlichen" Kommentaren des Entwicklers die Dokumentation zu erstellen.

```
1   //!  A test class.
2   /*!
3     A more elaborate class description.
4   */
5
6   class Test
7   {
8     public:
9
10      //! An enum.
11      /*! More detailed enum description. */
12      enum TEnum {
13                   TVal1, /*!< Enum value TVal1. */
14                   TVal2, /*!< Enum value TVal2. */
15                   TVal3  /*!< Enum value TVal3. */
16                 }
17           //! Enum pointer.
18           /*! Details. */
19           *enumPtr,
20           //! Enum variable.
21           /*! Details. */
22           enumVar;
23
24      //! A constructor.
25      /*!
26        A more elaborate description of the constructor.
27      */
28      Test();
29
30      //! A destructor.
31      /*!
32        A more elaborate description of the destructor.
33      */
34     ~Test();
35
36      //! A normal member taking two arguments and returning an integer value.
37      /*!
38        \param a an integer argument.
39        \param s a constant character pointer.
40        \return The test results
41        \sa Test(), ~Test(), testMeToo() and publicVar()
42      */
43      int testMe(int a,const char *s);
```

[Bild 13]: Quelltext mit Doxygen-Formatierung

Im oben angeführten Beispiel sieht man eine einfache Testklasse in C++ mit der Doxygen-Notation der Kommentare. Was vor allem bei einer so kleinen Klasse sofort auffällt, ist der deutlich schwerer zu lesende Quelltext. Dafür erhält man allerdings eine übersichtliche Dokumentation, wie dem nächsten Bild zu entnehmen ist:



Test Class Reference abstract

Public Types | Public Member Functions | Public Attributes | List of all members

A test class. More...

Public Types

enum	**TEnum** { **TVal1**, **TVal2**, **TVal3** } An enum. More...

Public Member Functions

	Test () A constructor.
	~Test () A destructor.
int	**testMe** (int a, const char *s) A normal member taking two arguments and returning an integer value.
virtual void	**testMeToo** (char c1, char c2)=0 A pure virtual member.

Public Attributes

enum **Test::TEnum** *	**enumPtr** Enum pointer.
enum **Test::TEnum**	**enumVar** Enum variable.
int	**publicVar** A public variable.
int(*	**handler**)(int a, int b) A function variable.

[Bild 14]: Ausgabe von Doxygen

Hier sieht man eine Ausgabe im html-Format. Man erhält beispielsweise Übersichten, aber auch detailierte Angaben zu einzelnen Klassen/Funktionen.

6. Zusammenfassung

Zusammenfassend lässt sich sagen, dass Kommentierung leider immer noch unterschätzt und daher oftmals vernachlässigt wird. Allerdings kann es die Arbeit an Software entscheidend vereinfachen. Zu beachten ist jedoch, dass Kommentare sinnvoll gewählt werden, da ansonsten die Kommentierung keine Vorteile mit sich bringt.

Außerdem hat sich gezeigt, dass Kommentierung sehr vielseitig genutzt werden kann. Die verschiedenen Möglichkeiten lassen sich zusätzlich problemlos auf andere Programmiersprachen übertragen.

In Projekten, an denen mehrere Entwickler mitarbeiten, ist Kommentierung außerdem unverzichtbar.

7. Quellen

- *http://de.wikipedia.org/wiki/Kommentar_%28Programmierung%29*
- *http://en.wikipedia.org/wiki/Comment_%28computer_programming%2*
- *http://queue.acm.org/detail.cfm?id=1053354*
- *http://de.wikipedia.org/wiki/Software-Dokumentationswerkzeug*
- Codebeispiele [1], [2], [3]: Eigenes Beispiel
- Codebeispiele [5], [7], [8]: Eigene Arbeit
- Codebeispiele [4] + [6]: GLib Version 2.26.1 (Windows), *glib-2.26.1**glib**gregex.c*
- Codebeispiele [9], [10], [11], [12]: GLib Version 2.26.1 (Windows), *glib-2.26.1**glib**gmain.c*
- Codebeispiele [13] *+* [14]: Doxygen Beispiel in C++, Qt Style
- Alle Codebeispiele wurden mit *QtCreator 2.2.0* noch einmal überarbeitet