Softwareentwicklung in der Wissenschaft

Code-Qualität

Johann Weging

4. Oktober 2011

Inhaltsverzeichnis

1 Einführung

Software hat in Forschungsbereichen, wie zum Beispiel der Meteorologie, einen hohen Stellenwert. Sie wird hier schon seit vielen Jahre zur Simulation von Klima und Wettersystemen eingesetzt. Einige dieser Modelle haben einen langen Entwicklungszeitraum hinter sich und werden heute immer noch eingesetzt. Um die Verständlichkeit und Wartbarkeit des Codes zu gewährleisten, ist die Qualität des Quellcodes von großer Bedeutung.

Wenn Softwareentwickler sich in ein Projekt einarbeiten, ist dies mit einem qualitativ hochwertigem Code einfacher. Der Entwickler, ist dadurch in der Lage, sich schneller in das Programm einzuarbeiten und damit Produktiv in das Projekt einbringen.

1.1 Was ist Code-Qualität?

Die Sicherstellung der Code-Qualität fördert die Verständlichkeit und Wartbarkeit des Quellcodes. Erreicht wird dies durch formale Sauberkeit und einer verständlicher Struktur.[1] Unter dem Begriff Code-Qualität verbergen sich eine Reihe von Teilzielen, die über die Qualität des Codes definiert ist. An erster Stelle ist eine verbesserte Lesbarkeit des Codes zu nennen, dies verbessert die Verständlichkeit des Codes. Die Verständlichkeit ist ein weiterer wesentlicher Bestandteil der Code-Qualität ist. Die Verständlichkeit eines Codes wird weiterhin durch ein einheitliche und konsistente Programmierung erreicht, die sich im ausführlichen und sauberen kommentieren und dokumentieren widerspiegelt. Auch wenn das kommentieren die Software nicht direkt beeinflussen, gehört dies doch zu der Code-Qualität.[2]

Ein weiteres Teilziele der Code-Qualität ist die bessere Wartbarkeit, hierdurch wird die Fehlersuche erleichtert. Der Code soll auch leicht zu ändern sein, dies ist wichtig für die Weiterentwicklung eines Programms. Damit soll die Korrektheit der bisherigen Funktionen bei der Weiterentwicklung gewährleistet werden und das Hinzufügen neuer Funktionen wird erleichtert.[1] Erreicht werden werde diese Ziele durch ein gut strukturiertes Projekt, formale Sauberkeit, aussagekräftige Benennung von Variablen, ein sauberes Design des Codes und nicht zu letzte einer gründlichen Dokumentation.

Weiterhin ist gründliches Testen der Software ein Punkt die Code-Qualität zu gewährleisten., allerdings wird auf dieses Thema hier nicht weiter eingegangen

1.2 Über diese Seminararbeit

Was wird in diesem Paper nicht erläutert?
Nicht erläutert wird Qualität im Bezug auf das Projektmanagement. Auch auf die Integration und Umsetzung von Code-Qualität im Kontext eines Kompletten Projektes wird nicht erläutert. Auch auf das Testen von Software wird nicht weiter eingegangen, da es in einem anderem Paper dieses Seminars von Florian Ehmke behandelt wird.[4]

Dieses Paper wird klären was sich hinter dem Begriff Code-Qualität verbigt und wird dabei die unterschiedlichen Teilbereiche anschneiden. Es wird an Hand einiger einfacher Beispiele gezeigt:
Wie ein Projekt korrekt strukturiert wird.
Wie sauberer Quellcode geschreiben wird.
Wie es möglich ist Code-Qualität in Zahlen auszudrücken.
Und wobei man bei der Dokumentation des Quellcodes zu achten hat.

2 Struktur des Projektes

Die Code-Qualität fängt schon mit der Struktur des Projektes an. Es ist darauf zu achten, dass eine passende Ordnerstruktur gewählt wird. Die Dateinamen sollten aussagekräftige sein und die Ordner für Übersicht sorgen. Dies ist Grundlage für eine verbesserte Orientierung in dem Projekt und eine klare Trennung der einzelnen Bereiche.

2.1 ecoham

Listing 1: Dateilistung von ecoham ohne Ordnerstruktur

```
[...]
NSriver.inc
NStime.inc
NStopo.inc
NSvelfield.inc
PDIndexL21.asc
River.dat
ZetaM2.dat
biogeo.f90
biogeo.lst
biogeo.o
dep.dat
eco4.f90
eco4.lst
eco4.o
eco4_atm_n_mon_2001.dat
eco4_atm_n_mon_2002.dat
eco4_atm_n_mon_2003.dat
eco4_atm_n_mon_2004.dat
eco4_flu1.com
eco4_flu1.read
eco4_flu2.com
eco4_flud.com
eco4_indh.dat
eco4_neigh.dat
eco4_neigh_sk.dat
[...]
```

Listing 1 zeigt eine Teildateiauflistung von ecoham. Es ist quasi nur aus der Dateiendung zu erkennen wozu die einzelnen Dateien dienen. Die Quellcode-, Input- und Outputdateien liegen alle in dem selben Verzeichnis. Dies hat mehrere Nachteile. Auf der einen Seite haben es neue Entwickler schwieriger sich in das Projekt ein zu arbeiten und sich zurecht zu finden, da das Fehlen einer klaren Struktur den Überblick erschwert. Dies kann dazu führen, dass es für den Entwickler nur möglich ist sich mit Hilfe ‚eines eingearbeiteten Entwicklers, in das Projekt ein zu arbeiten. Auf der anderen Seite wird die Wartbarkeit des Codes beeinträchtigt. Wird das Modell um einen Teil erweitert, kann es sein, dass neue Datei erstellt werden, bei den es nicht ersichtlich ist, welchem Zweck diese dienen.

Listing 2: Mögliche Projektstrucktur

```
/project
  /build
    main.o
  /doc
    README.txt
  /src
    main.c
  /model-input
    input.dat
  /model-output
    output.dat
```

Listing 2 zeigt eine von mehreren Möglichkeiten das Projekt zu strukturieren. In “build“ wird das compilierte Programm und alle zum compilieren nötigen Daten abgelegt. “model-input“ und “model-output“ ist für die Modelldaten zuständig. In “src“ liegen die Quellcode-Dateien, auch hier kann man noch weiter unterteilen und den Code in weitere Module einteilen. In “doc“ wird die Dokumentation abgelegt, hier können Paper, Informationen zum Modell und die Quellcode- Dokumentation abgelegt werden. Die Quellcode-Dokumentation wird später noch genauer untersucht.

3 Konsistente Programmierung

Ein wesentlicher Punkt der Code-Qualität ist konsistente und saubere Programmierung. Die Einhaltung einiger Regel und formaler Grundlagen fördert die Lesbarkeit und Verständlichkeit des Quellcodes. Im Folgenden wird gezeigt wo drauf bei der Programmierung geachtet werden sollte, um möglichst sauberen Code zu erzeugen.

Listing 3: Aufsummieren einer Liste (Dirty-Code C)

```
float suml(float* l,int c){
float s=0;
for(int i=0;i<c;++i){s+=l[i];}
return s;}
```

Listing 3 zeigt unsauber geschriebene C-Code. Es handelt sich hier bei um eine Funktion zum Aufsummieren einer Liste von Floats. Der Sinn der Funktion ist nicht auf den ersten Blick zu erkennen. Der Code ist sehr unlesbar geschrieben, und die Variablennamen sind kryptisch. Somit ist der Code auf lange Sicht nicht mehr einfach zu Pflegen.

Listing 4: Aufsummieren einer Liste mit sprechenden Variablen (C)

```
float sumUpList(float* list, int listElementCount)
{

  float sum = 0;

  for(int index = 0; index < listElementCount; ++index)
  {
    sum = sum + list[index];
  }

  return sum;
}
```

Listing 4 zeigt den selben Code, dieses mal allerdings sauber Formatiert und mit deutlichen Variablennamen. Es ist nun einfacher den Sinn der Funktion zu erkennen und die einzelnen Operationen und Befehle leichter auseinander zu halten. Allerdings fehlen noch Kommentare, um die Code-Qualität steigern.

Listing 5: Aufsummieren einer Liste mit Kommentaren (C)

```c
/**
* \author Johann Weging
* \brief This function sums up all elements in a list.
*
* This function sums up all elements in a list of floats and returns the sum
* as a float. It although needs the count of the elements in the list.
*
* \param [in] list a list of floats to sum up.
* \param [in] listElementCount a integer specify the count of the
* list elements.
* \param [out] sum a integer containing the sum.
*/
float sumUpList(float* list, int listElementCount)
{
	float sum = 0; // Set sum to zero

	/*Iterate over the list elements*/
	for(int index = 0; index < listElementCount; ++index)
	{
		/*Add the current element to the sum*/
		sum = sum + list[index];
	}

	return sum; // Return the sum
}
```

Listing 5 zeigt den Code nun mit Kommentaren. Der Funktionskopf wurde so kommentiert, dass mit Doxygen[11] eine automatische Dokumentation generiert werden kann, es wird später noch auf Kommentierung und Doxygen eingegangen. Es reicht nun aus, die Kommentare zu lesen um den Code zu verstehen. Das Verständnis des Programms ist nun nicht mehr nur von dem Code abhängig dem Entwickler, der sich in das Projekt einarbeitet. Was auffällig ist, das die Kommentare etwas das gleiche Volumen wie die Codezeilen einnehmen. Dies ist auch ein guter Richtwert um, eine gute Kommentarabdeckung zu erreichen.

Des Weiteren ist es ratsam sich an Styleguids zu orientieren. Für C ist ein gängiger Styleguid der “Indian Hill C Style and Coding Standards“.[5] Wenn man mit Fortran programmiert ist der “European Standards For Writing and Documenting Exchangeable Fortran 90 Code“[12] eine gute Richtlinie. Es ist nicht zwingend sich an einen Styleguid zu halten, allerdings sollte innerhalb eines Projekts der gleiche Programmierstiel gepflegt werden. Man kann zu einem Projekt Quellcode-Konventionen veröffentlichen, die jeder beteiligte Programmierer einzuhalten hat.

4 Analyse von Code-Qualität

Es gibt verschiedene Kennzahlen und Metriken um Code auf seine Qualität hin zu Analysieren und in konkreten Zahlen auszudrücken. Hiermit ist es möglich die Analyse zu automatisieren und in Tools einzubinden. Die Automatisierung macht es möglich die Analyse des Codes in den Entwicklungsprozess zu integrieren und so für eine bessere Qualität zu sorgen.[1]

4.1 Messwerte

Die Messwerte können direkt aus dem Quellcode erhoben und später zum berechnen der Metriken verwendet werden. Erst über die spätere Berechnung durch Metriken lässt sich eine Aussage über die Qualität des Codes treffen.[1]

Die Anzahl der Codezeilen kann man auf unterschiedlich Art und Weise erheben. Die erste Variante ist den gesamten Code zu betrachten, mit Codezeilen, Withespace und Kommentaren. Die zweite Variante ist die Kommentare weg zu lassen und nur noch den formatierten Code zu betrachten. Die dritte Variante besteht darin nur die relevanten Zeilen sind zu betrachten, also jene Zeilen, die nur Code enthalten ohne Withespace.[2]

Ein weiterer Messwert, der zumindest in der Welt der Objektorientierung interessant ist, ist die Anzahl der Klassen. Wird das Projekt größer, steigt auch die Anzahl der Klassen, allerdings benötigt man diesen Wert um Mittelwerte bilden zu können und das Projekt als ganzes oder nur bestimmte (kritische) Teile zu bewerten. Bei Software die nicht der Objektorientierung unterliegt, könnte man hier die Anzahl der Quellcode-Dateien anführen.[2]

Ein weiterer Messwert ist die Anzahl der Methoden oder Funktionen eines Programmes. Wichtig wird dies im Zusammenhang mit der Anzahl der Funktionen pro Datei, Methoden pro Klasse. Hier durch lassen sich sehr große Dateien erkennen, welche man wahrscheinlich besser in kleinere Einheiten aufteilt. Dies erzeugt einige Richtwerte. Eine Programmdatei sollte nicht mehr als 400 Zeilen umfassen. Eine Funktion sollte zwischen 4 und 40 Programm Zeilen haben.[2]

Unter anderem lässt sich auch die Anzahl der Felder oder globalen Variablen betrachten. Globale Variablen können von mehreren Stellen im Code verändert werden und wirken sich so auf Seiteneffekte in dem Programm aus. Dies ist eine potentielle Fehlerquelle. Je weniger ein Programm mit globalen Variablen arbeitet, je weniger Fehler treten auf.[2]

4.2 Softwaremetriken

Softwaremetriken sind Funktionen die dazu verwendet werden Eigenschaften von Software in Kennzahlen zu fassen. Durch diese Kennzahlen wird es möglich die Software objektiv zu bewerten.[1] Es gibt unterschiedliche Metriken, die auf die einzelnen Bereiche der Softwareentwicklung abzielen, wie Entwicklungsaufwand oder Ressourcenaufwand.[1] Im Rahmen dieses Seminars liegt der Bezug auf die Analyse des Quellcodes. Zur Analyse des Quellcodes werden hauptsächlich die Halstead-Metriken[10][9] und die McCabe-Metrik[2] verwendet. Die McCabe-Metrik wird auch zyklomatische Komplexität genannt. Im folgenden werden die beiden Metriken genauer erläutert und dessen Funktionsweise erklärt.

4.2.1 Halstead-Metriken

Listing 6: Minimal Beispielcode (C)

```
1 int x = 0;
2 x = x + 1;
```

Die Halstead-Metriken sind 1977[10] von Maurice Howard Halstead erfunden worden. Es handelt sich hierbei um eine Sammlung von Metriken, um durch verschiedene Kennzahlen Aussagen über die Qualität des Quellcodes zulassen. Es handelt sich hierbei um statische Metriken, die über den Quellcode erhoben werden und Sprach unabhängig sind.[1][2]

Für die Berechnung der Halstead-Metriken werden unterschiedliche Messwerte erhoben.

OP ist die Anzahl der Operanten $(+, -, *, =)$ und von Deklaren/ (int).

UOP ist die Anzahl einmaliger Operatoren. Jeder Operator wird nur ein mal gezählt, während bei OP jedes Vorkommen eines Operators gezählt wird.

OD ist die Anzahl der Operanden, wie Variablen, Strings und Zahlen.

UOD ist die Anzahl einmaliger Operanden.[9,10]

Listing 6 zeigt ein kleines Stück Quellcode, dieser Code enthält folgende Operatoren: $int, =, ;, =, +, ;$, damit enthält der Quellcode 6 Operatoren und 4 einmalige Operatoren.

$$OP = 6$$
$$UOP = 4$$

Des weiteren enthält der Code folgende Operanden: $x, 0, x, x, 1$ und die einmaligen Operanden: $x, 0, 1$. Somit sind 5 Operanden und 3 einmalige Operanden vorhanden.

$$OD = 5$$
$$UOD = 3$$

Mit diesen erhobenen Werten lassen sich nun die Halstead-Metriken berechnen. Die Halstead-Länge (LTH Halstead-Length) gibt die Länge des Quellcodes wider.

$$LTH = OP + OD$$
$$5 + 6 = 11$$

Eine weitere Metrik ist das Halstead-Vokabular (VOC Halstead-Vocabulary) es lässt eine Aussage über die Menge der verwendeten Symbole zu.

$$VOC = UOP + UOD$$
$$3 + 4 = 7$$

Die Halstead-Schwierigkeit (DIF Halstead-Difficulty) soll ein Anhaltspunkt sein wie schwer der Quellcode zu verstehen ist. Ein breiteres Vokabular an Operanden macht den Code leichter verständlich.

$$DIF = \frac{UOP}{2} \cdot \frac{OD}{UOD} \quad \frac{3}{2} \cdot \frac{6}{4} = 2.25$$

Das Halstead-Volumen (VOL Halstead-Volume) gibt die Menge der Information an, die der Programmiere verarbeiten muss, um den Code zu verstehen. Dabei ist die entscheidende Größe die Halstead-Länge.

$$VOL = LTH \cdot \log_2(VOC)$$
$$11 \cdot \log_2(7) = 30.88$$

Der Halstead-Aufwand (EFF Halstead-Effort) gibt den nötigen geistigen Aufwand an, den der Programmier aufbringen muss um den Code zu verstehen.

$$EFF = DIF \cdot VOL$$
$$2.25 \cdot 30.88 = 69.48$$

Die Metrik der Halstead-Fehler (BUG Halstead-Bugs) soll angeben wie viele Fehler sich möglicherweise in dem Quellcode befinden.

$$BUG = \frac{VOL}{3000}$$
$$\frac{30.88}{3000} = 0.01$$

[9,10]

4.2.2 Zyklomatische Komplexität

Listing 7: Berechnung des Betrages (C)

```
INTEGER :: Value

READ(*,*) Value !read the Value

IF (MOD(Value, 2) == 0) THEN !modulo of Value = 0
WRITE(*,*) Value , 'is even'

ELSE !modulo of Value != 0
WRITE(*,*) Value, 'is odd'
END IF

IF(Value >= 0) THEN !Value is positive
WRITE(*,*) Value, 'is positive'

ELSE ! Value is negative
WRITE(*,*) Value, 'is negative'
END IF
```

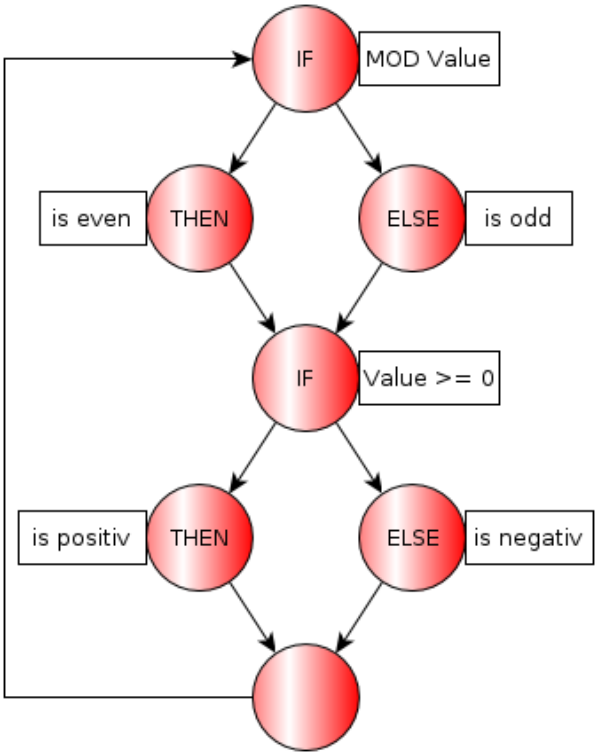


Abbildung 1: Betragsberechnung als Kontrollflußgraph

Über die zyklomatische Komplexität lässt sich die Komplexität eines Programms bestimmen. Diese Metrik wird über den Kontrollflußgraph des Programms erstellt. If-Verzweigungen werden in dem Graphen als Verzweigungen dargestellt und Schleifen als Zyklen. Der Graph in Abbildung 1 zeigt den Beispielcode aus Listing 7 als Kontrollflußgraph. Für die Berechnung wird dem Graph zusätzlich eine Kante vom Endknoten zum Startknoten hinzugefügt.[2,16]

Die zyklomatische Komplexität (M) wird über die Anzahl der Kanten (E Edges), und die Anzahl der Knoten (N Nodes) und den zusammenhängenden Komponenten definiert.

$$M = E - N + 2P$$

Bezogen auf Abbildung 1 erhält man folgende Werte.

$$E = 9$$
$$N = 7$$
$$P = 1$$

$$9 - 7 + 2 \cdot 1 = 4$$

[2,16]

4.2.3 Wartbarkeitsindex

Der Wartbarkeitsindex ist ein Wert zwischen 0 und 100, der angibt wie leicht oder wie schwer der Quellcode zu warten ist.[2] Die Wartbarkeit des Codes ist besser, je höher der Wert ist. Der Wartbarkeitsindex setzt sich zusammen aus der zyklomatische Komplexität, der Anzahl der Codezeilen und dem Halstead-Volumen. Der Wartbarkeitsindex arbeitet mit dem Arithmetischenmittel einiger Werte. Das Mittel wird über die einzelnen Softwaremodule erhoben. Bei nicht objektorientierten Sprachen kann man den Durchschnitt über die einzelnen Dateien berechnen. Der Wartbarkeitsindex (MI Maintainability Index) wird aus dem Wartbarkeitsindex ohne Kommentaren ($MIwoc$: MI without comments) und dem Wartbarkeitsindex mit Kommentaren ($MIwc$: MI with comments).

$$MI = MIwoc + MIwc$$

Der Wartbarkeitsindex ohne Kommentare setzt sich zusammen aus dem durchschnittlichem Halstead-Volumen ($avgVOC$), der durchschnittlichen zyklomatischen Komplexität ($avgM$) und den durchschnittlichen Codezeilen ohne Kommentaren ($avgLOC$).

$$MIwoc = 171 - 5.2 \cdot ln(aveVOL) - 0.23 \cdot aveM - 16.2 \cdot ln(aveLOC)$$

Der Wartbarkeitsindex mit Kommentaren wird über den prozentualen Anteil an Kommentarzeilen zu Codezeilen ($perCM$) berechnet.[2]

$$MIwc = 50 \cdot sin(2.4 \cdot perCM)$$

Die verwendeten Konstanten in dem Wartbarkeitsindex sind auf den ersten Blick nicht nachvollziehbar, die Berechnungen wurden über mehrere Jahre angepasst.

Eine Weitere Metrik die noch erwähnenswert ist, ist die Codeänderungsmetrik. Diese gibt Aufschluss wie stark der Code in welchen Bereichen geändert wurde. Für Entwickler ist dies hilfreich um geänderte Bereiche noch ein mal auf Fehler und Inkonsistenz zu überprüfen.

4.3 CCCC

CCCC[17] ist ein Konsolenprogramm zum automatischen Berechnen von Metriken. Das Tool unterstützt Java und C++. Wendet man CCCC auf den Code aus Abbildung **??** erhält man folgende Ausgabe.

Tabelle 1: CCCC Ausgabe

Metric	TAG	Overall	Per Module
Number of modules	NOM	1	-
Lines of Code	LOC	13	13.000
McCabe's Cyclomatic Number	MVG	3	3.000
Lines of Comment	COM	16	16.000

Natürlich ist das Berechnen der Metriken über so einen kleines Programm nicht repräsentativ. Was allerdings auffällt ist, dass die Code- und Kommentarzeilen etwa den gleichen Umfang haben.

Es gibt noch weitere Tools, die es ermöglichen Programmcode zu Analysieren[6]. Wichtig bei allen Tools ist es, diese immer wieder auf den Entwicklungsprozesses anzuwenden, um einen ständig sauberen Code zu gewährleisten.

5 Dokumentations Qualität

Die Softwaredokumentation soll zum Verständnis der Software beitragen und den Anwender über den Programmcode informieren, so dass dieser schnell in der Lage ist, die ihm gestellten Aufgaben zu erfüllen.

Der Anwender ist bei Wissenschaftlichersoftware häufig auch der Entwickler, so umfasst die Dokumentation auch die Quellcode-Dokumentation. Man kann bei der Dokumentationsqualität zwischen zwei Wesentlichen Bereichen unterscheiden.

5.1 Formale Qualität

Neben offensichtlichen formalen Merkmalen der Dokumentation wie Rechtschreibung und Sauberkeit, ist vor allem auf eine konsistente Dokumentation zu achten. Werden verschiedener Teile der Software unterschiedlich dokumentiert, kann dies den Anwender verwirren.

5.2 Inhaltliche Qualität

Bei dem Inhalt ist vor allem auf die Korrektheit und Vollständigkeit der Dokumentation achten. Die Korrektheit kann durch Änderungen in der Software beeinträchtigt werden. Daher ist immer darauf zu achten, dass mit Änderung an der Software auch die Quellcodedokumentation überarbeitet wird. Genau so müssen neue Teile der Software dokumentiert werden. Fehlende Dokumentation kann zu zwei größeren Problemen führen. Erstens, der Anwender wird nicht über das Feature informiert und verwendet es daher nicht.

Oder zweitens, der Anwender muss übermäßig viel Zeit aufwenden um sich in den Code ein zu lesen und verliert so Zeit sich Produktiv einzubringen.

Gegen die oben genannten Probleme ist eine von der Entwicklung getrennte Qualitätsprüfung der Dokumentation einzusetzen. Diese Prüfung zielt auf die Vollständigkeit und Korrektheit der Dokumentation ab. Dies lässt sich auch in teilen automatisieren. Es ist möglich den Kommentaranteil in einzelnen Bereichen des Quellcodes analysieren und den Entwickler auf nur schwach kommentierte Bereiche hinweisen.

5.3 Doxygen

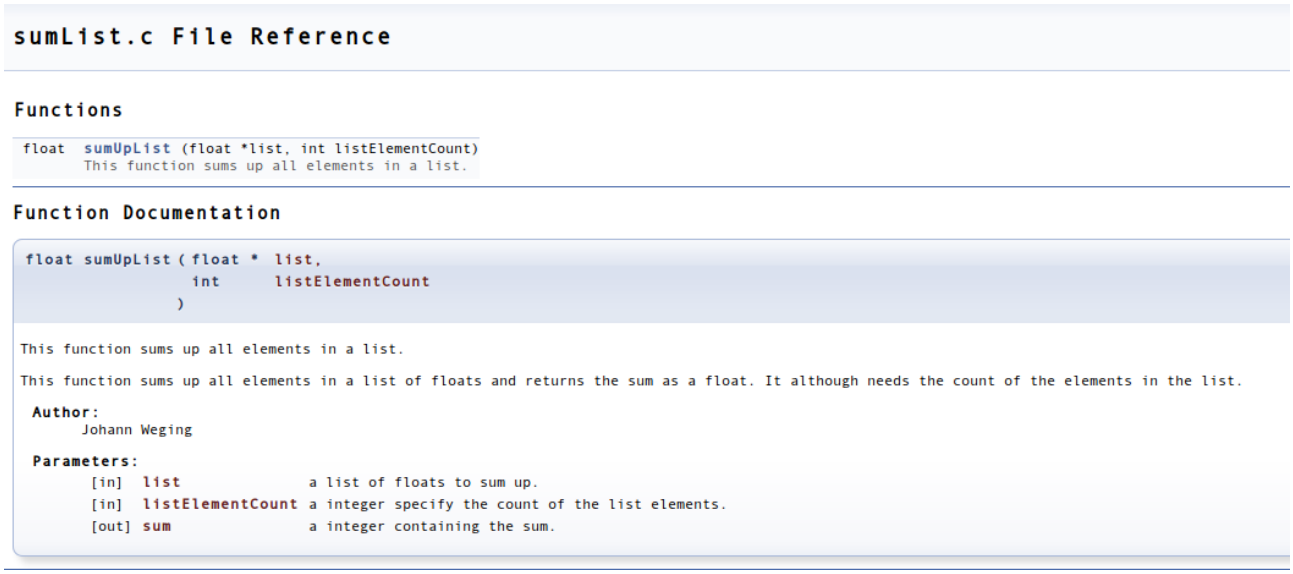
Doxygen ist ein Tool um Softwaredokumentationen automatisch zu generieren. Es unterstützt mehrere Sprachen unter anderem Fortan und C. Damit Doxygen die Dokumentation automatisch generieren kann, muss der Anwender einen bestimmten Kommentarstiel verwenden.

Listing 8: Funktionskommentar der Listenfunktion (C).

```
1
2  /**
3  *\author Johann Weging
4  *\brief This function sums up all elements in a list.
5  *
6  * This function sums up all elements in a list of floats and returns the sum
7  * as a float. It although needs the count of the elements in the list.
8  *
9  *\param [in] list a list of floats to sum up.
10 *\param [in] listElementCount a integer specify the count of the
11 * list elements.
12 *\param [out] sum a float containing the sum.
13 */
14 float sumUpList(float* list, int listElementCount)
15 {
16         ...
17 }
```

Listing 9: Funktionskommentar der Listenfunktion (Fortran).

```
1  !>
2  !!@author Johann Weging
3  !!@brief This function sums up all elements in a list.
4  !!
5  !! This function sums up all elements in a list of floats and returns the sum
6  !! as a float. It although needs the count of the elements in the list.
7  !!
8  !!@param [in] list a list of floats to sum up.
9  !!@param [in] listElementCount a integer specify the count of the
10 !! list elements.
11 !!@param [out] sum a float containing the sum.
12 !!
13 subroutine sumUpList( list, listElementCount, sum)
```



sumList.c File Reference

Functions

float sumUpList (float *list, int listElementCount)
This function sums up all elements in a list.

Function Documentation

float sumUpList (float * list,
int listElementCount
)

This function sums up all elements in a list.

This function sums up all elements in a list of floats and returns the sum as a float. It although needs the count of the elements in the list.

Author:
Johann Weging

Parameters:
[in] list a list of floats to sum up.
[in] listElementCount a integer specify the count of the list elements.
[out] sum a integer containing the sum.

Abbildung 2: Die von Doxygen erzeugte Dokumentation in HTML.

Listing 8 und 9 zeigen ein Stück Quellcode in C und Fortran, der so kommentiert wurde das Doxygen die Dokumentation erzeugen kann. Über der Funktion befindet sich der Funktionskopf. Eingeleitet wird der Doxygen-Kommentarblock durch "/**" oder "!>", Doxygen liest nun den Kommentarblock und kann mit Hilfe von Schlüsselwörtern die Dokumentation erzeugen. Durch "\/ @author" wird der Autor der Funktion angegeben. "\/ @brief" wird eine kurze Beschreibung der Funktion angegeben, die nur kurz den Zweck der Funktion beschreibt. Da nach folgt eine längere Beschreibung der Funktion, hier kann unter anderem auch auf die genauere Funktionsweise eingegangen werden. Zuletzt werden noch mit "\/ @param" die Übergabeparameter und dessen zweck angegeben. In den eckigen Klammern wird häufig angegeben, ob es ich um einen Eingabe- oder Ausgabeparameter handelt.

Da Doxygen mehre Programmiersprachen unterstützt, gibt es unterschiedliche Möglichkeiten den Code so zu kommentieren, dass die Kommentare von Doxygen gelesen werden können.

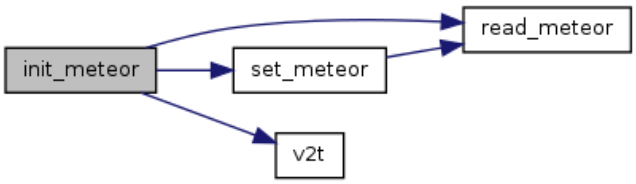


Abbildung 3: Funktionen die von init_meteor aufgeruhfen werden.

Unter anderem ist Doxygen in der Lage, automatisch Aufrufgraphen der Programme zu erstellen, Abbildung 3 zeigt einen Teilgraphen von ecoham. Dies ist hilfreich, um den Ablauf eines Programms zu verstehen. Es kann aber auch helfen, Zyklen oder ähnliche Fehlerquellen aufzudecken.

6 Zusammenfassung

Abschließend ist festzustellen das besonders bei Software bei der mit einem langem Entwicklungs- und Lebenszyklus zu rechnen ist, auf die Code-Qualität geachtet werden muss. Es ist zwar ratsam, sich an einen offiziellen Styleguid zu halten, wichtiger ist allerdings die Umsetzungen von Richtlinien, damit die Software einheitlich gestaltet wird. Hierdurch wird die Konsistenz der Software verbessert und somit Fehlerquellen zu minimieren. Zu dem sollten Tools mit statischer Codanalyse in den Entwicklungsprozess integriert werden und der Code daraufhin optimiert werden. Eine umfangreiche Dokumentation ist besonders in dynamischen und sich häufig verändernden Teams wichtig um die Einarbeitungszeit so gering wie möglich zu halten. Tool wie zum Beispiel Doxygen helfen hier um eine funktionale Dokumentation zu erzeugen, die zu dem Verständnis des Programms beiträgt. Abschließen ist noch zu sagen, es gibt nicht den Fall, das ein Code zu wenig Kommentare enthält.

7 Quellen

1. Donis Marschall & John Bruno, Soid Code, Microsoft Press Deutschland, 2009
2. Komplexität und Qualität von Software, In: MSCoder 1/2007, Seite 36-43
3. IEEE 754-2008
4. http://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2010_2011/siw-1011-ehmke-tests-ausarbeitung.pdf
5. http://www.cs.arizona.edu/ mccann/cstyle.html
6. http://metrics.sourceforge.net/
7. http://software-kommunikation.net/qm_qualitaet.html
8. http://msdn.microsoft.com/en-us/library/ms182021%28v=vs.90
9. http://www.virtualmachinery.com/sidebar2.htm
10. http://en.wikipedia.org/wiki/ Halstead_complexity_measures
11. http://www.stack.nl/ dimitri/doxygen/
12. http://research.metoffice.gov.uk/research/nwp/numerical/ fortran90/f90_standards.html
13. https://srv.rz.uni- bayreuth.de/lehre/fortran90/vorlesung/V10/V10.html
14. http://en.wikipedia.org/wiki/ Software_quality#Source_code_quality
15. T. Panas, D. Quinlan, R. Vuduc, Tool Support for Inspecting the Code Quality of HPC Applications 2007
16. http://en.wikipedia.org/wiki/Cyclomatic_complexity