Software

Komplexität und Qualität von Software

Wie bestimmt man die Komplexität von Quellcode und welchen Einfluss hat sie auf die Wartbarkeit? Die Softwarequalität und die Wartbarkeit von Applikationen hängt von der Codekomplexität ab. Eine angemessene Komplexität vereinfacht Softwaretests und Wartung.

In diesem Artikel erfahren Sie...
- wie die Komplexität einer Software gemessen wird;
- den Einfluss der Komplexität auf die Qualität;
- die traditionellen Softwaremetriken.

Was Sie vorher wissen sollten...
- Verständnis für die bei der Programmierung verwendeten Begriffe;
- Programmierkenntnisse, insbesondere der Programmiersprache C.

Schwierigkeitsgrad

Die Komplexität einer Software hat einen direkten Einfluss auf deren mögliche Nutzungsdauer, ihre Fehlerträchtigkeit, Testbarkeit und Wartbarkeit. Qualität und Kosten der Erstellung und Instandhaltung von Applikationen hängen entscheidend von der Code-Komplexität ab. Fehleranzahl und Robustheit eines Codes stehen meist in einem engen Zusammenhang zur Softwarekomplexität.

Komplexe Software ist schwierig zu testen und zieht höhere Kosten beim Softwaretest und der Fehlerbehebung nach sich. Selbst nach dem Softwaretest hat ein zu komplexer Code in der Regel noch mehr Bugs als ein Programm, dessen Struktur relativ einfach gehalten wurde.

Um die Qualität zu garantieren und die Kosten für den Test und die Wartung der Software möglichst gering zu halten, ist es ratsam, die Softwarekomplexität bereits während des Entwicklungsprozesses zu messen. Werden die empfohlenen Werte überschritten, kann man frühzeitig korrigierende Maßnahmen ergreifen. Um die Qualität eines Programms zu messen, bedient man sich so genannter Softwaremetriken.

Der IEEE-Standard 1061 aus dem Jahr 1992 definiert die Metriken einer Software als eine Größe, welche die Qualität der einzelnen Programmteile in einem numerischen Wert abbildet. Man unterteilt die Metriken generell wie folgt:

- Metriken zur Messung des Software-Entwicklungsprozesses;
- Metriken zur Messung der verfügbaren Ressourcen;
- und Metriken zur Evaluation des Softwareproduktes.

Die Produktmetriken messen die Qualität der Software. Bei diesen Metriken unterscheidet man zwischen traditionellen und objektorientierten Metriken. Objektorientierte Metriken berücksichtigen die Verhältnisse der einzelnen Elemente (Klassen, Methoden) eines Programms untereinander. Die traditionellen Metriken werden in zwei Gruppen eingeteilt: Metriken zur Messung der Programmgröße und dessen Komplexität, sowie Metriken zur Messung der Programmstruktur.

Die wichtigsten Metriken zur Messung von Programmgröße und dessen Komplexität sind Zeilenmetriken und Halstead-Metriken. Die McCabe Cyclomatic Number ist eine Metrik für die Messung der Programmstruktur. Dieser Artikel beschreibt die drei oben genannte traditionellen Metriken sowie den daraus abgeleiteten Wartbarkeitsindex der Software. Die verschiedenen Metriken und deren empfohlenen Werte werden anhand des folgenden C-Programms illustriert. Beim Beispiel handelt es sich um ein Programm zur Berechnung eines mathematischen Ausdrucks der Form *Wert 1* - *Operator* - *Wert 2*:

Die Zeilenmetriken

Die Messung der Codezeilen (LOC, *Lines of Code)* ist die gebräuchlichste Messung für die Quantifizierung der Komplexität einer Software. Diese Metrik ist einfach, leicht zu messen und sehr verständlich. Die Messung lässt die *Intelligenz* und den Aufbau des Codes allerdings unberücksichtigt. Man unterscheidet die folgenden Zeilenmetriken:

- LOCphy: Anzahl der physikalischen Zeilen (Number of physical lines);
- LOCpro: Anzahl der Programmzeilen (Number of program lines): Deklarationen, Definitionen, Direktiven und Code;
- LOCcom: Anzahl der Zeilen mit Kommentaren (Number of commented lines);
- LOCbl: Anzahl der Leerzeilen (blank lines), Hinweis: Leerzeilen innerhalb eines Kommentarblocks werden als Kommentarzeile gezählt.

Die Messung von LOCphy ergibt für unsere Beispielfunktion eval1 den Wert von 72. Die Funktion enthält 57 Programmzeilen (LOCpro), 7 Leerzeilen (LOCbl) sowie 11 Zeilen mit Kommenten (LOCcom). Die Summe ist größer als LOCphy, da einige Zeilen sowohl Programmteile als auch Kommentare enthalten.

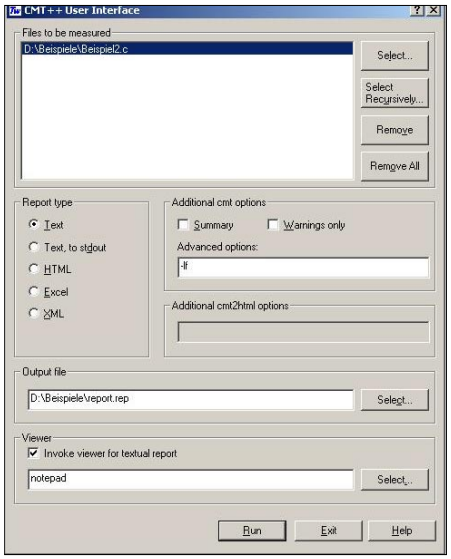

Abbildung 1. GUI eines Komplexitätsmesstools

Welche Werte sind akzeptabel?

Die *Länge einer Funktion* sollte zwischen 4 und 40 Programmzeilen liegen. Eine Funktionsdefinition beinhaltet mindestens einen Prototyp, eine Zeile des Codes und ein Paar Klammern, also mindestens 4 Zeilen.

Eine Funktion mit mehr als 40 Programmzeilen implementiert wahrscheinlich mehrere Funktionen. Hierbei sind Funktionen, die eine switch-Anweisung mit mehreren case-Anweisungen beinhalten. Die Funktion eval1 unseres Beispiel ist mit einem LOCphy von 72 zu lang. Die Zerlegung dieser Funktionen in mehrere kleinere Funktionen würde deren Lesbarkeit erhöhen. Die Anzahl der physikalischen Zeilen der anderen Funktionen des Programms liegt unter 40, also innerhalb des empfohlenen Grenzwertes.

Die Länge einer Datei sollte 4 bis 400 Programmzeilen sein. Die kleinste Einheit, die sinnvoll eine Datei füllen kann, ist eine Funktion und die minimale Länge einer Funktion beträgt 4 Zeilen. Dateien, die länger als 400 Programmzeilen sind (10-100 Funktionen) sind meistens zu lang, um als Ganzes verstanden zu werden.

Mindestens 30 Prozent und maximal 75 Prozent einer Datei sollten Kommentare sein. Falls Kom-

Listing 1. Beispielprogramm

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void extract(char* zeichenkette1, char* zeichenkette2, int
                    anfang, int nb);
int eval1(char * ch)
{
   int i;
   int wert1, wert2;
   int lgval2;
   char *val1, *val2;
   char operation;
   int resultat;
   /* Suche des Operators und seiner Position */
   for( i=0 ; *(ch+i) !='+' && *(ch+i) !='-' && *(ch+i) !='*' &&
                    *(ch+i) !='/' && *(ch+i) !='\0'; i++)
   {
   }
   /* Fehlerbehandlung */
   if(i==0) /* der erste Operand fehlt */
   {
      printf("Error: kein <wert1>");
      exit(0);
   }
   else if(i==strlen(ch)-1) /* der zweite Operand fehlt */
   {
      printf("Error: kein <wert2>");
      exit(0);
   }
   else if(i==strlen(ch)) /* kein Operator vorhanden */
   {
      printf("Error: kein <operator>");
      exit(0);
   }
   /* Zeichenkette fuer den ersten Operanden */
   val1=(char*) malloc((i+1)*sizeof(char));
   extract(ch,val1,0,i);
   /* Umwandlung der Zeichenkette */
   sscanf(val1,"%d",& wert1);
   /* Erfassung des Operators */
   operation=*(ch+i);
   /* Zeichenkette fuer den zweiten Operanden*/
   lgval2=strlen(ch)-(i+1);
   val2=(char*) malloc((lgval2+1)*sizeof(char));
   extract(ch,val2,i+1,lgval2);
   /* Umwandlung der Zeichenkette */
   sscanf(val2,"%d",&wert2);
   /* Berechnung */
   switch(operation)
   {
      case '+':
         resultat=wert1+wert2;
         break;
      case '-':
         resultat=wert1-wert2;
         break;
      case '*':
         resultat=wert1*wert2;
         break;
      case '/':
         if(wert2 != 0)
         resultat=wert1/wert2;
         else
         {
          resultat=0;
          printf("Error: Division durch 0 nicht moeglich");
          exit(0);
         }
   }
   return resultat;
}
/* Funktion zum Extrahieren einer Unterzeichenkette
   von zeichenkette1 in zeichenkette2 */
void extract(char* zeichenkette1, char* zeichenkette2,
      int anfang, int nb)
{
   int i;
   zeichenkette1= zeichenkette1+anfang;
   i=0;
   while(i<nb)
   {
      * zeichenkette2=* zeichenkette1;
      zeichenkette1++;
      zeichenkette2++;
      i++;
   }
   * zeichenkette2='\n';
}
int main(int argc, char** argv)
{
   int res;
   if(argc!=2)
   {
      printf("Error, Nutzung des Programms : eval1
                     <expression>");
   }
   else
   {
      res=eval1(argv[1]);
      printf("Das Ergebnis der Berechnung ist : %d",res);
   }
}
```

Listing 2. Ausgaben aller Metriken

```
Metrics for eval1()
Operator                                    Frequency
--------                                    ---------
!=                                                  6
&                                                   2
&&                                                  4
()                                                 31
*                                                  14
+                                                  11
++                                                  1
,                                                  12
-                                                   3
/                                                   1
;                                                  33
=                                                  10
==                                                  3
break                                               3
case ...:                                           4
else                                                3
for()                                               1
if()                                                4
return                                              1
sizeof                                              2
switch()                                            1
{}                                                  7

Operand                                     Frequency
-------                                     ---------
"%d"                                                2
"Error: Division durch 0 nicht moeglich"            1
"Error: kein <operator>"                            1
"Error: kein <wert1>"                               1
"Error: kein <wert2>"                               1
'*'                                                 2
'+'                                                 2
'-'                                                 2
'/'                                                 2
'\0'                                                1
0                                                   9
1                                                   5
ch                                                 12
char                                                7
eval1                                               1
exit                                                4
extract                                             2
i                                                  16
int                                                 5
lgval2                                              4
malloc                                              2
operation                                           3
printf                                              4
resultat                                            7
sscanf                                              2
strlen                                              3
val1                                                4
val2                                                4
wert1                                               6
wert2                                               7

Metrics                                     Value
-------                                     -----
Cyclomatic number          (v(G))                  14

Number of physical lines   (LOCphy)                72
Number of program lines    (LOCpro)                57
Number of blank lines      (LOCbl)                  7
Number of commented lines  (LOCcom)                11

Program length             (N)                    279
Number of operators        (N1)                   157
Number of operands         (N2)                   122
Vocabulary size            (n)                     52
Number of unique operators (n1)                    22
Number of unique operands  (n2)                    30
Program volume             (V)                1590.423

Number of delivered bugs   (B)                   0.572
Difficulty level           (D)                  44.733
Effort to implement        (E)               71144.908
Program level              (L)                   0.022
Implementation time        (T)                3952.495

Max nesting depth          (MaxND)                 3

Maintainability Index w.o.c(MIwoc)              60.165
MI comment weight          (MIcw)               28.460
MI with comments           (MI)                 88.625

-----------------------------------------------------------
Metrics for extract()

Operator                                    Frequency
--------                                    ---------
()                                                  1
*                                                   5
+                                                   1
++                                                  3
,                                                   3
;                                                   8
<                                                   1
=                                                   4
while()                                             1
{}                                                  2

Operand                                     Frequency
-------                                     ---------
'\n'                                                1
0                                                   1
anfang                                              2
char                                                2
extract                                             1
i                                                   4
int                                                 3
nb                                                  2
void                                                1
zeichenkette1                                       5
zeichenkette2                                       4

Metrics                                     Value
-------                                     -----
Cyclomatic number          (v(G))                   2
```

Listing 2. Ausgaben aller Metriken

```
Number of physical lines    (LOCphy)          17
Number of program lines     (LOCpro)          14
Number of blank lines       (LOCbl)            2
Number of commented lines   (LOCcom)           1

Program length              (N)               55
Number of operators         (N1)              29
Number of operands          (N2)              26
Vocabulary size             (n)               21
Number of unique operators  (n1)              10
Number of unique operands   (n2)              11
Program volume              (V)           241.577

Number of delivered bugs    (B)             0.067
Difficulty level            (D)            11.818
Effort to implement         (E)          2855.006
Program level               (L)             0.085
Implementation time         (T)           158.611

Max nesting depth           (MaxND)            2

Maintainability Index w.o.c(MIwoc)         96.109
MI comment weight           (MIcw)         18.348
MI with comments            (MI)          114.456

Metrics for main()

Operator                                   Frequency
--------                                   ---------
!=                                                 1
()                                                 4
*                                                  2
,                                                  2
;                                                  4
=                                                  1
[]                                                 1
else                                               1
if()                                               1
{}                                                 3

Operand                                    Frequency
-------                                    ---------
"Das Ergebnis der Berechnung ist : %d"             1
"Error, Nutzung des Programms : eval1 <expression>"
                                                   1
1                                                  1
2                                                  1
argc                                               2
argv                                               2
char                                               1
eval1                                              1
int                                                3
main                                               1
printf                                             2
res                                                3

Metrics                                    Value
-------                                    -----
Cyclomatic number           (v(G))             2

Number of physical lines    (LOCphy)          14
Number of program lines     (LOCpro)          13
Number of blank lines       (LOCbl)            1
Number of commented lines   (LOCcom)           0

Program length              (N)               39
Number of operators         (N1)              20
Number of operands          (N2)              19
Vocabulary size             (n)               22
Number of unique operators  (n1)              10
Number of unique operands   (n2)              12
Program volume              (V)           173.918

Number of delivered bugs    (B)             0.041
Difficulty level            (D)             7.917
Effort to implement         (E)          1376.850
Program level               (L)             0.126
Implementation time         (T)            76.492

Max nesting depth           (MaxND)            2

Maintainability Index w.o.c(MIwoc)         100.963
MI comment weight           (MIcw)          0.000
MI with comments            (MI)           100.963

-------------------------------------------------------

Beispiel2.c
Operator                                   Frequency
--------                                   ---------
!=                                                 7
#                                                  3
&                                                  2
&&                                                 4
()                                                37
*                                                 23
+                                                 12
++                                                 4
,                                                 20
-                                                  3
/                                                  1
;                                                 46
<                                                  1
=                                                 15
==                                                 3
[]                                                 1
break                                              3
case ...:                                          4
else                                               4
for()                                              1
if()                                               5
return                                             1
sizeof                                             2
switch()                                           1
while()                                            1
{}                                                12

Operand                                    Frequency
-------                                    ---------
"%d"                                               2
"Das Ergebnis der Berechnung ist : %d"             1
"Error, Nutzung des Programms : eval1 <expression>"
```

Listing 2. Ausgaben aller Metriken

```
                                                1
"Error: Division durch 0 nicht moeglich"        1
"Error: kein <operator>"                        1
"Error: kein <wert1>"                           1
"Error: kein <wert2>"                           1
'*'                                             2
'+'                                             2
'-'                                             2
'/'                                             2
'\0'                                            1
'\n'                                            1
0                                              10
1                                               6
2                                               1
anfang                                          3
argc                                            2
argv                                            2
ch                                             12
char                                           12
eval1                                           2
exit                                            4
extract                                         4
i                                              20
include                                         3
int                                            13
lgval2                                          4
main                                            1
malloc                                          2
nb                                              3
operation                                       3
printf                                          6
res                                             3
resultat                                        7
sscanf                                          2
strlen                                          3
val1                                            4
val2                                            4

void                                            2
wert1                                           6
wert2                                           7
zeichenkette1                                   6
zeichenkette2                                   5

Metrics                                     Value
-------                                     -----
Cyclomatic number          (v(G))              16
Max cyclomatic number                          14
Average cyclomatic number                       6

Number of physical lines   (LOCphy)            111
Number of program lines    (LOCpro)             88
Number of blank lines      (LOCbl)              14
Number of commented lines  (LOCcom)             12

Program length             (N)                396
Number of operators        (N1)               216
Number of operands         (N2)               180
Vocabulary size            (n)                 70
Number of unique operators (n1)                26
Number of unique operands  (n2)                44
Program volume             (V)           2427.196

Number of delivered bugs   (B)              0.851
Difficulty level           (D)             53.182
Effort to implement        (E)         129082.700
Program level              (L)              0.019
Implementation time        (T)            7171.261

Max nesting depth          (MaxND)             3

Maintainability Index w.o.c(MIwoc)         76.458
MI comment weight          (MIcw)          24.381
MI with comments           (MI)           100.839
```

mentare weniger als ein Drittel einer Datei ausmachen, ist die Datei entweder sehr trivial oder schlecht kommentiert. Falls mehr als 75 Prozent einer Datei Kommentare sind, ist die Datei kein Programm, sondern ein Dokument. In einer gut dokumentierten header-Datei kann der Anteil der Kommentare manchmal 75 Prozent überschreiten.

Unsere Beispielfunktion enthält 72 physikalische Zeilen. 11 Zeilen hiervon sind Zeilen mit Kommentaren. Mit 15% Kommentarzeilen ist dieser - wenn auch relativ triviale - Codeteil unzureichend kommentiert.

Die zyklomatische Komplexität von McCabe

Die zyklomatische Komplexität (auch bekannt als zyklomatische Zahl, Programmkomplexität oder McCabe-Komplexität) wurde bereits 1976 durch Thomas McCabe eingeführt. Sie ist die am weitesten verbreitete statische Softwaremetrik und unabhängig von der Programmiersprache.

Die die zyklomatische Zahl McCabes, abgekürzt mit v(G), zeigt die Komplexität des Kontrollflusses im Code. v(G) ist die Anzahl der konditionellen Zweige des Flussdiagramms. Für ein Programm, welches lediglich aus sequentiellen Anweisungen besteht, beträgt v(G) = 1.

Für eine einzelne Funktion ist v(G) um eins geringer als die Anzahl der Verzweigungspunkte für Bedingungen (conditional branching points) in dieser Funktion. Je größer die zyklomatische Zahl ist, um so mehr Pfade sind in der Funktion und desto schwieriger ist diese zu verstehen und zu testen. Es sei darauf hingewiesen, dass die Cyclomatic-Number die Komplexität von Datenstrukturen, Datenfluss und Modul-Interfaces unberücksichtigt läßt.

Da die zyklomatische Zahl die Steuerungflusskomplexität beschreibt, ist es offensichtlich, dass Module und Funktionen mit einer hohen zyklomatischen Komplexität mehr Testfälle als solche mit einem niedrigen McCabe-Wert brauchen. Als Faustregel kann man festhalten, dass für jede Funktion mindestens so viele Testfälle ausgeführt werden sollte wie v(G) angibt.

Wie wird die McCabe-Metrik berechnet?

Die McCabe Cyclomatic Number v(G) wird auf der Basis von (Member) Funktionsdefinitionen und Class-/Struct-Deklarationen gemessen.

Hier die Pfade, die in die Berechnung von v(G) einfließen:

- jedes if-Statement führt einen neuen Zweig ein und erhöht deshalb v(G) um den Wert 1;
- Iterationskonstrukte wie for- und while-Schleifen haben ebenfalls neue Zweige zur Folge und erhöhen v(G);
- jede case-Anweisung in switch-Anweisungen erhöht v(G) um eins;
- case-Anweisungen erhöhen v(G) nicht, da die Anzahl der Zweige im Kontrollfluss nicht erhöhen. Bei zwei oder mehr case-Anweisungen, die keinen Code beinhalten wird die McCabe-Zahl lediglich um den Wert Eins für die Gesamtheit dieser case-Anweisungen inkrementiert;

- jede catch-Anweisung in einem Try-Block erhöht v(G) um Eins;
- die Anweisung exprl ? expr2 : expr3 inkrementiert v(G).

Es ist zu beachten, dass v(G) unkonditionelle Zweige wie goto-, return- und break-Anweisungen nicht bei der Berechnung berücksichtigt, obwohl diese sicherlich die Komplexität erhöhen.

Zusammenfassend kann man festhalten, dass die folgenden Sprachkonstruktionen die zyklomatische Zahl erhöhen: *if (...), for (...), while (...), case ...:, catch (...), &&, ||, ?, #if, #ifdef, #ifndef, #elif*

Welches sind akzeptable Werte für v(G)?

Die zyklomatische Zahl einer Funktion sollte unter 15 liegen. Bei einem v(G) von 15, gibt es mindestens 15 (meist jedoch mehr) Ausführungspfade. Mehr als 15 Pfade sind jedoch schwierig zu identifizieren und zu testen.

Eine vernünftige Obergrenze für die zyklomatische Zahl einer Datei liegt bei 100. Sehen wir uns jetzt die Berechnung von v(G) für unser Beispiel an:

- die Berechnung beginnt immer mit dem Wert 1. Zu diesem Wert wird die Anzahl der Verzweigungen addiert;
- für die Funktion eval1 ist v(G)=14 (for : 1 ; && : 4 ; if : 4 ; case : 4);
- für die Funktion extract beträgt v(G)=2 (while : 1);
- für die Funktion main ergibt sich ein v(G)=2 (if : 1).

Die Berechnung der zyklomatischen Zahl auf Dateiebene beginnt ebenfalls mit dem Wert 1. Für jede Funktion wird dann der 1 übersteigende Wert für v(G) addiert.

Unsere Beispieldatei hat demnach folgende zyklomatische Komplexität:

```
v(G) = 1 (Start) + 13 (für eval1) +
 1 (für extract) + 1 (für main) = 16
```

Die McCabe Cyclomatic Number liegt also für alle Funktionen innerhalb der Empfehlung.

Die Halstead-Metriken

Die Halstead-Komplexitätsmetriken wurden durch Maurice Halstead als quantitatives Maß für die Komplexität entwickelt. Sie basieren direkt auf der Anzahl der Operatoren und Operanden in einem Modul.

Die Halstead-Messgrößen zählen zu den ältesten Softwaremetriken. Sie wurden bereits 1977 eingeführt und sind seither intensiv genutzt und verifiziert worden. Die Metriken von Halstead interpretieren den Quellcode als eine Folge von Operatoren und Operanden. Sie stehen im direkten Zusammenhang zum Programmcode und werden daher oft als Metrik zur Messung der Wartbarkeit eingesetzt. Die Halstead-Messungen sind auch nützlich, um die Codequalität bereits während der Entwicklungsphase positiv zu gestalten. Hier die Liste der einzelnen Halstead-Metriken:

- N – Programmlänge (Program length);
- n – Vokabulargröße (Vocabulary size);
- V – Volumen des Programms (Program volume);
- D – Schwierigkeitsgrad (Difficulty level);
- L – Programmniveau (Program level);
- E – Implementierungsaufwand (Effort to implement);
- T – Implementierungszeit (Implementation time);
- B – Anzahl der ausgelieferten Bugs (Number of delivered bugs).

Wie werden diese Metriken berechnet?

Alle Halstead-Metriken werden von der Anzahl der Operanten und Operatoren abgeleitet. Zu den Operanden werden gezählt:

- IDENTIFIER – alle Identifiers, die keine reservierten Wörter sind;
- TYPESPEC – (type specifiers) Reservierte Wörter, die Typen spezifizieren: *bool, char, double, float, int, long, short, signed, unsigned, void*. Hierzu gehören auch einige kompilerspezifische nichtstandardisiert Keywörter;
- CONSTANT – Zeichenkonstanten, numerische oder String-Konstanten.
- Als Operatoren werden angesehen:
- SCSPEC – (storage class specifiers) Reservierte Wörter, die Speicherklassen beschreiben: *auto, extern, inlin, register, static, typedef, virtual, mtuable*;
- TYPE_QUAL – (type qualifiers) Reservierte Wörter, die Typen qualifizieren: *const, friend, volatile*;
- RESERVED – Andere reservierte Wörter der C++ Programmiersprache: *asm, break, case, class, continue, default, delete, do, else, enum, for, goto, if, new, operator, private, protected, public, return, sizeof, struct, switch, this, union, while, namespace, using, try, catch, throw, const_cast, static_cast, dynamic_cast, reizterpret_cast, typeid, template, explicit, true, false, typename*. Diese Kategorie beinhaltet auch einige kompilerspezifische (und nicht standardisierte) Keywords;
- OPERATOR - !; !=; %; %=; &; &&; ||; &=; (); *; *=; +; ++; +=; ,; -; --; -=; ->; .; ...; /; /=; :; :: ; <; <<; <<=; <=; =; ==; >; >=; >>; >>=; ?; []; ^; ^=; {; }; |; |=; ~ in der Regel zählt man auch das Semikolon ; zu den Operatoren (da alle Tokens entweder als Operanden oder Operatoren betrachtet werden, hat man das Semikolon den Operatoren zugerechnet)

Sehen wir uns nochmal die Funktion `eval1` unseres Beispiels an, um die Halstead-Metriken zu berechnen (die Anzahl der Operatoren und Operanden in den Tabellen 1 und 2).

Wir zählen 22 verschiedene Operatoren (n1). Die Gesamtanzahl der Operatoren (N1) beträgt 157. Es gibt 30 verschiedene Operanden (n2) sowie insgesamt 122 Operanden (N2).

Auf der Basis dieser Zahlen errechnet man die *Programmlänge* (*N*). N ist die Summe der Gesamtanzahl der Operatoren (N1) und der Operanden (N2):

```
N = N1 + N2
```

Für unser Beispiel ergibt dies 157 + 122 = 279.

Die Summe der Anzahl der verschiedenen Operatoren (n1) und der verschiedenen Operanden (n2) ergibt die *Vokabulargröße* (*n*)

```
n = n1 + n2
```

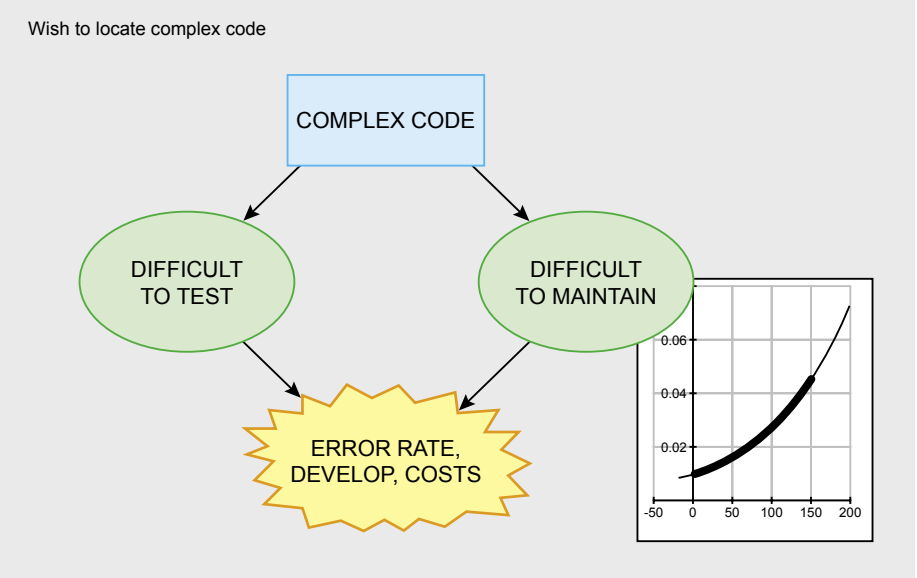

Abbildung 2. Auswirkungen von Komplexität auf die Softwaretests und -wartung

Die Vokabulargröße unserer Beispielfunktion beträgt `22 + 30 = 52`.

Multipliziert man die Programmlänge mit dem Zweierlogarithmus der Vokabulargröße (n), erhält man das *Volumen des Programms* (*V*). Das Halstead-Volumen (V) beschreibt die Größe der Implementierung des Algorithmus und basiert auf der Anzahl der durchgeführten Operationen und der bearbeiteten Operanden im Algorithmus. Der Wert V ist daher im Vergleich zu den Zeilenmetriken weniger vom Code-Layout abhängig. Hier die Formel:

```
V = N * log2(n)
```

Für unser Beispiel errechnen wir folgenden Wert: `V = 279 * log2(52) = 1590,423`

Das Volumen einer Funktion sollte mindestens 20 und höchstens 1000 betragen. Eine parameterlose Funktion, die aus einer nicht leeren Zeile besteht beträgt etwa 20. Wenn das Volumen den Wert von 1000 übersteigt, macht die Funktion wahrscheinlich zu viele Dinge.

Das Volumen einer Datei sollte zwischen 100 und höchstens 8000 liegen. Diese Grenzwerte basieren auf Volumen, deren Anzahl an physikalischen Zeilen und deren zyklomatische Zahl v(G) nahe der empfohlenen Limits liegen. Sie können daher zur doppelten Kontrolle dienen.

Tabelle 1. Die Anzahl Operatoren

Operator	Frequency
!=	6
&	2
&&	4
()	31
*	14
+	11
++	1
,	12
-	3
/	1
;	33
=	10
==	3
break	3
case ...:	4
else	3
for()	1
if()	4
return	1
sizeof	2
switch()	1
{}	7

Der *Schwierigkeitsgrad* (difficulty level, D) oder Fehlerneigung eines Programms ist proportional zur Anzahl der verschiedenen Operatoren in diesem Programm. D ist ebenfalls proportional zum Verhältnis zwischen der Gesamtzahl der Operanden und der Anzahl der verschiedenen Operanden. Wird der gleiche Operand beispielsweise mehrmals im Programm benutzt, wird er dadurch fehleranfälliger.

```
D = ( n1 / 2 ) * ( N2 / n2 )
```

Hier der Wert für D in unserem Beispiel:

```
( 22 / 2 ) * ( 122 / 30 ) =
   11 * 4,0666 = 44,733
```

Über den Kehrwert des Schwierigkeitsgrades erhalten wir das *Programmniveau* (*L*) bzw. die Fehlerneigung eines Programms. Ein Programm mit einem niedrigen Niveau ist fehlerträchtig.

```
L = 1 / D
```

Für unser Beispiel errechnen wir somit `L = 1 / 44,733 = 0,022` Der Implementieraufwand (Effort to implement, E) ist proportional zum Volumen und zum Schwierigkeitsgrad des Programms. Er wird mit folgender Formel errechnet:

```
E = V * D
```

Der *Implementieraufwand* unserer Beispielfunktion eval1 liegt also bei

```
E = 1590,423 * 44,733 =
   71144,908
```

Tabelle 2. Die Anzahl der Operanden

Operand	Frequency
"%d"	2
"Error: Division durch 0 nicht moeglich"	1
"Error: kein <operator>"	1
"Error: kein <wert1>"	1
"Error: kein <wert2>"	1
'*'	2
'+'	2
'-'	2
'/'	2
'\0'	1
0	9
1	5
ch	12
char	7
eval1	1
exit	4
extract	2
i	16
int	5
lgval2	4
malloc	2
operation	3
printf	4
resultat	7
sscanf	2
strlen	3
val1	4
val2	4
wert1	6
wert2	7

Die *Implementierzeit* (*T*) und die Zeit, die notwendig ist ein Programm zu verstehen, ist proportional zum Implementieraufwand. Empirische Untersuchungen haben ergeben, daß man eine guten Wert für die notwendige Zeit in Sekunden erhält, wenn man den Implementieraufwand (E) durch 18 teilt.

```
T = E / 18
```

Die Implementierzeit unserer Beispielfunktion beträgt `T = 71144,908 / 18 = 3952,466`

Um die Funktion zu schreiben, sollte man also etwas mehr als eine Stunde einplanen.

Der Implementieraufwand (E) dient zur Berechnung der *Anzahl der ausgelieferten Bugs* (*B*) gemäß folgender Formel:

```
B = ( E ** (2/3) ) / 3000
```

`**` steht für den Exponenten. Dieser Wert gibt eine Schätzung über die Zahl der Fehler in der Implementierung. Die Berechnung für

```
B = ( 71144,908 ** (2/3) ) / 3000
```

zeigt, dass ein Softwaremodul mit der Komplexität unserer Beispielfunktion wahrscheinlich 0,572 Fehler enthält.

Die Metrik B kann als Anhaltspunkt für die Anzahl der Fehler genutzt werden, welche während des Modultests aufgedeckt werden sollten. Die Erfahrung zeigt, dass eine in C oder C++ geschriebene Quellcodedatei fast immer mehr Fehler enthält, als B anzeigt.

Der Wartbarkeitsindex (Maintainability Index, MI)

Der Wartbarkeitsindex oder Maintainability Index ist während der letzten zehn Jahre in den USA definiert worden. Die Messung der Wartbarkeit hilft unter anderem dabei, festzustellen, wann es günstiger bzw. weniger riskant ist Codeteile neu zu schreiben, anstatt sie zu verändern. Der Wartbarkeitsindex wird auf der Basis der Zeilenmetriken, der zyklomatischen Zahl von McCabe und der Halstead-Metriken berechnet. Es gibt drei Varianten für die Messung der Wartbarkeit:

- Maintainability Index ohne Kommentare (MIwoc, Maintainability Index without comments)

  ```
  MIwoc = 171 - 5.2 * ln(aveV) -
     0.23 * aveG -16.2 * ln(aveLOC)
  ```

  - aveV – durchschnittliches Halstead Volumen (V) pro Modul
  - aveG – durchschnittliche zyklomatische Komplexität v(G) pro Modul
  - aveLOC – durchschnittliche Anzahl der Programmcodezeilen (LOCphy) pro Modul
- Maintainability Index der Kommentare (MIcw, Maintainability Index comment weight)

  ```
  MIcw = 50 * sin(sqrt(2.4 * perCM))
  ```

  - perCM ist hier der durchschnittliche Anteil der Zeilen mit Kommentaren pro Modul (average percent of lines of comments per Module)
- der eigentliche Wartbarkeitsindex (MI, Maintainability Index) ist die Summe der beiden vorhergehenden Metriken

  ```
  MI = MIwoc + MIcw
  ```

Der Wert für MI gibt an, wie schwierig (oder einfach) eine Applikation zu warten ist. Berechnen wir die Wartbarkeit unserer Beispielfunktion `eval1`:

Zunächst ermitteln wir die Wartbarkeit ohne Berücksichtigung der Kommentare:

```
MIwoc = 171 - 5,2 * ln(aveV)
      -0,23 * aveG -16,2 * ln(aveLOC)
```

- hierbei sind aveV = durchschnittliches Halstead Volumen (V) = 1590,423
- aveG = durchschnittliche zyklomatische Komplexität v(G) = 14
- aveLOC = durchschnittliche Anzahl der Programmcodezeilen (LOCphy) = 72

Da wir hier nur den MI für eine einzige Funktion berechnen, entspricht der Durchschnittswert den Werten für die Funktion `eval1`.

```
MIwoc = 171 - 5.2 * ln(1590,423)
        - 0.23 * 14 - 16.2 * ln(72)
MIwoc = 171 - 5.2 * 7,371755
        - 0.23 * 14 - 16.2 * 4,276666
MIwoc = 171 - 38,333126
        - 3,22 - 69,281989
MIwoc = 60,165
```

Nun berechnen wir die Wartbarkeit bezüglich des Kommentaranteils:

```
MIcw = 50 * sin(sqrt(2.4 * perCM))
```

Weiter oben hatten wir bereits festgestellt, dass der Anteil der Kommentare an der gesamten Funktion 11/72, also etwas 15% beträgt

```
MIcw = 50 * sin(sqrt(2.4 * 11/72))
MIcw = 28,460
```

Schließlich ermitteln wir den Wartbarkeitsindex wie folgt:

```
MI = MIwoc + MIcw
MI = 60,165 + 28,460
MI = 88,625
```

Ein Wert von mindestens 85 zeigt eine gute Wartbarkeit an. Liegt der Wert zwischen 65 und 85 ist die Wartbarkeit mäßig. Module mit einem MI unter 65 sind schwierig zu warten. Diese Funktionen sollte man möglichst neu schreiben. Unsere Funktion eval1 hat mit einem MI von 88,625 also eine korrekte Wartbarkeit.

Berechnung der Metriken für die anderen Funktionen unseres Beispiels

Das Listing 2 zeigt Ihnen die Ergebnisse für die Metriken der anderen Funktionen. Versuchen Sie doch einmal die Metriken hierfür zu berechnen.

Wahrscheinlich wird Ihnen dabei auffallen, dass die Ermittlung der Komplexität *von Hand* relativ aufwendig ist. Aus diesem Grund gibt es verschiedene Tools, die die Metriken automatisch berechnen und Ihnen Komplexität und Wartbarkeit Ihrer Softwaremodule innerhalb von nur wenigen Sekunden anzeigen.

Zusammenfassung

Zeilenmetriken, Halstead-Metriken, die zyklomatische Komplexität von McCabe und der Maintainability Index (MI) sind wirksame Mittel für die Feststellung der Komplexität und Wartbarkeit einer Software. Diese Metriken geben Ihnen eine gute Aussage über die Codequalität.

Mit Hilfe der verschiedenen Metriken lassen sich komplexe Codeteile auf relativ einfache Weise ermitteln. Problemstellen im Code, die beim Test und der Wartung der Software Schwierigkeiten bereiten, werden somit aufgedeckt.

Es empfiehlt sich, die entsprechenden Module frühzeitig zu modifizieren, um schwierige Softwaretests und hohe Wartungskosten zu vermeiden.

XAVIER-NOËL CULLMANN

Xavier-Noël Cullmann, geboren 1984 in Mülhausen (Elsaß), hat Informatik an der Universität Straßburg studiert. Seit Anfang 2006 hat er sich bei Verifysoft Technology mit Softwareanalysetools beschäftigt. Xavier-Noël ist Vorsitzender von Zilat http://www.zilat.net, eines Vereins, der Lanparties im Departement Haut-Rhin organisiert.

Kontakt zum Autor: cullmann@verifysoft.com

KLAUS LAMBERTZ

Klaus Lambertz ist einer der Gründer von Verifysoft Technology http://www.verifysoft.com in Offenburg. Verifysoft Technology ist Spezialist für Softwaretest- und -analysetools. Klaus wurde 1962 in Köln geboren und lebt seit 1993 in Frankreich. Seine langjährige Erfahrung im Testtoolbereich hat er bei verschiedenen Firmen in Deutschland, Frankreich und den USA gesammelt.

Kontakt zum Autor: lambertz@verifysoft.com