Analysierende Testverfahren

Software-Metriken

Kontrolle der Software-Entwicklung:

- Pläne und Standards einrichten
- messen der Ausführung gegen Pläne und Standards
- korrigieren der Abweichungen

Eine **Software-Metrik** definiert, wie eine Kenngröße eines Software-Produkts oder -Prozesses gemessen wird.

Software-Metriken (2)

Messgrößen im Software-Prozess:

- Ressourcenaufwand (Mitarbeiter, Zeit, Geld)
- Fehler
- Kommunikationsaufwand

Messgrößen im Software-Produkt:

- Umfang (LOC, %Wiederverwendung, #Prozeduren)
- Komplexität
- Lesbarkeit (Stil)
- Entwurfsqualität (Modularität, Kopplung, Bindung)
- Produktqualität (Testergebnisse, Testabdeckung)

Software-Metriken (3)

Gütekriterien für Software-Metriken:

- **Objektivität** keine subjektiven Einflüsse des Messenden
- **Zuverlässigkeit** bei Wiederholung gleiche Ergebnisse
- **Normierung** Skala für Messergebnisse und Vergleichsskala
- **Vergleichbarkeit** Maß steht mit anderen Maßen in Relation
- **Ökonomie** Messung hat geringe Kosten
- **Nützlichkeit** Messung erfüllt praktische Bedürfnisse
- **Validität** Messergebnisse ermöglichen Rückschluss auf Kenngröße

Software-Metriken (4)

Messen der Verständlichkeit:

- Gutachter lesen Programme und beurteilen sie anhand von Fragen:
  - Ist das Programm an veränderte Rahmenbedingungen anpassbar?
  - Sind die Bezeichner aussagekräftig und konsistent?
- Beurteilung erfolgt z.B. in Form von Schulnoten

Der Ansatz ist wenig objektiv, wenig zuverlässig, halbwegs normiert, halbwegs vergleichbar, nicht ökonomisch, aber nützlich und valide.

→ maschinell prüfbare Indikatoren für Verständlichkeit?

Umfangs-Metriken

Lines Of Code (LOC): zählt Anzahl Zeilen im Code

```
> cat *.[ch] | wc -l
182377
> _
```

Non-Commented Source Statements (NCSS): wie LOC, aber Leerzeilen und Kommentarzeilen werden ignoriert

```
> grep -v '^[ \t]*$' *.c | grep -v '^[ \t]*//' | wc -l
127387
> _
```


Umfangs-Metriken (2)

positiv:

- sehr einfach zu messen und zu berechnen
- anwendbar auf alle Arten von Programmen

negativ:

- was soll gezählt werden?
- Umfang abhängig von der Sprache (Perl vs. C++)

Halstead-Metriken

- eingeführt zur Messung der textuellen Komplexität
- Einteilen des Programms in Operatoren und Operanden

```
int ggt(int a, int b) {
    assert(a >= 0);
    assert(b >= 0);
    while (b > 0) {
        int r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

- **Operatoren:** kennzeichnen Aktionen, typisch: Sprachelemente des Programms
  hier: `int`, `()`, `,`, `{}`
- **Operanden:** kennzeichnen Daten, typisch: Bezeichner und Literale
  hier: `ggt`, `a`, `b`, `0`

Halstead-Metriken (2)

Basisgrößen:

- n_1: Anzahl unterschiedlicher Operatoren
- n_2: Anzahl unterschiedlicher Operanden
- N_1: Anzahl verwendeter Operatoren
- N_2: Anzahl verwendeter Operanden
- $n = n_1 + n_2$: Größe des Vokabulars
- $N = N_1 + N_2$: Länge der Implementierung

im Beispiel: $n_1 = 10$, $n_2 = 4$, $N_1 = 19$, $N_2 = 16$

Halstead-Metriken (3)

abgeleitete Größen:

- **difficulty D:** Schwierigkeit, ein Programm zu verstehen

$$D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}$$

- **volume V:** Umfang des Programms

$$V = N \cdot \log(n)$$

- **effort E:** Aufwand, das Programm zu verstehen

$$E = D \cdot V$$

n_1: # unterschiedlicher Operatoren N_1: # verwendeter Operatoren
n_2: # unterschiedlicher Operanden N_2: # verwendeter Operanden

Halstead-Metriken (4)

positiv:

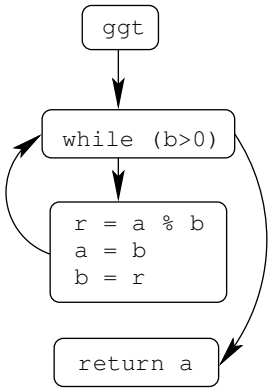
- einfach zu ermitteln und zu berechnen
- für alle Programmiersprachen einsetzbar
- Experimente zeigen: gutes Maß für Komplexität

negativ:

- berücksichtigt nur textuelle Komplexität
- moderne Programmierkonzepte wie Sichtbarkeit oder Namensräume werden nicht berücksichtigt
- Aufteilung Operatoren/Operanden sprachabhängig

McCabe-Metriken

- strukturelle Komplexität berücksichtigen
- Quelle: Kontrollflussgraph G
- Ziel: zyklomatische Komplexität $V(G)$



$V(G) = e - n + 2p$

- e: Anzahl Kanten (hier 4)
- n: Anzahl Knoten (hier 4)
- p: Anzahl Komponenten (hier 1)

$\Rightarrow$ hier: $V(G) = 2$

für $p = 1$ gilt: $V(G) = \pi + 1$
π: Anzahl der Bedingungen

McCabe-Metriken (2)

Faustregeln:

$V(G)$	Risiko
1-10	einfaches Programm, geringes Risiko
11-20	komplexeres Programm, erträgliches Risiko
21-50	komplexes Programm, hohes Risiko
> 50	untestbares Programm, extrem hohes Risiko

beginne bei einer Umstrukturierung mit der Komponente, die die höchste zyklomatische Komplexität hat

McCabe-Metriken (3)

positiv:

- einfach zu berechnen
- Integration mit Testplanung (Bedingungsüberdeckung)
- Studien zeigen: gute Korrelation zwischen zyklomatischer Zahl und Verständlichkeit der Komponente

negativ:

- Metrik berücksichtigt nur Kontrollfluss
- Komplexität des Datenflusses nicht berücksichtigt
- Kontrollfluss zwischen Komponenten evtl. sehr komplex
- ungeeignet für objektorientierte Programme (viele triviale Funktionen)

hybride Metriken

Idee: kombiniere Metriken

Wartbarkeits-Index MI bei Hewlett-Packard:

$$171 - 5.2 \ln(\overline{V}) - 0.23\overline{V(G)} - 16.2 \ln(\overline{L}) + 50 \sin\left(\sqrt{2.4\overline{C}}\right)$$

- $\overline{V}$: durchschnittliches Halstead-Volumen pro Modul
- $\overline{V(G)}$: durchschnittliche zyklomatische Zahl pro Modul
- $\overline{L}$: durchschnittliche LOC pro Modul
- $\overline{C}$: durchschnittlicher Prozentsatz an Kommentarzeilen

unterschreitet ein Modul einen vorgegebenen Wert (z.B. 30), so wird das Modul restrukturiert

Metriken für OO-Komponenten

McCabe-Metrik versagt bei OOP, da die Kontrollflusskomplexität der meisten Methoden gering ist: $V(G) = 1$

Metriken müssen Zusammenspiel der Klassen betrachten!

- typisch: anhand des statischen Objektmodells
- Metriken für dynamische Aspekte (Sequenzdiagramme, Zustandsautomaten)???

signifikante OO-Metriken:

- **DIT (Depth of Inheritance Tree)**
  Anzahl Oberklassen einer Klasse
  DIT größer $\Rightarrow$ Fehlerwahrscheinlichkeit größer

Metriken für OO-Komponenten (2)

- **NOC (Number Of Children of a class)**
  Anzahl direkter Unterklassen
  NOC größer ⇒ Fehlerwahrscheinlichkeit geringer
- **RFC (Response For a Class)**
  Anzahl Funktionen, die direkt durch Klassenmethoden aufgerufen werden
  RFC größer ⇒ Fehlerwahrscheinlichkeit größer
- **WMC (Weighted Methods per Class)**
  Anzahl definierter Methoden
  WMC größer ⇒ Fehlerwahrscheinlichkeit größer
- **CBO (Coupling Between Object classes)**
  Anzahl Klassen, auf deren Dienste die Klasse zugreift
  CBO größer ⇒ Fehlerwahrscheinlichkeit größer

Metriken für OO-Komponenten (3)

Kombination von DIT, NOC, RFC usw. mit McCabe- oder Halstead-Metrik möglich

in der Literatur:

- Autoren schlagen neue Kombinationsmetrik vor ...
- und beschreiben, welche Werte die Metrik berechnet.
- **Aber:** es fehlt die Validierung

Kritik an Metriken:

- berechnen von Metriken ist kein Ersatz für Gegenlesen, Test oder Verifikation
- was ist mit neuronalen Netzen, Data Mining, ...?