Software-Metriken

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

Software-Metriken

Zu den Aufgaben eines Managers gehört die *Kontrolle* der Software-Entwicklung:

1. Pläne und Standards einrichten







Software-Metriken

Zu den Aufgaben eines Managers gehört die *Kontrolle* der Software-Entwicklung:

1. Pläne und Standards einrichten
2. Messen der Ausführung gegen Pläne und Standards







Software-Metriken

Zu den Aufgaben eines Managers gehört die *Kontrolle* der Software-Entwicklung:

1. Pläne und Standards einrichten
2. Messen der Ausführung gegen Pläne und Standards
3. Korrektur der Abweichungen







Software-Metriken

Zu den Aufgaben eines Managers gehört die *Kontrolle* der Software-Entwicklung:

1. Pläne und Standards einrichten
2. Messen der Ausführung gegen Pläne und Standards
3. Korrektur der Abweichungen

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

> *You can't control what you can't measure!* (DeMarco)







Was kann man messen?

Im *Software-Prozess:*

- Ressourcenaufwand (Mitarbeiter, Zeit, Kosten, . . .)
- Fehler
- Kommunikationsaufwand

Im *Produkt:*

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







Wie muß man messen?

Gütekriterien für Software-Metriken:

Objektivität – keine subjektiven Einflüsse des Messenden







Wie muß man messen?

Gütekriterien für Software-Metriken:

Objektivität – keine subjektiven Einflüsse des Messenden

Zuverlässigkeit – bei Wiederholung gleiche Ergebnisse







Wie muß man messen?

Gütekriterien für Software-Metriken:

Objektivität – keine subjektiven Einflüsse des Messenden

Zuverlässigkeit – bei Wiederholung gleiche Ergebnisse

Normierung – Es gibt eine Skala für Meßergebnisse und eine Vergleichbarkeitsskala







Wie muß man messen?

Gütekriterien für Software-Metriken:

Objektivität – keine subjektiven Einflüsse des Messenden

Zuverlässigkeit – bei Wiederholung gleiche Ergebnisse

Normierung – Es gibt eine Skala für Meßergebnisse und eine Vergleichbarkeitsskala

Vergleichbarkeit – Man muß das Maß mit anderen Maßen in eine Relation setzen können







Wie muß man messen?

Gütekriterien für Software-Metriken:

Objektivität – keine subjektiven Einflüsse des Messenden

Zuverlässigkeit – bei Wiederholung gleiche Ergebnisse

Normierung – Es gibt eine Skala für Meßergebnisse und eine Vergleichbarkeitsskala

Vergleichbarkeit – Man muß das Maß mit anderen Maßen in eine Relation setzen können

Ökonomie – Messung hat geringe Kosten







Wie muß man messen?

Gütekriterien für Software-Metriken:

Objektivität – keine subjektiven Einflüsse des Messenden

Zuverlässigkeit – bei Wiederholung gleiche Ergebnisse

Normierung – Es gibt eine Skala für Meßergebnisse und eine Vergleichbarkeitsskala

Vergleichbarkeit – Man muß das Maß mit anderen Maßen in eine Relation setzen können

Ökonomie – Messung hat geringe Kosten

Nützlichkeit – Messung erfüllt praktische Bedürfnisse

Wie muß man messen?

Gütekriterien für Software-Metriken:

Objektivität – keine subjektiven Einflüsse des Messenden

Zuverlässigkeit – bei Wiederholung gleiche Ergebnisse

Normierung – Es gibt eine Skala für Meßergebnisse und eine Vergleichbarkeitsskala

Vergleichbarkeit – Man muß das Maß mit anderen Maßen in eine Relation setzen können

Ökonomie – Messung hat geringe Kosten

Nützlichkeit – Messung erfüllt praktische Bedürfnisse

Validität – Meßergebnisse ermöglichen Rückschluß auf Kenngröße

Wie muß man messen?

Gütekriterien für Software-Metriken:

Objektivität – keine subjektiven Einflüsse des Messenden

Zuverlässigkeit – bei Wiederholung gleiche Ergebnisse

Normierung – Es gibt eine Skala für Meßergebnisse und eine Vergleichbarkeitsskala

Vergleichbarkeit – Man muß das Maß mit anderen Maßen in eine Relation setzen können

Ökonomie – Messung hat geringe Kosten

Nützlichkeit – Messung erfüllt praktische Bedürfnisse

Validität – Meßergebnisse ermöglichen Rückschluß auf Kenngröße *(schwierigstes Kriterium)*

Anwendung: Praktomat

Praktomat – Ziele

Wir möchten,

1. daß Studenten *Lehrbuchmäßige Programme* schreiben:
   ⇒ Hohe Anforderung an Funktionalität und Verständlichkeit







Praktomat – Ziele

Wir möchten,

1. daß Studenten *Lehrbuchmäßige Programme* schreiben:
   ⇒ Hohe Anforderung an Funktionalität und Verständlichkeit
2. daß wir ein *faires* Bewertungsverfahren haben
   ⇒ gleiche Kriterien für alle







Praktomat – Ziele

Wir möchten,

1. daß Studenten *Lehrbuchmäßige Programme* schreiben:
   ⇒ Hohe Anforderung an Funktionalität und Verständlichkeit
2. daß wir ein *faires* Bewertungsverfahren haben
   ⇒ gleiche Kriterien für alle
3. daß die Bewerter so wenig Arbeit wie möglich haben
   ⇒ weitestgehende Automatisierung







Praktomat – Bewertung

Was möchten wir bewerten?

Funktionalität – insbesondere das *Bestehen von Tests*







Praktomat – Bewertung

Was möchten wir bewerten?

Funktionalität – insbesondere das *Bestehen von Tests*

Verständlichkeit – insbesondere

- Dokumentation
- Einrückung
- Struktur, Komplexität
- Wahl von Bezeichnern
- Lokalität
- Anpaßbarkeit an neue Anforderungen







Messen der Funktionalität

1. Wir definieren *Testfälle* nach den Regeln der Kunst
2. Wir messen, wieviele Testfälle bestanden wurden
3. Betreuer schließen daraus auf die Funktionalität des Programms

Ansatz ist *objektiv, zuverlässig, normierbar, vergleichbar, ökonomisch, nützlich* und *valide*.

Folge: Bestehen von Testfällen ist *Bedingung* für das Einreichen







Messen der Verständlichkeit

1. Betreuer lesen Programme. . .
2. und beurteilen sie anhand einer Menge von Fragen:
   - *Kann das Programm an veränderte Randbedingungen angepaßt werden?*
   - *Sind die Bezeichner aussagekräftig und konsistent gewählt?* . . .

   Beurteilung:
   von A („ja + Weiterempfehlung“) bis D („nein + Warnung“)







Messen der Verständlichkeit

1. Betreuer lesen Programme. . .
2. und beurteilen sie anhand einer Menge von Fragen:
   - *Kann das Programm an veränderte Randbedingungen angepaßt werden?*
   - *Sind die Bezeichner aussagekräftig und konsistent gewählt?* . . .

   Beurteilung:
   von A („ja + Weiterempfehlung“) bis D („nein + Warnung“)

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

Gibt es *maschinell prüfbare* Indikatoren für Verständlichkeit?

Umfangs-Metriken

Die einfachsten Metriken, um den *Umfang* eines Programms zu bestimmen, sind

Lines of code (LOC) zählt einfach Zeilen der Quelldateien:

```
[ddd] $
```







Umfangs-Metriken

Die einfachsten Metriken, um den *Umfang* eines Programms zu bestimmen, sind

Lines of code (LOC) zählt einfach Zeilen der Quelldateien:

```
[ddd] $ cat *.[Ch] | wc -l
```







Umfangs-Metriken

Die einfachsten Metriken, um den *Umfang* eines Programms zu bestimmen, sind

Lines of code (LOC) zählt einfach Zeilen der Quelldateien:

```
[ddd] $ cat *.[Ch] | wc -l
182423
[ddd] $ _
```

Non-commented source statements (NCSS) Wie LOC; aber: Leerzeilen und Zeilen, die nur aus Kommentaren bestehen, werden ignoriert

```
[ddd] $
```







Umfangs-Metriken

Die einfachsten Metriken, um den *Umfang* eines Programms zu bestimmen, sind

Lines of code (LOC) zählt einfach Zeilen der Quelldateien:

```
[ddd] $ cat *.[Ch] | wc -l
182423
[ddd] $ _
```

Non-commented source statements (NCSS) Wie LOC; aber: Leerzeilen und Zeilen, die nur aus Kommentaren bestehen, werden ignoriert

```
[ddd] $ grep -v '^[ \t]*$' *.[Ch] | grep -v '[ \t]*//' | wc -l
```







Umfangs-Metriken

Die einfachsten Metriken, um den *Umfang* eines Programms zu bestimmen, sind

Lines of code (LOC) zählt einfach Zeilen der Quelldateien:

```
[ddd] $ cat *.[Ch] | wc -l
182423
[ddd] $ _
```

Non-commented source statements (NCSS) Wie LOC; aber: Leerzeilen und Zeilen, die nur aus Kommentaren bestehen, werden ignoriert

```
[ddd] $ grep -v '^[ \t]*$' *.[Ch] | grep -v '[ \t]*//' | wc -l
127465
[ddd] $ _
```

Gängige Metrik in der Industrie für *Produktivität*: z.B. „300 NCSS/Ingenieurmonat“







Umfangs-Metriken: Bewertung

Für Umfangs-Metriken gilt:

- ✔ Extrem einfach zu messen und zu berechnen
- ✔ Garantiert anwendbar auf alle Arten von Programmen

- ✘ Was soll gezählt werden?
- ✘ Umfang variiert abhängig von der Sprache (z.B. Perl vs. C++)
- ✘ Besser strukturierte Programme können kleineren Umfang haben







Halstead-Metriken

eingeführt zur Messung der *textuellen Komplexität*







Halstead-Metriken

eingeführt zur Messung der *textuellen Komplexität*

Einteilung des Programms in **Operatoren** und *Operanden*

```
int ggt(int x, int y)
{
    assert (x >= 0);
    assert (y >= 0);
    while (x != y)
        if (x > y)
            x -= y;
        else
            y -= x;

    return x;
}
```







Halstead-Metriken

eingeführt zur Messung der *textuellen Komplexität*

Einteilung des Programms in **Operatoren** und *Operanden*

```
int ggt(int x, int y)
{
    assert (x >= 0);
    assert (y >= 0);
    while (x != y)
        if (x > y)
            x -= y;
        else
            y -= x;

    return x;
}
```

Operatoren (`int`, `()`, `,`, `{}`, ...) kennzeichnen *Aktionen*; typischerweise *Sprachelemente* des Programms

Operanden (*ggt*, *x*, *y*, *0*) kennzeichnen *Daten*; typischerweise *Bezeichner* und *Literale*

Halstead-Metriken (2)

Wir bestimmen folgende *Basisgrößen:*

```
int ggt(int x, int y)
{
    assert (x >= 0);
    assert (y >= 0);
    while (x != y)
        if (x > y)
            x -= y;
        else
            y -= x;

    return x;
}
```

η_1: # unterschiedliche **Operatoren** (12)

η_2: # unterschiedliche *Operanden* (4)







Halstead-Metriken (2)

Wir bestimmen folgende *Basisgrößen:*

```
int ggt(int x, int y)
{
    assert (x >= 0);
    assert (y >= 0);
    while (x != y)
        if (x > y)
            x -= y;
        else
            y -= x;

    return x;
}
```

η_1: # unterschiedliche **Operatoren** (12)

η_2: # unterschiedliche *Operanden* (4)

N_1: # verwendete **Operatoren** (20)

N_2: # verwendete *Operanden* (16)







Halstead-Metriken (2)

Wir bestimmen folgende *Basisgrößen:*

```
int ggt(int x, int y)
{
    assert (x >= 0);
    assert (y >= 0);
    while (x != y)
        if (x > y)
            x -= y;
        else
            y -= x;

    return x;
}
```

η_1: # unterschiedliche **Operatoren** (12)

η_2: # unterschiedliche *Operanden* (4)

N_1: # verwendete **Operatoren** (20)

N_2: # verwendete *Operanden* (16)

$\eta = \eta_1 + \eta_2$: Größe des Vokabulars

$N = N_1 + N_2$: Länge der Implementierung

Halstead-Metriken (2)

Wir bestimmen folgende *Basisgrößen:*

```
int ggt(int x, int y)
{
    assert (x >= 0);
    assert (y >= 0);
    while (x != y)
        if (x > y)
            x -= y;
        else
            y -= x;

    return x;
}
```

η_1: # unterschiedliche **Operatoren** (12)

η_2: # unterschiedliche *Operanden* (4)

N_1: # verwendete **Operatoren** (20)

N_2: # verwendete *Operanden* (16)

$\eta = \eta_1 + \eta_2$: Größe des Vokabulars

$N = N_1 + N_2$: Länge der Implementierung

Halstead-Metriken (3)

η_1: # unterschiedliche **Operatoren** N_1: # verwendete **Operatoren**
η_2: # unterschiedliche *Operanden* N_2: # verwendete *Operanden*
$\eta = \eta_1 + \eta_2$: Größe des Vokabulars $N = N_1 + N_2$: Länge der Implementierung

Aus diesen Basisgrößen lassen sich weitere Größen ableiten:

$$D = \frac{\eta_1}{2} * \frac{N_2}{\eta_2}$$

D (difficulty) ist die *Schwierigkeit, ein Programm zu verstehen* (mit N_2/η_2: durchschnittliches Auftreten der Operanden)

Halstead-Metriken (3)

η_1: # unterschiedliche **Operatoren** N_1: # verwendete **Operatoren**
η_2: # unterschiedliche *Operanden* N_2: # verwendete *Operanden*
$\eta = \eta_1 + \eta_2$: Größe des Vokabulars $N = N_1 + N_2$: Länge der Implementierung

Aus diesen Basisgrößen lassen sich weitere Größen ableiten:

$$D = \frac{\eta_1}{2} * \frac{N_2}{\eta_2}$$

D (difficulty) ist die *Schwierigkeit, ein Programm zu verstehen* (mit N_2/η_2: durchschnittliches Auftreten der Operanden)

$$V = N * \log_2 \eta$$

V (volume) ist der *Umfang des Programms* („Größe des Algorithmus in Bits“)







Halstead-Metriken (3)

η_1: # unterschiedliche **Operatoren** N_1: # verwendete **Operatoren**
η_2: # unterschiedliche *Operanden* N_2: # verwendete *Operanden*
$\eta = \eta_1 + \eta_2$: Größe des Vokabulars $N = N_1 + N_2$: Länge der Implementierung

Aus diesen Basisgrößen lassen sich weitere Größen ableiten:

$$D = \frac{\eta_1}{2} * \frac{N_2}{\eta_2}$$

D (difficulty) ist die *Schwierigkeit, ein Programm zu verstehen* (mit N_2/η_2: durchschnittliches Auftreten der Operanden)

$$V = N * \log_2 \eta$$

V (volume) ist der *Umfang des Programms* („Größe des Algorithmus in Bits“)

$$E = D * V$$

E (effort) ist der Aufwand, das *gesamte Programm zu verstehen*

Halstead-Metriken: Bewertung

Für Halstead-Metriken gilt:

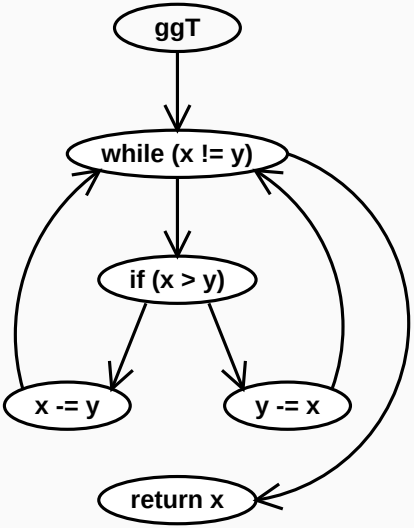
- ✔ Einfach zu ermitteln und zu berechnen (Scanner genügt)
- ✔ Für alle Programmiersprachen einsetzbar
- ✔ Experimente zeigen, daß Halstead-Metriken gutes Maß für Komplexität sind

- ✘ Metrik berücksichtigt nur lexikalische/textuelle Komplexität
- ✘ Moderne Programmierkonzepte wie Sichtbarkeit oder Namensräume völlig unberücksichtigt
- ✘ Aufteilung Operatoren/Operanden sprachabhängig

McCabe-Metriken

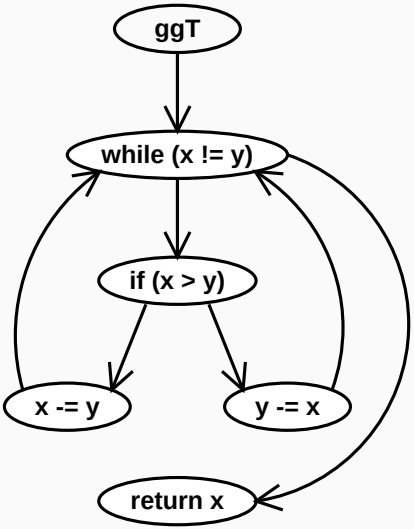
Ansatz: *Strukturelle Komplexität* berücksichtigen

Quelle: *Kontrollflußgraph*



McCabe-Metriken

Ansatz: *Strukturelle Komplexität* berücksichtigen

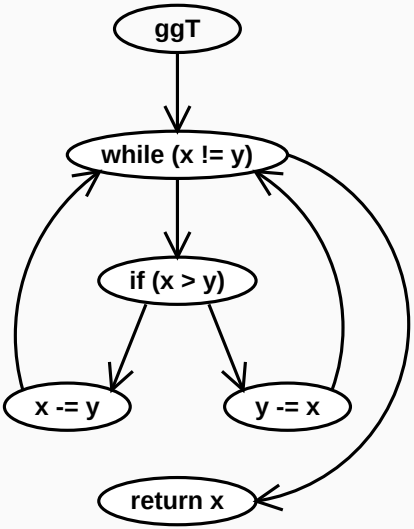


Quelle: *Kontrollflußgraph* G
Ziel: *Zyklomatische Komplexität* $V(G)$

$V(G) = e - n + 2p$
e: Anzahl der Kanten (hier: 7)
n: Anzahl der Knoten (6)
p: Anzahl der Komponenten (1)

McCabe-Metriken

Ansatz: *Strukturelle Komplexität* berücksichtigen

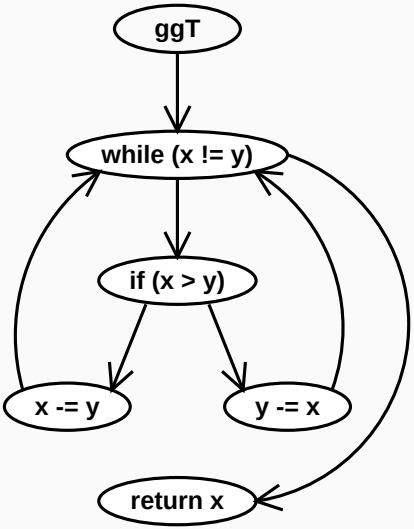


Quelle: *Kontrollflußgraph* G
Ziel: *Zyklomatische Komplexität* $V(G)$

$V(G) = e - n + 2p$
e: Anzahl der Kanten (hier: 7)
n: Anzahl der Knoten (6)
p: Anzahl der Komponenten (1)
Hier: $V(G) = 3$

McCabe-Metriken

Ansatz: *Strukturelle Komplexität* berücksichtigen



Quelle: *Kontrollflußgraph* G
Ziel: *Zyklomatische Komplexität* $V(G)$

$V(G) = e - n + 2p$
e: Anzahl der Kanten (hier: 7)
n: Anzahl der Knoten (6)
p: Anzahl der Komponenten (1)
Hier: $V(G) = 3$

Ist $p = 1$, so ist $V(G) = \pi + 1$
mit π: Anzahl der *Bedingungen* (2)

McCabe-Metriken (2)

Faustregeln für die zyklomatische Komplexität $V(G)$:

$V(G)$	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

oder einfacher:

Wenn eine Umstrukturierung ansteht, dann beginne mit der Komponente, die die höchste zyklomatische Komplexität hat!







McCabe-Metriken: Bewertung

Für McCabe-Metriken gilt:

- ✔ Einfach zu berechnen (Parser genügt)
- ✔ Integration mit Test-Planung (z.B. Bedingungsüberdeckung)
- ✔ Studien zeigen: generell gute Korrelation zwischen zyklomatischer Zahl und Verständlichkeit der Komponente

- ✘ Metrik berücksichtigt nur Kontrollfluß
- ✘ Komplexität des Datenflusses wird nicht berücksichtigt
- ✘ Kontrollfluß zwischen Komponenten kann sehr komplex sein
- ✘ Ungeeignet für objektorientierte Programme (zahlreiche triviale Funktionen)

Hybride Metriken

Grundidee: Zahlreiche Metriken kombinieren







Hybride Metriken

Grundidee: Zahlreiche Metriken kombinieren

Beispiel: *Wartbarkeits-Index* (MI) bei Hewlett-Packard:

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

$\overline{V}$: Durchschnittliches Halstead-Volumen $V = N\log\eta$ pro Modul
$\overline{V(G)}$: Durchschnittliche zyklomatische Komplexität pro Modul
$\overline{L}$: Durchschnittliche LOC pro Modul
$\overline{C}$: Durchschnittlicher Prozentsatz an Kommentarzeilen

Hybride Metriken

Grundidee: Zahlreiche Metriken kombinieren

Beispiel: *Wartbarkeits-Index* (MI) bei Hewlett-Packard:

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

$\overline{V}$: Durchschnittliches Halstead-Volumen $V = N\log\eta$ pro Modul
$\overline{V(G)}$: Durchschnittliche zyklomatische Komplexität pro Modul
$\overline{L}$: Durchschnittliche LOC pro Modul
$\overline{C}$: Durchschnittlicher Prozentsatz an Kommentarzeilen

Je größer der MI, desto besser die Wartbarkeit

Unterschreitet der MI einen gewissen Wert (z.B. 30), wird das Modul restrukturiert

Konkrete Faktoren wurden aus Fragebögen gewonnen

Metriken für OO-Komponenten

In *objektorientierten Programmen* versagt die McCabe-Metrik, da die Kontrollflußkomplexität der meisten Methoden gering ist ($V(G) = 1$).

Metriken müssen deshalb das *Zusammenspiel der Klassen* betrachten – typischerweise anhand des (statischen) Objektmodells

Für *dynamische Aspekte* (z.B. Zustandsautomaten, Sequenzdiagramme) gibt es (noch?) keine Metriken







Signifikante OO-Metriken

DIT (Depth of Inheritance Tree) — # Oberklassen einer Klasse

DIT größer ⇒ Fehlerwahrscheinlichkeit *größer*







Signifikante OO-Metriken

DIT (Depth of Inheritance Tree) — # Oberklassen einer Klasse

DIT größer ⇒ Fehlerwahrscheinlichkeit *größer*

NOC (Number of Children of a Class) — # direkter Unterklassen

NOC größer ⇒ Fehlerwahrscheinlichkeit *geringer*
(Ausnahme: GUI-Klassen)







Signifikante OO-Metriken

DIT (Depth of Inheritance Tree) — # Oberklassen einer Klasse

DIT größer ⇒ Fehlerwahrscheinlichkeit *größer*

NOC (Number of Children of a Class) — # direkter Unterklassen

NOC größer ⇒ Fehlerwahrscheinlichkeit *geringer*
(Ausnahme: GUI-Klassen)

RFC (Response For a Class) — # Funktionen, die direkt durch Klassenmethoden aufgerufen werden

RFC größer ⇒ Fehlerwahrscheinlichkeit *größer*

Signifikante OO-Metriken

DIT (Depth of Inheritance Tree) — # Oberklassen einer Klasse

DIT größer ⇒ Fehlerwahrscheinlichkeit *größer*

NOC (Number of Children of a Class) — # direkter Unterklassen

NOC größer ⇒ Fehlerwahrscheinlichkeit *geringer*
(Ausnahme: GUI-Klassen)

RFC (Response For a Class) — # Funktionen, die direkt durch Klassenmethoden aufgerufen werden

RFC größer ⇒ Fehlerwahrscheinlichkeit *größer*

WMC (Weighed Methods per Class) — # definierter Methoden

WMC größer ⇒ Fehlerwahrscheinlichkeit *größer*

Signifikante OO-Metriken

DIT (Depth of Inheritance Tree) — # Oberklassen einer Klasse

DIT größer ⇒ Fehlerwahrscheinlichkeit *größer*

NOC (Number of Children of a Class) — # direkter Unterklassen

NOC größer ⇒ Fehlerwahrscheinlichkeit *geringer*
(Ausnahme: GUI-Klassen)

RFC (Response For a Class) — # Funktionen, die direkt durch Klassenmethoden aufgerufen werden

RFC größer ⇒ Fehlerwahrscheinlichkeit *größer*

WMC (Weighed Methods per Class) — # definierter Methoden

WMC größer ⇒ Fehlerwahrscheinlichkeit *größer*

CRO (Coupling Between Object Classes) — # Klassen, auf deren Dienste die Klasse zugreift)

CRO größer ⇒ Fehlerwahrscheinlichkeit *größer*







Metriken für OO-Komponenten (3)

OO-Metriken wie DIT, NOC, RFC, WMC, CRO können erneut mit Komplexitätsmetriken wie McCabe oder Halstead kombiniert werden

Das wird auch getan:

- Autoren schlagen neue Kombinationsmetrik vor. . .
- . . . und beschreiben, welche Werte die Metrik berechnet
- Was fehlt, ist die *Validierung:* Mißt die Metrik irgendetwas sinnvolles?







Anwendung: Praktomat

Können wir irgendeine der Metriken für Praktomat einsetzen?

Erfahrung bis jetzt:

— Als *Anhaltspunkt* für Bewerter: *ja*

— Als *Einreichungsbedingung*: *nein*
(zu große Streuung)

Prozeßmetriken

Mit Metriken läßt sich auch der *Entwicklungsprozeß* messen – etwa die aufgewendeten Ressourcen oder die Zahl der Fehler.

Typischerweise geschieht dies mit *Fragebögen:*

Projektname:

Aktivitäten	Fehler eingebracht	Fehler behoben	Fehlerbehebung ausgeschlossen
Definition			
Entwurf			
Implementierung			
Test			
Summen			

„Fehler eingebracht“: Fehler, die der entsprechenden Aktivität entstammen

Prozeßmetriken (2)

Probleme bei Prozeßmetriken:

- Metriken werden nicht mit der gleichen Begeisterung verfolgt wie Entwicklungsaktivitäten
- Gewisser bürokratischer Aufwand
- Metriken, die die *Produktivität* messen, erfordern viel Fingerspitzengefühl (da die erhobenen Daten zur Leistungsbeurteilung eingesetzt werden können)







Ein pragmatischer Meßprozeß

Pragmatischer Ansatz: Statt absoluter Einschätzungen und Projektionen Fokus auf *Anomalien* (= Abweichungen)

Der Meßprozeß läßt sich dann in 6 Abschnitte einteilen:

1. Auswahl der zu messenden Eigenschaft
2. Auswahl der passenden Metrik
3. Auswahl der zu messenden Teile (aus Prozeß/Produkt)
4. Anwenden der Metrik auf die Teile
5. Ungewöhnliche Meßwerte erkennen
6. Ungewöhnliche Meßwerte analysieren

Zahlreiche Werkzeuge verfügbar, die alle Arten von Metriken anbieten







Kritik

Das, was einen eigentlich interessiert, kann man nicht direkt messen (anders als in den Naturwissenschaften) – etwa die Qualität des Produktes

Kritik

Das, was einen eigentlich interessiert, kann man nicht direkt messen (anders als in den Naturwissenschaften) – etwa die Qualität des Produktes

Um dieses Problem zu lösen, stützt man sich auf *Hypothesen:* In einer Formel faßt man die quantitative Beziehung zwischen meßbaren Größen und interessierenden Größen zusammen

Kritik

Das, was einen eigentlich interessiert, kann man nicht direkt messen (anders als in den Naturwissenschaften) – etwa die Qualität des Produktes

Um dieses Problem zu lösen, stützt man sich auf *Hypothesen:* In einer Formel faßt man die quantitative Beziehung zwischen meßbaren Größen und interessierenden Größen zusammen

Modelle sind sehr simpel – offensichtlich *von Managern für Manager*

Wo bleiben Data Mining, Machine Learning, neuronale Netze, Bayes'sche Netze?

Kritik

Das, was einen eigentlich interessiert, kann man nicht direkt messen (anders als in den Naturwissenschaften) – etwa die Qualität des Produktes

Um dieses Problem zu lösen, stützt man sich auf *Hypothesen:* In einer Formel faßt man die quantitative Beziehung zwischen meßbaren Größen und interessierenden Größen zusammen

Modelle sind sehr simpel – offensichtlich *von Managern für Manager*

Wo bleiben Data Mining, Machine Learning, neuronale Netze, Bayes'sche Netze?

Nur sinnvoll, wenn Metrik *validiert* – und zwar nicht nur in der Literatur, sondern in der eigenen Anwendung

Zusammenfassung

- Eine *Software-Metrik* definiert, wie eine Kenngröße eines Software-Produkts oder Software-Prozesses gemessen wird
- Software-Metriken müssen *valide* sein: „Erfüllt das Maß das, was es messen soll?“
- Messen der *Funktionalität* durch Tests: unproblematisch
- Messen der *Komplexität* und der *Wartbarkeit* empfiehlt sich pragmatisch, um *Anomalien* zu entdecken
- Metriken müssen stets auf das Einsatzgebiet angepaßt werden; keine universellen Standards
- Berechnung von Metriken ist kein Ersatz für Gegenlesen, Test oder Verifikation







Literatur

Ressourcen des CMU *Software Engineering Institute*: `http://www.sei.cmu.edu/activities/str/descriptions`
mit zahlreichen weiteren Literaturverweisen

Lehrbuch der Softwaretechnik, Band 2 (Balzert)