# Software-Qualitätsmanagement

**Kernfach Angewandte Informatik**

Sommersemester 2004

Prof. Dr. Hans-Gert Gräbe

## Einführung

Quantitative Aussagen über die Produktqualität einer System-komponente können mit Hilfe von **Metriken** ermittelt werden.

- Mit solchen Metriken sind heute nur einfache Aussagen über Eigenschaften einer Komponente möglich.
- Eine Metrik bewertet ein Software-System immer nur unter einem sehr speziellen Blickwinkel.
- Aussagekräftiger Gesamteindruck von einer Systemkomponente nur durch Auswertung einer Gruppe von Metriken, oft auch nur im Vergleich zu Parametern anderer, bereits im Einsatz befindlicher Komponenten.
- Metriken können nicht nur für bereits implementierte Komponenten, sondern auch schon entwicklungsbegleitend eingesetzt werden.

## 3. Metriken für Komponenten

### Metriken zum Erfassen der prozeduralen Komplexität

- **Umfangsmetriken**
  - sind die ältesten Metriken
  - stellen ab auf die textuelle Komplexität
  - verwenden einfach verfügbare Informationen (Anzahl an Programmzeilen, Dateigröße, Zahl der Funktionen, ...)
  - Vertreter: Halstead-Metrik, Function Points (zur Erfassung des Umfangs verbaler Anforderungen)

- **Logische Strukturmetriken** (Kontrollfluss-Metriken)
  - Analyse des Kontrollfluss-Graphen
  - Wird samt seiner Begleitobjekte (Symboltabelle) sowieso vom Compiler ausgewertet
  - Vertreter: McCabe-Metrik

## 3. Metriken für Komponenten

- **Datenstrukturmetriken**
  - messen die Anzahl an Variablen, deren Gültigkeit und Lebensdauer sowie die Referenzierung der Variablen
- **Stilmetriken**
  - messen ob die Programme richtig eingerückt wurden und ob die Namenskonventionen eingehalten wurden
- **Interne Bindungsmetriken**
  - messen die syntaktische Bindung durch Prüfen des Codes jeder Komponente

## Umfangsmetriken – Die Halstead-Metrik

- Misst die textuelle Komplexität eines Programms, indem die Zahl der verwendeten Funktionen und der verwendeten Variablen ins Verhältnis gesetzt werden.
  - Es wird jeweils die Gesamtzahl (Programmtext) und die Zahl verschiedener Objekte (Symboltabelle) bestimmt.
  - $\mathbf{?_1}$, $\mathbf{N_1}$ = Zahl der (verschiedenen) Funktionen, Operatoren, Symbole oder Schlüsselwörter (z. B.: +, -, *, /, **while**, **if**, ...)
  - $\mathbf{?_2}$, $\mathbf{N_2}$ = Zahl der (verschiedenen) Variablen, Operanden ...
- **Interpretation**:
  - $\mathbf{?} = ?_1 + ?_2$: Größe des Vokabulars
  - $\mathbf{N} = N_1 + N_2$: Länge der Implementierung

## 3.1. Die Halstead-Metrik

- **Vorteile:**
  - einfach zu ermitteln,
  - bei jeder Programmiersprache verwendbar und
  - gute Eignung der Metriken für die zu messenden Größen
- **Nachteile:**
  - nur der Implementierungsaspekt betrachtet und
  - Mehrdeutigkeiten im Messansatz, z. B. bei den Klassifikationsregeln für Operatoren und Operanden

- **abgeleitete Größen**:
  - **D** = $?_1/2 \cdot N_2 /?_2$, Parameter für die Schwierigkeit, den Code zu verstehen
  - Interpretation: $N_2 /?_2$ = durchschnittliches Vorkommen jeder Variablen, $?_1$ = Anzahl der verwendeten Funktionen

## 3.1. Die Halstead-Metrik

```
int ZaehleVokale(String s) {
    int VokalAnzahl; char Zchn; int i;
    for(i=0; i < s.length(); i++) {
        Zchn=s[i];
        if ((Zchn == 'A') || (Zchn == 'E') || (Zchn == 'I') ||
            (Zchn == 'O') || (Zchn == 'U')) VokalAnzahl++;
    }
    return VokalAnzahl;
}
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ZaehleVokale | 1 | **int** | 3 | () | 9 | ++ | 2 |
| VokalAnzahl | 3 | **String** | 1 | {} | 2 | [] | 1 |
| Zchn | 7 | **char** | 1 | ; | 8 | == | 5 |
| s | 3 | **for** | 1 | = | 2 | \|\| | 4 |
| i | 5 | **if** | 1 | < | 1 | . | 1 |
| Konstanten (6) | 6 | **return** | 1 | length | 1 | | |

$?_1=17$, $N_1=44$, $?_2=11$, $N_2=25$, $D=19.32$

## Kontrollfluss-Metriken – Die McCabe-Metrik

- Misst die strukturelle Komplexität eines Programms, indem eine Grapheninvariante, die **zyklomatische Zahl** V(G) des Kontrollflussgraphen, bestimmt wird.
- $V(G) = e - n + 2p$ mit
  - e = Anzahl der Kanten des Graphen
  - n = Anzahl der Knoten
  - p = Anzahl der Zusammenhangskomponenten
- Kontrollflussgraph wird für jede Prozedur aufgestellt (p=1).
- Für solchen Graphen gilt

$$\mathbf{V(G) = b + 1}$$

  mit **b** = Anzahl der Bedingungen.
- Zyklomatische Zahl ist additiv auf Komponenten.
- Lineare Teilstücke können zusammengezogen werden.

## 3.2 McCabe-Metrik

- **Vorteile:**
  - einfach zu berechnen,
  - grobes Maß für die Kontrollflusskomplexität: je größer, desto weiter weicht der Kontrollfluss vom linearen (V(G)=1) ab.

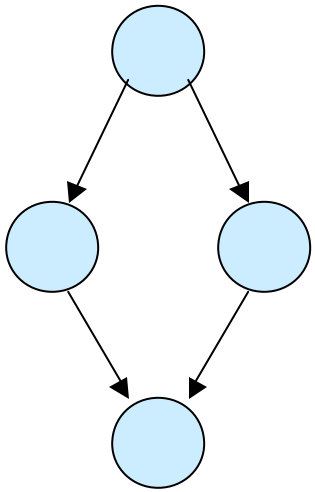
- **Nachteile:**
  - unterschiedliche Programmmerkmale werden stark vereinfacht
  - Quellprogramm als zentrales Messobjekt überbetont
  - Es wird nur das Programmgerüst, nicht aber die Komplexität einzelner und verschachtelter Anweisungen berücksichtigt

## 3.2 McCabe-Metrik - Beispiel

Sequenz

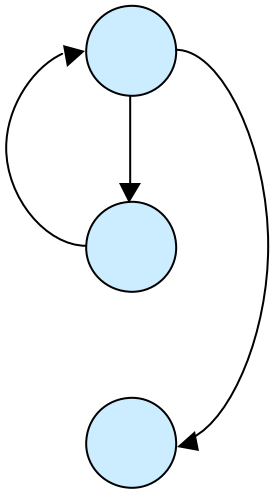
V(G) = 1 – 2 + 2 = 1

Auswahl



V(G) = 4 – 4 + 2 = 2

Abweisende Schleife



V(G) = 3 – 3 + 2 = 2

- das Programm ZaehleVokale hat die zyklomatische Zahl

  V(G) = 7 – 6 + 2 = 3

  ZaehleVokale enthält zwei Bedingungen ➔ V(G) = 3

## Einführung

- Metriken der klassischen Software-Entwicklung sind in unveränderter Form für OO-Projekte nur bedingt aussagefähig.
  - Umfangsmetriken: Wie mit geerbtem Code umgehen?
    - Durch Vererbung und Polymorphismus sinkt die Zeilenzahl signifikant
  - McCabe-Metrik: Kontrollflusskomplexität bei OO meist sehr gering
- Frage nach Maßen für die OO-spezifischen Effekte
  - zusätzliche Maße waren erforderlich
    - Breite und Höhe der Vererbungshierarchie
    - Anzahl der Klassen, die eine spezielle Operation erben
    - Anteil wieder verwendeter Komponenten
    - Anzahl der Objekt- und Klassenattribute
    - Anzahl der Objekt- und Klassenoperationen

## Typische Metriken (Beispiele)

- Objekt- und Klassenattribute einer Klasse:
  - Anzahl: |OV|, |CV|
  - gewichtete Anzahl: $\sum T(v)$
    - T(v) = Gewicht des Typs des Attributs v
    - gewichtet nach Zahl der Vorkommen, nach Zahl der Vorkommen in verschiedenen Methoden der Klasse etc.
- Objekt- und Klassenmethoden einer Klasse:
  - |OM|, |CM|, evtl. wieder gewichtet nach Komplexität
  - Parameterkomplexität einer Methode
    - Zahl und Gewicht der Typen der Aufrufparameter
  - McCabe-Metrik
- durchschnittliche Komplexität von Klassen in einem Paket
  - Durchschnittswerte der einzelnen Klassenparameter
  - Kreuzreferenzparameter (Bindungsanalyse im Paket)

## 3.3 Metriken für objektorientierte Komponenten



Folgende Metriken wurden als signifikant befunden
[Basili, Briand, Melo 1996], [Chidamber, Kemerer 1994]

- DIT (*Depth of Inheritance Tree*):
  - Tiefe des Vererbungsbaumes (Zahl der Vorfahren einer Klasse)
  - je höher der Wert von DIT, desto höher die Fehlerwahrscheinlichkeit (Hoher Nachnutzungsgrad, Nichtbeachtung verdeckter Annahmen)

- NOC (*Number of Children of a Class*):
  - Anzahl der direkten Nachfolger einer Klasse
  - je höher der Wert von NOC, desto geringer die Fehlerwahrscheinlichkeit (sehr präzise Abstraktion, Nichtvorhandensein verdeckter Annahmen)

- RFC (*Response For a Class*):
  - Anzahl der Funktionen, die direkt durch die Methoden einer Klasse aufgerufen werden.
  - je höher der Wert von RFC, desto größer die Fehlerwahrscheinlichkeit (Hoher Delegationsgrad, Nichtbeachtung verdeckter Annahmen)

- WMC (*Weighted Methods per Class*):
  - Anzahl aller neu definierten oder überschriebenen Methoden, die in jeder Klasse definiert sind.
  - Geerbte Methoden werden nicht gezählt.
  - je größer der Wert von WMC, desto höher die Fehlerwahrscheinlichkeit (??)

- CBO (*Coupling Between Object Classes*):
  - eine Klasse ist mit einer anderen Klasse gekoppelt, wenn sie deren Methoden und/oder Attribute benutzt.
  - CBO ist die Anzahl der Klassen, mit der eine Klasse gekoppelt ist.
  - je größer der Wert von CBO, desto größer die Fehlerwahrscheinlichkeit (höherer Verschränkungsgrad)

## 3.3 Metriken für objektorientierte Komponenten

**Vorteile**:

- erste Ansätze zur Verbesserung objektorientierter Komponenten
- erste empirische Untersuchungen zeigen die Eignung einiger Metriken als Qualitätsindikatoren

**Nachteile**:

- die Ziele der Metriken sind nur implizit erkennbar
  - Zielrelevanz der Metriken noch ungenügend untersucht
- keine Metriken für dynamische Aspekte
  - z. B.: Zustandsautomaten
- keine semantische Unterscheidung der Methoden
  - Standardoperationen (lesen, schreiben, erzeugen,...) sind weniger fehleranfällig als auszuprogrammierende "fachkonzeptspezifische" Methoden
- keine Berücksichtigung der Oberklassenqualität
  - ob eigene, geerbte oder fremde Methoden
  - ob Vererbung von Methoden aus Standardklassen oder eigenen
- keine Metriken, die eine "gute" Vererbungsstruktur prüfen

## 3.4 Anomalienanalyse

**Begriff**:
Eine **Anomalie** ist jede Abweichung bestimmter Eigenschaften eines Programms von der korrekten Ausprägung dieser Eigenschaften.

- Konstruktive Sprachkonzepte erlauben das Aufdecken von Anomalien durch statische Quelltextanalyse
  - Beispiel: Typprüfung durch den Compiler
  - Oft ist keine unmittelbare Fehlererkennung, jedoch eine Identifikation von Fehlerort und -symptomen möglich.

- **Datenflussanomalien-Analyse**:
  - **Ziel**: Aufdecken von Datenflussanomalien durch Analyse von Programmpfaden auf sinnvolle Datenfluss-Sequenzen

## 3.4 Anomalienanalyse

- **Datenfluss-Eigenschaften von Variablen**
  Auf eine Variable x kann entlang eines Programmpfades wie folgt zugegriffen werden:
  - x wird definiert (d),
  - x wird referenziert (r),
  - x wird undefiniert (u) (z.B. beim Verlassen einer Methode)
  - x wird „geleert“ (e), d.h. der Wert an einen anderen Ort übertragen
- Enthält die Sequenz Teile die keinen Sinn ergeben, so liegt eine Datenfluss-Anomalie vor.
  - Beispiele:
    - **r**dru ➔ die Sequenz beginnt mit einer Referenz, vor der Definiton, diese Anomalie ist vom Typ **ur**
    - **dd**r**du** ➔ diese Sequenz beginnt mit einer doppelten Definition (Anomalientyp dd) und endet mit **du**

## 3.4 Anomalienanalyse

### Beispiel

```
/* swap (int a, int b) */
   int hilf; a=hilf; a=b; hilf=b;
```

| Analyse: | | Anomalietyp: |
|---|---|---|
| a | d : d d : r | mehrfach nacheinander überschrieben |
| b | d : r r : r | nie verändert |
| hilf | u : r d : e | neu definiert vor e |

```
/* swap (int a, int b) korrigierte Version */
   int hilf; hilf=b; b=a; a=hilf;
```

| | |
|---|---|
| a | d : r d : r |
| b | d : r d : r |
| hilf | u : d r : e |

## 3.4 Anomalienanalyse

**Leistung:**
entdeckt Wertzuweisungen an falsche Variablen, Anweisungen an unkorrekter Stelle und fehlende Anweisungen

- **Vorteile:**
  - sichere Entdeckung bestimmter Fehlertypen,
  - geringer Aufwand im Vergleich zu dynamischen Verfahren,
  - direkte Fehlerlokalisierung und
  - gute Ergänzung zu anderen Testverfahren
- **Nachteile:**
  - Leistungsfähigkeit auf schmalen Fehlerbereich begrenzt

## Ergänzung zum Kapitel
## 5. Testende Verfahren
## 5.9. Test von Klassen

Im [Balzert] wenig systematisch aufbereitet, dort im Kap. 5.13.

- Alle bisherigen Testverfahren waren auf den funktionalen Test von Methoden unter dem imperativen Paradigma ausgerichtet.
- Kommunikation zwischen den Methoden desselben Objekts erfolgt sowohl über die Aufrufparameter als auch den Zustand der Objektattribute (Objekt ist immer implizit ein Parameter).

**Kleinste sinnvolle Testeinheit im OO-Bereich ist also die Klasse**

## Weitere Besonderheiten von Tests im OO-Bereich

- Wiederverwendbarkeitskonzept
  - Einsatzzweck von Klassen oft nicht genau umrissen
  - Allgemeinheit führt zu vielen möglichen Testfällen
- Vererbung von Attributen und Methoden
  - Redundanz wird eliminiert zu Lasten von zusätzlichen Abhängigkeiten
- Polymorphismus und dynamische Bindung
  - neue Testverfahren nötig
  - Test jeder möglichen Bindung für Polymorphismus nötig
- folgende Arten von Klassen sind zu unterscheiden:
  - normale Klassen
  - abstrakte Klassen
  - parametrisierte Klassen

## 9. Testen von Klassen

### Testen normaler Klassen:

1. Erzeugung einer instrumentierten Instanz der zu testenden Klasse.
2. Überprüfung jeder einzelnen Operation für sich.
   - ° zunächst Operationen, die den Objektzustand nicht ändern, anschließend die zustandsverändernden Operationen
   - ° Testfälle wie besprochen herleiten und Tests aufsetzen
   - ° Zustandsräume sind lokal an Objekte gebunden. Initialisierung und Auswertung des Tests erfolgt deshalb am Objekt.
3. Test jeder Folge abhängiger Operationen in der gleichen Klasse.
   - Alle potenziellen Verwendungen einer Operation sollten unter allen praktisch relevanten Bedingungen ausprobiert werden.
   - In den Tests muss jede Objektausprägung (bzw. wenigstens Äquivalenzklassen von Ausprägungen) simuliert werden.
   - ° Existiert Objektlebenszyklus, dann Zustands- und Zustandsübergangs-Überdeckungstests

## 9. Testen von Klassen

4. Anhand der Instrumentierung prüfen, wie die Testüberdeckung aussieht. Fehlende Überdeckungen durch zusätzliche Testfälle abdecken.
   - ° bereits beschriebenes klassisches Vorgehen

**Testen abstrakter Klassen:**

- Aus einer abstrakten Klasse muss eine konkrete Klasse gemacht werden
- bei der Realisierung abstrakter Operationen ist die leere oder eine einfache, die Spezifikation erfüllende Implementierung zu wählen.

**Testen parametrisierter Klassen (Template-Klassen, C++)**:

- Zunächst eine möglichst einfache konkrete Klasse erzeugen
- Parameter so wählen, dass der Test möglichst einfach wird

## 9. Testen von Klassen

### Testen von Unterklassen:

Besondere Gesichtspunkte beim Testen von Unterklassen:

- Alle Testfälle für geerbte und **nicht redefinierte** Operationen der Oberklasse müssen erneut ausgewertet werden.
  - Unterklasse definiert neuen Kontext
- Für **redefinierte** Operationen sind vollständig neue strukturelle Testfälle zu erstellen.
  - redefinierte Operation hat neue Implementierung
- Für **redefinierte** Operationen müssen die alten funktionalen Testfälle ausgewertet und durch neue ergänzt werden.
  - Instanzen der Unterklasse sind spezielle Instanzen der Oberklasse
  - Zusätzlich muss die neue Semantik der redefinierten Operation getestet werden.