

Softwaretechnik und Programmierparadigmen

VL10: Metriken

Prof. Dr. Sabine Glesner
FG Programmierung eingebetteter Systeme
Technische Universität Berlin

Software-Metriken: Motivation

- **Metrik**

- misst die Komplexität von Software
- weist Programmen Zahlen zu, mit dem Ziel, sie vergleichbar zu machen (Kenngröße)

Misst auch die Qualität von Software

- **zur Bewertung von Software**

- Hoffnung: gemessene Größe in Relation zur Qualität



- **zur Steuerung des Software-Entwicklungsprozesses**

- zur Qualitätsverbesserung
- zur Abschätzung des Aufwands und der Kosten

Warum Messen helfen kann

- **Historisch:** Galileo Galilei (1564-1642):
Miß alles, was sich messen läßt, und mach alles meßbar, was sich nicht messen läßt.
- **In der Management-Theorie:** Peter Drucker (1909-2005), Ökonom:
Was man nicht messen kann, kann man nicht lenken.
- **Im Software-Engineering:** Tom DeMarco (*1940):
Erster Satz aus seinem Buch „Controlling Software Projects: Management, Measurement, and Estimation“, Prentice Hall 1982.
You can't control what you can't measure.

Überblick

- Definition von Software-Metriken
- Anforderungen an Software-Metriken
- Überblick über Metriken
- Software-Metriken im Detail
 - Zeilenmetriken
 - Halstead-Metriken
 - Zyklomatische Komplexität
 - Objektorientierte Metriken
- Metriken: Maß aller Dinge?

Software-Metriken: Definitionen

Eine Softwaremetrik ist jede Art von Messung, die sich auf ein Softwaresystem, einen Prozess oder die dazugehörige Dokumentation bezieht.

(aus: Ian Sommerville, Software Engineering)

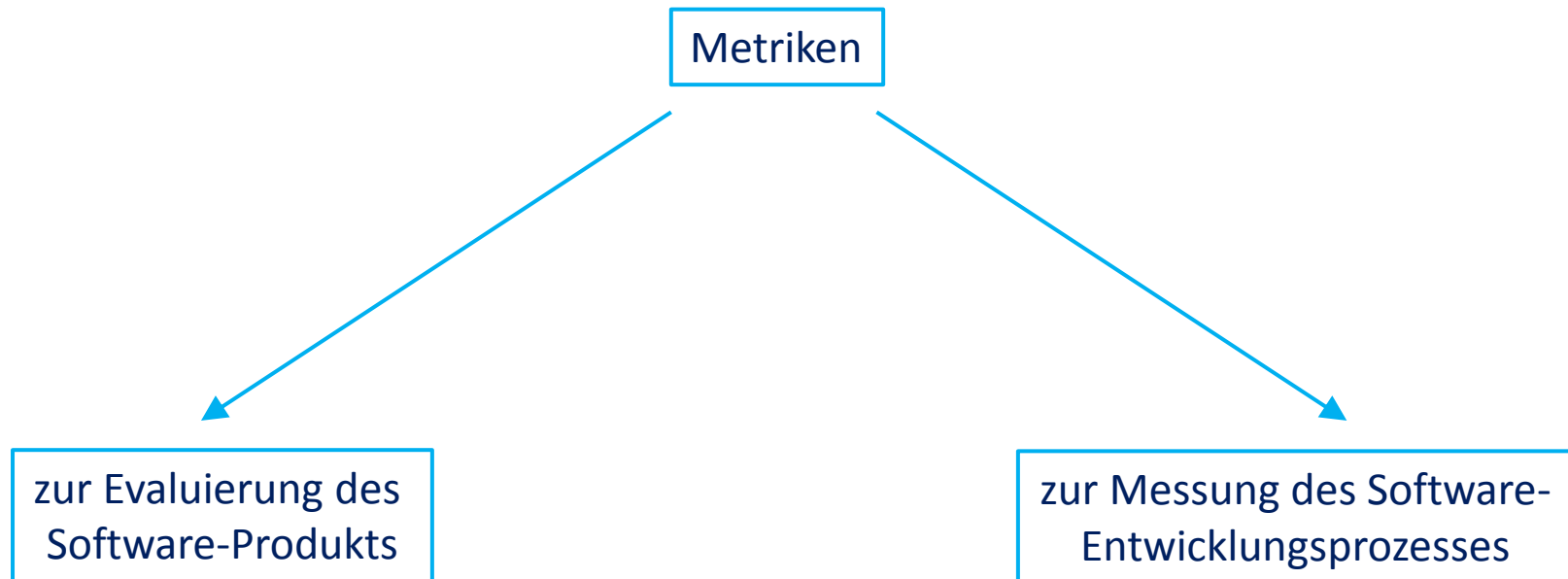
Definition des IEEE Standard 1061 (1992):

Eine Softwaremetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit.

Anforderungen an Software-Metriken

- **Validität:** sinnvoll, misst das Richtige
- **Objektivität:** kein Einfluss durch den Messenden
- **Zuverlässigkeit:** Ergebnis für dieselbe Messung immer gleich
- **Normierung:** Skala existiert, die Messergebnisse einordnet
- **Vergleichbarkeit:** mit anderen Maßen in Relation
- **Ökonomie:** Messung nicht zu teuer
- **Nützlichkeit:** hilft in der Praxis

Zwei Arten von Software-Metriken

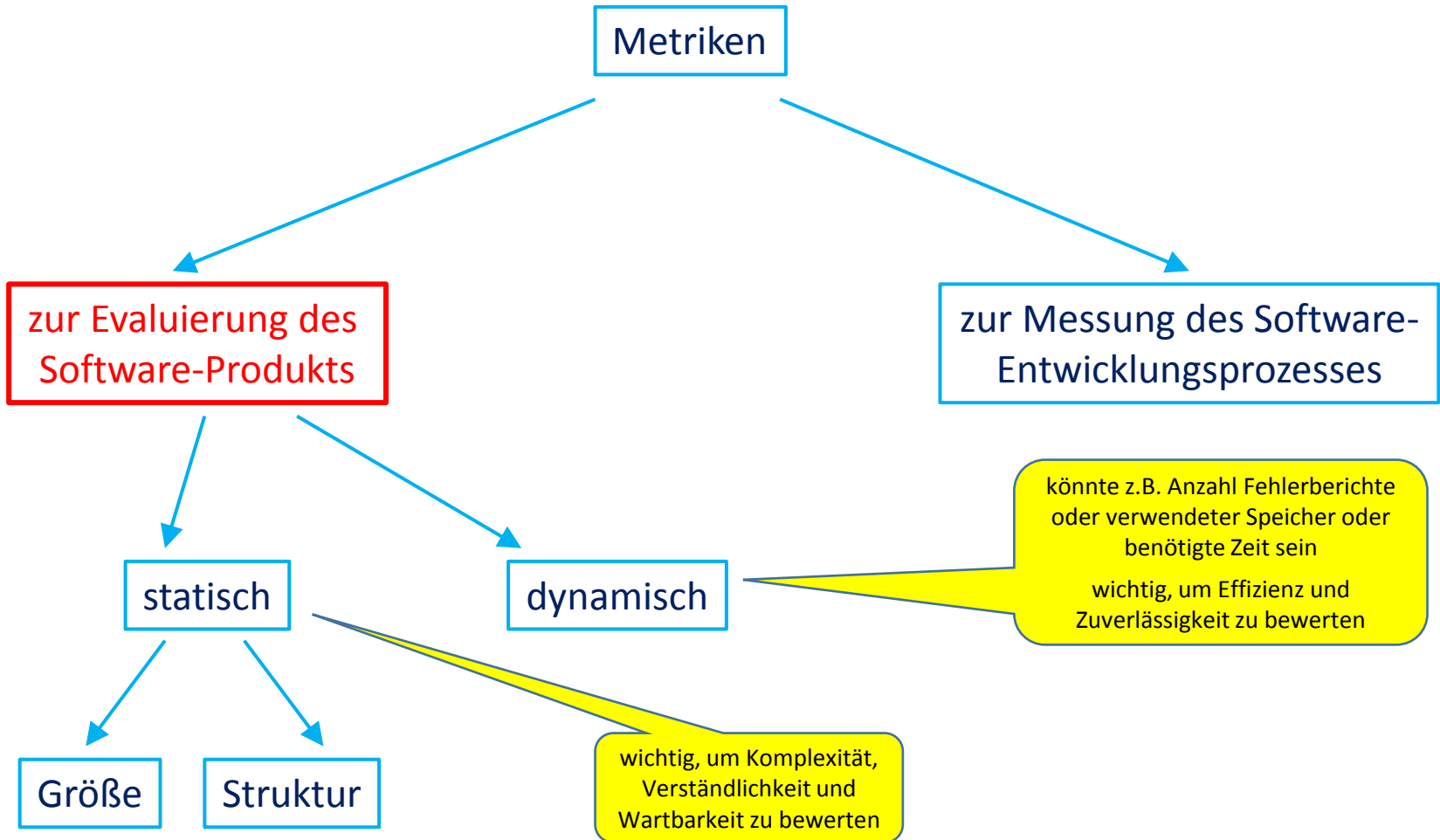


Metriken für Softwareentwicklungsprozesse

- zur Messung nicht der Software selbst, sondern ihrer Entwicklung
 - z.B. dafür benötigte Ressourcen (Personentage, Reisekosten, Computerressourcen etc.)
 - oder Häufigkeit bestimmter Ereignisse (Anzahl gefundener Fehler bei Programminspektionen, Anzahl Anforderungsänderungen und daraus resultierender Programmänderungen etc.)

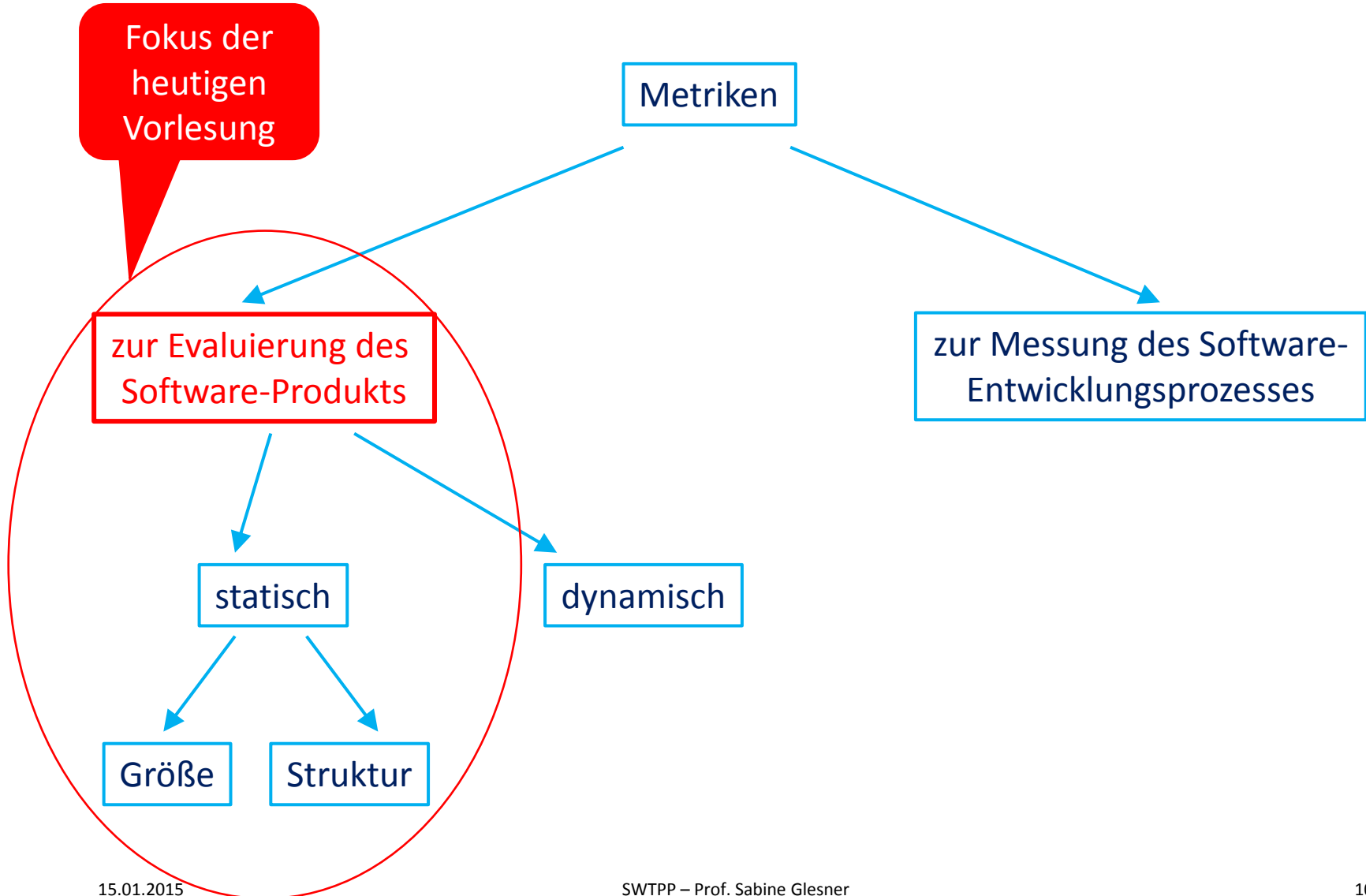
Viele Änderungen der Anforderungen sind schlecht, weil das bedeutet, dass Anforderungen am Anfang nicht sorgfältig erfasst wurden
- Maß für Produktivität
 - kann auch eingesetzt werden, um Leistungsfähigkeit von Teams oder einzelnen Programmierern zu bewerten
- nicht im Fokus dieser Vorlesung

Zwei Arten von Software-Metriken



Zwei Arten von Software-Metriken

Fokus der
heutigen
Vorlesung



Statische Produktmetriken

- Unterscheidung zwischen
 - traditionellen Metriken, unterteilt in
 - Metriken zur Messung der Programmgröße und dessen Komplexität
 - Zeilenmetriken (loc) und Halstead-Metriken
 - Metriken zur Messung der Programmstruktur
 - McCabe Cyclomatic Number
 - objektorientierten Metriken
 - Verhältnisse der einzelnen Elemente (Klassen, Methoden) untereinander

Zeilenmetriken

- **lines of code (loc)**

- zähle Zeilen Code im Programm

```
#include <stdio.h>
int main(void)
{
    printf("Hello, World\n");
    return 0;
}
```

6 Zeilen

- Verfeinerungen von loc: **non commenting lines of code (NCSS)**

- ignoriere Leerzeilen und reine Kommentarzeilen = 4 Zeilen (?)
- Anteil an Kommentarzeilen:
 - sollte zwischen 30% und 75% liegen

Problem: manipulierbar (mehr oder weniger Zeilen erzeugen
aber: Konventionen geben Leerzeilen und Anzahl Instruktionen
pro Zeile vor und ermöglichen bessere Vergleichbarkeit)

- Typische Werte für loc:

- Länge einer Funktion zwischen 4 und 40 Zeilen
- Länge einer Datei zwischen 40 und 400 Zeilen (10-100 Funktionen)

Zeilenmetriken: Vor- und Nachteile

- Vorteil:
 - leicht zu berechnen
 - leicht nachzuvollziehen
- Nachteil:
 - wenig aussagekräftig
 - abhängig vom Programmierstil
 - bessere Programmstruktur kann auch weniger Zeilen bedeuten

Halstead-Metriken

1960er ALGOL-Sprachen verbreitet:
strikt imperativ

-> Ursprung der Halstead-Metrik

- 1977 durch Maurice Halstead eingeführt
- textuelle bzw. lexikalische Komplexität
- Programmcode als Sequenz von
 - Operatoren und
 - Operanden

Halstead Metriken: Definition

- Anzahl unterschiedlicher Operatoren: n_1
- Anzahl unterschiedlicher Operanden: n_2 in der Programmiersprache
- Anzahl Operatoren im Programm: N_1
- Anzahl Operanden im Programm: N_2
- **Größe des Vokabulars**: $n = n_1 + n_2$
- **Länge** des Programms: $N = N_1 + N_2$
- **Volumen** des Programms: $V = N * \log_2(n)$
(Anzahl Bits, um Programm darzustellen)
- **Schwierigkeit (difficulty)**, um Programm zu verstehen: $D = (n_1/2) * (N_2/n_2)$
(N_2 / n_2 : durchschnittliches Auftreten der Operanden)
- **Aufwand (effort)**, um Programm zu verstehen: $E = D * V$

Halstead Metriken: Definition

```
main()
{
    int a, b, c, avg;
    scanf("%d %d %d", &a, &b, &c);
    avg = (a + b + c) / 3;
    printf("avg = %d", avg);
}
```

- Anzahl unterschiedlicher Operatoren: n_1
- Anzahl unterschiedlicher Operanden: n_2
- Anzahl Operatoren im Programm: N_1
- Anzahl Operanden im Programm: N_2
- **Größe des Vokabulars:** $n = n_1 + n_2$
- **Länge** des Programms: $N = N_1 + N_2$
- **Volumen** des Programms: $V = N * \log_2(n)$
(Anzahl Bits, um Programm darzustellen)
- **Schwierigkeit (difficulty)**, um Programm zu verstehen:
 $D = (n_1/2) * (N_2/n_2)$
(N_2 / n_2 : durchschnittliches Auftreten der Operanden)
- **Aufwand (effort)**, um Programm zu verstehen: $E = D * V$

Achtung:

Unterscheidung Operand / Operator
nicht immer eindeutig.
Am besten für eigenes Projekt und
verwendete Programmiersprache
definieren.

Operatoren sind main(), {}, int, scanf, &, =, +, /, printf

Operanden sind a, b, c, avg, "%d, %d, %d", 3, "avg = %d"

$$n_1 = 10$$

$$n_2 = 7$$

$$n = 17$$

$$N_1 = 16$$

$$N_2 = 15$$

$$N = 31$$

$$V = 31 * \log 17 = 126,7$$

$$D = 10/2 * 15/7 = 10,7$$

$$E = 10,7 * 126,7 = 1.355,7$$

recht hoher Wert,
Skala ist nach oben
offen

(Beispiel siehe Wikipedia)

Halstead Metriken: Vor- und Nachteile

- Vorteile:

- leicht zu berechnen
- in Studien nachgewiesen:
korrespondiert mit echter Komplexität

Problem der Studien:
Kaum Zugang zu Code aus der
Wirtschaft

- Nachteile

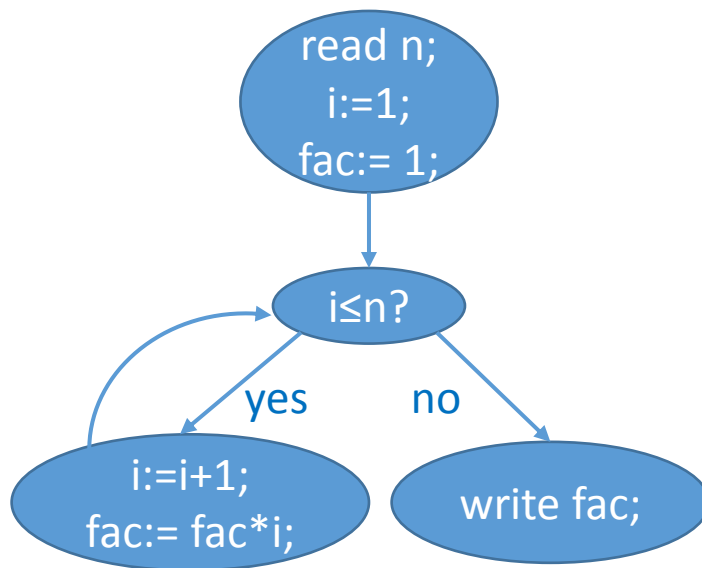
- Struktur und Anzahl Programmpfade unberücksichtigt
- Konzepte moderner Programmiersprachen
unberücksichtigt
(Namensräume, Sichtbarkeiten, Vererbungen, ...)
- Aufteilung in Operatoren vs. Operanden nicht immer
möglich

Strukturmetriken:

Zyklomatische Komplexität von McCabe

- eingeführt 1976 von Thomas McCabe
- abgekürzt als $v(G)$
- definiert als
 - Anzahl binärer Verzweigungen + 1
 - Anzahl konditioneller Zweige im Steuerflussgraphen des Programms
- je höher, desto komplexer das Programm, desto mehr Testfälle nötig
- zyklomatische Zahl > 10 : Fehler nehmen stark zu

Zyklomatische Komplexität



$v(G) \leq 10$: einfache Programme

$v(G) \geq 50$: sehr bzw. zu komplexe

Programme, kaum noch zu testen

- Definition:

- e: Anzahl Kanten im Graphen
- n: Anzahl Knoten im Graphen
- p: Anzahl Komponenten
- $v(G) = e - n + 2 * p$

Zusammenhangskomponenten (sollte immer $p=1$ sein, sonst enthält der Code unerreichbare Stellen)

- für die Fakultätsfunktion:

- $e = 4$
- $n = 4$
- $p = 1$
- $v(G) = 4 - 4 + 2 * 1 = 2$

Wie berechnet man die Anzahl an Verzweigungen?

- Definition:
 - e: Anzahl Kanten im Graphen
 - n: Anzahl Knoten im Graphen
 - p: Anzahl Komponenten
 - $v(G) = e - n + 2 * p$
- keine Verzweigung: n und ~~e-1~~^{e+1} gleich
 - der Knoten ohne ausgehende Kante ist der, in dem Komponentenberechnung terminiert
- e-n ist gleich „Anzahl Entscheidungen – 1“
- wenn $p=1$, dann $e - n + 2 * p = \text{Anzahl Entscheidungen} - 1 + 2$

Zyklomatische Komplexität: Vor- und Nachteile

- Vorteile:
 - leicht zu berechnen
 - in Fallstudien: gute Korrelation zwischen zyklomatischer Komplexität und Programmverständlichkeit
 - zur Testplanung: alle Bedingungen überdecken
- Nachteile:
 - berücksichtigt Steuer-, aber nicht Datenfluss
 - schwierig für objektorientierte Software mit vielen einfachen Zugriffsmethoden (z.B. Attribute)

Weitere Metriken

- NBD: Verschachtelungstiefe (nested block depth)
- NST: Number of Statements (sollte <50 sein)
(entspricht ungefähr der Anzahl Semikolons in Java-Programmen)
- NFC: Number of Function Calls (sollte pro Funktion <5 sein)
- NOM: Number of Methods

Überblick

- Definition von Software-Metriken
- Anforderungen an Software-Metriken
- Überblick über Metriken
- Software-Metriken im Detail
 - Zeilenmetriken
 - Halstead-Metriken
 - Zyklomatische Komplexität
 - **Objektorientierte Metriken**
- Metriken: Maß aller Dinge?

Warum braucht man spezielle OO-Metriken?

- Eigentlich könnte man klassische Metriken auch anwenden, aber:
 - objektorientierte Aspekte (Vererbung, Polymorphie, Klassen) dann nicht erfasst
- Trotzdem: klassische Metriken können auf einzelne Methoden angewendet werden.

Objektorientierte Metriken I

- **Depth of Inheritance Tree (DIT)**
 - maximaler Abstand von der Wurzel der Klassenhierarchie zur Klasse
 - Wahrscheinlichkeit für Fehler größer, wenn DIT größer, weil
 - Komplexität größer, Code schwerer verständlich
 - Testaufwand größer
 - aktuelle Klasse selbst schwerer wiederverwendbar
- **Number of Children (NOC):**
 - Anzahl direkter Subklassen
 - nicht immer eindeutig interpretierbar
 - interpretierbar als Fortpflanzungswahrscheinlichkeit für Fehler
 - Fehlerwahrscheinlichkeit geringer, wenn NOC größer (inverses Maß)

Objektorientierte Metriken II

- **Response for a Class (RFC):**
 - Anzahl der Methoden, die evtl. direkt aufgerufen werden, wenn ein Objekt der Klasse eine eingehende Methode ausführt
 - Fehlerwahrscheinlichkeit steigt mit RFC-Wert
- **Weighted Methods per Class (WMC):**
 - Anzahl der Methoden einer Klasse, kann gewichtet werden nach Größe oder Komplexität
 - je größer WMC, umso größer die Fehlerwahrscheinlichkeit

Objektorientierte Metriken III

- **Coupling Between Objects (CBO):**
 - Anzahl Klassen, mit denen eine Klasse gekoppelt ist
 - hoher Kopplungsgrad erhöht Fehlerwahrscheinlichkeit
 - niedriger Kopplungsgrad zeigt bessere Wiederverwendbarkeit an
- **Lack of Cohesion in Methods (LCOM):**
 - Anzahl Methodenpaare in einer Klasse ohne gemeinsame Instanzvariablen
 - hohe Kohäsion zeigt gute Kapselung innerhalb einer Klasse an, reduziert Programmkomplexität
 - niedrige Kohäsion: Programmstruktur kann verbessert werden, z.B. durch Aufteilung in mehrere Klassen

Objektorientierte Metriken IV

- Metriken zur Beurteilung von Paketen in Java
 - Anzahl an Klassen und Interfaces
 - Afferent Coupling (Ca): eingehende Abhängigkeiten; Anzahl Klassen in anderen Packages, die von Klassen im vorliegenden Package abhängen
 - Efferent Coupling (Ce): ausgehende Abhängigkeiten; Anzahl Klassen in anderen Packages, von denen Klassen im vorliegenden Package abhängen
 - Abstractness (A): Anteil der Klassen und Interfaces, die abstrakt sind
 - ...

Überblick

- Definition von Software-Metriken
- Anforderungen an Software-Metriken
- Überblick über Metriken
- Software-Metriken im Detail
 - Zeilenmetriken
 - Halstead-Metriken
 - Zyklomatische Komplexität
 - Objektorientierte Metriken
- **Metriken: Maß aller Dinge?**

Sind Metriken wirklich das Maß aller Dinge?

- **Alternativen: z.B. maschinelles Lernen**, um aus statischen Merkmalen auf dynamisches Verhalten zu schließen
- **Erfolgreiche Software-Projekte ohne Steuerung bzw. Kontrolle:**
 - Open Source Projekte, Wikipedia, GoogleEarth, Leo, Guttenplag ...
- Tom DeMarco: „**You can't control what you can't measure.**“ (1982)
- Tom DeMarco: „**Software Engineering: An idea whose time has come and gone?**“, IEEE Software 2009.
 - besser: Kosten von Software im Verhältnis zu ihrem Nutzen betrachten
 - bezweifelt, dass Metriken für jede Software-Entwicklung notwendig sind
 - Software Engineering oft experimentell, vor allem, wenn durch Software Dinge verändert werden
 - z.B. die Firma, die Art der Geschäftsabwicklung, die Welt

Literatur

- Ian Sommerville: Software Engineering, Pearson, 2007.
- Helmut Balzert: Lehrbuch der Softwaretechnik: Softwaremanagement, Spektrum Akademischer Verlag, 2008.
- Tom DeMarco: Software Engineering: An idea whose time has come and gone?, IEEE Software, 2009.