

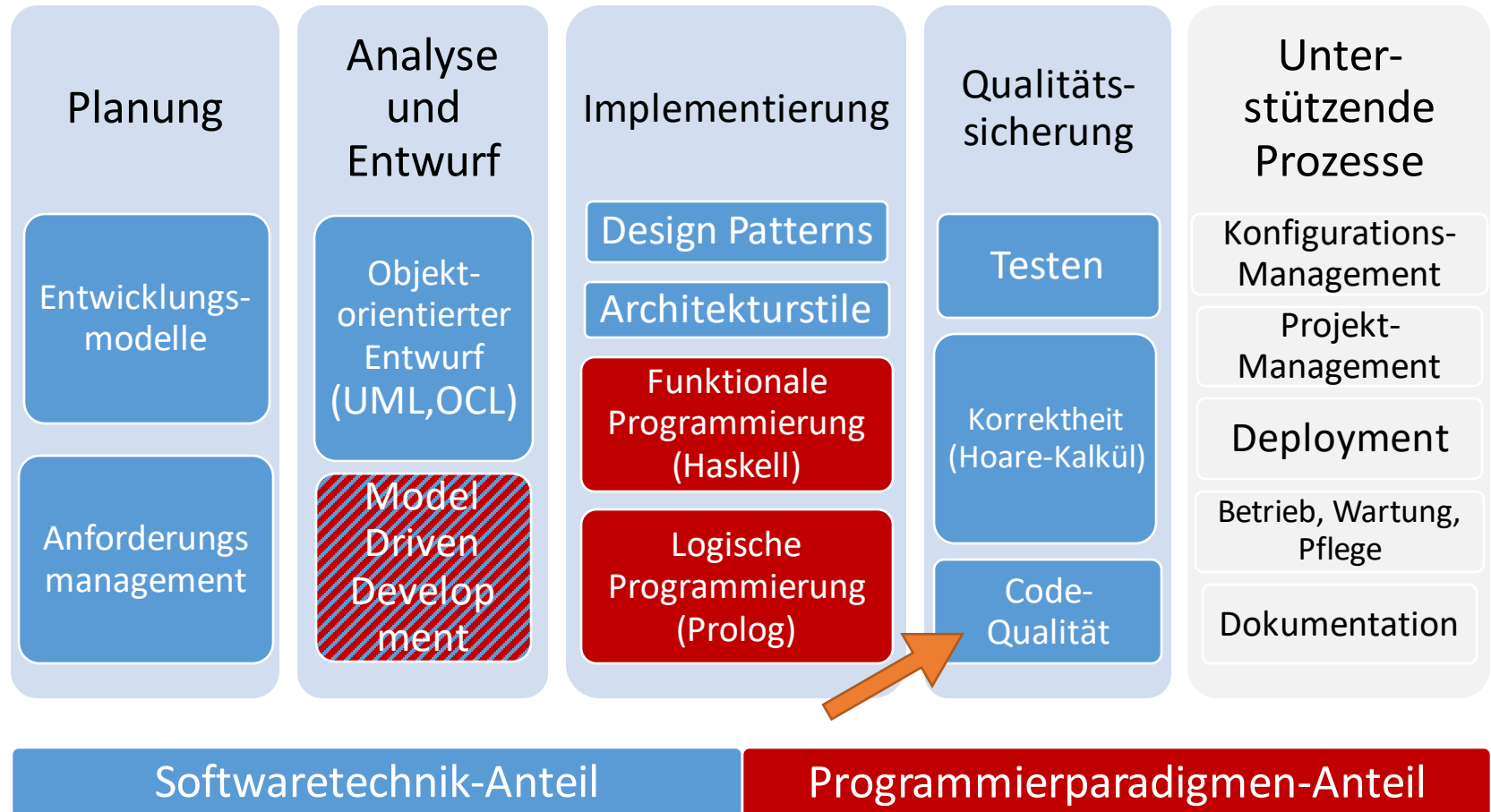
# Softwaretechnik und Programmierparadigmen

## 04 Code-Qualität

---

Prof. Dr. Sabine Glesner  
Software and Embedded Systems Engineering  
Technische Universität Berlin

# Diese VL



# Inhalt

## Code-Qualität

- Einführung
- Kontrollflussgraphen
- Metriken
- Code-Smells
- Refactoring

# Inhalt

## Code-Qualität

- Einführung
- Kontrollflussgraphen
- Metriken
- Code-Smells
- Refactoring

# Qualitätssicherung

## Prozessqualität

*Befasst sich mit der Verbesserung der Entstehung des Software-Produkts (Prozess).*

- Managementprozesse und Entwicklungsmodelle
- Software-Infrastruktur (Build-Automatisierung, Testautomatisierung, ...)

## Produktqualität

*Befasst mit der Verbesserung der genannten Qualitätsmerkmale des Software-Produkts.*

- Korrektheit
- Testen
- Konventionen
- Kommentare
- Statische Analyse
- Metriken
- ...

← Heute: Methoden zur Verbesserung des Codes

# Konventionen ...

*... verhindern Fehler und Missverständnisse  
und erleichtern Teamarbeit.*

## **Mögliche Konventionen:**

- Einheitliche IDE (Integrated Development Environment)
- Inhalt, Format und Sprache von Kommentaren
- Namensgebung und Umfang von Funktionen, Klassen, Variablen, ....
- Formatierung / Coding Style:
  - Formatierung ist individueller Geschmack
  - Einheitliche Formatierung erleichtert das Lesen fremden Codes
  - Automatische Überprüfung / Anpassung durch IDE oder andere Tools
- Verwendung statischer Analyse (oft auch: *linten*):
  - Statische Programmanalyse erweitert die Überprüfung der Formatierung
  - Identifiziert falschen Code Style, Fehlerquellen, ineffiziente Abschnitte, ...
  - Ursprüngliches Tool von Bell Labs für C: „Lint“,

# Dokumentation ...

*...beschreibt die Erstellung von Artefakten, die notwendige oder hilfreiche Informationen übermitteln.*

## Arten von Dokumentation

1. Extern: sind dem Kunden zugänglich
  - Beispiele: Spezifikation, Handbücher, ...
2. Intern: sind dem Kunden nicht zugänglich
  - Beispiele: interne Planungsdokumente, Kommentare, ...

# Kommentare

*Kommentare sind zusätzliche Informationen zum Code, die dessen Verständlichkeit erleichtern sollen.*



[Quelle](#)

- Sinnvoll (bspw. an unverständlichen Code-Stellen)
- Nicht zu viel: Triviale und redundante Kommentare vermeiden!
- Kommentare müssen verständlich sein.

Spezielle Kommentarformate ermöglichen die automatische Generierung von Dokumentation (Beispiel: Javadoc).



# Code Reviews

*Vier Augen sehen mehr als zwei:  
Menschen machen Fehler, doch diese können mit der Durchsicht des  
Codes durch einen anderen Entwickler identifiziert werden.*

## Vorteile

- Mehraufwand kann unnötige Fehler und aufwändigere Bugfixes verhindern
- Schlechter Code (Code-Smells) wird durch Überprüfung verhindert
- Systematische Überprüfung anhand von Checklisten möglich

## Umsetzung

- Spontan: Findet diese wirklich statt?
- Regelmäßig: Die Einbettung in den Entwicklungsprozess ist oft sinnvoll.
- Tool-basiert: Ticket wird als „fertig zur Durchsicht“ markiert.  
(bspw. *Merge Request* in Git).

# Inhalt

## Code-Qualität

- Einführung
- Kontrollflussgraphen
- Metriken
- Code-Smells
- Refactoring

# Kontrollflussgraph (CFG)

Für komplexere Analysemethoden, die die Programmstruktur bewerten, wird eine **abstrakte Repräsentation des Kontrollflusses** benötigt

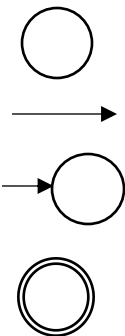
Der Kontrollflussgraph (CFG) einer Prozedur ist ein gerichteter Graph  $G = \{V, E, V_{entry}, V_{exit}\}$ , wobei

$V$  : Knoten (Instruktionen bzw. Basisblöcke)

$E \subseteq V \times V$  : Kanten (Kontrollfluss zwischen Knoten)

$V_{entry} \in V$  : Startknoten

$V_{exit} \in V$  : Endknoten



Statt der formalen verwendet man häufig die **graphische Notation**

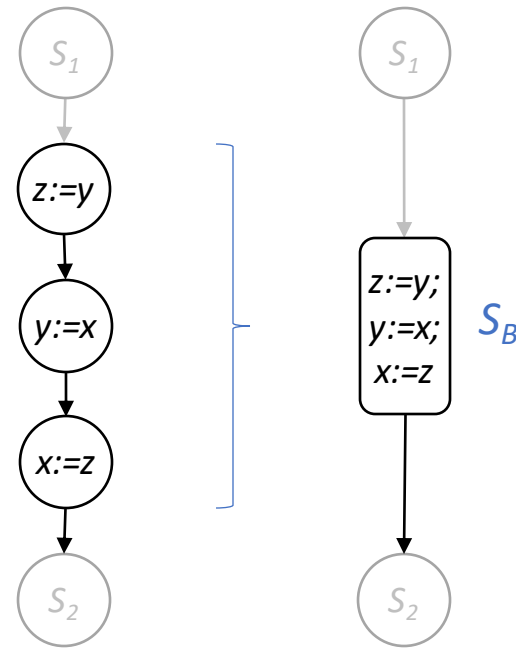
# Basisblöcke

Oft ist es übersichtlicher, anstelle von Instruktionen Basisblöcke als Knoten zu verwenden

Ein Basisblock ist eine **Sequenz von Instruktionen**, wobei

- alle inneren Knoten genau eine eingehende und eine ausgehende Kante haben
- der erste Knoten hat beliebig viele eingehende Kanten und eine ausgehende Kante
- der letzte Knoten hat eine eingehende Kante und beliebig viele ausgehende Kanten

$S_1; \overbrace{z:=y; y:=x; x:=z}^{S_B}; S_2$

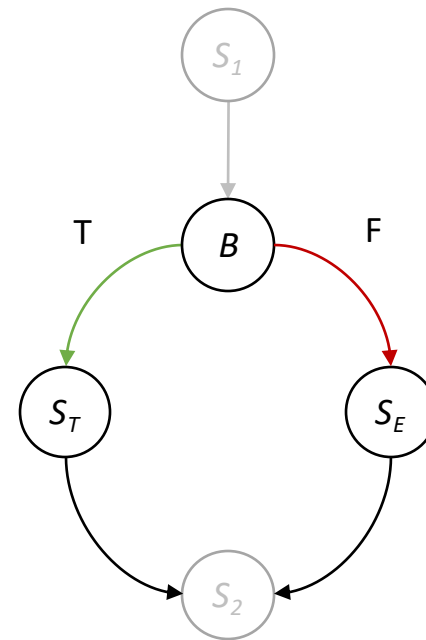


# Verzweigung

$S_1$ ; **if**  $B$  **then**  $S_T$  **else**  $S_E$  **fi**;  $S_2$

Verzweigungen werden über mehrere ausgehende bzw. eingehende Kanten realisiert

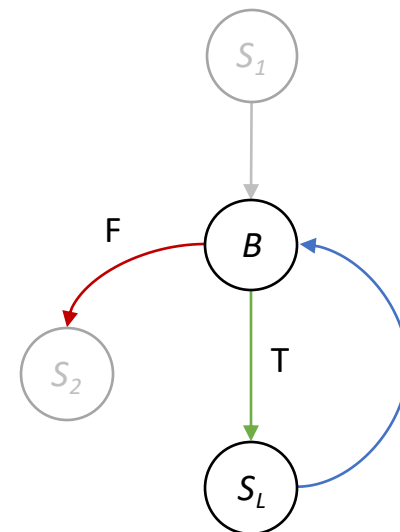
Dabei **können** die Kanten mit zusätzlicher Information dekoriert werden



# Schleifen

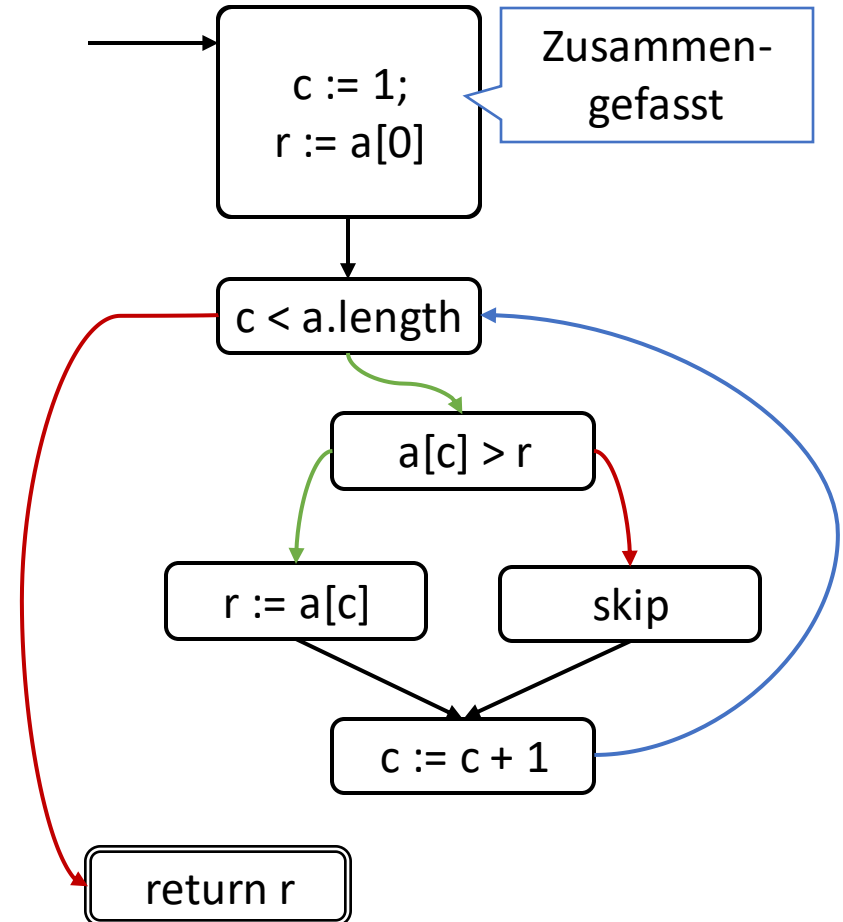
$S_1$ ; **while**  $B$  **do**  $S_L$  **od**;  $S_2$

Schleifen werden durch  
**rückwärtige Kanten** (*back edges*)  
repräsentiert



# Beispiel

```
maxArrayElem(int a[])  
c := 1;  
r := a[0];  
while c < a.length do  
    if a[c] > r then  
        r := a[c]  
    else  
        skip  
    fi;  
    c := c + 1  
od;  
return r
```



# Kontrollflussgraphen für JAVA

CFGs eignen sich zur Darstellung des **intra**prozeduralen Kontrollflusses

- Aufrufgraphen (*callgraph*) stellen **Inter**prozeduralen Kontrollfluss dar
- Daher stellen wir **Methodenaufrufe** nicht explizit dar

Außergewöhnlicher Kontrollfluss (Exceptions, Interrupts, Signale, ...) wird im Kontrollflussgraphen meist nicht explizit dargestellt

- Können jederzeit eintreten: Kanten **von/zu allen** Knoten unübersichtlich
- Innerhalb dieser Lehrveranstaltung abstrahieren wir davon

Weitere **komplexe Konstrukte** wie switch-case Anweisungen, break/continue, Zählschleifen, Zuweisungen in Bedingungen etc. werden durch den Compiler in einfache Konstellationen zerlegt

- Einige Fälle betrachten wir in der Übung genauer



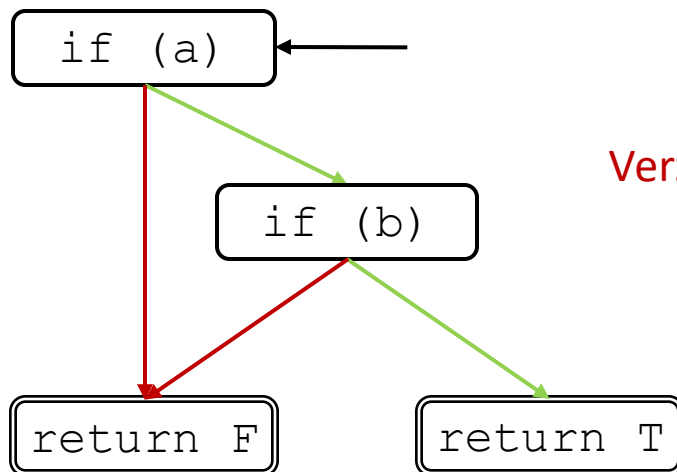
# Problem: Bedingte Auswertung in JAVA

JAVA ermöglicht **bedingte Auswertung** durch `||` und `&&`

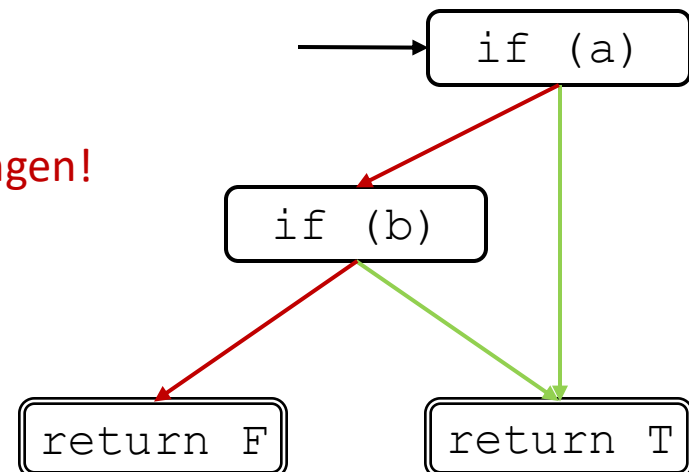
- Dadurch ergeben sich **zusätzliche Verzweigungen**

```
if (a && b) return true;  
else return false;
```

```
if (a || b) return true;  
else return false;
```



**Zwei  
Verzweigungen!**



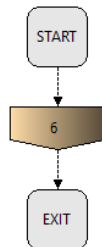
# Problem: Optimierung

**Achtung:** Auch beim Compilieren kann sich der Kontrollfluss ändern!

## Quellcode

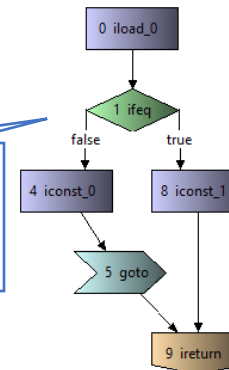
```
boolean neg(boolean x) {  
    return !x;  
}
```

Keine Verzweigung



## Bytecode

Verzweigung



Entspricht eher

```
boolean neg(boolean x) {  
    if (x) return false;  
    else return true;  
}
```

JAVA 7

# Inhalt

## Code-Qualität

- Einführung
- Kontrollflussgraphen
- **Metriken**
- Code-Smells
- Refactoring

# Motivation

## Metrik

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

## 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

„**Miss**, was gemessen werden kann und  
**made messbar**, was nicht messbar ist.“

Galileo Galilei

Im **Software-Engineering** (Tom DeMarco (\*1940)):

„You can't **control** what you can't **measure**.“

Erster Satz aus seinem Buch „Controlling Software Projects: Management, Measurement, and Estimation“, Prentice Hall 1982.

# Software-Metriken: Definitionen

## **Ian Sommerville** (aus: Software Engineering)

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

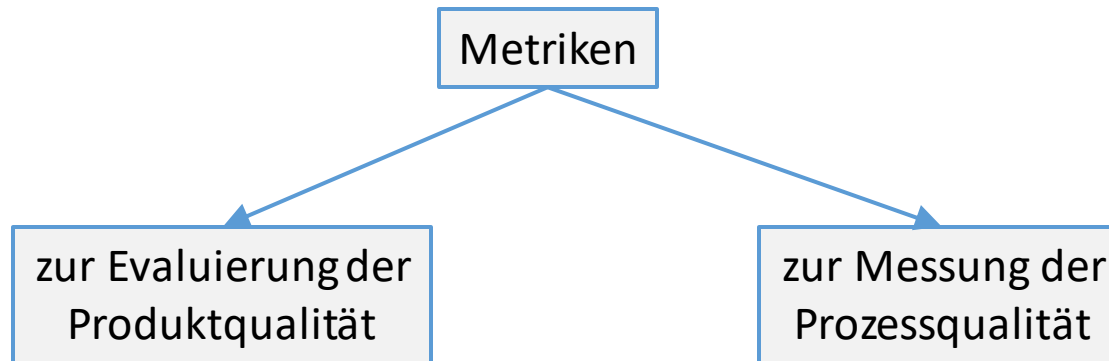
## **Definition des IEEE Standard 1061 (1992):**

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

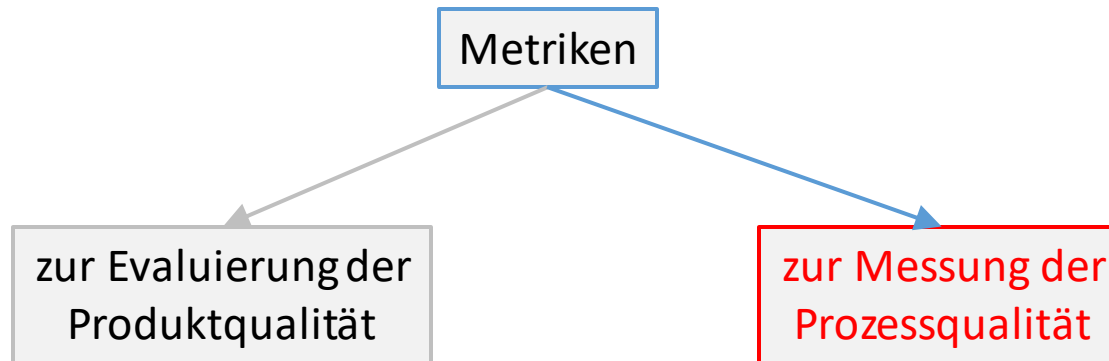
<b>Objektivität</b>	kein Einfluss durch den Messenden
<b>Zuverlässigkeit</b>	Ergebnis für dieselbe Messung immer gleich
<b>Normierung</b>	Skala existiert, die Messergebnisse einordnet
<b>Vergleichbarkeit</b>	mit anderen Maßen in Relation
<b>Ökonomie</b>	Messung nicht zu teuer/durchführbar
<b>Nützlichkeit</b>	hilft in der Praxis
<b>Validität</b>	sinnvoll, misst das Richtige (Messergebnisse erlauben die gewünschten Rückschlüsse)

# Arten von Software-Metriken





# Arten von Software-Metriken



# Prozessmetriken

## Zur Messung der Entwicklung von Software

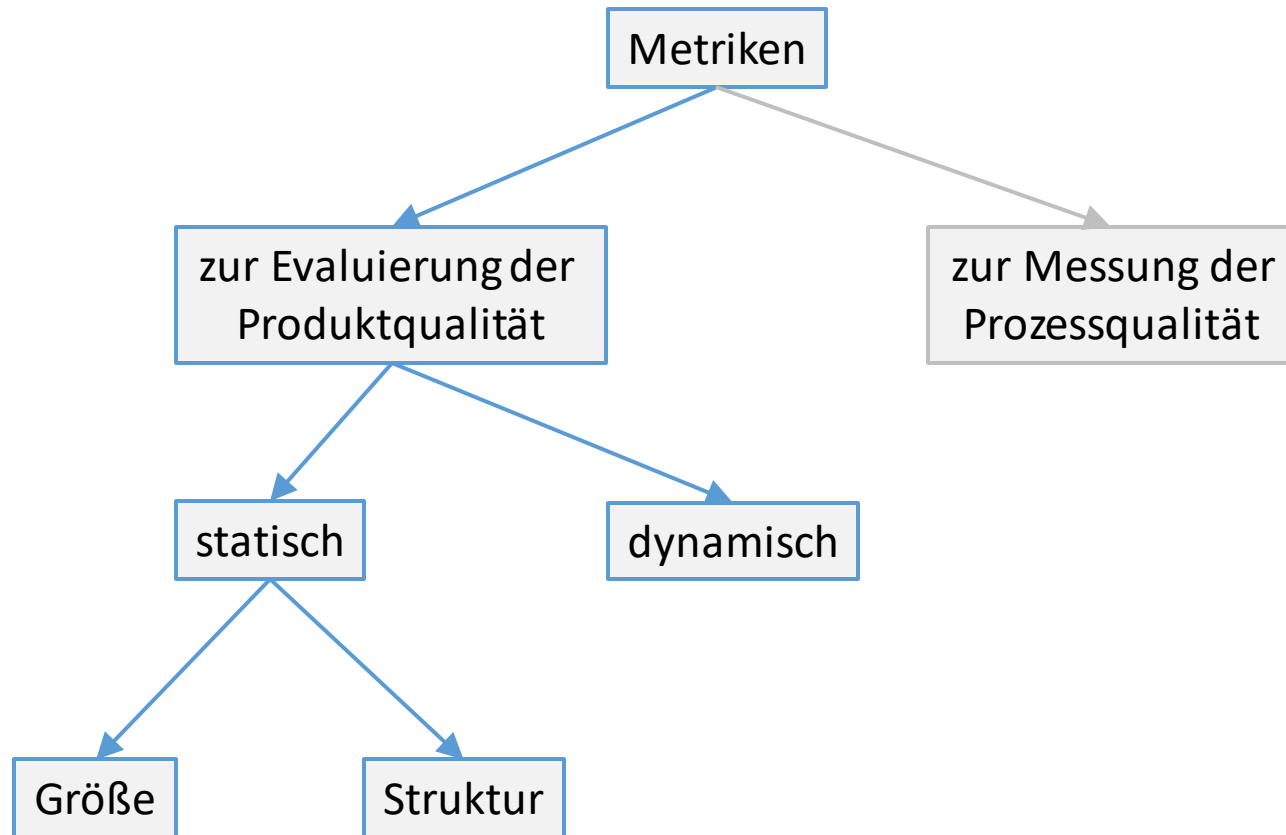
- Benötigte Ressourcen (Personentage, Reisekosten, Computerressourcen etc.)
- Absolute Zeiten (z.B. wie lange es dauert, eine Änderung durchzuführen bzw. Fehler zu beheben)
- Häufigkeit bestimmter Ereignisse (Anzahl gefundener Fehler bei Programminspektionen, Anzahl Anforderungsänderungen und daraus resultierender Programmänderungen etc.)

## Maß für Produktivität

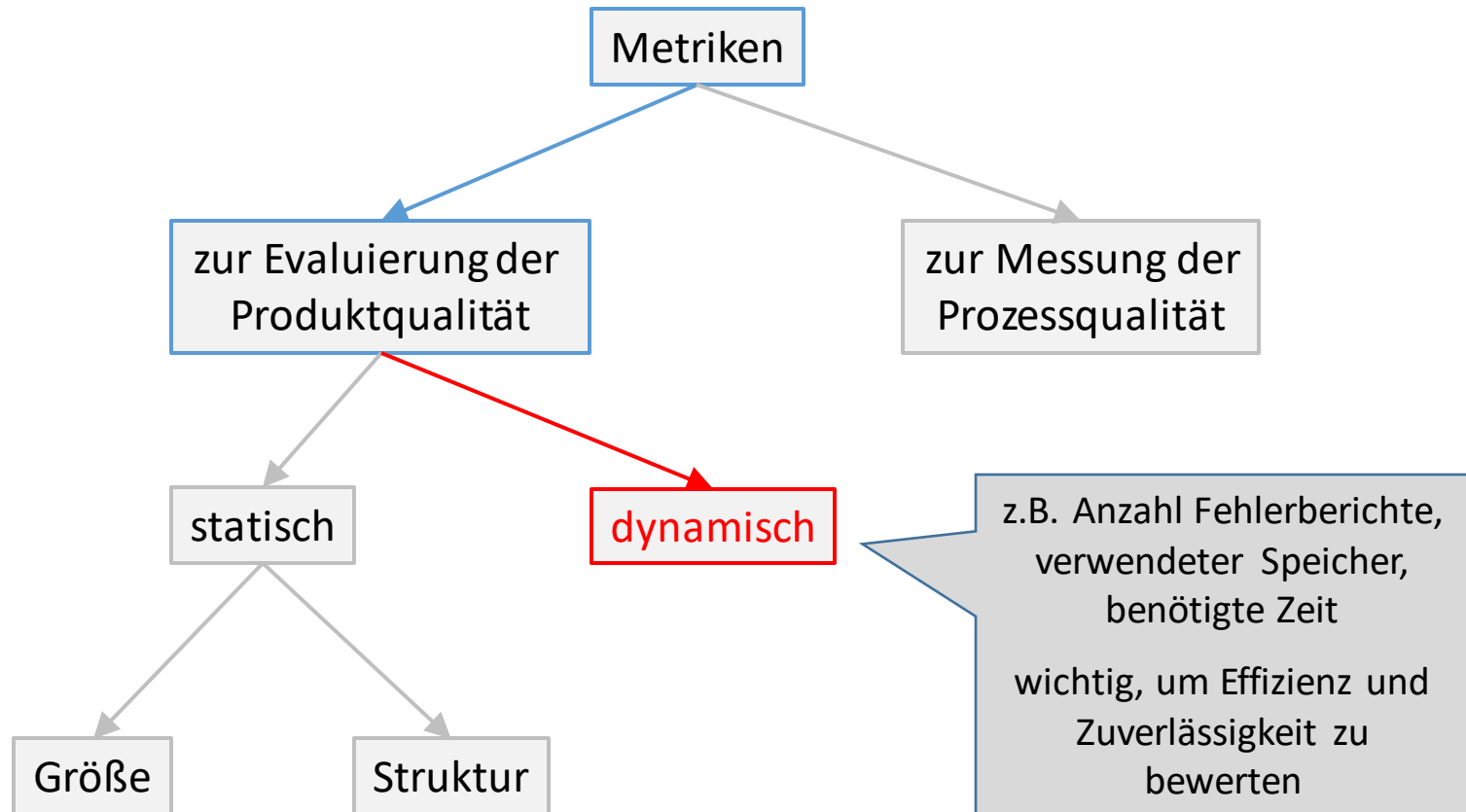
- kann auch eingesetzt werden, um Leistungsfähigkeit von Teams oder einzelnen Programmierern zu bewerten

Abhängig vom Management- bzw. Entwicklungsprozess  
und daher nicht im Fokus dieser Vorlesung!

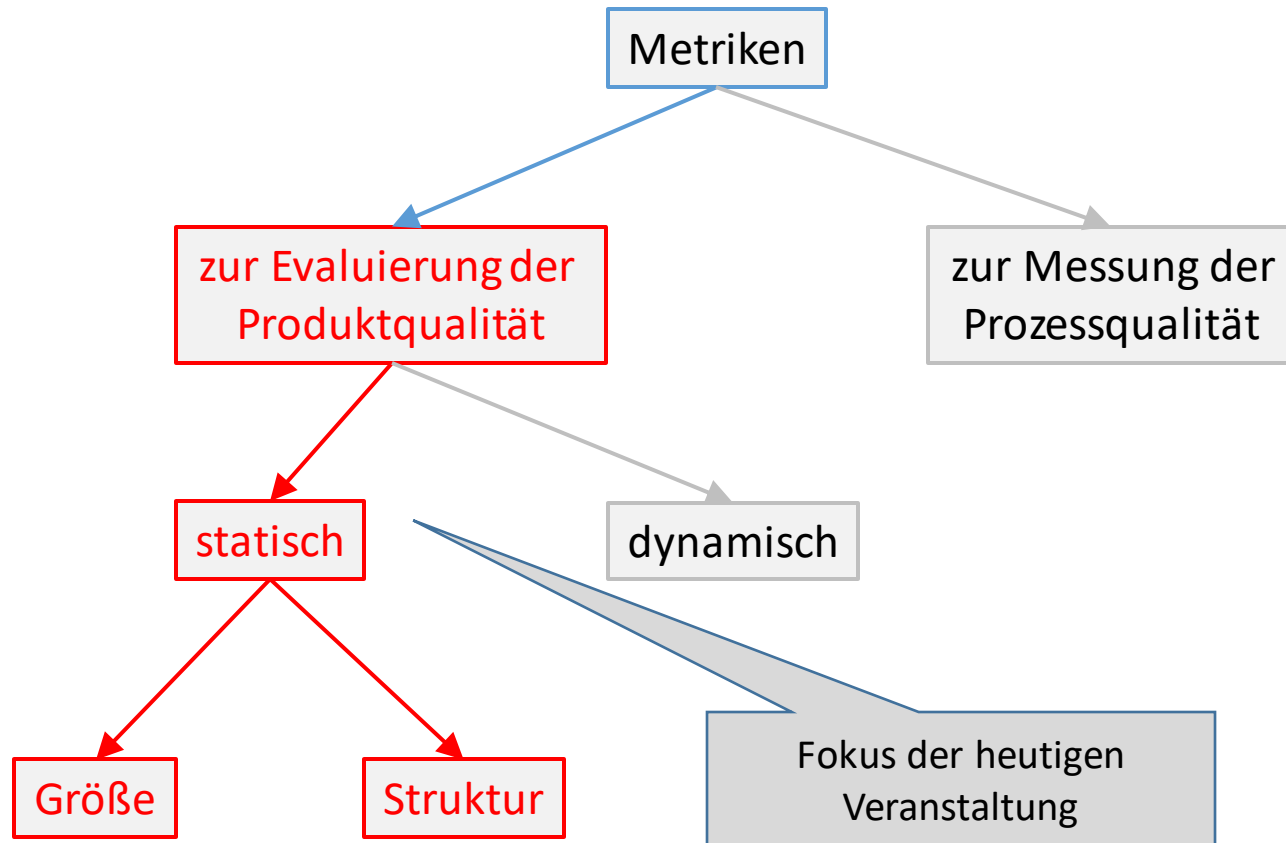
# Arten von Software-Metriken



# Arten von Software-Metriken



# Arten von Software-Metriken



# Statische Produktmetriken

## Traditionellen Metriken

Metriken zur Messung der **Programmgröße** und dessen **Komplexität**

- Zeilenmetriken (LOC) und Halstead-Metriken

Metriken zur Messung der **Programmstruktur**

- Zyklomatische Komplexität nach McCabe

## Objektorientierte Metriken

- **Verhältnisse** der einzelnen Elemente (Klassen, Methoden) untereinander

# Zeilenmetriken

## Lines Of Code (**LOC**)

- Anzahl der Zeilen im Programm

## Non-Commenting Lines Of Code (**NCLOC**)

- ignoriere Leerzeilen und reine Kommentarzeilen
- Anteil an Kommentarzeilen:
  - sollte zwischen 30% und 75% liegen

## Typische Vorgabewerte für LOC

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

```
01 // author sg
02 #include <stdio.h>
03
04 /* prints "hello world"
05    returns: error code 0 */
06 int main(void)
07 {
08     printf("Hello, World\n");
09
10     return 0;
11 }
```



LOC = 11

NCLOC = 6

# Zeilenmetriken



**Crista Lopes**  
@cristalopes

 Follow

The largest C++ file we found in GitHub has 528Mb, 57 lines of code. Contains the first 50,847,534 primes, all hard coded in an array.

## Vorteil

- leicht zu berechnen
- leicht nachzuvollziehen / intuitiv

## Nachteil

- wenig aussagekräftig
- abhängig vom Programmierstil
- bessere Programmstruktur kann auch weniger Zeilen bedeuten
- weniger Zeilen kann auch komplexeren Code bedeuten



# Halstead Metriken

Misst die textuelle bzw. **lexikalische Komplexität**

- 1977 durch Maurice Halstead eingeführt

Programm aufgefasst als Sequenz von **Operatoren** und **Operanden**

- Damit lassen sich **verschiedene Metriken** berechnen
- Problem: Definition von Operanden und Operatoren **nicht eindeutig**
  - Beispiel
    - Operatoren: feste Sprachelemente, z.B. if, (), int, public...
    - Operanden: alles andere, u.A. Variablen, Werte, Bezeichner...
  - Hängt von der Sprache ab
  - Projekt/Implementierungsspezifisch
  - Muss bei Angabe von Halstead-Metriken mit angegeben werden

# Halstead Metriken

## Definition **Werte**

Unterschiedliche **Operatoren**

$n1$

Unterschiedliche **Operanden**

$n2$

**Operatoren** im Programm

$N1$

**Operanden** im Programm

$N2$

Größe des Vokabulars

$n = n1 + n2$

Länge des Programms

$N = N1 + N2$

## Definition **Metriken**

Volumen des Programms

$V = N * \log_2(n)$

Schwierigkeit (difficulty)

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

Aufwand, um Programm zu verstehen

$E = D * V$

# Halstead Metriken

```
int max(int a[],
        int len)
{
    int m = a[0];

    for(int i = 1;
        i < len; i++)
    {
        if(m < a[i])
        {
            m = a[i];
        }
    }
    return m;
}
```

Operatoren	#	Operanden	#
int	5	max	1
[]	4	a	4
{}	3	len	2
,	1	i	5
()	3	0	1
for	1	1	1
=	3	m	4
<	2		
if	1		
++	1		
;	5		
return	1		

$$n1 = 12$$

$$n2 = 7$$

$$N1 = 30$$

$$N2 = 18$$

$$n = n1 + n2 = 19$$

$$N = N1 + N2 = 48$$

$$V = N * \log_2(n) = 203.9$$

$$D = \left(\frac{n1}{2}\right) * \left(\frac{N2}{n2}\right) = 15.43$$

$$E = D * V = 3146$$

# Halstead Metriken

## Vorteile

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

## Nachteile

- Konzepte moderner Programmiersprachen unberücksichtigt (Namensräume, Sichtbarkeiten, Vererbung, ...)
- Aufteilung in Operatoren vs. Operanden nicht immer möglich
- **Struktur** und Ausführung nicht berücksichtigt

Wie beschreibt man eigentlich die  
**Struktur** eines Programmes?

# Strukturmetriken

## Zyklomatische Komplexität $v(G)$

- eingeführt 1976 von Thomas McCabe

### Definiert als

- Anzahl konditioneller Zweige im **Kontrollflussgraphen** des Programms
- Entspricht Anzahl binärer Verzweigungen + 1

## Zyklomatische Zahl

- $v(G) \leq 10$ : einfache Programme
- $v(G) > 10$ : Fehler nehmen stark zu
- $v(G) \geq 50$ : sehr bzw. zu komplexe Programme, kaum noch zu testen
- **je höher, desto komplexer** das Programm, desto mehr Testfälle nötig

# Zyklomatische Komplexität

## Definition

$e$	Anzahl Kanten im Graphen
$n$	Anzahl Knoten im Graphen
$p$	Anzahl Komponenten (zusammenhängende Graphen)

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

## Beobachtung:

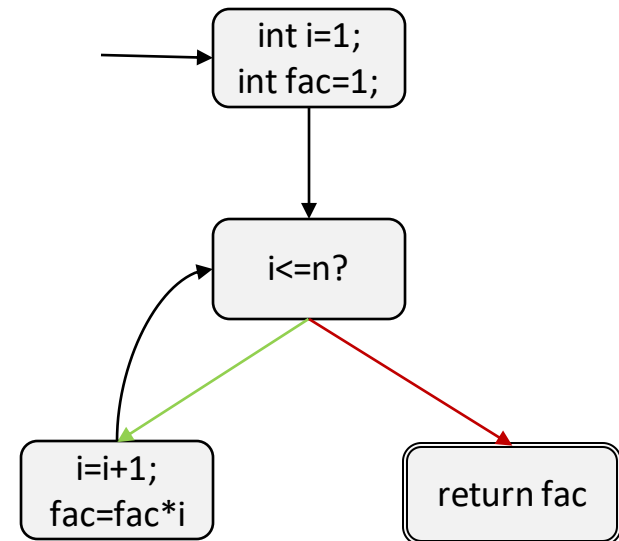
Ohne Verzweigungen gilt  $n = e + 1$ , also  $v(G) = e - (e + 1) + 2 * 1 = 1$

Wichtig ist, dass der Kontrollflussgraph nur **einen Endknoten** hat

- Sollten z.B. mehrere returns existieren, müssen diese bei der Konstruktion des CFG mit einem „virtuellen“ Endknoten verbunden werden

# Zyklomatische Komplexität

```
int faculty(int n) {  
    int i=1;  
    int fac=1;  
    while(i<=n) {  
        i=i+1;  
        fac=fac*i;  
    }  
    return fac;  
}
```



## Berechnung

$e$	Anzahl Kanten	= 4
$n$	Anzahl Knoten	= 4
$p$	Anzahl Komponenten	= 1

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

# Zyklomatische Komplexität

## **Vorteile**

- leicht zu berechnen
- in Fallstudien: gute Korrelation zwischen zyklomatischer Komplexität und Programmverständlichkeit
- Eignet sich zur Testplanung

## **Nachteile**

- berücksichtigt Kontroll-, aber nicht Datenfluss
- Wenig aussagekräftig in objektorientierter Software mit vielen einfachen Zugriffsmethoden (z.B. Attribute)



# Weitere Metriken

**NBD:** Verschachtelungstiefe (nested block depth)

- sollte <5 sein

**NST:** Number of Statements (etwa Anzahl Semikolons in JAVA)

- sollte <50 sein

**NFC:** Number of Function Calls

- sollte <5 sein

**NOM:** Number of Methods

**Objektorientierte Metriken** für besondere Aspekte

- klassische Metriken können aber auf Methoden angewendet werden

# Objektorientierte Metriken 1

## 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 2

## **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 3

## **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

...

# Sind Metriken 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: „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

# Beispiel: Eclipse Metrics Plug-In (JAVA)

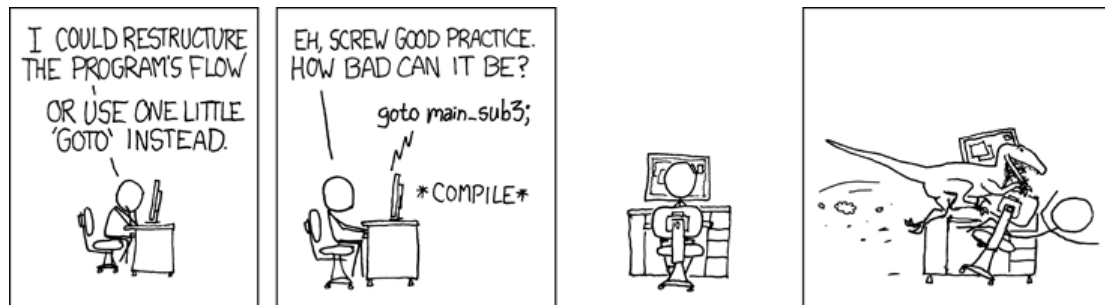
Metric	Total	Mean	Std. Dev.	Maxim...	Method
> Number of Overridden Methods (avg/max per type)	12	0,364	0,54	2	
> Number of Attributes (avg/max per type)	43	1,303	2,713	11	
> Number of Children (avg/max per type)	19	0,576	1,759	8	
> Number of Classes (avg/max per packageFragment)	33	4,714	3,057	9	
> Method Lines of Code (avg/max per method)	523	4,395	8,253	59	doGet
> Number of Methods (avg/max per type)	118	3,576	3,806	17	
> Nested Block Depth (avg/max per method)		1,143	0,584	5	tryMove
> Depth of Inheritance Tree (avg/max per type)		2,182	1,167	4	
> Number of Packages	7				
> Afferent Coupling (avg/max per packageFragment)		4	2,449	8	
> Number of Interfaces (avg/max per packageFragment)	1	0,143	0,35	1	
> McCabe Cyclomatic Complexity (avg/max per method)		2,101	3,394	23	Board
> Total Lines of Code	1088				
> Instability (avg/max per packageFragment)		0,37	0,283	1	
> Number of Parameters (avg/max per method)		0,798	0,846	4	Move
> Lack of Cohesion of Methods (avg/max per type)		0,122	0,263	0,8	
> Efferent Coupling (avg/max per packageFragment)		2,143	1,457	5	
> Number of Static Methods (avg/max per type)	1	0,03	0,171	1	
> Normalized Distance (avg/max per packageFragment)		0,557	0,273	1	
> Abstractness (avg/max per packageFragment)		0,073	0,089	0,2	
> Specialization Index (avg/max per type)		0,142	0,216	0,545	
> Weighted methods per Class (avg/max per type)	250	7,576	12,962	61	
> Number of Static Attributes (avg/max per type)	4	0,121	0,409	2	

# Inhalt

## Code-Qualität

- Einführung
- Kontrollflussgraphen
- Metriken
- **Code-Smells**
- Refactoring

# Wie erkennt man schlechten Code?



[Quelle](#)

*Schlecht strukturierter Code („Code-Smells“)  
ist oft offensichtlich und lässt sich kategorisieren.*

## Maßnahmen

- Code Reviews
- Konvention: Metriken, Code Style und Linting
- Pair Programming



# Smell - Datenklumpen

*Gleiche Variablen treten an vielen Stellen des Programms gemeinsam auf:*

```
public class Adressbook {  
  
    // ...  
  
    void callContact(String name, String countryCode, String phonenumber) {  
        // ...  
    }  
  
    void showContact(String name, String countryCode, String phonenumber) {  
        // ...  
    }  
  
    void sendMessage(String name, String countryCode, String phonenumber, String msg) {  
        // ...  
    }  
  
    // ...  
  
}
```

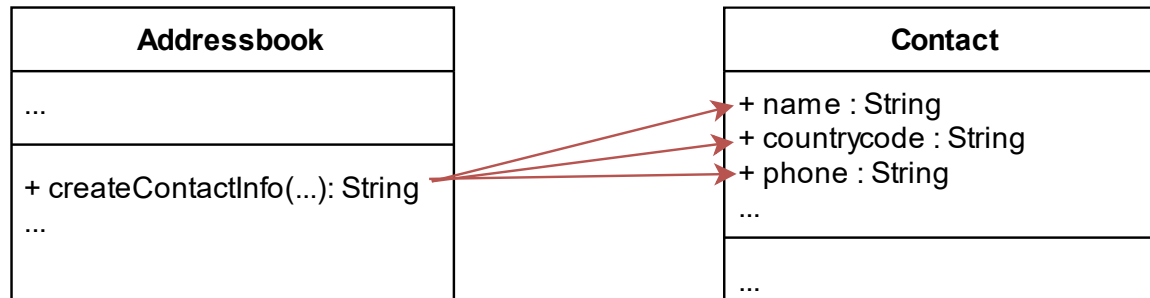
# Smell - viele Bedingungen (Switch Anweisung )

*Viele Bedingungen im Kontrollfluss, die durch eine Switch-Anweisung entstehen, können schwer verständlich sein.*

```
void doesNotLookTooBad(int i, boolean y) {  
    switch (i) {  
        case 0:  
            // do something  
        case 1:  
            // do something else  
        case 2:  
            // do another thing  
        case 3:  
            if (y) {  
                break;  
            }  
        // More possible cases here  
        default:  
            throw new IllegalArgumentException("Unexpected value: " + i);  
    }  
}
```

# Smell – Neid (falsche Zuständigkeit)

*Eine Methode in einer Klasse verwendet viele Attribute einer anderen Klasse.*



# Und viele mehr!

- Lange Methoden
- Lange Parameterliste
- Große Klasse
- Duplizierter Code
- Neigung zu elementaren Datentypen  
*Viele elementare Datentypen weisen darauf hin, dass diese eventuell in einer Datenstruktur gekapselt werden können.*
- Unangebrachte Intimität  
*Eine Klasse verwendet viele Teile einer weiteren Klasse, bei denen Details der Implementierung (im Gegensatz zur Schnittstelle) eine Rolle spielen.*
- Viele Kommentare  
*Kommentare sind dort notwendig, wo der Code schwer verständlich ist.*
- ...

# Inhalt

## Code-Qualität

- Einführung
- Kontrollflussgraphen
- Metriken
- Code-Smells
- Refactoring

# Motivation

Metric	Total	Mean	Std. Dev.	Maxim...	Method
> Number of Overridden Methods (avg/max per type)	12	0,364	0,54	2	
> Number of Attributes (avg/max per type)	43	1,303	2,713	11	
> Number of Children (avg/max per type)	19	0,576	1,759	8	
> Number of Classes (avg/max per packageFragment)	33	4,714	3,057	9	
> Method Lines of Code (avg/max per method)	523	4,395	8,253	59	doGet
> Number of Methods (avg/max per type)	118	3,576	3,806	17	
> Nested Block Depth (avg/max per method)		1,143	0,584	5	tryMove
> Depth of Inheritance Tree (avg/max per type)		2,182	1,167	4	
> Number of Packages	7				
> Afferent Coupling (avg/max per packageFragment)		4	2,449	8	
> Number of Interfaces (avg/max per packageFragment)	1	0,143	0,35	1	
> McCabe Cyclomatic Complexity (avg/max per method)		2,101	3,394	23	Board
> Total Lines of Code	1088				
> Instability (avg/max per packageFragment)		0,37	0,283	1	
> Number of Parameters (avg/max per method)		0,798	0,846	4	Move
> Lack of Cohesion of Methods (avg/max per type)		0,122	0,263	0,8	
> Efferent Coupling (avg/max per packageFragment)		2,143	1,457	5	
> Number of Static Methods (avg/max per type)	1	0,03	0,171	1	
> Normalized Distance (avg/max per packageFragment)		0,557	0,273	1	
> Abstractness (avg/max per packageFragment)		0,073	0,089	0,2	
> Specialization Index (avg/max per type)		0,142	0,216	0,545	
> Weighted methods per Class (avg/max per type)	250	7,576	12,962	61	
> Number of Static Attributes (avg/max per type)	4	0,121	0,409	2	

Ist der Code noch zu retten?

# Problem

Bereits geschriebenen Code **umzuschreiben** ist **riskant**

- **Äquivalentes Verhalten** des neuen Codes muss **gewährleistet** werden
- Eventuell werden neue **Fehler** eingebaut
- *Betroffene* Funktionalität muss neu **getestet** werden (change impact analysis)
- Änderungen können **weiteres Umschreiben** notwendig machen
- In **zeitkritischen** Systemen dürfen keine Schranken verletzt werden

“There are two ways of constructing a software design:  
One way is to make it so **simple** that there are **obviously no deficiencies**,  
the other way is to make it so **complicated** that there are **no obvious deficiencies**.”

C. A. R. Hoare

“The computing scientist’s **main challenge** is not to get confused  
by the **complexities** of his own making.”

E. W. Dijkstra

# Refactoring

Im **Refactoring** werden **mehrere Regeln** schrittweise angewendet, um

- Die **interne Struktur** des Codes zu **verbessern**
- Improvisierte **Lösungen** dem Design **anzupassen**
- **Verständlichkeit** und **Wartbarkeit** der Codebasis zu erhöhen

Dabei gilt für jede **Regel**

- Sie dient **einem** primären, ausgewiesenen **Zweck**
- Sie ist **möglichst einfach**, um neue Fehler zu vermeiden
- Sie ist **konstruktiv** und besteht wiederum aus mehreren Schritten, die eine **Ausgangssituation** in einen **Zielzustand** überführen
- Beobachtbares **Verhalten wird beibehalten**

“One of my most **productive** days was  
**throwing away 1000 lines of code.**”

K. Thompson



# Gründe für Refactoring

Refactoring kann zu verschiedenen Zeiten **im Entwicklungsprozess notwendig** werden

- Falls das **Hinzufügen von neuen Funktionen** von früheren Designentscheidungen **erschwert** wird
- Falls **Fehler behoben** werden, deren Identifikation durch unverständlichen Code **erschwert** wurde
- Nach **Code-Reviews**

*Kategorisierung der Probleme im Code nach „Code-Smells“ kann helfen die passenden Maßnahmen festzulegen.*

# Code-Smells

Metric	Total	Mean	Std. Dev.	Maxim...	Method	
> Number of Overridden Methods (avg/max per type)	12	0,364	0,54	2		
> Number of Attributes (avg/max per type)	43	1,303	2,713	11		Datenklumpen
> Number of Children (avg/max per type)	19	0,576	1,759	8		
> Number of Classes (avg/max per packageFragment)	33	4,714	3,057	9		
> Method Lines of Code (avg/max per method)	523	4,395	8,253	59	doGet	Lange Methode
> Number of Methods (avg/max per type)	118	3,576	3,806	17		
> Nested Block Depth (avg/max per method)		1,143	0,584	5	tryMove	Lange Methode
> Depth of Inheritance Tree (avg/max per type)		2,182	1,167	4		
> Number of Packages	7					
> Afferent Coupling (avg/max per packageFragment)		4	2,449	8		
> Number of Interfaces (avg/max per packageFragment)	1	0,143	0,35	1		
> McCabe Cyclomatic Complexity (avg/max per method)		2,101	3,394	23	Board	Switch-Befehl
> Total Lines of Code	1088					
> Instability (avg/max per packageFragment)		0,37	0,283	1		
> Number of Parameters (avg/max per method)		0,798	0,846	4	Move	Lange Parameterliste
> Lack of Cohesion of Methods (avg/max per type)		0,122	0,263	0,8		
> Efferent Coupling (avg/max per packageFragment)		2,143	1,457	5		
> Number of Static Methods (avg/max per type)	1	0,03	0,171	1		
> Normalized Distance (avg/max per packageFragment)		0,557	0,273	1		
> Abstractness (avg/max per packageFragment)		0,073	0,089	0,2		
> Specialization Index (avg/max per type)		0,142	0,216	0,545		
> Weighted methods per Class (avg/max per type)	250	7,576	12,962	61		Große Klasse
> Number of Static Attributes (avg/max per type)	4	0,121	0,409	2		

# Refactoring Katalog

Jeder **Code-Smell** weist auf eine Reihe von **Refactoring-Schritten** hin, die helfen, das Problem zu beheben

- Die einzelnen Regeln wurden in einem **Katalog** zusammengetragen
- Sie werden unter anderem durch ihren **Namen**, der **Motivation**, einer ausführlichen Beschreibung der Vorgehensweise, sowie einer kurzen **Zusammenfassung** charakterisiert
- Die ausführliche Beschreibung der Vorgehensweise weist auf **Fehlerpotenziale** und **andere Regeln** hin

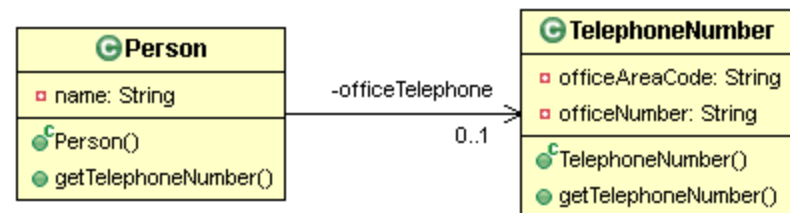
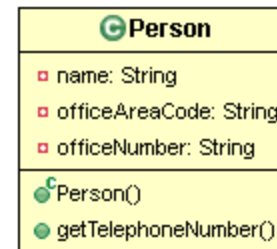
Der **Katalog** wurde ursprünglich von Martin Fowler zusammengetragen und umfasst über 70 **Refactorings**

# Klasse extrahieren

*Eine Klasse macht die Arbeit, die von zwei Klassen zu erledigen wäre.*

**Smells:** Große Klasse, Datenklumpen, duplizierter Code

**Zusammenfassung:** Erstellen Sie eine neue Klasse, und verschieben Sie die relevanten Felder und Methoden von der alten Klasse in die neue



# Methode extrahieren

*Ein Fragment im Code kann zusammengefasst werden.*

**Smells:** Lange Methode, duplizierter Code, Kommentare

**Zusammenfassung:** Machen Sie aus dem Fragment eine Methode, deren Name die Aufgabe der Methode erklärt.

```
void printOwning(int amount) {  
    printBanner();  
  
    // print details  
    System.out.println("name: "+this.name);  
    System.out.println("amount: "+amount);  
}
```



```
void printOwning(int amount) {  
    printBanner();  
    printDetails(amount);  
}  
  
void printDetails(int amount) {  
    System.out.println("name: "+this.name);  
    System.out.println("amount: "+amount);  
}
```

# Geschachtelte Bedingungen durch Wächterbedingungen ersetzen

*Eine Methode weist ein bedingtes Verhalten auf, das den normalen Ablauf nicht leicht erkennen lässt.*

**Smells:** Lange Methode

**Zusammenfassung:** Verwenden Sie Wächterbedingungen für die Spezialfälle

```
int getPayAmount() {  
    int result;  
    if (isIntern()) result = internAmount();  
    else {  
        if (isRetired()) result = retiredAmount();  
        else result = fullAmount();  
    }  
    return result;  
}
```



```
int getPayAmount() {  
    if (isIntern()) return internAmount();  
    if (isRetired()) return retiredAmount();  
    return fullAmount();  
}
```

# Parameter durch explizite Methode ersetzen

*Eine Methode führt abhängig von einem ihrer Parameter unterschiedlichen Code aus.*

**Smells:** Lange Parameterliste, Switch-Befehl

**Zusammenfassung:** Erstellen Sie eine separate Methode für jeden Wert des Parameters

```
void setValue(String name, int value) {  
    switch (name) {  
        case "height": height = value; break;  
        case "width" : width  = value; break;  
        default: throw new  
            IllegalArgumentException();  
    }  
}
```



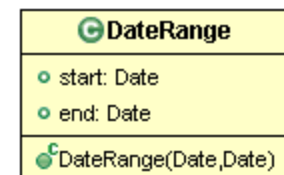
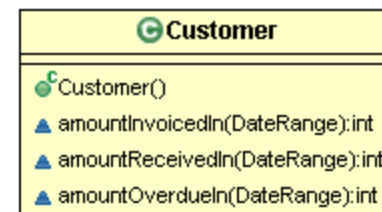
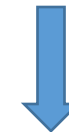
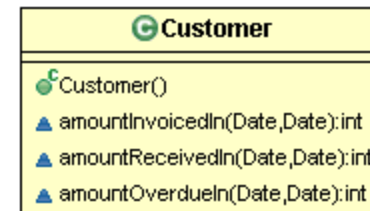
```
void setHeight(int value) {  
    height = value;  
}  
  
void setWidth(int value) {  
    width  = value;  
}
```

# Parameterobjekt einführen

*Eine Gruppe von Parametern gehört auf natürliche Weise zusammen.*

**Smells:** Lange Parameterliste,  
Neigung zu elementaren Typen,  
Datenklumpen

**Zusammenfassung:** Ersetzen Sie sie durch ein Objekt





# Methode verschieben

*Eine Methode nutzt mehr Elemente einer anderen Klasse oder wird von mehr Elementen einer Klasse benutzt als von denen, in der sie definiert ist.*

**Smells:** Datenklassen, unangebrachte Intimität, Neid

**Zusammenfassung:** Ersetzen Sie sie durch eine neue Methode in der Klasse, die sie am meisten verwendet

```
public class Account {  
    AccountType type;  
  
    int overdraftCharge(int daysOverdrawn) {  
        if (type.isPremium())  
            return 10 + 2 * daysOverdrawn;  
        else if (type.isDiscount())  
            return daysOverdrawn;  
        else return 3 * daysOverdrawn;  
    }  
}
```



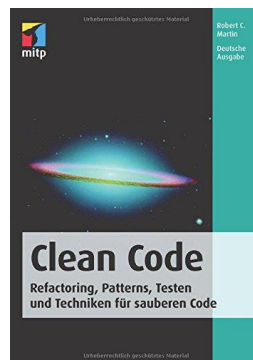
```
class Account {  
    AccountType type;  
  
    int overdraftCharge(int daysOverdrawn) {  
        return  
            type.overdraftCharge(daysOverdrawn);  
    }  
}
```

# Besser: Von Anfang an alles richtig machen

*Es gibt viele Informationen und Hilfsmittel,  
die beim Programmieren helfen.*

TICS Coding Standard Viewer TIOBE - Java Coding Standard		
TIOBE Software Quality Framework		
Home   Coding Rules   Categories   Literature   Change History   Generate PDF   Log in		
Name	Checked	Synopsis
Basic2	✓	Avoid empty 'if' statements
Basic3	✓	Avoid empty 'while' statements
Basic4	✓	Avoid empty try blocks
Basic5	✓	Avoid empty finally blocks
Basic6	✓	Avoid empty switch statements
Basic7	✓	Avoid modifying an outer loop incrementer in an inner loop for update expression
Basic8	✓	This for loop could be simplified to a while loop
Basic9	✓	Avoid unnecessary temporaries when conve
Basic10	✓	Ensure you override both equals() and hash
Basic11	✓	Double checked locking is not thread safe in
Basic12	✓	Avoid returning from a finally block
Basic13	✓	Avoid empty synchronized blocks
Basic14	✓	Avoid unnecessary return statements
Basic15	✓	Empty static initializer was found
Basic16	✓	Do not use 'if' statements that are always tru

<http://tics.tiobe.com/viewer/java/index.php>



## Java Coding Guidelines

Erstellt von Joe McManus MGR, zuletzt geändert von Sandy Shrum am Mär 05, 2015

### CERT Books



### CERT Websites

CERT  
Secure Coding  
Tech Tips

<https://www.securecoding.cert.org/confluence/display/java/java+Coding+Guidelines>

## Google Java Style Guide

### Table of Contents

#### 1 Introduction

[1.1 Terminology notes](#)

[1.2 Guide notes](#)

#### 2 Source file basics

[2.1 File name](#)

[2.2 File encoding: UTF-8](#)

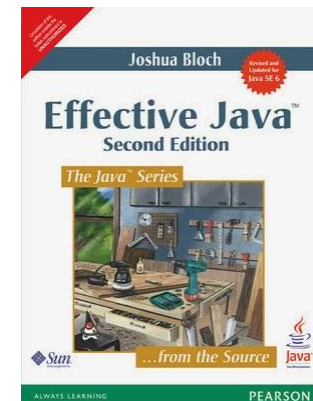
[2.3 Special characters](#)

<https://google.github.io/styleguide/javaguide.html>

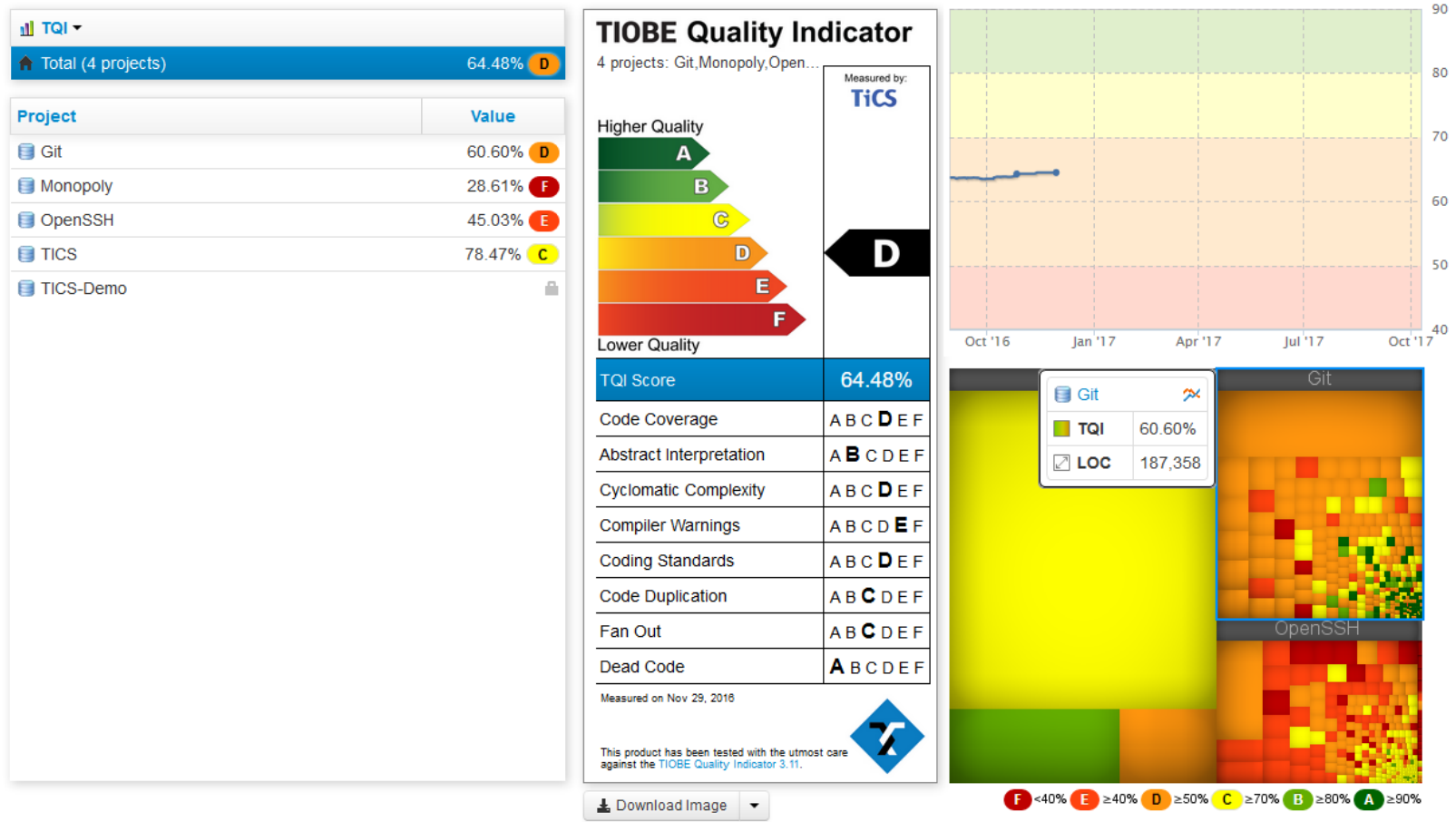
The *Java Coding Guidelines* includes recomm making this effort a success. We thank and ac

### Preface

- 1. Security
- 2. Defensive Programming
- 3. Reliability
- 4. Program Understandability
- 5. Programmer Misconceptions
- Rule 00. Input Validation and Data Sanitization (IDS)
- Rule 01. Declarations and Initialization (DCL)
- Rule 02. Expressions (EXP)
- Rule 03. Numeric Types and Operations (NUM)



# Beispiel: TiCS (C/C++, C#, JAVA, Python, ...)



<https://www.tiobe.com/tics/fact-sheet/>

# Lernziele

- ☐ Welche wesentlichen Methoden können die Produktqualität verbessern?
- ☐ Wie hängen diese Methoden mit dem Entwicklungsprozess zusammen?
- ☐ Welche Anforderungen werden an Metriken generell gestellt?
- ☐ Welche Arten von Metriken gibt es? Wie unterscheiden sie sich?
- ☐ Wofür eignen sich Zeilenmetriken? Was sind ihre Schwächen?
- ☐ Was messen die Metriken nach Halstead? Wie funktionieren sie?
- ☐ Wie lässt sich die Struktur eines Programmes beschreiben?
- ☐ Was versteht man unter einem Basisblock?
- ☐ Worauf muss man bei der Struktur von JAVA Code achten?
- ☐ Was erfasst die zyklomatische Komplexität nach McCabe?
- ☐ Wie funktioniert die zyklomatische Komplexität?
- ☐ Welche Metriken messen Eigenschaften Objektorientierter Software?
- ☐ Sind Metriken das Maß aller Dinge?
- ☐ Wie kann ich schlecht bewerteten Code verbessern?
- ☐ Was versteht man unter Code-Smells?
- ☐ Wie lassen sich große Klassen, lange Methoden, etc. verbessern?
- ☐ Wer hilft mir dabei von Anfang an alles richtig zu machen?

# Literatur

**Post, Uwe.** *Besser Coden*

2. Aufl. ; Bonn: Rheinwerk Verlag., 2018.

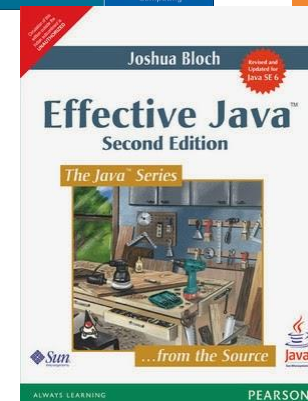
**Hoffmann, Dirk W.** *Software-Qualität*

2. aktualisierte und korrigierte Auflage;  
Berlin ; Heidelberg: Springer Verlag.,  
2013.

**Bloch, Joshua.** *Effective Java*

Upper Saddle River, NJ ; Munich [u.a.]:  
Addison-Wesley.

[TU-Bestand](#)



# Literatur

**Fowler, Martin.** *Refactoring : Wie Sie Das Design Vorhandener Software Verbessern.*

Studentenausg., 2[. Dr.] ed. München [u.a.]: Addison-Wesley, 2006.

[TU-Bestand](#)

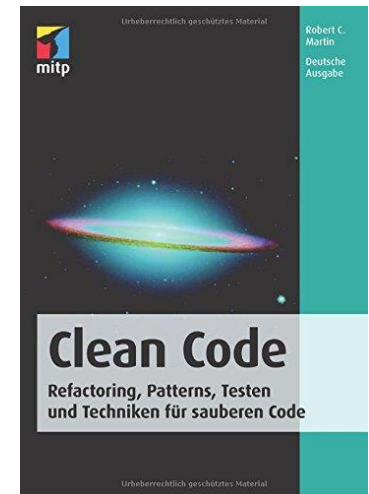
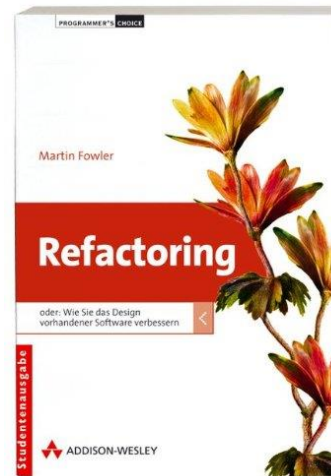
**Martin, Robert C., and Michael C. Feathers.** *Clean Code : Refactoring, Patterns, Testen Und Techniken Für Sauberen Code*

1. Aufl. ; Dt. Ausg. ed. Heidelberg ; München ; Landsberg [u.a.]: Mitp-Verl., 2009.

[TU-Bestand](#)

**Philipps, Jan, and Bernhard Rumpe.** *Roots of Refactoring*

In: Tenth OOPSLA Workshop on Behavioral Semantics: Northeastern University, 2001.



# Tools

## Metrics plugin for Eclipse

net.sourceforge.metrics.feature.group

Frank Sauer, über Marketplace

Metric	Total	Mean	Std. Dev.	Maxim...	Method
> Number of Overridden Methods (avg/max per type)	12	0,364	0,54	2	
> Number of Attributes (avg/max per type)	43	1,303	2,713	11	
> Number of Children (avg/max per type)	19	0,576	1,799	8	
> Number of Classes (avg/max per package/fragment)	33	4,714	3,057	9	
> Method Lines of Code (avg/max per method)	523	4,395	8,253	59	doGet
> Number of Methods (avg/max per type)	118	3,576	3,806	17	
> Nested Block Depth (avg/max per method)		1,143	0,584	5	tryMove
> Depth of Inheritance Tree (avg/max per type)		2,182	1,167	4	
> Number of Packages	7				
> Affluent Coupling (avg/max per package/fragment)		4	2,449	8	
> Number of Interfaces (avg/max per package/fragment)	1	0,143	0,35	1	
> McCabe Cyclomatic Complexity (avg/max per method)		2,101	3,394	23	Board
> Total Lines of Code	1088				
> Instability (avg/max per package/fragment)		0,37	0,283	1	
> Number of Parameters (avg/max per method)		0,798	0,846	4	Move
> Lack of Cohesion of Methods (avg/max per type)		0,122	0,263	0,8	
> Effluent Coupling (avg/max per package/fragment)		2,143	1,457	5	
> Number of Static Methods (avg/max per type)	1	0,03	0,171	1	
> Normalized Distance (avg/max per package/fragment)		0,557	0,273	1	
> Abstractness (avg/max per package/fragment)		0,073	0,089	0,2	
> Specialization Index (avg/max per type)		0,142	0,216	0,545	
> Weighted methods per Class (avg/max per type)	250	7,576	12,962	61	
> Number of Static Attributes (avg/max per type)	4	0,121	0,409	2	

## Control Flow Graph Factory

com.drgarbage.controlflowgraphfactory.feature.group

Eclipse Plug-In zur Erstellung von CFGs aus JAVA Quell- & Bytecode

Dr. Garbage Community, [Link](#)

