HOCHSCHULE HEILBRONN

Software Engineering komplexer Systeme

Vorlesung an der Hochschule Heilbronn 2019

Dr. Andreas Heil

HEUTIGER INHALT

- Metriken
  - Grundlagen
  - Arten von Metriken
- Software-Metriken
  - Ausgewählte Code-Metriken
  - Kombinierte Metriken
  - Entwurfs Metriken

LERNZIELE

Typen von Metriken

- unterscheiden können

Dimensionen von Software-Metriken

- kennen

Code- und Entwurfsmetriken

- unterscheiden können

Geeignete Metriken zu Messung von Software-Komplexität

- auswählen und bewerten können

Software-Metriken

METRIKEN

Metrik – Das meint Wikipedia…

- Metrik steht für:
  - in der Literatur die rhythmische Bestimmung von Texten, siehe Verslehre
  - in der Musik die Lehre von der Bewertung der Töne, siehe Metrum (Musik)
  - ein Fachbereich der Chemie, siehe Chemometrik
  - in der Physik die Geometrie der Raumzeit, siehe Allgemeine Relativitätstheorie
  - in der Betriebswirtschaft eine Methode zur Planung, Steuerung und Kontrolle von Strategien,
  - den Künstlernamen des britischen DJs und Produzenten Tom Mundell

- in der Mathematik:
  - eine Methode, die Größe von Abständen zu ermitteln, siehe metrischer Raum
  - die topologische Verallgemeinerung des Abstandes, siehe Pseudometrik
  - in der Differentialgeometrie eine Maßvorschrift, siehe metrischer Tensor

- in der Informatik:
  - **eine Methode zur Bewertung der Qualität eines Codes, siehe Softwaremetrik**
  - ein Bewertungskriterium beim Routing, siehe Metrik (Netzwerk)

WOZU METRIKEN?

- Metriken erlauben es uns
  - Dinge / Sachverhalte zu **verstehen**,
  - Dinge / Sachverhalte zu **vergleichen**,
  - **Vorhersagen** zu treffen und
  - zu **steuern**

Beispiel Metrik eines E-Autos

Modell	**Renault ZOE ZE40**
Leistung	66 kW
Reichweite	400 KM
Kapazität	41 kWh
Ladestrom	43 kW (IEC Typ2)

Metriken ermöglichen sachlichen Vergleich einer Renault ZOE mit Tesla Model 3

WAS IST MESSBAR?

Business-Metriken

Beispiele:
- Umsatz
- Gewinn nach Steuer
- Marktanteil
- Anzahl verkaufte Lizenzen
- …

Projekt-Metriken (agil)

Beispiele:
- Release Burndown
- Sprint Burndown
- Velocity
- Anzahl Tests
- …

Software-Metriken

Beispiele:
- Lines of Code
- Anzahl Klassen
- Anzahl Methoden pro Klassen
- Komplexität nach McCabe
- Anzahl Fehler
- Testabdeckung
- …

Software-Metriken

GRUNDLAGEN

SOFTWARE-METRIKEN
Definition

> "*Software quality metric: A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality.*"

IEEE 1061-1998 IEEE Standard for a Software Quality Metrics Methodology

SOFTWARE-METRIKEN
Definition

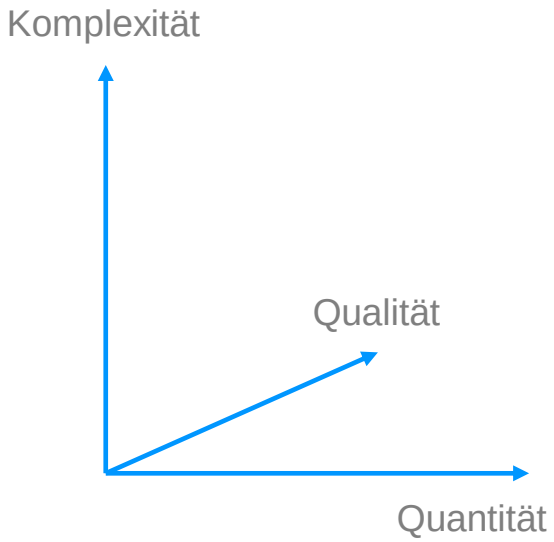
- Eine Software-Metrik ist eine (mathematische) **Funktion**, die einer Eigenschaft von Software einen **Zahlenwert** (auch Maßzahl) **zuordnet**
- Die Maßzahl dient der **qualitativen Bewertung der Software**
- Besser: Die Maßzahl dient der **qualitativen Bewertung des Quellcodes** einer Software
- Noch Besser: Die Maßzahl dient der **qualitativen Bewertung des Quellcodes** einer Software im Rahmen der **statischen Code-Analyse**

SOFTWARE-METRIKEN

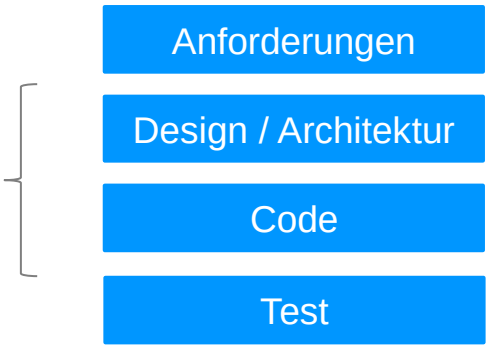
Statische vs. Dynamische Analyse

Was	Wann	Grundlage	Beispiel	Wie
Statische Code-Analyse	Entwicklungs-. oder Compile-Zeit	Sourcecode	Lines of Code Anzahl Klassen …	Zählen Berechnen
Dynamische Code-Analyse	Laufzeit	Laufzeitverhalten	Speicherverbrauch Antwortzeiten Fehlerrate …	Profilling Debugging Testen

SOFTWARE-METRIKEN
Dimensionen und Ebenen



Dimensionen



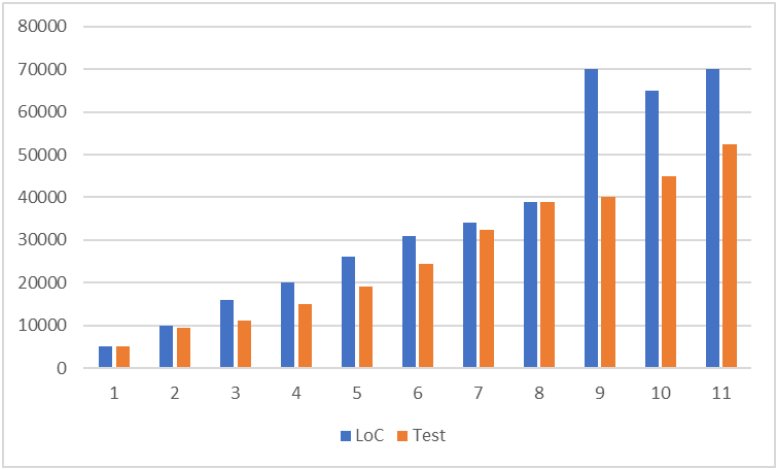
Ebenen

CODE-QUANTITÄT

- Quantitativ messbar
  - Line of Code (LoC)
  - Anzahl Kommentare (pro LoC)
  - Anzahl Klassen
  - Anzahl Methoden pro Klasse
- Praxistipp
  - Ungeeignet für Vergleiche verschiedener Projekte
  - Gut als Indikator in einem Projekt

Progamm	LoC
Unix v1.0	1.000
Space Shuttle	40.000
Windows 3.1	2.500.000
Large Hadron Colllider	3.500.000
Chrome	6.700.000
Linux 3.1	10.000000
Mac OS X	86.000.000
Google Internet Services	2.000.000.000

vs.



Quelle: https://informationisbeautiful.net/visualizations/million-lines-of-code/

CODE-QUALITÄT

Definition: Qualität

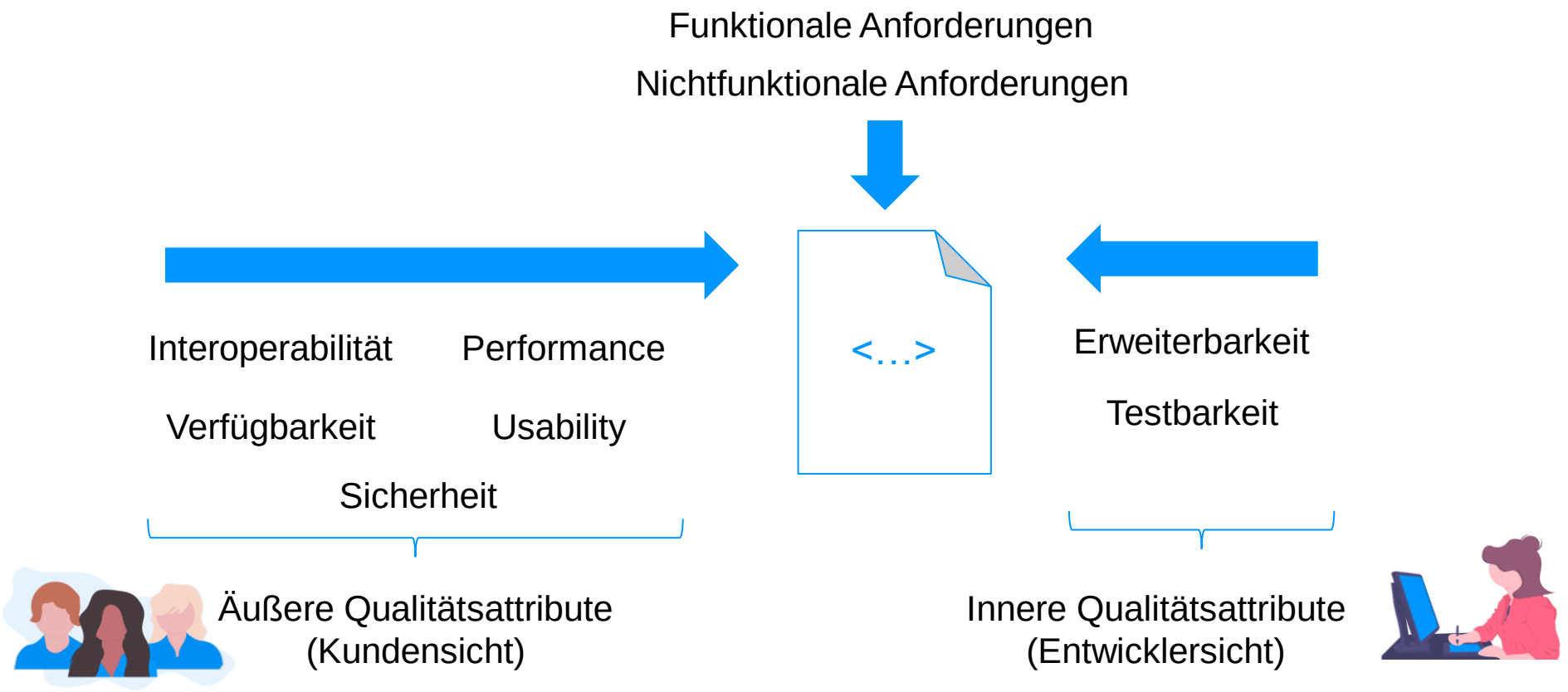
> „Grad der Übereinstimmung mit Anforderungen oder Normen“

Definition: Code-Qualität

> „Grad der Übereinstimmung mit Anforderungen an den Code“

➔Woher kommen Anforderungen an den Code?

CODE-QUALITÄT
Anforderungen an den Code



CODE-QUALITÄT
Messen von Code-Qualität

- Erfüllung von funktionalen Anforderungen
  - Verifikation: Nachweis über Test
  - Validierung: Nachweis über Benutzerakzeptanz

- Erfüllung nichtfunktionaler Anforderungen
  - Nachweis über Tests (Performance-, Lasttests)
  - Nachweis über Review
    - Einhaltung von Architekturvorgaben (Wartbarkeit)
    - Einhaltung von Coding Guidelines (Wartbarkeit, Testbarkeit)
  - Nachweis über statische Code-Analyse
    - Einhaltung von Coding Guidelines (Wartbarkeit, Testbarkeit)
    - Einhaltung von Quantitäts- und/oder Komplexitätsmetriken (Wartbarkeit, Testbarkeit)

CODE-KOMPLEXITÄT

SW-Complexity: Degree of complication of a system or system component determined by factors as the number and intricacy of interfaces, the number and intricacy of conditional branches, the degree of nesting, the types of data structures and other system characteristics.

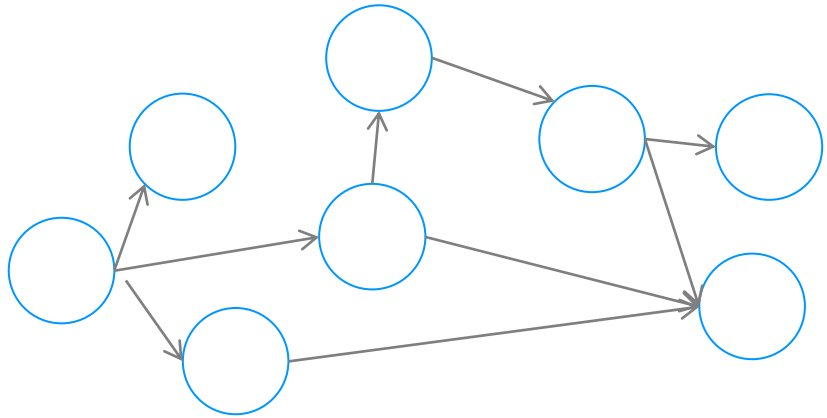
IEEE Standard Glossary, 1983

SW-Complexity: Degree to which a system or component has a design or implementation that is diffcult to understand and verifiy.

IEEE Standard Glossary, 1990

CODE-KOMPLEXITÄT

Struktureller Komplexität



- Ablaufkomplexität
- Datenflusskomplexität

- Assembler oder BASIC Code enthält viele GOTO Befehle, hat somit eine hohe Ablaufkomplexität
- Prozedurale Sprachen verwenden Methoden, haben somit eine geringere Ablaufkomplexität erhöhen aber die Datenflusskomplexität
- Funktionelle Sprachen haben unter Umständen eine noch höhere Datenflusskomplexität

CODE-KOMPLEXITÄT

Semantische Komplexität

- Abhängig von
  - Größe des Wortschatzes und
  - Anzahl der verwendeten Wörter im Code

  - BASIC 14 Keywords
  - C hat ca. 32 Keywords
  - C# 79 Keywords
  - C++ 100 Keywords
  - Assembler > 200 Keywords

SOFTWARE-METRIKEN

AUSGEWÄHLTE CODE-METRIKEN

CODE-METRIKEN

- Code lässt sich sehr einfach analysieren und messen
- Es existieren hunderte Code-Metriken

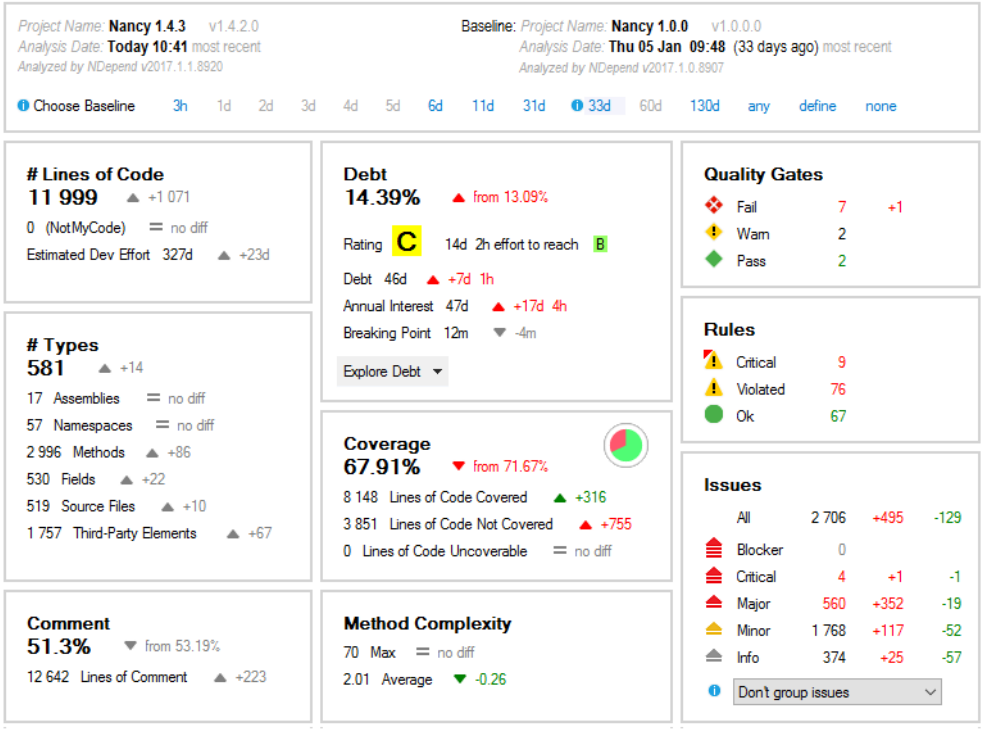
- Beispiel NDepend (Tool zur statischen Code-Analyse von .NET Code)

- **12 metrics on application**:
  NbLinesOfCode, NbLinesOfComment, PercentageComment, NbILInstructions, NbAssemblies, NbNamespaces, NbTypes, NbMethods, NbFields, PercentageCoverage, NbLinesOfCodeCovered, NbLinesOfCodeNotCovered
- **18 metrics on assemblies**:
  NbLinesOfCode, NbLinesOfComment, PercentageComment, NbILInstructions, NbNamespaces, NbTypes, NbMethods, NbFields, Assembly level, Afferent coupling (Ca), Efferent coupling (Ce), Relational Cohesion(H), Instability (I), Abstractness (A), Distance from main sequence (D), PercentageCoverage, NbLinesOfCodeCovered, NbLinesOfCodeNotCovered
- **13 metrics on namespaces**:
  NbLinesOfCode, NbLinesOfComment, PercentageComment, NbILInstructions, NbTypes, NbMethods, NbFields, Namespace level, Afferent coupling at namespace level (NamespaceCa), Efferent coupling at namespace level (NamespaceCe), PercentageCoverage, NbLinesOfCodeCovered, NbLinesOfCodeNotCovered
- **22 metrics on types**:
  NbLinesOfCode, NbLinesOfComment, PercentageComment, NbILInstructions, NbMethods, NbFields, NbInterfacesImplemented, Type level, Type rank, Afferent coupling at type level (TypeCa), Efferent coupling at type level (TypeCe), Lack of Cohesion Of Methods (LCOM), Lack of Cohesion Of Methods Henderson-Sellers (LCOM HS), Code Source Cyclomatic Complexity, IL Cyclomatic Complexity (ILCC), Size of instance, Association Between Class (ABC) Number of Children (NOC), Depth of Inheritance Tree (DIT), PercentageCoverage, NbLinesOfCodeCovered, NbLinesOfCodeNotCovered
- **19 metrics on methods**:
  NbLinesOfCode, NbLinesOfComment, PercentageComment, NbILInstructions, Method level, Method rank, Afferent coupling at method level (MethodCa), Efferent coupling at method level (MethodCe), Code Source Cyclomatic Complexity, IL Cyclomatic Complexity (ILCC), IL Nesting Depth, NbParameters, NbVariables, NbOverloads, PercentageCoverage, NbLinesOfCodeCovered, NbLinesOfCodeNotCovered, PercentageBranchCoverage
- **2 metrics on fields**:
  Size of instance, Afferent coupling at field level (FieldCa)

Quelle: https://www.ndepend.com/docs/code-metrics

CODE-METRIKEN

- Einfache Code-Metriken
  - Lines of Code
  - Comment Ratio
  - Lines of Comment
  - Paramters per Method
  - Nesting Level
  - Lines of Code Covered
  - Number of Fields
  - …

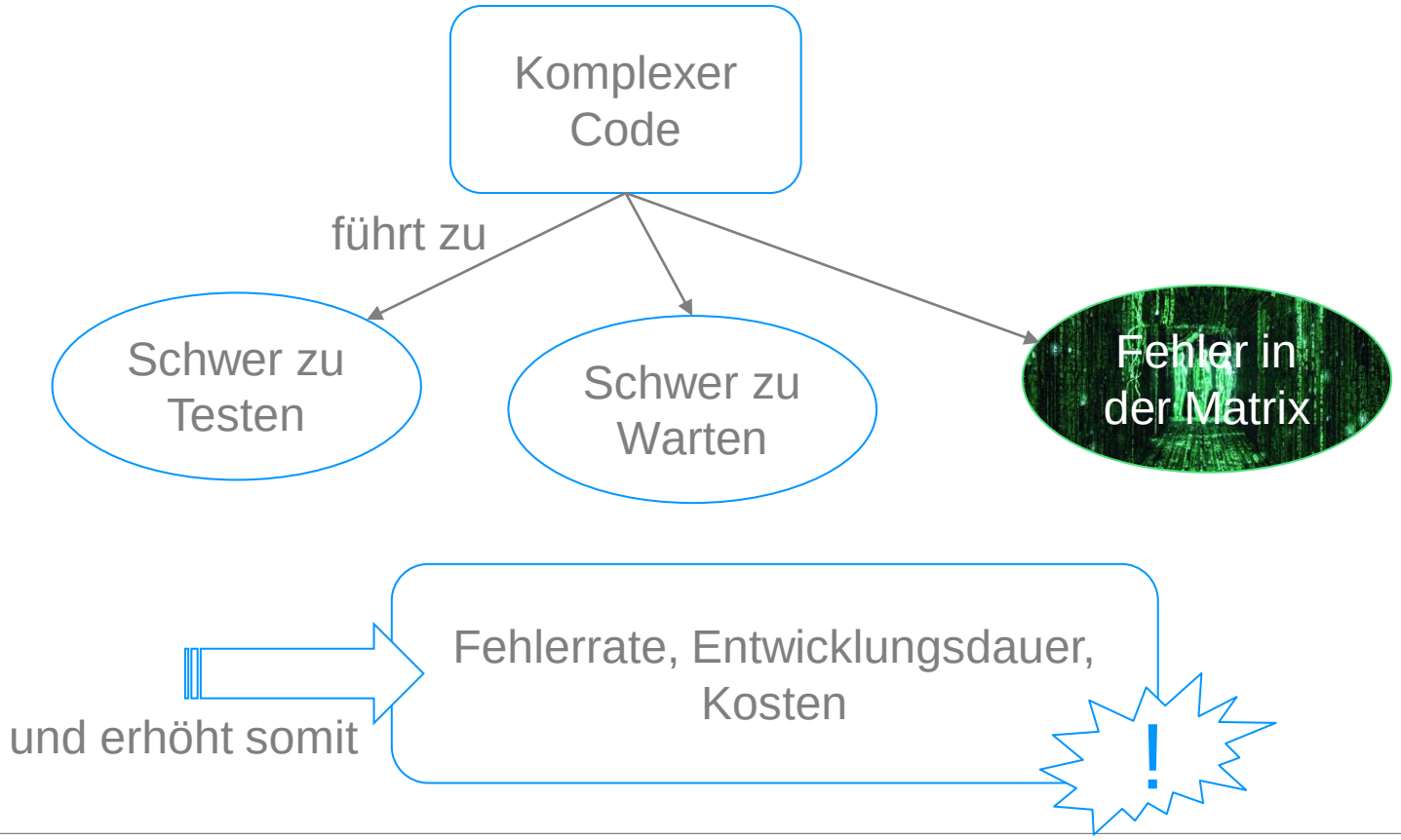


Quelle: https://www.ndepend.com/docs/code-metrics

SOFTWARE-METRIKEN

ZYKLOMATISCHE KOMPLEXITÄT NACH MCCABE

ZYKLOMATISCHE KOMPLEXITÄT
Motivation



ZYKLOMATISCHE KOMPLEXITÄT
Grundlagen

Zyklomatische Komplexität nach McCabe (engl. Cyclomatic Complexity)

- Beschreibt die Komplexität des Kontrollflusses von Programm-Code

- Durch welche Sprachkonstrukte lässt sich der Kontrollfluss beeinflussen?
  - Schleifen (z.B. `for`, `do`, `while`)
  - Verzweigungen (z.B. `if`, `switch`, `goto`)

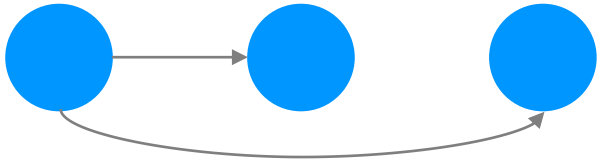
- Basiert auf Graphentheorie
  - Grundlage sind die unabhängigen Ausführungspfade in Code
  - Anzahl Kanten E (engl. edges)
  - Anzahl Knoten V (engl. vertices)

ZYKLOMATISCHE KOMPLEXITÄT
Kontrollfluss

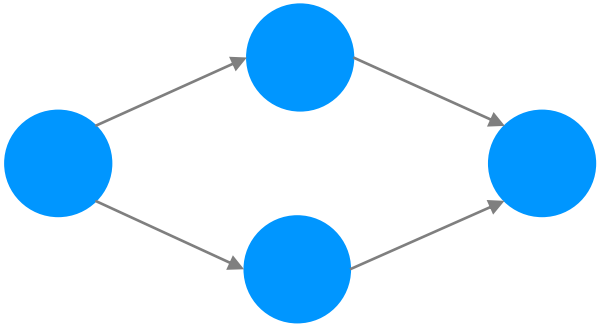
Beispiele für den Kontrollfluss verschiedener Sprachkonstrukte

Sequenz

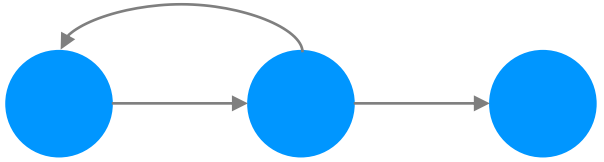
While-Schleife



If-Then-Else

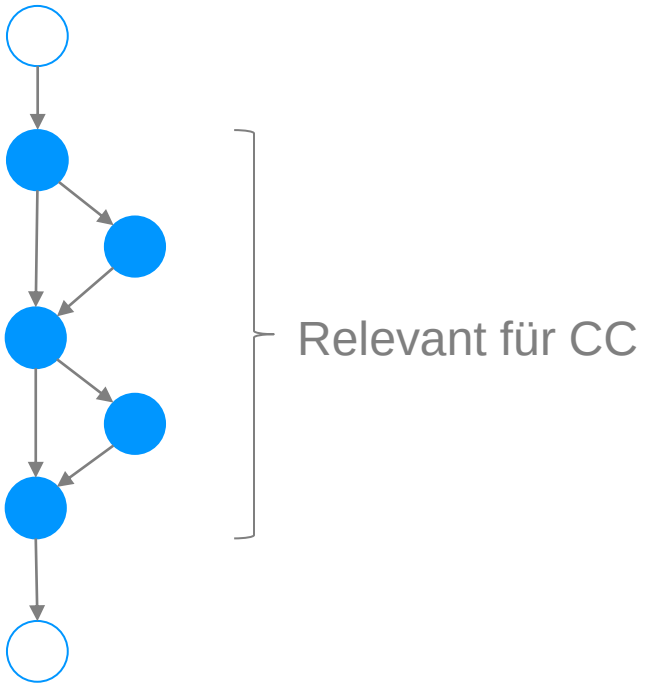


Do-Until



ZYKLOMATISCHE KOMPLEXITÄT
Code-Beispiel

```
public int myMethod(int x, int y) {
   int retVal = x;
   if (x > 5 ) {
        retVal = 5;
   }
   if (y < 5) {
        retVal = y;
   }
   return retVal;
}
```



ZYKLOMATISCHE KOMPLEXITÄT
Zählweise

- Jedes `if`-Statement führt einen neuen Zweig ein und erhöht die CC um den Wert 1
- Iterationskonstrukte wie `for-` und `while`-Schleifen haben ebenfalls neue Zweige zur Folge und erhöhen die CC
- Jede `case`-Anweisung in `switch`-Anweisungen erhöht die CC um eins
- Eine `case`-Anweisungen selbst erhöht die CC nicht, da die Anzahl der Zweige im Kontrollfluss nicht erhöht werden
- Bei zwei oder mehr `case`-Anweisungen, die keinen Code beinhalten wird die CC lediglich um den Wert 1 für die Gesamtheit dieser `case`-Anweisungen inkrementiert;
- Jede `catch`-Anweisung in einem `try`-Block erhöht die CC um 1
- Die Anweisung `ausdurck1 ? ausdruck2 : ausdruck3` erhöhte die CC (vgl. if-Statement)

ZYKLOMATISCHE KOMPLEXITÄT
Zählweise

Was ist mit unkoditionellen Verzweigungen?

- `goto-`, `return-` und `break-`Anweisungen werden nicht bei der Berechnung berücksichtigt, obwohl sie die Komplexität erhöhen

- Zusammenfassung der typischen Sprachkonstrukture für die Berechnung

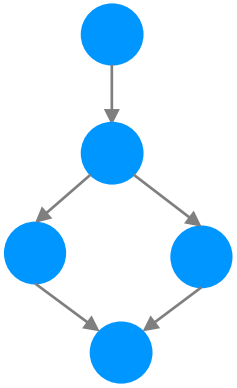
```
if (...), for (...), while (...),
case ...:, catch (...), &&, ||, ?,
#if, #ifdef, #ifndef, #elif
```

ZYKLOMATISCHE KOMPLEXITÄT
Berechnung

Hinweis: Unterschiedliche Berechnungen möglich

$$V(G) = e - v + 2$$

e: Anzahl Kanten und *v*: Anzahl Knoten



V(G) = *e* - *v* + *2* = 5 - 5 + 2 = 2

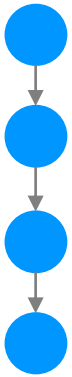
V(G) = 1

ZYKLOMATISCHE KOMPLEXITÄT
Berechnung

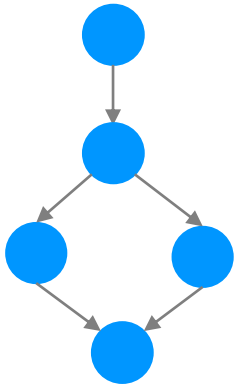
Alternative Berechnung

$$V(G) = b + 1$$

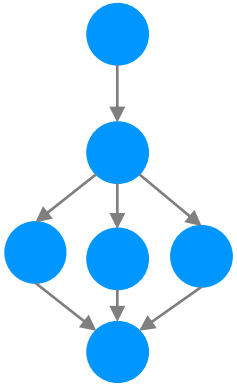
e: Anzahl Kanten und *v*: Anzahl Knoten, n-fache Mehrfachverzweigung kann aufgeteilt werden *in b = n – 1* Binärverzweigungen



Keine Verzweigung
V(G) = 0 + 1 = 1



Eine Binärverzweigung
V(G) = 1 +1 = 2

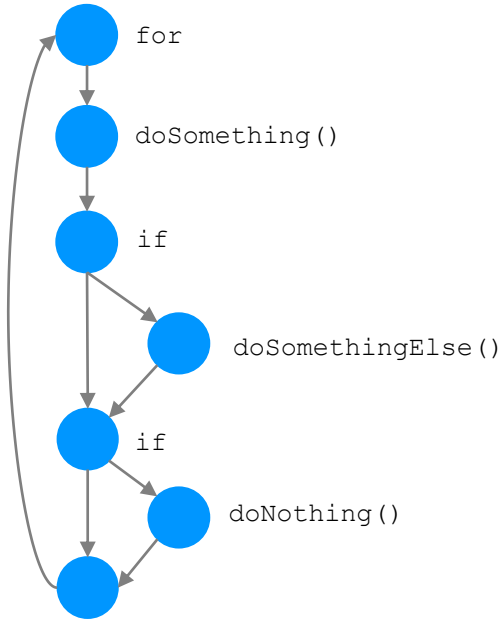


Eine Dreifachverzweigung
V(G) = 2 +1 = 3

ZYKLOMATISCHE KOMPLEXITÄT

Beispiel

```
for (int i = 0; i < 10; i++) {
    doSomething();
    if (i == 5)
        doSomethingelse();
    if (i == 8)
        doNothing();
}
```



Variante 1:

- Anzahl Kanten: 9
- Anzahl Knoten: 7
- V(G) = 9 – 7 + 2 = 4

Variante 2:

- Anzahl Binärverzweigungen
- V(G) = 3 + 1 = 4

Variante 3:

- Starte mit V(G) = 1, erhöhe V(G) um 1 bei Operatoren: `if`, `for`, `do`, `while`, `case`, `&&`, `||`
- V(G) = 4

ZYKLOMATISCHE KOMPLEXITÄT

Empfehlung

- V(G) < 10 — Einfach, verstehbar, testbar
- V(G) < 20 — komplexeres Programm
- V(G) > 20 — komplex, ggf. schwer zu verstehen

ACHTUNG:

Hohe zyklomatische Komplexität bedeutet nicht immer schlechte Wartbarkeit

```
Public string GetMonth(int m) {
String month = new String();
  Switch (m) {
    case 1: month = „Januar“;
    ....
    case 12: month = „Dezember“;
    default: month = „unkown“;
  }
  Return month;
}
```

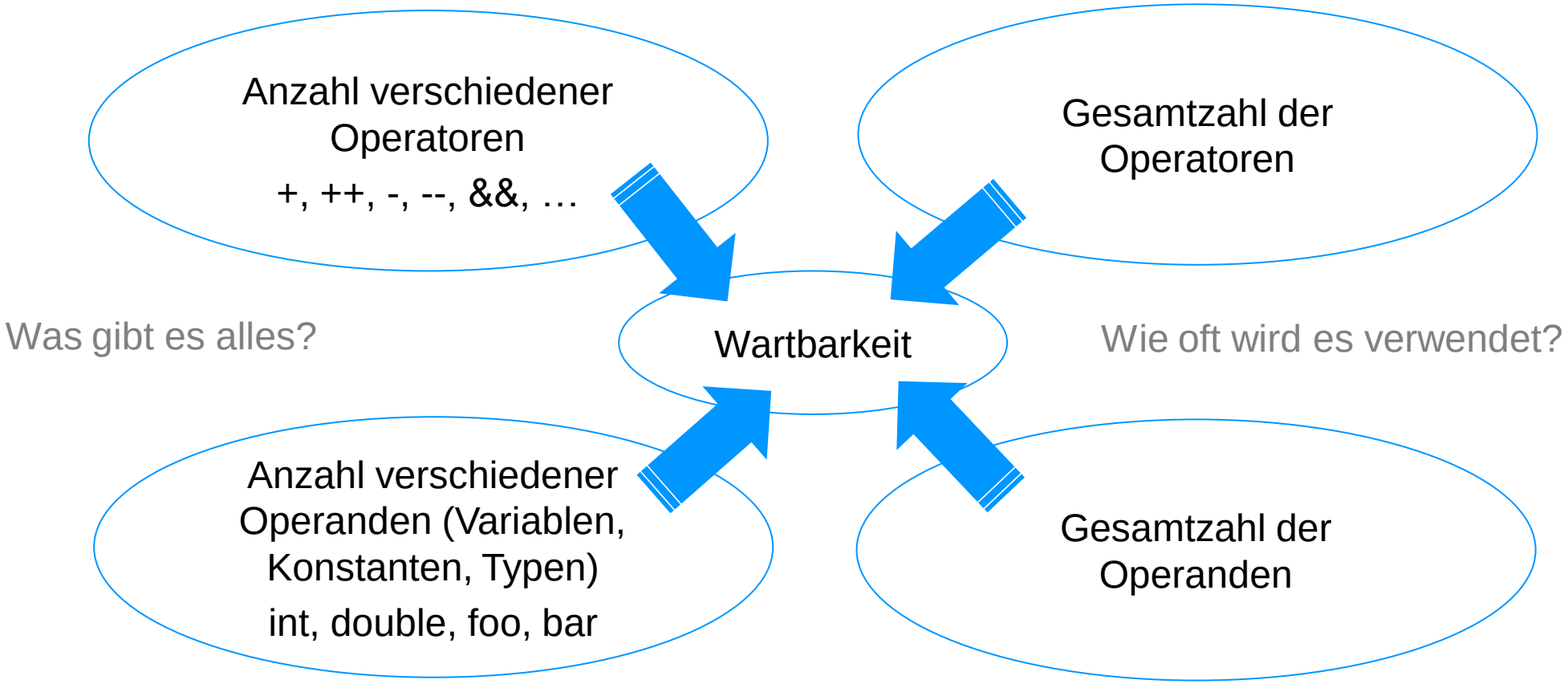
V(G) = 1 + 12 x „`case`“

V(G) = 13

SOFTWARE-METRIKEN

HALSTEAD METRIKEN

HALSTEAD METRIKEN
Motivation



HALSTEAD METRIKEN

Grundlagen

n_1: die Anzahl der verschiedenen Operatoren

n_2: die Anzahl der verschiedenen Operanden

N_1: die Gesamtanzahl der Operatoren

N_2: die Gesamtanzahl der Operanden

Alle weiteren Halstead-Maße berechnen sich aus dem im Weiteren vorgestellten Formeln

HALSTEAD METRIKEN

Operanden

- Identifier, die keine reservierten Wörter sind
  - foo, bar
- Type Names
  - public class TYPENAME
- Type Specifiers
  - bool, char, double, float, int, long, short, signed, unsigned, void
- Constants
  - Zeichenkonstanten, numerische oder String-Konstanten

HALSTEAD METRIKEN
Operatoren

- Typische Operatoren
  - `!` `!=` `%` `%=` `&` `&&` `||` `&=` `( )` `*` `*=` `+` `++` `+=` `,` `-` `--` `=` `>` `.` `...` `/` `/=` `:` `::` `<` `<<` `<<=` `<=` `=` `==` `>` `>=` `>>` `>>=` `?` `[ ]` `^` `^=` `{` `}` `|` `|=` `~`
- In C++ (GMT++) reservierte Wörter
  - `asm, break, case, class, continue, default, delete, do, else, enum, for, goto, if, new, operator, private, protected, public, return, sizeof, struct, switch, this, union, while, namespace, using, try, catch, throw, const_cast, static_cast, dynamic_cast, reinterpret_cast, typeid, template, explicit, true, false, typename`
- Sonderbehandlung bei Kontrollstrukturen
  - `case ...:` `for (...)` `if (...)` `switch (...)` `while for (...)` und `catch (...)`
    Der Doppelpunkt und die Klammern werden als Teil des Konstrukts betrachtet. Case und der Doppelpunkt oder `for (...)` `if (...)` `switch (...)` `while for (...)` und *`catch (...)`* sowie die Klammern werden zusammen als ein Operator gezählt.

HALSTEAD METRIKEN
Beispiel

```
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;
}
```

n_1: unterschiedliche Operatoren: 12

n_2: unterschiedliche Operanden: 4

N_1: verwendete Operatoren: 22

N_2: verwendete Operanden: 16

HALSTEAD METRIKEN

1. Programmlänge (N)
   - engl. program length
   - Summe der Gesamtzahl aller Operatoren und Operanden eines Programms
   - $N = N_1 + N_2$

2. Vokabulargröße (n)
   - engl. vocabulary size
   - Anzahl der verschiedenen Operatoren und Operanden
   - $n = n_1 + n_2$

HALSTEAD METRIKEN

3. Programmvolumen (V)
   - engl. program volume
   - V = N * log2(n)
   - Informationsgehalt der Software gemessen in mathematischen Bits
   - Berechnung indem die Programmlänge mit dem Zweierlogaritmus der Vokabulargröße (n) multipliziert wird
   - Berechnung erfolgt mit Hilfe der Anzahl der ausgeführten Operationen und der Bearbeiteten Operanden im Algorithmus
   - Der Wert V ist daher im Vergleich zu den Zeilenmetriken (LoC) weniger vom Code-Layout abhängig.
   - Richtwert:
     - mind. 20 (paramterlose Methode mit einer nichtleeren Zeile)
     - max. 1.000 (bei > 1.000 macht die Methode vermutlich zu viel)

HALSTEAD METRIKEN

4. Schwierigkeitsgrad / auch Fehlerneigung (D)
   - engl. difficulty level
   - $D = (n_1 / 2) * (N_2 / n_2)$
   - Proportional zur Anzahl versch. Operatoren
   - Proportional zum Verhältnis der Gesamtzahl der Operatoren und der Anzahl der verschiedenen Operanden
   - Bedeutung: Wird ein Operand mehrmals im Programm benutzt, wird es dadurch fehleranfälliger

HALSTEAD METRIKEN

4. Programmierniveau (L)
   - engl. program level
   - L = 1 / D
   - Kehrwert des Schwierigkeitsgrades
   - Bedeutung: Programme mit niedrigem Niveau sind fehleranfälliger

5. Implementieraufwand (E)
   - engl. efftort to implement
   - E = V * D
   - Proportional zum Volumen und Schwierigkeitsgrads eines Programms

HALSTEAD METRIKEN

6. Implementierzeit (T)
   - engl. Time to implement
   - T = E / 18
   - Zeit, die notwendig ist, ein Programm zu verstehen
   - Propertional zum Implementieraufwand
   - Empirische Untersuchung: 18 liefert einen “guten” Wert für E in Sekunden

5. Ausgelieferte Bugs (B)
   - engl. Halsteads delivered bugs
   - $B = E^{2/3} / 3.000$
   - Schätzung für Fehler in Implementierung
   - Bedeutung: Beim Testen sollen mind. So viele Fehler gefunden werden wie die Metrik „vorhersagt“

SOFTWARE-METRIKEN

KOMBINIERTE METRIKEN

KOMBINIERTE METRIKEN
Maintainability Index

- Beschreibt wie wartbar der Quell-Code ist
- Zusammengesetzte Metrik aus
  - Lines of Code (LOC)
  - Zyklomatischer Komplexität (CC)
  - Halstead Volume (V)
- Eine abgewandelte Variante des Software-Engineering Institutes der Carnegie Mellon University (SEI) nutzt darüber hinaus
  - Precent of Lines of Commment (CM)
- Findet u.a. in Visual Studio Verwendung
- Interpretation
  - MI < 65 schlechte Wartbarkeit
  - MI ≥ 85 gute Wartbarkeit.

KOMBINIERTE METRIKEN

Maintainability Index

Original aus: *Oman, Paul, and Jack Hagemeister. "Metrics for assessing a software system's maintainability." Conference on Software Maintenance. IEEE, 1992.*

$$MI = 171 - 5{,}2 \cdot \ln(V) - 0{,}23 \cdot CC - 16{,}2 \cdot \ln(LOC)$$

Variante des SEI:

$$MI = 171 - 5{,}2 \cdot log_2(V) - 0{,}23 \cdot CC - 16{,}2 \cdot log_2(LOC) + 50 \cdot \sin(\sqrt{2{,}4 \cdot CM})$$

Variante in Microsoft Visual Studio (ab 2008/2010)

$$MI = MAX(0, (171 - 5{,}2 \cdot \ln(V) - 0{,}23 \cdot CC - 16{,}2 \cdot \ln(LOC)) \cdot 100/171)$$

KOMBINIERTE METRIKEN

Maintainability Index – Kritische Betrachtung

- Nicht objektiv, da unterschiedliche Werkzeuge unterschiedliche Formeln verwenden
- Formel für Entwickler nur schwer zu verstehen bzw. zu Interpretieren
  - Schwer beurteilbar welcher Auswirkung eine Code-Änderung auf die Wartbarkeit (lt. MI) haben könnte
- Auswirkungen bei Code-Änderungen unklar
  - Refactoring z.B. bei extrahieren einer Methode, um Code zu vereinfachen erhöht die LOC und verschlechtert nach MI die Wartbarkeit
- Keine Aussage über Maßnahmen
  - Ein beliebiger Wert, z.B. 75, des MI gibt keine Auskunft welche Entwicklungsaktivitäten zu einer Verbesserung der Wartbarkeit führen können

SOFTWARE-METRIKEN

ENTWURFS METRIKEN

ENTWURFSMETRIKEN

OO-Metiken nach Chidamer and Kemerer

- Methoden pro Klasse (engl. *methods per class*)
- Vererbungstiefe (engl. *depth of inheritance tree* / DIT)
- Anzahl der Unterklassen (engl. *number of children* / NOC)
- Kopplung der Klassen (engl. *coupling between object classes* / COB)
  - via Methodenaufrufe, Zugriffe auf Felder, Vererbung, Argumente, Rückgabewerte und Exceptions
- Zusammenhalt der Klasse (engl. cohesion among methods of class / CAM)
  - Summe der verschiedenen Typen in jeder Methode, dividiert durch Anzahl der verschiedenen Methodenparamter der gesamten Klassen multipliziert mit der Anzahl der Methoden
  - Ergibt einen Wert zwischen 0 und 1 wobei nahe 1 der präferierte Wert ist

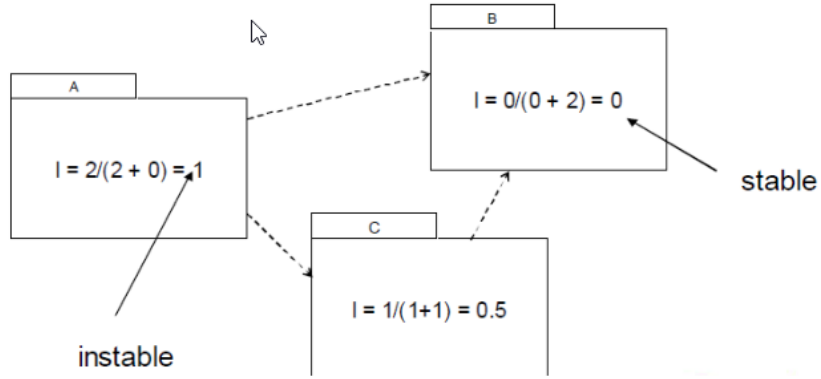
ENTWURFSMETRIKEN

Komponenten-Metriken nach Robert C. Martin

- Afferent Couplings (Ca): (Afferent = incoming)
  - Anzahl an Klassen anderer Komponenten, welche von Klassen
  - dieser Komponente abhängen.

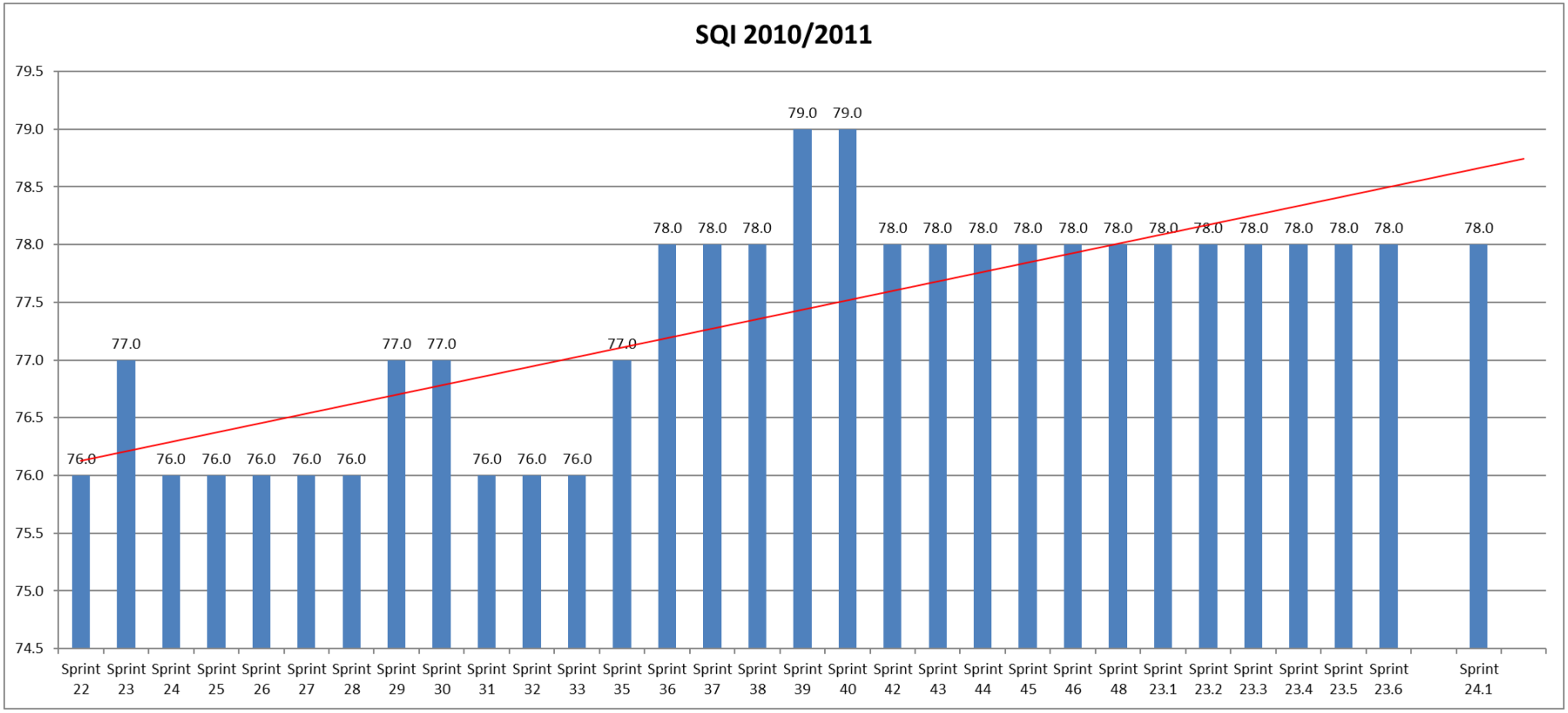
- Efferent Couplings (Ce): (Efferent = outgoing, herausführend)
  - Anzahl an Klassen anderer Komponenten, von welchen Klassen dieser Komponente abhängen.
  - Indikator bzgl. externer Abhängigkeiten.

- Instability (I): I = Ce / (Ce + Ca).
  - I=0 → stabile Komponente
  - I=1 → instabile Komponente



SOFTWARE-METRIKEN

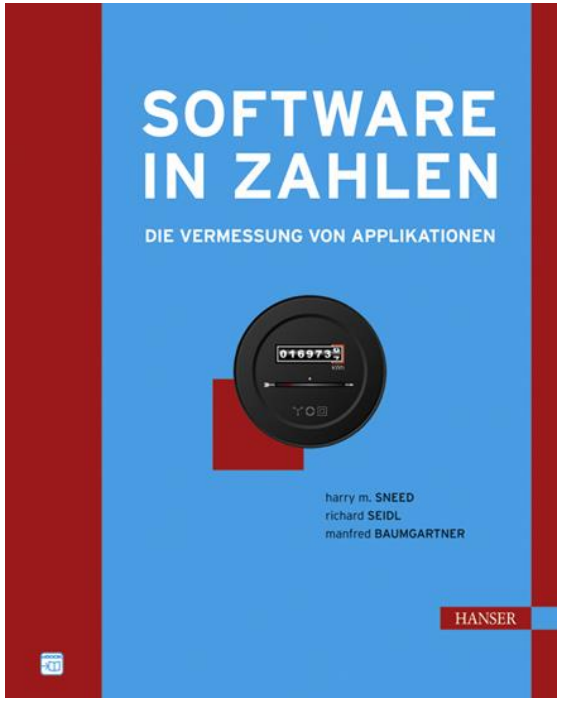
Ein Beispiel aus dem echten Leben



SOFTWARE-METRIKEN
Zusammenfassung

- Metriken sind kein Selbstzweck
- Messung muss einem konkreten Ziel folgen (was soll erreicht werden)
- Maßnahmen müssen klar formuliert werden (was, wann)
- Nur auf Metrik hinzuarbeiten macht nur wenig Sinn
- Metriken als Indikator verwenden

WEITERFÜHRENDE LITERATUR



Harry M. Sneed, Richard Seidl, Manfred Baumgartner

Software in Zahlen - Die Vermessung von Applikationen

Hanser Verlag

ISBN 978-3-446-42175-2

Fragen bis hier her?