

- Einführung
- Software Fehler
- Konstruktive Qualitätssicherung
- Software Test
- ➔ ■ Statische Analyse

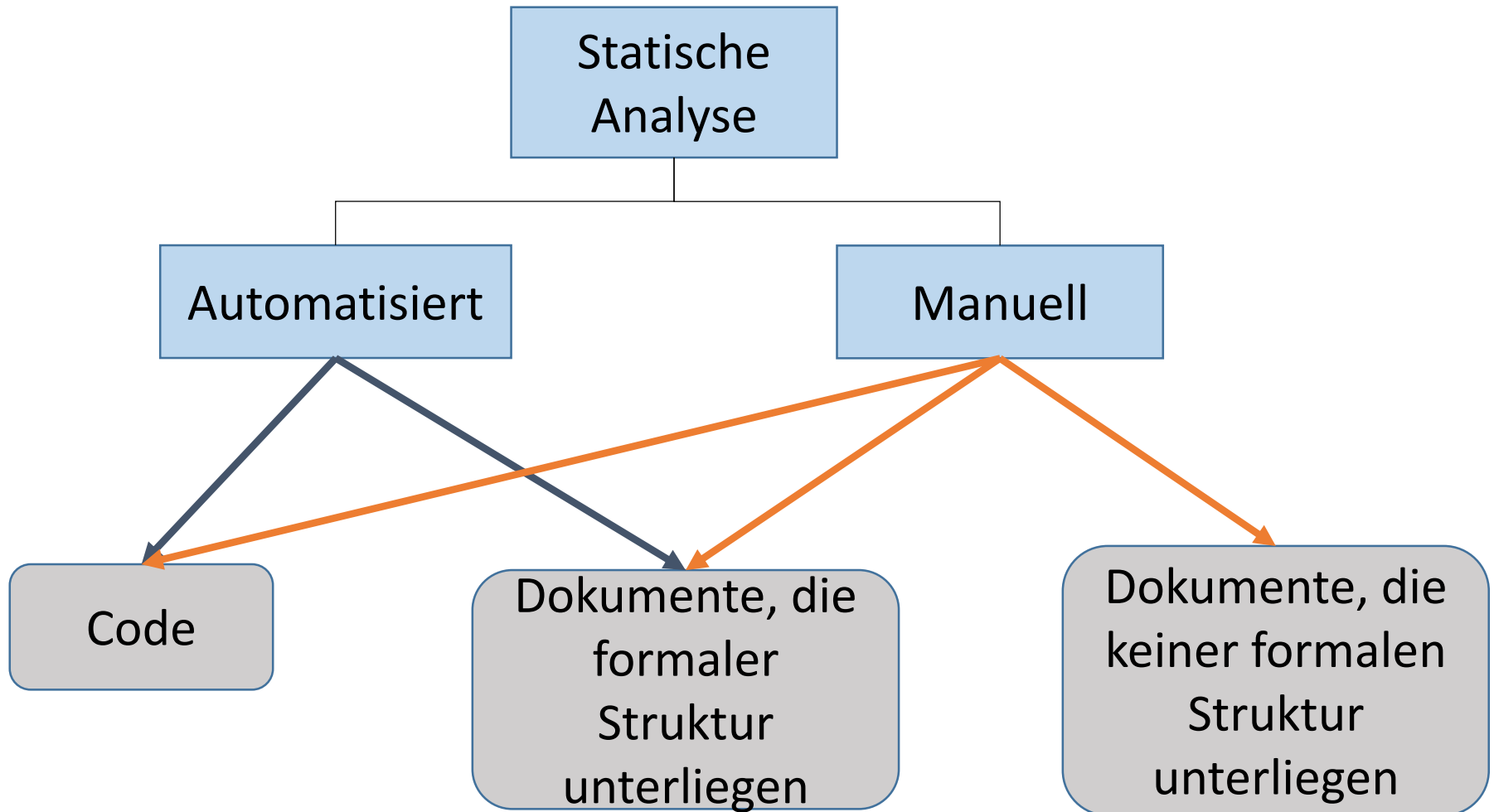


Voriges Kapitel: Tests, d.h. dynamische Tests, Analyse mithilfe des ausgeführten Programms

Jetzt: Statische Methoden: Sichtung der Quelltexte, kann automatisiert oder manuell erfolgen.



Statische Analyse



Vorteile von Software Inspektionen


- Beim Testen können Fehler andere Fehler überdecken.
- Sie können unvollständige Versionen inspizieren aber nicht ohne weiteres testen.
- Sie können über die Fehlersuche hinaus Qualitätsmerkmale prüfen.

Nachteile von SW Inspektionen

- Inspektionen können prüfen, ob Spezifikationen eingehalten werden, aber nicht, ob die wirklichen Benutzeranforderungen erfüllt sind.
- Inspektionen können nicht oder nur schwer die nicht-funktionalen Charakteristiken überprüfen (Performance, Usability etc.).
- → Inspektionen und Tests ergänzen sich.



Statische Code Analyse

- 
- Manuelle Prüfung
 - Software Metriken
 - Konformitätsanalyse
 - Exploit Analyse
 - Anomalienanalyse



1. Planung

- Definition der Reviewobjekte
- Definition der Reviewer
- Definition der Eingangs- und Ausgangskriterien

2. Einführung

- Alle am Review beteiligten Personen erhalten die nötigen Informationen.
- Klarstellung von Bedeutung, Sinn, Zweck und Ziele des durchzuführenden Reviews.

3. Vorbereitung

- Die Reviewer bereiten sich auf die Reviewsitzung individuell vor. Mängel, Fragen, Kommentare werden notiert.

4. Reviewsitzung

- Durchführung durch einen Moderator.
- Ziel: Beurteilung des Prüfobjekts in Bezug auf die Einhaltung und Umsetzung der Vorgaben und Richtlinien sowie das Aufzeigen von Fehlern, Abweichungen und Unstimmigkeiten
- Ergebnis: Objekt wird akzeptiert, mit noch zu erbringenden Änderungen akzeptiert, nicht akzeptiert
- Ergebnis muss von allen Gutachtern mitgetragen werden.

5. Überarbeitung

- Der Autor arbeitet die gewünschten Änderungen ein.

6. Nachbereitung

- Kontrolle der eingearbeiteten Änderungen.

Statische Code Analyse

- Manuelle Prüfung
- ➔ ■ Software Metriken
- Konformitätsanalyse
- Exploit Analyse
- Anomalienanalyse



Software Metriken

*You can't manage what you can't control,
and you can't control what you don't measure.
To be effective software engineers or software
managers, we must be able to control
software development practice. If we don't
measure it, however, we will never have that
control.*

Tom DeMarco



- Erinnerung: Qualitätsmerkmale von Software, Bsp. Transparenz.
- Was bedeutet z.B. „hohe Transparenz“?
- Wie kann man das messen?
- ➔ Software Metriken



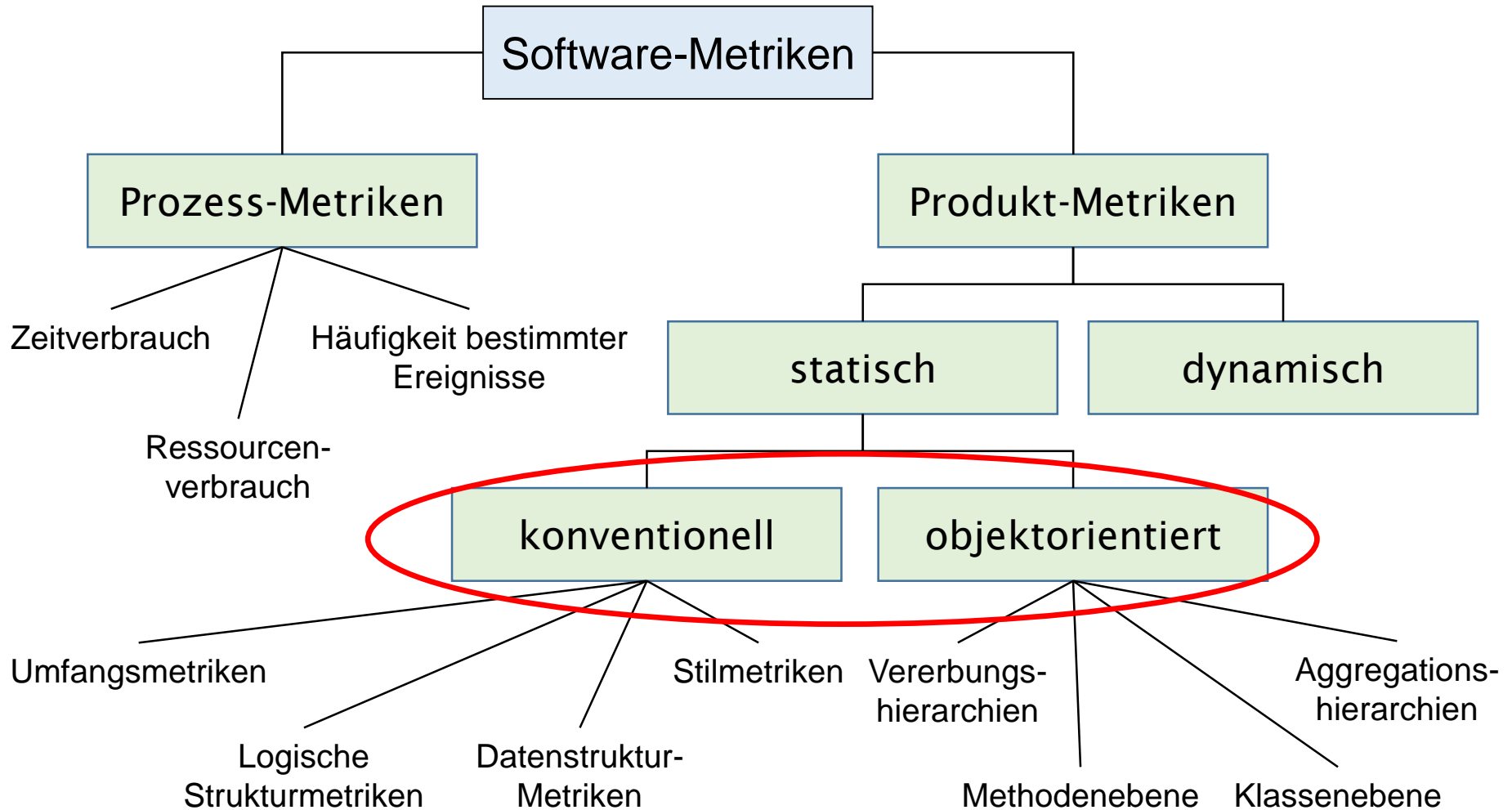
Definition

Die Definition IEEE 1061, 1992 besagt:

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



Klassifikation



- **Umfangsmetrik:** Messen die Größe eines Programms beispielweise in Größe der Sourcecode-Datei in Lines-of-Code (LOC) oder die Umfangsmetrik nach Halstead.
- **Logische Strukturmetriken:** Messen die Struktur wie Anzahl der Pfade, Tiefe der Verschachtelungen.
- **Datenstrukturmetriken:** Messen die Verwendung von Daten im Programm wie Anzahl der Variablen, deren Gültigkeit und Lebensdauer sowie Referenzierung.
- **Stilmetriken:** Messen des Anteils der Kommentare, die Einhaltung von Namenskonventionen.



- Messung der Eigenschaften einzelner Methoden, wofür häufig konventionelle Methoden wie Lines-Of-Code (LOC) oder das Verfahren nach McCabe (siehe später) eingesetzt werden.
- Klassen-Merkmale; wobei die Komplexität der Methode einer Klasse gewichtet wird.
- Hierarchien von Vererbungen
- Aggregation: Verknüpfung der Klassen.



■ Probleme in der Praxis

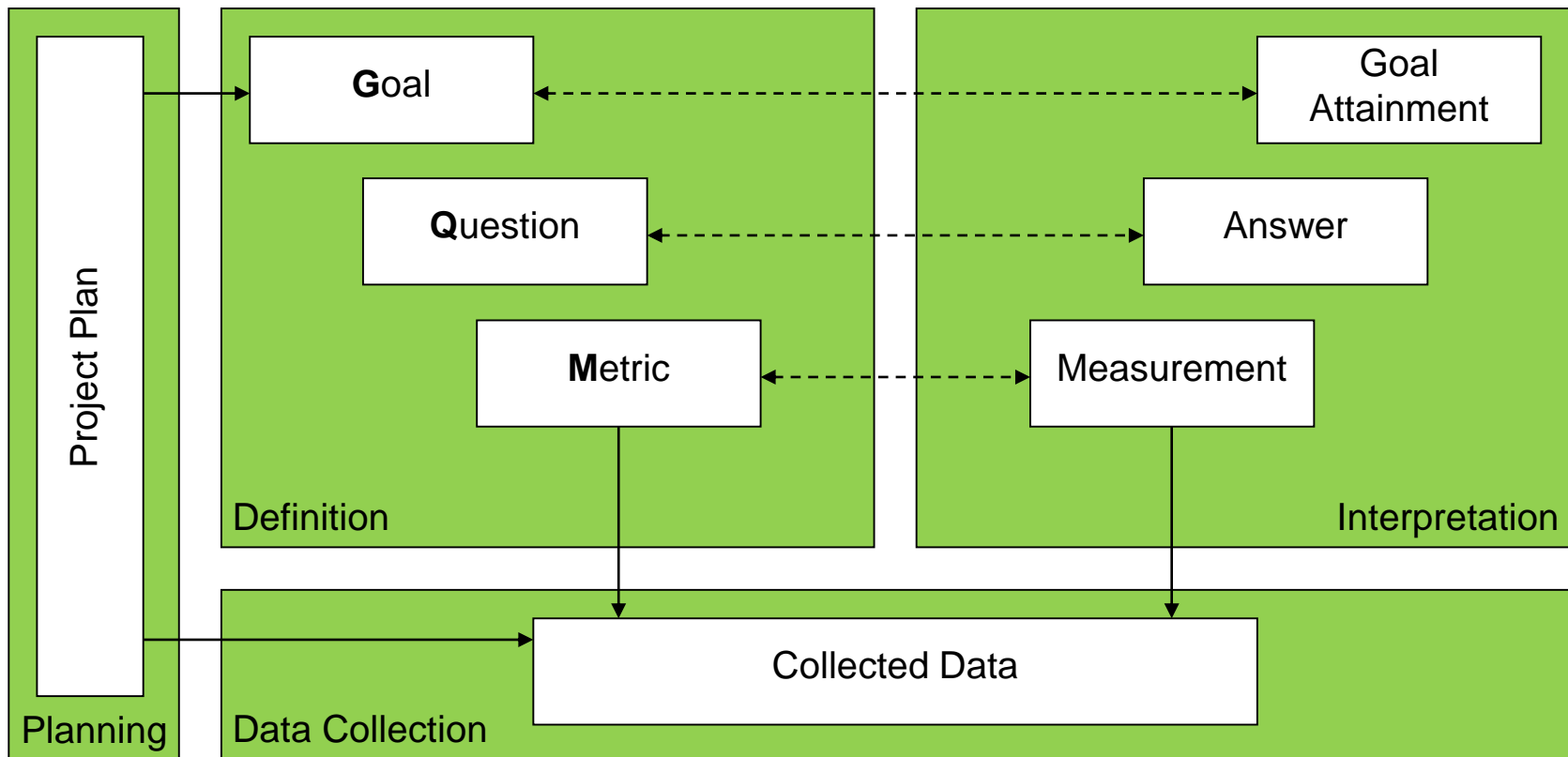
- Nutzen von Metriken oft unklar.
- Fehlen von Standards.
- Programmierer wehren sich dagegen.

■ Durchführung von Messungen

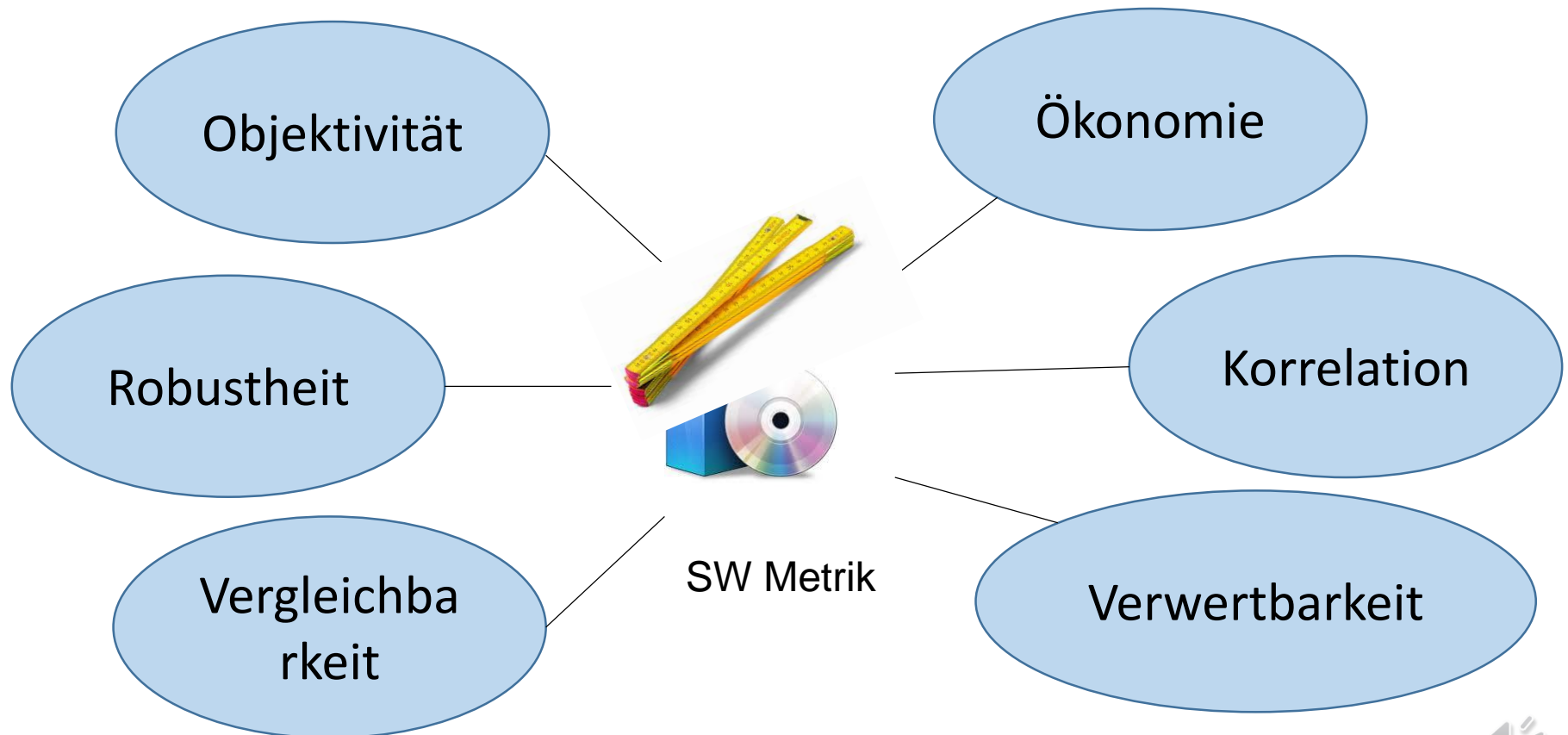
- Messgrößen werden definiert und Messwerte gesammelt.
 - Welche Ziele verfolgt man?
- Messwerte werden analysiert, interpretiert und beurteilt
 - Gibt es eine Bewertungsskala?
- Gefahr von schwer interpretierbaren Zahlenfriedhöfen
 - → Zielorientiertes Messen!



GQM (Goal-Question-Metric)- Ansatz



Gütekriterien von Software Metriken



Gütekriterien - Detail

1. **Objektivität:** frei von subjektiven Einflüssen
2. **Robustheit:** Bei Wiederholung immer das gleiche Ergebnis
3. **Vergleichbar:** Verschiedene Messungen der gleichen Komponente müssen in Relation gesetzt werden können und dadurch für die Steuerung geeignet sein.
4. **Ökonomisch:** billig, sonst macht's keiner
5. **Korrelation:** Aussage der Metrik in Bezug auf die relevante Kenngröße
6. **Verwertbarkeit:** Unterschiedliche Messergebnisse führen zu unterschiedlichem Handeln.



Verwendung von Metriken

- Beurteilung von selbst entwickelter Software.
- Beurteilung fremder Software/neuer Technologien.
- Kostenabschätzungen.
- Beurteilung des Grads, in dem ein Programmierparadigma umgesetzt ist.
- Finden von kritischen Stellen im Code.



Metriken - Beispiele

- 
- LoC/NCSS
 - Halstead Metriken
 - McCabe Metrik
 - Objektorientierte Metriken



Einfache Metriken: LoC, NCSS

- **LoC**: Lines of Code
Maßzahl für Programmkomplexität
- **NCSS**: Non Commented Source Statements
Wie LoC, aber ohne Kommentare mitzuzählen

+: Einfach, schnell automatisiert zu erheben

- : Mangelnde Vergleichbarkeit für Module unterschiedlicher Sprachen

Dennoch Bedeutung als Basisparameter und Abschätzung der Verantwortung eines Programmierers (für die Projektplanung).



- LoC/NCSS
- ➔ ■ Halstead Metriken
- McCabe Metrik
- Objektorientierte Metriken

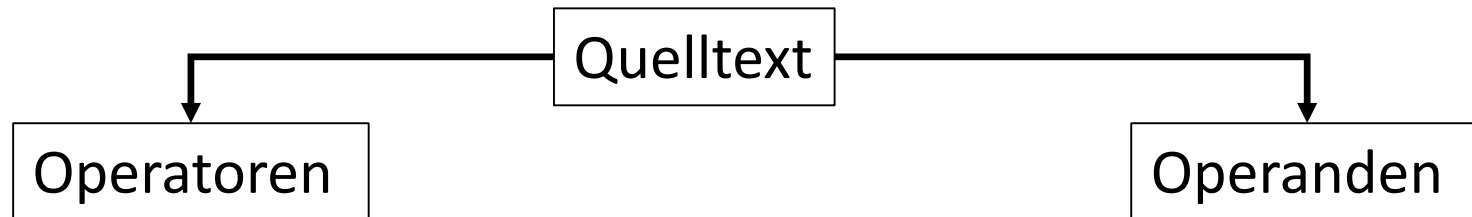


Halstead Metriken

- Metriken, die auf der lexikalischen Struktur eines Programms aufbauen.
- Interpretation beruht auf empirischen Zusammenhängen.



Grundlage der Halstead Metriken: Aufteilung des Source Codes in Operatoren und Operanden



- Schlüsselwörter
- Operatoren
- Öffnende und schließende Klammern
- ...

- Bezeichner
- Konstanten
-



Halstead Metriken: Basisgrößen



Basis
Größen

η_1 ■ Anzahl unterschiedlicher Operatoren

η_2 ■ Anzahl unterschiedlicher Operanden

N_1 ■ Gesamtzahl der Vorkommen aller Operatoren

N_2 ■ Gesamtzahl der Vorkommen aller Operanden

Erweite
rung

η_2^* ■ Anzahl der Ein- und Ausgabeoperanden



Übersicht der Halstead Metriken

- Unique Operators and Operands: $n1, n2$
- Total amount of Operators and Operands: $N1, N2$
- Program vocabulary: Wie LoC, aber formatierungsu
nabhängig
- Program length: $n = n1 + n2$
- Calculated program length: $N = N1 + N2$
- Program volume: $\hat{N} = n1 * \log_2(n1) + n2 * \log_2(n2)$
- Program difficulty level: $V = N * \log_2(n)$
- Program level: $D = (\frac{N1}{2}) * (\frac{n2}{2})$
- Effort to implement: $L = \frac{1}{D}$
- Time to implement: $E = V * D$
- Number of delivered bugs: $T = \frac{E}{18 \cdot 2}$

Wie LoC, aber
formatierungsu
nabhängig

$$n = n1 + n2$$

$$N = N1 + N2$$

$$\hat{N} = n1 * \log_2(n1) + n2 * \log_2(n2)$$

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

$$D = (\frac{N1}{2}) * (\frac{n2}{2})$$

$$L = \frac{1}{D}$$

$$E = V * D$$

$$T = \frac{E}{18 \cdot 2} \quad \text{In sec}$$

$$B = \frac{E^3}{3000} \text{ oder neu: } B = \frac{V}{3000}$$



Halstead Metriken

Laut Halstead besteht folgende Beziehung zwischen Aufwand E und Code Volumen V

E proportional zu V^2



Einfaches Beispiel

```
public class Fibonacci {

    public static int compute(int n) {

        int result = 0;
        if(n <= 1) {
            result = n;
        } else {
            result = compute(n-1) + compute(n-2);
        }
        return result;
    }
}
```

Betrachte die Methode, Signatur nicht mitgezählt

Operatoren	Operanden
int = ; if () <= {} else – + return compute	result 0 n 1 2
n1 = 12, N1 = 20	n2 = 5, N2 = 12

Vocabulary = $n1 + n2 = 17$

Program Length = $N1 + N2 = 32$

Volume = $N * \log_2(n) = 131$

Difficulty Level = $N1/2 * n2/2 = 12,5$

Time to implement $(V * D)/18 \text{ s} = \text{ca } 91 \text{ s}$



Halstead Metriken

Zahlenbeispiel (Quelle

http://www.verifysoft.com/de_cmtpp_mscoder.pdf)

Listing 1. Beispielprogramm

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void extract(char* zeichenkette1, char* zeichenkette2, int
            anfang, int nb);

int eval(char * ch)
{
    int i;
    int wert1, wert2;
    int lgwert1;
    char *w1, *w2;
    char operation;
    int resultat;

    /* Suche des Operators und seiner Position */
    for( i=0 ; *(ch+i) != '+' as *(ch+i) != '-' as *(ch+i) != '*' as *(ch+i) != '/' as *(ch+i) != '^' ; i++)
    {
        ;
    }

    /* Fehlerbehandlung */
    if(i==0) /* der erste Operand fehlt */
    {
        printf("Error: kein <wert1>");
        exit(0);
    }
    else if(i==strlen(ch)-1) /* der zweite Operand fehlt */
    {
        printf("Error: kein <wert2>");
        exit(0);
    }
    else if(i==strlen(ch)) /* kein Operator vorhanden */
    {
        printf("Error: kein <operator>");
        exit(0);
    }

    /* Zeichenkette fuer den ersten Operanden */
    w1=(char*) malloc((i+1)*sizeof(char));
    extract(ch,w1,0,i);

    /* Umwandlung der Zeichenkette */
    sscanf(w1,"%d",&wert1);

    /* Erfassung des Operators */
    operation=(ch[i]);

    /* Zeichenkette fuer den zweiten Operanden */
    lgwert1=strlen(ch)-(i+1);
    w2=(char*) malloc((lgwert1+1)*sizeof(char));
    extract(ch,w2,i+1,lgwert1);

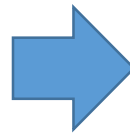
    /* Umwandlung der Zeichenkette */
    sscanf(w2,"%d",&wert2);

    /* Berechnung */
    switch(operation)
    {
        case '+':
            resultat=wert1+wert2;
            break;
        case '-':
            resultat=wert1-wert2;
            break;
        case '*':
            resultat=wert1*wert2;
            break;
        case '/':
            if(wert2 != 0)
            {
                resultat=wert1/wert2;
            }
            else
            {
                resultat=0;
                printf("Error: Division durch 0 nicht moeglich");
                exit(0);
            }
        ;
    }

    return resultat;
}

/* Funktion zum Extrahieren einer Unterzeichenkette
   von zeichenkette1 in zeichenkette2 */
void extract(char* zeichenkette1, char* zeichenkette2,
            int anfang, int nb)
{
    int i;
    zeichenkette1= zeichenkette1+anfang;
    i=0;
    while(i<nb)
    {
        * zeichenkette2=* zeichenkette1;
        zeichenkette1++;
        zeichenkette2++;
        i++;
    }
    * zeichenkette2='\0';
}

int main(int argc, char** argv)
{
    int res;
    if(argc!=2)
    {
        printf("Error, Nutzung des Programms : eval
            <expression>");
    }
    else
    {
        res=eval(argv[1]);
        printf("Das Ergebnis der Berechnung ist : %d",res);
    }
}
```



Programm Volume: 1590,423
Difficulty Level: 44,733
Effort to implement: 71144,908
Implementation time: 3952,495 (≈ 1h)

Die Zahlen incl Angabe bis zur dritten Nachkommastelle stammen aus dem oben zitierten Artikel.

Empfehlung:

- Volumen einer Funktion zwischen 20 und 1000
- Volumen eines Files zwischen 100 und 8000



Vorteile:

- Einfach zu ermitteln
- automatisiert auswertbar
- Für alle (textuellen) Programmiersprachen geeignet
- Empirisch bestätigt als gutes Maß für Komplexität

Kritik:

- Berücksichtigung ausschließlich lexikalischer Komplexität
- Konzepte wie Sichtbarkeit oder Namensräume nicht berücksichtigt
- Aufteilung Operator/Operand sprachabhängig

Verwendung

- zur Bewertung der Wartbarkeit eines Programmmoduls.
- Beurteilung der Entwicklung über die Zeit ein und desselben Programms.



- LoC/NCSS
- Halstead Metriken
- ➔ ■ McCabe Metrik
- Objektorientierte Metriken



McCabe Metrik

- McCabe Metrik = zyklomatische Komplexität
- Ziel: Komplexität einer Methode in einer Zahl auszudrücken.
- Basiert auf Graphentheorie.
- Auf alle strukturierten Sprachen anwendbar.
- Entspricht der Anzahl linear unabhängiger Pfade auf dem Kontrollflussgraphen eines Moduls.
- Entspricht der oberen Schranke für die mindestens nötige Anzahl von Testfällen für vollständige Zweigüberdeckung.

Definition aus der Graphentheorie

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

- e = Anzahl der Kanten
- n = Anzahl der Knoten
- p = unabhängige Teile des Graphs

In der weiteren Diskussion ist $p=1$ (d.h. der betrachtete Graph ist zusammenhängend).



McCabe Metrik

- Berechnet die strukturelle Komplexität.
- Wird i.A. pro Methode berechnet.
- Darstellung des Programms als Kontrollflussgraph.

Hier verwendete Definition (für zusammenhängende Graphen):

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

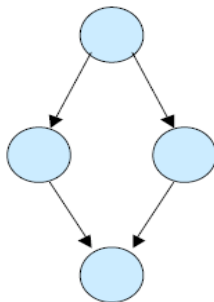
- e = Anzahl der Kanten
- n = Anzahl der Knoten

Sequenz



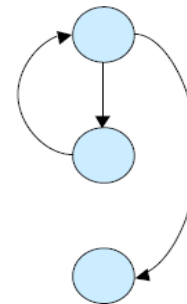
$$V(G) = 1 - 2 + 2 = 1$$

Auswahl



$$V(G) = 4 - 4 + 2 = 2$$

Abweisende Schleife



$$V(G) = 3 - 3 + 2 = 2$$



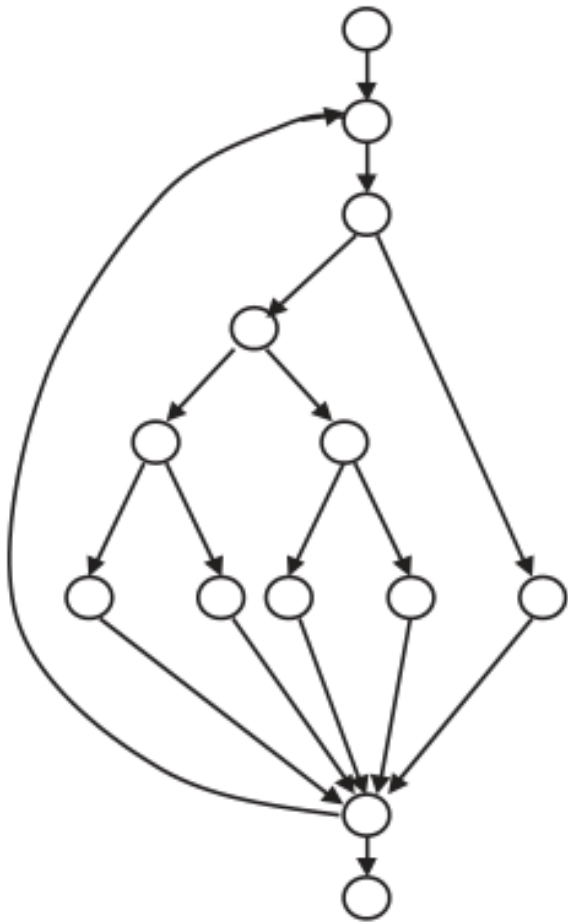
McCabe Metrik

McCabe Metrik in der vorgestellten Form nur für einzelne Kontrollflussgraphen anwendbar.

Bsp: Nebenstehender Graph hat die zyklomatische Komplexität von

$$V(G) = e - n + 2 = 17 - 13 + 2 = 6$$

Gilt als akzeptabler Wert.



McCabe Metrik

Nutzung für Abschätzungen in Bezug auf die Testbarkeit und die Wartbarkeit des Programmteils vorzunehmen.

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

$V(G)$	Risiko
1 – 10	Einfaches Programm, geringes Risiko
11 – 20	komplexeres Programm, erträgliches Risiko
21 – 50	komplexes Programm, hohes Risiko
> 50	untestbares Programm, extrem hohes Risiko

➔ Konsequenz:

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



McCabe Metrik

Pro

- Einfach zu berechnen (Parser genügt)
- Im Prinzip gute Korrelation zwischen zyklomatischer Zahl und Verständlichkeit der Komponente.
- Für Berechnung des Testaufwands geeignet.

Con

- Metrik berücksichtigt nur Kontrollfluss.
- Komplexität des Datenflusses wird nicht berücksichtigt.
- Teilweise nicht übereinstimmend: Metrik vs. subjektives Empfinden

Methodenbasiert (Normalfall)

- Es werden die Kontrollstrukturen innerhalb von Methoden untersucht.

Prozessbasiert

- Sämtliche für einen Prozess relevanten Methoden werden als unabhängige Teilgraphen betrachtet.
- e sind die Kanten der Kontrollstrukturgraphen aller Methoden.
- n sind die Knoten der Kontrollstrukturgraphen aller Methoden.
- p ist die Anzahl der beteiligten Methoden.
- Aussagen über Gesamtkomplexität von Prozessen



Sinnvoll: Verwendung gemeinsam mit anderen einfachen Komplexitätsmetriken

z.B.:

- Schachtelungstiefe
- Länge einer Methode
- ..



MCCabe – zum Weiterlesen

Original Publikation:

<http://www.literateprogramming.com/mccabe.pdf>

Kritik an McCabe Metrik:

<https://www.cqse.eu/en/blog/mccabe-cyclomatic-complexity/>



- LoC/NCSS
- Halstead Metriken
- McCabe Metrik
- ➔ ■ Objektorientierte Metriken



Objektorientierte Metriken

- Bisher: Metriken der imperativen Programmierung
 - Auch anwendbar für objektorientierte Programme
 - Aber
 - Rein imperativ ausgerichtete Metriken erfassen die Komplexität objekt-orientierter Programme nur unvollständig.
 - Klassische Metriken berücksichtigen objektorientierte Besonderheiten nicht (z.B. Vererbung).
- ➔ objektorientierte Metriken



Komponentenmetriken

Bewertung von einzelnen Komponenten (Klasse, Paket, Bibliothek)

Strukturmetriken

Bewertung des Klassenverbunds als Ganzes.
Beurteilung des Zusammenspiels von Komponenten.



Komponentenmetriken - Übersicht

Abkürzung	Metrik	Typ
OV	Object Variables	Umfangsmetrik
CV	Class Variables	Umfangsmetrik
NOA	Number of Attributes	Umfangsmetrik
WAC	Weighted attributes per class	Umfangsmetrik
WMC	Weighted Methods per class	Umfangsmetrik
DOI	Depth of Inheritance	Vererbungsmetrik
NOD	Number of Descendants	Vererbungsmetrik
NORM	Number of redefined Methods	Vererbungsmetrik
LCOM	Lack of Cohesion in methods	Kohäsionsmetrik



OV: Anzahl der Objektvariablen

CV: Anzahl der Klassenvariablen

NOA: Summe der beiden obigen

Umfangsmetriken, die

- Hinweise auf funktional bedeutsame Klassen geben.
- Hinweise auf potentielle Monolithen geben.

Weighted Methods per class (WMC)

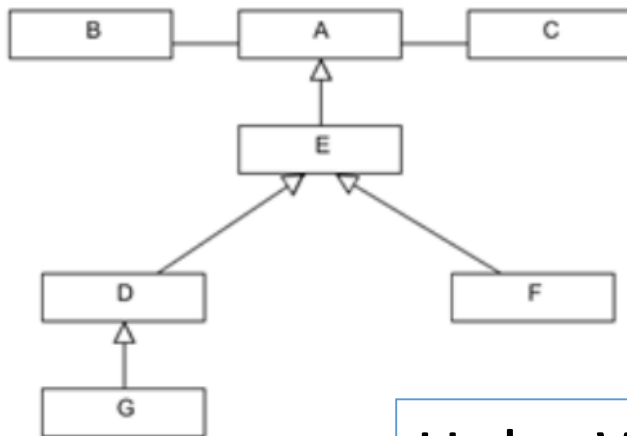
- Summe über die Methoden, Gewichtung c_i ist die zyklomatische Komplexität der einzelnen Methode

$$WMC = \sum_{i=0}^n c_i$$

Ein großer WMC Wert deutet auf eine schwer wartbare und schwer wiederverwendbare Klasse hin.

Depth of Inheritance Tree (DIT)

- Größter Abstand von der Wurzel des Vererbungsbaums bis zur betrachteten Klasse

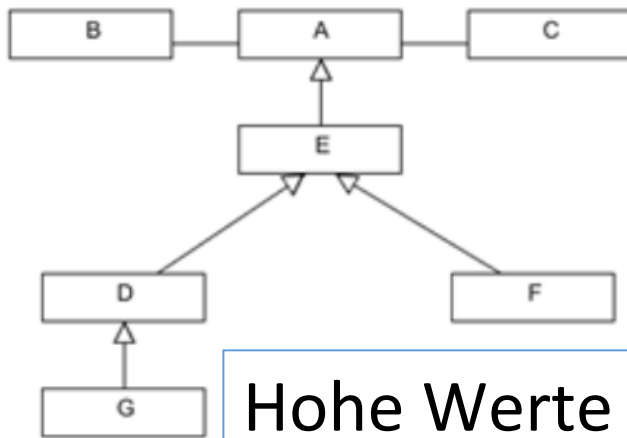


- $DIT(A) = 0$
- $DIT(G) = 3$

Hohe Werte für DIT → schwer verständlich, schwer wiederverwendbar

Number of Descendants (NOD)

Zahl der Kinder, die von der betrachteten Klasse direkt erben.



■ $\text{NOD}(A) = 1$

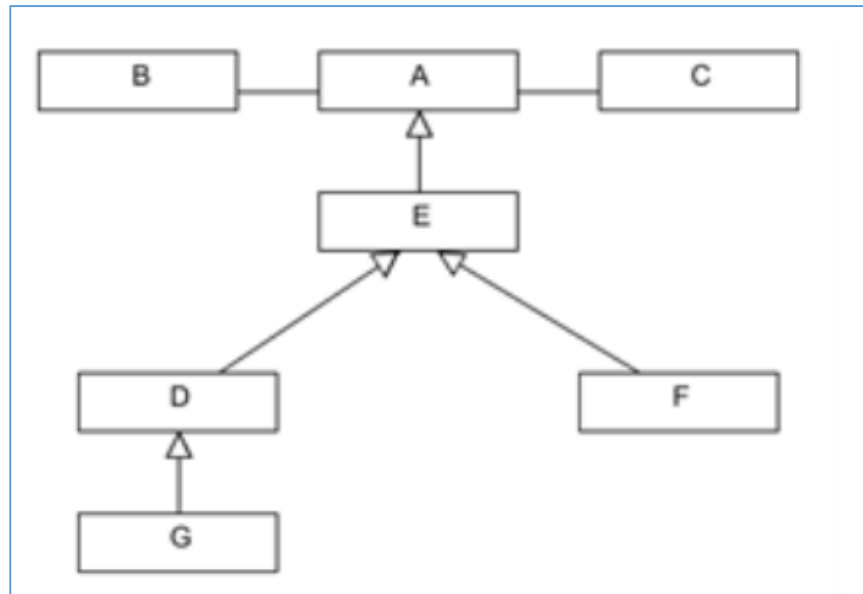
■ $\text{NOD}(E) = 2$

Hohe Werte für NOD → potentiell hohe Wiederverwendung

→ Potentiell falsche Verwendung der Abstraktion

CBO – Coupling between Object Classes

Der CBO-Wert einer Klasse C gibt die Anzahl der an diese gekoppelten Klassen wieder. Die gekoppelten Klassen sind diejenigen, die die Methoden oder Objektvariablen der Klasse C nutzen oder eine Instanz der Klasse C auf eine sonstige Weise nutzen. In dem Klassendiagramm ist $CBO(A) = 2$, da A von B und C verwendet wird.



LCOM (Lack of Cohesion in Methods):

Anzahl der Methoden Paare, die über disjunkte Variablensätze verfügen abzüglich Anzahl der Methodenpaare, die gemeinsame Variablen verwenden. (definitionsgemäß ≥ 0)

Niedriger Wert \rightarrow Hohe Kohäsion

Hohe Werte für LCOM \rightarrow schlechte Kohäsion

Empirisch ist der Nutzen der Metrik nicht gut bestätigt.

LCOM Beispiel

```
package demo;

public class Example {
    int inta;
    int intb;

    int firstMethod(){
        return inta;
    }

    int secondMethod(){
        return intb;
    }

    int thirdMethod(){
        return intb * intb;
    }

    int fourthMethod(){
        return intb*intb*intb;
    }
}
```

Folgende Paare haben ein gemeinsames Attribut:

- secondMethod, thirdMethod
- secondMethod, fourthMethod
- thirdMethod, fourthMethod

Folgende Paare haben kein gemeinsames Attribut:

- firstMethod, secondMethod
- firstMethod, thirdMethod
- firstMethod, fourthMethod

$$\rightarrow \text{LCOM} = 3 - 3 = 0$$



Im Package:

- Kohäsionsmetrik für Klasse C: Anzahl der von C abhängigen Klassen geteilt durch Klassen ohne Abhängigkeit von C.
- Abstraktionsniveau: Anzahl abstrakte Klassen und Interfaces geteilt durch Anzahl aller Klassen.



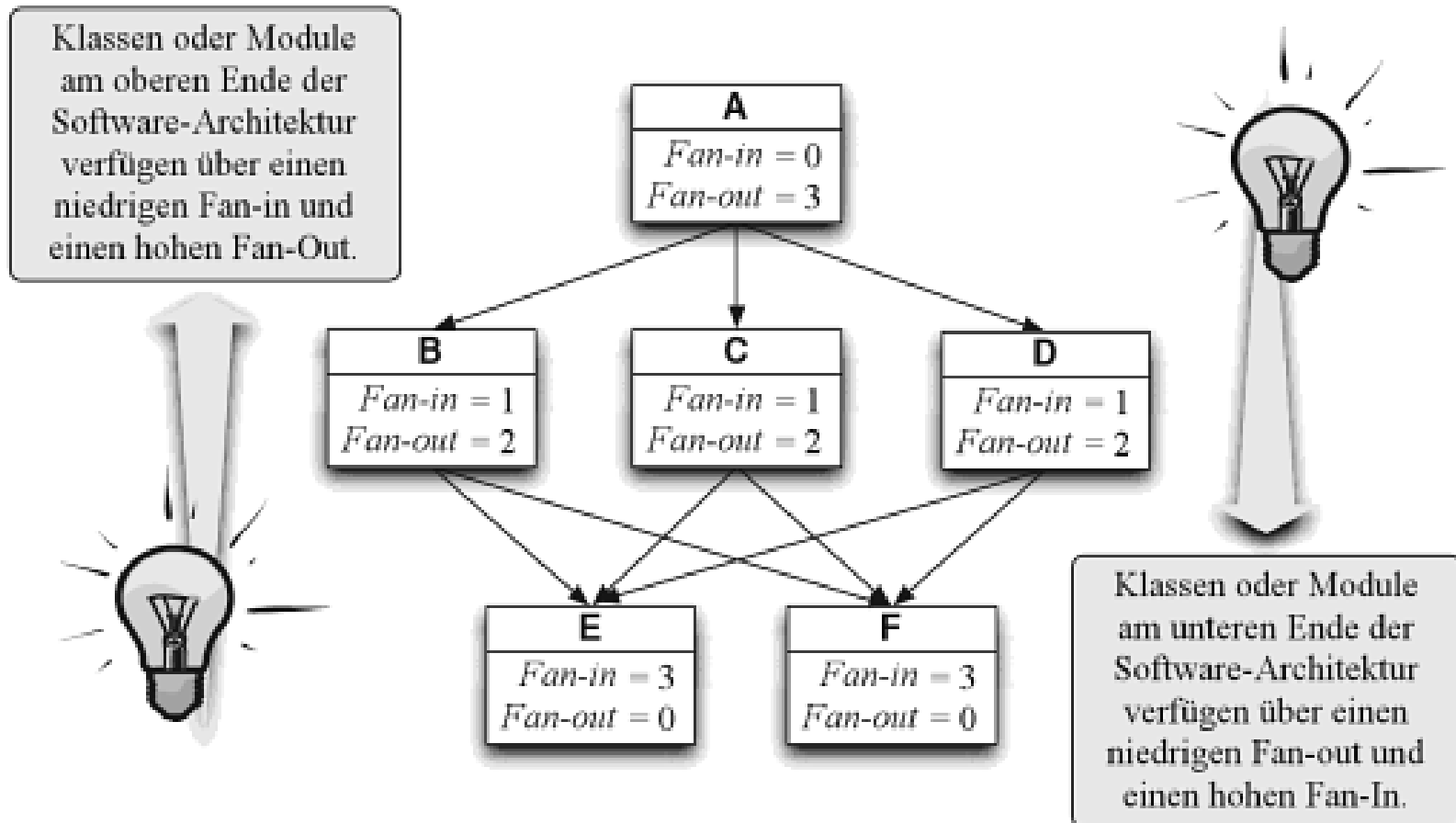
Strukturmetriken analysieren den Klassenverbund als Ganzes.

Wichtige Begriffe

- **Fan-In** eines Moduls :
Anzahl der Module, die auf dieses Modul zugreifen.
- **Fan-out**: Anzahl der Module, auf die dieses Modul zugreift.



Fan-in / Fan-out



Quelle des Bilds: D. Hoffmann, Software-Qualität, Springer, 2013



- **Henry und Kafura:**

Komplexität eines Moduls:

$$C_M = (\text{Fan-in} * \text{Fan-out})^2$$

- **Henry und Selig:**

$$C_M = \text{interne Komplexität} * (\text{Fan-in} * \text{Fan-out})^2$$

Bestimmung z.B. mit einer Halstead Metrik



Strukturmetriken

- Fan-in und Fan-out der Klassen eines Systems werden verwendet, um in komplexen Formeln ein Maß für die Komplexität des Systems zu gewinnen.
- Bsp: Card und Glass definieren

$$v_{CG} = s_{CG} + d_{CG}$$

Strukturkomplexität

$$s_{CG} = \sum_{i=1}^n F_{out}(C_i)^2$$

Datenkomplexität

$$d_{CG} = \sum_{i=1}^n \frac{IO(C_i)}{F_{out}(C_i) + 1}$$

IO: Anzahl der Input und Outputparameter



Bsp: Metriken aus JDepend

Quelle: Dokumentation von JDepend (<https://github.com/clarkware/jdepend>)

- **Number of Classes and Interfaces** The number of concrete and abstract classes (and interfaces) in the package is an indicator of the extensibility of the package.
- **Afferent Couplings (Ca)** The number of other packages that depend upon classes within the package is an indicator of the package's responsibility.
- **Efferent Couplings (Ce)** The number of other packages that the classes in the package depend upon is an indicator of the package's independence.
- **Abstractness (A)** The ratio of the number of abstract classes (and interfaces) in the analyzed package to the total number of classes in the analyzed package.
The range for this metric is 0 to 1, with $A=0$ indicating a completely concrete package and $A=1$ indicating a completely abstract package.



Bsp: Metriken aus Jdepend

- **Instability (I)** The ratio of efferent coupling (C_e) to total coupling ($C_e + C_a$) such that $I = C_e / (C_e + C_a)$. This metric is an indicator of the package's resilience to change. The range for this metric is 0 to 1, with $I=0$ indicating a completely stable package and $I=1$ indicating a completely instable package.
- **Distance from the Main Sequence (D)** The perpendicular distance of a package from the idealized line $A + I = 1$. This metric is an indicator of the package's balance between abstractness and stability. A package squarely on the main sequence is optimally balanced with respect to its abstractness and stability. Ideal packages are either completely abstract and stable ($x=0, y=1$) or completely concrete and instable ($x=1, y=0$). The range for this metric is 0 to 1, with $D=0$ indicating a package that is coincident with the main sequence and $D=1$ indicating a package that is as far from the main sequence as possible.



Bsp: Metriken aus Jdepend

- **Package Dependency Cycles**

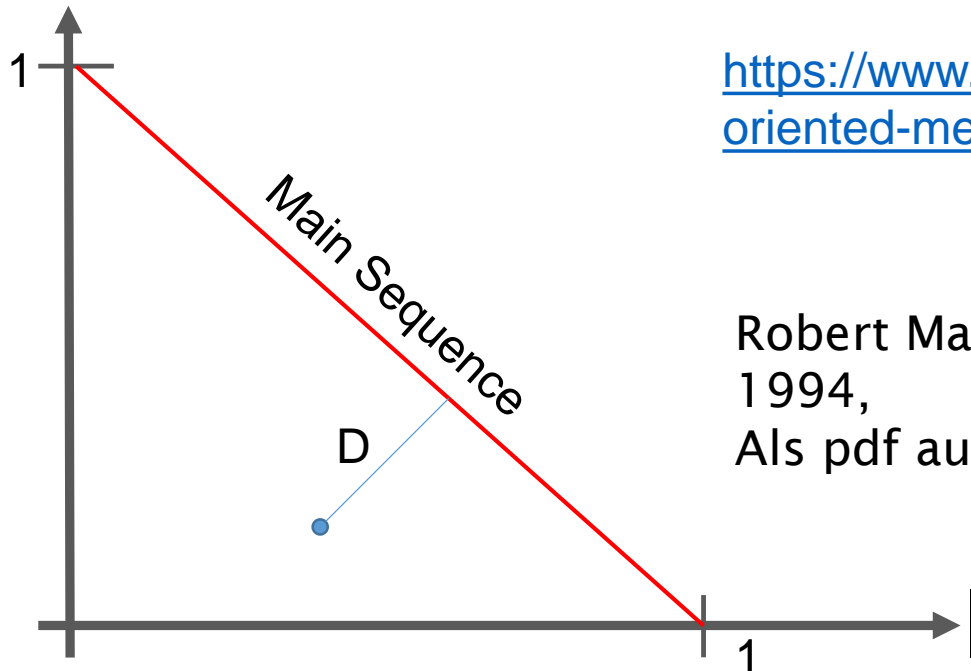
Package dependency cycles are reported along with the hierarchical paths of packages participating in package dependency cycles.

Quelle: Dokumentation von Jdepend (<https://github.com/clarkware/jdepend>)



Diskussion

- **Instability (I)**
- **Abstractness (A)**
- **Distance from the Main Sequence (D)**

A

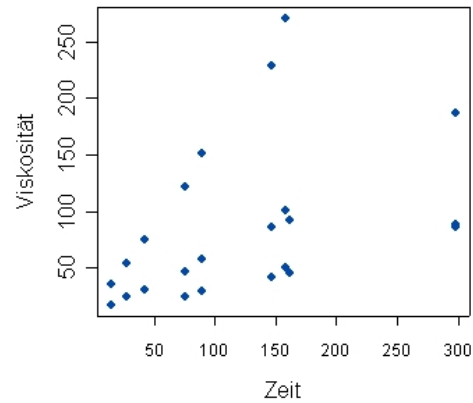
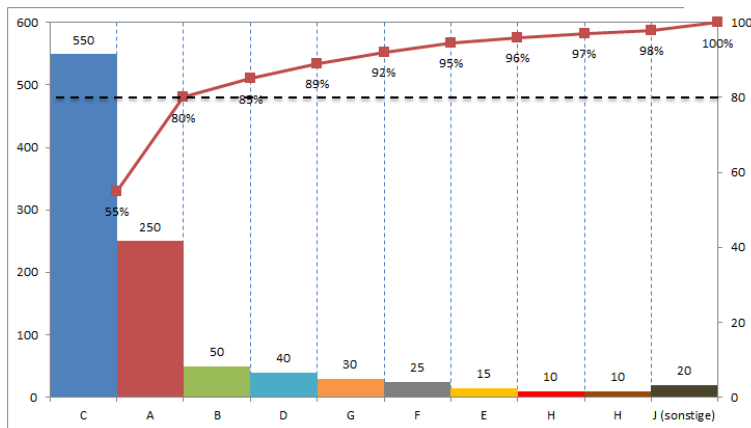
<https://www.future-processing.pl/blog/object-oriented-metrics-by-robert-martin/>

Robert Martin: OO Design Quality Metrics,
1994,
Als pdf auf Moodle



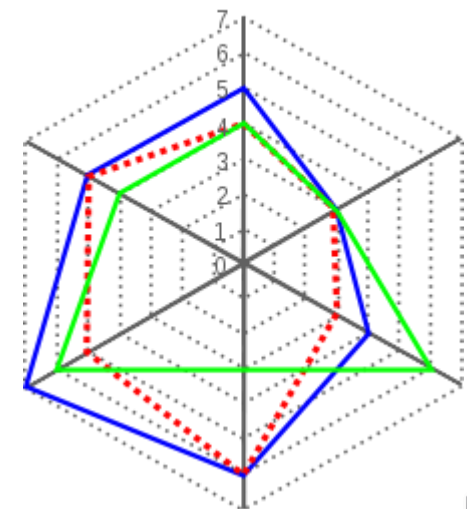
Visualisierung von Messwerten

Pareto Diagramm



Streudiagramm

Netzdiagramm



Statische Code Analyse

- Manuelle Prüfung
- Software Metriken
- ➔ ■ Konformitätsanalyse
- Exploit Analyse
- Anomalienanalyse



Syntax Analyse

→ in den Compiler integriert,
Details werden hier nicht besprochen (→
VL Compilerbau).

Semantik Analyse

Ziel: Fragwürdige und fehleranfällige,
aber syntaktisch korrekte Code Stellen
zu identifizieren.

- kein lauffähiger Code nötig
- Hohe Praxisbedeutung



■ GNU C-Compiler

```
int main(int argc, char *argv[])  
{  
    if(argc = 1){  
        /*erster Zweig*/  
    }else{  
        /*zweiter Zweig*/  
    }  
    return 0;  
}
```

```
gcc -Wall comparison.c
```

```
comparison.c: In function 'main':  
comparison.c:3:5: warning: suggest parentheses around assignment used as truth value [-Wparentheses]  
    if(argc = 1){  
      ^
```

■ GNU C-Compiler

```
#include <stdio.h>

int main()
{
    long value=42L;
    printf("%d", value);
    return 0;
}
```

```
D:\OTH\Vorlesungen\MST-SS2015\C-Beispiele>gcc -Wall printf.c
```

```
printf.c: In function 'main':
```

```
printf.c:65: warning: format '%d' expects argument of type 'int', but argument 2 has type 'long int' [-Wformat=]
    printf("%d", value);
    ^
```

Beispiele der semantischen Analyse 3

```
#include <stdio.h>

int main()
{
    int i=0;
    int p[4]={1,2,3,4};
    int q[4]={5,6,7,8};

    while(i<4)
    {
        p[i++] = q[i];
    }

    return 0;
}
```

```
D:\OTH\Vorlesungen\MST-SS2015\C-Beispiele>gcc -Wall arrayCopy.c
arrayCopy.c: In function 'main':
arrayCopy.c:11:12: warning: operation on 'i' may be undefined [-Wsequence-point]
    p[i++] = q[i];
      ^
arrayCopy.c:6:9: warning: variable 'p' set but not used [-Wunused-but-set-variable]
    int p[4]={1,2,3,4};
      ^
```

- **C:**

- Bsp: Splint (Secure Programming Lint)

- <http://www.splint.org/>

- **C/C++:**

- CPPCheck <http://cppcheck.sourceforge.net/>

- **Java: Bsp:**

- <http://checkstyle.sourceforge.net/> → analysiert Einhaltung von Coding Conventions



Aus dem Splint manual:

Problems detected by Splint include:

- Dereferencing a possibly null pointer (Section 2);
- Using possibly undefined storage or returning storage that is not properly defined (Section 3);
- Type mismatches, with greater precision and flexibility than provided by C compilers (Section 4.1–4.2);
- Violations of information hiding (Section 4.3);
- Memory management errors including uses of dangling references and memory leaks (Section 5);
- Dangerous aliasing (Section 6);
- Modifications and global variable uses that are inconsistent with specified interfaces (Section 7);
- Problematic control flow such as likely infinite loops (Section 8.3.1), fall through cases or incomplete switches (Section 8.3.2), and suspicious statements (Section 8.4);
- Buffer overflow vulnerabilities (Section 9);
- Dangerous macro implementations or invocations (Section 11); and
- Violations of customized naming conventions. (Section 12).



Splint

Aus dem Splint manual:

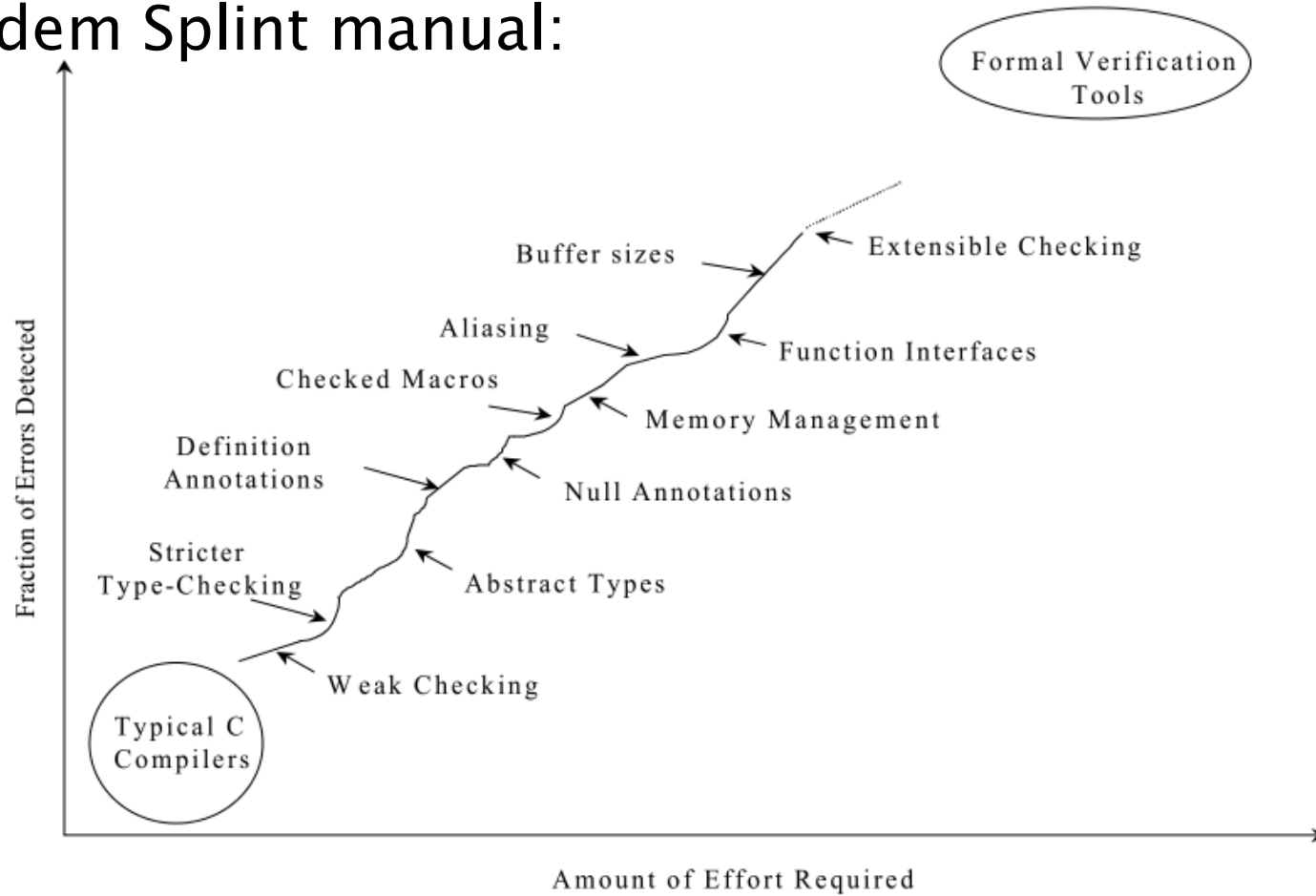


Figure 1. Typical Effort-Benefit Curve



Splint Bsp

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {

    size_t i;
    char *s = "Hello world\n";
    for (i=strlen(s); i>=0;i--) {
        printf("%c\n", s[i]);
    }
    return 0;
}
```

```
charprint.c(8,23): Comparison of unsigned value involving zero: i >= 0
  An unsigned value is used in a comparison with zero in a way that is either a
  bug or confusing. (Use -unsignedcompare to inhibit warning)
```

Splint Bsp

```
#include <stdio.h>
#include <string.h>
int *allocate();

int main(int argc, char *argv[]){
    int wert;
    int *ptr = allocate();
    wert = *ptr;
    printf("Wert ist %i", wert);
    return 0;
}
```

```
int *allocate(){

    int ret = 15;
    int *ptr= &ret;
    return ptr;
}
```

```
false_allocate.c(18,12): Stack-allocated storage ptr reachable from return
                           value: ptr
A stack reference is pointed to by an external reference when the function
returns. The stack-allocated storage is destroyed after the call, leaving a
dangling reference. (Use -stackref to inhibit warning)
false_allocate.c(17,20): Storage ptr becomes stack-allocated storage
```

Hinweis zu false negatives

Oftmals melden die Tools Probleme, die aber keine sind, sogenannte False Negatives

Umgang damit:

1. Verstehen
2. Entweder durch einfache Änderungen im Programm vermeiden (normalerweise wird die Änderung vom Tool vorgeschlagen).
3. Oder: Die Meldung ausblenden (dafür bieten die Tools Möglichkeiten, z.B. durch geeignete Kennzeichnung im Code).



Statische Code Analyse

- Manuelle Prüfung
- Software Metriken
- Konformitätsanalyse
- ➔ ■ Exploit Analyse
- Anomalienanalyse



- Analyse von sicherheitstechnischen Programmschwachstellen
- Mittlerweile ähnlicher Stellenwert wie die Elimination klassischer SW Fehler.
- ➔ Verweis auf die Sec VL
- Hier nicht weiter behandelt



Statische Code Analyse

- Manuelle Prüfung
- Software Metriken
- Konformitätsanalyse
- Exploit Analyse
- ➔ ■ Anomalienanalyse



Suche nach ungewöhnlichen oder auffälligen Code Sequenzen

Unterscheidung

- **Kontrollflussanomalien**

- Unstimmigkeiten im Programmablauf
- schwer automatisiert zu finden

- **Datenflussanomalien**

- Anweisungssequenzen, die zweifelhafte Variablenzugriffe enthalten.



Anomalien – Bsp Kontrollfluss

Von Splint und manchem Compiler erkannt.

```
int sign(int x)
{
    if(x>=0)
    {
        return 1;
    }
    else
    {
        return -1;
    }
    return 0;
}
```

Weder vom Compiler noch von Splint erkannt.

```
int even(unsigned int x) {
    if(x%2 == 0) {
        return 1;
    }
    if(x%2 ==1) {
        return 0;
    }
    return 0;
}
```

Anomalien – Bsp Kontrollfluss

Kontrollflussgraph

```
int sign(int x)
{
    if (x >= 0)
    {
        return 1;
    }
    else
    {
        return -1;
    }
    return 0;
}
```

0

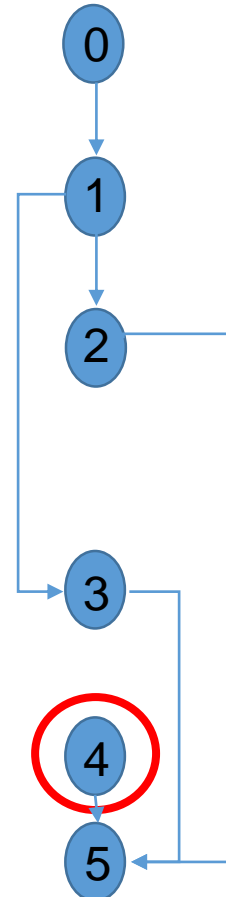
1

2

3

4

5



Anomalien – Bsp Datenfluss

```
#include <stdio.h>

int main(int argc, char *argv[]){
    int result=0;
    char *ptr;

    if(argc>1){
        ptr= (char *)malloc(1024);
        /*irgendwas*/
        result =1;
    }
    free(ptr);
}
```

Analyse mit der
Notation:

d(x): x wird definiert

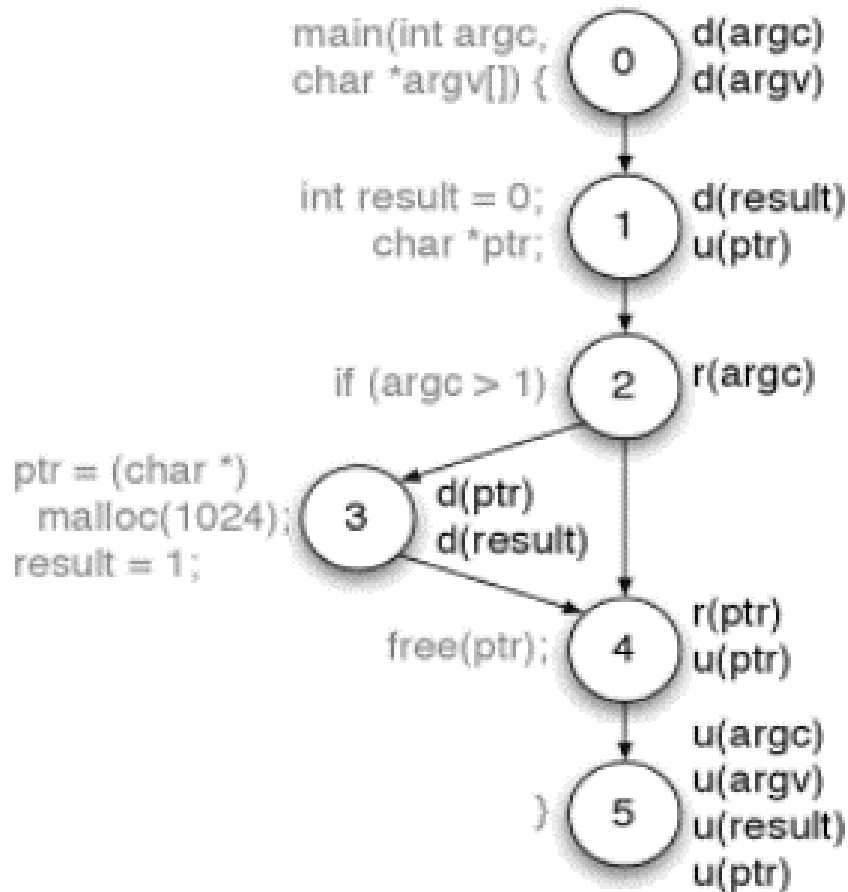
r(x): x wird referenziert

u(x): x ist undefiniert



Anomalien – Bsp Datenfluss

Datenflussgraph



Pfad 1: (0, 1, 2, 4, 5)

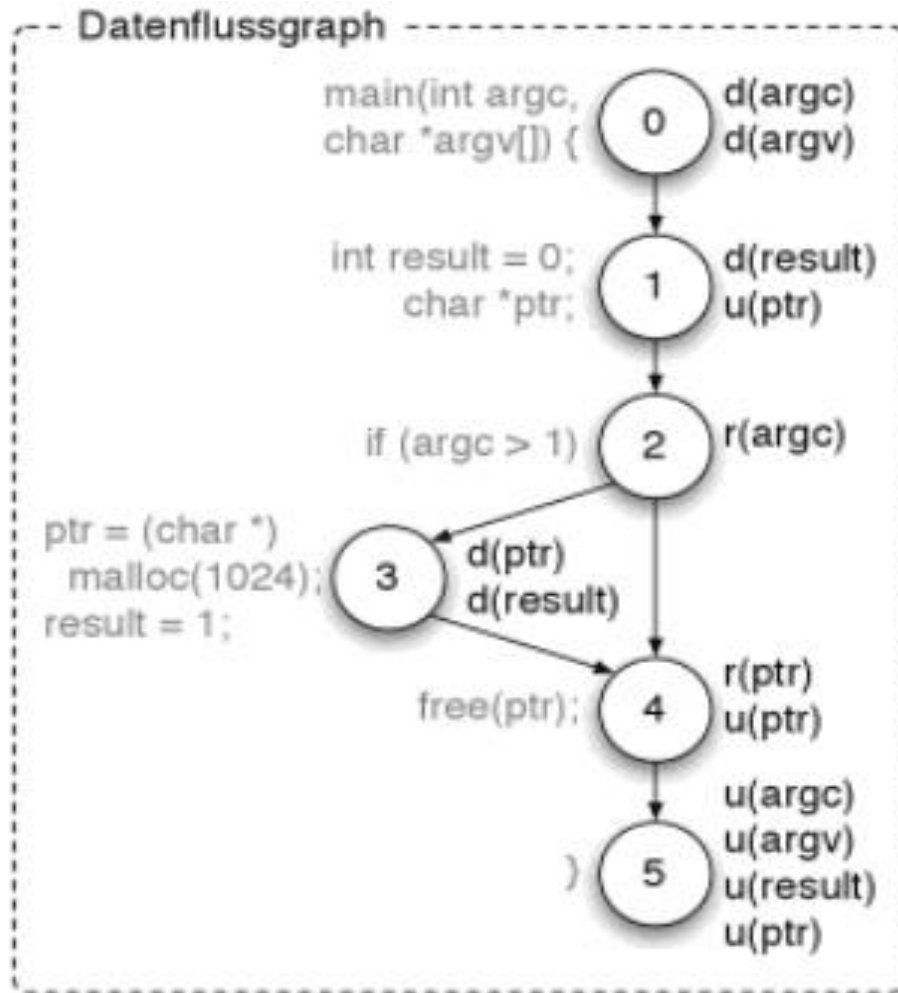
	0	1	2	4	5
argc	d		r		u
argv	d				u
result		d			u
ptr		u		ru	u

Pfad 2: (0, 1, 2, 3, 4, 5)

	0	1	2	3	4	5
argc	d		r			u
argv	d					u
result		d		d		u
ptr		u		d	ru	u

Muster	Beschreibung	Anomalie
dd	Variable wird zweimal hintereinander überschrieben	Ja
dr	Variable wird überschrieben und dann verwendet	nein
du	Variable wird geschrieben und dann gelöscht	Ja
rd	Variable wird gelesen und dann überschrieben	nein
rr	Variable wird zweimal hintereinander gelesen	nein
ru	Variable wird gelesen und dann gelöscht	nein
ud	Undefinierte Variable wird geschrieben	nein
ur	Undefinierte Variable wird verwendet	ja
uu	Undefinierte Variable wird gelöscht	nein

Anomalien – Bsp Datenfluss



■ Pfad 1: (0, 1, 2, 4, 5)

	0	1	2	4	5
argc	d		r		u
argv	d				u
result		d			u
ptr		u		ru	u

■ Pfad 2: (0, 1, 2, 3, 4, 5)

	0	1	2	3	4	5
argc	d		r			u
argv	d					u
result		d		d		u
ptr		u		d	ru	u

Anomalien im Bsp beseitigt

```
#include <stdio.h>
```

```
int main(int argc, char *argv[]){
```

```
    int result;
```

```
    char *ptr;
```

```
    if(argc>1){
```

```
        ptr= (char *)malloc(1024);
```

```
        /*irgendwas*/
```

```
        free(ptr)
```

```
        result =1;
```

```
    }else{
```

```
        result = 0;
```

```
    }
```

```
    return result;
```

```
}
```

free() im If Zweig →
Beseitigung der ur Anomalie

Zusätzlicher Else Zweig →
Beseitigung der dd Anomalie

Return Anweisung →
Beseitigung der du Anomalie

- Manuelle Prüfung
- Software Metriken
- Konformitätsanalyse
- Exploit Analyse
- Anomalienanalyse
- ➔ ■ Tools zur statischen Code Analyse



Tools zur statischen Code Analyse unterstützen bei:

- Einhaltung von Konventionen
- Erfassung von Metriken
- Lokalisierung von potentiellen Bugs im Code



Beispiele im Java Umfeld:

- **SonarCube**
 - <https://www.sonarqube.org/>
- **SpotBugs**
 - <https://spotbugs.github.io/index.html>
- **CheckStyle**
 - <http://checkstyle.sourceforge.net/>
- **JDepend**
 - <http://clarkware.com/software/JDepend.html>

