3. Statische Programmanalyse & Metriken

Statische Codeanalyse verlangt ein stures, monotones Anwenden von relativ einfachen Regeln auf oft umfangreichen Code. Diese Aufgabe erfordert keinerlei Kreativität aber eine sehr große Übersicht und Kontrolle. ... Statische Codeanalyse ist daher prädestiniert zur Automatisierung durch Werkzeuge. Ich empfehle Ihnen, diese Techniken entweder werkzeuggestützt oder gar nicht einzusetzen. [Li02]

Lernziele:

- verschiedene Arten der statischen Programmanalyse kennenlernen
- einige Werkzeuge zur statischen Programmanalyse einsetzen können
- Zusammenhänge zwischen Softwarequalität und Analyseverfahren verstehen
- strukturorientierte Analyse- und Messverfahren lernen

3.1 Einleitung

- Oft (fast immer) finden sich **80% aller Probleme** mit einem Softwaresystem in 20% des entwickelten Codes.
- Die statische Programmanalyse versucht meist werkzeuggestützt frühzeitig die **problematischen 20%** eines Softwaresystems zu finden.
- Statische Analyseverfahren identifizieren Programmteile von **fragwürdiger Qualität** und liefern damit Hinweise auf potentielle Fehlerstellen.
- Statische Analyseverfahren **versuchen** die Qualität von Software zu **messen**, können deshalb zur Festlegung von Qualitätsmaßstäben eingesetzt werden.
- Die statische Programmanalyse setzt **keine vollständig ausführbaren** Programme voraus.
- Die statische Programmanalyse kann also **frühzeitig** bei der Neuentwicklung und kontinuierlich bei der Wartung eines Softwaresystems eingesetzt werden.

Analytisches Qualitätsmanagement zur Fehleridentifikation:

- **analysierende Verfahren**:

  der „Prüfling" (Programm, Modell, Dokumentation) wird von Menschen oder Werkzeugen auf Vorhandensein/Abwesenheit von Eigenschaften untersucht

  - ⇨ **Review** (Inspektion, Walkthrough): Prüfung durch (Gruppe v.) Menschen
  - ⇨ **statische Analyse**: werkzeuggestützte Ermittlung von „Anomalien"
  - ⇨ **(formale) Verifikation**: werkzeuggestützter Beweis von Eigenschaften

- **testende Verfahren**:

  der „Prüfling" wird mit konkreten oder abstrakten Eingabewerten auf einem Rechner ausgeführt

  - ⇨ **dynamischer Test**: Prüfung des Testobjekts durch Ausführung auf einem Rechner (mit ganz konkreten Eingaben)
  - ⇨ **[symbolischer Test**: Ausführung mit symbolischen Eingaben (die oft unendliche Mengen möglicher konkreter Eingaben repräsentieren)]

Arten der Programmanalyse - 1:

- **Visualisierung von Programmstrukturen**: unästhetisches Layout liefert Hinweise auf sanierungsbedürftige Teilsysteme
  - ⇨ siehe Abschnitt 3.2 über Softwarearchitekturen und Visualisierung
- **manuelle Reviews**: organisiertes Durchlesen u. Diskutieren von Entwicklungsdokumenten durch Menschen
  - ⇨ siehe Abschnitt 3.3 hierzu
- **Compilerprüfungen**: Syntaxprüfungen, Typprüfungen, ...
  - ⇨ sollten hinreichend bekannt sein
- **Programmverifikation und symbolische Ausführung**: Beweis der Korrektheit eines Programms mit Logikkalkül oder anderen mathematischen Mitteln
  - ⇨ siehe etwa Vorlesungen zu „Verifikationstechnologien“
- **Stilanalysen**: Programmierkonventionen für Programmiersprachen
  - ⇨ Java-Programmierkonventionen im „Software-Praktikum“ und ...

Beispiele stilistischer Regeln für C++ - 1:

- vermeide komplexe Zuweisungen:

```
// anstelle von i *= j++ + 20;
```

- nicht zu viele „Laufvariablen“ in Schleifen:

```
for ( i = 0, j = 0, k = 10, l = -1 ; i < cnt;   i++, j++, k--, l += 2  ) {
  // do something
}
```

besser:

```
l = -1;
for ( i = 0, j=0, k=10; i < cnt; i++, j++, k-- ) {
  // do something
  l += 2;
}
```

Beispiele stilistischer Regeln für C++ - 2:

- Beachte übliche Namenskonventionen:
  - ⇨ Klassennamen beginnen immer mit Großbuchstaben
  - ⇨ alle anderen Namen mit Kleinbuchstaben
  - ⇨ Variablennamen enthalten nur Kleinbuchstaben und Ziffern
- keine komplexen Ausdrücke in Schleifenbedingungen:

```
for (int i = 0; i < vector.size(); i++) {
   // do something
}
```

Besser:

```
vectorsize = vector.size();
for (int i = 0; i < vectorsize; i++) {
   // do something
}
```

- ...

MISRA-Programmierrichtlinien für C / C++:

MISRA-C und **MISRA-C++** sind Programmierstandards der Automobilindustrie für sicherheitskritische Programme, die in C bzw. in C++ implementiert sind. Sie wurden von der MISRA (Motor Industry Software Reliability Association) erarbeitet und definieren „sichere“ Teilmengen der entsprechenden Programmiersprachen. Typische Regeln sind (http://www.misra.org.uk/):

- die Verwendung von Rekursion ist verboten
- Pointerarithmetik ist zu vermeiden
- Feldgrenzen und Division durch Null sind zu prüfen
- keine goto-Anweisungen
- kein Vergleich (mit == oder !=) von Float-Werten
- keine Zuweisungen in Bedingungen (von if-Anweisungen etc.)
- ...

Arten der statischen Programmanalyse - 2:

- **Kontroll- und Datenflussanalysen**: die Programmstruktur wird untersucht, um potentielle Zugriffe auf undefinierte Variablen, möglicherweise nie ausgeführten Code, etc. zu entdecken.

  ⇨ siehe Abschnitt 3.4

- **Metriken**: Programmeigenschaften werden gemessen und als Zahl repräsentiert - in der Hoffnung, dass kausaler Zusammenhang zwischen Softwarequalität (z.B. Fehlerzahl) und berechneter Maßzahl besteht.

  ⇨ siehe Abschnitt 3.5

Anmerkung:

In Abschnitt 3.5 mehr zur Bewertung von Metriken und Validierung von Hypothesen über Zusammenhang von Softwarequalität und Maßzahlen.

3.2 Softwarearchitekturen und -visualisierung

Große Systeme sind immer in Subsysteme gegliedert, von denen jedes eine Anzahl von Diensten bereitstellt. Der fundamentale Prozess zur Definition dieser Subsysteme und zur Errichtung eines Rahmenwerkes für die Steuerung und Kommunikation dieser Subsysteme wird ***Entwurf der Architektur*** *... genannt. [So07]*

Begriffe nach Sommerville:

- Ein **Softwaresystem** besteht aus Teilsystemen, die zusammengehörige Gruppen von Diensten anbieten und möglichst unabhängig voneinander realisiert sind.
- Ein **Teilsystem** kann wiederum aus Teilsystemen aufgebaut werden, die aus Moduln (Paketen) bestehen.
- Ein **Modul** (Paket) bietet über seine Schnittstelle Dienste an und benutzt (importiert) zu ihrer Realisierung Dienste anderer Module (Pakete).
- Ein Modul fasst „verwandte“ Prozeduren, **Klassen**, ... zusammen.

Programmarchitekturen in C++:

- C++ besitzt keine Sprachmittel zur Deklaration von Teilsystemen. Oft werden **Dateibäume** (Subdirectories) zur Teilsystembildung eingesetzt.
- Module (Pakete) werden in C++ durch cpp/cc-**Dateien** realisiert; h(eader)-Dateien definieren wie in C die Schnittstellen von Moduln.
- **include**-Beziehungen (textuelles Einkopieren) von h-Dateien realisieren Modulimporte (in Java gibt es stattdessen Paketimporte).

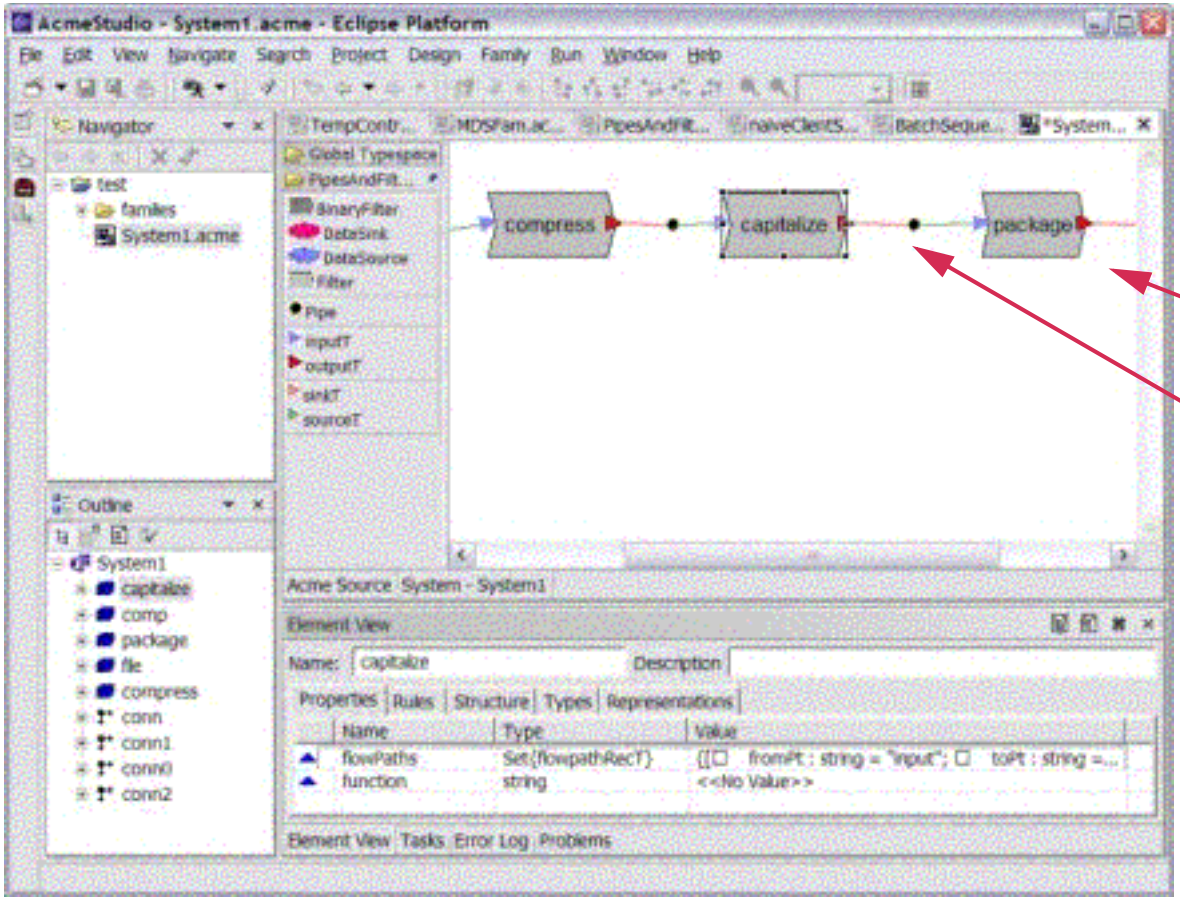
Zusätzliche Programmstrukturen in C++:

- Vererbungshierarchien auf Klassen
- sonstige Beziehungen zwischen Klassen
- Kontrollfluss eines Programms
- ...

Softwarearchitekturen sind mehr als Teilsysteme und Module:

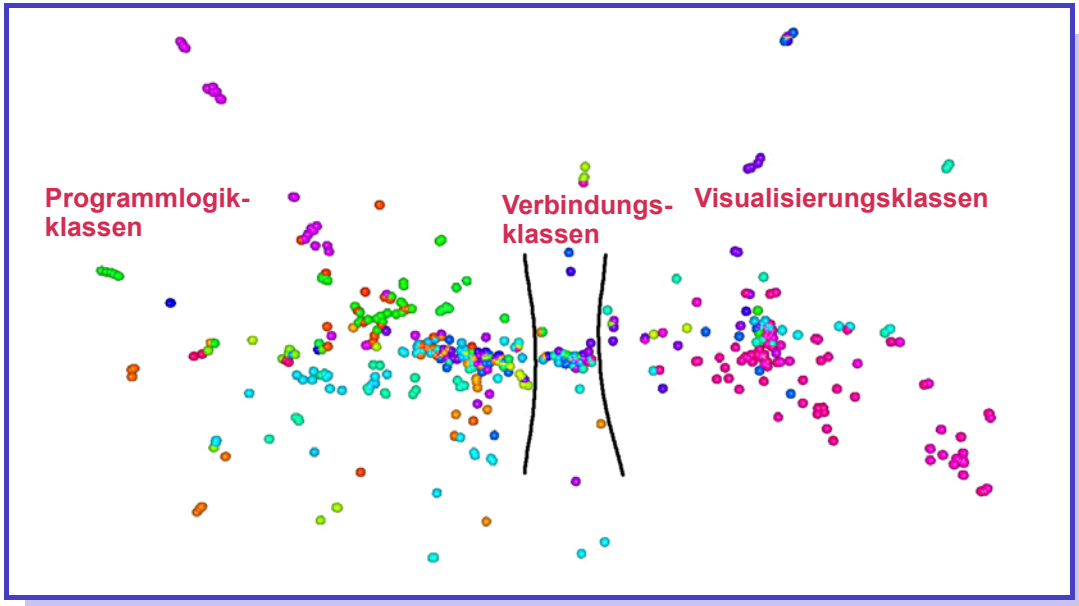
- In den 80er Jahren wurden Programmarchitekturen mit sogenannten „**M**odule **I**nterconnection **L**anguages (**MIL**)“ definiert, die nur Module und Import-Beziehungen kennen.
- Seit den 90er Jahren werden auch „**A**rchitecture **D**escription **L**anguages“ (**ADL**s) eingesetzt, die Komponenten mit Eingängen und Ausgängen und Verbindungen dazwischen verwenden (siehe Vorlesung „Echtzeitsysteme“); siehe auch [SG96]
- Heute verwendet man einen noch umfassenderen Architekturbegriff; in „Software Engineering I“ werden folgende **Architektursichten** im Zusammenhang mit der „**U**nified **M**odeling **L**anguage“ (**UML**) eingeführt:
  - ⇨ Teilsystem-Sicht (Paketdiagramme)
  - ⇨ Struktur-Sicht (Klassendiagramme, Kollaborationsdiagramme)
  - ⇨ Kontrollfluss-Sicht (Aktivitätsdiagramme, ...)
  - ⇨ Datenfluss-Sicht (Aktivitätsdiagramme, ...)
  - ⇨ ...

Beispiel für ADL - Filterketten in ACME:



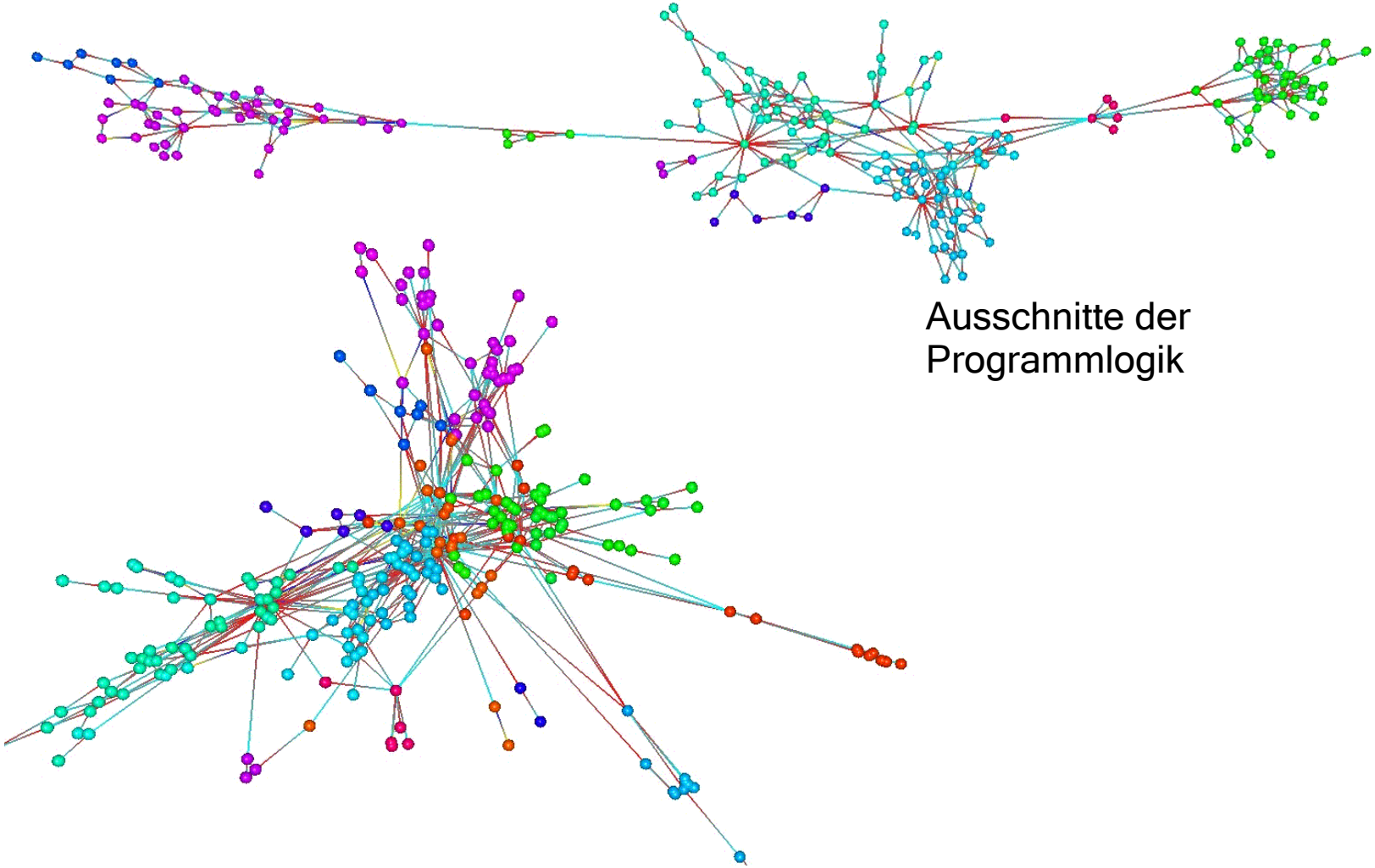
Acme-Studio-Werkzeug: http://www-2.cs.cmu.edu/~acme/AcmeStudio

Statische Visualisierung von Programmstruktur mit CrocoCosmos :



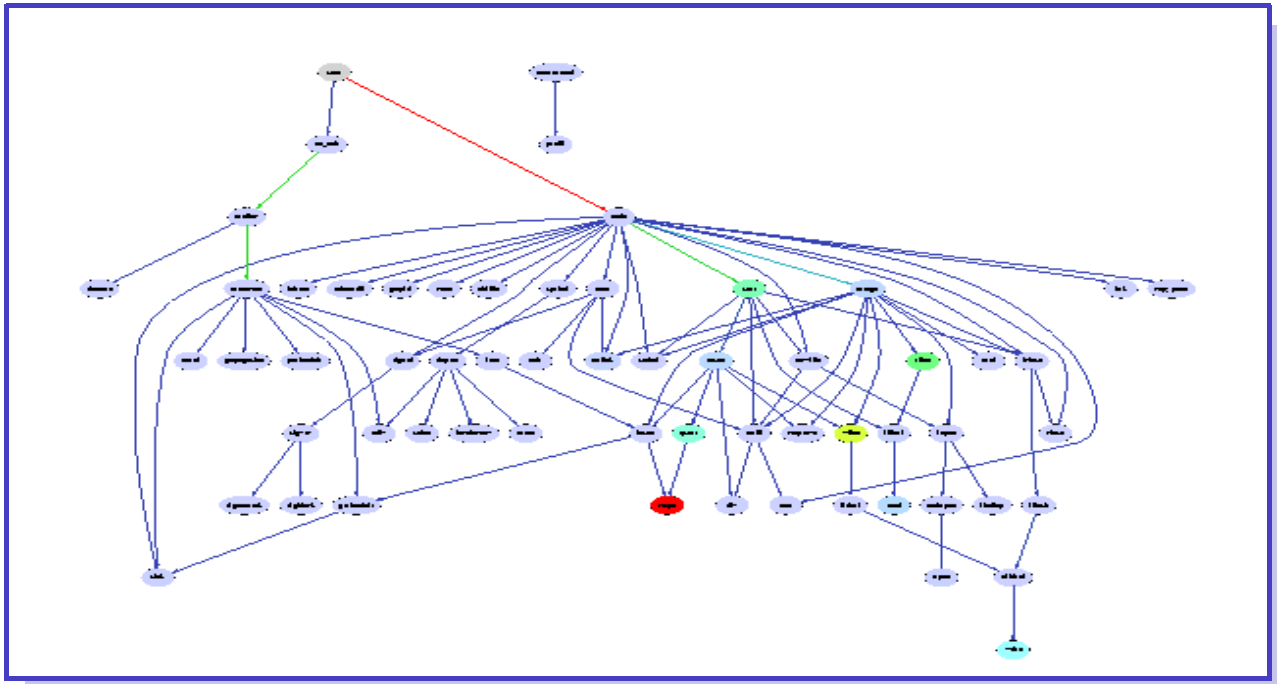
- grafische Darstellung eines Rahmenwerks für interaktive Anwendungen
- Klassen eines Teilsystems besitzen dieselbe Farbe, Beziehungen zwischen Klassen (aus Gründen der Übersichtlichkeit weggelasssen)
- siehe https://www.b-tu.de/fg-software-systemtechnik/forschung/projekte

Statische Visualisierung von Programmstrukturen - Fortsetzung:



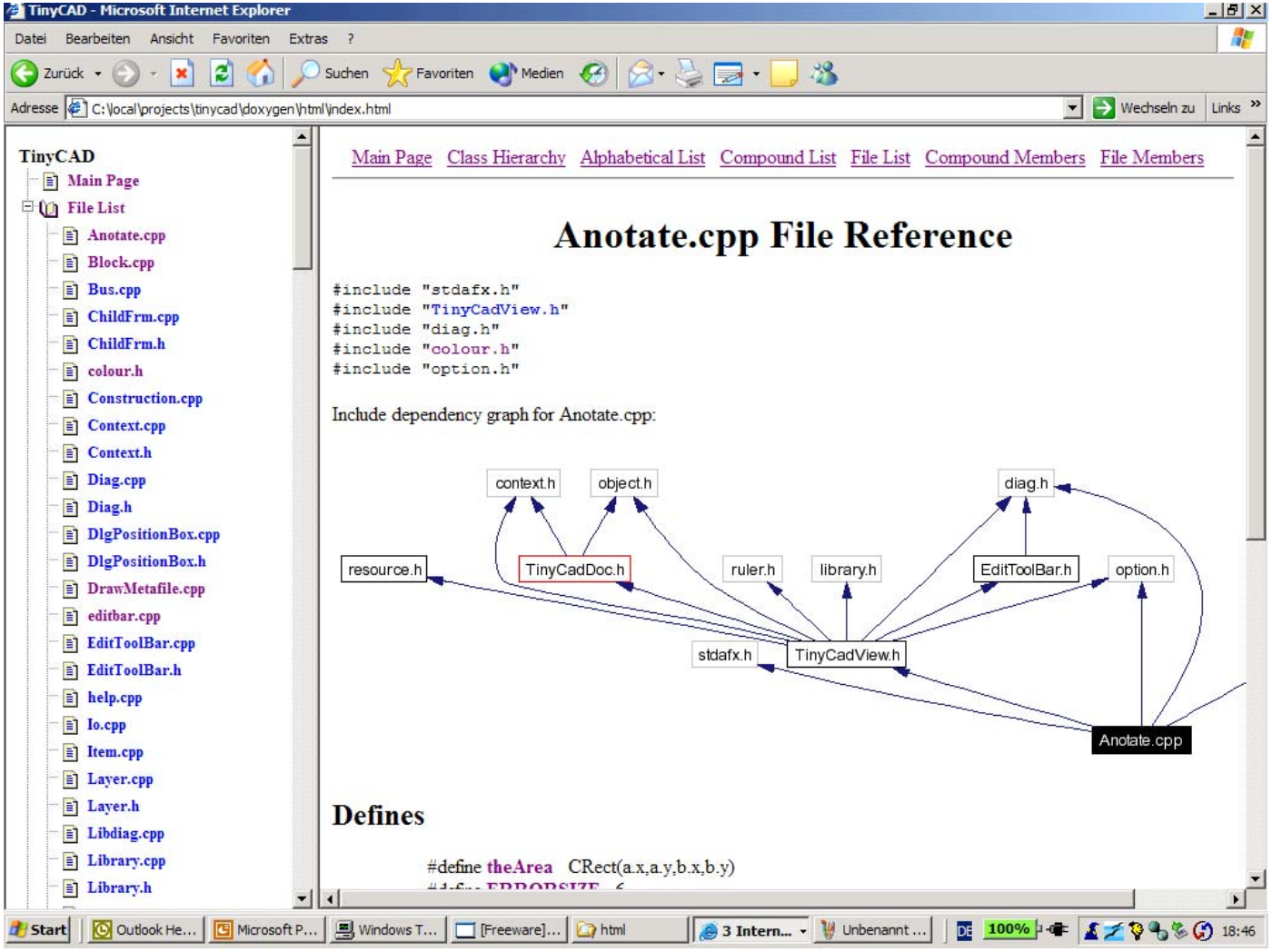
Ausschnitte der Programmlogik

Programmvisualisierung mit Doxygen und Graphviz:



- Doxygen generiert Programmdokumentation, Architekturdiagramme, ... ; siehe http://www.doxygen.nl/
- Graphviz (dot) ist Hilfsprogramm für Layoutberechnung; siehe http://www.graphviz.org/

Programmvisualisierung mit Doxygen - „include“-Abhängigkeiten:



Programmvisualisierung mit Doxygen - generierte Dokumentation:

3.3 Strukturierte Gruppenprüfungen (Reviews)

Systematische Verfahren zur gemeinsamen „Durchsicht“ von Dokumenten (wie z.B. erstellte UML-Modelle, implementierten Klassen, …):

- **Inspektion**: ein stark formalisiertes Verfahren, dessen Ziel die Identifizierung von Befunden in einem Arbeitsprodukt ist und das Statistiken (Messungen) zur Verbesserung des Reviewprozesses und des Softwareentwicklungsprozesses liefert
- **Technisches Review**: weniger formale manuelle Prüfmethode; weniger Aufwand als bei Inspektion, ähnlicher Nutzen
- **Informelles Review (Walkthrough)**: unstrukturiertere Vorgehensweise; Autor des Dokuments liest vor, Gutachter stellen spontan Fragen
- [**Pair Programming**: Programm wird von vornherein zu zweit erstellt]

Empirische Ergebnisse zu Programminspektion:

- Prüfaufwand liegt bei ca. 15 bis 20% des Erstellungsaufwandes
- 60 bis 70% der Fehler in einem Dokument können gefunden werden
- Nettonutzen: 20% Ersparnis bei Entwicklung, 30% bei Wartung

Psychologische Probleme bei Reviews:

- ☹ Entwickler sind in aller Regel von der Korrektheit der erzeugten Komponenten überzeugt (ihre Komponenten werden höchstens falsch benutzt)
- ☹ Komponententest wird als lästige Pflicht aufgefasst, die
  - ⇨ Folgearbeiten mit sich bringt (Fehlerbeseitigung)
  - ⇨ Glauben in die eigene Unfehlbarkeit erschüttert
- ☹ Entwickler will eigene Fehler (unbewusst) nicht finden und kann sie auch oft nicht finden (da ggf. sein Testcode von denselben falschen Annahmen ausgeht)
- ☹ Fehlersuche durch getrennte Testabteilung ist noch ärgerlicher (die sind zu doof zum Entwickeln und weisen mir permanent meine Fehlbarkeit nach)
- ☺ Inspektion und seine Varianten sind u.a. ein Versuch, diese psychologischen Probleme in den Griff zu bekommen
- ☺ Rolle des Moderators ist von entscheidender Bedeutung für konstruktiven Verlauf von Inspektionen

Vorgehensweise bei der Inspektion:

- **Inspektionsteam** besteht aus Moderator, Autor (passiv), Gutachter(n), Protokollführer und ggf. Vorleser (nicht dabei sind Vorgesetzte des Autors / Manager)
- **Gutachter** sind in aller Regel selbst (in anderen Projekten) Entwickler
- Inspektion **überprüft**, ob:
  - ⇨ Dokument Spezifikation erfüllt (Implementierung konsistent zu Modell)
  - ⇨ für Dokumenterstellung vorgeschriebene Standards eingehalten wurden
- Inspektion hat **nicht zum Ziel**:
  - ⇨ zu untersuchen, wie entdeckte Fehler behoben werden können
  - ⇨ Beurteilung der Fähigkeiten des Autors
  - ⇨ [lange Diskussion, ob ein entdeckter Fehler tatsächlich ein Fehler ist]
- **Inspektionsergebnis**:
  - ⇨ formalisiertes Inspektionsprotokoll mit Fehlerklassifizierung
  - ⇨ Fehlerstatistiken zur Verbesserung des Entwicklungsprozesses

Ablauf einer Inspektion:

- **Planung** des gesamten Verfahrens durch Management und Moderator
- **Auslösung** der Inspektion durch Autor eines Dokumentes (z.B. durch Freigabe)
- **Eingangsprüfung** durch Moderator (bei vielen offensichtlichen Fehlern wird das Dokument sofort zurückgewiesen)
- **Einführungssitzung**, bei der Prüfling den Gutachtern vorgestellt wird
- **Individualuntersuchung** des Prüflings (Ausschnitt) durch Gutachter (zur **Vorbereitung** auf gemeinsame Sitzung) anhand ausgeteilter Referenzdokumente
- auf **Inspektionssitzung** werden Prüfergebnisse mitgeteilt und protokolliert sowie Prüfling gemeinsam untersucht
- **Nachbereitung** der Sitzung und **Freigabe** des Prüflings durch Moderator (oder Rückgabe zur Überarbeitung)

Technisches Review (abgeschwächte Form der Inspektion):

- Prozessverbesserung und Erstellung von Statistiken steht nicht im Vordergrund
- Moderator gibt Prüfling nicht frei, sondern nur Empfehlung an Manager
- kein formaler Inspektionsplan mit wohldefinierten Inspektionsregeln
- Ggf. auch Diskussion alternativer Realisierungsansätze

Informelles Review (Walkthrough):

- Autor des Prüflings liest ihn vor (ablauforientiert im Falle von Software)
- Gutachter versuchen beim Vorlesen ohne weitere Vorbereitung Fehler zu finden
- Autor entscheidet selbst über weitere Vorgehensweise
- Zielsetzungen:
  - ⇨ Fehler/Probleme im Prüfling identifizieren
  - ⇨ Ausbildung/Einarbeitung von Mitarbeitern

3.4 Kontroll- und Datenflussorientierte Analysen

Der Kontrollflussgraph ist ... eine häufig verwendete Methode zur Darstellung von Programmen. ... Die Verarbeitungssteuerung übernehmen die Kontrollstrukturen der Software unter Nutzung der Datenwerte. Eingabedaten werden gelesen, um Zwischenergebnisse zu bestimmen, die in den Speicher geschrieben werden, ... Die Daten „fließen“ quasi durch die Software; von Eingaben zu Ausgaben.

[Li02]

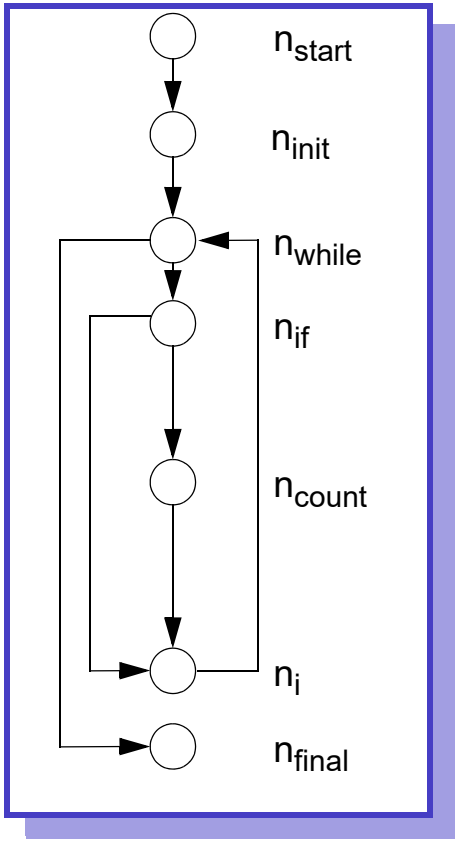
Definition „gerichteter Graph“:

Ein **gerichteter Graph** G ist ein Tupel (N, E) mit

- ⇨ N := Menge von **Knoten** (Nodes)
- ⇨ $E \subseteq N \times N$ einer Menge gerichteter **Kanten** (Edges); eine Kante $e = (v_1 , v_2) \in E$ wird Kante von v_1 nach v_2 genannt

Kontrollflussgraph eines Programms (Prozedur, Methode):

```
PROCEDURE countVowels(IN s: Sentence; OUT count: INTEGER);          (* start *)
   (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)

VAR i: INTEGER;

BEGIN

   count := 0; i := 0;(* init *)

   WHILE s[i] # '.' DO

      IF s[i]= 'a' OR s[i]= 'e' OR
         s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'
      THEN

         count := count+1;

      END;

      i := i+1;

   END (* WHILE *)
END countVowels;          (* final *)
```

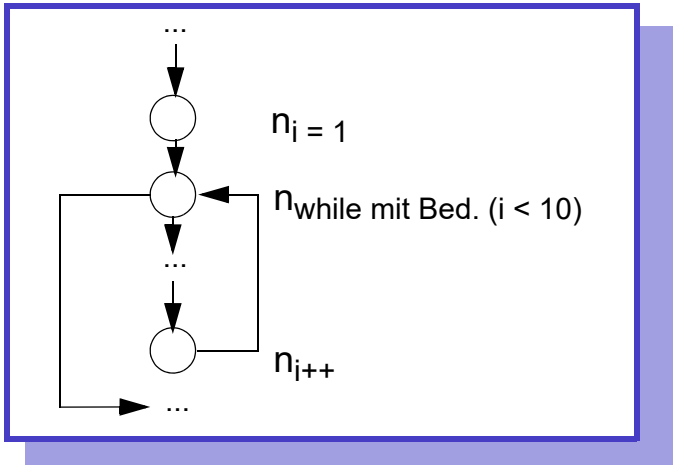


Kontrollflussgraph
mit
- 7 **Knoten**
- 8 **Kanten**

Behandlung von Schleifen in C, ... :

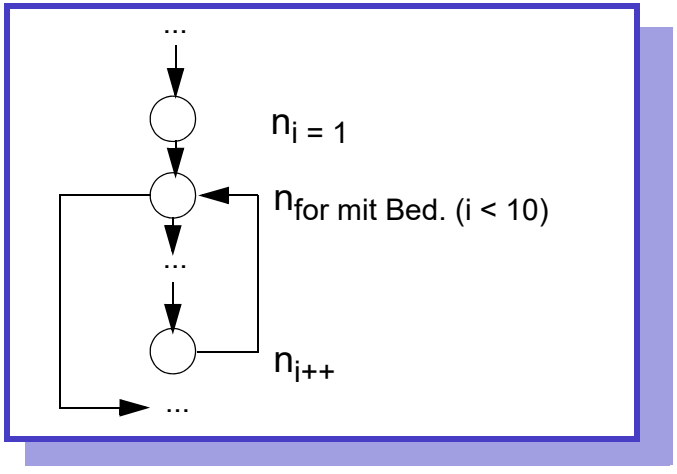
- while-Schleife:

```
...
int i = 1;
while(i < 10){
  ...
  i++;
}
...
```



- for-Schleife:

```
... ;
for(i = 1; i < 10; i++) {
  ...
}
```



Kontrollflussgraph - formale Definition:

Ein **Kontrollflussgraph** eines Programms (Prozedur) ist ein gerichteter Graph $G = (N, E, n_{start}, n_{final})$ mit

- ⇨ N ist die Menge der **Anweisungen** (Knoten = **N**odes) eines Programms
- ⇨ $E \subseteq N \times N$ ist die Menge der **Zweige** (Kanten = **E**dges) zwischen den Anweisungen des Programms
- ⇨ $n_{start} \in N$ ist der **Startknoten** des Programms
- ⇨ $n_{final} \in N$ ist der **Endknoten** des Programms (alternativ könnte man auch eine Menge von Endknoten betrachten)

Es gilt für den Kontrollflussgraphen:

- ⇨ $\neg\exists\, n \in N : (n, n_{start}) \in E$ - keine in den Startknoten einlaufenden Kanten
- ⇨ $\neg\exists\, n \in N : (n_{final}, n) \in E$ - keine aus dem Endknoten auslaufenden Kanten

Manchmal wird zusätzlich gefordert, dass Kontrollflussgraph zusammenhängend ist:

- ⇨ $\forall\, n \in N$: es gibt einen Pfad von n_{start} zu n
- ⇨ $\forall\, n \in N$: es gibt einen Pfad von n nach n_{final}

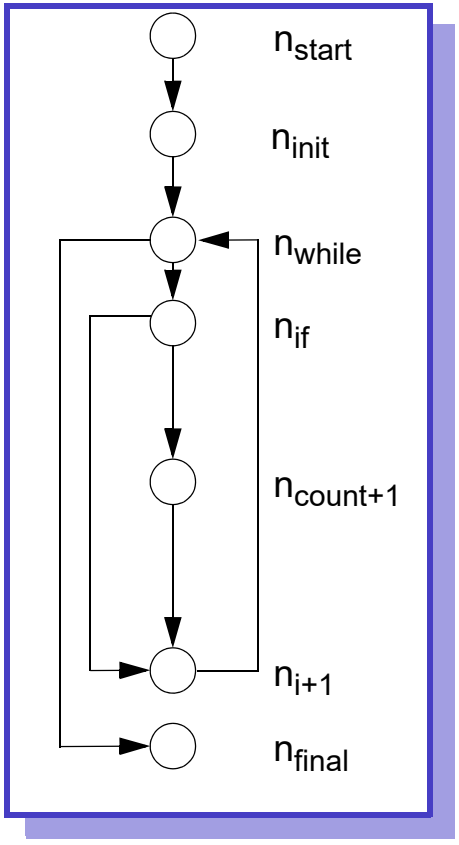
Pfade im Kontrollflussgraphen - formale Definition:

Ein **Pfad der Länge k** in einem Kontrollflussgraphen $G = (N, E, n_{start}, n_{final})$ ist eine Knotenfolge $n_1, \ldots n_k$, sodass gilt:

- ⇨ $n_1, \ldots, n_k \in N$
- ⇨ $\forall\, i \in 1, \ldots, k\text{-}1 : (n_i, n_{i+1}) \in E$

Ein **zyklenfreier Pfad** enthält keinen Knoten zweimal.

Ein **Zyklus** ist ein Pfad mit $n_1 = n_k$



Kontrollflussgraph
mit
- 7 **Knoten**
- 8 **Kanten**

Beispiel für einen Pfad:

n_{start} , n_{init} , n_{while} , n_{if} , n_{count} , n_{i+1} , n_{while} , …

Der Pfad ist nicht zyklenfrei.

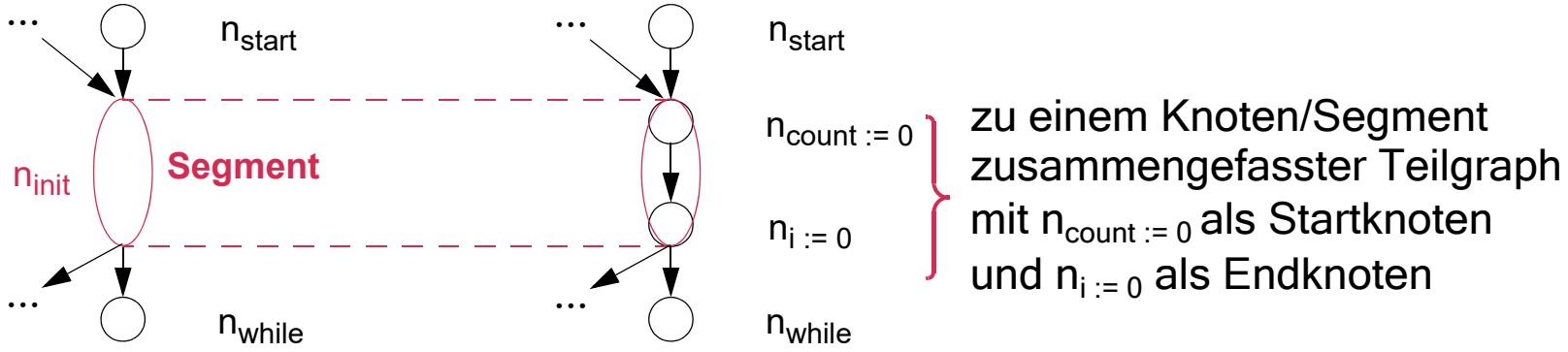
Kontrollflussgraphsegmente - formale Definition:

Ein **Segment** oder **Block** eines Kontrollflussgraphen $G = (N, E, n_{start}, n_{final})$ ist ein Knoten s, der einen Teilgraph $G' = (N', E', n'_{start}, n'_{final})$ von G ersetzt, der aus einem zyklenfreien Pfad $n'_{start} = n_1, \ldots, n_k = n'_{final}$ besteht mit $N' = \{n_1, \ldots, n_k\}$:

- $\Rightarrow$ $N' \subseteq N \wedge E' = E \cap (N' \times N') \wedge \neg (n'_{final}, n'_{start}) \in E'$
- $\Rightarrow$ $\forall n' \in N' \setminus \{n'_{final}\} \; \exists_1 n \in N : (n', n) \in E$ - genau eine auslaufende Kante
- $\Rightarrow$ $\forall n' \in N' \setminus \{n'_{start}\} \; \exists_1 n \in N : (n, n') \in E$ - genau eine einlaufende Kante

Einlaufende Kanten von n'_{start} und auslaufende von n'_{final} werden auf s umgelenkt.

Beispiel für ein Segment (kompakte Notation von Graphen):



Datenflussattribute eines Kontrollflussgraphen:

Die Anweisungen eines Kontrollflussgraphen $G = (N, E, n_{start}, n_{final})$ besitzen Datenflussattribute, die den Zugriffen auf Variable (Parameter, ...) in den Anweisungen entsprechen:

- ⇨ $n \in N$ besitzt das Attribut **def(v)** oder kurz **d(v)**, falls
  n eine Zuweisung an v enthält (Wert von v definiert);
  gilt auch für Eingabeparameter bei n_{start}
- ⇨ $n \in N$ besitzt das Attribut **c-use(v)** oder kurz **c(v)**, falls
  n eine Berechnung mit Zugriff auf v enthält (c = compute);
  implizite Zuweisung an Ausgabeparameter am Ende ist auch c-use
- ⇨ $n \in N$ besitzt das Attribut **p-use(v)** oder kurz **p(v)**, falls
  n eine Fallentscheidung mit Zugriff auf v enthält (p = predicative)
- ⇨ $n \in N$ besitzt das Attribut **r(v)**, falls es das Attribut **c(v)** oder **p(v)** besitzt, also lesend auf v zugreift (r = reference); dient nur der Zusammenfassung von c(v) und p(v), wenn Unterschied c/p irrelevant ist
- ⇨ $n \in N$ besitzt das Attribut **u(v)**, falls v in dieser Anweisung (noch) keinen definierten Wert (mehr) besitzen kann

Kontrollflussgraph mit Datenflussattributen:

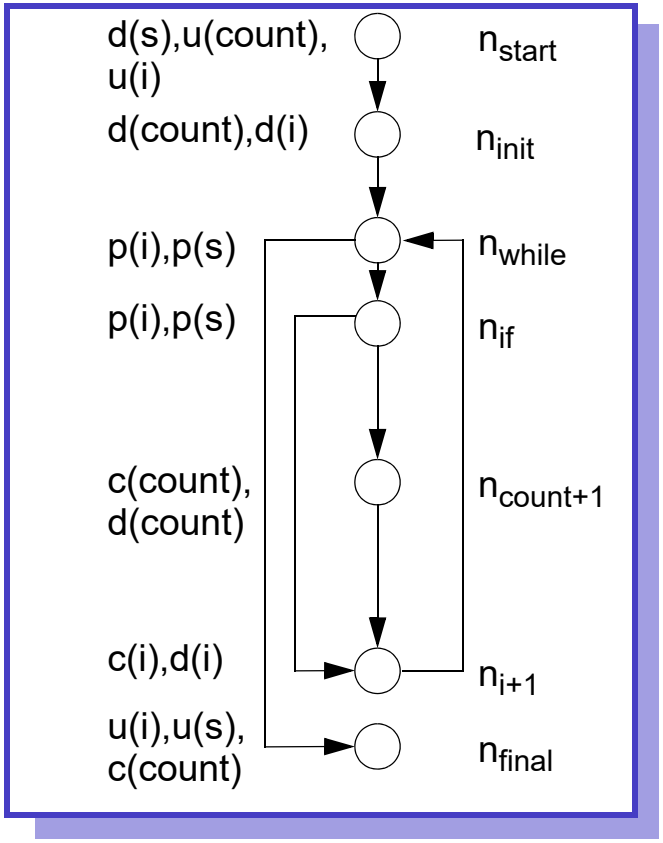
```
PROCEDURE countVowels(IN s: Sentence; OUT count: INTEGER);          (* start *)
  (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)

VAR i: INTEGER;

BEGIN

  count := 0; i := 0;(* init *)

  WHILE s[i] # '.' DO

    IF s[i]= 'a' OR s[i]= 'e' OR
       s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'
    THEN

      count := count+1;

    END;

    i := i+1;

  END (* WHILE *)
END countVowels;(* final *)
```

Achtung:

Reihenfolge der Datenfluss-attribute ist (manchmal) wichtig:

von links nach rechts (oben nach unten) lesen und auswerten.

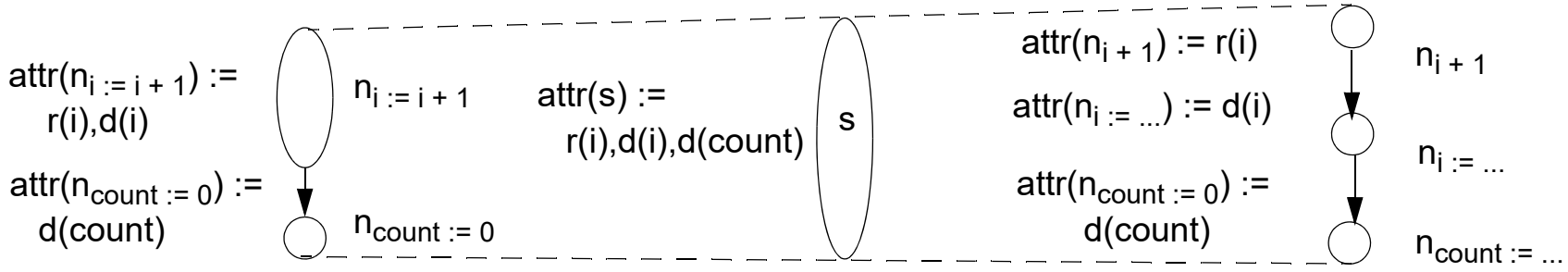
Mehrfache Auftreten von Datenfluss-attributen an einem Knoten werden meist zusammen-gefasst.



Kontrollflussgraphsegmente mit Datenflussattributen:

- Sei s ein Kontrollflussgraphsegment mit Pfad $n_{start} = n_1, \ldots, n_k = n_{final}$ und $attr(n_i)$ die Sequenz der Datenflussattribute der n_i. Dann ist $attr(s) := attr(n_1), \ldots, attr(n_k)$ die Konkatenierung der Datenflussattributsequenzen von n_1 bis n_k
- Ein Knoten n mit einer Sequenz von Datenflussattributen $attr(n) := attr_1, \ldots, attr_k$ ist immer eine abkürzende Schreibweise für einen Pfad von Knoten $n_1, \ldots, n_k$, sodass jeder Knoten n_i genau ein Datenflussattribut besitzt $attr(n_i) := attr_i$
- Allen folgenden Definitionen liegt immer ein „segmentfreier“ Kontrollflussgraph zugrunde, in dem jeder Knoten genau ein Datenflussattribut besitzt

Beispiel für Zusammenfassung/Expansion von Kontrollflussgraphknoten:



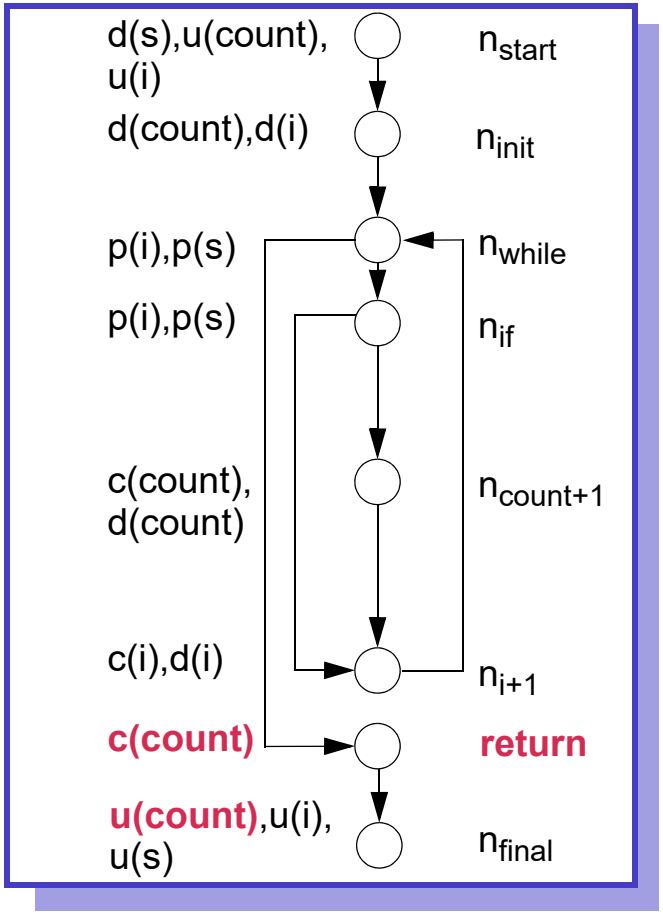
Werkzeug PMD - Kontrollflussgraph mit Datenflussattributen:

Line	Graph	Next nodes	Dataflow types
4		1	u(r)
4		2	
5		3	
7		4, 7	
9		5, 6	
10		4	d(r)
12		8	r(r)
14		8	d(r)
16		9	r(r)
17			u(r)

Type	Line(s)	Variable	Method
UR	4, 12	r	littleGauss
DD	10	r	littleGauss

Kontrollflussgraph mit Datenflussattributen - verändertes Beispiel:

```
PROCEDURE countVowels(IN s: Sentence) : INTEGER;          (* start *)
VAR count, i: INTEGER;
BEGIN
   count := 0; i := 0;(* init *)
   WHILE s[i] # '.' DO
      IF s[i]= 'a' OR s[i]= 'e' OR
         s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'
      THEN
         count := count+1;
      END;
      i := i+1;
   END (* WHILE *)
   RETURN count;
END countVowels;(* final *)
```



Inout-Parameterdiskussion und Aufruf von Prozeduren:

Die meisten Programmiersprachen erlauben **nicht** die Auszeichnung von Variablen, die reinen Ausgabeparameter-Charakter besitzen; in Pascal/Modula gibt es nur mit VAR gekennzeichnete **Inout-Parameter**, in Sprachen wie C oder C++ hat man nur die Möglichkeit, out-Parameter als Zeiger/Referenzen auf Variable/Objekte zu simulieren.

Für solche Parameter mit Ein- und Ausgabecharakter müssen wir wie folgt vorgehen:

- ⇨ Deklaration der Prozedur countVowels(IN s: ... ; INOUT count: ...):
  für n_{start} wird d(count) sowie d(s) angenommen, da beide Variablen bei der Übergabe einen definierten Wert haben sollten
  für n_{final} wird c(count) angenommen, da am Ende der Prozedur der Wert von count durch versteckte Zuweisung an Aufrufstelle übergeben wird
- ⇨ Aufruf der Prozedur countVowels(aSentence, aCounter):
  es wird c(aSentence) und c(aCounter) in normaler Anweisung oder p(aSentence) und p(aCounter) in Prädikat gefolgt von d(aCounter) angenommen, da beim Aufruf versteckte Zuweisungen die Werte von aSentence und aCounter an die Parameter s und count zuweisen

Felder, globale Variablen und Strukturen:

- Zugriff auf **globale Variablen** in einer Prozedur (Methode): werden bei Ein- und Austritt aus der Prozedur ignoriert und ansonsten wie lokale Variablen behandelt
- Zugriff auf **Felder (Arrays)**:
  - ⇨ anArray[index] := ... wird als r(index) und d(anArray) gewertet
  - ⇨ ... := ... anArray[index] ... wird als r(index) und r(anArray) gewertet
- Zugriff auf **Strukturen**: wenn notwendig, können die Bestandteile (Variablen) einer Struktur als einzelne Variablen behandelt werden:

```
struct adresse {
  char name[50];
  ...
};
struct adresse adrAndy;
```

Anstelle von adrAndy werden also Variablen adrAndy.name, ... betrachtet.

Datenflussgraph - formale Definition:

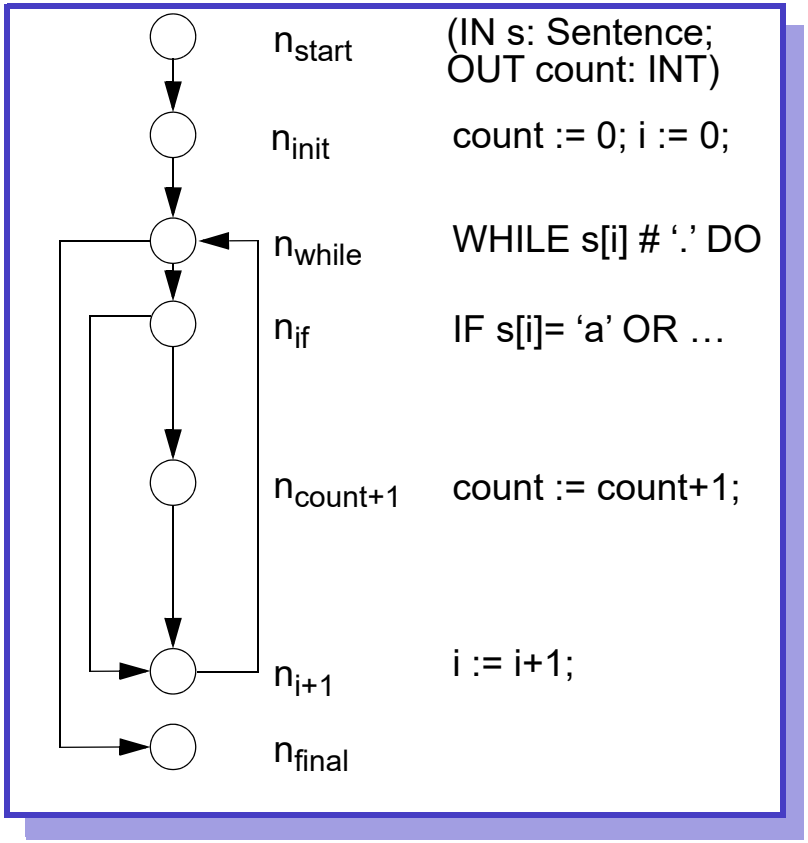
Ein **Datenflussgraph** $D = (V_d, E_d)$ zu einem Kontrollflussgraphen G eines Programms besteht aus

- ⇨ einer Menge von Knoten V_d für alle Anweisungen V des Programms
- ⇨ einer Menge von **Datenflusskanten** E_d :
  $(n_1, n_2) \in E_d$ genau dann, wenn es einen Pfad p im Kontrollflussgraphen G von n_1 nach n_2 gibt, sodass für eine Variable/Parameter v gilt:
  1. d(v) für n_1 : Anweisung n_1 definiert Wert für v
  2. r(v) für n_2 : Anweisung n_2 benutzt Wert von v
  3. für alle Anweisungen n auf Pfad p (ohne n_1) gilt nicht d(v) für n:
     n ändert also nicht den bei n_1 festgelegten Wert von v

Die Kanten des Datenflussgraphen verbinden also Zuweisungen an Variablen oder Parameter mit den Anweisungen, in denen die zugewiesenen Werte benutzt werden.

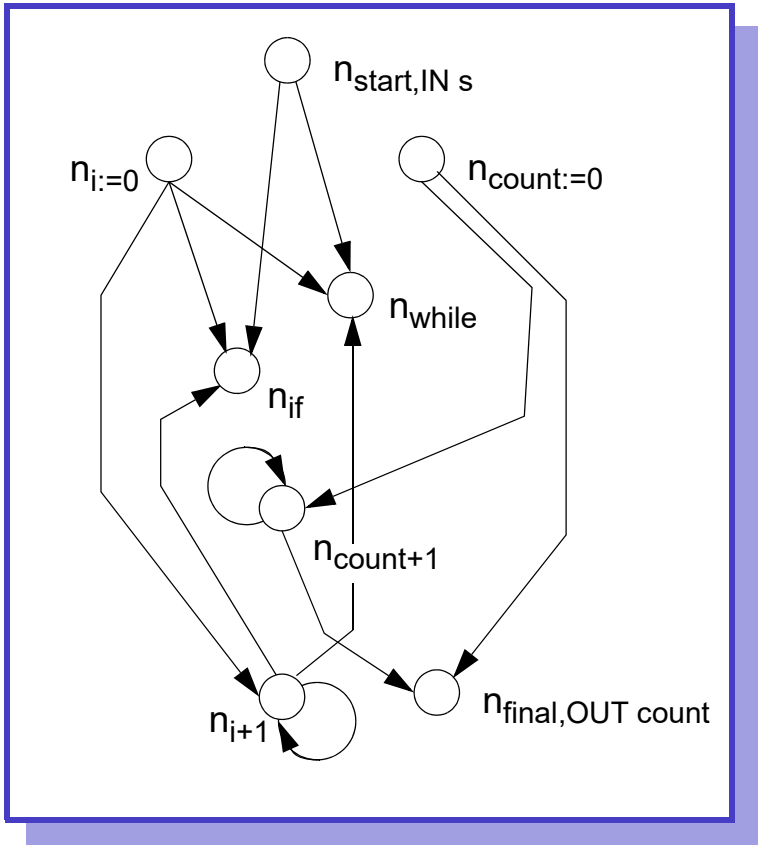
Beispiel für Datenflussgraph zu countVowels:

Kontrollflussgraph (countVowels):



Knoten = Anweisungsblöcke
Kanten = Kontrollfluss

Datenflussgraph (countVowels):



Knoten = setzende/lesende Variablenzugriffe
Kanten = Datenfluss der zugewiesenen Werte

Programm-Slices zur Fehlersuche und Änderungsfolgeschätzung:

Ein **Abhängigkeitsgraph** $A = (N_a, E_a)$ eines Programms ist ein gerichteter Graph, der

- ⇨ alle Knoten und Kanten des Datenflussgraphen enthält (ohne Zusammenfassung von Teilgraphen zu Segmenten) sowie zusätzlich
- ⇨ Kanten (des Kontrollflussgraphen) von allen Bedingungen zu direkt kontrollierten Anweisungen (das sind die Anweisungen, deren Ausführung von der Auswertung der betrachteten Bedingung abhängt).

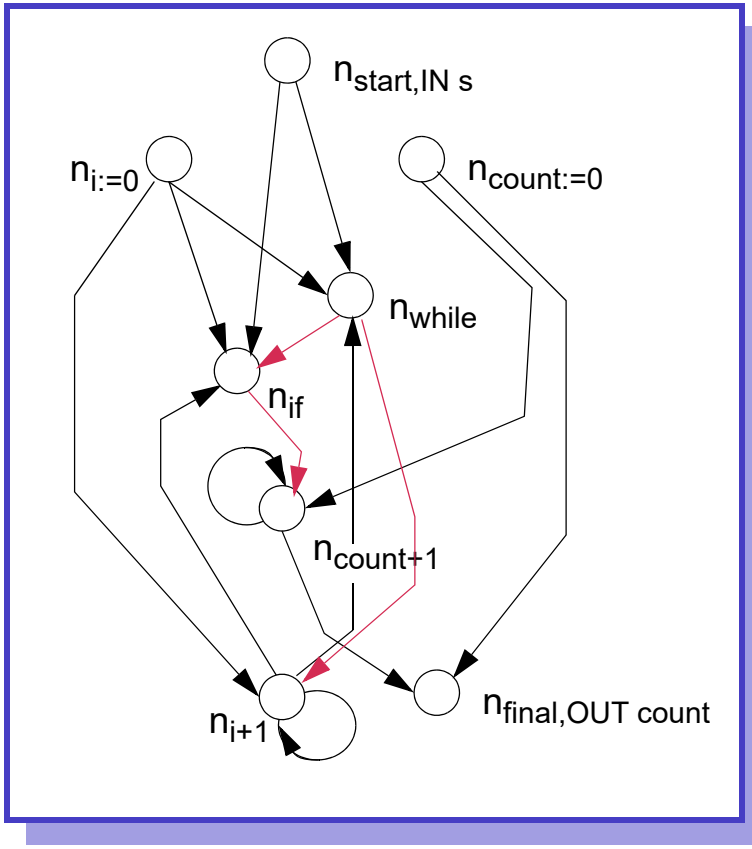
Es gibt zwei Arten von Ausschnitten (Slices) eines Abhängigkeitsgraphen A:

- ❑ **Vorwärts-Slice** für Knoten $n \in N_a$ mit Datenflussattribut d(v):
  alle Pfade in A, die von Knoten n ausgehen, der Variable v definiert
  (der Slice-Graph enthält alle Knoten und Kanten der Pfade)
- ❑ **Rückwärts-Slice** für Knoten $n \in N_a$ mit Datenflussattribut r(v):
  alle Pfade in A, die in Knoten n einlaufen, der Variable v referenziert
  (der Slice-Graph enthält alle Knoten und Kanten der Pfade)

Abhängigkeitsgraph zu countVowels:

```
PROCEDURE countVowels(IN s: Sentence;
          OUT count: INTEGER);

VAR i: INTEGER;

BEGIN

   count := 0; i := 0;(* init *)

   WHILE s[i] # '.' DO

      IF s[i]= 'a' OR s[i]= 'e' OR
         s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'
      THEN

         count := count+1;

      END;

      i := i+1;

   END (* WHILE *)
END countVowels;(* final *)
```

Slice für alle Knoten von countVowels:



schwarze Kanten: Datenflusskanten
rote Kanten: Kontrollflusskanten

Vorwärts-Slice - Beispiel und Nutzen:

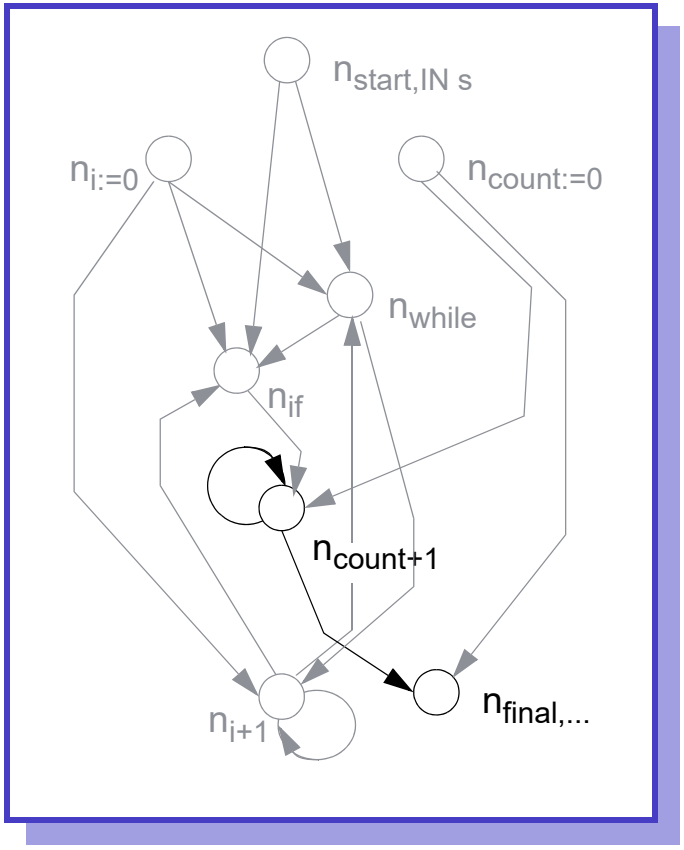
Ein **Vorwärts-Slice** zu einer Anweisung n, die einer Variable v einen Wert zuweist, bestimmt alle die Stellen eines Programms, die von einer Änderung des zugewiesenen Wertes (Berechnungsvorschrift) betroffen sein könnten.

Der Vorwärts-Slice zu count := count+1:

```
...
        count := count+1;
...
END countVowels;(* final mit Rückgabe count *)
```

Ein Vorwärts-Slice dient der Abschätzung von Folgen einer Programmänderung.

Vorwärts-Slice für count := count+1:



Rückwärts-Slice - Beispiel und Nutzen:

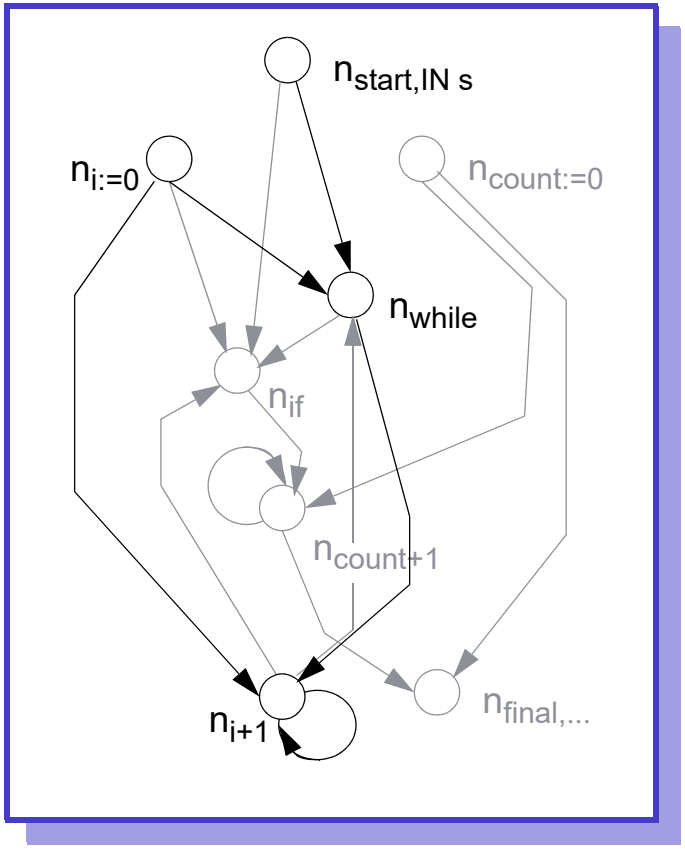
Ein **Rückwärts-Slice** zu einer Anweisung n, die eine Variable referenziert, bestimmt alle die Stellen eines Programms, die den Wert der Variable direkt oder indirekt bestimmt haben.

Der Rückwärts-Slice zu i := i+1:

```
PROCEDURE countVowels(IN s: Sentence;
            OUT count: INTEGER);
   VAR i: INTEGER;
BEGIN   (* start *)
   count := 0; i := 0;
   WHILE s[i] # '.' DO
      ...
      i := i+1;
```

Ein Rückwärts-Slice ist bei der Fehlersuche hilfreich, um schnell irrelevante Programmteile ausblenden zu können.

Rückwärts-Slice für i := i+1:



Datenfluss- und Kontrollflussanomalien:

Eine **Anomalie** eines Softwareprodukts ist eine Unstimmigkeit, die durch Abweichung von (berechtigten) Erwartungen ausgelöst wird, bzw. eine verdächtige Stelle in einem Softwareprodukt. Eine solche verdächtige Stelle ist keine garantiert fehlerhafte, aber eine potentiell fehlerhafte Stelle im betrachteten Softwareprodukt.

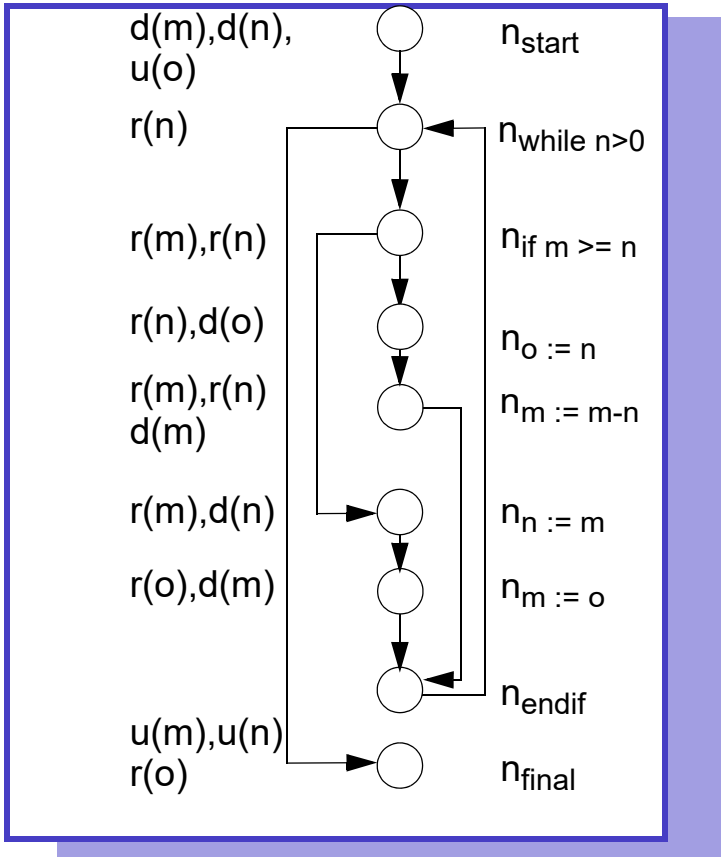
Datenflussanomalien (meist deutliche Hinweise auf Fehler) sind etwa:

- ⇨ es gibt einen Pfad im Kontrollflussgraphen, auf dem eine Variable v referenziert wird bevor sie zum ersten Mal definiert wird (Zugriff auf undefinierte Variable)
- ⇨ es gibt einen Pfad im Kontrollflussgraphen, auf dem eine Variable v zweimal definiert wird ohne zwischen den Definitionsstellen referenziert zu werden (nutzlose Zuweisung an Variable)

Kontrollflussanomalien sind bei modernen Programmiersprachen von geringerer Bedeutung. Im wesentlichen handelt es sich dabei bei Programmiersprachen ohne „goto“-Anweisungen um nicht erreichbaren Code (ansonsten beispielsweise Sprünge in Schleifen hinein).

Beispiel für Programm mit Datenflussanomalien:

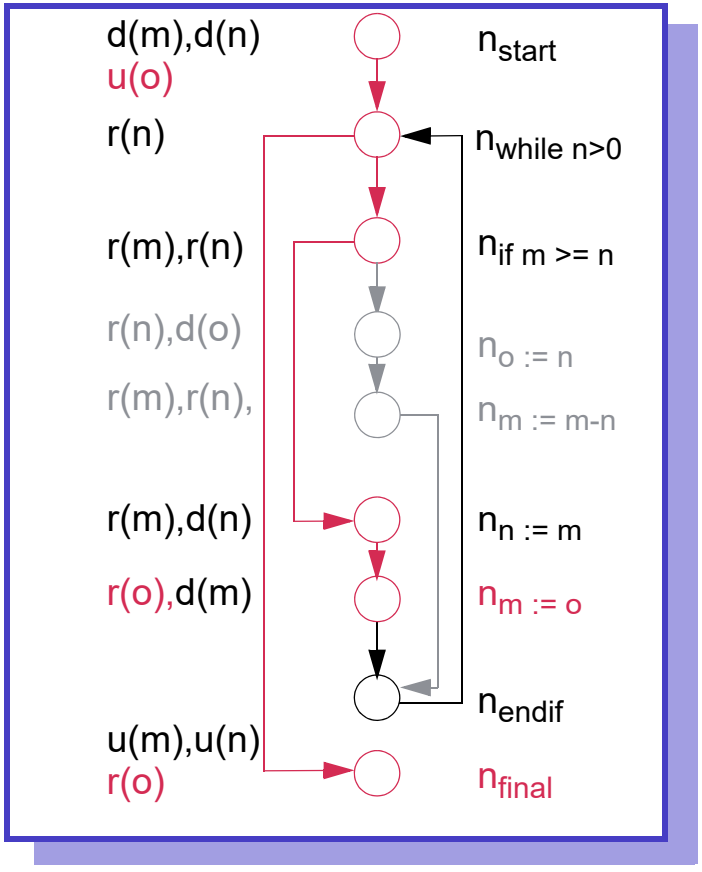
```
PROCEDURE ggT(IN m, n: INTEGER; OUT o: INTEGER);          (* start *)
BEGIN
   WHILE n>0 DO
      IF m >= n  THEN
         o := n;
         m := m-n;
      ELSE
         n := m;
         m := o;
      END (* endif *)
   END (* while *)
END ggT; (* final *)
```



Undefined-Reference-Datenflussanomalie:

Eine **ur-Datenflussanomalie** bezüglich einer Variable v ist wie folgt definiert:

- ⇨ es gibt einen segmentfreien Pfad $n_1, \ldots n_k$
- ⇨ n_1 hat Attribut u(v), v besitzt also keinen definierten Wert bei n_1
- ⇨ $n_2, \ldots, n_{k-1}$ hat nicht Attribut d(v), v erhält also keinen definierten Wert
- ⇨ n_k hat Attribut r(v), auf v wird also lesend zugegriffen

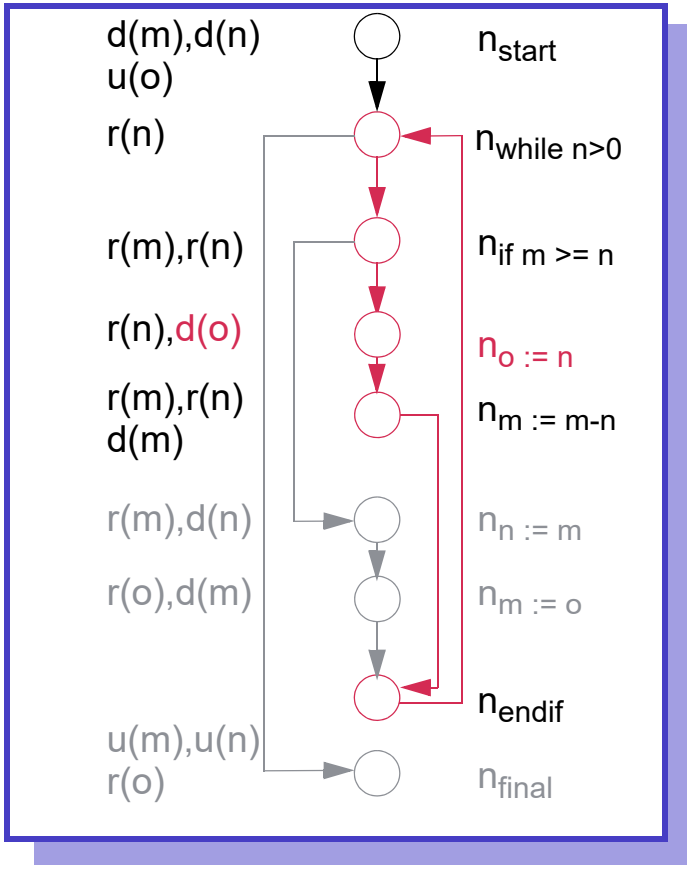


Beispiele für ur-Datenflussanomalien:

Defined-Defined-Datenflussanomalie:

Eine **dd-Datenflussanomalie** bezüglich einer Variable v ist wie folgt definiert:

- ⇨ es gibt einen segmentfreien Pfad $n_1, \ldots, n_k$
- ⇨ n_1 hat Attribut d(v), v erhält also bei n_1 einen neuen Wert
- ⇨ $n_2, \ldots, n_{k-1}$ hat nicht Attribut [r|d|u](v), v wird also bis n_k nicht verwendet
- ⇨ n_k hat Attribut d(v), alter Wert von v wird bei n_k unbenutzt überschrieben

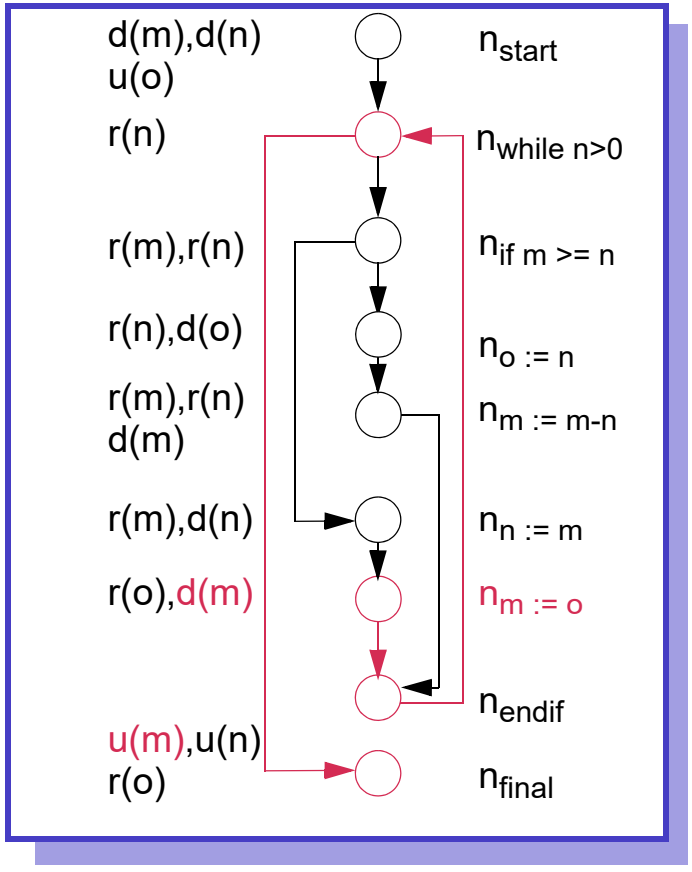


Beispiel für dd-Datenflussanomalie:

Defined-Undefined-Datenflussanomalie:

Eine **du-Datenflussanomalie** bezüglich einer Variable v ist wie folgt definiert:

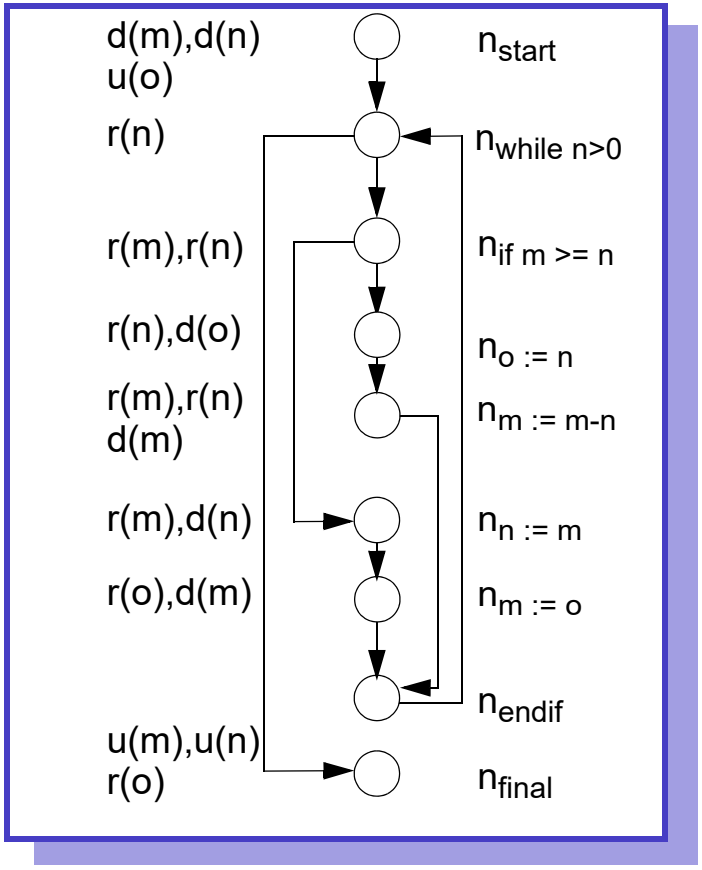
- ⇨ es gibt einen segmentfreien Pfad $n_1, \ldots, n_k$
- ⇨ n_1 hat Attribut d(v), v erhält also einen definierten Wert bei n_1
- ⇨ $n_2, \ldots, n_{k-1}$ hat nicht Attribut [r|d|u](v), v wird also bis n_k nicht verwendet
- ⇨ n_k hat Attribut u(v), v wird also auf undefiniert gesetzt



Beispiel für du-Datenflussanomalie:

Korrektur des Programms mit Datenflussanomalien - geht das?

```
PROCEDURE ggT(IN m, n: INTEGER; OUT o: INTEGER);          (* start *)
BEGIN
   WHILE n>0 DO
      IF m >= n  THEN
         o := n;  (* dd für o *)
         m := m-n;  (* du für m *)
      ELSE
         n := m;
         m := o;   (* ur für o, du für m *)
      END (* endif *)
   END (* while *)
END ggT;  (* ur für o *)
```



Korrigiertes ggT-Programm mit verbleibender du-Datenflussanomalie:

```
PROCEDURE ggT(IN m, n: INTEGER; OUT o: INTEGER);          (* start *)
BEGIN
  o := n;
  WHILE n > 0 DO
    IF m >= n  THEN
      m := m-n (**)
    ELSE
      n := m;
      if n > 0 THEN
        m := o; (***)
        o := n;
      END
    END
  END
END ggT; (* final *)
```

- die Anweisung „m := o;“ führt statisch gesehen zu einer du-Anomalie, da while-Schleife nach Zuweisung beendet sein könnte
- tatsächlich passiert aber zur Laufzeit das:
  - ⇨ wenn m ein neuer Wert bei (***) zugewiesen wird, muss n größer als 0 sein
  - ⇨ dann wird die while-Schleife wieder betreten und auf m lesend zugegriffen
  - ⇨ es gibt also in Realität keine nutzlose Zuweisung an m
- weitere scheinbare du-Anomalie wird durch die Zuweisung (**) hervorgerufen
- echte du-Anomalie: auf m wird für n = 0 nie lesend zugegriffen

Probleme mit statischer Datenflussanalyse für Datenstrukturen:

- ☹ funktioniert nicht (gut) für komplexe Datenstrukturen wie Felder (Arrays), bei denen man jede einzelne Komponente wie eigene Variable behandeln müsste:

```
s[i] := x;
y := s[k];
```

  Obiges Programmfragment hat keine du- und ur-Anomalie, falls i = k ; aber die Gleichheit von i und k (oder anderen Ausdrücken) lässt sich im allgemeinen nur schwer garantieren.

- ☹ noch größere Probleme hat man bei verzeigerten Datenstrukturen:

```
delete(obj1);
obj3 := obj2;
```

  Obiges Programmfragment ist nur dann in Ordnung, wenn die Variablen obj1 und obj2 nicht auf dasselbe Objekt zeigen (sonst würde nämlich in der zweiten Zeile u(obj1) und u(obj2) gelten).

- ☹ Unterscheidung von in-, out-, und inout-Parametern (in Java, C++, ...)

Probleme mit statischer Datenflussanalyse bei Fallunterscheidungen:

```
IF Bed1 THEN x := expr1 ELSE y := expr2 END;
IF Bed2 THEN z := x ELSE z := y END;
```

Das obige Programm funktioniert, falls Bed1 und Bed2 äquivalent sind.
Trotzdem liefert die Datenflussanalyse immer Anomalien, da von ihr die Äquivalenz von Bed1 und Bed2 nicht erkannt wird.

Problem:

- ⇨ reale Programme enthalten oft viele Anomalien, die nicht echte Programmierfehler sind (zu viele nutzlose Warnungen werden erzeugt)
- ⇨ restriktivere Definitionen von sogenannten starken Anomalien übersehen andererseits u.U. zu viele echte Fehler (siehe folgende Folien)

Lösung:

- ⇨ zunächst neue Definitionen **„starker“ Anomalien** verwenden
- ⇨ dann bisherige Definitionen von **(schwachen) Anomalien** verwenden

Definitionen starker Datenflussanomalien:

Geg. Kontrollflussgraph G zu Programm P mit Datenflussattributen (ohne Segmente):

- **starke ur-Anomalie**: zu Anweisungen n mit Attribut u(v) und über einen Pfad von n erreichbarem n' mit r(v) gibt es **keinen** segmentfreien Pfad in G $n = n_1, \ldots, n_k = n'$ in dem n' nur einmal auftritt und für den gilt:

  es existiert $i \in 2, \ldots, k\text{-}1$ mit: n_i besitzt Attribut d(v) oder u(v)

  Idee dieser Definition:

  - ⇨ von n mit u(v) nach n' mit r(v) gibt es **mindestens** einen Ausführungspfad
  - ⇨ ausgeschlossen werden **zyklische Pfade** durch n', um so die Analyse auf die erste Ausführung einer Anweisung in einer Schleife zu einzuschränken
  - ⇨ auf **keinem** Pfad wird an Variable v ein Wert zugewiesen, bevor bei n' lesend auf v zugegriffen wird
  - ⇨ des weiteren werden Situationen ausgeschlossen, bei denen die gerade betrachtete ur-Anomalie (teilweise) durch irgendeine andere Anomalie „**überlagert**" wird

Definitionen starker Datenflussanomalien - Fortsetzung:

- **starke du-Anomalie**: zu Anweisungen n mit Attribut d(v) und über einen Pfad von n erreichbarem n' mit u(v) gibt es **keinen** segmentfreien Pfad in G $n = n_1 , ... , n_k = n'$ in dem n' nur einmal auftritt und für den gilt:

  es existiert $i \in 2, ... , k-1$ mit: n_i besitzt Attribut d(v) oder u(v) oder r(v)

  Idee dieser Definition:

  - ⇨ von n mit d(v) nach n' mit u(v) gibt es **mindestens** einen Ausführungspfad
  - ⇨ ausgeschlossen werden wieder **bestimmte Zyklen**, um so die Analyse auf die erste Ausführung einer Anweisung in einer Schleife zu einzuschränken
  - ⇨ auf **keinem** Pfad wird an Variable v bei n zugewiesener Wert verwendet, bevor er bei n' undefiniert wird
  - ⇨ des weiteren werden Situationen ausgeschlossen, bei denen die gerade betrachtete du-Anomalie (teilweise) durch irgendeine andere Anomalie „**überlagert**“ wird

Definitionen starker Datenflussanomalien - Fortsetzung:

- **starke dd-Anomalie**: zu Anweisungen n mit Attribut d(v) und über einen Pfad von n erreichbarem n' mit d(v) gibt es **keinen** segmentfreien Pfad in G $n = n_1, \ldots, n_k = n'$ in dem n' nach n_1 nur einmal auftritt und für den gilt

  es existiert $i \in 2, \ldots, k\text{-}1$ mit: n_i besitzt Attribut d(v) oder u(v) oder r(v)

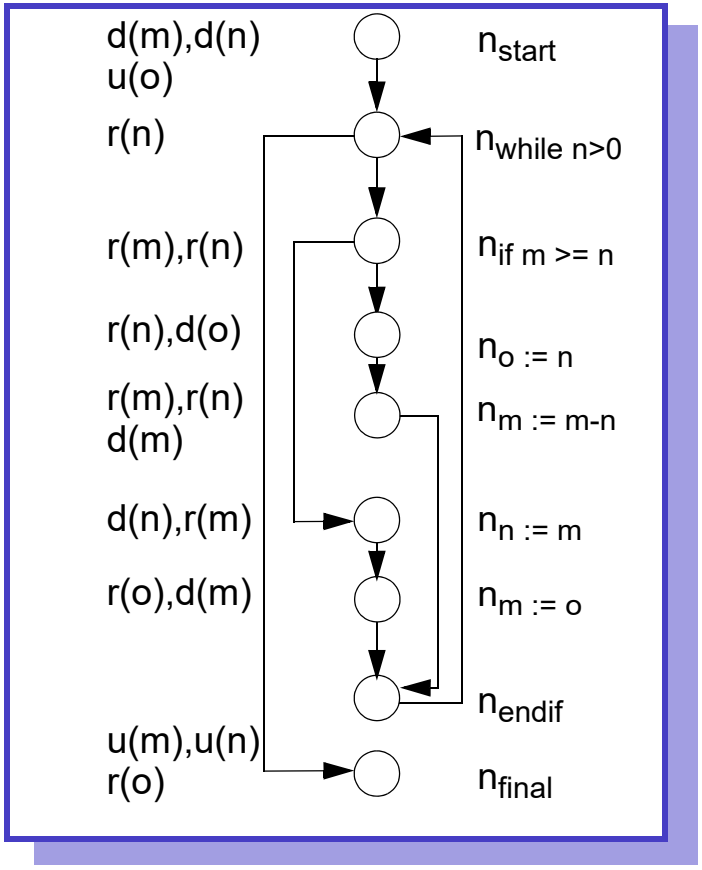
  Idee dieser Definition:

  - ⇨ von n mit d(v) nach n' mit d(v) gibt es **mindestens** einen Ausführungspfad
  - ⇨ ausgeschlossen werden wieder **bestimmte Zyklen**, um so die Analyse auf die erste Ausführung einer Anweisung in einer Schleife zu einzuschränken
  - ⇨ auf **keinem** Pfad wird an Variable v bei n zugewiesener Wert verwendet, bevor bei n' erneut ein Wert zugewiesen wird
  - ⇨ des weiteren werden Situationen ausgeschlossen, bei denen die gerade betrachtete dd-Anomalie (teilweise) durch irgendeine andere Anomalie „**überlagert**" wird

Achtung: unser Programm ggT enthält keine starken Datenflussanomalien, sondern ausschließlich schwache Datenflussanomalien!

Beispiel für Programm ohne starke Datenflussanomalien:

```
PROCEDURE ggT(IN m, n: INTEGER; OUT o: INTEGER);          (* start *)
BEGIN
  WHILE n>0 DO
    IF m >= n  THEN
      o := n; (* keine starke dd-Anomalie *)
      m := m-n;
    ELSE
      n := m;
      m := o; (* keine starke ur-Anomalie *)
    END (* endif *)
  END (* while *)
END ggT; (* keine starke du-Anomalie für m *)
```



3.5 Softwaremetriken

Die Definition von Software-Maßen basiert auf dem Wunsch, einen quantitativen Zugang zum abstrakten Produkt Software zu gewinnen. Dabei ist zwischen der Vermessung von Eigenschaften einer Software und der quantitativen Kontrolle des zugrundeliegenden Entwicklungs-prozesses zu unterscheiden. *[Li02]*

- **Produktmetriken** messen Eigenschaften der Software:
  - ⇨ Qualität der Software (z.B. als Anzahl gefundener Fehler)
  - ⇨ Einhaltung von Standards (z.B. als Anzahl Verletzung von Stilregeln)
- **Prozessmetriken** messen Eigenschaften des Entwicklungsprozesses:
  - ⇨ Dauer oder Kosten der Entwicklung (z.B. als Mitarbeitermonate)
  - ⇨ Zufriedenheit des Kunden (z.B. als Anzahl Änderungswünsche)?

Achtung: die Software-Engineering-Literatur verwendet meist die Begriffe „Maß“ und „Metrik“ als Synonyme (Metrik in der Mathematik: Entfernung zweier Punkte, die gemessen werden kann)

Gewünschte Eigenschaften von Maß/Metrik:

- **Einfachheit**: berechnetes Maß lässt sich einfach interpretieren (z.B. Zeilenzahl einer Datei)
- **Eignung** (Validität): es besteht ein (einfacher) Zusammenhang zwischen der gemessenen Eigenschaft und der interessanten Eigenschaft (z.B. zwischen Programmlänge und Fehleranzahl)
- **Stabilität**: gemessene Werte sind stabil gegenüber Manipulationen untergeordneter Bedeutung (z.B. die Unterschiede zwischen zwei Projekten, wenn man aus erstem Projekt Rückschlüsse auf zweites Projekt ziehen will)
- **Rechtzeitigkeit**: das Maß kann zu einem Zeitpunkt berechnet werden, zu dem es noch zur Steuerung des Entwicklungsprozesses hilfreich ist (Gegenbeispiel: Programmlänge als Maß für Schätzung des Entwicklungsaufwandes)
- **Reproduzierbarkeit**: am besten automatisch berechenbar ohne subjektive Einflussnahme des Messenden (Gegenbeispiel: Beurteilung der Lesbarkeit eines Programms durch manuelle Durchsicht)

Maßskalen:

- **Nominalskala**: frei gewählte Menge von Bezeichnungen wie etwa Programm in C++, Java, Fortran, ... geschrieben
- **Ordinalskala**: geordnete Menge von Bezeichnern wie etwa Programm gut lesbar, einigermaßen lesbar, ... , absolut grauenhaft
- **Rationalskala**: Messwerte können zueinander in Relation gesetzt werden und prozentuale Aussagen mit Multiplikation und Division sind sinnvoll wie etwa Programm A besitzt doppelt/halb soviele Programmzeilen wie Programm B

  weiteres Beispiel:

Zielsetzung von Softwaremetriken hier:

- Metrik soll zur **Prognose der Fehler** (einer bestimmten Art) in einem Softwaresystem eingesetzt werden
- **Softwarequalität** wird also mit etwas „leicht“ messbaren wie der Anzahl der Fehler pro Codezeile (die bis zum Zeitpunkt x gefunden wurden) gleichgesetzt
- **1. Hypothese:** komplexer Code enthält mehr Fehler als einfacher Code (pro Zeile Quelltext)
- gesucht werden also **Metriken für Komplexität** eines untersuchten (gemessenen) Softwaresystems
- **2. Hypothese**: es gibt einfachen Zusammenhang zwischen der gemessenen Softwarekomplexität und der Anzahl später gefundener Fehler (pro Codezeile)

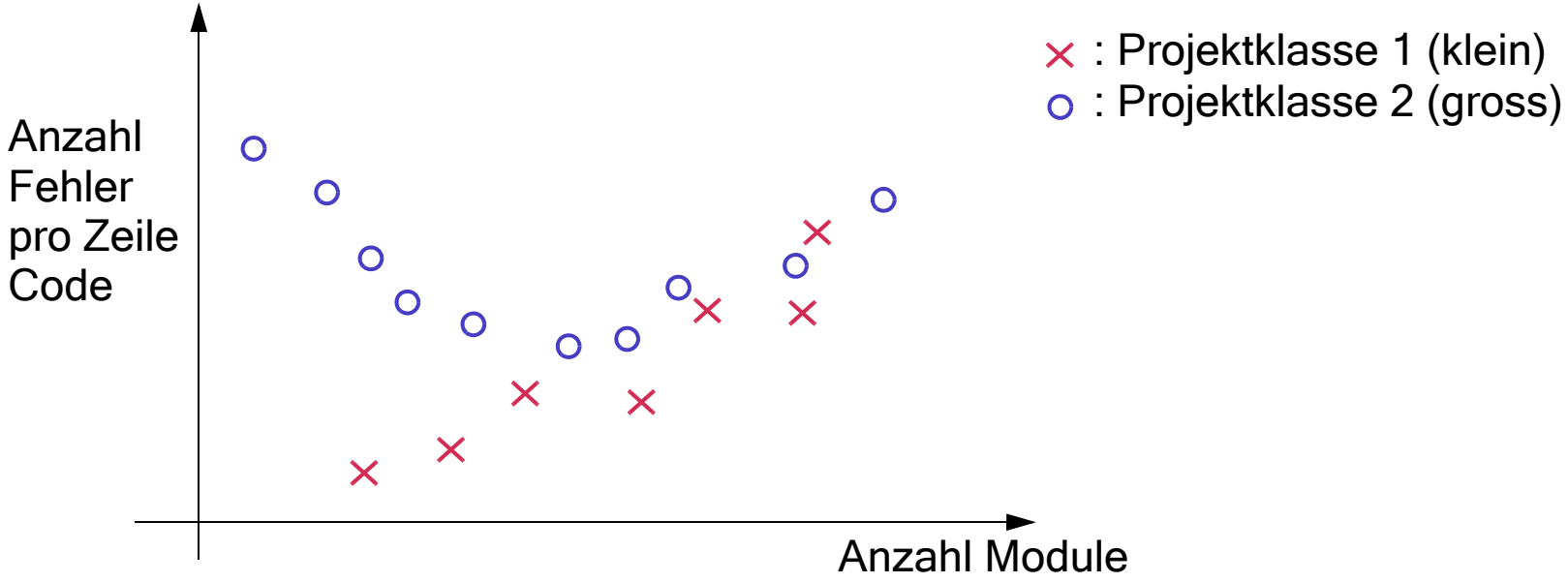
Frage:

Wie „validiert“ man all diese Hypothesen???

Beispiel für eine falsche Hypothese:

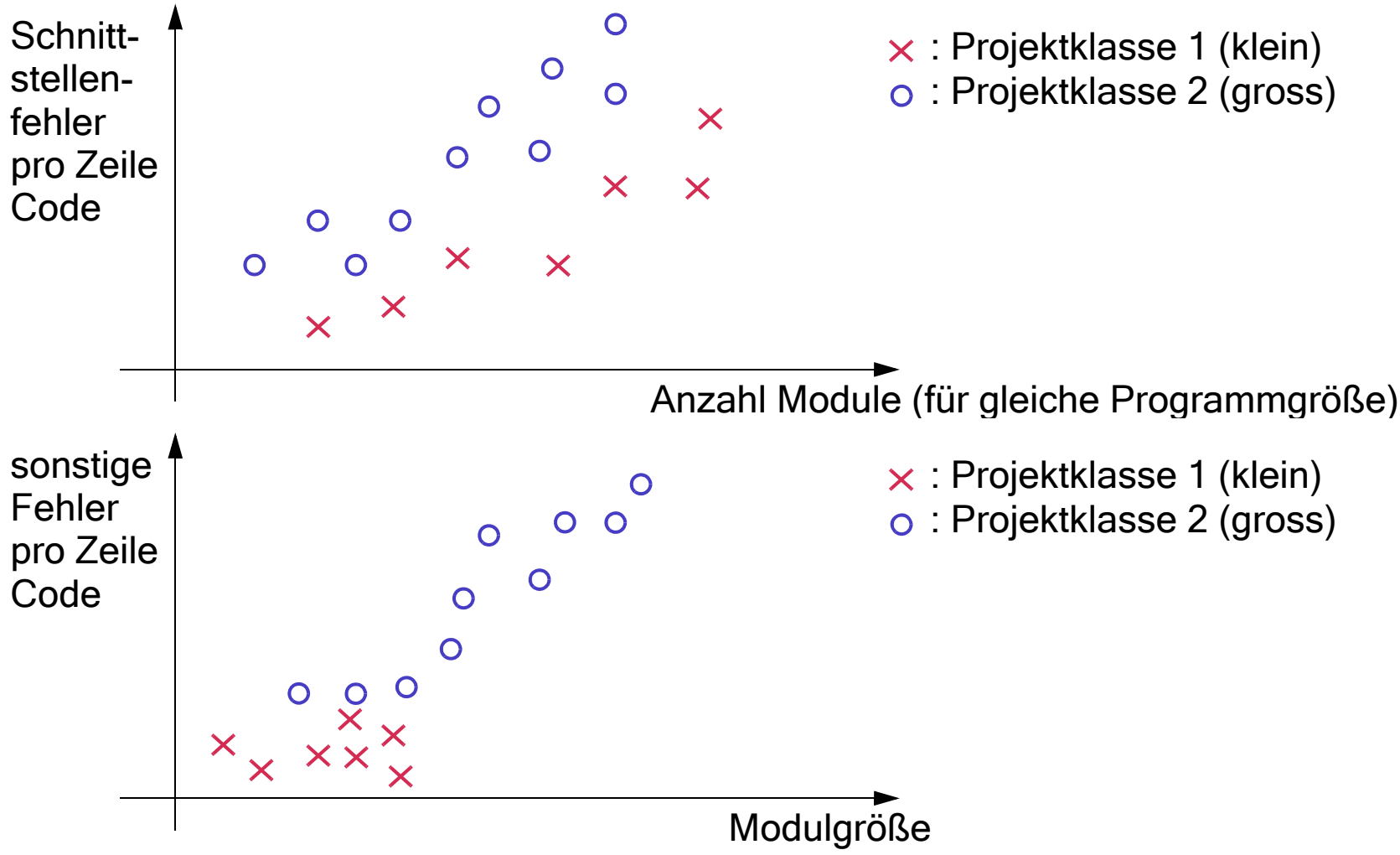
Modularisierung von Programmen verringert die Zahl der Fehler, also steigende Anzahl von Modulen verursacht fallende Fehlerzahl.

Gemessene Werte für verschiedene Projekte:

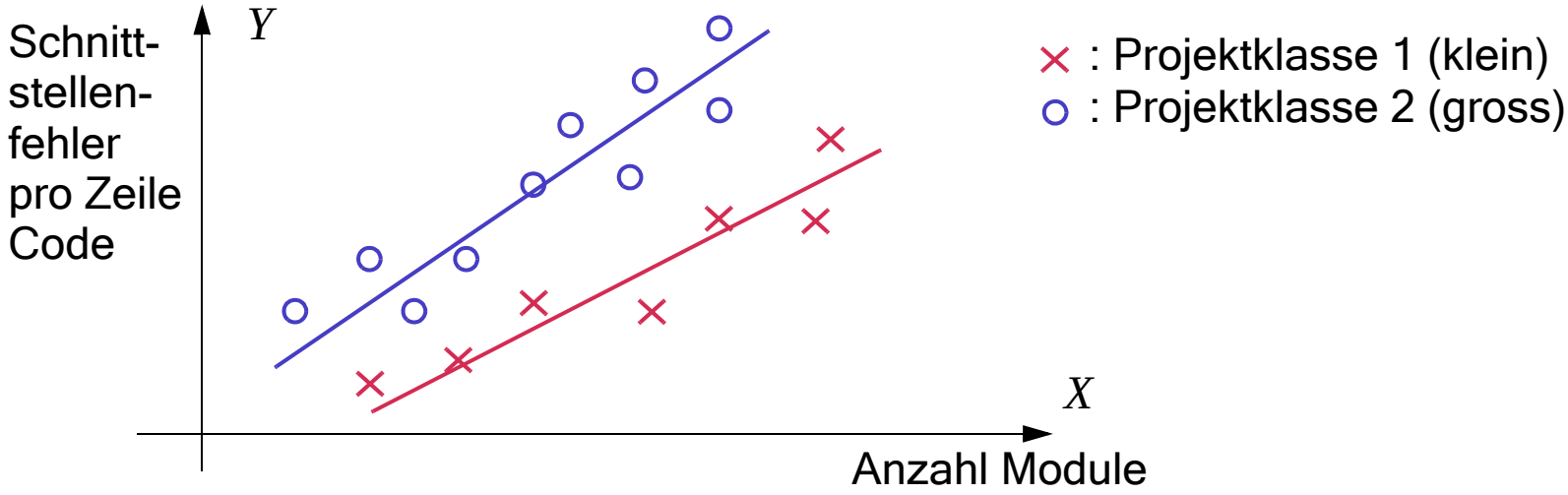


Die hier aufgetragenen Werte sind fiktiv, reale Untersuchungen haben aber qualitativ ähnliche Ergebnisse geliefert.

Eine Erklärung für tatsächlichen Zusammenhang Modulzahl - Fehlerzahl:



Testen von Hypothesen mit Regressionsrechnung:



Getestet werden soll die Hypothese, dass ein linearer Zusammenhang zwischen der Anzahl der Module und der Anzahl der Schnittstellenfehler pro Zeile Code in einem Programm besteht.

1. es wird die Gerade ermittelt, die die Messwerte nach der Methode der kleinsten Fehlerquadrate am besten approximiert (nach Gauss)

2. es wird berechnet, wie gut die Streuung der Messwerte in *Y*-Richtung durch zufällig verteilte Messfehler um berechnete Gerade herum erklärt wird

Berechnung der Regressionsgeraden:

Gesucht wird: $Y = b_0 + b_1 X$

Gegeben sind Paare von Messwerten: (x_1,y_1) , ... , (x_n,y_n)

Berechnung der Mittelwerte:

Mittelwert $\bar{x} = (x_1 + ... + x_n) / n$

Mittelwert $\bar{y} = (y_1 + ... + y_n) / n$

Berechnung von Koeffizient b_1 :

$$b_1 = \frac{\frac{1}{n-1} \cdot \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2} = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n} (x_i - \bar{x})^2}$$

Berechnung von Koeffizient b_0 :

$$b_0 = \bar{y} - b_1 \bar{x}$$

Berechnung des Korrelationskoeffizienten r:

$$r = \frac{\sum_{i=1}^{n} (x_i - \bar{x}) \cdot (y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \bar{x})^2 \cdot \sum_{i=1}^{n} (y_i - \bar{y})^2}}$$

- man kann zeigen, dass der **Korrelationskoeffizient** $r \in [-1 .. +1]$ gilt
- die Grenzfälle $r = +1$ und $r = -1$ treten auf, wenn schon alle gemessenen Punkte (x_i, y_i) auf einer Geraden liegen
- die Regressionsgerade steigt für $r = +1$ und fällt für $r = -1$
- für $r = 0$ verläuft die Gerade parallel zur X-Achse, es besteht also kein (linearer) Zusammenhang zwischen X- und Y-Werten
- r^2 heisst **Bestimmtheitsmaß** und lässt sich interpretieren als Anteil der durch die Regression erklärten Streuung der Y-Werte
- hat man z. B. $r = 0.7$ erhalten, dann ist $r^2 = 0.49$, d.h. 49 % der Streuung der Y-Werte werden durch die lineare Abhängigkeit von X erklärt

Auswertung von ordinalen/rationalen Metriken:

1. aufgrund von „Erfahrungswerten“ sind sinnvolle untere und obere Grenzwerte für einen Messwert bekannt (siehe Kiviatdiagramm)

   ⇨ alle Komponenten (Module, Klassen, Methoden, ...) mit kritischen Werten werden genauer untersucht und ggf. „saniert“ (neu geschrieben)

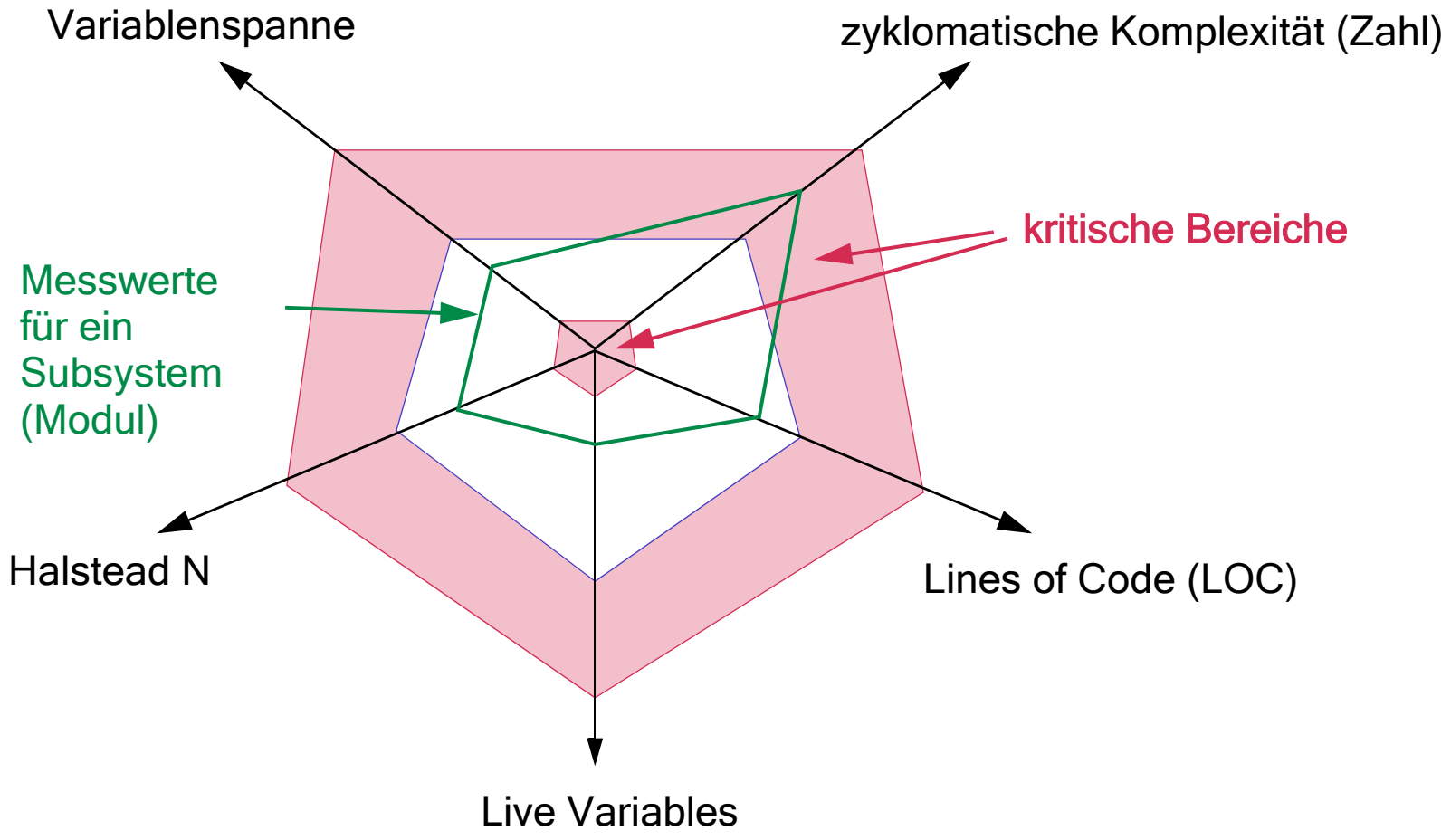
2. solche Grenzwerte für Messergebnisse sind nicht bekannt

   ⇨ alle Komponenten (Module, Klassen, Methoden, ...) werden untersucht, deren Messwerte ausserhalb des Bereichs liegen, in dem 95% der Messwerte liegen (oder 80% oder ...)

3. funktionaler Zusammenhang zwischen Metrik und gewünschtem Qualitätsmerkmal genauer bekannt

   ⇨ zulässige Werte für Metrik werden aus Qualitätsanforderungen errechnet (ein Wunschtraum ...)

Gleichzeitige Darstellung mehrerer Messwerte mit Kiviatdiagramm:



Lines Of Code = LOC:

Die Zeilenzahl des Quelltextes ist die naheliegendste Metrik für ein Softwaresystem (betrachtetes Programmteil).

Doch wie legt man die „**Lines Of Code = LOC**“ einen Programmteil fest?

Möglichkeiten:

Lines of Code (LOC):

LOC(Programmteil) = Anzahl der Knoten im Kontrollflussgraphen

Beispiel:

LOC(countVowels) = 7 (oder 8 ohne init-Segment)

Idee dieser Maßzahl:

- ⇨ betrachtete Programmteile oder ganze Programme mit hoher LOC sind zu komplex (no separation of concerns) und deshalb fehlerträchtig
- ⇨ [Programmteile mit geringer LOC sind zu klein und führen zu unnötigen Schnittstellenproblemen]

Probleme mit dieser Maßzahl:

- ⇨ Kanten = Kontrollflusslogik spielen keine Rolle
- ⇨ wie bewertet man geerbten Code einer Klasse

Zyklomatische Komplexität (Zahl) nach McCabe:

ZZ(Programmteil) = |E| - |N| + 2k

mit G als Kontrollflussgraph des untersuchten Programmteils und

- ⇨ |E| := Anzahl Kanten von G
- ⇨ |N| := Anzahl Knoten von G
- ⇨ k := Anzahl Zusammenhangskomponenten von G
  (Anzahl der nicht miteinander verbundenen Teilgraphen von G)

Beispiel:

ZZ(countVowels) = 8 - 7 + 2 = 3

Regel von McCabe:

ZZ eines in sich abgeschlossenen Teilprogramms (Zusammenhangskomponente) sollte nicht höher als 10 sein, da sonst Programm zu komplex und zu schwer zu testen ist.

Interpretation und Probleme mit der zyklomatischen Komplexität (Zahl):

- es wird die Anzahl der Verzweigungen (unabhängigen Pfade) in einem Programm gemessen
  - ⇨ es wird davon ausgegangen, dass jede Zusammenhangskomponente (Teilprogramm) genau einen Eintritts- und einen Austrittsknoten hat
  - ⇨ damit besitzt jede Zusammenhangskomponente mit *n* Knoten mindestens *n-1* Kanten; diese immer vorhandenen Kanten werden nicht mitgezählt
  - ⇨ die kleinste Komplexität einer Zusammenhangskomponente soll *1* sein, also wird von der Anzahl der Kanten *n* abgezogen und *2* addiert
- in GOTO-freien Programmen wird damit genau die Anzahl der bedingten Anweisungen und Schleifen (if/while-Statements) gemessen
- die Zahl ändert sich nicht beim Einfügen normaler Anweisungen
- deshalb ist die Regel von McCabe mit **ZZ(Komponente) < 11** umstritten, da allenfalls eine Aussage über Testaufwand (Anzahl der zu testenden unabhängigen Programmpfade) getroffen wird

Halstead-Metriken - Eingangsgrößen:

Die Halstead-Metriken messen verschiedene Eigenschaften einer Softwarekomponente. Als Eingabe dienen immer:

- η_1 : Anzahl der unterschiedlichen Operatoren eines Programms (verwendete arithmetische Operatoren, Prozeduren, Methoden, ...)
- η_2: Anzahl der unterschiedlichen Operanden eines Programms (verwendete Variablen, Parameter, Konstanten, ...)
- N_1 : Gesamtzahl der verwendeten Operatoren in einem Programm (jede Verwendungsstelle wird separat gezählt)
- N_2 : Gesamtzahl der verwendeten Operanden in einem Programm (jede Verwendungsstelle wird separat gezählt)
- $\eta := \eta_1 + \eta_2$: Anzahl der verwendeten Deklarationen (**Programmvokabular**)
- $N := N_1 + N_2$: Anzahl der angewandten Auftreten von Deklarationen (wird auch „normale“ **Programmlänge** genannt)

Halstead-Größen eines Programms (Prozedur, Methode):

```
PROCEDURE countVowels(IN s: Sentence; OUT count: INTEGER);          (* start *)
  (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)
VAR i: INTEGER;

BEGIN

  count := 0; i := 0;(* init *)

  WHILE s[i] # '.' DO

  IF s[i]= 'a' OR s[i]= 'e' OR
     s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'
  THEN

     count := count+1;

  END;

  i := i+1;

END countVowels;(* final *)
```

η_1 = Anzahl unterschiedlicher Operatoren =

η_2 = Anzahl unterschiedlicher Operanden =

N_1 = Gesamtzahl verwendeter Operatoren =

N_2 = Gesamtzahl verwendeter Operanden =

$\eta := \eta_1 + \eta_2 =$

$N := N_1 + N_2 =$

Achtung: es ist Vereinbarungssache, ob Kontrollstrukturen, Klammen etc. wie „WHILE“, „IF“, „(“ auch als Operatoren gezählt werden.

In Literatur vorgeschlagene Zählregeln für Java-Programme:

- Arithmetische und logische Standardoperatoren:
  !, !=, %, %=, &, &&, &=, ...
- Auch weitere Operatoren (Sonderzeichen):
  - ⇨ = also Zuweisung
  - ⇨ ; Konkatenation von Anweisungen
  - ⇨ . Attributselektion
  - ⇨ (...) also Klammerungen in Ausdrücken
  - ⇨ ...
- Alle reservierten Java-Schlüsselwörter:
  if, else, switch, case, default, while, ... , class, extends, package, import, static, ...
- Definitionen von Methoden und Funktionen

Halstead-Metriken - Definition:

1. **Berechnete Programmlänge** $L := \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$
   (hängt also nur von Anzahl verwendeter Operatoren und Operanden ab; postuliert wird, dass man mit einer festen Anzahl von Operatoren und Operanden immer Programme einer bestimmten logischen Größe schreibt)
2. **Programmgröße** $V = N \log_2 \eta$ (Programme Volume)
   (optimale Codierung des Programms als Bitvektor)
3. ...

Bewertung:

Es gibt eine ganze Reihe weiterer Halstead-Metriken, deren Nutzen umstritten ist, und die versuchen zu bewerten:

- ⇨ **Schwierigkeit** der Erstellung eines Programms
- ⇨ **Adäquatheit** einer bestimmten Programmiersprache für Problemstellung
- ⇨ **Aufwand** für Erstellung eines Programms

„Live Variable“-Definition:

Die „**Live Variables**“-Metrik berechnet für eine Programmkomponente die durchschnittliche Anzahl lebendiger Variablen dieser Komponente je Knoten des zugehörigen Kontrollflussgraphen; eine Variable ist dabei von ihrer ersten Definitionsstelle (vom Startknoten aus) bis zur letzten Definitions- oder Referenzierungsstelle (vor dem Endknoten) **lebendig.**

Präzise Definition von „Live Variable“ (eine Möglichkeit):

Sei n ein Knoten in einem Kontrollflussgraphen K mit Startknoten n_{start} und Endknoten n_{final}. Dann ist eine Variable v an diesem Knoten n lebendig, falls es

- ⇨ einen Pfad gibt mit n_{start} , ... , n_1 , ... , n , ... , n_2 , ... , n_{final}
- ⇨ in dem der Knoten n_1 das Attribut d(v) besitzt (n_1 = n ist erlaubt)
- ⇨ im dem der Knoten n_2 das Attribut d(v) oder r(v) besitzt (n_2 = n ist erlaubt)
- ⇨ und kein Knoten auf dem Teilpfad von n_1 nach n_2 das Attribut u(v) hat

LV(K) = durchschnittliche Summe lebendiger Var. an allen Knoten von K

Beispiel für „Live Variables“:

```
0:  PROCEDURE swap(INOUT x: INT; INOUT y: INT),
1:     h: INT := x
2:     x := y;
3:     y := h;
4:  END;
```

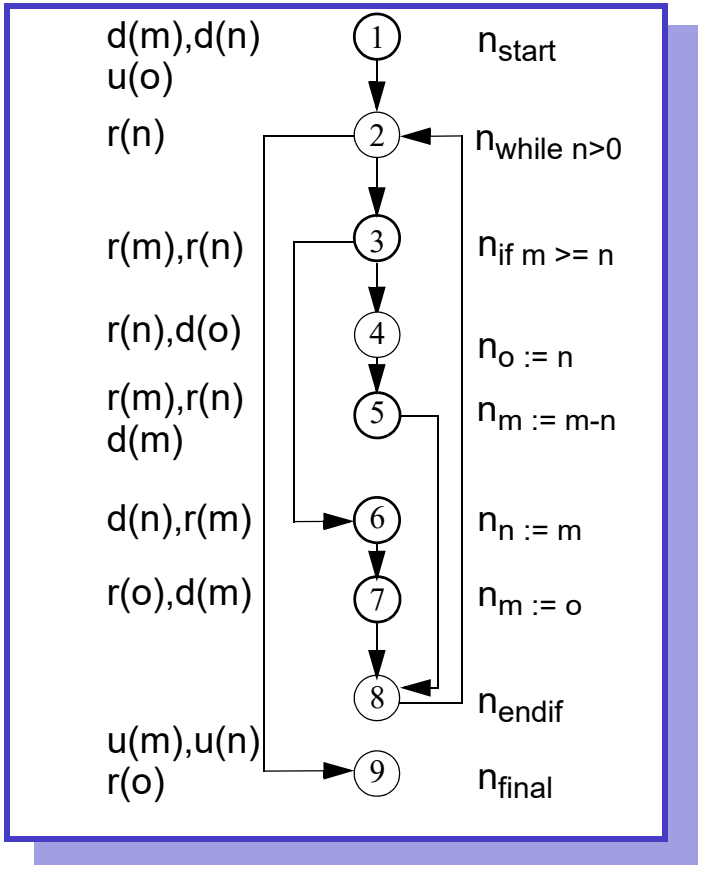
Berechnung der Metrik:

Knoten	define	reference	lebendige Variablen (LV)	Anzahl
0	x, y	---	x, y	2
1	h	x	x, y, h	3
2	x	y	x, y, h	3
3	y	h	x, y, h	3
4	---	x, y	x, y	2

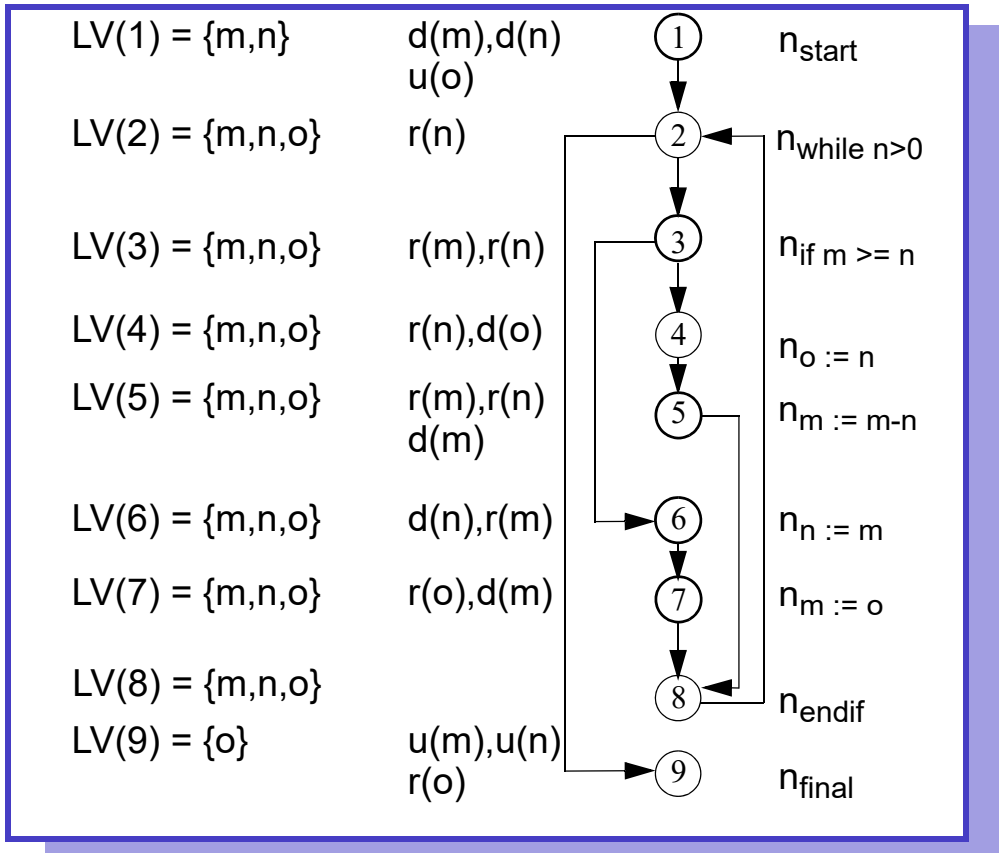
LV(swap) = (2+3+3+3+2) / 5 = 2,6

Größeres Beispiel für die Berechnung von „Live Variables“:

```
PROCEDURE ggT(IN m, n: INTEGER; OUT o: INTEGER);          (* start *)
BEGIN
   WHILE n>0 DO
   IF m >= n  THEN
      o := n;
      m := m-n;
   ELSE
      n := m;
      m := o;
   END;(* endif *)
END ggT;(* final *)
```



Größeres Beispiel für die Berechnung von „Live Variables“:



$$LV(ggT) = (2 + 7 * 3 + 1) / 9 = 2{,}67$$

„Variablenspanne“-Definition:

Die „**Variablenspannen**“-Metrik einer Programmkomponente berechnet die durchschnittliche Spanne zweier direkt aufeinander folgender definierender oder referenzierender Auftreten derselben Variable im zugehörigen Kontrollflussgraphen; die Spanne zweier Knoten in einem Kontrollflussgraphen entspricht der Länge des kürzesten Pfades (Anzahl Kanten dieses Pfades) zwischen diesen beiden Knoten.

Präzise Definition von „Variablenspanne“ (eine Möglichkeit):

Sei v eine Variable und n_1 und n_2 zwei Knoten in einem Kontrollflussgraphen K mit Attributen d(v) und/oder r(v). Dann gilt:

- ⇨ $VS(v, n_1, n_2) = \min(\{ k \mid p(v, n_1, n_2, k) \}) \text{ mit } \min(\{ \}) = 0$
- ⇨ $p(v, n_1, n_2, k)$ = true, falls Pfad der Länge k von n_1 nach n_2 existiert, auf dem alle Zwischenknoten $n \notin \{n_1, n_2\}$ weder d(v) noch r(v) oder u(v) besitzen.

VS(K) = durchschnittlicher Wert aller $VS(v, n_1, n_2) \neq 0$ für alle möglichen Kombinationen von Variablen v und Knotenpaaren n_1, n_2.

Beispiel für „Variablenspanne“:

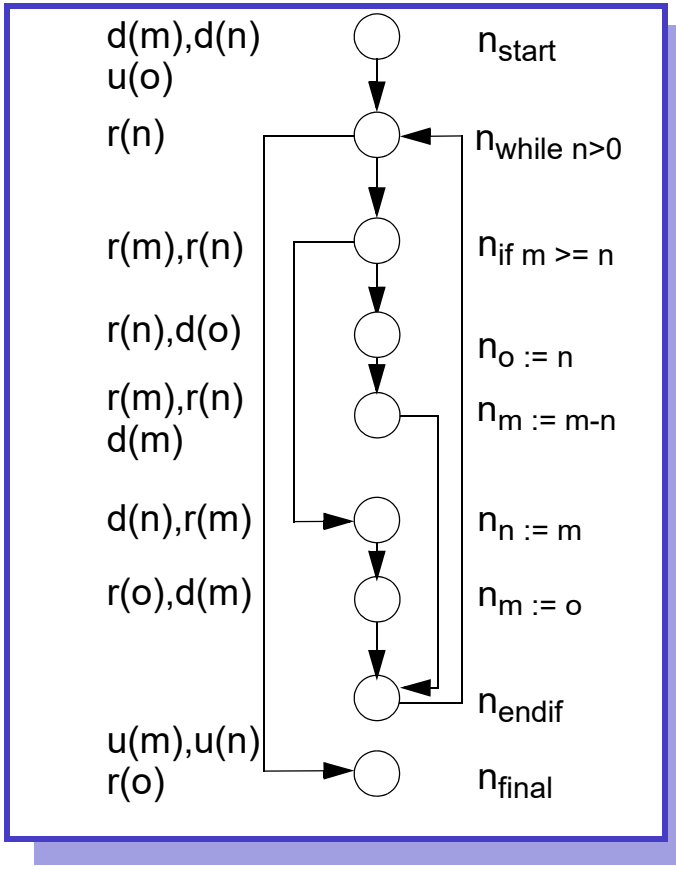
```
0:  PROCEDURE swap(INOUT x: INT; INOUT y: INT),
1:     h: INT := x
2:     x := y;
3:     y := h;
4:  END;
```

Berechnung der Metrik:

- Variablenspannen von x, die ungleich 0 sind:
  VS(x, 0, 1) = VS(x, 1, 2) = 1; VS(x, 2, 4) = 2;
- Variablenspannen von y, die ungleich 0 sind:
  VS(y, 0, 2) = 2; VS(y, 2, 3) = VS(y, 3, 4) = 1;
- Variablenspannen von h, die ungleich 0 sind:
  VS(h, 1, 3) = 2;
- $VS(swap) = ((1+1+2)_{\text{für x}} + (2+1+1)_{\text{für y}} + 2_{\text{für h}}) / 7 = 1{,}43$

Größeres Beispiel für die Berechnung von „Variablenspannen“:

```
PROCEDURE ggT(IN m, n: INTEGER; OUT o: INTEGER);          (* start *)
BEGIN
   WHILE n>0 DO
   IF m >= n  THEN
      o := n;
      m := m-n;
   ELSE
      n := m;
      m := o;
   END;(* endif *)
END ggT;(* final *)
```

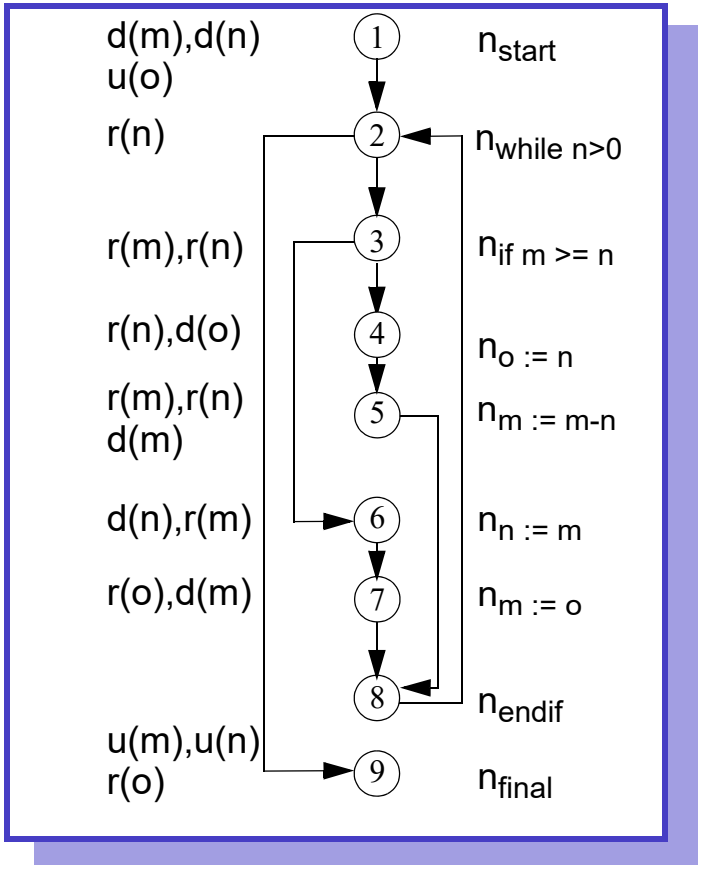


Größeres Beispiel für die Berechnung von „Variablenspannen“:

- Variablenspannen von m:
  VS(m,1,3) = VS(m,3,5) = 2;
  VS(m,3,6) = VS(m,6,7) = 1;
  VS(m,5,3) = VS(m,7,3) = 3;

- Variablenspannen von n:
  VS(n,1,2) = VS(n,2,3) = VS(n,3,4) =
  VS(n,4,5) = VS(n,3,6) = 1;
  VS(n,5,2) = 2; VS(n,6,2) = 3;

- Variablenspannen von o:
  VS(o,4,4) = 5;
  VS(o,4,7) = 6;
  VS(o,4,9) = VS(o,7,4) = 4;
  VS(o,7,7) = 5;
  VS(o,7,9) = 3;



- VS(ggT) = ((2*2 + 2*1 + 2*3) + (5*1 + 2*3) + (2*5 + 1*6 2*4 + 1*3)) / 19 = 2,63

„Live Variables“ und Variablenspanne - Zusammenfassung:

- „**Live Variables**“ einer Komponente ist durchschnittliche Anzahl lebendiger Variablen in einem Programm pro Zeile (Knoten im Kontrollflussgraphen)
- eine Variable ist von ihrer ersten Definitionsstelle (vom Startknoten aus) bis zur letzten Definitions- oder Referenzierungsstelle (vor dem Endknoten) **lebendig**
- „**Variablenspanne**“ einer Komponente ist die durchschnittliche Spanne zweier direkt aufeinander folgender definierender oder referenzierender Auftreten derselben Variable

Anmerkung:

Mit diesen beiden Metriken versucht man nicht die „Kontrollflusskomplexität“ oder einfache Größe einer Softwarekomponente, sondern die Komplexität des Datenflusses zu bewerten (wieviele Variablen muss man wie lange beim Erstellen von Programm-teilen oder beim Nachvollziehen des Programmablaufs „im Kopf behalten“).

Erste Überlegungen zu Metriken für objektorientierte Programme:

Betrachtet wird oft Kopplung von Klassen = Benutzt-Beziehungen zwischen Klassen:

- ⇨ **geringer fan-out** (wenige auslaufende Benutzt-Beziehungen) ist positiv, da sich dann eine Klasse auf wenige andere Klassen abstützt
- ⇨ **hoher fan-in** (viele einlaufende Benutzt-Beziehungen) ist positiv, da dann eine Klasse von vielen Klassen (wieder-)verwendet wird
- ⇨ beides kann nicht maximiert werden, da über alle Klassen hinweg gilt: **Summe fan-in = Summe fan-out**

Eine Klasse A benutzt eine Klasse B, wenn:

- ⇨ in A ein Verweis auf Objekt der Klasse B verwendet wird
- ⇨ in A eine Operation einen Parameter der Klasse B verwendet
- ⇨ in A eine Operation der Klasse B aufgerufen wird

Achtung: ähnlich kann man Kopplung von Modulen (Pakete) bzw. Kopplung von Methoden studieren.

Metriken für OO-Programme - weitere Überlegungen:

Gesucht werden Metriken, die neben der Kopplung von Klassen folgende Aspekte in Maßzahlen zusammenfassen:

- die Methoden einer Klasse sollten **enge Bindung (high cohesion)** besitzen, also einem ähnlichen Zweck dienen (wie misst man das?)
- die Klassen einer Vererbungshierarchie sollten ebenfalls **enge Bindung** besitzen
- die in einem Modul bzw. Paket zusammengefassten Klassen oder die in einer Klasse zusammengefassten Methoden sollten **enge Bindung** besitzen
- Klassen in verschiedenen Modulen bzw. Paketen sollte **lose gekoppelt** sein (wie misst man das?) **(loose coupling)**
- Klassen und Module bzw. Pakete sollten ein Implementierungsgeheimnis verbergen **(data abstraction, encapsulation)**
- ...

Bindungsmetriken - LOCOM1 (Low Cohesion Metric):

Die Bindung der Methoden einer Klasse wird untersucht. Methoden sind eng aneinander gebunden, wenn sie auf viele gemeinsame Attribute/Felder zugreifen:

- ⇨ P := Anzahl der Paare von Methoden ohne gemeinsame Attributzugriffe
- ⇨ Q := Anzahl der Paare von Methoden mit gemeinsamen Attributzugriffen
- ⇨ **LOCOM1** := if P > Q then P-Q else 0

Gewünscht wird Wert von LOCOM1 nahe bei 0.

Beispiel:

getName und setName der Klasse Person greift auf Attribut name zu
getAddr und setAddr der Klasse Person greift auf Attribut addr zu
compare greift auf Attribute name und addr zu

Es gilt dann:

es gibt 10 verschiedene Paarungen und P = 4 und Q = 6, also LOCOM1 = 0

Bindungsmetriken - LOCOM2 (Low Cohesion Metric):

Die Bindung der Methoden einer Klasse wird untersucht. Methoden sind eng aneinander gebunden, wenn sie auf viele gemeinsame Attribute/Felder zugreifen:

- m := Anzahl Methoden m_i einer Klasse
  $m(a_i)$:= Anzahl Methoden, die auf Attribut a_i zugreifen
- n := Anzahl Attribute a_i einer Klasse
- **LOCOM2** := $1 - (m(a_1) + ... + m(a_n)) / (m*n)$

Gewünscht wird kleiner Wert von LOCOM2 (z.B. kleiner 0,3 = 30%).

Beispiel:

getName und setName der Klasse Person greift auf Feld name zu
getAddr und setAddr der Klasse Person greift auf Feld addr zu
compare greift auf Felder name und addr zu

Es gilt dann:

Weitere Metriken - Kopplung von Klassen - 1:

- **Afferent Coupling (Ca/AC)**:
  Die Anzahl der Klassen ausserhalb eines betrachteten Teilsystems (Kategorie), die von den Klassen innerhalb des Teilsystems abhängen

- **Efferent Coupling (Ce/EC)**:
  Die Anzahl der Klassen innerhalb eines betrachteten Teilsystems (Kategorie), die von Klassen ausserhalb des betrachteten Teilsystems abhängen

- **Instabilität (I)**: $I = Ce / (Ce+Ca)$
  - ⇨ I hat einen Wert zwischen 0 und 1, falls nicht Ce + Ca = 0 gilt
    mit 0 = max. stabil u. 1 = max. unstabil
  - ⇨ der Wert 1 besagt, dass Ca = 0 ist; das betrachtete Teilsystem exportiert also nichts nach außen (keine Klassen und deren Methoden)
  - ⇨ der Wert 0 besagt, dass Ce = 0 ist; das betrachtete Teilsystem importiert also nichts von außen (keine Klassen und deren Methoden)
  - ⇨ Der „undefinierte“ Fall Ca = 0 und Ce = 0 kann nur auf ein (sinnloses) isoliertes Teilsystem zutreffen, das weder importiert noch exportiert

Weitere Metriken - Kopplung von Klassen - 2:

- **Coupling** als **Change Dependency between Classes (CDBC)**:
  CDBC zwischen Client Class CC und Server Class SC :=
  - ⇨ n falls SC Oberklasse von CC ist (n = Anzahl Methoden in CC)
  - ⇨ n falls CC ein Attribut des Typs SC hat
  - ⇨ j falls SC in j Methoden von CC benutzt wird
    (als Typ lokaler Variable, Parameter oder Methodenaufruf von SC)

  CDBC bewertet Aufwand, der mit Überarbeitung von CC wegen Änderung in SC verbunden sein könnte (Anzahl potentiell zu überarbeitender Methoden in CC).

- **Encapsulation** als **Attribute Hiding Factor (AHF)**:
  - ⇨ Summe der Unsichtbarkeiten aller Attribute in allen Klassen geteilt durch die Anzahl aller Attribute
  - ⇨ Unsichtbarkeit eines Attributs := Prozentzahl der Klassen, für die das Attribut nicht sichtbar ist (abgesehen von eigener Klasse)

  Sind alle Attribute als „private“ definiert, dann ist AHF = 1.

Weitere Metriken:

- **Tiefe von Vererbungshierarchien**:
  zu tiefe Hierarchien werden unübersichtlich; man weiss nicht mehr, was man erbt
- **Breite von Vererbungshierarchien**:
  zu breite Vererbungshierarchien deuten auf Fehlen von zusammenfassenden Klassen hin
- **Anzahl Redefinitionen in einer Klassenhierarchie**:
  je mehr desto gefährlicher
- **Anzahl Zugriffe auf geerbte Attribute**:
  sind ebenfalls gefährlich, da beim Ändern von Attributen oder Attributzugriffen in Oberklasse die Zugriffen in den Unterklassen oft vergessen werden
- **Halstead-Metriken**: ...
- ...

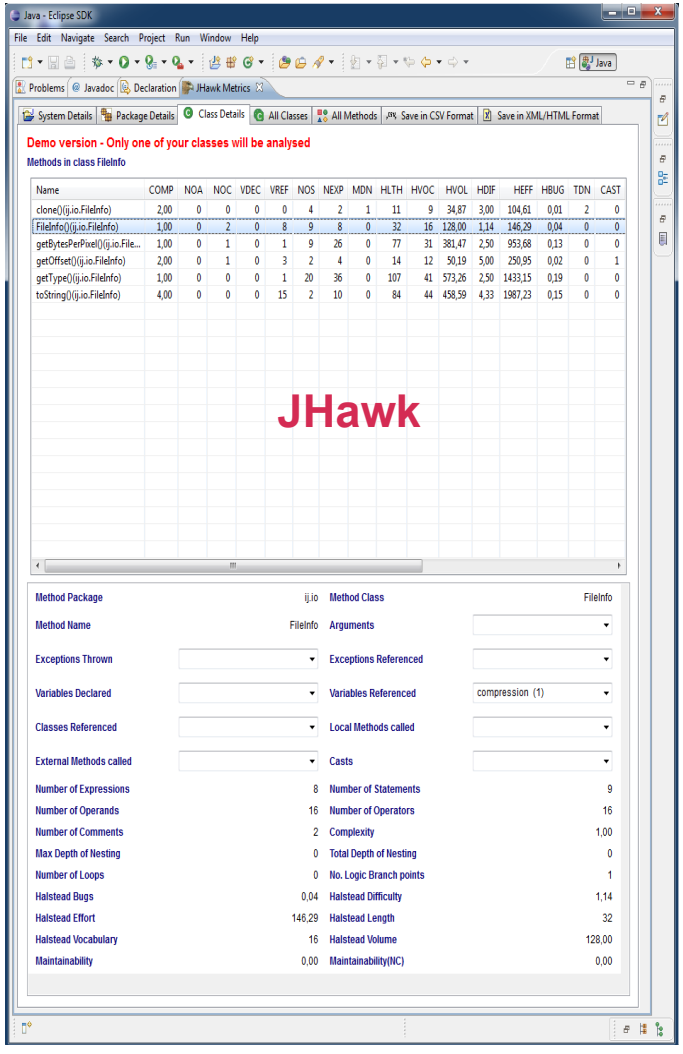
Komplexitätsmaße:

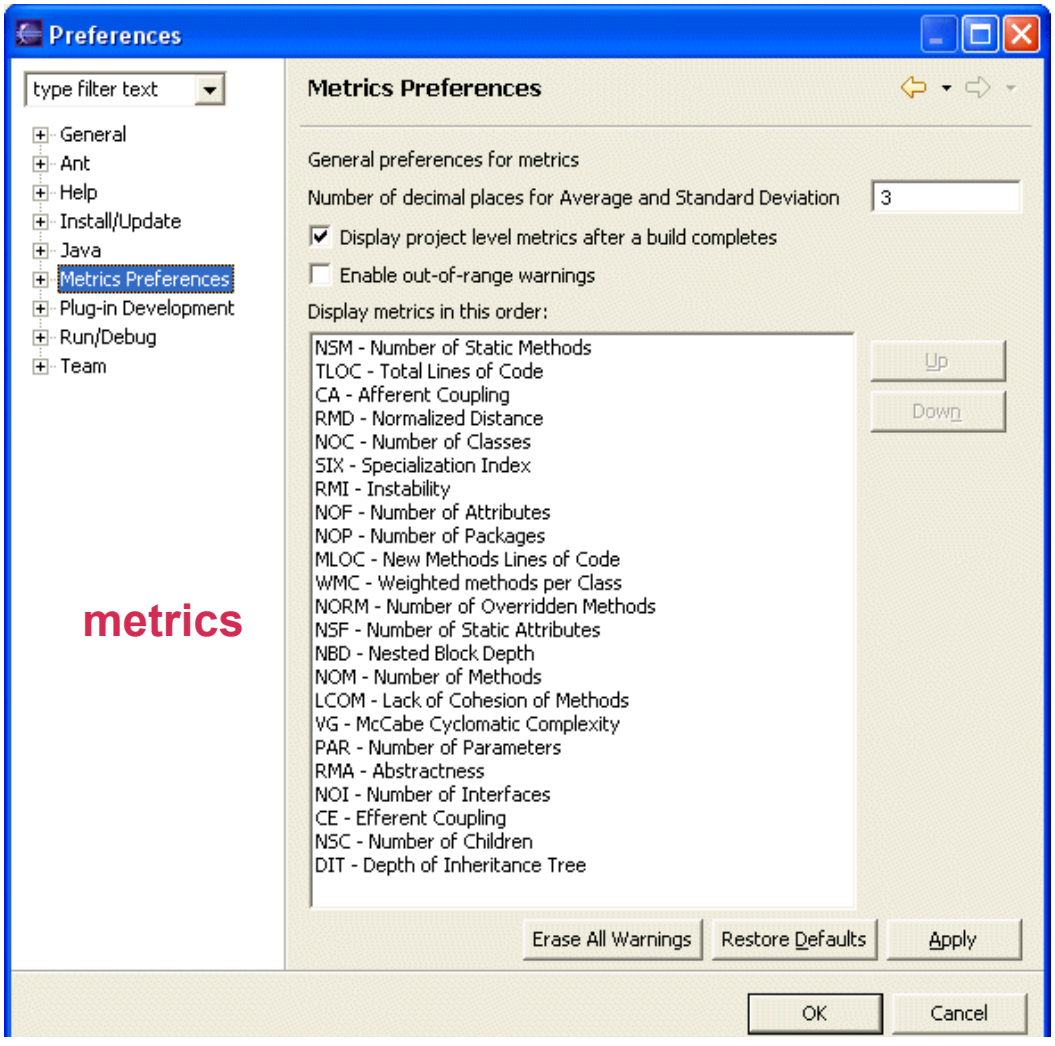
- **Response For Class (RFC)**:
  die Anzahl der in der Klasse deklarierten Methoden + die Anzahl der geerbten Methoden + die Anzahl sichtbarer Methoden anderer Klassen
  (Alle Methoden, die aufgerufen werden können? Sehr schwammig definiert!!!)
- **Weighted Methods Per Class (WMPC1)**:
  die Summe der zyklomatischen Zahlen ZZ aller Methoden der Klasse (ohne geerbte Methoden)
- **Number of Remote Methods (NORM)**:
  die Anzahl der in einer Klasse gerufenen Methoden „fremder“ Klassen (also nicht die Klasse selbst oder eine ihrer Oberklassen)
- **Attribute Complexity (AC)**:
  die gewichtete Summe der Attribute einer Klasse wird gebildet; Gewichte werden gemäß Typen/Klassen der Attribute vergeben.
- ...

Beispiele für „Mess“-Werkzeuge:

- JHawk (http://www.virtualmachinery.com/jhawkprod.htm) für Java:
  - ⇨ unterstützt viele Zähl-Metriken (Anzahl von Parametern, ...)
  - ⇨ unterstützt die klassischen zweifelhaften Metriken (Halstead, ...)
  - ⇨ kommerzielles Produkt
- metrics (http://metrics.sourceforge.net/) für Java:
  - ⇨ „Open Source“-Software
  - ⇨ unterstützt deutlich weniger Metriken als JHawk
  - ⇨ aber ebenfalls Ca, Ce, Tiefe von Vererbungshierarchien, ...
- Together (scheint von Borland nicht mehr unterstützt zu werden)
  - ⇨ UML-Werkzeug mit enger Modell-Code-Integration
  - ⇨ unterstützt(e) viele Metriken für C++ und Java
  - ⇨ fast alle hier vorgestellten komplexeren OO-Metriken stammen aus der Together-Dokumentation

Screendumps von JHawk und metrics:





3.6 Zusammenfassung

Die **Visualisierung von Software** ist sowohl beim „Forward Engineering“ für den Entwurf neuer Programmarchitekturen als auch beim „Reverse Engineering“ für das Studium von „Legacy Software“ mit unbekannter Programmstruktur sehr hilfreich.

Werkzeugunterstützte **statische Analyseverfahren** helfen frühzeitig bei der Identifikation kritischer Programmstellen. Es sollten folgende Analyseverfahren immer eingesetzt werden:

- ⇨ **Stilanalyse** (Überprüfung vereinbarter Programmierkonventionen)
- ⇨ **„dead code“-Analyse** (oft in Compiler eingebaut): nie verwendete Methoden, Variablen, Parameter, ... (wurde bisher nicht angesprochen)
- ⇨ **Datenflussanalyse** (wenn Werkzeug verfügbar)

Weitere Analyseverfahren und vor allem **Metriken** sollten in großen Projekten zumindest versuchsweise eingesetzt werden.

Vorgehensweise beim Einsatz von Maßen aus [Li02]:

1. **Fragen zur Ausgangssituation**:
   - ⇨ In welcher Phase (Aktivitätsbereich) des Softwareentwicklungsprozesses soll eine Verbesserung eingeführt werden (z.B. Design, Codierung, ...)?
   - ⇨ Was soll damit erreicht werden bzw. welche Art von Fehler soll reduziert werden (z.B. Reduktion C++ Codierungsfehler)?
   - ⇨ Welche Methode soll eingesetzt werden (z.B. OO-Metriken)?
   - ⇨ Welche Technik/Werkzeug soll eingesetzt werden
2. **Bewertung des aktuellen Standes** des Entwicklungsprozesses:
   - ⇨ Welche Kosten u. welcher Aufwand entstehen in welcher Phase?
   - ⇨ Wie ist die Qualität der Ergebnisse jeder Phase?
   - ⇨ In welcher Phase entsteht welcher Anteil an Fehlern und welcher Teil der Fehlerbeseitigungskosten?

Vorgehensweise - Fortsetzung:

3. Mittel zur Bestimmung des aktuellen Standes, **zu messende Aspekte**:
   - ⇨ Kosten- und Zeitverfolgung beim Entwicklungsprozess
   - ⇨ Definition von Qualitätsmaßen für Produkt pro Phase
   - ⇨ Erhebung von Fehlerstatistiken
4. **Analyse der Ergebnisse** und Erarbeitung von Verbesserungsvorschlägen:
   - ⇨ Auswertung der Maße
   - ⇨ Definition von Zielen auf Basis der Messwerte
   - ⇨ Entscheidung für Verbesserung in bestimmten Phasen
   - ⇨ Auswahl geeigneter Methoden und Werkzeuge
   - ⇨ Einführung der Methoden und Werkzeuge in Entwicklungsprozess
5. **Bewertung** der durchgeführten Änderungen:
   - ⇨ Kontinuierliche Weiterauswertung der Maße
   - ⇨ erneute Analyse nach „Abklingen von Einschwingvorgängen“

Abstraktes Fallbeispiel aus [Li02]:

1. Angestrebt wird die deutliche Reduktion der Fehleranzahl mit dem Ziel den Kunden in Zukunft zuverlässigere Produkte zur Verfügung zu stellen
2. Minimiert wird dafür das **Zuverlässigkeitsmaß MTBF** = „Mean Time Between Failure“ (mittlere verstrichene Zeit zwischen zwei Fehlern):
   - ⇨ innerhalb von 343 Tagen wurden 17 (schwerwiegende) Fehler gemeldet
   - ⇨ MTBF = 20,2 - alle 20,2 Tage tritt durchschnittlich ein Fehler auf
3. Die gemeldeten Fehler lassen sich wie folgt den Phasen des Entwicklungs-prozesses zuordnen und der Aufwand für ihre Behebung ist wie folgt
   - ⇨ 5 Fehler aus der Anforderungsdefinitionsphase mit 27 PT durchschnittlichem Korrekturaufwand (PT = Personentage)
   - ⇨ 3 Fehler aus der Entwurfsphase mit 5,7 PT Korrekturaufwand
   - ⇨ 10 Fehler aus der Implementierungsphase mit 0,6 PT Korrekturaufwand

Abstraktes Fallbeispiel - Fortsetzung :

4. Ziele zur Verbesserung des Entwicklungsprozesses und Softwareprodukts:
   - ⇨ Reduktion der Fehleranzahl in der Definitionsphase zur deutlichen Aufwandsreduktion
   - ⇨ Reduktion der Fehleranzahl in der Implementierungsphase zur deutlichen Verbesserung der Qualität der ausgelieferten Software

5. Auswahl von Verbesserungsmaßnahmen in der Anforderungsdefinitionsphase:
   - ⇨ falls ungenaue Spezifikation der Anforderungen des Kunden, dann Einsatz von (semi-)formalen Spezifikationstechniken
   - ⇨ falls genaue Spezifikation falscher Anforderungen des Kunden, dann Einsatz von Rapid-Prototyping-Vorgehensweise

6. Auswahl von Verbesserungsmaßnahmen in der Implementierungsphase
   - ⇨ hätte Datenflussanalyse die Fehler gefunden
   - ⇨ hätte Metrix X fehlerhaften Code als „qualitativ zweifelhaft“ erkannt
   - ⇨ hätten systematischere Testverfahren die Fehler aufgedeckt

3.7 Zusätzliche Literatur

[Ka00] St. Kan: Metrics and Models in Software Quality Engineering, Addison-Wesley, 2nd Ed. (2003), 528 Seiten

[SD98] J. Stasko, J. Domingue, M. Brown et al.: *Software Visualization*, MIT Press (1998), 562 Seiten

[SG96] M. Shaw, D. Garlan: *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, (1996)

[Tha00] G. Thaller: *Software-Metriken einsetzen, bewerten, messen*, VT Verlag Technik Berlin, 2. Auflage (2000), 208 Seiten