

# *Programmverstehen*

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken



# *Software Reengineering*

An der Universität lernen Sie in der Regel das Entwickeln eines Programms ohne Vorgaben – „from scratch“ oder „auf der grünen Wiese“.

Viel realistischer ist es jedoch, daß Sie *alte Software* vorfinden und warten – ergänzen, portieren oder erweitern.

*Der Umgang mit Software-Altlasten ist das Problem der Softwaretechnik!*







# *Software Reengineering (2)*



- Wartungskosten: nach wie vor 50–75% der Gesamtkosten







# *Software Reengineering (2)*



- Wartungskosten: nach wie vor 50–75% der Gesamtkosten
- 80% der kommerziellen Anwendungssoftware ist in COBOL geschrieben







# *Software Reengineering (2)*



- Wartungskosten: nach wie vor 50–75% der Gesamtkosten
- 80% der kommerziellen Anwendungssoftware ist in COBOL geschrieben – 3/4 davon monolithisch und $>$ 20 Jahre







# *Software Reengineering (2)*



- Wartungskosten: nach wie vor 50–75% der Gesamtkosten
- 80% der kommerziellen Anwendungssoftware ist in COBOL geschrieben – 3/4 davon monolithisch und $>$ 20 Jahre
- Ein typischer Anwender in den USA hat 2200 Programme mit insgesamt 1,15 Mio LOC







# *Software Reengineering (2)*



- Wartungskosten: nach wie vor 50–75% der Gesamtkosten
- 80% der kommerziellen Anwendungssoftware ist in COBOL geschrieben – 3/4 davon monolithisch und $> 20$ Jahre
- Ein typischer Anwender in den USA hat 2200 Programme mit insgesamt 1,15 Mio LOC
- Deutsche COBOL-Programme
  - sind zu 80% monolithisch
  - sind zu 77% unstrukturiert
  - enthalten zu 93% redundant gehaltene Daten, die aus Unwissenheit oder Unsicherheit nicht gelöscht werden







# *Software Reengineering (2)*



- Wartungskosten: nach wie vor 50–75% der Gesamtkosten
- 80% der kommerziellen Anwendungssoftware ist in COBOL geschrieben – 3/4 davon monolithisch und $> 20$ Jahre
- Ein typischer Anwender in den USA hat 2200 Programme mit insgesamt 1,15 Mio LOC
- Deutsche COBOL-Programme
  - sind zu 80% monolithisch
  - sind zu 77% unstrukturiert
  - enthalten zu 93% redundant gehaltene Daten, die aus Unwissenheit oder Unsicherheit nicht gelöscht werden
- Jede dritte Anweisung verwendet eine numerische Konstante oder ein Literal (d.h. hartcodierte Datenwerte)







# *Software Reengineering (3)*



- Ein Wartungsprogrammierer ist für 32.000 Codezeilen verantwortlich







# *Software Reengineering (3)*



- Ein Wartungsprogrammierer ist für 32.000 Codezeilen verantwortlich
- Ein Wartungsprogrammierer benötigt
  - 47% seiner Zeit für Programmanalyse
  - 15% für Programmierung
  - 28% für den Test
  - 9% für die Dokumentation







# *Software Reengineering (3)*



- Ein Wartungsprogrammierer ist für 32.000 Codezeilen verantwortlich
- Ein Wartungsprogrammierer benötigt
  - 47% seiner Zeit für Programmanalyse
  - 15% für Programmierung
  - 28% für den Test
  - 9% für die Dokumentation
- *Entropiezuwachs:* Die Komplexität einer Prozedur steigt
  - nach einer Fehlerkorrektur um durchschnittlich 4%
  - nach einer Änderung um 17% und
  - nach einer Erweiterung um 26%







# *Software Reengineering (3)*



- Ein Wartungsprogrammierer ist für 32.000 Codezeilen verantwortlich
- Ein Wartungsprogrammierer benötigt
  - 47% seiner Zeit für Programmanalyse
  - 15% für Programmierung
  - 28% für den Test
  - 9% für die Dokumentation
- *Entropiezuwachs:* Die Komplexität einer Prozedur steigt
  - nach einer Fehlerkorrektur um durchschnittlich 4%
  - nach einer Änderung um 17% und
  - nach einer Erweiterung um 26%
- Bei der *US Air Force* kostet die Änderung einer einzelnen Zeile 2500–3000$ (1990)









# *Einige Begriffe*

**Reverse Engineering** Extraktion und Repräsentation von Informationen aus einem Software-System

- in einer anderen Form oder
- auf höherem Abstraktionsniveau









# *Einige Begriffe*

**Reverse Engineering** Extraktion und Repräsentation von Informationen aus einem Software-System

- in einer anderen Form oder
- auf höherem Abstraktionsniveau

**Restrukturierung** Transformation zwischen Repräsentationsformalismen ohne Änderung der Funktionalität









# *Einige Begriffe*

**Reverse Engineering** Extraktion und Repräsentation von Informationen aus einem Software-System

- in einer anderen Form oder
- auf höherem Abstraktionsniveau

**Restrukturierung** Transformation zwischen Repräsentationsformalismen ohne Änderung der Funktionalität

**Reengineering** Alle Aktivitäten, die nach Inbetriebnahme eines Programmsystems

- das Verständnis von Software erhöhen oder
- Wartbarkeit, Wiederverwendbarkeit oder Weiterentwickelbarkeit verbessern oder erst ermöglichen



# *Einige Begriffe*

**Reverse Engineering** Extraktion und Repräsentation von Informationen aus einem Software-System

- in einer anderen Form oder
- auf höherem Abstraktionsniveau

**Restrukturierung** Transformation zwischen Repräsentationsformalismen ohne Änderung der Funktionalität

**Reengineering** Alle Aktivitäten, die nach Inbetriebnahme eines Programmsystems

- das Verständnis von Software erhöhen oder
- Wartbarkeit, Wiederverwendbarkeit oder Weiterentwickelbarkeit verbessern oder erst ermöglichen

*Reengineering = Reverse Engineering + Restrukturierung*









# *Programmverstehen*

**Programmverstehen** ist der wichtigste (und älteste) Bestandteil des *Reverse Engineering*

Typischerweise Konstruktion *alternativer Sichten,* die helfen, das System besser zu verstehen:

- Erzeugen eines Ablaufdiagramms / Struktogramms
- Querverweis-Tabellen (*cross references*)
- Anreichern von Quellcode um Metrikwerte
- UML-Diagramme aus OO-Quellcode
- Software-Visualisierung

Wir betrachten nun einige typische Werkzeuge.

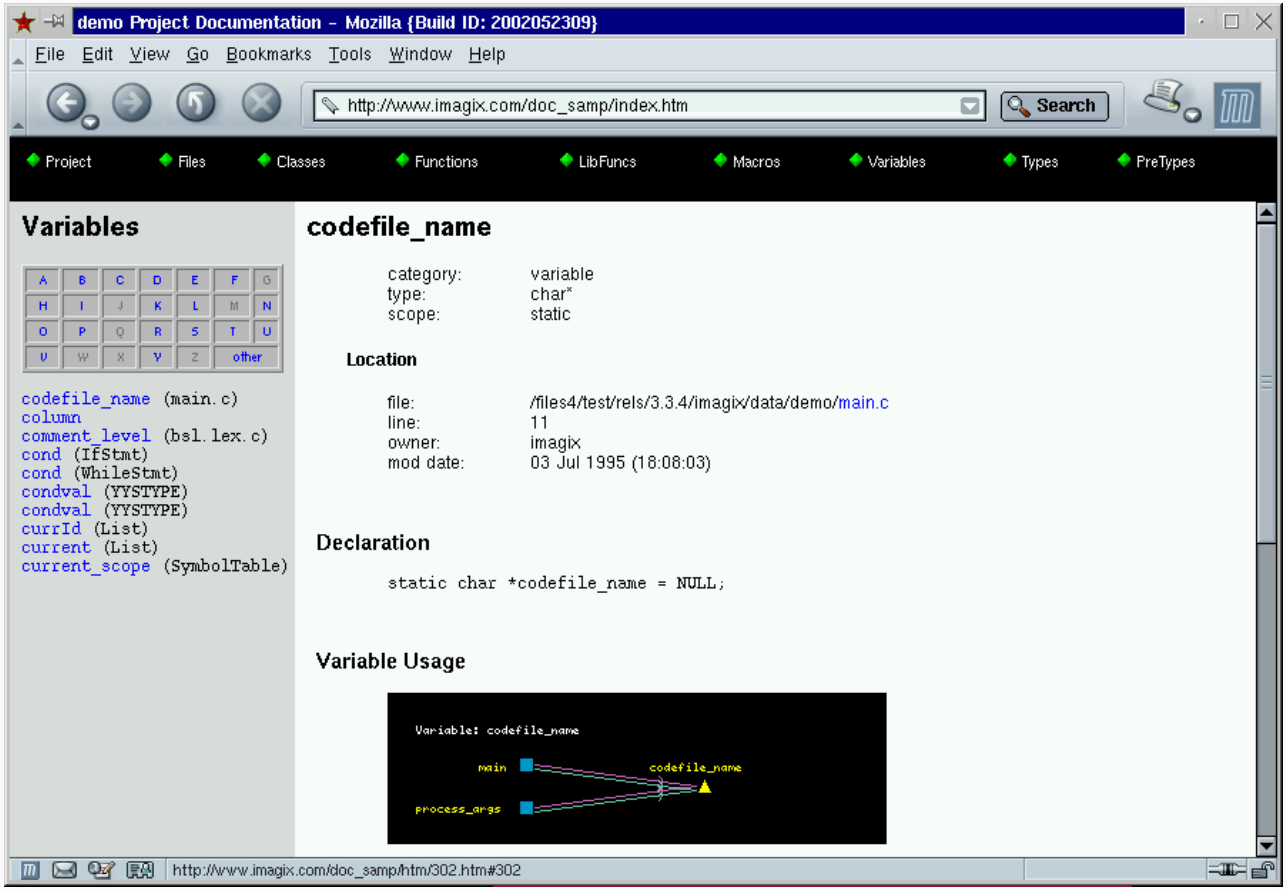






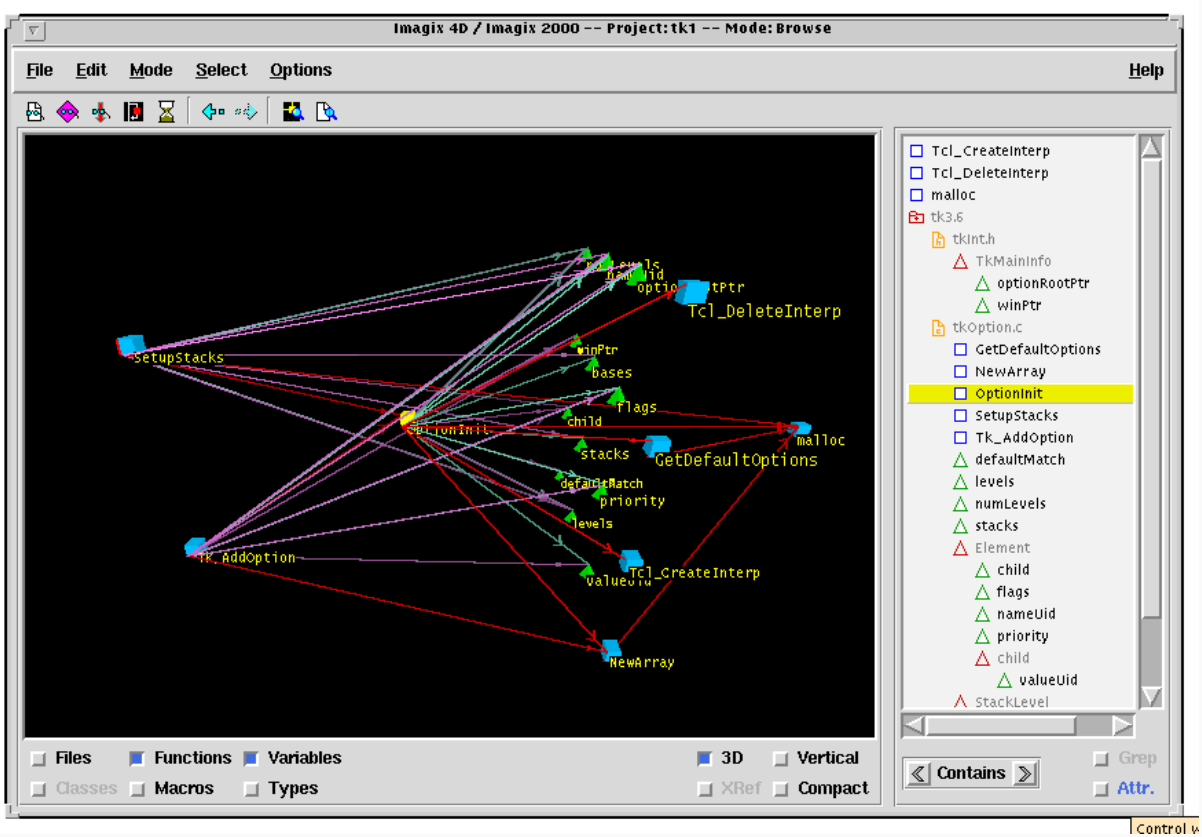
# Imagix: Dokumentation





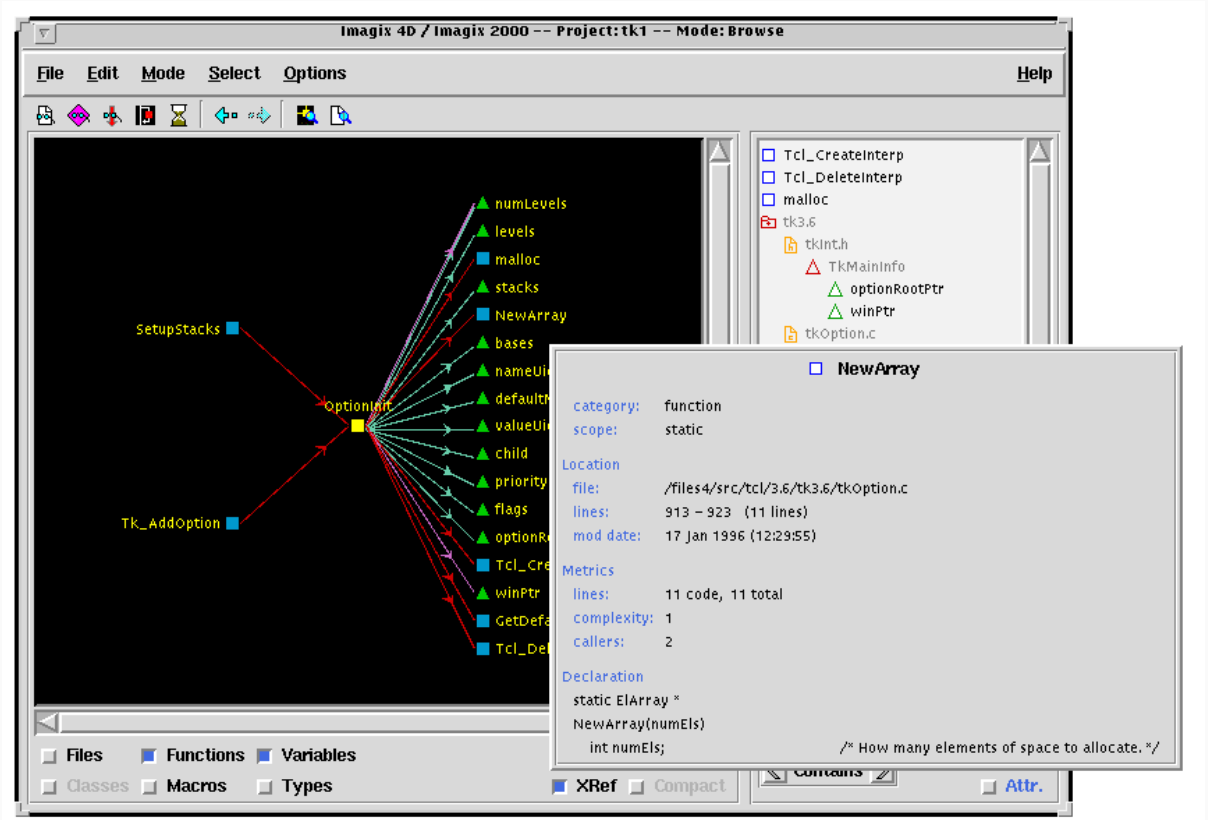
# Imagix: Funktionen und Variablen





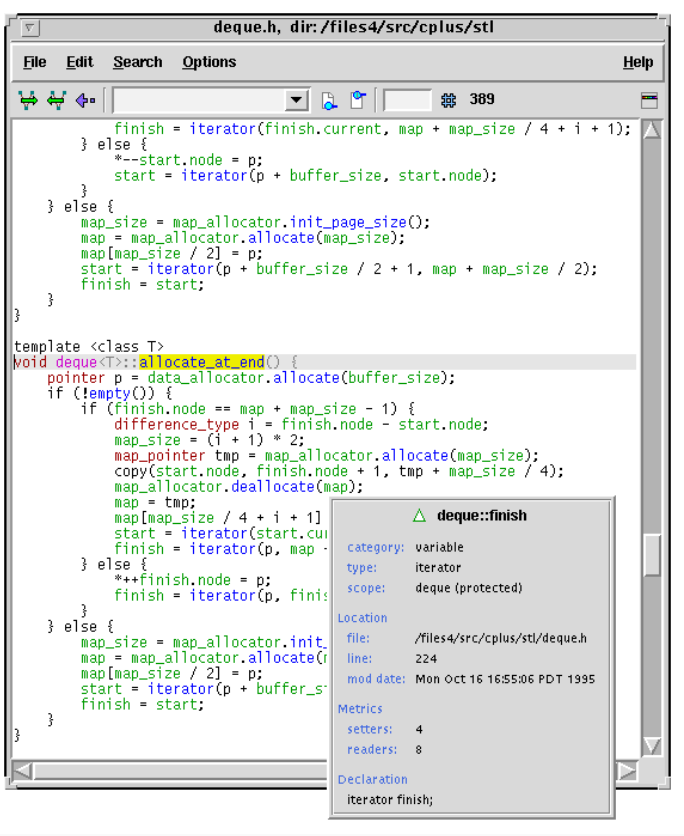
# Imagix: Benutzung



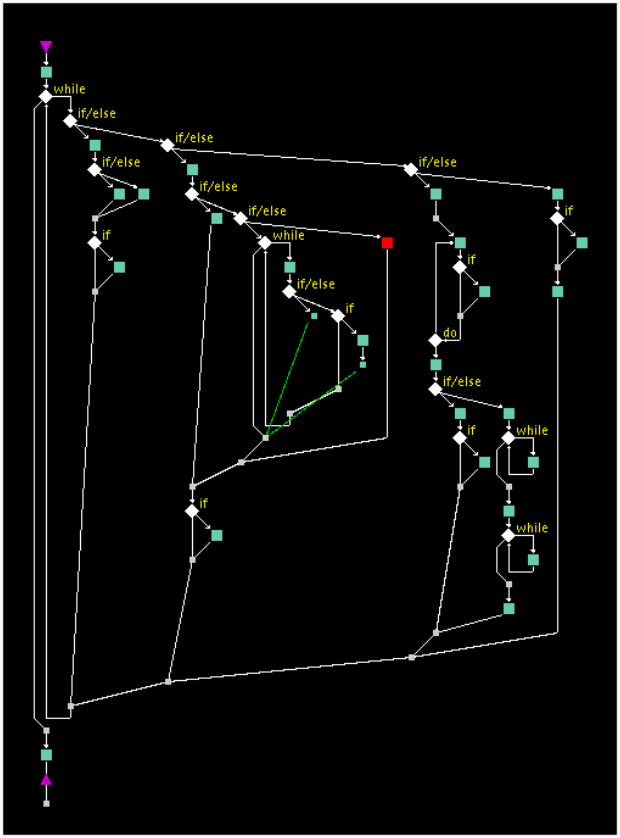


# Imagix: Pretty Printing

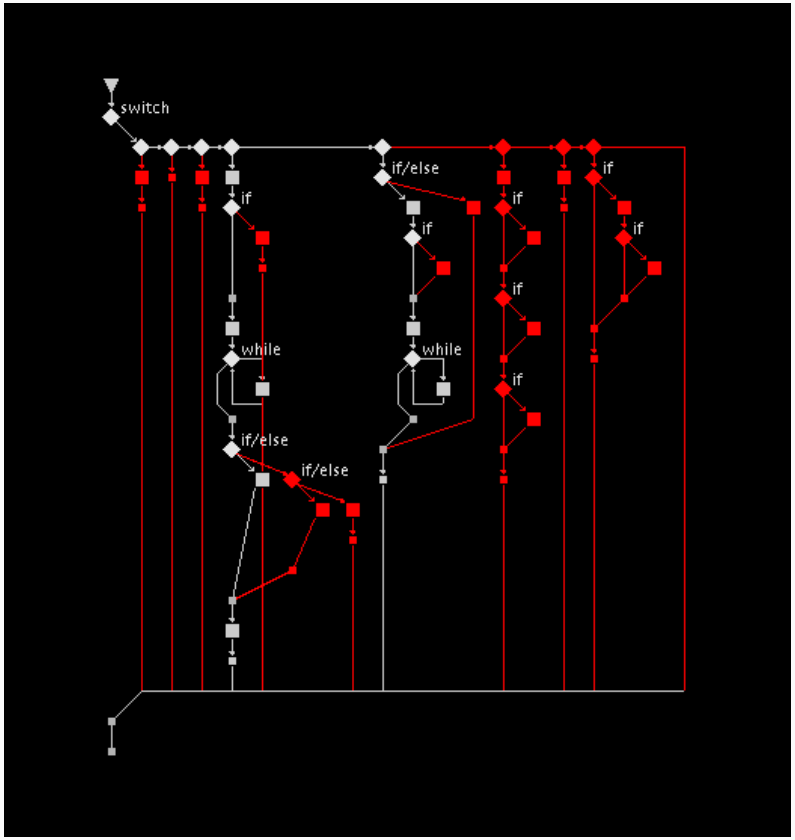




# Imagix: Flussdiagramm

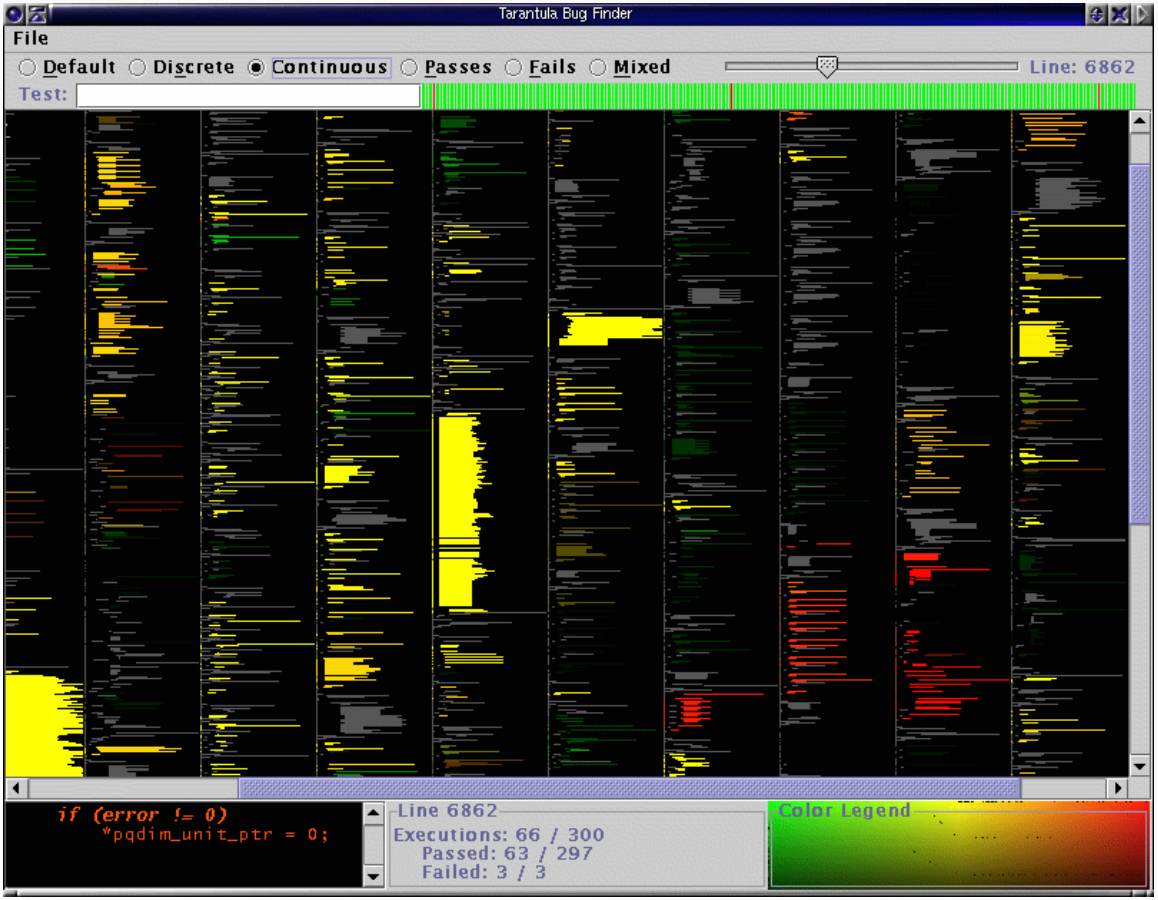


# Imagix: Abdeckung



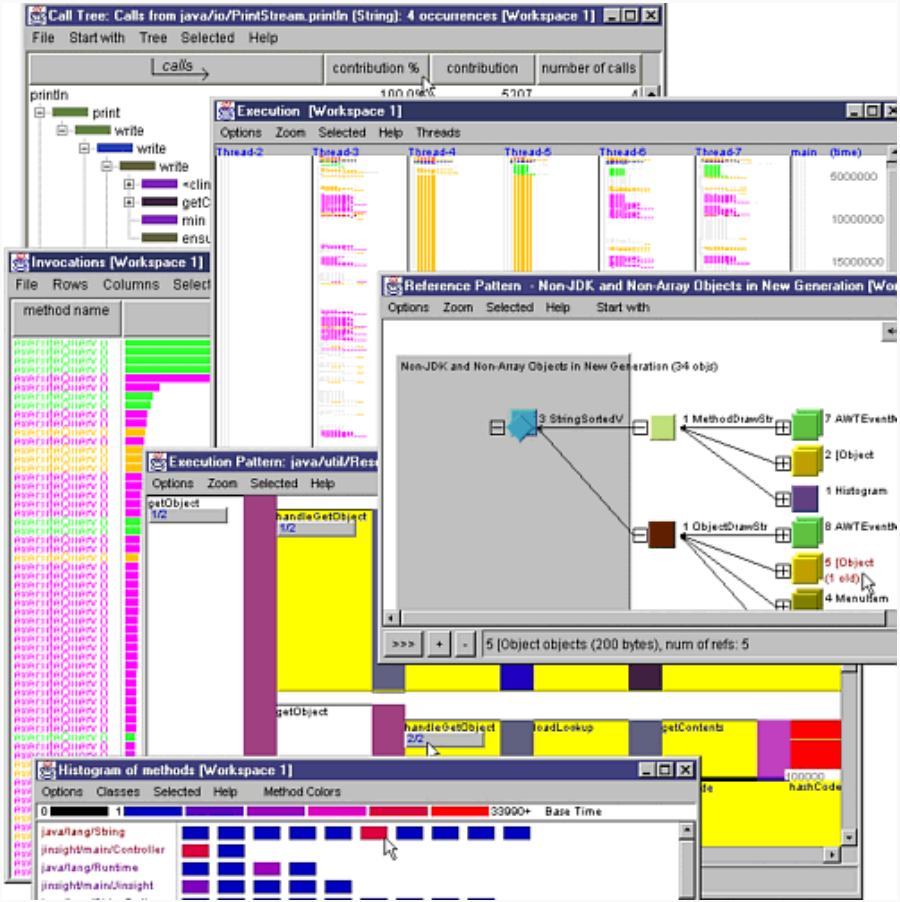
# Tarantula: Abdeckung vergleichen





# Jinsight: Programmläufe analysieren







# *Fazit: Visualisierung*

Die Darstellung *alternativer Sichten* kann sehr hilfreich beim Programmverstehen sein – insbesondere,

- wenn mit Navigation gekoppelt
- wenn dynamische Information dargestellt wird, die sich sonst der Untersuchung entzieht

Weitere Ansätze:
Vorlesung *Software-Visualisierung* (WS 2002/03)







# *Programmanalyse*



Jede Visualisierung hat ihre Grenzen in der Aufnahmefähigkeit des Menschen

Deshalb wichtig: *Fokussierung* auf Teilaspekte des Programms

Das ist die Aufgabe der *Programmanalyse*







# *Programmanalyse (2)*



Man unterscheidet

**Statische Programmanalyse** bestimmt Eigenschaften eines Programms (= über alle möglichen Läufe)

**Dynamische Programmanalyse** bestimmt Eigenschaften eines Programmlaufs (oder mehrerer Programmläufe)

Wichtiges Hilfsmittel im *Reverse Engineering* (aber auch für Prüfung der Korrektheits und der Sicherheit)









# *Program Slicing*

Program Slicing hat das Ziel, *Anweisungen zu isolieren,* die für bestimmte Programmzustände relevant sind:

- *Abhängigkeiten* zwischen Ein- und Ausgaben erkennen
- Anweisungen bestimmen, die andere *beeinflussen* können
- Auswirkungen von *Code-Änderungen* eingrenzen

Zentrale Technik der *Fokussierung* auf relevante Teilaspekte des Programms







# Beispiel: Summe und Produkt berechnen



```
int main() {
  int a, b, sum, mul;
  sum = 0;
  mul = 1;
  a = read();
  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }
  write(sum);
  write(mul);
```







# *Beispiel: Summe und Produkt berechnen*



```
int main() {
  int a, b, sum, mul;
  sum = 0;
  mul = 1;
  a = read();
  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }
  write(sum);
  write(mul); ⇐ Was kann mul hier beeinflussen?
}
```







# *Slicing*



*Program Slicing* versucht, für eine Berechnung irrelevante Anweisungen eines Programms zu entfernen oder relevante Anweisungen zu markieren.

Ergebnis: Eine Teilmenge des Programms – ein *Slice*.









# *Slicing*

*Program Slicing* versucht, für eine Berechnung irrelevante Anweisungen eines Programms zu entfernen oder relevante Anweisungen zu markieren.

Ergebnis: Eine Teilmenge des Programms – ein *Slice*.

Grundlage: Das *Slicing-Kriterium* $(v, n)$ spezifiziert den Slice für eine Variable $v$ bei einer Anweisung $n$.

Ein Slice wird berechnet, indem möglichst viele Anweisungen des Programms *gelöscht* werden, ohne daß sich die Berechnung des Wertes für die Variable $v$ bei der Anweisung $n$ ändert.

# *Backward Slicing*



*Backward-Slice*: Anweisungen, die den Wert einer Variablen an einer Stelle im Programm beeinflussen

```
int main() {
  int a, b, sum, mul;
  sum = 0;
  mul = 1;
  a = read();
  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }
  write(sum);
  write(mul);
}
```

Programm

```
int main() {
  int a, b, sum, mul;

  mul = 1;
  a = read();
  b = read();
  while (a <= b) {

    mul = mul * a;
    a = a + 1;
  }

  write(mul);
}
```

Backward-Slice für (`mul`, 13)

Man betrachtet *rückwärts* Wirkungen früherer Anweisungen









# *Forward Slicing*

*Forward-Slice:* alle Anweisungen, die durch eine Änderung der Variablen $v$ bei der Anweisung $n$ betroffen sind

```
int main() {
  int a, b, sum, mul;
  sum = 0;
  mul = 1;
  a = read();
  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }
  write(sum);
  write(mul);
}
```

Programm

```
int main() {
  int a, b, sum, mul;


  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }
  write(sum);
  write(mul);
}
```

Forward-Slice für (b, 6)

Man betrachtet *vorwärts* die Auswirkungen im Programm



# *Programm-Abhängigkeits-Graph*

Grundlage des Program Slicing sind *Programm-Abhängigkeits-Graphen* (engl. *program dependency graph*, PDG)

PDGs sind ähnlich aufgebaut wie Kontrollflußgraphen; ihre Kanten geben jedoch *Kontroll-* und *Datenabhängigkeiten* zwischen zwei Anweisungen $A$ und $B$ an:

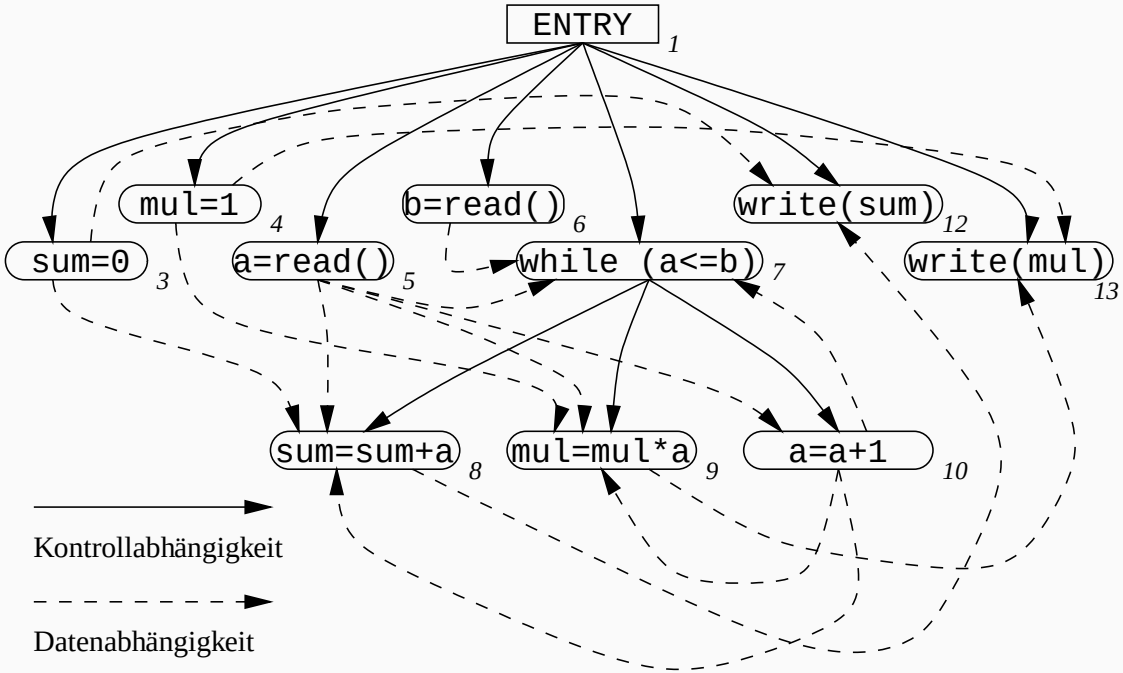
- Zwischen $A$ und $B$ besteht eine *Kontrollabhängigkeit*, wenn $A$ beeinflußt, ob oder wie oft $B$ ausgeführt wird.
- Zwischen $A$ und $B$ besteht eine *Datenabhängigkeit*, wenn in $A$ einer Variablen ein Wert zugewiesen und dieser Wert in $B$ durch Auswertung dieser Variablen benutzt wird.







# Programm-Abhängigkeits-Graph (2)





# *Slicing anhand des PDG*

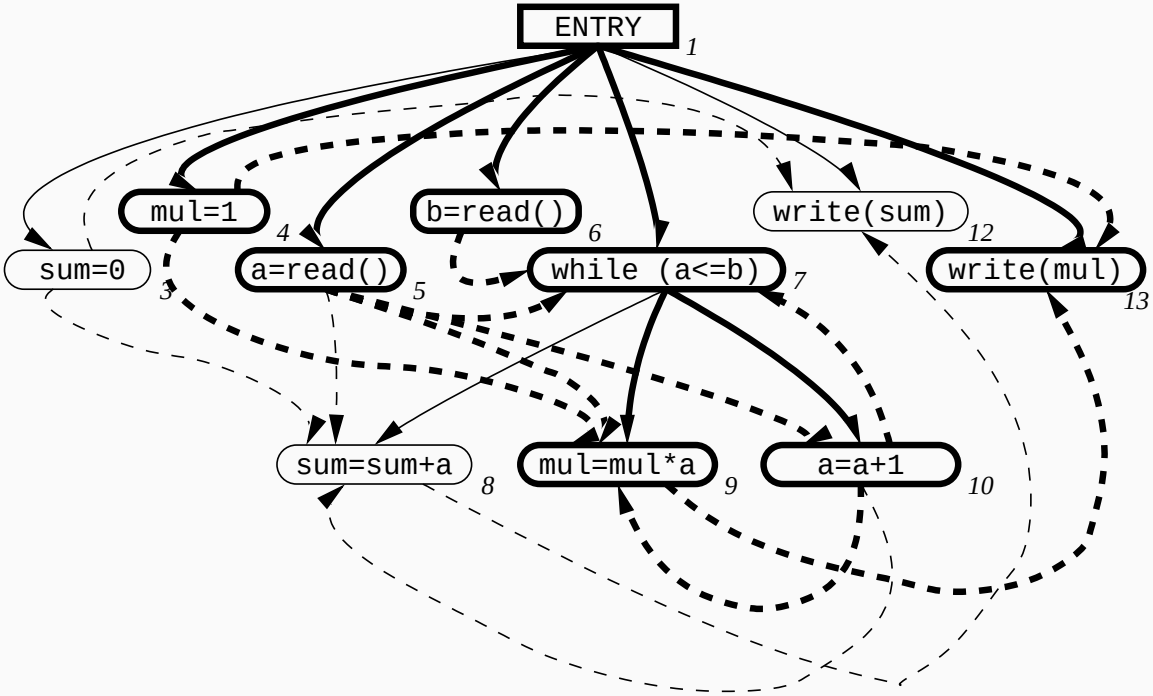
Liegt der PDG vor, ist das Slicing selbst nur noch ein Erreichbarkeitsproblem.

Zuerst besteht ein Slice-Kriterium im Graphen nur noch aus einem Knoten $n$, der auch für die in diesem Knoten zugewiesene Variable steht.

- Ein *Forward-Slice* besteht aus dem Teilgraphen, der alle von $n$ aus erreichbaren Knoten enthält.
- Ein *Backward-Slice* besteht aus dem Teilgraphen, der alle Knoten enthält, von denen aus $n$ zu erreichen ist.

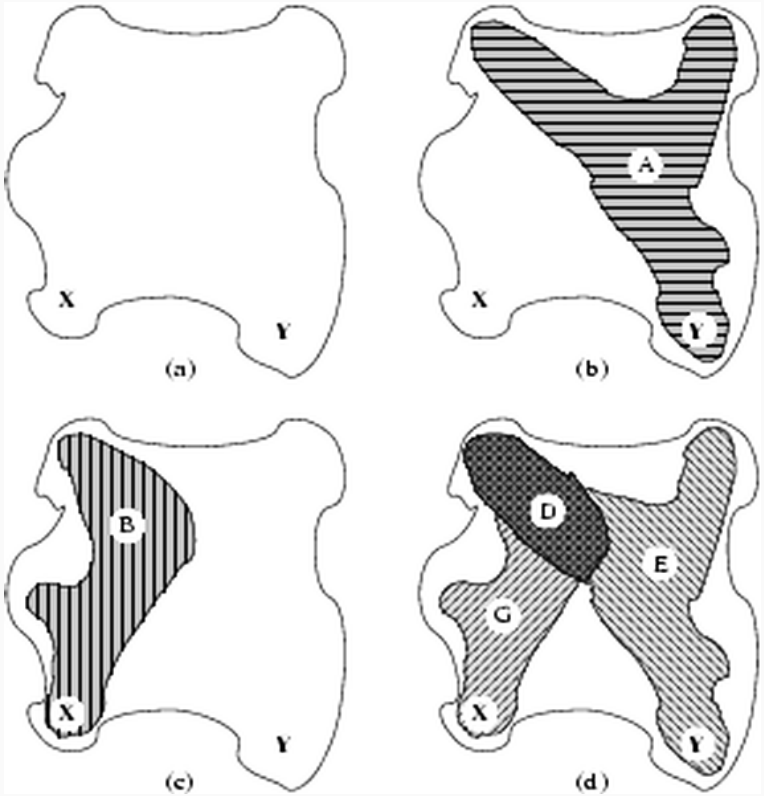
# *Slice für* (`mul`, 13) *im PDG*

Backward-Slice für Zeile 13:



# *Mengenoperationen mit Slices*






$A$: Slice für Y
$B$: Slice für X
$D$: Schnitt $A \cup B$ (*Backbone*)
$E$: Subtraktion $A - B$ (*Dice*)
$G$: Subtraktion $B - A$ (*Dice*)

# Dice

Dice: Subtraktion zweier Slices

```
int main() {
  int a, b, sum, mul;
  sum = 0;
  mul = 1;
  a = read();
  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }
  write(sum);
  write(mul);
}
```

Programm

```
int main() {
  int a, b, sum, mul;

  mul = 1;




    mul = mul * a;



  write(mul);
}
```

Dice für $(\texttt{sum}, 12)$ und $(\texttt{mul}, 13)$

Grundidee: *Fehlereingrenzung durch Dices*
(Subtraktion „fehlerhafter“ − „korrekter“ Slice)









# *Erweiterungen und Varianten*

**Interprozedurales Slicing** bestimmt Slices über Prozedurgrenzen hinweg

Herausforderung: Trade-off zwischen

- Entfalten der Prozeduren an Aufrufstelle (⇒ viel Platz)
- Zusammenfassen der Aufrufe (⇒ Ungenauigkeit)









# *Erweiterungen und Varianten*

**Interprozedurales Slicing** bestimmt Slices über Prozedurgrenzen hinweg

Herausforderung: Trade-off zwischen

- Entfalten der Prozeduren an Aufrufstelle (⇒ viel Platz)
- Zusammenfassen der Aufrufe (⇒ Ungenauigkeit)

**Dynamisches Slicing** bestimmt Slices für einen bestimmten Programmlauf

Vorteil: Kontrollfluß ist bekannt
⇒ exaktere Datenabhängigkeiten
⇒ höhere Präzision als statisches Slicing

Herausforderungen: Effiziente Code-Instrumentierung, präzise Kontrollabhängigkeiten







# *Weitere Techniken der Programmanalyse*



**Laufzeitanalyse** Bestimmen einer Obergrenze für die maximale Laufzeit eines Programms oder Programmteils (vgl. Lehrstuhl Wilhelm)







# *Weitere Techniken der Programmanalyse*



**Laufzeitanalyse** Bestimmen einer Obergrenze für die maximale Laufzeit eines Programms oder Programmteils (vgl. Lehrstuhl Wilhelm)

**Ursache-Wirkungs-Ketten** Bestimmen von Ursachen und Wirkungen im Programm durch systematische Experimente (vgl. Lehrstuhl Zeller)

All diese Ergebnisse müssen geeignet dargestellt werden!



# *Weitere Techniken der Programmanalyse*

**Laufzeitanalyse** Bestimmen einer Obergrenze für die maximale Laufzeit eines Programms oder Programmteils (vgl. Lehrstuhl Wilhelm)

**Ursache-Wirkungs-Ketten** Bestimmen von Ursachen und Wirkungen im Programm durch systematische Experimente (vgl. Lehrstuhl Zeller)

All diese Ergebnisse müssen geeignet dargestellt werden!

Mehr zum Thema „Programmanalyse und Programmverstehen“: Vorlesung *Automated Debugging* (WS 2002/03)









# *Zusammenfassung*

- *Software Reengineering* = Reverse Engineering + Restrukturierung
- *Reverse Engineering* ist die Extraktion und Repräsentation von Informationen aus einem Software-System
- *Programmverstehen* ist der wichtigste Bestandteil des Reverse Engineering
- Sichten wie Dokumentation, Benutzung, Abdeckung helfen, das Programm zu verstehen
- *Program Slicing* kann helfen, die für ein Verhalten *wesentliche Aspekte* eines Programms zu isolieren