Universität Zürich[UZH]

Institut für Informatik

Martin Glinz

Software-Qualität – Ausgewählte Kapitel

Kapitel 4
Debugging

© 2007- 2012 Martin Glinz. Alle Rechte vorbehalten. Speicherung und Wiedergabe für den persönlichen, nicht kommerziellen Gebrauch gestattet; bei auszugsweiser Verwendung mit Quellenangabe. Verwendung für Unterrichtszwecke oder kommerziellen Gebrauch nur mit vorheriger schriftlicher Genehmigung des Autors.

4.1 Grundlagen

- Terminologie

  Debugging (Fehlerbehebung) – Der Prozess, den Defekt zu finden und zu korrigieren, der einen beobachteten Fehler verursacht

  Defekt (defect, fault) – eine schadhafte Stelle in einem Programm

  Fehler (error) – Eine Abweichung der tatsächlichen Ergebnisse von den erwarteten / richtigen Ergebnissen

- Hinweise:
  - Ein Fehler kann durch eine Kombination mehrerer Defekte verursacht sein
  - Der selbe Defekt kann sich in mehreren Fehlern manifestieren
  - „Programm“ ist hier in umfassendem Sinn gemeint: kann eine einzelne Methode oder ein ganzes Software-System sein

Ursache und Wirkung

- In der Regel führt ein Defekt
  - nicht sofort zu einem beobachtbaren Fehler,
  - sondern zu Fehlern im Programmzustand,
  - die sich fortpflanzen
  - und sich irgendwann als beobachtbare Fehler manifestieren
- Die Hauptaufgabe des Debugging ist die Erkennung/Rekonstruktion der Ursache-Wirkungskette von einem Defekt zu einem beobachtbaren Fehler

Defekte und ihre Lokalisierung

- Klassisch: der fehlerverursachende Defekt ist ein Programmierfehler
- Alternativen:
  - Defekte in anderen Artefakten: Anforderungsspezifikation, Entwurfsdokument, Benutzerhandbuch, ...
  - Defekte in Datenbeständen
  - Defekte in Prozessen
  - Benutzerirrtümer
- Manche Defekte sind nicht lokal, sondern betreffen ein ganzes (Sub-) System

Ein Beispiel: Zahlen sortieren [Zeller 2005]

Name: `sample`

Autor: Andreas Zeller

Sprache: C

Aufruf: `./sample` arg_1 arg_2 `...` arg_n

Voraussetzung: arg_1 arg_2 `...` arg_n sind Integer-Zahlen, n $\in$ IN

Ergebniszusicherung: Auf der Standard-Ausgabe erscheinen die eingegebenen n Zahlen sortiert in aufsteigender Reihenfolge

Ausführung von sample mit Testdaten:

```
$  ./sample 9 7 8
Output: 7 8 9
$ _
```

```
$  ./sample 11 14
Output: 0 11
$ _
```

Programm `sample`: Code

```
/* sample.c -- Sample C program to be debugged */

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

static void shell_sort(int a[], int size)
{
    int i, j;
    int h = 1;

do {
        h = h * 3 + 1;
    } while (h <= size);
    do {
        h /= 3;
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}
```

Programm `sample`: Code – 2

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

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    printf("Output: ");
    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);

    return 0;
}
```

Was tun?

Feststellung:

Es gibt Eingabedaten, für die `sample` ein falsches Resultat liefert

Frage:

- Wie finden wir im Code den Defekt, der diesen Fehler verursacht?
- Können wir die Suche nach dem Defekt systematisieren?

4.2 Der Debugging-Prozess

- Das Problem exakt beschreiben
  - Manchmal erschließt das bereits die Problemursache
- Ist das Problem ein Softwarefehler?
  Wenn ja:
  - Fehler reproduzieren
  - Fehlerverursachenden Testfall vereinfachen und wenn möglich automatisieren
  - Verursachenden Defekt lokalisieren
    - Hypothesen bilden und testen
    - Programmzustände beobachten
    - Zusicherungen erstellen und automatisiert prüfen
    - Ursache-Wirkungskette isolieren
  - Gefundenen Defekt beheben

Der Debugging-Prozess –2

- Wenn Problem kein Softwarefehler ist
  - Andere Problemursache suchen und beheben
    Beispiele: Defekte in Handbüchern, Prozess- oder Schulungsdefizite
- Effektivität der Problembehebung prüfen (egal, ob Softwarefehler oder anderer Fehler):
  - Tritt das Problem nicht mehr auf?
  - Treten keine neuen, bisher nicht bekannten Probleme auf?
- Notwendige Infrastruktur
  - Problemmeldewesen
    - Prozess für die Behandlung von Problemmeldungen
    - Werkzeug für die Verwaltung von Problemmeldungen
      Beispiel: Bugzilla
  - Konfigurationsmanagement für Software-Artefakte

4.3 Fehler reproduzieren

Beispiel: Mozilla Problemmeldung Nr. 24735 aus dem Jahr 1999
[Zeller 2005, p. 55]

-> Start mozilla
-> Go to bugzilla.mozilla.org
-> Select search for bug
-> Print to file setting the bottom and right margins to .50
(I use the file /var/tmp/netscape.ps)
-> Once it's done printing do the exact same thing again on the same file (/var/tmp/netscape.ps)
-> This causes the browser to crash with a segfault

Ziel: eine möglichst einfache Testvorschrift, mit der ein gemeldeter Fehler reproduziert wird

Probleme

- Reproduzieren der Umgebung, in der das Problem auftritt
- Gegebenenfalls Reproduzieren der Vorgeschichte
- Bei Softwarefehlern: Reproduktion eines Programmlaufs, bei dem der Fehler auftritt
  Zu reproduzieren sind gegebenenfalls
  - Eingaben
  - Initiale persistente Daten
  - Benutzerinteraktion, Interaktion mit anderen Systemen
  - Zeit
  - Kommunikation mit anderen Prozessen
  - Prozessabläufe
  - Zufallsdaten

4.4 Testfälle vereinfachen und automatisieren

- Gegeben: ein Testfall, welcher reproduzierbar einen Fehler erzeugt
- Ziel:
  - Entfernen aller irrelevanten Bestandteile des Testfalls
  - Automatisieren des vereinfachten Testfalls
- In einem optimal vereinfachten Testfall sind alle Bestandteile relevant, d.h. Weglassen eines Bestandteils erzeugt den Fehler nicht mehr in der ursprünglichen Form
- Ansatzpunkte
  - Umgebung vereinfachen
  - Geschichte reduzieren
  - Eingaben / Interaktionen vereinfachen

Automatisierung des Testfalls

- Der fehlerverursachende Testfall muss oft ausgeführt werden
  - Zur Untersuchung von Vereinfachungen
  - Zum Testen von Hypothesen bei der systematischen Defektlokalisierung
- ➪ Automatisierung zahlt sich rasch aus
- Techniken zur Testautomatisierung → siehe Kapitel 4

Umgebung vereinfachen

- Feststellen, welche Umgebungsbedingungen für den Fehler relevant sind und welche nicht, zum Beispiel
  - Hardware und Betriebssystem
  - Zustand persistenter Daten
  - Zeit
  - Zustand von Nachbarsystemen
- Alle irrelevanten Bedingungen können vernachlässigt werden
- Ziel: minimaler Aufwand für das Einrichten der Testumgebung, in der ein Test den Fehler erzeugt
- Mittel: systematisches Probieren

Geschichte vereinfachen

- Lässt sich die Zahl der Schritte, die zur Auslösung des Fehlers führen, reduzieren?
- Mittel: systematisches Probieren
- Beispiel: Mozilla Problemmeldung Nr. 24735 [Zeller 2005, p. 55]
  Folgende Interaktionsschritte lösen den Fehler aus:
  Start mozilla; Go to bugzilla.mozilla.org; Select search for bug; Print to file setting the bottom and right margins to .50; Once it's done printing do the exact same thing again on the same file.

  Es zeigt sich, dass folgende Schritte genügen, um den Fehler auszulösen:
  Start mozilla; Go to bugzilla.mozilla.org; Select search for bug; Press Alt-P; Left-click on the Print button in the print dialog window

Eingaben vereinfachen

- Beispiel: Mozilla Problemmeldung Nr. 24735 (vgl. 4.3)
  - Die Eingabe der Druckfunktion ist die zu druckende Webseite
  - Diese besteht aus 896 Zeilen html-Code
- Welche Stelle in den Eingabedaten führt dazu, dass der Fehler eintritt?
- Mittel: binäres Suchen [Kernighan and Pike 1999]
  - Die Menge der Eingabedaten in zwei Hälften teilen
  - Beide Hälften testen
  - Rekursiv mit der Hälfte fortfahren, in welcher der Fehler auftritt

Eingaben vereinfachen – 2: Beispiel [Zeller 2005]

- Beispiel: Mozilla Problemmeldung Nr. 24735
- Binäres Suchen führt in zwölf Schritten zu einer einzigen, fehlerverursachenden html-Zeile:

1	896 Zeilen	✘
2	448 Zeilen	✘
3	224 Zeilen	✘
4	112 Zeilen	✔
5	112 Zeilen	✘
6	56 Zeilen	✘
...		
12	<SELECT NAME="priority" MULTIPLE SIZE=7> 1 Zeile	✘

Eingaben vereinfachen – 3

- Was, wenn der Fehler in beiden Hälften nicht auftritt, wohl aber im Ganzen?

  <SELECT NAME="priority" MULTIPLE SIZE=7> ✘
  <SELECT NAME="priority" MULTIPLE SIZE=7> ✔
  <SELECT NAME="priority" MULTIPLE SIZE=7> ✔

- Statt Hälften kleinere Stücke ausschneiden, zum Beispiel Viertel

  <SELECT NAME="priority" MULTIPLE SIZE=7> ✔
  <SELECT NAME="priority" MULTIPLE SIZE=7> ✘
  ☞ <SELECT NAME="priority" MULTIPLE SIZE=7> ✘
  <SELECT NAME="priority" MULTIPLE SIZE=7> ✔

- Mit Achteln, etc. fortfahren

- Endergebnis hier: <SELECT> ✘

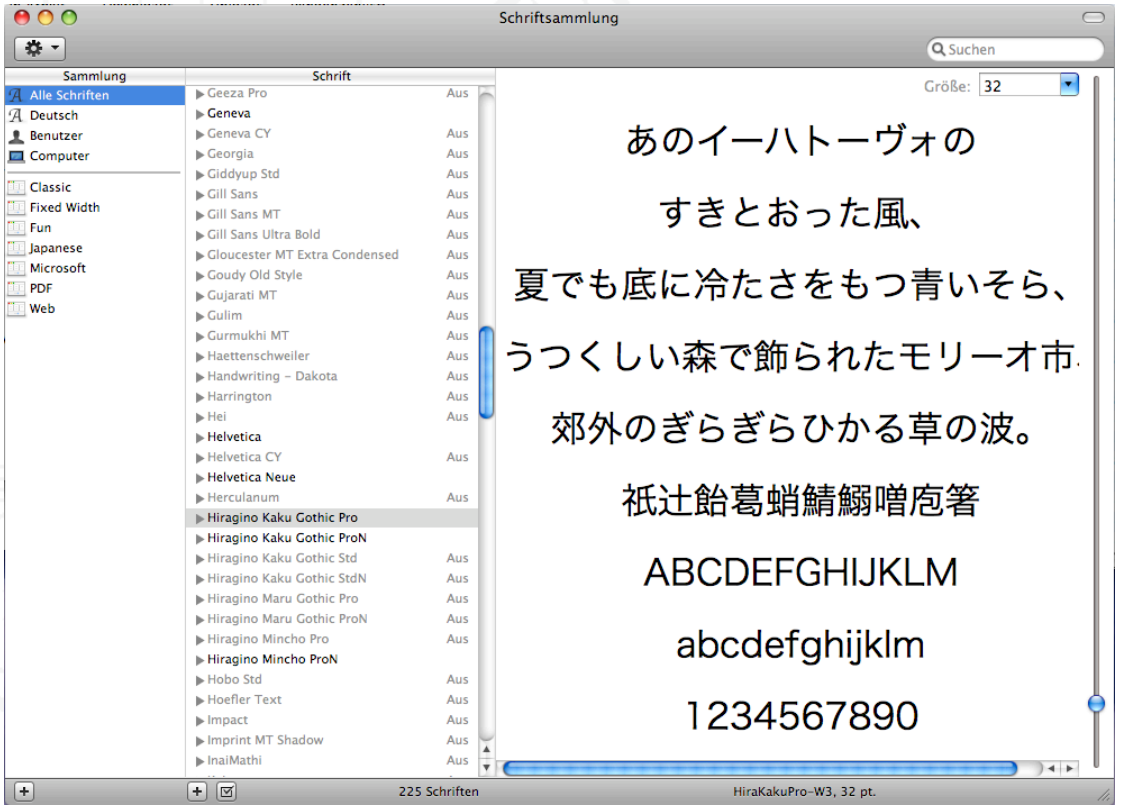
Automatisierung der Vereinfachung

- Die Vereinfachung ist teilweise automatisierbar
  - insbesondere das Prinzip des binären Suchens
  - anwendbar beispielsweise zur Vereinfachung von Eingabedaten oder Interaktionssequenzen
- Beispiel: Zellers ddmin delta debugging Algorithmus [Zeller 2005, Kapitel 5.4-5.5]

Ein weiteres Beispiel

Microsoft PowerPoint 2004 Version 11.0 stürzt auf dem MacBook Pro mit Mac OS 10.5.6 beim Aufstarten ab, wenn in der Schriftsammlung die Schrift Hiragino Kaku Gothic Pro deaktiviert wird.

Durch Intervallhalbierung wird die minimale Differenz zwischen einem funktionierenden und einem fehlerverursachenden Testfall gefunden



4.5 Techniken der Defektlokalisierung

- Hypothesen aufstellen und testen
- Statische und dynamische Programmanalyse
  - Steuerfluss
  - Datenfluss
- Programmzustände analysieren
- Programmablauf beobachten
- Zusicherungen im Programm dynamisch prüfen
- Ursache-Wirkungsketten erkennen und isolieren
- Debugging nach Gefühl

Hypothesen aufstellen und testen

- Die Grundlage des systematischen Debuggings
- Prinzip: Erkenntnisgewinn durch Theoriebildung und Experiment
  1. Eine Hypothese aufstellen
  2. Aus der Hypothese Prognosen ableiten
  3. Prognosen experimentell überprüfen
  4. Wenn Prognose und Ergebnis übereinstimmen
     - Richtigkeit der Hypothese wird wahrscheinlicher
     - Versuchen, die Hypothese weiter zu erhärten

     → Theorie

     Sonst:
     - Hypothese verwerfen
     - Neue oder geänderte Hypothese aufstellen, weiter mit Schritt 2
- Wichtig: Hypothesen und deren Überprüfung systematisch aufzeichnen

Hypothesen finden

Mögliche Ansätze:

- Analyse der Problembeschreibung
- Statische Analyse des Codes
- Analyse einer fehlerhaften Ausführung
- Vergleich von korrekten und fehlerhaften Ausführungen
- Aufbauen auf den bisherigen Hypothesen: eine neue Hypothese
  - muss kompatibel mit den bisher angenommenen Hypothesen sein
  - darf nicht auf Annahmen basieren, die auf bereits verworfenen Hypothesen basieren

Prognosen ableiten und überprüfen

- Möglichkeiten
  - Statische Analyse des Codes
  - Dynamische Analyse des Codes
  - Beobachten von Systemzuständen
  - Dynamisches Prüfen von Zusicherungen
- Deduktives Vorgehen: Logische Schlüsse ziehen aus
  - vorhandenem Wissen
  - dem Code
  - Testfällen und ihren Resultaten
- Experimentelles Vorgehen
  - Programmausführung beobachten
  - Programmablauf beobachten

Beispiel: Programm `sample` (vgl. 4.1)

- Erste Hypothese

  Programm läuft korrekt

  Prognose: Eingabe von `11 14` liefert als Resultat `11 14`

  Experiment: `$ ./sample 11 14`

  `Output: 0 11` ✘

  ➥ Hypothese wird verworfen

- Zweite Hypothese

  Druckt die falschen Variablen

  Prognose: `a[0]=11 a[1]=14`, aber Ausgabe ist `Output: 0 11`

  Experiment: Code für Eingabe und Sortieren ersetzen durch

  ```
  a[0] =11; a[1] =14; argc = 3;
  ```

  Resultat: `Output: 11 14` ✘

  ➥ Hypothese wird verworfen

Statische und dynamische Analyse

- Analyse von
  - Steuerfluss (Untersuchen des Steuerflussgraphen des Programms)
  - Datenfluss (Program slicing; siehe Grundvorlesung)

- Statische Analyse
  - Liefert die prinzipiell möglichen Abläufe bzw. Datenflüsse
  - Keine Ausführung erforderlich
  - Unabhängig von konkreten Testfällen
- Dynamische Analyse
  - Analysiert einen konkreten Programmlauf
  - Liefert die tatsächlichen Abläufe bzw. Datenflüsse

Beispiel: statisches vs. dynamisches Slicing

```
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);
}
```

Beispielprogramm

```
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);
}
```

Statischer Slice
von mul in Zeile 13

```
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);
}
```

Dynamischer Slice
von mul in Zeile 13
mit a=5, b=2

Zustände analysieren

- Grundlage: ein Defekt
  - führt typisch zu einer Folge fehlerhafter Programmzustände
  - die sich irgendwann in beobachtbaren Fehlern manifestieren
- „Verdächtige“ Programmzustände oder Teile davon überprüfen
  - Instrumentieren des Codes: gezielt Variablenwerte
    - aufzeichnen
    - ausdrucken
    - direkt, oder mit Hilfe eines Logging-Rahmens, zum Beispiel LOG4J [Logging Services]
  - Verwenden eines Debuggers
    - Programm im Debug-Modus übersetzen
    - Programmausführung an kritischen Stellen anhalten
    - Aktuelle Variablenwerte inspizieren

Beispiel: Programm `sample` (vgl. 4.1)

- Dritte Hypothese

  Sortierprozedur wird mit falschen Parametern aufgerufen

  Prognose: Werte im Feld `a` und/oder Wert von `argc` falsch

  Experiment: Vor dem Aufruf von shell_sort wird der Code instrumentiert mit

```
printf("Parameter von shell_sort: ");
for (i = 0; i <= argc; i++)
  printf("%d ", a[i]);
printf ("%d ", argc);
printf("\n");
```

  Resultat: `Parameter von shell_sort: 11 14 0 3` ✘

  ➥ Hypothese wird bestätigt

- Alternativ könnte für das Experiment auch ein Debugger verwendet werden

Programmablauf beobachten

Mit Hilfe eines Debuggers

- Programm schrittweise ausführen
  - Erwarteten und tatsächlichen Steuerfluss vergleichen
  - Gegebenenfalls Teile des Systemzustands inspizieren
- Variablenzugriffe verfolgen

Zusicherungen prüfen

- Verträge von Klassen bzw. Methoden in Form von Zusicherungen, d.h.
  - Voraussetzungen
  - Ergebniszusicherungen
  - Invarianten

  formal spezifizieren (vgl. Grundvorlesung Software Engineering)
- Zusicherungen mittels eines geeigneten Laufzeitsystems dynamisch prüfen
- Programmzustände, in denen Zusicherungen verletzt sind, analysieren

Ursache und Wirkung

- Das Problem der Kausalität
  - Im Zeitraum von 1950-1960 sind der Rückgang der Storchpopulation und die Zunahme der Teerstraßen stark korreliert
  - Ist die Zunahme der Teerstraßen eine/die Ursache für das Verschwinden der Störche?
- Testen von Kausalität: a ist eine Ursache für b wenn
  - b eintritt, wenn a vorher eingetreten ist
  - b nicht eintritt, wenn a nicht eingetreten ist
  - Alle übrigen Variablen konstant gehalten werden

Ursache und Wirkung – 2

- Experimenteller Nachweis von Kausalität
  - Im Allgemeinen schwierig: Problem des kontrollierten Experiments
  - Beim Debugging machbar:
    - Kontrollierbare Umgebung
    - Testfälle reproduzierbar
- Beim Debugging kann eine Ursache für einen Fehler f aufgefasst werden als die Differenz zwischen
  - einem Fall, in dem f auftritt und
  - einem Fall, in dem f nicht auftritt
- Problem: eine Wirkung kann viele Ursachen haben
- Gesucht: eine minimale Ursache
- Mittel: Suchen einer minimalen Differenz

Beispiel: Programm `sample` (vgl. 4.1)

- Vierte Hypothese

  `shell_sort` *müsste mit* `argc-1` *(statt* `argc`*) aufgerufen werden*

  Prognose: Resultat ist korrekt

  Experiment: Ausführen mit geändertem Code (oder durch Eingriff in den Programmzustand mittels eines Debuggers)

  Resultat: `Output: 11 14` ✔

  ➥ Hypothese wird bestätigt

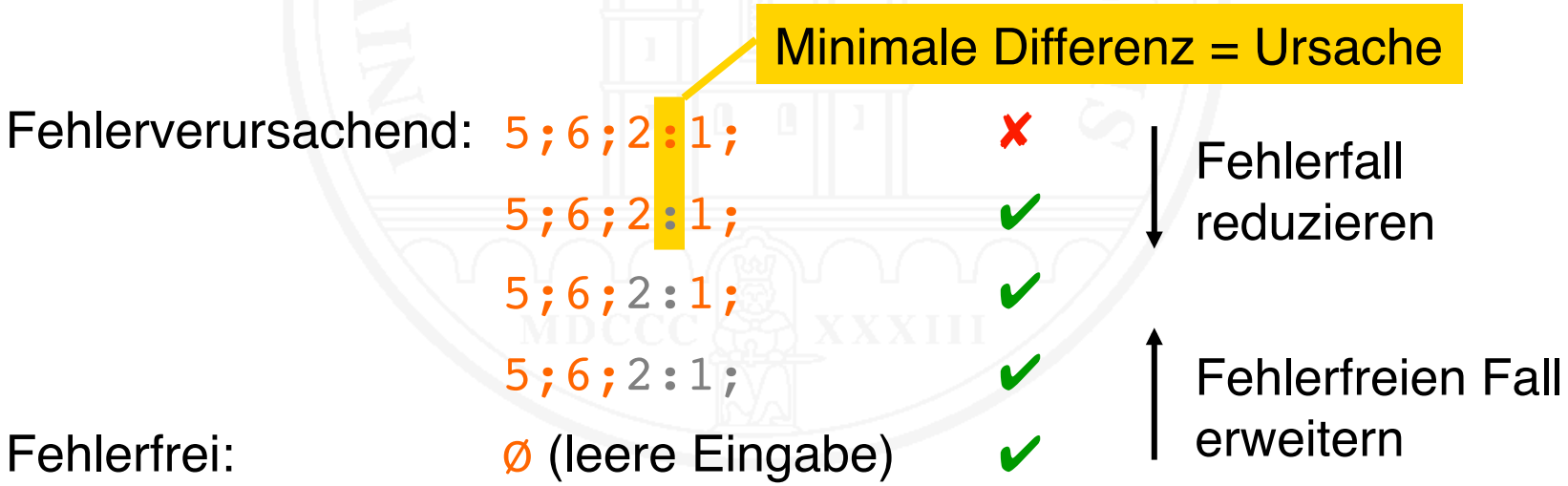
- Aus dem Test der ersten Hypothese wissen wir, dass ein Aufruf mit `argc` zum Fehler führt
- Die Differenz zwischen den beiden Programmversionen ist „`-1`“ in Zeile 36
- Dies ist eine minimale Fehlerursache

Ursache-Wirkungsketten erkennen und isolieren

- Die Ursache eines Fehlers ist in der Regel nicht ein Defekt sondern ein fehlerhafter Zustand
  - Ursache-Wirkungsketten erkennen
  - und vom irrelevanten Rest isolieren
- Aufwendig: Erfordert das Aufstellen und Testen einer Menge von Hypothesen
- Systematik erforderlich
- Automatisierbar: Zellers Delta Debugging Algorithmus [Zeller 2002]

Ursachen isolieren mit Delta Debugging

- Unterschied von Isolieren und Vereinfachen:
  - Vereinfachen: Suche nach einem minimalen fehlerverursachenden Testfall
  - Isolieren Suche nach einem fehlerverursachenden und einem fehlerfreien Testfall mit minimaler Differenz
- Beispiel: Isolieren der kleinsten Fehlerursache in einer Eingabe



Debugging nach Gefühl

- Erfahrene Leute entwickeln ein Gespür für mögliche Fehlerursachen
- Intuition führt manchmal viel schneller zum Defekt als alle systematischen Verfahren
- Problem:
  - Rechtzeitig merken, wenn Intuition nicht zum Ziel führt
  - Dann konsequent umschalten auf systematisches Vorgehen
- Mögliches Vorgehen
  - Für eine strikt begrenzte Zeit nach Gefühl suchen
  - Wenn dies in der gegebenen Zeit zum Ziel führt: Heureka!
  - Sonst: abbrechen und neu mit systematischer Suche beginnen

4.6 Defektbehebung

Wenn der Defekt erkannt ist

- Tragweite des Defekts abschätzen
- Art und Umfang der Defektbehebung klären
- Auswirkungen auf andere Systemteile abschätzen
- Notwendige Änderungen systematisch durchführen
- Behelfs-Reparaturen vermeiden

4.7 Effektivität der Problembehebung prüfen

- Feststellen, ob das Problem nicht mehr auftritt
- Bei Software-Fehlern durch erneutes Ausführen von Testfällen, die bisher einen Fehler erzeugt haben
- Auf unerwünschte Nebenwirkungen prüfen
  - Sicherstellen, dass durch die Änderung keine neuen, bisher nicht bekannten Probleme auftreten
  - Bei Software-Fehlern durch Regressionstest

Literatur

Bugzilla. http://www.bugzilla.org

S.C. McConnell (1993). *Code Complete: A Practical Handbook of Software Construction*. Redmond: Microsoft Press.

B.W. Kernighan, R. Pike (1999). *The Practice of Programming*. Reading, Mass.: Addison-Wesley.

Logging Services. http://logging.apache.org

M. Weiser (1992). Programmers Use Slices When Debugging. *Communications of the ACM* **25**, 7. 446-452.

A. Zeller (2002). Isolating Cause-Effect Chains from Computer Programs. *Proceedings of the 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Charleston, South Carolina.1-10.

A. Zeller (2005). *Why Programs Fail: A Guide to Systematic Debugging*. Amsterdam: Morgan Kaufmann und Heidelberg: dpunkt.