16 Testen

Problem:

Tests können immer nur die Anwesenheit, nie aber die Abwesenheit von Fehlern beweisen (E. Dijkstra?).

Wirklich?

Warum dann Testen?

- Im Prinzip einfach
- Weniger irrtumsanfällig als Verifikation?
- Gutes Kosten/Nutzen Verhältnis für die ersten x % der Fehler

(Folien aus einer Vorlesung v. Prof. Tichy, Universität Karlsruhe)

Testsuiten

- Testfälle zusammentragen
- Automatisch durchlaufen

⇒ Zusätzliche Infrastruktur, (Skripte, Dateiformate, ...

+ Routinemäßig nach jeder Änderung laufen lassen

- Sowohl für Modultest als auch für Integrationstest

Problemfälle

- Interaktiver Code
- Ausnahmebehandlungen (Platte voll, ...)

⇒ Anteil schwer testbaren Codes kleinhalten!

⇒ Spezielle Werkzeuge (UI Traces?)

- Sonderfälle trotzdem ab und zu testen (Debugger, spezieller setup)

17 Werkzeuge

17.1 Debugger

17.1.1 Aufruf

- Code **mit** Debug-option (z.B. `-g`) **ohne** Optimierung übersetzen
- Vernünftigen Debugger finden (früher noch nicht mal Quelltextorientiert (**ddd** zur Not `xxgdb`)
- Passend zur Sprache (bei C++ oft nur bedingt)
- Passend zum Compiler

17.1.2 Grundfunktionen

- Einzelschrittmodus
- Variablenwerte darstellen
- Haltepunkte (warum?)
- Wo werden Signale ausgelöst? (z.B. segmentation violation)
- Stapelinhalt darstellen (Zeilennummern, Variableninhalte)

17.1.3 Advanced Features

Display: Variablenwerte bei jedem Schritt anzeigen

„Zeitzünder“: Haltepunkte erst beim k-ten Durchlauf beachten

Bedingter Haltepunkt: beliebige Bedingung

Watchpoints: Stoppen wenn bestimmte Variable geändert wird. !

ändern von Variablenwerten

aufrufen von Prozeduren, Signale schicken, Sprünge . . .

17.1.4 Nutzungsweise

- Routinemäßiges checken neuen Codes (McGuire) Problem damit? **auf Datenfluß** konzentrieren. (Display, Scaffolding)
- White-box test abrunden. Problem?
- Signal nachspüren (Quellcodeposition, Stack) *Post Mortem*
- Fehlerhaftes Verhalten ⇒ Rückwärtsverfolgung.
- Assertion violation aufklären. Je mehr und besser desto schneller eingekreist

17.1.5 Bewertung

+ Sehr mächtig, scheinbar unentbehrlich
– Was wenn man ihn entbehren muß? . . .
– Kann bei komplizierten Programmen schwerfällig sein
– Kein Ersatz für
  – Assertions
  – Testsuite
  – Inspektionen
  – **DENKEN**
– Komplizierte Situationen
  – Parallel/Multithreaded (selbst mit speziellen Debuggern)
  – Embedded Systems
  – Race Conditions
  – Fehler im optimierten Code

Debugger sind nett, aber wer sich auf sie verläßt, der ist verlassen

17.1.6 Wie geht das?

- Compiler speichert Assoziationen zwischen Maschinenbefehlen und Quelltext bzw. zwischen Adressen und Variablennamen im Objektcode.
- Debugger muß Objektcodeformat und Datenlayout des Compilers kennen
- Breakpoints modifizieren Code
- Prozessor unterstützt Einzelschrittmodus (und sogar Watchpoints)

17.2 Purify

C/C++ auf die Finger schauen

Instrumentieren von Programmen:

```
purify CC -g fehler.C
```

Debug-Optionen nicht vergessen. Purify instrumentiert den Objektcode.

- Lesen nicht initialisierten Speichers
- Out of bound read/write
  - Array, Stack-Array
  - Stack
  - Freed memory
  - Invalid pointer
  - Null pointer
- Speicherlecks
- Signale (division durch 0, seg. viol. ...)

17.3 Code Coverage

Beispiel: `gcov`

```
g++ -g -fprofile-arcs -ftest-coverage cover.C
a.out
gcov cover.C
 83.33% of 6 source lines executed in file
cover.C
more cover.C.gcov
          #include <stdlib.h>
          #include <iostream.h>
     1    main () {
     1      int littlePrimes[] = {2,3,5,7};

     1      int fourthPrime = littlePrimes[3];
     1      if (fourthPrime < 7) {
######        cout << "Haeh?";
            }
     1      cout << fourthPrime;
          }
```


17.4 make

- Übersetzen und linken
- Tut nur das nötige (spart Rechenzeit)
- Zentrale Kontrolle der Compileroptionen
- Sehr flexibel (versch. Programmiersprachen, ...)

Probleme

- Üble Syntax
- Manuelle Spezifikation der Abhängigkeiten ist fehleranfällig (Es gibt Werkzeuge, die das automatisieren.)

17.5 Sonstige Unix-Entwicklungswerkzeuge

- Moderne shells (history, name completion, ...)
- Shell-Skripte (oder `awk` oder `perl`)
  - Dateiformate umwandeln
  - Testsuiten ausführen
- `grep` z.B. in welchen Dateien wird `fooBar()` aufgerufen?
- `diff` Unterschiede ermitteln
- `wc` Dateilänge
- `time` Zeitbedarf ermitteln
- `nm` Namen in Objektdatei

17.6 Versionsverwaltung

Beispiel: `RCS` – versuchen sie mal `man rcsintro`

Einchecken: Ich bin mit einer Änderung fertig. Automatisches einfügen von History-Information

Auschecken (`co`): „Ich will es mir nur ansehen“

Lock `co -l`: Ich will was ändern. Warnung falls anderes lock

Platz sparen: Nur Änderungen werden gespeichert

... Versionsbaum, Auswahl, Merge, ...

17.7 Hilfen durch den Editor

Beispiel Emacs:

Klammerung: Hinweis auf mögliche Fehler

C++ Mode: Automatische Einrückung (an Programmierkonventionen anpassen)

Font-lock-mode: Syntaxorientiertes Einfärben (Kommentare, ...)

Übersetzen: Automatisches Anspringen der Fehlerstelle

gdb: Bessere Umgebung als `xxgdb` (aber nicht so gut wie `ddd`)

RCS Unterstützung

Aber nicht übertreiben. Syntaxorientierte Editoren haben sich z.B. nicht durchgesetzt.