Programmentwicklung II

Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Brandt

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2021

Testen von Software

- Motivation
- Software-Entwicklungsprozess
- Komponententest

Software-Fehler haben Konsequenzen

- 1979: erste Venussonde flog am Ziel vorbei
  Grund: im Programm Punkt mit Komma vertauscht
  Kosten: mehrere hundert Millionen Dollar
- 1984: Überschwemmung in Frankreich
  Grund: Überlaufgefahr nicht erkannt
  Kosten: mehrere Tote
- 1985-87: computergesteuertes Bestrahlungsgerät Therac-25 zur Behandlung von Tumoren ist fehlerhaft
  Anmerkungen: Programm in PDP-11 Assembler war unzureichend dokumentiert; es gab keine Hinweise darauf, dass es jemals getestet wurde
  Kosten: mehrere Tote; mehrere Schwerverletzte mit Lähmungen, Verbrennungen und Verstrahlungen
- 1991: Patriot-Raketen-Fehler
  Grund: ungenaue Berechnung der Zeit seit Systemstart wegen Rundungsfehler
  Kosten: 28 Tote, weil die zu treffende Rakete verfehlt wurde und diese in eine Kaserne einschlug

Software-Fehler haben Konsequenzen

- 1993: Gepäcksortierung am Flughafen Denver

  Grund: Sortieranweisungen kamen zu spät wegen Überlastung des Netzwerks

  Kosten: 3 Milliarden Dollar, Gepäcksortierung musste zunächst per Hand durchgeführt werden; Eröffnung des Flughafens musste mehrfach um mehrere Monate verlegt werden
- 1994: Pentium-Bug: Fehler bei Division

  Grund: Tabelle mit Schätzwerten für nächste Stelle war zu klein

  Kosten: 400 Millionen Dollar
- 1996: Ariane 5 verglüht

  Grund: Software-Problem im Trägheitsnavigationssystem

  Kosten: ca. 1 Milliarde Euro
- Bank of New York buchte 32 Milliarden Dollar zuviel

  Grund: Überlauf eines 16-Bit Zählers

  Kosten: 5 Millionen Dollar (Zinsverlust)

Literatur

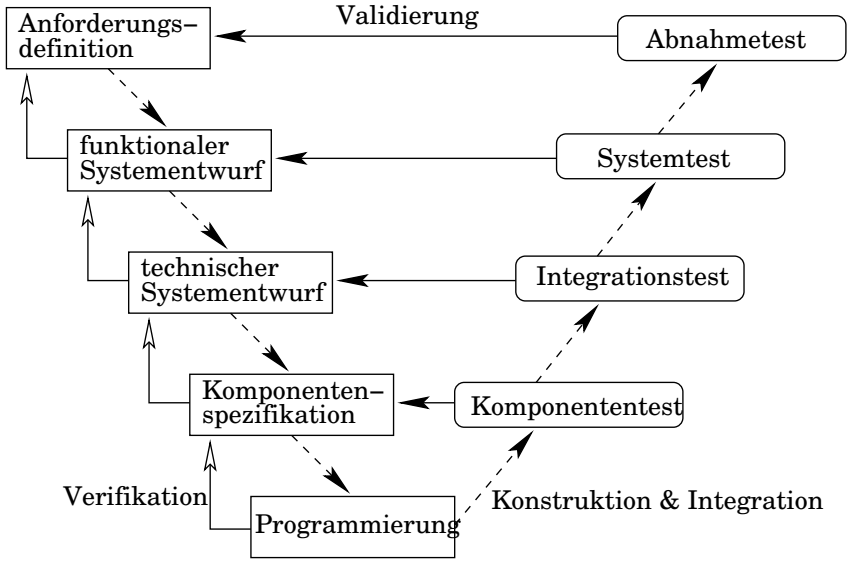
- Andreas Spillner, Tilo Linz:
  Basiswissen Softwaretest.
  dpunkt.verlag
- Harry M. Sneed, Mario Winter:
  Testen objektorientierter Software.
  Hanser Verlag.
- Wolfgang Ehrenberger:
  Software-Verifikation.
  Hanser Verlag.
- Andreas Zeller, Jens Krinke:
  Programmierwerkzeuge.
  dpunkt.verlag.

Übersicht

Testen von Software

- Motivation
- *Software-Entwicklungsprozess*
- Komponententest

Allgemeines V-Modell



Anforderungsdefinition Wünsche und Anforderungen des Auftraggebers sammeln, spezifizieren und verabschieden.

> Konfuzius: Wenn die Sprache nicht stimmt, ist das was gesagt wird nicht das was gemeint ist. So kommen keine guten Werke zustande. Also dulde man keine Willkür in den Worten.

Funktionaler Systementwurf Anforderungen auf Funktionen und Dialogabläufe des neuen Systems abbilden.

Technischer Systementwurf Technische Realisierung des Systems entwerfen (Definition der Schnittstellen, Zerlegung in Teilsysteme)

Komponentenspezifikation Für jedes Teilsystem werden Aufgabe, Verhalten, innerer Aufbau und Schnittstelle zu anderen Teilsystemen spezifiziert.

Programmierung Implementierung jedes Bausteins

Komponententest Erfüllt jeder Baustein für sich die Vorgaben seiner Spezifikation?

Integrationstest Spielen Gruppen von Komponenten so zusammen, wie im technischen Systementwurf vorgesehen?

Systemtest Erfüllt das System als Ganzes die spezifizierten Anforderungen?

Abnahmetest Weist das System aus Kundensicht die vertraglich vereinbarten Leistungsmerkmale auf?

In jeder Phase der Software-Entwicklung sind Qualitätsprüfungen durchzuführen!

Agile Methoden*

Softwaretechnik muss effizienter werden:

- Software mit immer höherer Qualität
- ist mit immer weniger personellen Ressourcen
- in immer kürzeren Zeiten zu erstellen.

Der *Rational Unified Process* oder das *V-Modell*

- sind mit Tätigkeiten überfrachtet, die für das Endergebnis nicht essentiell sind und
- sind zu schwerfällig, um auf die sich schnell ändernde Umgebung (Technik, Anforderungen, Konkurrenz) eines Software-Entwicklungsprojekts reagieren zu können.

Agile Softwareentwicklung ist Mainstream. iX 3/2010

---

* aus Bernhard Rumpe: Modellierung mit UML. Springer Verlag.

Projekte scheitern wegen schlechten Projektmanagements:

- Es wird nicht adäquat kommuniziert,
- zu viel, zu wenig oder das Falsche dokumentiert,
- Risiken nicht rechtzeitig entgegengesteuert oder
- zu spät Rückkopplung von den Anwendern eingefordert.

Die Produktivität wird bei den neuen, agilen Methoden durch das Weglassen von Arbeiten und Hierarchien gesteigert.

Derek Coleman:

- There is a move away from software processes that are hierarchical and management driven.
- The trend is to cooperative styles of development where management dictate is replaced by ethical considerations of community membership.

Es gibt nicht *den* Prozess für Software-Entwicklung. Stattdessen: Sammlung von *Konzepten*, *Best Practices* und *Werkzeugen* um projektspezifischen Erfordernissen in einem individuellen Prozess Rechnung zu tragen.

Die Softwareentwicklung hat sich durch die Verfügbarkeit verbesserter Programmiersprachen, Compiler und Entwicklungswerkzeugen geändert: Es ist heute bspw. genauso effizient eine GUI direkt zu programmieren, wie sie zu spezifizieren.

→ Die Spezifikation kann durch einen für den Anwender ausprobierbaren Prototypen ersetzt werden.

Weniger Projektbeteiligte bedeuten weniger Projektmanagement.

→ Mündige und motivierte Projektbeteiligte handeln von sich aus verantwortungsvoll und couragiert, vorausgesetzt das Projektumfeld stimmt.

siehe „Das Wettrudern“ unter `www.scheissprojekt.de/wettrudern.html`

Extreme Programming (XP)

Extreme Programming fokussiert auf das ultimative Ziel der Softwareentwicklung: den Code.

- Alles an zusätzlicher Dokumentation wird als vermeidbarer Ballast betrachtet.
- Eine Dokumentation ist aufwändig zu erstellen und oft sehr viel fehlerhafter als der Code, weil sie nicht ausreichend automatisiert analysierbar und testbar ist.
- Dokumentation reduziert Flexibilität bei Weiterentwicklung und Anpassung des Systems als schnelle Reaktion auf neue oder veränderte Anforderungen des Kunden.

Keine Dokumentation erstellen, aber Wert legen auf

- gute Kommentierung des Quellcodes (`javadoc, doxygen`)
- und eine umfangreiche Testsammlung.

Test-First-Ansatz bei XP

Vor der Entwicklung der eigentlichen Funktionalität sind Testfälle zu überlegen:

- Der Testentwurf ist de facto die Spezifikation.
- Fertig gestellter Code kann sofort überprüft werden.
- Der logische Entwurf wird von der Implementierung getrennt, denn: XP ist kein Hacking.

Egal ob V-Modell, agile Methoden oder anderer SWE-Prozess: Tests sind wichtig und immer durchzuführen!

Tests müssen voll automatisiert ablaufen und ihre Ergebnisse selbst überprüfen!

XP nicht ohne Refactoring

In der Praxis: Inkrementelle Verbesserungen und Modifikation von Programmcode.

- Um Komplexität beherrschbar zu machen: „Wir fangen erstmal klein an.“
- Um auf geänderte Kundenwünsche zu reagieren: „Wir wissen auch noch nicht so genau, was wir wollen.“

Bereits geschriebener Code muss evtl. umgeschrieben werden, um Änderungen oder Erweiterungen einfacher durchführen zu können, siehe „Softwareentwicklung als Hausbau“ unter `www.scheissprojekt.de/hausbau.html`

Refactoring:

- Semantikerhaltende Transformation eines bestehenden Programms.
- Dient nicht zur Erweiterung der Funktionalität, sondern zur Verbesserung der Qualität des Entwurfs unter Beibehaltung der Funktionalität.
  - Beispiel: Anstelle einer Liste soll in einem Programm jetzt ein Suchbaum verwendet werden, um die Applikation zu beschleunigen.
  - wichtig: Automatisch ablaufende Tests prüfen, ob das Programm noch das Gleiche wie vor der Änderung macht.

XP nicht ohne Refactoring

Weiteres Beispiel:

- Um die neue Suchfunktion besser unterstützen zu können, wird eine Adresse nicht mehr als `string` abgelegt, sondern als Struktur mit den Elementen Straße, Hausnummer, Land, Postleitzahl und Ort.
- Tests stellen sicher, dass die Software nach der Änderung immer noch funktioniert!

Entwickler lieben es, schöne Architekturen zu erstellen, die man von allen Seiten verändern kann. In den meisten Fällen ist das unnötig: Es erzeugt Komplexität, es kostet Zeit und damit dem Kunden Geld, und es ist oft schwer nachzuvollziehen.

> Halten Sie Ihren Entwurf so einfach wie möglich!

KISS-Prinzip: keap it simple, stupid!

Übersicht

Testen von Software

- Motivation
- Software-Entwicklungsprozess
- *Komponententest*

Edsger W. Dijkstra:

- The art of programming is the art of organizing complexity.
- Testen kann die Anwesenheit, aber nicht die Abwesenheit von Fehlern zeigen.

Auswahl der Testfälle:

- aus Spezifikation der Testobjekte → Black-Box
- auf Basis des Programmtextes → White-Box

Komponententest

- Die unmittelbar nach der Programmierphase erstellten Software-Bausteine, bei uns also die Klassen, werden erstmalig systematisch getestet.
- Wird ein Software-Baustein isoliert von anderen Komponenten des Systems geprüft, lässt sich eine Fehlerursache eindeutig der Komponente zuordnen.
- Ein *Testtreiber* ist ein Programm, dass Schnittstellenaufrufe absetzt und die Reaktion des Testobjekts entgegennimmt.

Damit wollen wir sicherstellen, dass das Testobjekt die geforderte Funktionalität, also das Ein-/Ausgabeverhalten, korrekt und vollständig realisiert.

Testtreiber für die Liste

```c
#include <stdio.h>
#include "liste.h"

const char OK = 1;

// Black-Box Test: Ist insert() ok? ------------------------
char test01(void) {
    list_t *l = create();

    append(l, 1);
    if (getValueAt(l, 0) != 1)
        return !OK;

    getValueAt(l, 1);
    if (getError(l) != 0)
        return OK;

    return !OK;
}
```

Testtreiber für die Liste

```
// Black-Box Test: Ist erase() ok? -------------------------
char test02(void) {
    list_t *l = create();

    append(l, 1);
    erase(l, 1);
    if (getError(l) != 0)
        return !OK;

    getValueAt(l, 0);
    if (getError(l) != 0)
        return OK;

    return !OK;
}
```

Testtreiber für die Liste

```
// White-Box Test: Ist increase() ok? ----------------------
char test03(void) {
    list_t *l = create();

    for (int i = 0; i < 50; i++)
        append(l, i);

    for (int i = 0; i < 50; i++) {
        int val = getValueAt(l, i);
        if ((getError(l) != 0) || (val != i))
            return !OK;
    }
    return OK;
}
....
```

Diese Testfälle reichen bei weitem nicht aus und müssen durch weitere Testfälle ergänzt werden. Es soll nur gezeigt werden, wie die automatische Überprüfung der Ergebnisse aussehen kann.

Testtreiber für die Liste

```
int main(void) {
    if (OK == test01())
        printf("test01 passed\n");
    else printf("!!! test01 failed !!!\n");

    if (OK == test02())
        printf("test02 passed\n");
    else printf("!!! test02 failed !!!\n");

    if (OK == test03())
        printf("test03 passed\n");
    else printf("!!! test03 failed !!!\n");

    .....
    return 0;
}
```

Wie viele Tests sind durchzuführen?

- C_0-Test: Anweisungsüberdeckung

$$\frac{\text{Anzahl durchlaufene Anweisungen}}{\text{Anzahl Anweisungen}}$$

- C_1-Test: Zweigüberdeckung

$$\frac{\text{Anzahl durchlaufene Zweige}}{\text{Anzahl Zweige}}$$

Anmerkungen:

- $C_0 = 100\% \not\Rightarrow C_1 = 100\%$ (`if`-Anweisung ohne `else`-Zweig)
- In der Regel sollte $C_1 = 100\%$ erreicht werden.

Mit dem Gnu COVerage tool gcov kann man die Testüberdeckung ermitteln und anzeigen lassen:

- Dazu wird das Programm kompiliert mittels
  `gcc -g -fprofile-arcs -ftest-coverage <prog.c> -o <prog>`
- anschließend wird der Testtreiber ausgeführt und
- schließlich wird mittels `gcov -b <prog>` die Testüberdeckung angezeigt.

Es gibt weitere Kriterien für Testüberdeckung:

- Pfadüberdeckung
- Bedingungsüberdeckung

Testüberdeckung: Beispiel

Zähle in einer Textdatei die Anzahl der Zeilen, Wörter und Zeichen.

- Ein Wort ist eine Zeichenfolge, die nicht durch ein Blank getrennt wird.
- Die zu untersuchende Datei wird als Parameter an das Programm übergeben.

Das Programm:

wordcount.c

```c
#include <stdio.h>
#include <ctype.h>
#define N 80

int main(int argc, char *argv[]) {
    if (argc != 2) {  // Programmparameter vorhanden?
        printf("usage: %s filename\n", argv[0]);
        return 1;
    }
```

Testüberdeckung: Beispiel

```
FILE *file = fopen(argv[1], "r");

if (file == NULL) {  // konnte Datei geöffnet werden?
    perror(argv[1]);
    return 2;
}

long int line_cnt = 0;
long int word_cnt = 0;
long int char_cnt = 0;

char l[N];
fgets(l, N, file);  // erste Zeile der Datei lesen
```

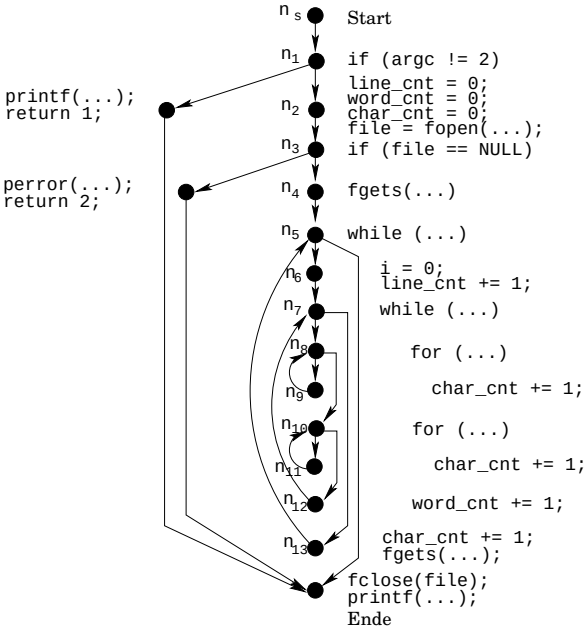
Testüberdeckung: Beispiel

```
    while (!feof(file)) {
        int i = 0;
        line_cnt += 1;

        while (l[i] != '\n') {
            for (; isblank(l[i]) && l[i] != '\n'; i++)
                char_cnt += 1;

            for (; !isblank(l[i]) && l[i] != '\n'; i++)
                char_cnt += 1;
            word_cnt += 1;
        }
        char_cnt += 1;  // count new-line character
        fgets(l, N, file);
    }
    fclose(file);
    printf("%ld %ld %ld\n", line_cnt, word_cnt, char_cnt);
    return 0;
}
```

Testüberdeckung: Kontrollflussgraph



Testüberdeckung: Beispiel

Testüberdeckung ermitteln:

- Übersetzen mit: `gcc -g -fprofile-arcs -ftest-coverage \`
  `wordcount.c -o wordcount`
- Programmausführung mit: `./wordcount test1.txt`

  Die Datei `test1.txt` ist eine speziell für das Testen angelegte Datei, bei der wir die Anzahl der Zeilen, Wörter und Zeichen kennen.
- Der Aufruf `gcov -b wordcount` liefert:

```
Lines executed:84.62% of 26
Branches executed:100.00% of 16
Taken at least once:81.25% of 16
Calls executed:75.00% of 8
```

Testüberdeckung: Beispiel

- In der Datei `wordcount.c.gcov` finden wir weitere Informationen, unter anderem den Hinweis, dass die Zeile

  ```
  perror(argv[1]);
  ```

  nicht durchlaufen wurde.
- Daher starten wir das Programm erneut, aber jetzt geben wir eine Datei an, die nicht existiert, und starten das Programm erneut.
- Nun erhalten wir die folgende Ausgabe von `gcov -b wordcount`:

```
Lines executed:92.31% of 26
Branches executed:100.00% of 16
Taken at least once:87.50% of 16
Calls executed:87.50% of 8
```

Testüberdeckung: Beispiel

- Diesmal finden wir in der Datei `wordcount.c.gcov` den Hinweis, dass die Zeile

  ```
  printf("usage: %s filename\n", argv[0]);
  ```

  nicht durchlaufen wurde.
- Daher starten wir das Programm erneut, geben aber keinen Dateinamen an.
- Nun erhalten wir die folgende Ausgabe von `gcov -b wordcount`:

  ```
  Lines executed:100.00% of 26
  Branches executed:100.00% of 16
  Taken at least once:93.75% of 16
  Calls executed:100.00% of 8
  ```

Das sieht doch schon ganz gut aus. Nicht ausführbarer Code wie im folgenden Beispiel kann gefunden werden:

```
if (2 * x > 0) {
    x = x / 2;
    if (x < 0)       // kann nicht erfüllt sein!
        x += 10;     // wird nie ausgeführt!
}
```

Testüberdeckung: Beispiel

Überdeckung hängt (leider) von der Compiler-Optimierung ab!

- Wenn wir den Compiler ohne Optimierung starten und dann obige Testfälle ausführen, erhalten wir für die folgende Zeile

  ```
  for (; isblank(l[i]) && l[i] != '\n'; i++)
  ```

  die Ausgabe:

  ```
  branch  0 taken 67%
  branch  1 taken 100%
  branch  2 never executed
  branch  3 taken 100%
  ```

- Kompilieren mit Optimierung `-O3` liefert für obige Testfälle und die obige Zeile die Ausgabe

  ```
  call    0 returns 100%
  branch  1 taken 100%
  branch  2 taken 67%
  branch  3 taken 100%
  ```

Pfadüberdeckung

Fehlende Zweige werden so leider nicht erkannt: Wenn die Datei Zeilen mit mehr als 80 Zeichen enthält, liefert das Programm falsche Werte oder stürzt ab!

Eigenschaften eines Pfadüberdeckungstests:

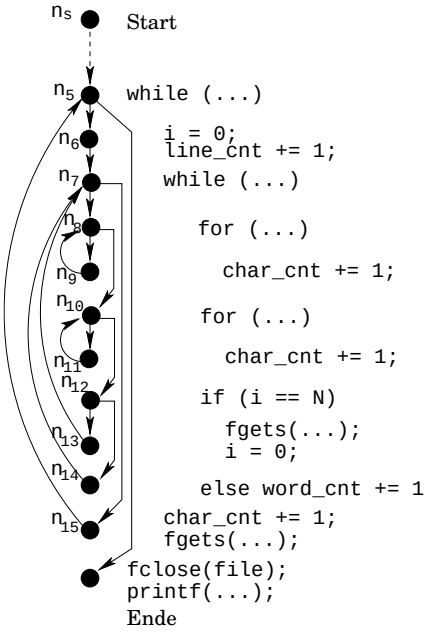
- verlangt Ausführen aller Programmpfade (unrealistisch)
- entwickelt zum Testen von Schleifen
- Pfadanzahl wächst bei unbestimmten Wiederholungen explosiv
- ein Teil der konstruierbaren Pfade ist in der Regel nicht ausführbar, da sich Bedingungen gegenseitig ausschließen können
- gutes Testverfahren
- besser nur in Kombination mit anderen Verfahren

In unserem Programmbeispiel: $n_s, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, [n_9, n_8]^{80}, \ldots$ führt zu falschen Ergebnissen, eventuell zum Programmabbruch!

Korrektur des Programms

```
....
while (!feof(file)) {
   int i = 0;
   line_cnt += 1;
   // Zeilen in Blöcken lesen
   while (l[i] != '\n') {
      for (; (i < N) && isblank(l[i]) && l[i] != '\n'; i++)
          char_cnt += 1;
      for (; (i < N) && !isblank(l[i]) && l[i] != '\n'; i++)
          char_cnt += 1;
      if (i == N) {
          fgets(l, N, file);
          i = 0;
      } else word_cnt += 1;
   }
   char_cnt += 1;  // count new-line character
   fgets(l, N, file);
}
....
```

angepasster Kontrollflussgraph



Weitere entdeckte Fehler:

- $n_s, \ldots, n_5, n_6, n_7, n_8, n_9, n_8, n_{10}, n_{12}, n_{14}, n_7, n_{15}, n_5, n_t$
  - $\rightarrow$ Zeile enthält ein Blank: es wird ein Wort gezählt!
    allg: Anzahl Wörter falsch, wenn Blank am Zeilenende
- $n_s, \ldots, n_5, n_6, n_7, n_8, n_{10}, [n_{11}, n_{10}]^N, n_{12}, n_{13}, n_7, n_8, n_9, n_8,$
  $n_{10}, [n_{11}, n_{10}]^x, n_{12}, n_{14}, n_7, n_{15}, n_5, n_t$
  - $\rightarrow$ falsche Anzahl Wörter, wenn Block endet bei Wortende und weitere Wörter in Zeile vorhanden
  - $\rightarrow$ falsche Anzahl Zeichen, da `fgets` am Blockende automatisch ein `'\0'` einfügt

Übung: Schreiben Sie eine korrekte Version des Programms `wordcount` und Testen Sie das Programm.