11. Qualitätssicherung

Ziele:
Qualitätssicherung: Klassifikation und häufigste Arten
Manuelle nichtaut. Prüfungen: Inspektionen, Walkthroughs, Reviews
Test: Begriffsbestimmung, auffindbare Fehler
Modul-, Teilsystemtest: White-Box-Methoden, Black-Box-Methoden
Integrationstest; Abnahmetest
Testplanung, Testbeendigung

Begriffe, Klassifikation, Bedeutung

Qualitätssicherung: Sprachgebrauch

SW-Qualitätssicherung (kurz: **QS**)**:** Teilgebiet der Softwaretechnik, das sich mit der Qualitätssicherung von →SW-Produkten (Dokumenten, Teilkonfig.), →SW_Bausteinen oder Herstellungsprozessen für →Software befaßt. Sie erfolgt in der Regel fremdgesteuert, d.h. unabhängig von den Entwicklungstätigkeiten. Die wesentlichen Tätigkeiten sind →Q_vorgeben, Q_steuern und →Q_prüfen.

Qualitäts-Eigenschaft: Eigenschaft, die zur Unterscheidung von →Produkten, →Bausteinen oder Herstellungsprozessen in qualitativer (subjektiver) oder quantitativer (meßbarer) Hinsicht herangezogen werden kann.

Beispiele für Q_Eigenschaften von Software sind Korrektheit, Zuverlässigkeit, Benutzungs- und Wartungsfreundlichkeit etc.

Qualitäts-Merkmal: →Qualitäts-Eigenschaften, die im konkreten Fall als relevant erachtet und zur Unterscheidung herangezogen wird.

Qualitäts-Anforderungen: → gewichtete Anforderungen an die Ausprägung von →Qualitäts-Merkmalen.

B-Qualität (eines →Produkts oder →Bausteins): Grad der Erfüllung vorgegebener → Qualitäts-Anforderungen an das Produkt oder den Baustein.

Vgl. dazu auch die allgemeine Erklärung für Qualität in Kap. 2.

Qualität_vorgeben: Tätigkeit der →Qualitätssicherung, in der die →Qualitäts-Anforderungen festgelegt werden.

Dies geschieht durch Auswahl der Q_Merkmale aus den Q_Eigenschaften, Festlegung der Relationen und Gewichtungen zwischen ihnen sowie Angabe der erforderten Ausprägungen.

Qualität_organisieren: Tätigkeit der →Qualitätssicherung, die sich mit der Planung, Durchführung, Überwachung und Auswertung der QS-Maßnahmen befaßt.

Hierzu gehört der Vergleich von Q_Prüfergebnissen mit den vorgegebenen Q_Anforderungen, die Bestimmung des Erfülltheitsgrades (nicht erfüllt, erfüllt, übererfüllt) einzelner Q_Merkmale sowie die Entscheidung über das Ergebnis der gesamten QS.

Qualität_prüfen: Durchführung der im Rahmen des →Q_vorgebens und Q_steuerns festgelegten Prüfmaßnahmen.

Hierhin gehört insbesondere die Bestimmung der tatsächlich vorliegenden Ausprägungen der Q_Merkmale.

Nach Hesse et al.

Bemerkungen

- vgl. allg. Qualitätseigenschaften von Software: früher behandelt

- Modell für ein Q-Merkmal: Annahmen, Einschränkungen, Vereinfachungen,
  welche quantitativen Kenngrößen überhaupt, in welcher Form ein best. Q-Merkmal beeinflussen

- augenblicklicher Stand: subjektive Bewertung

- zukünftig: Software-Meßtechnik
  basierend auf quantitativen Kenngrößen und ihren Maßen: Software-Metriken

Klassifikation der Qualitätsprüfung

(A) Gegenstand der Prüfung

Prüfungen können sich mindestens auf die folgenden drei Arten von Prüfobjekten beziehen:

(1) Prüfung der Entwicklung, d.h. normalerweise Prüfung von *Arbeitsergebnissen* gegenüber ihren Vorgaben bzw. den gerade vorhergegangenen Schritten.

Beispiele: Prüfung einer Spezifikation gegenüber den gestellten Anforderungen, Prüfung einer Konstruktion gegenüber der entsprechenden Spezifikation.

(2) Prüfung von realisierten *Bausteinen* gegenüber ihrer Spezifikation und Konstruktion gemäß den gestellten Anforderungen an die B_Qualität.

Diese Art der Prüfung entspricht i. W. dem „verification“ in der amerikanischen Literatur („Are we building the product right?“ [4]).

(3) Prüfung der *Anwendbarkeit* von Problemlösungen in ihrer Umwelt, i.a. repräsentiert durch die Benutzer. Diese Art der Prüfung beginnt bereits bei den Anforderungen an das künftige System und endet bei der Prüfung seines Einsatzes.

Diese Art Prüfung entspricht i. W. dem „validation“ in der amerikanischen Literatur („Are we building the right product?“ [4]).

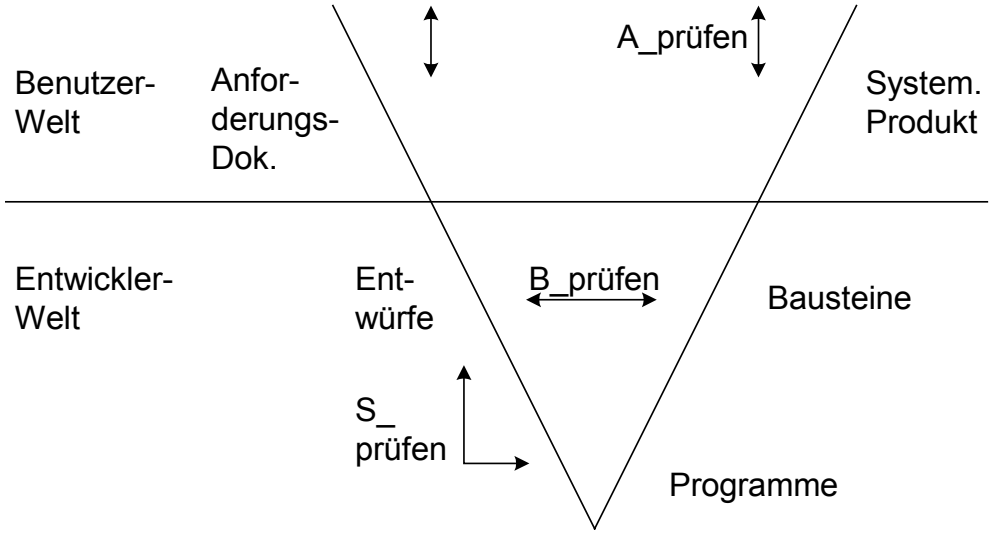


Abb. 2. Verschiedene Formen der Prüfung im SW-Lebenszyklus (Darstellung nach [4])

(B) Beschreibungsform des Prüf- und Bezugsobjekts

Jede Prüfung erfolgt relativ zu einer Vorgabe, dem *Bezugsobjekt*. Prüf- und Bezugsobjekte lassen sich hinsichtlich ihrer Beschreibungsform als (1) *formal*. (2) *teilweise formal* oder (3) *informell* klassifizieren.

Häufig sind Prüf- und Bezugsobjekt unterschiedlich formal beschrieben. Die Möglichkeiten der Prüfung hängen selbstverständlich von beiden Beschreibungen ab.

(C) Vorgehensweise

Hier läßt sich unterscheiden in (1) *formales* Beweisen mit Hilfe eines logischen Kalküls, (2) Ausführung von *Experimenten* und (3) menschliche *Begutachtung*.

Welche Vorgehensweise gewählt werden kann, hängt vom Formalitätsgrad des Prüf- und Bezugsobjekts ab.

(D) Ablauf der Prüfung

Je nachdem, wie der Prüfvorgang am Prüfobjekt abläuft, unterscheiden wir zwischen (1) *statischer* Prüfung (in der Reihenfolge der Aufschreibung des Prüfobjekts) und (2) *dynamischer* Prüfung (in der Reihenfolge der Ausführung des Prüfobjekts).

(E) Vollständigkeit der Prüfung

Hier wird danach unterschieden, ob der Prüfbereich (1) *vollständig* oder (2) *stichprobenartig* abgedeckt wird.
Die Vollständigkeit läßt sich i.a. nur durch formales Beweisen erreichen.

(F) Rechnerunterstützung

Hinsichtlich der Rechnerunterstützung unterscheiden wir zwischen (1) *automatisierten*, (2) *teilweise automatisierten* und (3) *nichtautomatisierten* Prüfungen.

Die Möglichkeit der Rechnerunterstützung hängt u.a. davon ab, ob die Anwendung der Technik algorithmisierbar ist.

(G) Verantwortung/Regie für die Überprüfung

Hinsichtlich der Personen bzw. Institutionen, die die Verantwortung für die Prüfung tragen und die die Regie dabei übernehmen (ohne notwendigerweise selbst bei den Prüfmaßnahmen beteiligt zu sein), unterscheiden wir zwischen (1) *selbstgesteuerten* und (2) *fremdgesteuerten* Prüfungen. Selbstgesteuert heißt dabei unter der Verantwortung und Regie der direkt an der Entwicklung beteiligten Personen, fremdgesteuert unter der Verantwortung und Regie von nicht direkt an der Entwicklung beteiligten Personen.

Dieses Kriterium ist ausschlaggebend für die Unterscheidung der Tätigkeiten der Entwicklungsprüfung (selbstgesteuert, vgl. Kap. 4) von denen der Qualitätssicherung (fremdgesteuert, vgl. 5. 2).

Prüfungen, die von an der Entwicklung beteiligten Personen an andere delegiert werde, sind in diesem Sinne als selbstgesteuert zu bewerten, da das Interesse an der Prüfung bei den an Entwicklung beteiligten Personen liegt.

(H) Zeitpunkt der Qualitätssicherung

- nachher (üblich): analytisch
  - Compilerüberprüfungen
  - konventionelles Testen
  - Programmverifikation
  - symb. Testen
  - Reviews, Inspektionen, Walkthroughs

  } für Programme, Programmstücke, Dokumente oder deren Ergebnisse

- vorab, begleitend (durch Werkzeuge unterstützt): konstruktiv
  - syntaxgest. Editoren
  - Bewerten vor Realisierungen (Simulation)
  - erstellungsbegleitendes Testen (inkr. Compiler, Debugger)
  - Vorgabe von Strukturen, Richtlinien für Prozeß, Teilprodukte

Häufigste Arten von Prüftechniken

Im folgenden benutzen wir die eingeführten Kriterien zur Erklärung einiger gebräuchlicher Begriffe für Prüftechniken. Dabei werden wir nur die jeweils relevanten Kriterien aufführen, nicht aufgeführte Kriterien betrachten wir für den jeweiligen Begriff als irrelevant.

Test: jede (i.a. stichprobenartige) Ausführung von Experimenten zum Zwecke des →Prüfens eines →Produkts oder →Bausteins.

Entscheidend ist hier die Vorgehensweise (Kriterium C) und damit eng verbunden die Vollständigkeit (oder genauer: Nicht-Vollständigkeit, Kriterium E). Normalerweise wird mit Test außerdem ein dynamischer Ablauf (D) und eine Rechnerunterstützung (F) assoziiert.

Verifikation: jede formale Prüfung mit Hilfe eines logischen Kalküls.

Hauptkriterium ist auch hier die Vorgehensweise (C). Verifikation setzt formal beschriebene Prüf- und Bezugsobjekte voraus (B), läuft statisch ab (D) und zielt auf vollständige Prüfung (E). Damit kommt sie i.a. nur für die S_ und B_Prüfung in Frage (A).

Review, Inspektion, Walkthrough: spezielle Formen nicht-automatischer Prüfungen durch menschliche Begutachtung.

Die entscheidenden Kriterien sind hier die Vorgehensweise (C) und die (noch?) fehlende Rechnerunterstützung (F). (Rechnerunterstützte Inspektionen sind denkbar.) Dabei sind Reviews und Inspektionen vorwiegend statische, Walkthroughs dynamische Prüfungen (D). Von Reviews spricht man hauptsächlich bei A_ und S_Prüfungen, während Inspektionen und Walkthroughs sich meistens auf B_Prüfungen beziehen. (Kriterium A)

Bedeutung der Qualitätssicherung — Kosten von Fehlern

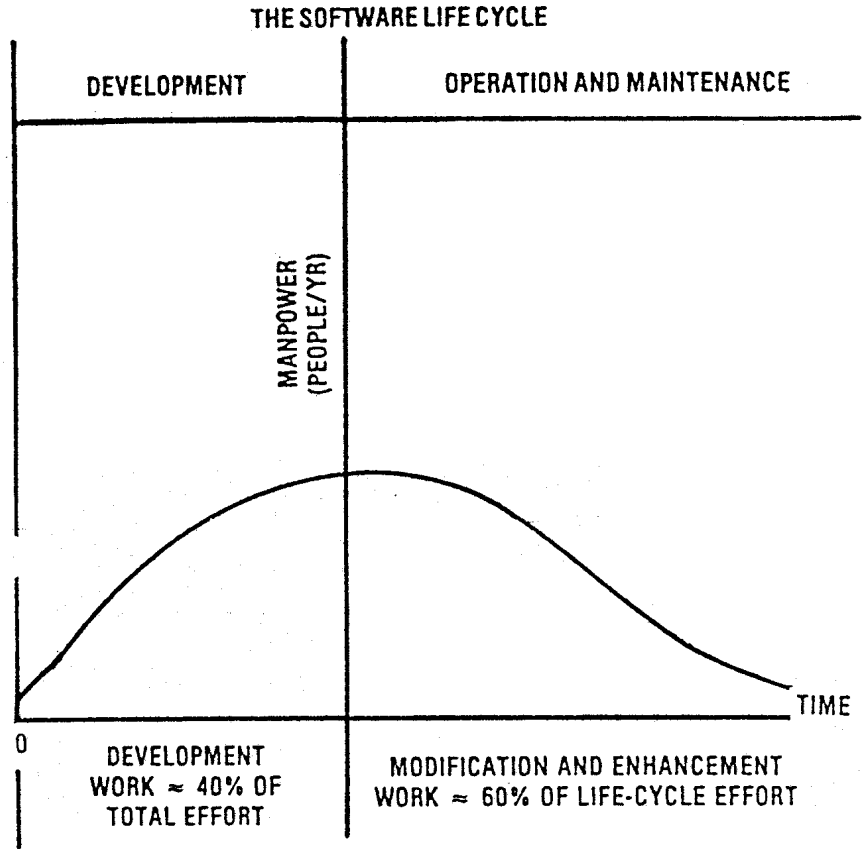
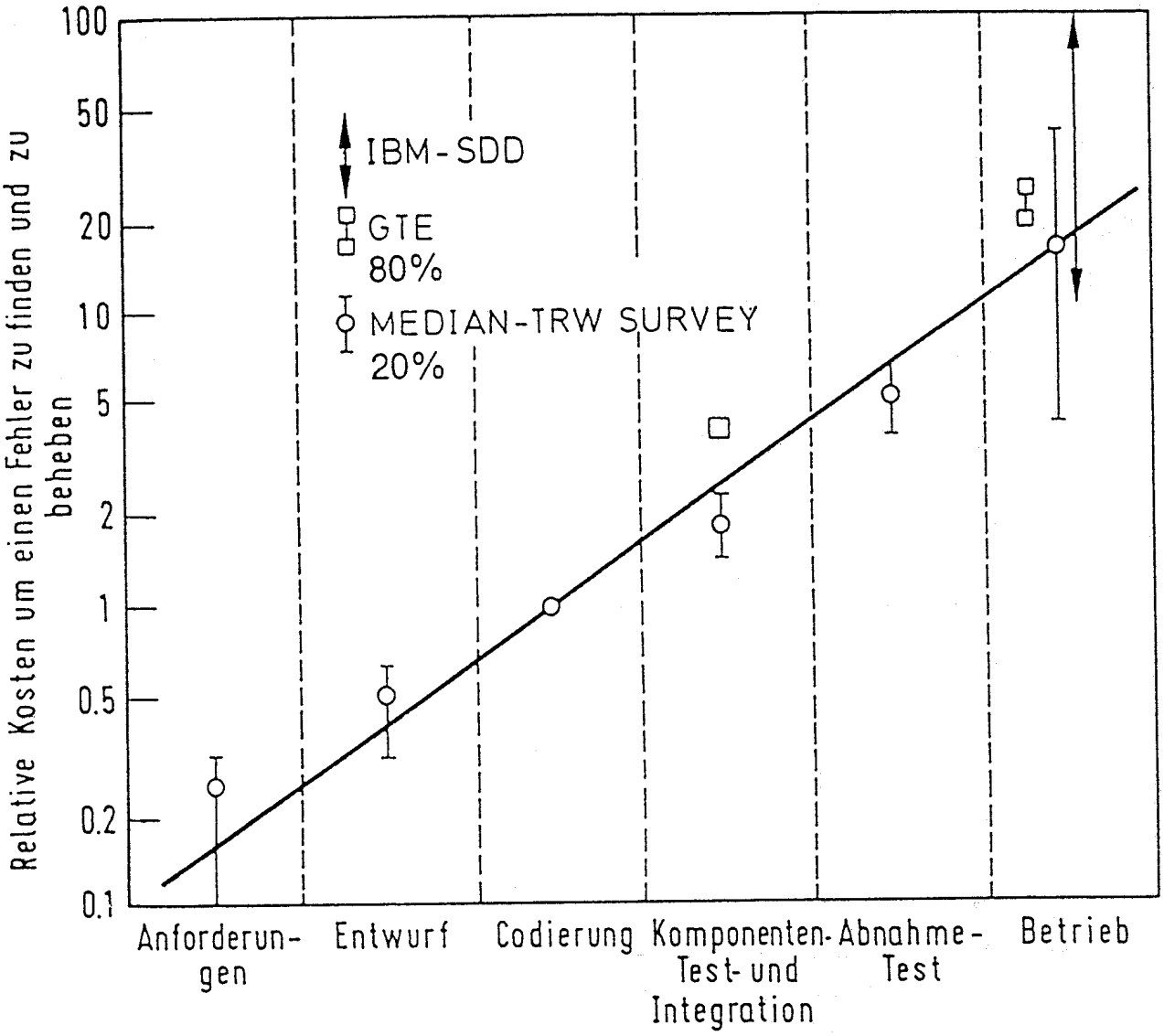
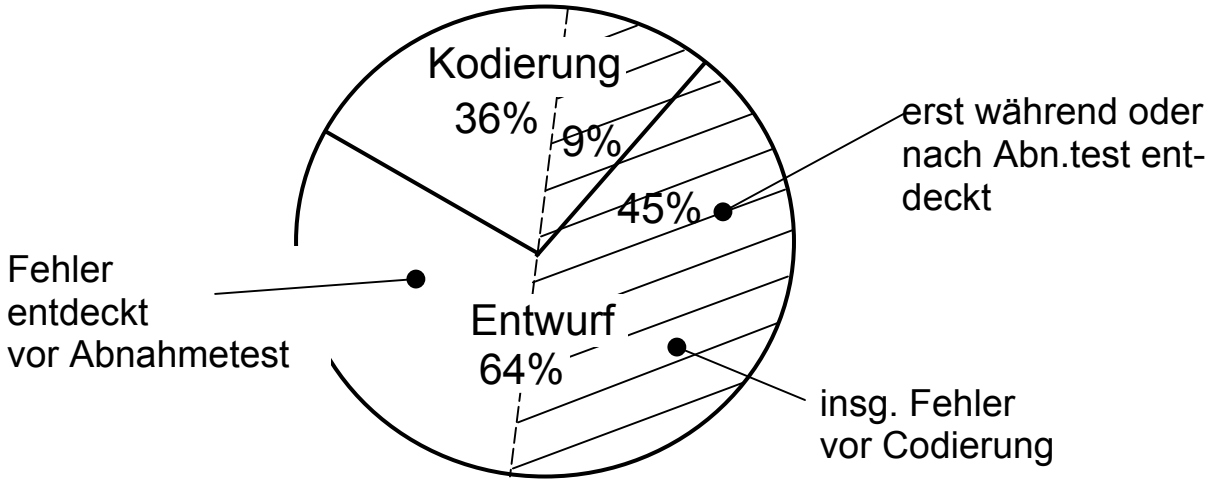


Figure 3. An example of a Rayleigh c
representing the profile of
human effort over time.


Relative Kosten um einen Fehler zu finden und zu beheben
100
50
20
10
5
2
1
0.5
0.2
0.1
IBM-SDD
GTE
80%
MEDIAN-TRW SURVEY
20%
Anforderungen
Entwurf
Codierung
Komponenten-Test- und Integration
Abnahme-Test
Betrieb
Phase in der ein Fehler entdeckt wird

Fehlerhäufigkeit, Zeitpunkt des Auftretens

Aussage Boehm 75



Aussage Endres 75

46% der Fehler wegen Verständnisproblemen, unzul. Entwurfsmeth, in Entwurfsphase

Aussage Alberts 76

46% - 64% Entwurfsfehler

21% - 38% Implementierungsfehler

Fehlerarten (in einem Entwicklungsprojekt)

a) bzgl. vorangeg. Dokumente im Lebenszyklus

- genügt nicht den Wünschen des Auftraggebers (Transformationsprobl.)
  fehlerhafte Istanalyse
  anderes Sollkonzept als Auftraggeber im Hinterkopf hat
- genügt nicht der Anforderungsdefinition
  fehlerhafte Entwurfsspezifikation bzgl. Anf. def.
- Falsche Implementation
  genügt nicht der Entwurfsspez.

b) wann werden sie erkannt, wann treten sie auf

- vor Spezifikation (Anf.-, Entw.-)
- vor Programmerstellung
- vor der Laufzeit
- während der Laufzeit
- im Feld

c) sie sind

- reproduzierbar
- treten nur in best. Situationen auf (z.B. Abarbeitungszust. von Tasks, Lastprofil der Maschine)
- sporadisch (z.B. übertragungsfehler)
- sind abhängig von der Implementation eines Programmsystems (erroneous programs, versch. Varianten einer Sprache durch versch. Compiler)

d) innerhalb einer Dokumentenklasse

- bzgl. k. f. Syntax
- bzgl. k. s. Syntax (stat. Sem.)

} e. Dokumentenklasse

- dyn. Semantik (bei ausführb. Dokumenten)
- Pragmatik (z.B. Implementierungstechnik aufgrund Eigensch. der Basismaschine nicht durchführbar)
- dokumentübergreifend

Manuelle (nichtautom.) Prüfungen

Manuelles "Testen" (Human Testing):

- "Manuelle" Überprüfung im Gegensatz zu (manuell oder autom.) mit Hilfe der Maschine

- Reviews prinzipiell für Überprüfung jedes Zwischenergebnisses im Softwarelebenszyklus

Inspektionen und Walkthroughs

- Schreibtischtest vom Entwickler selbst / Insp.u. Walkthroughs durch "Test"team
- "Test"gruppe 3-4 Personen einschl. Autor
- 30-70% der durch Test auffindbaren Fehler
- eher "logische" Fehler (high-level Fehler, Entwurfsfehler)
- komplementär zu Testen mit Maschine
- insb. auch einsetzbar bei Programm(system)modifikationen
- Wirksamkeit bisher wissenschaftlich ungeklärt

Codeinspektionen

– Team:

- Moderator (Verteilung der Unterlagen, Zeitplanung, Sitzungsleitung, Fehlerprotokollierung, Überwachung der Fehlerbehebung): entspricht Qualitätsingenieur
- Programmautor
- Programmentwerfer, falls ≠ Autor
- Testspezialist

– Vorgehensweise:

- Moderator verteilt Quellcode und entspr. Teil der Spezifikation einige Tage vor Inspektionssitzung
- Teilnehmer lesen Unterlagen vor Sitzung
- Sitzung (90-120 Min.)
  - Programmierer erklärt Programmstruktur bis auf Anweisungsebene (dabei entdeckt er viele seiner Fehler selbst)
  - Programm mithilfe von Checklisten analysiert
  - Moderator achtet darauf, daß keine Fehlerbehebung diskutiert wird
- nach Sitzung: Programmierer erhält Liste gef. Fehler
  - ggf. erneute Codeinsp. bei (vielen) Fehlern mit großer Änderung
  - ggf. weitere Codeinspektionen bei großen Programmen

– Prinzipien/weitere Vorteile
  - Programmierer muß versuchen „neutrale“ Haltung einzunehmen (ego-less)
  - Ergebnis der Inspektion vertraulich
  - Programmierer lernt aus seinen Fehlern erhält Rückkopplung über Programmierstil für andere Teilnehmer gilt das gleiche
– Fehlerprüfliste:

Datenreferenz

1. Werden Variable angesprochen, die nicht initialisiert wurden?
2. Befinden sich die Indizes innerhalb der Grenzen?
3. Werden nichtganzzahlige Indizes verwendet?
4. Hat der Zeiger einer noch gültigen Wert? (dangling reference)
5. Sind die Attribute bei einer Redefinition korrekt?
6. Stimmen Satz- und Strukturattribute überein?
7. Passen die Adressen der Bitketten? Werden Bitkettenattribute übergeben?
8. Sind die Attribute für den „based“ Speicherbereich korrekt?
9. Stimmen die Datenstrukturen in verschiedenen Prozeduren überein?
10. Werden die Indexgrenzen einer Kette überschritten?
11. Gibt es „off by one“ Fehler?

Berechnungen

1. Werden Rechnungen mit nichtarithmetischen Variablen durchgeführt?
2. Gibt es Rechnungen mit verschiedenen Datentypen?
3. Gibt es Rechnungen mit Variablen verschiedener Länge?
4. Ist die Größe der Zielvariablen kleiner als die des zugewiesenen Werts?
5. Gibt es einen Über- oder Unterlauf im Zwischenergebnis?
6. Division durch Null?
7. Treten Ungenauigkeiten bei der Dualdarstellung auf?
8. Liegt der Variablenwert innerhalb eines sinnvollen Bereichs?
9. Ist die Priorität der Operatoren richtig verstanden worden?
10. Ist die Division mit ganzzahligen Werten korrekt?

Datendeklaration

1. Sind alle Variablen erklärt?
2. Sind die Standardattribute verstanden worden?
3. Sind Felder und Ketten richtig initialisiert?
4. Sind die korrekten Längen, Typen und Speicherklassen zugewiesen?
5. Paßt die Initialisierung zu der Speicherklasse?
6. Gibt es Variable mit ähnlichen Namen?

Vergleich

1. Wird ein Vergleich zwischen inkonsisten Variablen durchgeführt?
2. Werden Daten verschiedenen Typs miteinander vergleichen?
3. Sind die Vergleichsoperatoren korrekt?
4. Sind die Boole’schen Ausdrücke korrekt?
5. Sind Vergleichs- und Boole’sche Ausdrücke vermischt?
6. Gibt es Vergleiche mit Brüchen zur Basis 2?
7. Ist die Priorität der Operatoren richtig verstanden worden?
8. Ist die Compilerdarstellung Boole’scher Ausdrücke richtig verstanden worden?

Abb. 3.1 Zusammenfassung der Fehlerprüfliste. Teil 1

Steuerfluß

1. Werden bei Mehrfachentscheidungen alle Sprungmöglichkeiten berücksichtigt?
2. Wird jede Schleife beendet?
3. Wird das Programm beendet?
4. Wird eine Schleife aufgrund der Eingangsbedingung nicht ausgeführt?
5. Sind mögliche Umgehungen von Schleifen korrekt?
6. Gibt es bei einer Iteration „off by one“ Fehler?
7. Gehören die DO/END-anweisungen zusammen?
8. Gibt es unvollständige Entscheidungen?

Schnittstellen

1. Stimmt die Anzahl der Eingabeparameter mit der Anzahl der Argumente überein?
2. Stimmen die Parameter- und Argumentattribute überein?
3. Stimmen die Einheiten der Parameter und der Argumente überein?
4. Stimmt die Anzahl der an den gerufenen Modul übergebenen Argumente mit der Anzahl der Parameter überein?
5. Stimmen die Attribute der an den gerufenen Modul übergebenen Argumente mit den Attributen der Parameter überein?
6. Entsprechen die Einheiten der dem gerufenen Modul übergebenen Argumente den Einheiten der Parameter?
7. Sind Anzahl, Attribute und Reihenfolge der Argumente für „eingebaute“ Funktionen korrekt?
8. Gibt es Referenzen auf Parameter, die nicht mit dem aktuellen Entrypoint assoziiert sind?
9. Werden Argumente verändert, die nur als Input dienen sollten?
10. Ist die Definition globaler Variabler über alle Module konsistent?
11. Werden Konstante als Argumente übergeben?

Ein-/Ausgabe

1. Sind die Dateiattribute korrekt?
2. Ist das OPEN-statement korrekt?
3. Passen die Formatspezifikationen zu den E/A-anweisungen?
4. Passen die Puffergrößen zu den Satzgrößen?
5. Wird die Datei vor Benutzung eröffnet?
6. Werden Dateiende-Bedingungen behandelt?
7. Werden E/A-Fehler behandelt?
8. Gibt es Textfehler in der Ausgabeinformation?

Andere Prüfungen

1. Gibt es nicht angesprochene Variable in der Crossreference-Liste?
2. Bringt die Attributliste das, was man erwartet?
3. Gibt es warnende oder informative Meldungen?
4. Werden die Eingabedaten auf Gültigkeit überprüft?
5. Fehlen Funktionen?

Abb. 3.2 Zusammenfassung der Fehlerprüfliste. Teil 2

Walkthroughs

- viel Ähnlichkeit zu Codeinspektion:
  - Sitzung 1-2 Std.
  - Team 3-5 Mitgl.: Moderator
    Sekretär
    Tester
    Programmierer
    weiter evtl.
    erfahrener Programmierer
    Programmiersprachenexperte
    neuer Mitarbeiter (unvoreing.)
    Programmpfleger
    Mitarbeiter aus anderem Projekt
    Anderes Mitgl. aus Team des Autors
- Zielsetzung anders: Teilnehmer spielen Computer nach Testdaten des Testers (wenige Testdaten)
- Testfälle dienen dem Einstieg in die Diskussion: meisten Fehler in Diskussion über Testfälle entdeckt
- Nachbearbeitung wie bei Codeinspektion

Reviews

Schreibtischtest

- Entwickler prüft selbst (oder 2 prüfen sich gegenseitig)
- Ist Einmann-Inspektion und Einmann-Walkthrough
- Beurteilung des Programms und Verbesserung

Peer Ratings (für kommerzielle Umgebung, bzw. Ausbildung)

- Hat nichts mit Testen zu tun, aber Bezug zur Codeinspektion.
- Beurteilung eines Programms eines i.a. unbekannten Autors: Wartungsfreundlichkeit, Erweiterbarkeit, Verständlichkeit etc.
- Ungef. 6 Teilnehmer (mit vergleichbarer Erfahrung)
  - Jeder gibt „gutes“ und „weniger gutes“ Programm ab
  - Jeder Teilnehmer bekommt 2 „gute“ und 2 „weniger gute“ Programme unbekannter Autoren zur Beurteilung
  - für jedes Programm ca. 30 Min.
  - Beurteilung:

    verständlich?

    Entwurf/Modulentwurf verständlich?

    leicht modifizierbar

    etc.

    plus allg. Kommentare
  - anonyme Beurteilung zurück: Selbsteinschätzung

Allgemeines zu Test

Testarten

- Modultest (Teil der Implementierung, PIK)
  „Korrektheit“ der Implementation gegenüber Spezifikation,
  Effizienzgesichtspunkte nur, falls in Anf.def. festgelegt Test-
  umgebung, ggs. Teststummel

- Integrationstest

  Testen eines Teilsystems gemäß Spezifikation oder Anf.def.

  Testen des Gesamtsystems gemäß Anf.def.

  „Korrektheit“ als auch Leistungsparameter

  entdeckt Fehler gegenüber Spezifikation

  gegenüber Anf.def.

  Testdatenentwicklung für Gesamtsystem beginnt in Problem-
  analyse

  Überprüfung (Nachprüfung) durch Qualitätssicherungs-
  abteilung

- Installationstest

  falls Entwicklungsmaschine ≠ Zielmaschine

- Abnahmetest

  Unter Regie des Auftraggebers

  Test gegenüber Anf.def.

  Testdaten nur aus Anf.def.

Testen-Begriffsbestimmung

– falsche Def.

„Testen ist der Prozeß, der zeigen soll, daß keine Fehler existieren“

„Testen soll zeigen, daß ein Programm korrekt ist“

„Testen ist ein Prozeß, der das Vertrauen erzeugt, daß ein Programm das tut, was es soll“

– richtig

„Mit Testen kann man nur die Anwesenheit, aber nicht die Abwesenheit von Fehlern zeigen“ (Dijkstra)

– besser

„Testen ist der Prozeß, der ein (fertiges/teilweise fertiges) Programm mit der Absicht ausführt, Fehler zu finden“

- zielgerichtete Tätigkeit zum Auffinden
- ökonomische Tätigkeit: Werterhöhung durch Verbesserung ≥ Aufwand für Testen
- Fehler_1: Programm tut etwas, was es nicht soll: Test durch Eingabe „aller“ Sonder- u. Fehlerfälle
- Fehler_2: Programm tut das nicht, was es soll: Test durch einige (repräsentative) Eingaben, die die Funktionen testen
- Testfälle sind nie vollständig (s.u.):
  Hoffnung, mit Testdatensatz „alle“ Fehler aufgef. zu haben

Eingangsbeispiel zu ”Auffinden aller Sonderfälle, einige der Normalfälle“

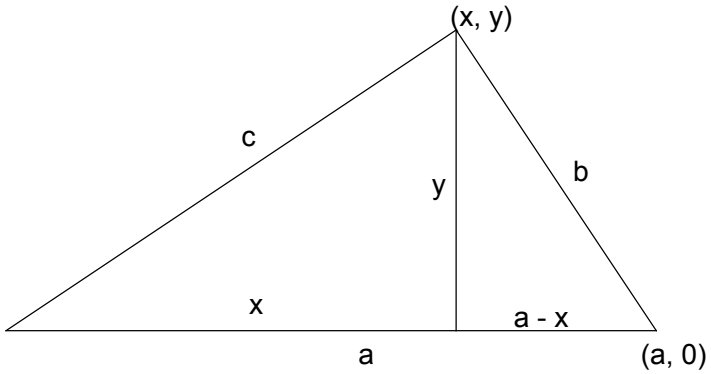
- Lies drei ganzz. Werte

  werden als Längen von Dreiecksseiten interpretiert
- Meldung: ungleichseitig

  gleichschenklig

  gleichseitig

  Fehler



Testdaten (intuitiver Ansatz)

1) Zulässiges ungleichs. Dreieck (z.B. (4, 3, 2))
2) Werte sind keine Seitenlängen (z.B. (1, 2, 3): a+b=c)
3) Längste Seite gemäß 2) permutieren:

   (1, 2, 3), (3, 2, 1), (2, 1, 3)
4) Werte sind keine Seitenlängen a+b < c: (1, 2, 4)
5) längste Seite gemäß 4) permutiert:

   (1, 2, 4), (4, 2, 1), (2, 4, 1)
6) zulässige gleichschenkliges Dreieck (3, 3, 4)
7) zulässige Werte gemäß 6) permutieren:

   (3, 3, 4), (4, 3, 3), (3, 4, 3)
8) unzulässiges gleichschenkliges Dreieck:

   a+b = c (2, 2, 4) ggf. permutieren

   a+b < c (2, 2, 5) ggf. permutieren
9) zulässiges gleichseitiges Dreieck (3, 3, 3)

10) eine Seite 0: (0, 2, 5) ggf. permutieren

11) zwei Seiten 0: (0, 0, 6) ggf. permutieren

12) drei Seiten 0: (0, 0, 0)

13) eine Seite negativ: (-3, 1, 5) ggf. permutieren
ggfs. 2 neg. Werte ggf. permutieren
ggf. drei neg. Werte ggf. permutieren

14) nichtganzz. Werte (3, 14, 2.3) ggf. permutieren
ggf. 2 nichtganzz. Werte ggf. permutieren
ggf. 3 nichtganzz. Werte ggf. permutieren

15) falsche Zahl von Eingabewerten (3, 4, 2, 5), (3, 4)

16) Wurden überall bei 1-15 passende Ausgaben gemacht?

Durch Testen können nicht alle Fehler gefunden werden

a) Blackbox-Test (datengetriebenes Testen, Ein-/Ausg. -Testen)

Testfälle ausschließl. aus der Spezifikation
Internstruktur des Programms nicht betrachten

- Z. B. für die gleichseitigen Dreiecke (obiges Beispiel)
  (1, 1, 1), (2, 2, 2), (3, 3, 3), … (intmax, intmax, intmax)
  ferner alle Sonder-/Fehlerfälle
  ⇒"astron. Zahl" von Testdaten (math. endl.)

- Alle mögl. Eingaben zum Test eines Cobol-Compilers
  Zeichenketten bel. Länge, die synt. gültige (synt. ungültige) Cobol-Programme darstellen
  Länge nur begrenzt durch max. Dateilänge (soft- oder hardwaremässige Begr.)
  ⇒"(astr. Zahl)²" von Testfällen (math. endl.)

- Programm mit Gedächtnis (Betriebssystem, Anwendungsprogr. für Datenbankanwendung z.B. Flugreservierung)
  Test nicht nur bezügl. aller zulässigen (unzul.) Eingaben, sondern deren Sequenzen
  ⇒"(astr. Zahl)³" von Testfällen (math. endl.)

somit: Aus ökon. Gründen oder grundsätzl. Gründen (kein Tester erlebt Testende) kann Fehlerfreiheit durch Testen nicht garantiert werden

Auswahl der Testdaten unter ökon. Gesichtspunkten

b) Whitebox-Test

Testdaten unter Betrachtung der Programmstruktur (Spezifikation nat. auch!)

– Jede Anwendung einmal genügt nicht (vgl. unten)

– Jeder Programmpfad einmal ausgeführt: hinreichend aber nicht ausführbar:

Schleife max 100 mal im Inneren eine bedingte Anweisung.

Anzahl Programmpfade:

$2+2^2+2^3+ \cdots +2^{100}$

1 mal durchlaufen — 100 mal

$$\frac{2(2^{101}-1)}{2-1} = 2^{102}-2 >$$

$$\underbrace{2^{10} * 2^{10} * \cdots * 2^{10}}_{10 \text{ mal}} \approx$$

$$= 10^3 \cdot \ldots \cdot 10^3 = 10^{30}$$

Jede 1/1000 sec 1 Testfall:
Brauchen 10^{19} Jahre!

Selbst bei vollst. Pfadtesten Fehlermöglichkeiten:

– Fehlende Programmpfade werden nicht entdeckt

– Keine datensensiblen Fehler werden entdeckt:

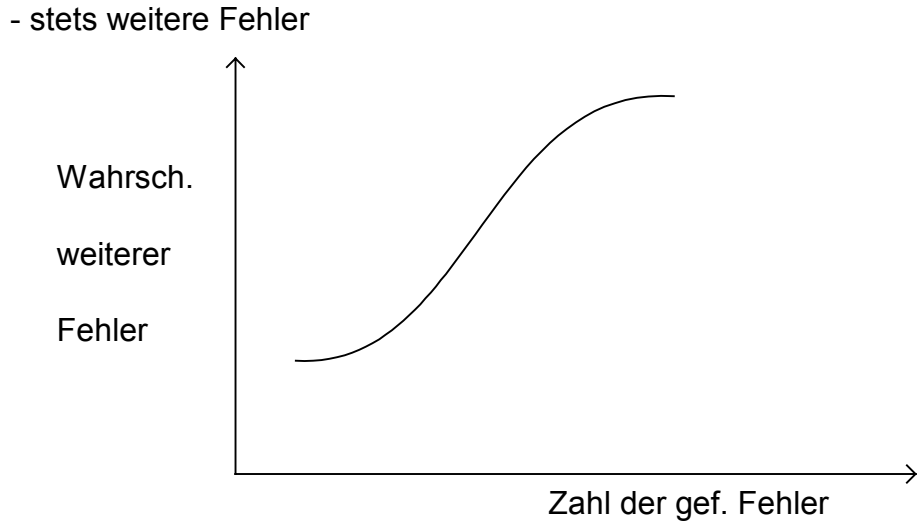
If (A-B) < EPS then ...

Testprinzipien

- Def. der erwarteten Resultate zu Eingabewerten (Ausgabewerte oder Fehlermeldungen) vor der Testdurchführung: hinterher mechanisch vergleichen auf Unstimmigkeiten

- Tester ≠ Modulentwickler
  - Psychologischer Grund: Testen destruktiv!
  - Nichtübereinst. mit Anforderungsdef. oder Spezifikation kann nicht von dem entdeckt werden, der bisher falsch verstanden hat
  - Fehlerbeseitigung von Modulentwickler, nicht Tester
  - Teilsystem nicht von Entwicklungsteam allein testen lassen

- gründliche Überprüfung der Testergebnisse (besser automatisch)
- Konzentration nicht nur auf zulässige Eingaben (Feststellung, daß Programm das tut, was es soll)
- „Alle“ Sonderfälle und Fehlerfälle (Feststellung, ob Programm etwas tut, was es soll)
- Testdaten reproduzierbar, wiederverwendbar (z.B. kein Verändern von Variablenwerten im Testdialog)



- beobachtetes, aber nicht erklärtes Phänomen
  Bsp. IBM 370 Betriebssystem: 47% der von Benutzern gefundenen Fehler in 4% des Codes:
  - Fehler treten lokal gehäuft auf
  - einige Abschnitte sind fehleranfälliger

$\Rightarrow$ Suche nach Fehlern in fehlerträchtigen Teilen

- Testen ist komplizierte u. aufwendige Aufgabe

- genauer
  - ”manuelles“ “Testen” (Überprüfung), s. o.
  - Modultest (s. u.)
  - Integrationstest (s. u.)

Modul- bzw. Teilsystemtest

Testfallentwurf

- vollständig unmöglich
  ökon. Rahmenbed. ⇒ wohlüberlegte Strategie/Methode
- Welche Teilmenge aller denkbaren Testfälle bietet größte Wahrscheinlichkeit, möglichst viele Fehler zu finden?
- zufällige Auswahl schlecht
- Kombination aus Blackbox- und Whitebox-Testen:

Whitebox:	Blackbox:
Überdeckung von	Äquivalenzklassen
Anweisungen	Grenzwertanalyse
Entscheidungen	Ursache-Wirkungsgraph
Bedingungen	Fehlereraten (error guessing)
Entscheidungen/Bed.	
Mehrfachbedingungen	

- am besten: Kombination der Methoden: Blackbox erweitert um Whitebox
- im folgenden zuerst Whitebox-Methoden

Testüberdeckungen (White-Box-Methoden)

- vollständiges Pfadtesten undurchführbar s. o.

- schwaches Kriterium: Jede Anweisung mindestens 1 mal
  notwendig aber nicht hinreichend:

```
if x ≥ y then max := x else max := y end if;
if z ≥ max then max := z else max := y end if;
```

  richtig für $z \geq x \geq y$

  $y > x \wedge y > z$

  falsch für $x \geq y \wedge x > z$

- stärkeres Kriterium:
  Überdeckung aller Entscheidungen und Sprünge (decision and branch covering) d.h. aller Verzweigungen

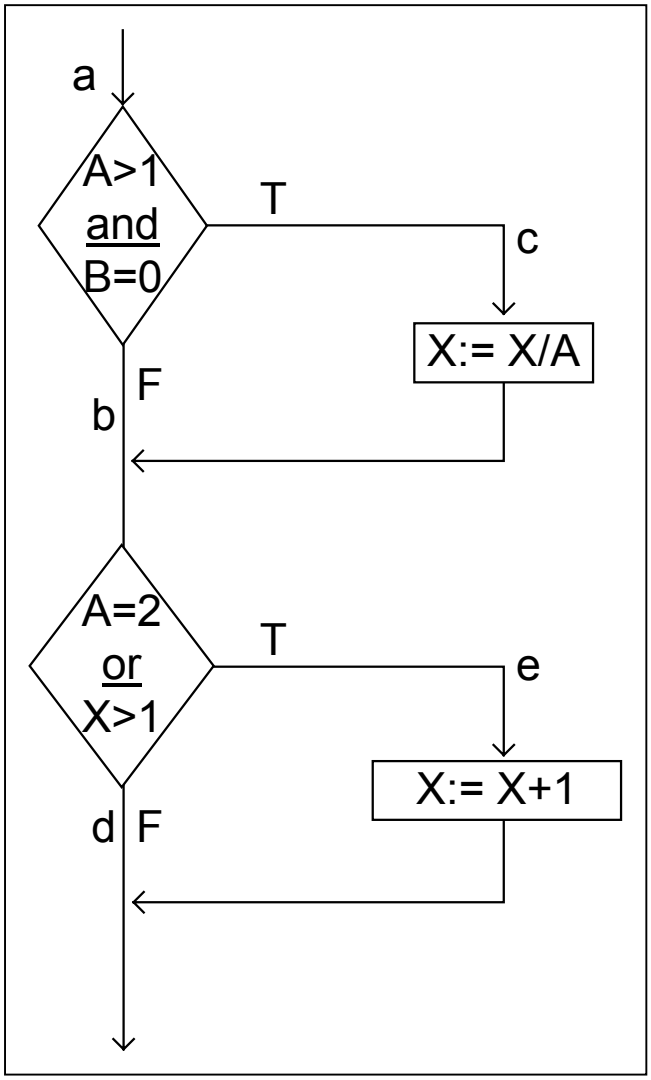
  dabei werden i.a. alle Anweisungen überdeckt

  Ausnahmen: Programm enthält keine Verzweigungen
  Unterprogramme mit Entrypoints
  on-conditions und zugehörige Programmteile

- Bedingungsüberdeckung (condition coverage)

```
if (A and B) then
   . . .
else
   . . .
end if;
```

⇒ Erfassen aller Mehrfachbedingungen



1) $A > 1, B = 0$
2) $A > 1, B \neq 0$
3) $A \leq 1, B = 0$
4) $A \leq 1, B \neq 0$
5) $A = 2, X > 1$
6) $A = 2, X \leq 1$
7) $A \neq 2, X > 1$
8) $A \neq 2, X \leq 1$

Testfälle:

A = 2, B = 0, X = 4 für 1), 5)
Pfad a c e

A = 2, B = 1, X = 1 für 2), 6)
Pfad a b e

A = 1, B = 0, X = 2 für 3), 7)
Pfad a b e

A = 1, B = 1, X = 1 für 4), 8)
Pfad a b d

Nicht alle „Pfade“ getestet: a c d

bei Mehrfachbed.:

alle mögl. Kombinationen von Bedingungen

plus alle Verzweigungen / Zusammenführungen

plus alle Anweisungen

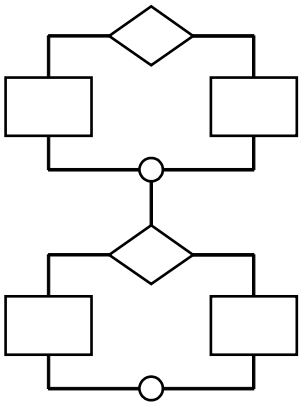
Überdeckungs- und Flußgraphenansatz

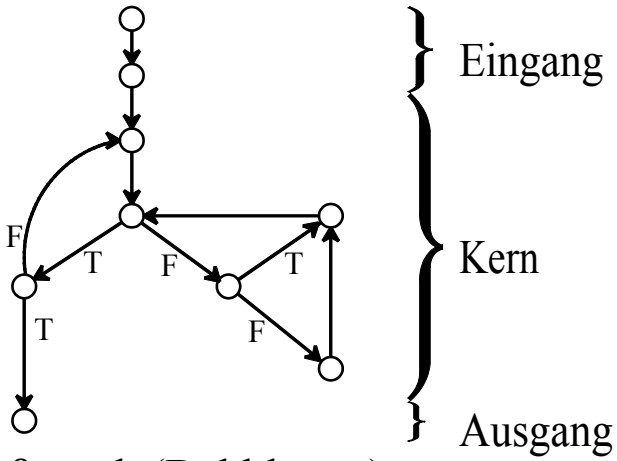
- Überdeckungen:

  (Anweisungen, Bedingungen, Mehrfachbedingungen)

  entspricht statischer Betrachtung des Flußgraphen:

  Überdeckung aller statischen Bestandteile



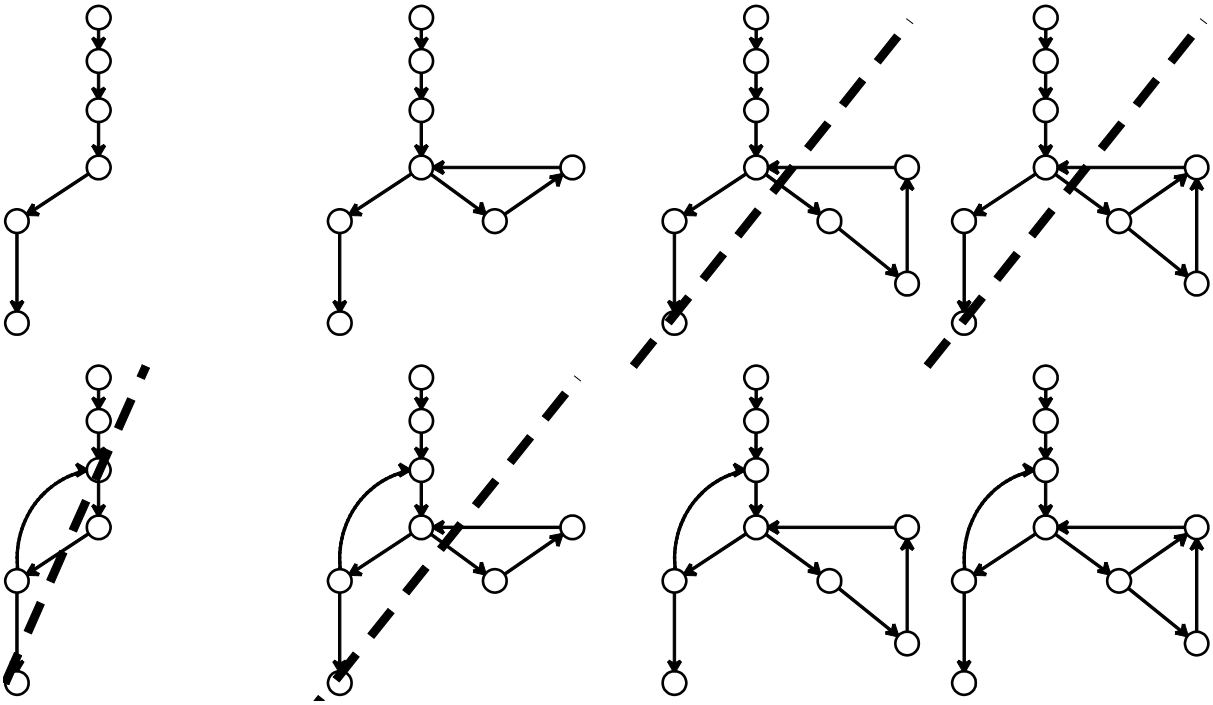


Flußdiagramm Flußgraph (Bubblesort)

- Flußgraphenbetrachtung:

  Betrachtung aller Programmpfade modulo

  Mehrfachdurchlauf bei Schleifen



Black-Box-Methoden

„Äquivalenzklassen“

- Suche Teilmenge der Testfälle mit der größten Wahrscheinl. Fehler zu finden:
  - Reduktion auf wenige
  - Überdeckung möglichst vieler Fälle

- $\Rightarrow$ Äquivalenzklasseneinteilung der Testfälle:
  jeder Repräsentant entdeckt gleichen Fehler (oder entdeckt best. Fehler nicht)

- obige Teilmenge = Menge von Repräsentanten

  Dreiecksbeisp.

  $K1 = (a, b, c) \in int^3 \quad mit \quad a+b>c; \qquad R_{K1}=(4, 3, 2)$

  $K2 = (a, b, c) \in int^3 \quad mit \quad a+b=c; \qquad R_{K2}=(5, 3, 2)$

  $K3 = (a, b, c) \in int^3 \quad mit \quad a+b<c; \qquad R_{K3}=(2, 1, 4)$

  .
  .
  .

- Vorgehen:
  - Bestimmung der Äquivalenzklassen
  - Definition der Testfälle

”Äquivalenzklassenbestimmung“

– gültige Äquivalenzklassen („reguläre“ Fälle, Grenzfälle)

– ungültige Äquivalenzklassen (fehlerhafte Eingabedaten)

Heuristik:

- Eingabe in best. Bereich [a, b]
  3 Klassen: Eingabe kleiner, Eingabe im Intervall, Eingabe größer
- Anzahl der Werte der Eingabe ist n
  2 zusätzliche Klassen: Anzahl der Werte<n, Anz. >n
- Vermutung, daß Programm Eingaben einer Klasse unterschiedlich behandelt: Verfeinern der Äquivalenzklasse
- Eingabebedingung: „... A muß sein ...“
  2 Klassen: A erfüllt, A nicht erfüllt

in der Regel ist Eingabe Kreuzprodukt

$$A_1 \times A_2 \times . . . \times A_m$$

die „Äquivalenzklassenbildung“ bezieht sich auf jede der Eingabemengen A_i

$\Rightarrow$ diese „Äquivalenzklassen“ sind also nicht Äquivalenzklassen für die gesamte Eingabe

Festlegung der Testfälle (Aussuchen der Repräsentanten)

- gültige Klassen: Fall, der so viele wie möglich der bisher nicht behandelten gültigen „Äquivalenzklassen“ überdeckt bis alle überdeckt sind (wegen obiger Bemerkung kann ein Testfall mehrere „Klassen“ überdecken)
- ungültige Klassen: Testfall für jede der „Klassen“

Grund: Dreiecksbeispiel

(3.14, 2, 3, 1.0) für 2 ungültige „Klassen“

- nicht int
- 4 Eingaben

zweite Möglichkeit würde nie überprüft!

Beispiel:

Syntaxanalyse einer vereinfachten DIMENSION-Anweisung

Dimension_deklaration ::= DIMENSION Felddekl {, Felddekl}

Felddekl ::= id (indexang {, indexang}) -- 1-7 'Dimensionen'

id ::= … -- Aufbau eines Bezeichners wie üblich, 1-6 Zeichen

indexang ::= [ug:] og

ug ::= id | literal -- letzteres zwischen -65534 und 65535

og ::= ... -- og ≥ ug

-- Dimension_deklaration über mehrere Zeilen möglich

Tab. 4.1 Äquivalenzklassen

Eingabebedingung	gültige Äquivalenz-Klassen	ungültige Äquivalenz-Klassen
Anzahl der Felddeskriptoren	Einer(1). > eins(2)	Keine(3)
Größe des Feldnamens	1-6(4)	0(5). >6 (6)
Feldname	mit Buchstaben(7) mit Ziffern(8)	Irgend etwas Sonst(9)
Feldname beginnt mit Buchstaben	Ja(10)	Nein(11)
Anzahl der Dimensionen	1-7(12)	0(13). > 7(14)
Obere Grenze ist	Konstante(15) Ganzzahlvariable(16)	Feldelement-Name(17). Irgendetwas sonst(18)
Ganzzahlvariablenname	hat Buchstaben(19) hat Ziffern(20)	hat sonst irgendetwas(21)
Ganzzahlvariable beginnt mit Buchstaben	ja(22)	Nein(23)
Konstante	-65534 –65535(24)	< -65534(25) > 65535(26)
Untere Grenze angeben	ja(27), nein(28)	
Untere Grenze zu oberer Grenze	kleiner als (29) gleich(30)	größer als (31)
Angegebene untere Grenze	negativ(32) null(33). > 0(34)	
Untere Grenze ist	eine Konstante(35) Grenzzahlvariable(36)	Feldelementnamen(37), irgendetwas sonst(38)
Mehrere Zeilen	ja(39), nein(40)	

DIMENSION A(2) (1), (4), (7), (10), (12), (15), (24), (25), (29), (40)

DIMENSION A12345(1.9, J4XXXX, 65535, 1. KLM, 100), - restl. gültige
× BBB(-65534: 100.0:1000, 10:10, 1:65535) - Äquivalenzkl.

(3): DIMENSION
(5): DIMENSION (10)
(6): DIMENSION A234567(2)
(9): DIMENSION A.1(2)
(11): DIMENSION 1A(10)
(13): DIMENSION B
(14): DIMENSION B(4, 4, 4, 4, 4, 4, 4, 4)
(17): DIMENSION B(4, A(2))
(18): DIMENSION B(4.7)
(21): DIMENSION C(1! 10)
(23): DIMENSION C(10, 1J)
(25): DIMENSION D(-65535:1)
(26): DIMENSION D(65536)
(31): DIMENSION D(4:3)
(37): DIMENSION D(A(2):4)
(38): DIMENSION D(.:4)

Grenzwertanalyse (Black-Box-Fortsetzung)

- Erfahrung: Testfälle mit Grenzwerten oder dicht daneben haben größere Wahrsch. zur Fehlerauffindung: Repräsentanten an den „Rändern“ der Äquivalenzklassen
- Unterschied zu „Äquivalenzklassen“ von oben:
  - nicht ein beliebiger Repräsentant, sondern einer vom Rand
  - nicht nur Eingabedaten, sondern auch Ausgabedaten werden betrachtet
- Richtlinien:

(a) Eingabeparameter in Wertebereich: gültige Testfälle für Grenzen des Wertebereichs, ungültige Testfälle für Werte dicht neben Grenze

z.B. $-1.0 \leq x \leq 1.0$; Testfälle -1.0, 1.0, -1.001, 1.001

(b) „Eingabebedingung“ analog:

z.B. $1 \leq$ Anzahl_der_Sätze_einer_Eingabedatei ≤ 255

Testfälle: 1.255, 0, 256

(c) Wende (a) für Ausgabewerte an:

z.B. Programm errechnet $0.0 \leq$ monatl_Sozialabgabe ≤ 2200.0

Testfälle, die Abzug 0.0 bzw. 2200.0 veranlassen

versuche Testfälle zu finden, die negativen Abzug, Abzug > 2200.0 veranlassen

Grenzwerte in Ausgabewerten müssen keineswegs zu Grenzfällen der Eingabewerte korrespondieren (vgl. Sinus-Funktion)

(d) Wende (b) für „Ausgabebedingungen“ an:

z.B. $1 \leq$ Anzahl_von_Ausgabedaten ≤ 5

Testfälle, die 1, 5 Ausgabedaten ausgeben,

versuche Testfälle zu finden, die 0 bzw. 6 Ausgaben erzeugen

(e) Ist Ein-/Ausgabemenge eine geordnete Menge (seq. Datei, lin. Liste, Tabelle etc.): Betrachte erstes und letztes Element

Beispiel:

MTEST ist ein Programm für Auswertung von multiple-choice Prüfergebnissen. Die Eingabedatei mit dem Namen OCR besteht aus Sätzen mit 80 Zeichen. Der erste Satz enthält den Titel, der als Überschrift in jeder Ausgabe erscheint. In den nächsten Sätzen stehen die korrekten Antworten auf die Prüfungsfragen, wobei eine „2“ als letztes Zeichen erscheint. In den Spalten 1-3 des ersten Satzes steht die Anzahl der Fragen (eine Zahl zwischen 1 und 999). Die Spalten 10-59 enthalten die richtigen Antworten für die Fragen 1-50 (jedes Zeichen wird als Antwort akzeptiert). In den Spalten 10-59 der folgenden Sätze sind die korrekten Antworten auf die Fragen 51-100, 101-150 usw. enthalten.
Im nächsten Satzabschnitt stehen die Antworten der Studenten mit einer „3“ in der Spalte 80. In den Spalten 1-9 befindet sich der Name des Studenten oder seine Nummer (alle Zeichen). Die Spalten 10-59 enthalten die Antworten des Studenten auf die Fragen 1-50. Besteht der Test aus mehr als 50 Fragen, so enthalten die nachfolgenden Sätze die Antworten 51-100, 101-150 usw. in den Spalten 10-59. Die Maximalzahl der Studenten beträgt 200. Die Eingabedaten sind in Abb. 4.4 dargestellt.
Als Ausgabe werden vier Berichte erstellt:

1. Ein Bericht, sortiert nach den Kennzeichen der Studenten, der seine Einstufung (prozentualer Anteil an richtigen Fragen) und seinen Rang zeigt,
2. einen ähnlichen Bericht, aber nach Noten sortiert,
3. einen Bericht über Mittelwert, Median und Standardabweichung der Noten, und
4. einen Bericht, geordnet nach Fragennummern, der zeigt, wieviel Prozent der Studenten die Fragen richtig beantwortet haben (Ende der Spezifikation).

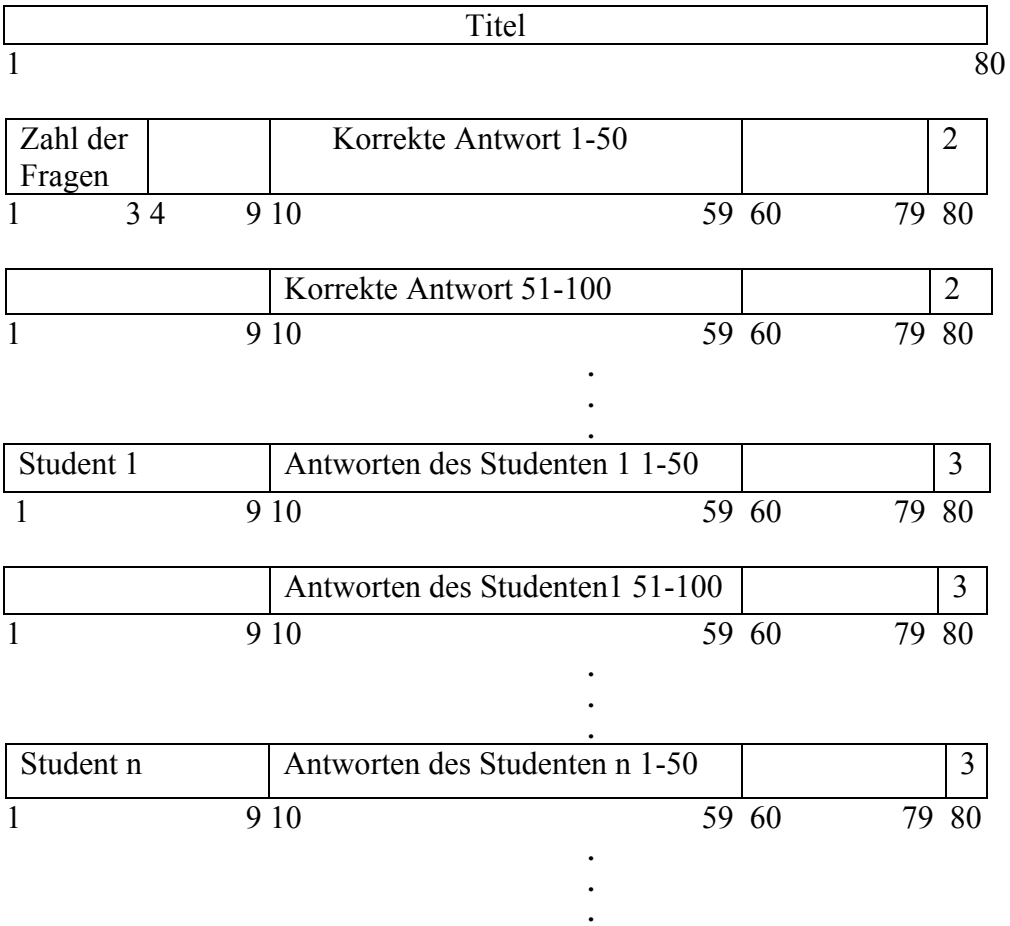


Abb. 4.4 Eingabe für das Programm MTEST

1. Leere Eingabedatei
2. Fehlender Titel
3. Titel, ein Zeichen lang
4. Titel, 80 Zeichen lang
5. Test mit einer Frage
6. Test mit 50 Fragen
7. Test mit 51 Fragen
8. Test mit 999 Fragen
9. Keine Frage im Test
10. Das Feld „Anzahl der Fragen“ hat einen nichtnumerischen Wert
11. Nach dem Titelsatz fehlen die Sätze mit den richtigen Anworten
12. Ein Satz mit Anworten ist zuviel vorhanden
13. Ein Satz mit Antworten fehlt

Die nächsten Eingabedaten beziehen sich auf die Antworten der Studenten. Als Grenzwerttestfälle bieten sich an:

14. Kein Student
15. Ein Student
16. 200 Studenten
17. 201 Studenten
18. Ein Student hat einen korrekten Antwortsatz, aber es gibt zwei korrekte Antwortsätze
19. Der obengenannte Student ist der erste Student in der Datei
20. Der obengenannte Student ist der letzte Student in der Datei
21. Ein Student hat zwei Antwortsätze angegeben, aber es gibt nur einen Satz mit korrekten Antworten
22. Dieser Student ist der erste Student in der Datei
23. Dieser Student ist der letzte Student in dieser Datei

Ebenso nützliche Testfälle erhält man, wenn man die Grenzwerte der Ausgabedaten untersucht, obwohl einige von schon existieren den Testfälle erfaßt werden (z.B. Bericht 1 leer).
Die Randwerte der Berichte 1 und 2 sind:

Kein Student	(Test 14)
1 Student	(Test 15)
200 Studenten	(Test 16)
201 Studenten	(Test 17)

24. Alle Studenten erhalten die gleiche Punktezahl
25. Jeder Student erhält eine andere Punktezahl
26. Einige, aber nicht alle Studenten erhalten die gleiche Punktezahl (um zu überprüfen, ob die Einstufung richtig durchgeführt wird).
27. Ein Student erhält 0 Punkte.
28. Ein Student erhält 100 Punkte.
29. Ein Student hat die niedrigstmögliche Kennzahl (um die Sortierung zu überprüfen)
30. Ein Student hat die höchstmögliche Kennzahl
31. Die Anzahl der Studenten ist gerade so groß, daß der Bericht genau in eine Seite paßt (um zu prüfen, ob eine zusätzliche Seite gedruckt wird)
32. Die Anzahl der Studenten ist gerade so groß, daß alle, bis auf einen, in eine Seite passen.

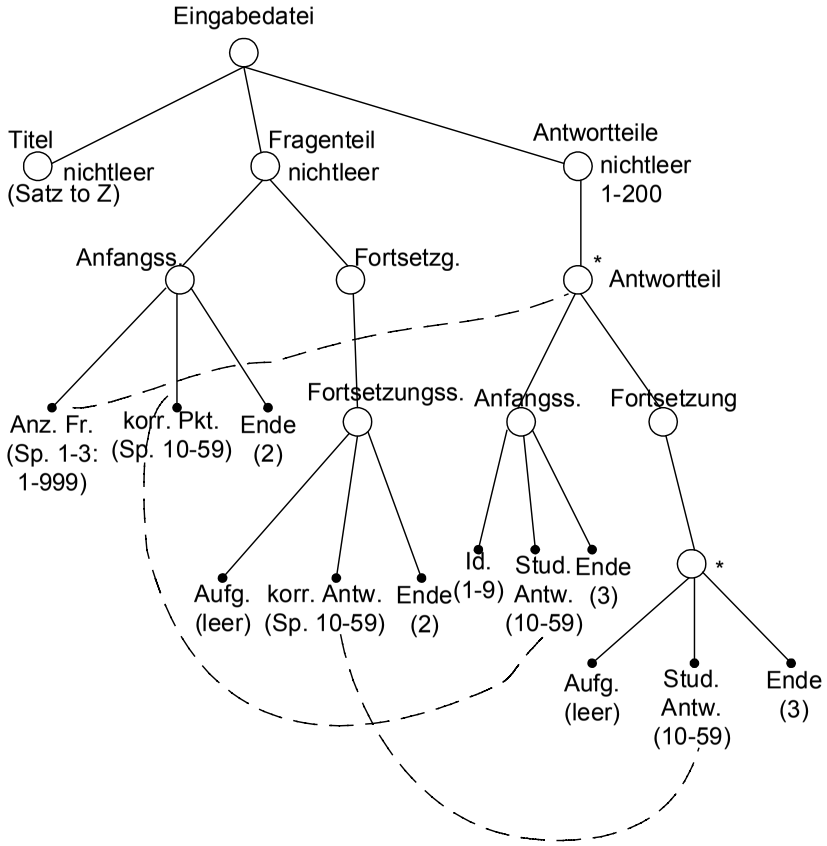
Die Grenzwerte für den Bericht 3 (Mittelwert, Median, Standardabweichung) bestehen aus

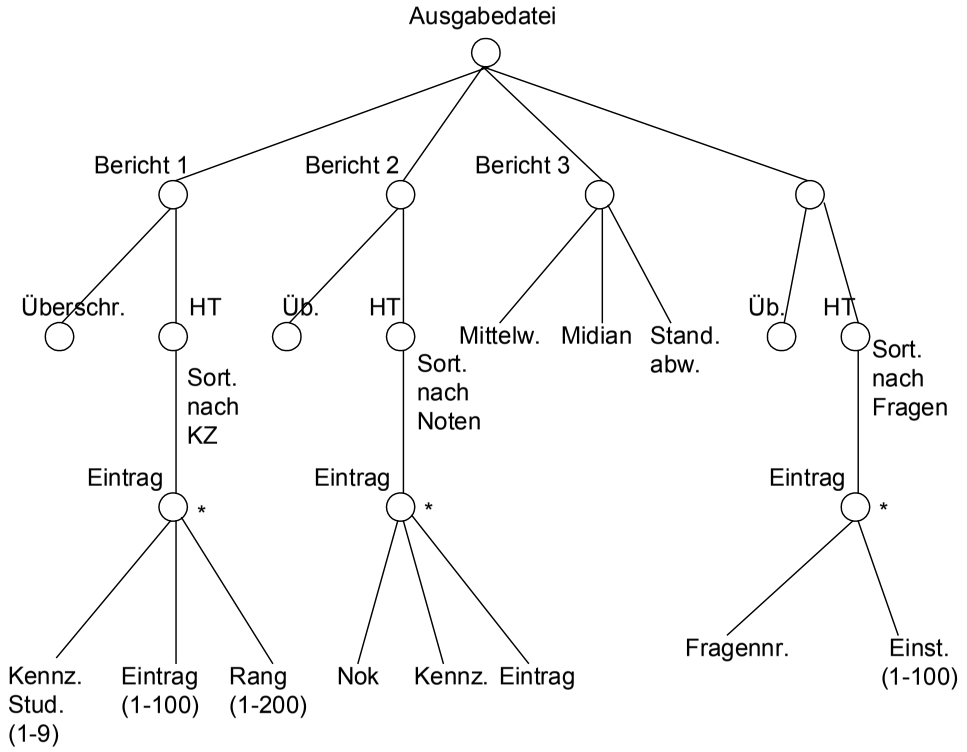
33. Der Mittelwert liegt bei seinem Maximum (alle Studenten haben die volle Punktezahl)
34. Der Mittelwert ist Null (alle Studenten haben 0 Punkte erhalten)
35. Die Standardabweichung liegt bei ihrem Maximum (ein Student hat 0 und der andere 100 Punkte erhalten)
36. Die Standardabweichung ist 0 (alle Studenten haben die gleiche Punktezahl erhalten)

Test 33 und 34 erfassen auch die Grenzwerte des Medians. Ein weiterer nützlicher Testfall ergibt sich für null Studenten (überprüfen auf Division mit 0 bei der Berechnung des Mittelwerts), aber der wird schon durch Testfall 14 erfaßt.

Eine Überprüfung des Berichtes 4 ergibt folgende Randwerte:

37. Alle Studenten beantworten Frage 1 richtig
38. Alle Studenten beantworten Frage 1 falsch
39. Alle Studenten beantworten die letzte Frage 1 richtig
40. Alle Studenten beantworten die letzte Frage 1 falsch
41. Die Anzahl der Fragen ist gerade so groß, daß sie auf eine Seite paßt
42. Die Anzahl der Fragen ist gerade so groß, daß alle Fragen bis auf eine auf eine Seite passen


Eingabedatei
Titel
nichtleer
(Satz to Z)
Fragenteil
nichtleer
Antwortteile
nichtleer
1-200
Anfangss.
Fortsetzg.
*
Antwortteil
Anz. Fr.
(Sp. 1-3:
1-999)
korr. Pkt.
(Sp. 10-59)
Ende
(2)
Fortsetzungss.
Anfangss.
Fortsetzung
Aufg.
(leer)
korr. Antw.
(Sp. 10-59)
Ende
(2)
Id.
(1-9)
Stud.
Antw.
(10-59)
Ende
(3)
*
Aufg.
(leer)
Stud.
Antw.
(10-59)
Ende
(3)


Ausgabedatei
Bericht 1
Bericht 2
Bericht 3
Überschr.
HT
Üb.
HT
Mittelw.
Midian
Stand.
abw.
Üb.
HT
Sort.
nach
KZ
Sort.
nach
Noten
Sort.
nach
Fragen
Eintrag
*
Eintrag
*
Eintrag
*
Kennz.
Stud.
(1-9)
Eintrag
(1-100)
Rang
(1-200)
Nok
Kennz.
Eintrag
Fragennr.
Einst.
(1-100)

Fehlerraten (error guessing)

- Intuition und Erfahrung: wahrscheinliche Fehler vermutet, entsprechende Testfälle aufgestellt (damit auch einige der oben systematisch gefundenen Fälle auffindbar)
- für Beispiel MTEST

1. Akzeptiert das Programm „blanks“ (Leerstellen) als Antwort?
2. Ein Satz vom Typ 2 (Antwort) erscheint in der Menge der Sätze vom Typ 3 (Student).
3. Ein Satz mit einer 2 oder 3 in der letzten Spalte erscheint als Anfangssatz.
4. Zwei Studenten haben den gleichen Namen oder die gleiche Nummer.
5. Da der Zentralwert (Median) unterschiedlich berechnet wird, abhängig davon, ob die Anzahl der Elemente gerade oder ungerade ist, testet man das Programm für eine gerade und eine ungerade Anzahl von Studenten.
6. Das Feld „Anzahl der Fragen“ hat einen negativen Wert.

Gesamtstrategie

BB

1) Aufstellen UW-Graph insbesondere, wenn Kombinationen von Eingabebedingungen vorliegen
2) Auf jeden Fall Grenzwertanalyse (ggf. können die gefundenen Testfällen in die UW-Testfälle eingearbeitet werden)
3) Ergänzen der Testfälle durch Äquivalenzklassenbetrachtung
4) Ergänzen durch Error Guessing

WB

5) Untersuche Programmstruktur:
   - Überdeckung aller Entscheidungen/Sprünge u. Anweisungen
   - Überdeckung aller Bedingungen

Integrationstest

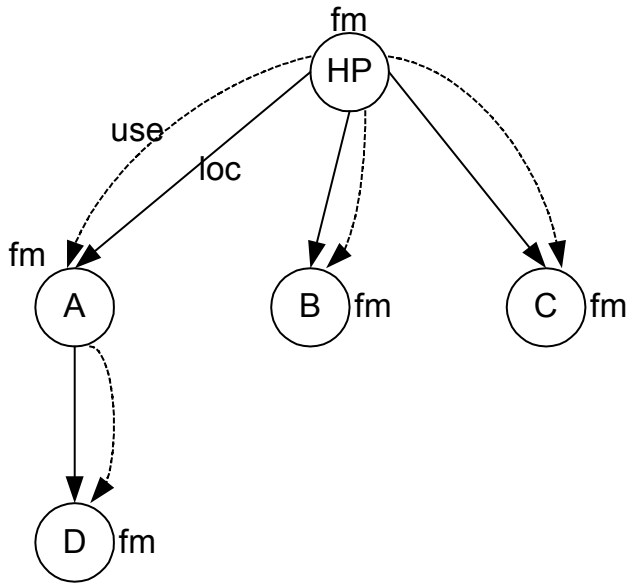
Modultest (Unit Testing)

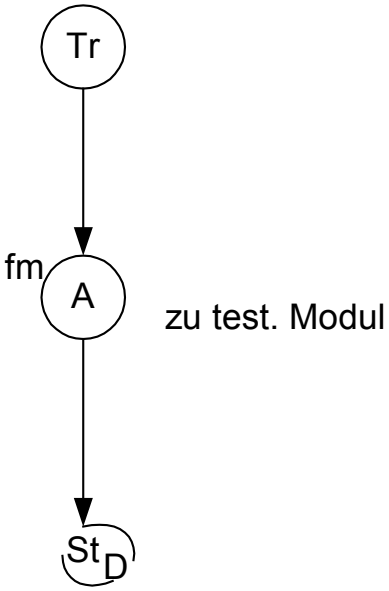
- Test einer “abgeschlossenen (Teil) Einheit” eines Programmsystems: Übersicht leichter, Fehleranalyse leichter, Fehlerbehebung leichter
- Zielsetzung: Feststellung, daß der Modul seiner Spezifikation widerspricht
- Spezifikation (santeil) und Quellcode des Moduls nötig
- Testfallentwurf für Modul mit Blackbox- und Whitebox-Methoden s.o.
- für jeden Modultest benötigen wir i.a.
  - Testtreiber: einf. Programm, das zu testenden Modul benutzt
  - Teststummel (Stub): einf. Programm, das zu benutzenden Modul simuliert
  - manchmal nur Treiber oder Stummel

Integrationstest

nichtinkrementelles Testen (big-bang-Testen)

– Testen der einzelnen Module nacheinander oder parallel für jeden Modul, dafür Treiber und Stubs





– Zusammenfassen der getesteten Einzelmodule und Test des gesamten Programmsystems

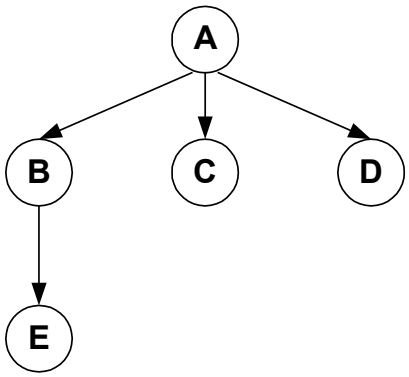
Inkrementelles Testen

Inkremente hier ganze Module (nicht Inkremente, vgl. SEU–Lit.)

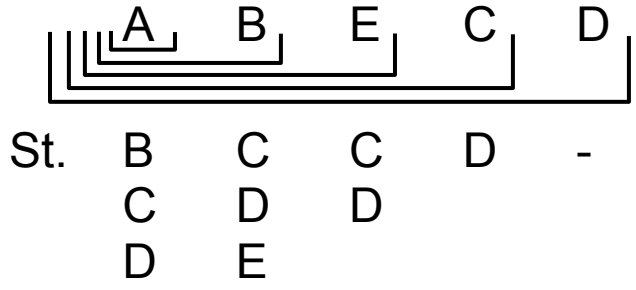
- Top-down-Testen
  (geht „in der Regel“ einher mit Top-down-Entwicklung)

a) Top-down-Testen (geht in der Regel einher mit Top-down-Entwicklung)

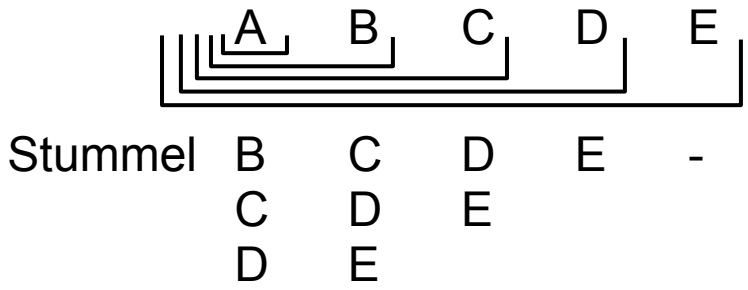
- Beginn mit der Wurzel
- Ende mit Blättern
- hier nur Teststummel zu verwenden
- Stummel simuliert die Arbeitsweise: gültige, ungültige Eingabedaten bzw. deren Grenzwerte



- Stub automatisch erzeugbar
- ist Modul ausgetestet, wird er zu Teilsystem hinzugenommen, Stub gestrichen: Test beginnt mit Einzelmodul Wurzel, endet mit gesamtem System
- Reihenfolge der Hinzunahme nicht festgelegt:
- z.B. Links-rechts-depth-first:



Links-rechts-breadth-first:

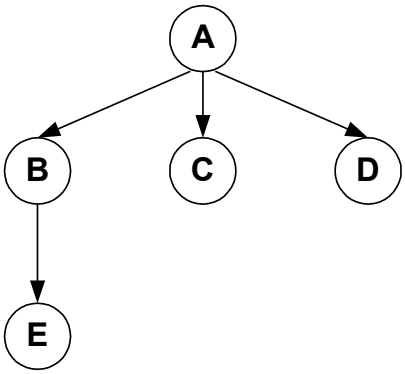


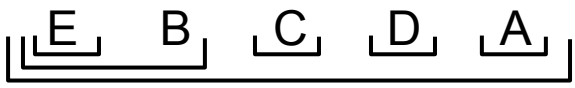
- Test ist parallelisierbar z.B. im Falle LR-breadth-first:
  so viele gleichzeitig, wie Ebene Module hat

- Gibt es „kritische“ Module: Testreihenfolge so, daß diese möglichst bald getestet werden
  E/A-Module mögl. bald, da von vielen gebraucht

b) Bottom-up-Testen
(in „der Regel“ mit Bottom-up-Entwicklung verknüpft)

- beginnt mit Blättern
- endet mit Wurzel
- hier nur Testtreiber zu verwenden
- Treiber automatisch erzeugbar
- Ist der Modul ausgetestet wird er hinzugenommen, Treiber gestrichen: Test beginnt mit Einzelmodul (Blatt) und endet mit Gesamtsystem
- Reihenfolge der Hinzunahme nicht festgelegt (alle untergeordneten Module müssen getestet sein, wenn gemeinsamer Vater getestet wird)





Treiber für E B C D -

- ablauffähiges Programm hier erst nach Hinzunahme des Hauptprogramms

Vergleich big-bang-Testen mit inkrementellem Testen

- big bang aufwendiger
- Schnittstellenfehler durch inkrementelles Testen eher entdeckt, da Schnittstellen früher überprüft werden, falls Implementierung mit Integration überlagert wird.
- Fehlerlokalisierung im big-bang-Fall für Schnitt stellenfehler schwerer
- inkr. Testen: ausgetesteter Modul übernimmt die Funktion des Treibers oder Stummels im nichtinkr. Fall: wird dabei zusätzlich noch einmal überprüft
- Rechenzeit bei big-bang-Testen kleiner, da Treiber und Stummel kaum Zeit verbrauchen. Gilt aber kaum, wenn Treiber und Stummel zu Fuß entwickelt werden müssen, da dann ihre Entwicklung Rechenzeit verbraucht
- big-bang stärker parallelisierbar: bei großen Projekten kann Testzeit verkürzt werden

Abnahmetest

Teilaspekte

- Vollständigkeit: alle Funktionen der Anforderungsdef. realisiert?
- Volumentest: Nachweis, daß Programmsystem bei max. Datenanfall überfordert wird
- Lasttest: Nachweis, daß Programmsystem einer max. Dauerlast nicht gewachsen ist
- Leistungstest: Nachweis daß Programmsystem die Leistungsparameter der Anforderungsdef. bzgl. Laufzeit, Reaktionszeit, Durchsatz, Hauptspeicherbedarf, Sekundärspeicherbedarf, Anzahl verarbeiteter Dateien etc. insb. unter „schwierigen“ Bedingungen nicht nachkommt
- Benutzerfreundlichkeit: Berücksichtigung Benutzermodell, Fehlerausgaben angemessen, Ausgaben aufbereitet, Benutzerschnittstelle einheitlich bzgl. Syntax und Semantik, Format, Stil, Abkürzungen etc. Genügend Redundanz in der Eingabe? Optionen, die nie benutzt werden? Leichte Anwendbarkeit?
- Konfigurationstest: wenn versch. Komponenten angeschlossen werden dürfen (versch. Platteneinh., Terminals)
- Konversion: ersetzt das Programmsystem ein anderes, so sind ggf. Konversionsroutinen für die Übertragung der Datenbestände nötig

- falls System Initialisierungs-/Installationsroutinen hat: Testen
- Ausfallsicherheit, Wiederaufsetzen: Erzeugung von Softwarefehlern, Dateifehlern, Hardwarefehlern
- Überprüfung der Dokumentations- / Wartbarkeitsanforderungen der Anforderungsdef.
- Überprüfung der Operatorfunktionen

Abnahmetest vom Auftraggeber durchgeführt

alle diese Punkte zuerst intern (beim Auftragnehmer) überprüft („Systemtest“, vor Installationstest und Abnahmetest)

Systemtest von Team ≠ Autorenteam

Planung und Beendigung

Testplanung

– Testen erzeugt Managementproblem: viele Module, viele Querbeziehungen, viele Fehler, viele Mitarbeiter damit beschäftigt
  ⇒ Testplanung wichtig

– Testplan:

- Ziel jeder Testphase muß bestimmbar sein
- Abschlußkriterium jeder Testphase muß festgel. sein
- Zeitplan für jede Testphase: Überlegungen Testentwurf, Entwurf der Testfälle, Beschreibung der Testfälle, Ausführung mit Testfällen
- Verantwortlichkeiten: Mitarbeiter für Test und Fehlerbehebung müssen benannt werden
- Testfallbibliotheken: für Beschreibung und Speicherung der Testdaten (wiederverwendbar)
- Testwerkzeuge: Beschreibung der erforderlichen Testwerkzeuge, ihre Anwendung, wann sie benötigt werden, Plan für die Entwicklung (falls noch nicht vorhanden), Plan, wer sie wann benutzt
- Rechenzeit: Plan, für die in jeder Testphase benötigte Rechenzeit
- Hardwarekonfiguration: Ist spez. Hardwarekonf. erforderlich? Ggf. Plan über Anforderung, Aufstellung und Einsatz von Geräten

- Integration: insb. bei inkrementellem Testen nötig:
  Plan für die Reihenfolge der Integration, Zeitpunkt des Vorhandenseins der get. Komp., Testtreiber und Teststummel
- Vorgaben für Testfortschritt, Lokalisierung fehleranfälliger Teile, Abschätzung der Testentwicklung (Zeitplan, Ressourcen, Beendigungsmerkmale)
- Fehlerbehebung: Festlegung von Mechanismen für Fehlermeldung und Weiterleitung des Fehlers, Fehlerverfolgung, Korrektur,
  Plan für Zeitspanne für Fehlerbehebung, Verantwortlichkeit, Werkzeuge, Rechenzeit
- Regressionstest: Test nach Verbesserung oder Reparatur zur Feststellung, ob andere Teile berührt wurden mit vorher festg. Testfällen aus Testfallbibliothek. Plan: wer, wie, wann

Test-Beendigungskriterien

– Testen kann nie alle Fehler finden: wann hört man auf?

– In Praxi: Beendigung wenn

- Testzeit abgelaufen
- alle Testfälle ohne Fehler

beides sinnlos, da durch Nichtstun erfüllbar

– 1. Ansatz: Testfälle nach vorgeg. Testfallentwurfsmethode (z.B. Mehrfachbed. plus Grenzwertanalyse für Modultest)

Es wird hier kein Ziel für das Testen gesetzt: nur praktikabel, falls Tester Testfallentwurfsmeth. bereits mit Erfolg angewandt hat

– 2. erweiterter Ansatz:

Vorgabe einer bestimmten Anzahl von Fehlern, die aufzufinden sind

Probleme:

- Abschätzung der Gesamtzahl der Fehler (z.B. Erfahrung mit früheren Projekten; allg. Schätzwerte der Industrie)
- Abschätzung, wieviel hiervon durch Testen aufgefunden werden können (abh. von der Art des Problems, Aufbau des Programmsystems, Konsequenz nicht entdeckter Fehler)

- Schätzung, welche Fehler, welchen Phasen entstanden sind und in welchen Testphasen sie erkannt werden

Annahme:

40% der Fehler Codierungs- u. Modulentwurfsfehler
60% der Fehler „Entwurfs“fehler

Tab. 6. 1 Schätzung des Zeitpunkts, zu dem die Fehler gefunden werden

	Codierungs- und logische Entwurfsfehler	Entwurfsfehler
Modultest	65%	0%
Funktionstest	30%	60%
Systemtest	3%	35%
Total	98%	95%

Beispiel: Programm mit 10.000 Anweisungen

Annahme: nach Codeinspektion ca. 5% der Anweisungen falsch

⇒ 500 Fehler insgesamt

⇒ 200 Codierungs-, 300 Entwurfsfehler

Annahme: 98% der Codierung- und 95% der Entwurfsfehler können aufgefunden werden

⇒ 175 Codierungsfehler, 285 Entwurfsfehler entdeckbar

⇒ Modultest: 114 Codierfehler (65% von 175)

Funktionst. 52 Codierf. (30% von 175)

170 Entwurfsf. (60% von 285)

Systemtest: 5 Codierf. (3% von 175)

100 Entwurfsf. (35% von 285)

Somit Beendigung

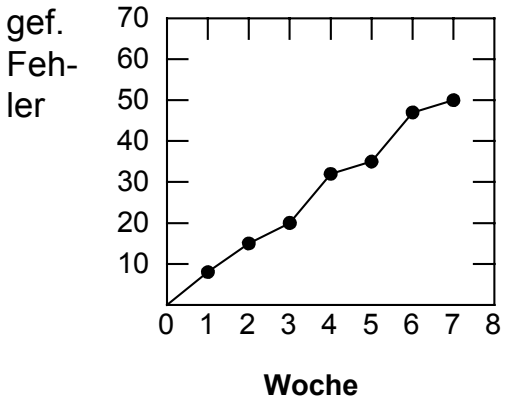
Modultest: 115 Fehler gefunden und verbessert

Funktionstest: 220 Fehler oder 4 Monate

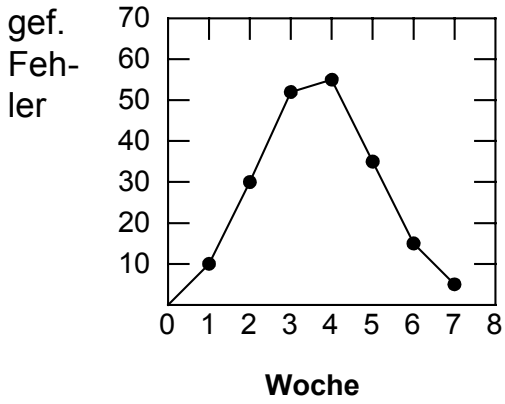
Systemtest: 105 Fehler oder 3 Monate

Problem: Was wenn im gesetzten Zeitraum mehr oder weniger Fehler entdeckt werden?

3. erweiterter Ansatz: zusätzlich Auftragung der Häufigkeit gefundener Fehler in Zeiteinheiten



Abbruch unklug



Achtung: Abfall könnte durch Motivationslosigkeit Rechenzeitmangel o. ä. bedingt sein

Somit insgesamt:

- Festlegung der Meth. der Testfallgewinnung
- Abschätzung der Anzahl aufzuf. Fehler, best. Arten, in welchen Testphasen
- Überwachung der Testphasen durch Häufigkeitsdarst.

Fragen zu Kap. 11

1) Klassifizieren Sie die in der Praxis am häufigsten benutzten QS-Maßnamen Modul/Teilsystemtest, Integrations-, Abnahmetest sowie Reviews (für Dokumente des RE und des PiG) bzgl. der oben angegebenen Klassifikation.
2) Charakterisieren Sie den Unterschied zwischen dem üblichen Überdeckungsverfahren und der Flußgraphenmethode.
3) Eine Funktion sei durch drei ganzzahlige Eingangsparameter gegeben und liefere einen ganzzahligen Wert. Wieviele mögliche Testfälle gibt es prinzipiell für den Black-Box-Test. Wieviele gibt es beim ”Äquivalenzklassen“-Ansatz?
4) Nehmen Sie ein Beispiel einer kleinen Architektur in der nur Module auftauchen und in dessen Realisierung auf die Konstrukte der zugrundeliegenden Programmiersprache Bezug genommen wird bzw. auf einige vordefinierte Bausteine. Wieviele Tests für einzelne Module stehen an? Wieviele Testreihenfolgen für inkrementelles Testen nach Top-down- bzw. Bottom-up-Ansatz gibt es?

   Zeichnen Sie einen Modul als kritisch aus, der möglichst früh geprüft werden soll. Ferner sollen mehrfach benutzte Teilarchitekturen (Modul plus weitere benutzte Module) vor deren ”Benutzung” im Test integriert sein. Geben Sie eine mögliche Integrationstest-Reihenfolge an.

Literatur zu Kap. 11

/Ba 98/ H. Balzert: Lehrbuch der Software-Technik, Band II, Kap. III, 253-554, Spektrum-Verlag (1998).

/Bei 83/ B. Beizer: Software Testing Techniques, New York: van Nostrand (1983).

/Bei 84/ B. Beizer: Software System Testing and Quality Assurance, New York: van Nostrand (1984).

/Boe 84/ B. W. Boehm: Verifying and Validating Software Requirements and Design Specifications, IEEE Software 1, 1, 75-88 (1984).

/CDS 86/ S. D. Conte/H. E. Dunsmore/V. Y. Shen: Software Engineering Metrics and Models, Menlo Park: Benjamin Cummings (1986).

/CSM 89/ Conference on Software Maintenance 1989, Proceedings, IEEE Comp. Soc. Press (1989).

/EM 87/ M. W. Evans/J. J. Marciniak: Software Quality Assurance and Management, New York: John Wiley (1987).

/FBB 82/ W. R. Franta/H. K. Berg/W. E. Boebert/T. G. Moher: Formal Methods of Program Verification and Specification, Englewood Cliffs: Prentice Hall (1982).

/GC 87/ R. B. Grady/D. R. Caswell: Software Metrics: Establishing a Company-wide Program, Englewood Cliffs: Prentice Hall (1987).

/He 84/ W. Hesse et al.: Ein Begriffssystem für die Softwaretechnik, Informatik-Spektrum 7, 1, 200-213 (1984).

/How 87/ W. E. Howden: Functional Program Testing and Analysis, New York: McGraw-Hill (1987).

/Kem 89/ R. A. Kemmerer (Ed.): Proc. 3rd Symp. on Software Testing, Analysis, and Verification (TAV 3), ACM Software Engineering Notes 14, 8 (1989).

/MH 81/ E. Miller/W. E. Howden (Eds.): Software Testing and Validation Techniques, New York: IEEE Comp. Soc. Press (1981).

/Mor 90/ M. Moriconi (Ed.): Int. Workshop on Formal Methods in Software Development, ACM Software Engineering Notes 15, 4 (1990).

/Mye 79/ G. Myers: The Art of Software Testing, New York: John Wiley (1979).

/PSS 81/ A. Perlis/F. Sayward/M. Shaw (Eds.): Software Metrics - An Analysis and Evaluation, Cambridge: MIT Press (1981).

Teil IV:

Abschluß, Ausblick

Wertung