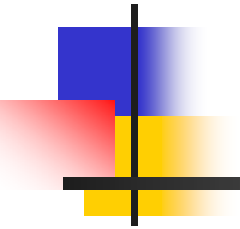
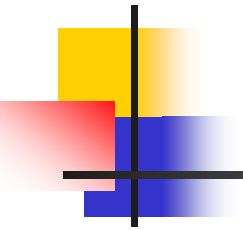


Methoden zur Überprüfung der Ergebnisse
9. Test und Integration

von

Prof. Dr. Wolfgang Weber

([Wi05] , [SW02: Sneed, Winter, Testen oo SW],
[Vi05: Vigenschow, oo Testen und Testautomatisierung in der Praxis]
Einzelne Folien zur Vorlesung entstammen aus Präsentationen von
Prof. R. Hahn und Prof. G. Raffius



Lernziel Test und Integration

Sie sollen in diesen Kapitel verstehen

- was ein Fehler ist,
- was Testen ist,
- dass Black-Box und White-Box unterschiedliche Testmöglichkeiten bieten,
- wie man geschickt Testfälle auswählt,
- welche Verfahren es für den White-Box-Test gibt
- ...

Anschließend können Sie systematisch Testfälle erstellen

Was ist ein Fehler?

= Abweichung vom erwarteten Verhalten

- Abstürze und / oder Datenverluste
  - Fehlerart, die uns als erstes einfällt
    - ⇒ Tests
- Fehlverhalten

  auf Grund falsch implementierter Regeln oder Berechnungen
  - Ursache: unzureichendes Verständnis der Sachverhalte
    - ⇒ besseres Verständnis vor Begin des Entwurfs und Nachfragen bei Unklarheiten während des Projektes
  - Ursache: Programmierfehler
    - ⇒ Test der Programme

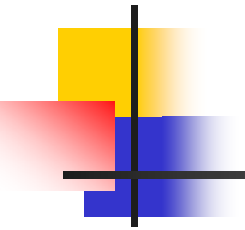
Was ist ein Fehler?

aber auch:

- unerfüllte Erwartungen
  - härtester Fehler
  - mangelnde, unvollständige Kommunikation mit den Anwendern am Anfang des Projektes
    - ⇒ Sorgfältige Anforderungsermittlung

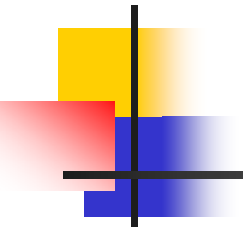
- Schlechte Performance
  - muss bei Anforderungsermittlung definiert werden

    Dann: technische Machbarkeit prüfbar
    - ⇒ Performance-Tests



Was ist ein Fehler?

- Inkonsistente Benutzerschnittstellen
  -> stören Arbeitsfluss
  -> Fehlbedienungen
  -> geringe Akzeptanz
  -> hoher Schulungsbedarf
    ⇒ Konsistente, ergonomische Benutzerschnittstellen entwerfen
      dazu: Benutzerschnittstellen-Entwerfer schulen



Testen von Software

demonstratives Testen

- =Entwicklertest.
- zeigt, dass Programm funktioniert (läuft)
- demonstratives Testen ist das, was Sie schon immer gemacht haben (beim Programmierpraktikum)

destruktives Testen

- = systematisches Testen
- in Absicht Fehler zu finden
- evtl. durch spezielles Testteam
- nächste Folien: destruktives Testen

Grundidee Testen

- Ein Test(fall) zu einem Fehler erzeugt ein vom Soll-Ergebnis abweichendes Ergebnis, wenn der Fehler vorliegt

- Vergleich Ist-Ergebnis mit Soll-Ergebnis
  - Stimmt das Ergebnis überein, ist dieser Testfall richtig abgelaufen und der Fehler liegt nicht vor
  - Stimmt das Ergebnis nicht überein, liegt mindestens ein Fehler vor (evtl. auch ein anderer, der sich bloß ähnlich äußert)

Teste alle möglichen Eingabekombinationen

Grenze des Testens

Beispiel: Test für einen Multiplizierer für 2 Zahlen –
bei einer (optimistischen) Testgeschwindigkeit von 10^9 Tests/Sek

- Für einen 32 Bit Multiplizierer:
  Ein vollständiger Test müsste $2^{64} = 1{,}8 * 10^{19}$ verschiedene Eingaben enthalten
  ⇨ Ein Testdurchlauf würde ca. 585 Jahre dauern! (Ohne Abgleich Ergebnisse!)

- Für einen 64 Bit Multiplizierer:
  ⇨ Ein Testdurchlauf würde ca. 10,8 Trilliarden Jahre dauern!

Wähle "geschickte" Stichproben statt die Vollständigkeit anzustreben!

Was verstehen Sie unter Testen von Software?

- stichprobenartige Ausführung des Testobjektes (des zu testenden Programms).
- Vergleich Ist-Ausgabewerte mit Soll-Ausgabewerte => ist dieser Testfall richtig abgelaufen.
- Durch Testen von Software kann normalerweise nicht die Fehlerfreiheit nachgewiesen werden,

  da man ja nicht alle möglichen Testfälle durchspielen kann.
- Testen kann nur die Zuverlässigkeit verbessern

Was ist ein Testfall?

- Für einen Testfall werden die Werte **aller Eingaben** eines zu testenden Programms als **konkrete** Werte vorgegeben und
  die Soll-Ausgabewerte zu diesen Eingaben dokumentiert.
- Stimmen die Ist-Ausgabewerte mit den Soll-Ausgabewerten überein, so ist das Programm mit den vorgegebenen Eingabewerten richtig abgelaufen.
- Was sind Eingabe- / Ausgabewerte?
  - Parameter einer Funktion

  Zusätzlich zu den Parametern einer Funktion?
  - Daten, die über die Benutzerschnittstelle ein-/ausgegeben werden
  - Daten von/zu Sensoren/Externen Geräten
  - Daten, die aus/in Datenbanken/Dateien gelesen und geschrieben werden
  - Globale Variablen
  - Attribute der Klasseninstanz (der eigenen oder fremden Klasseninstanzen)
  - (Berücksichtigung der Ein-/ Ausgabewerte der aufgerufenen Funktionen!)
- Dokumentation der Testfälle notwendig
  - zur Mitteilung der Fehler an Entwickler
  - um Tests nach Änderungen bzw. Fehlerkorrektur wiederholen zu können.
  - um beim der Fehlersuche zu wissen, was getestet wurde.

Wann ist das Programm (z. B. Funktion) fehlerfrei?

- wenn alle möglichen Testfälle fehlerfrei laufen

Ist es normalerweise möglich alle Testfälle laufen zu lassen?

- nein

was ist zu tun?

- Wir müssen uns auf gewisse Testfälle beschränken

Können wir damit Fehlerfreiheit testen?

- nein, nur Zuverlässigkeit verbessern

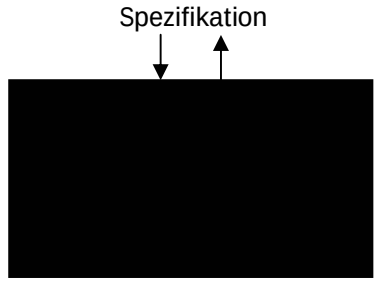
Wie erreichen wir, dass die Zuverlässigkeit nach dem Test möglichst hoch?

- geschickte Auswahl von Testfällen

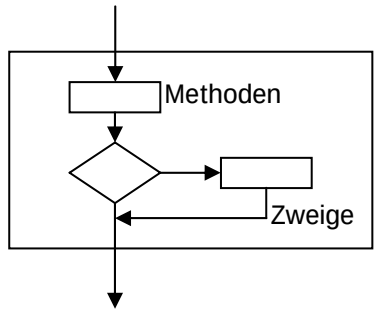
Auswahl von Testfällen

Wie gehen wir bei der Auswahl der Testfälle vor, um mit großer Wahrscheinlichkeit möglichst viele Fehlersituationen ausschließen zu können?

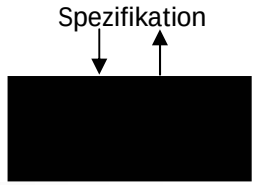
- Black-Box-Test
  - Das zu testende Modul ist "schwarzer Kasten"
  - Man sieht nur die Oberfläche (das "Was"), d. h. die Modulspezifikation.
    Die innere Struktur sei für den Tester unbekannt.



- White-Box-Test (Glas-Box-Test, Strukturtest)
  - Man schaut in den Kasten hinein.
  - Die innere Struktur (das "Wie") des Programms wird zur Überprüfung hinzugezogen



Black-Box-Test

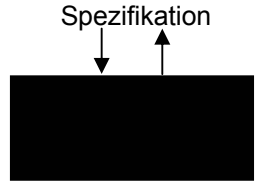


- Testplanung:
  - Ableitung von Testfällen aus der Spezifikation des Testobjekts.
- Testfälle:
  - Eingabe- und Ausgabedaten (Soll-Ergebnisse) zu Testobjekt.
- Vollständiger Test ist im allgemeinen nicht durchführbar

  =>Ziel:
  - Testfälle so auswählen, dass die Wahrscheinlichkeit groß ist, Fehler zu finden.

=> Äquivalenzklassen
=> Grenzwerte
=> Intuitive Testfallermittlung (Erfahrung)

Black-Box-Test - Äquivalenzklassen

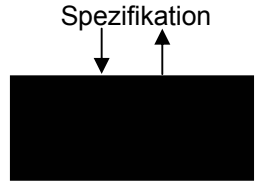


Methoden für Black-Box-Test:

- **Bilden von Äquivalenzklassen**
  - Einteilung der Menge der möglichen Werte zu einer Eingabe in Äquvalenzklassen
  - Annahme: Das Testobjekt reagiert bei der Verarbeitung eines Vertreters aus einer Äquivalenzklasse "genauso" wie bei allen anderen Werten dieser Klasse,
  - d. h.: Testobjekt läuft mit dem repräsentativen Wert (Repräsentant) der Äquivalenzklasse fehlerfrei.

    => Es läuft auch mit allen anderen Werten dieser Äquivalenzklasse fehlerfrei.
- Beim Äquivalenzklassentest testet man das System mit jeweils einem Repräsentant pro Äquivalenzklasse.
- Es gibt
  - gültige Äquivalenzklassen (mit gültigen Eingabewerten) und
  - ungültige Äquivalenzklassen (mit ungültigen Eingabewerten)

Black-Box-Test - Äquivalenzklassen



Beispiel: Spezifikation der Funktion Dreieck:

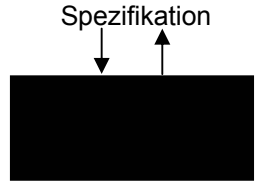
dreieck DREIECK (Integer SEITE1, SEITE2, SEITE3);

mit enum dreieck=(UNGLEICHSEITIG, GLEICHSCHENKLIG, GLEICHSEITIG, KEIN_DREIECK)

Zweck:

- SEITE1, SEITE2 und SEITE3 sind die Seitenlängen eines Dreiecks.
- Die Seitenlängen seien positive Werte.
- Die Funktion stellt fest, ob es sich um ein
  - ungleichseitiges,
  - ein gleichschenkliges,
  - ein gleichseitiges oder
  - um gar kein Dreieck handelt.
- Zulässige Eingabewerte seien die positiven ganzen Zahlen.

Black-Box-Test - Äquivalenzklassen



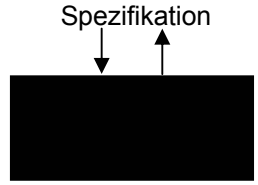
Ergebnis:

UNGLEICHSEITIG, wenn
(SEITE1 != SEITE2) and (SEITE1 != SEITE3) and (SEITE2 != SEITE3)
and ((SEITE1 + SEITE2) >= (SEITE3)),
and ((SEITE1 + SEITE3) >= (SEITE2)),
and ((SEITE2 + SEITE3) >= (SEITE1)),
GLEICHSCHENKLIG, wenn
((SEITE1 = SEITE2) and (SEITE1 != SEITE3)) or
((SEITE1 = SEITE3) and (SEITE1 != SEITE2)) or
((SEITE2 = SEITE3) and (SEITE2 != SEITE1)),
GLEICHSEITIG, wenn
SEITE1 = SEITE2 =SEITE3,
KEIN_DREIECK, sonst ;

Notwendigkeit einer formalen Spezifikation:

- Gemäß der Mathematik beinhalten gleichseitige Dreiecke gleichschenklige Dreiecke.
- Die obige formale Spezifikation von GLEICHSCHENKLIG definiert abweichend von der mathematische Definition: Bei GLEICHSCHENKLIG müssen zwei Seiten ungleich sei, also disjunkte Aufteilung der Menge der Dreiecke!

Black-Box-Test - Äquivalenzklassen



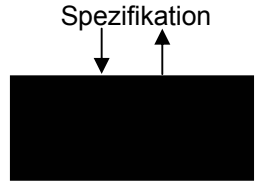
Diese Spezifikation gibt gute Anhaltspunkte für die Auswahl der Testfälle:

1. Testfälle, für ungleichseitige Dreiecke :
   10,12,5; 12,10,5; Soll-Ergebnisse : UNGLEICHSEITIG
2. Testfälle, für gleichschenklige Dreiecke :
   10,19,10; 19,10,10; Soll-Ergebnisse : GLEISCHENKLIG
3. Testfälle, für gleichseitige Dreiecke :
   5,5,5 Soll-Ergebnisse : GLEICHSEITIG
4. Testfälle, die die Bedingung nicht erfüllen :
   10,7,2; 7,10,2; 7,2,10; Soll-Ergebnisse : KEIN_DREIECK
5. Testfälle, für fehlerhafte Eingabewerte :
   10, 0, 20; 10, 0, 20; -1, -5, -10; 0,0,0; ; Soll-Ergebnisse : Typfehler

Man unterscheidet gültige Äquivalenzklassen, die zulässige Eingabewerte repräsentieren und ungültige Äquivalenzklassen, die nicht zulässige Eingabewerte darstellen.

Welche Äquivalenzklassen können wir auf Grund der Spezifikation erstellen?

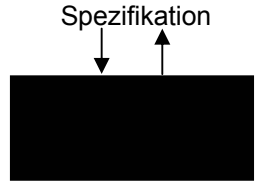
Black-Box-Test - Äquivalenzklassen



Welche Äquivalenzklassen ergeben sich auf Grund der Spezifikation?

- Äquivalenzklasse 1 : alle Zahlen ungleich,
  Repräsentant : 4,2,3
- Äquivalenzklasse 2a : 1. und 2 Zahl gleich. 3. ungleich,
  Repräsentant : 3,3,4
- Äquivalenzklasse 2b : 1. und 3 Zahl gleich. 2. ungleich,
  Repräsentant : 3,4,3
- Äquivalenzklasse 2c : 2. und 3 Zahl gleich. 1. ungleich,
  Repräsentant : 4,3,3
- Äquivalenzklasse 3 : 3 gleiche Zahlen,
  Repräsentant : 3,3,3
- Äquivalenzklasse 4 : Zahlen für die kein Dreieck konstruiert werden kann (eine Seitenlänge länger oder gleich der Summe der beiden anderen Seiten) ,
  Repräsentant : 1,2,5
- Äquivalenzklasse 5: fehlerhafte Eingabewerte,
  Repräsentant: -1,-15,0 (ungültige Äquivalenzklasse)

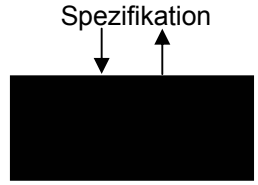
Black-Box-Test - Äquivalenzklassen



Folgende Regeln sollten beachtet werden:

- Falls Eingabetyp = Wertebereich,
  - z. B. Tage = (1 . . 31)
  - eine gültige Äquivalenzklasse (1 <= Tage <= 31)
  - zwei ungültige Äquivalenzklassen (Tage < 1, Tage > 31)
- Falls Eingabetyp = Aufzählungstype
  - z. B. Zahlungsart = (Scheck, Überweisung, bar)
  - für jeden Wert eine gültige Äquivalenzklasse (mit einem Repräsentant)
  - und eine ungültige Äquivalenzklasse (z.B. gemischte Zahlungsweise)
- Ist anzunehmen, dass Elemente einer Äquivalenzklasse unterschiedlich verarbeitet werden
  - z. B. gleichschenkliges Dreieck
  - evtl. Aufteilung in schmälere Äquivalenzklassen (s.o 2a, 2b, 2c)
- Falls durch Eingabetyp Eingabebedingungen beschreiben werden (z. B. erstes Zeichen muss Buchstabe sein)
  - eine gültige Äquivalenzklasse (1. Zeichen = Buchstabe)
  - eine ungültige Äquivalenzklasse (1. Zeichen = kein Buchstabe)

Black-Box-Test - Äquivalenzklassen



- **Wie ist zu testen, falls mehrere Eingabewerte existieren und zu jeden Eingabewert mehrere Äquivalenzklassen existieren?**

  -> Kombinationen aller kombinierbarer Repräsentanten

  z. B. bei 2 Eingabevariablen:

  => max. n * m Testläufe;

  n, m = Anzahl Äquivalenzklassen der Eingabevariablen

  oft weniger, da

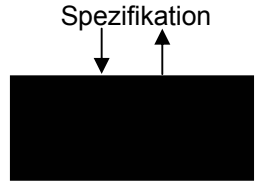
  - ungültige Äquivalenzklassen zu einem Eingabewert oft nur einmal getestet werden müssen, da unabhängig von anderen Eingabewerten (z. B. direkter Abbruch unabhängig von anderen Werten).
  - Auch Eingabewerte von gültige Äquivalenzklassen können unabhängig von anderen Werten sein.

  => zwischen n*m und n+m Testläufe

  falls zu viele Testläufe

  => Einschränkung auf Teilmenge der Eingabewertekombinationen

Black-Box-Test - Grenzwertanalyse



Grenzwertanalyse:

- Testwerte decken die Grenzwerte der Äquivalenzklassen ab.

  => Erhöhung der Quote der gefundenen Fehler

  (Nur sinnvoll, wenn Elemente einer Äquivalenzklasse auf natürliche Weise geordnet, z. B.: Tage aber nicht Zahlenart)

- Unterscheidung zu Äquivalenzklassentest :

  Nicht irgendein Element aus der Äquivalenzklasse wird als Repräsentant ausgewählt,

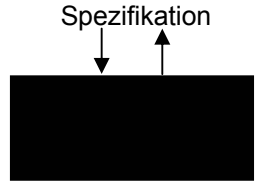
  sondern ein oder mehrere Elemente,

  so dass bezüglich der

  - Eingabewerte als auch der
  - **Ausgabewerte**,

- jeder Rand der Äquivalenzklasse getestet wird.

Black-Box-Test - Grenzwertanalyse



- Wie ist zu testen, falls mehrere Eingaben existieren und zu jeder Eingabe mehrere Äquivalenzklassen existieren?

  -> Kombinationen aller kombinierbarer Grenzwerte

  - bei 2 Eingabevariablen:

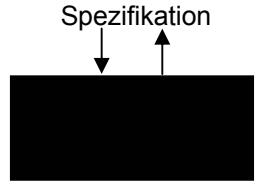
    => max. n * g1 * m * g2 Testläufe;

    n, m = Anzahl Äquivalenzklassen,

    g1, g2 = Durchschnittliche Anzahl der Grenzwerte pro Äquivalenzklasse

    oft weniger, da oft Werte voneinander unabhängig (s. Äquiv.-Test)

Black-Box-Test – Intuitive Testfallermittlung



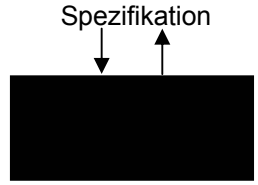
Ein erfahrener Tester kennt die "beliebten" Fehler und Umsetzungsvarianten der Entwickler!

- der Tester macht Annahmen über die (wahrscheinliche) Umsetzung innerhalb der Black-Box und ergänzt die Testfälle entsprechend.
- Oft findet man dabei Spezialfälle, die auch bei der Spezifikation übersehen wurden.
- Intuitive Testfälle liefern nur dann gute Ergebnisse, wenn Tester und Entwickler unterschiedliche Personen sind.

Beispiele

- Wert 0 oder MaxInt+1 bei Rechenoperationen
- Steuerzeichen in Zeichenketten
- Kein Eintrag oder leerer Eintrag bei Tabellenverarbeitung
- Gleichzeitiges oder andauerndes Drücken von Tasten
- ...

Black-Box-Test – Beispiel Fakultät



Übung: N-Fakultät

- Für welche Werte wird das Programm getestet?
  a) nach der Methode des Äquivalenzklassentestes
  b) nach der Methode der Grenzwertanalyse
- Finden Sie als erstes die Äquivalenzklassen und anschließend die zu testenden Werte.

Spezifikation des Moduls FAKULTÄT :

integer NFAK (integer N) {…}

ZWECK: Berechnung der Fakultät : N! = N * (N-1) *…3*2*1

Ergebnis:

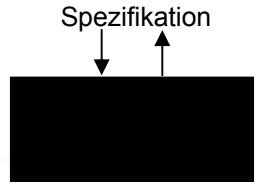
1 wenn N = 0 or N = 1

2 wenn N = 2

N! wenn N > 2

-1 wenn N < 0

Black-Box-Test – Beispiel Fakultät: Lösung



1. Äquivalenzklassen

   gültige Klassen:]-∞,0[, [0,1] und]2, ∞[, 2

   ungültige Klassen: nicht natürliche Zahlen

   - Testfälle: (-5,-1), (0,1), (5, 120), [(1.5, UNDEF)]

2. nach der Methode der Grenzwertanalyse

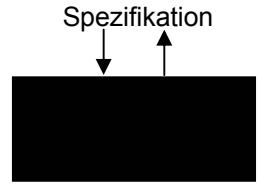
   - ergänze den Testfall (1,1)

3. Intuitive Testfälle

   Die Funktion wächst so schnell, dass die Berechnung oft durch eine Wertetabelle ersetzt wird
   (FAK(13)>MAX_INT, FAK(21)>MAX_LONG)

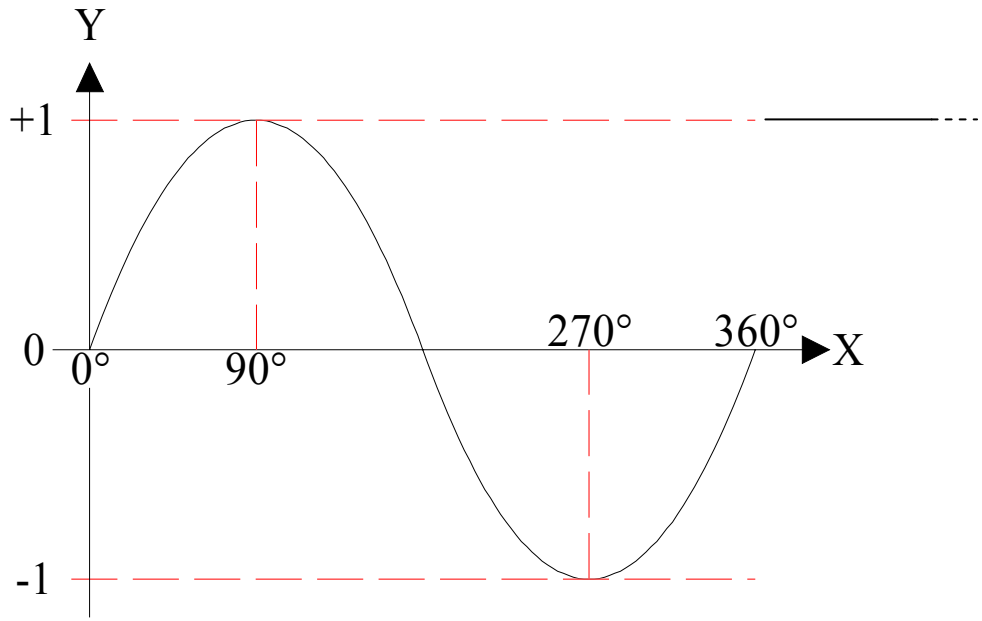
   - Teste *jeden* einzelnen Wert bis zur maximalen Größe
   - Überprüfe was bei Überschreitung des Zahlenbereichs geschieht

Black-Box-Test - Grenzwertanalyse

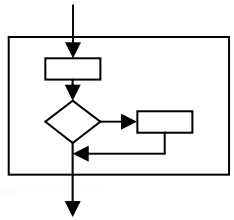


Übung: Sinus1 >= 0 Grad
Für 0°<= X <= 360°: Y=sin X,
Für X > 360 Grad: Y = 1

Wo liegen Äquivalenzklassen ?
Wie sieht die Grenzwertanalyse aus ?



White-Box-Test (Glas-Box-Test, Strukturtest)



zur Wiederholung:

- Man schaut in den Kasten hinein
- Die innere Struktur (das "Wie") des Programms wird zur Überprüfung hinzugezogen

statische Analyse

- Analyse des Quellcodes (siehe Kapitel Review, Codeinspektion)

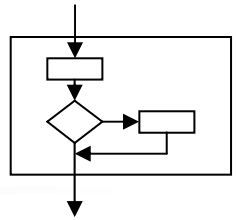
dynamische Analyse

- Testfälle werden aus der inneren Struktur abgeleitet. Es wird geprüft, ob alle
  - Anweisungen,
  - Zweige bzw.
  - Pfade

  des Programms durchlaufen werden (evtl. Einbau von Zählern an strategisch wichtigen Stellen)

  bzw. ob alle
  - Bedingungsteile von zusammengesetzten Bedingungen mindestens einmal true und einmal false sind.

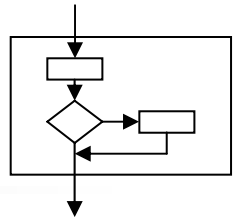
White-Box-Test - Anweisungsüberdeckung



vollständige Anweisungsüberdeckung:

- Jede Anweisung des zu testenden Programms wird mindestens einmal ausgeführt

White-Box-Test - Zweigüberdeckung



vollständige Zweigüberdeckung:

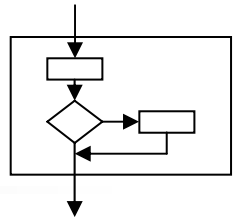
- Jeder Zweig des Programms wird mindestens einmal durchlaufen
- Was ist ein Zweig?
- *Zweig* : = Kante im gerichteten Programmgraphen.
- Was ist ein gerichteter Programmgraph?
- Ein gerichteter Programm-Graph, ist ein Graph, der jeden möglichen Kontrollfluss des Programms enthält (entspricht dem Aktivitätsdiagramm bzw. dem Programm-Ablauf-Plan (PAP)).
- Testfälle werden so ausgewählt, dass jeder Zweig des Programms mindestens einmal durchlaufen wird.
- Evtl. automatische Testfallerzeugung per Zufallszahlengenerator.

  Dann: Prüfen, welche Zweige durchlaufen wurden und

  manuelles suchen von Testfällen zur Abdeckung der restlichen Zweige.
- Instrumentierung: In Zweigen werden durch Testwerkzeug Zähler eingebaut.

  => Ableitung des Grades der Überdeckung, Anzeige der nicht durchlaufenen Zweige.

White-Box-Test - Zweigüberdeckung



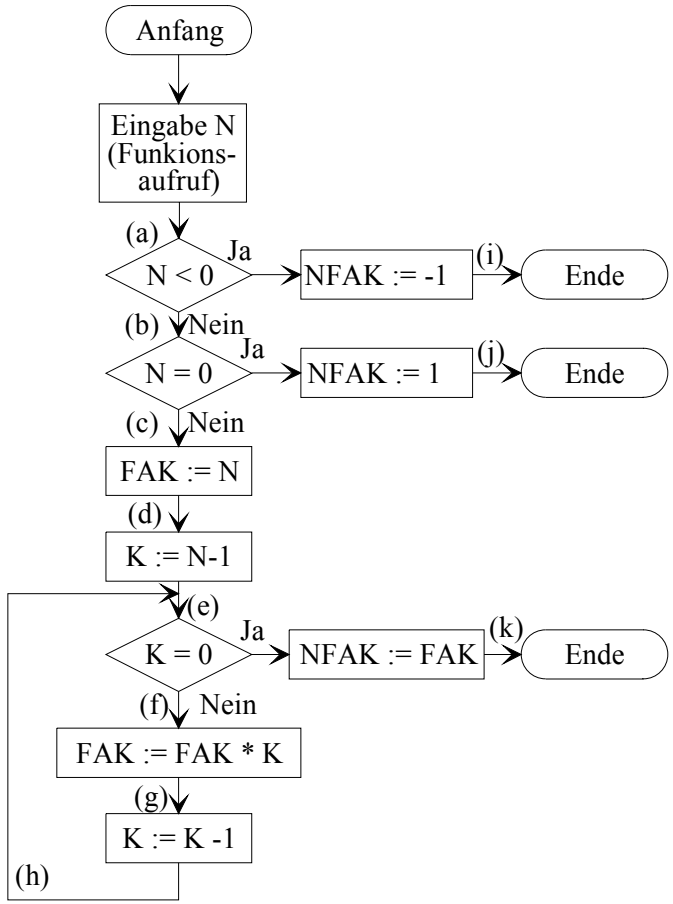


Abbildung: PAP der Funktion FAKULTAET

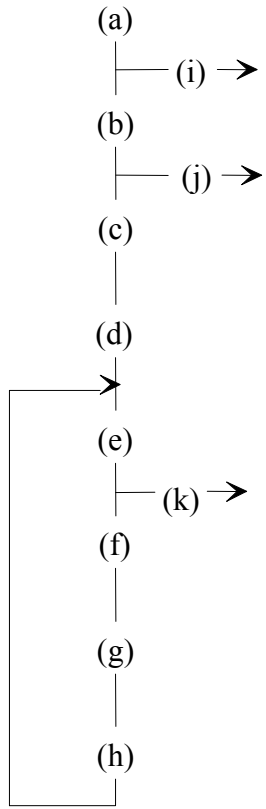
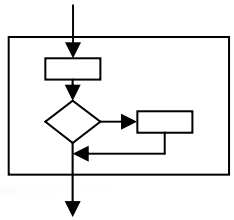


Abbildung: Gerichteter Programm-Graph der Funktion FAKULTAET

White-Box-Test - Zweigüberdeckung

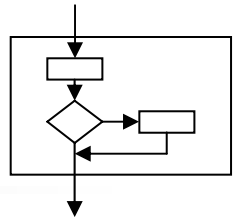


Wie viele Testfälle?

Welche Zweige?

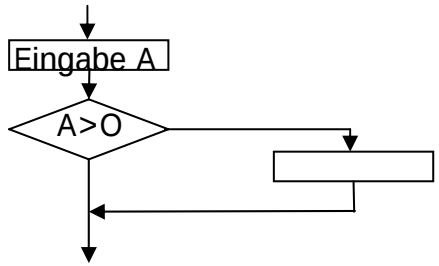
	Zweig	Eingabewert N	Soll-Ergebnis NFAK
Testfall 1	a, i	N < 0	-1
Testfall 2	a, b, j	0	1
Testfall 3	a, b, c, d, e, f, g, h, k	2	2

White-Box-Test – Zweig-/Anweisungsüberdeckung



Was ist der Unterschied zwischen Zweigüberdeckung und Anweisungsüberdeckung?

- Bei Anweisungsüberdeckung wird Zweig nur durchlaufen, wenn Anweisung im Zweig vorhanden
- also: Bei Zweigüberdeckung wird in einer if-Anweisung else-Zweig auch durchlaufen, wenn keine Anweisung darin enthalten ist, bei Zweigüberdeckung nicht.



Bei vollständiger Zweigüberdeckung: mindestens wie viele / welche Testfälle?

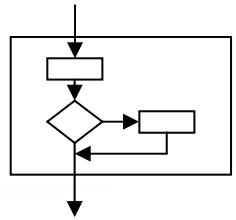
2 / a=0, a=1

Bei vollständiger Anweisungsüberdeckung: mindestens wie viele / welche Testfälle?

1 / a=1

Unterschied bei FAKULTÄT-Beispiel?

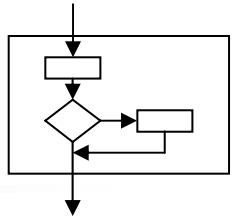
White-Box-Test -Pfadüberdeckung

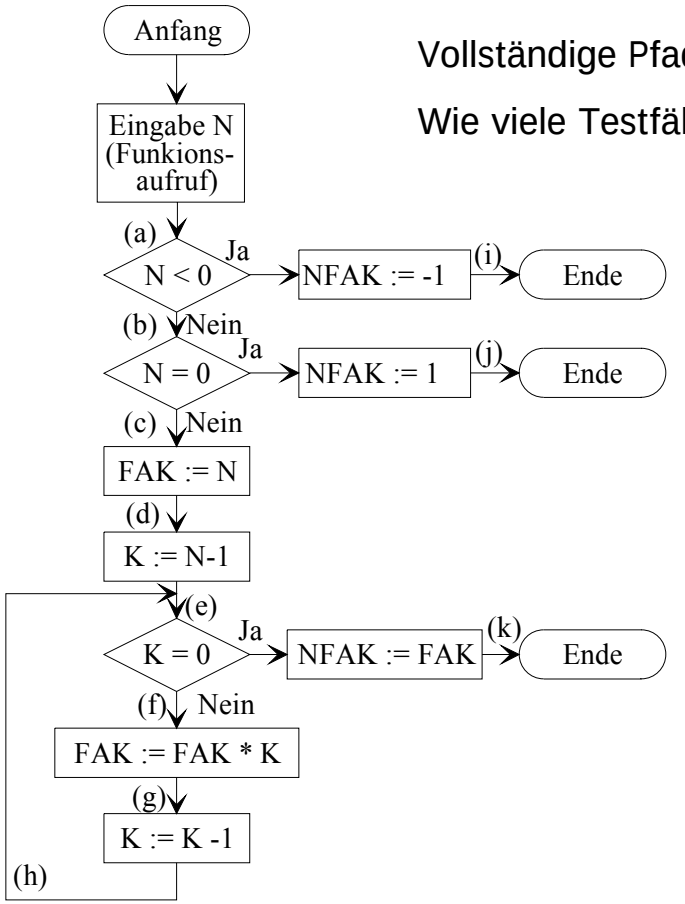


vollständige Pfadüberdeckung:

- Pfad := Weg durch den *Ausführungsbaum* (nicht durch PAP!) von seinem Anfang bis zu seinem Ende.
- Der Ausführungsbaum enthält *alle möglichen* Abläufe im Programm. Die Testfälle werden so ausgewählt, dass jeder Pfad im Ausführungsbaum mindestens einmal durchlaufen wird.

White-Box-Test - Pfadüberdeckung



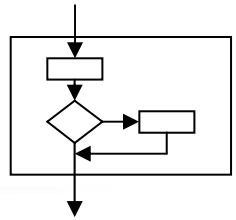


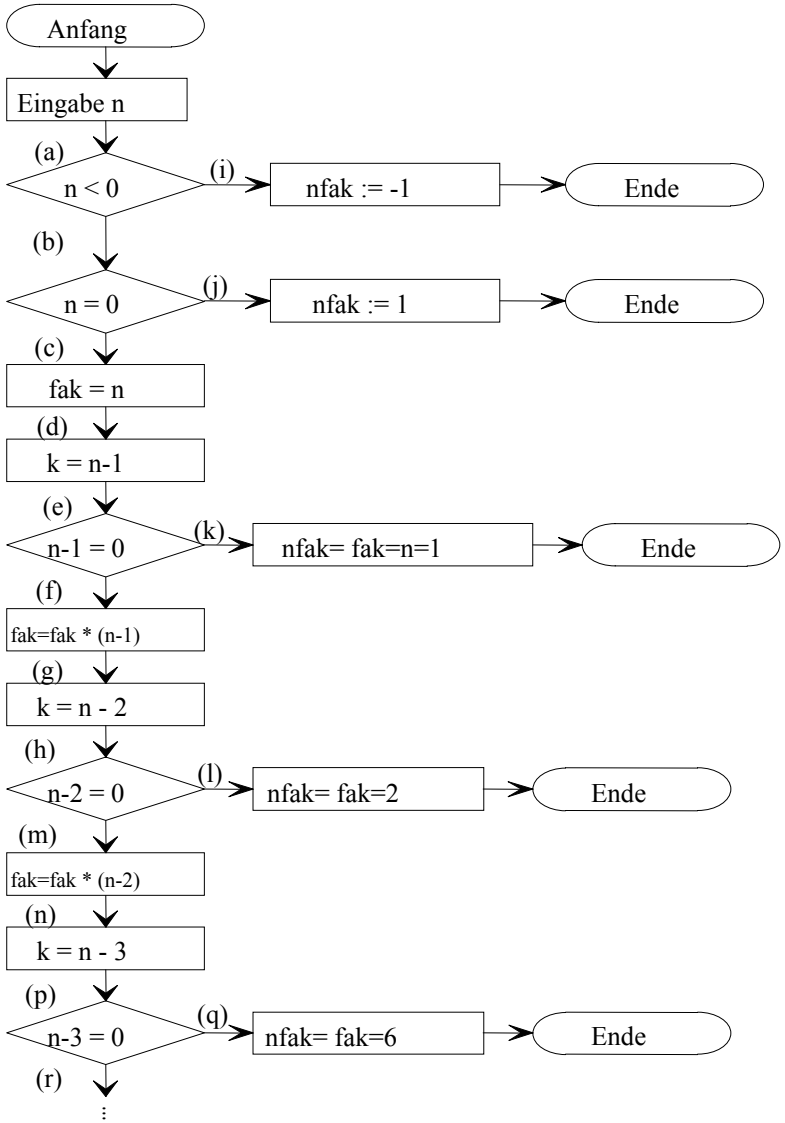
Vollständige Pfadüberdeckung:

Wie viele Testfälle müssen wir laufen lassen?

Abbildung: PAP der Funktion FAKULTAET

White-Box-Test -Pfadüberdeckung



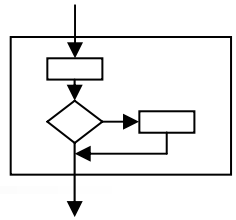


Es ergeben sich folgende Ergebnisse :

n < 0 : nfak = -1
n = 0 : nfak = 1
n = 1 : nfak = 1
n = 2 : nfak = 2
n = 3 : nfak = 6
usw.

Abbildung: Ausführungsbaum der Funktion FAKULTAET

White-Box-Test - Pfadüberdeckung

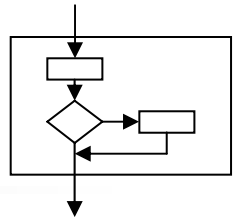


Testfälle :

	Pfad	Eingabewert N	Soll-Ergebnis NFAK
Testfall 1	a, i	N < 0	-1
Testfall 2	a, b, j	0	1
Testfall 3	a, b, c, d, e, k	1	1
Testfall 4	a, b, c, d, e, f, g, h, l	2	2
Testfall 5	a, b, c, d, e, f, g, h, m, n, p, q	3	6
:			
.			

Es kann zu vielen bzw. wie hier zu unendlich vielen Pfaden kommen (abgesehen von den Grenzen der Darstellbarkeit der Zahlentypen im Rechner).
=> 100%-ige Pfadüberdeckung kann in der Regel nicht erreicht werden.

White-Box-Test - Bedingungsüberdeckung



vollständige Bedingungsüberdeckung:

- Testfälle werden so ausgewählt, dass alle der mit AND oder OR verbunden (Teil-) Bedingungen in der Auswahl alle möglichen annehmen Zustände (mindestens einmal true und mindestens einmal false).

Beispielprogramm:

```
IF (Z="A") OR (Z="E") OR (Z="I") OR (Z="O") OR (Z="U")
{
        cout ("VOKAL")
ELSE
        cout ("KONSONANT")
}
```

Testfälle?

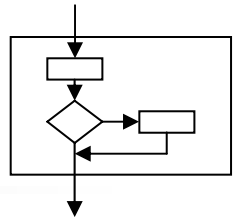
Eingabe A, E, I, O, U, B

Beim Zweigtesten?

nur A, B

Vorteil Bedingungstest: Fehler bei der Erstellung der Bedingungen werden auch erkannt.

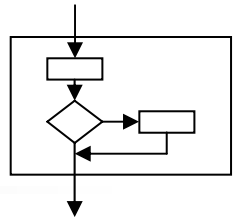
Vergleich der Testprinzipien des White-Box-Tests



Ergebnisse für vollständige Abdeckung	Anweisungs-überdeckung	Zweig-überdeckung	Pfad-überdeckung	Bedingungs-überdeckung
Fehlerhafte Anweisung	☺	☺	☺	☺
Vergessene Else-Anweisung	☹	☺	☺	☺
schwierige Fehler in IF-Bedingungen	☹	☹	☺	😐
Entdecken von konstanten Teilbedingungen im IF	☹	☹	☹	☺
Rand-Fehler in komplexer Bedingung (x>0 statt x ≥0)	☹	☹	☹	☹
Anzahl von Testfällen	viele	viele	sehr viele	viele

☺ wird entdeckt
☹ wird nicht entdeckt
😐 wird tlw. entdeckt

Kombination von Testprinzipien

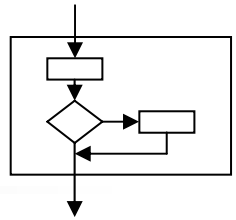


Häufig treten in der Software Fehler auf, weil eine If-Abfrage an der Grenze zwischen den beiden Fällen unsauber formuliert ist (z.B. $x>0$ statt $x\geq 0$)

Kombiniere die Zweigüberdeckung mit der Grenzwertanalyse!

- d.h. teste nicht nur einen Wert pro Zweig, sondern die Grenzwerte!
  - Werte, die - falls sie etwas größer bzw. kleiner wären - den Ablauf in den anderen Zweig steuern würden
  - Damit sind Grenzwert-Fehler bei Abfragen in Bedingungen beim Zweigtesten auffindbar
  - (Durch Setzen von Zählern in Zweigen können wir sehen, welcher Zweig wirklich durchlaufen wurde.)

White-Box-Test - Toolunterstützung



- Ohne Tool sind systematische White-Box-Tests für größere Programme kaum machbar
  - Testüberdeckungs-Tools
    - führen einige Testfälle durch
    - markieren anschließend durchlaufene Anweisungen, Zweige und Bedingungen
  - man muss "nur noch" fehlende Tests ergänzen
  - Erkennung von schwer / nicht testbaren Programmteilen
  - Überarbeitung des Codes zur Verbesserung der Testbarkeit
    - Einbau von Zugriffsmöglichkeiten zu Testzwecken (z. B. Setzen von Variablen an speziellen Stellen, ...)
    - Entfernen von unbenutztem Code

White-Box-Tests und Black-Box-Tests betonen verschiedene Aspekte beim Testen

Strukturorientierte Tests (‚White-Box-Test')

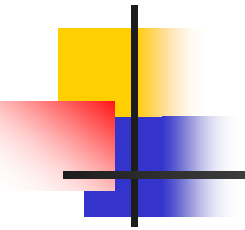
Typische Fragestellungen:

- Durchlaufen die Testdaten den richtigen Weg im Programm?
- Werden bestimmte Teile des Programms in bestimmten Situationen ausgeführt?
- Ist die Anzahl der Ausführungen korrekt?

Funktionsorientierte Tests (‚Black-Box-Test')

Typische Fragestellungen:

- Vollständige und korrekte Realisierung des gewünschten Funktions- und Leistungsumfangs?
- Sind alle Teilfunktionen der Spezifikation korrekt realisiert?
- Sind sie über den vorgesehenen Bereich realisiert?
- Werden Spezialfälle funktional korrekt behandelt?



Stärken und Schwächen von Black-Box- und White-Box-Tests

Bewertung Strukturtest (White-Box-Test)

- Fehlende Funktionalitäten werden **nicht** erkannt
- Ist eine spezifizierte Funktion **nicht** implementiert, wird dies nicht notwendig erkannt
  - Allein der Funktionstest erkennt derartige Fehler zuverlässig

Bewertung Funktionstest (Black-Box-Test)

- Keine Berücksichtigung der Implementierung
- Spezifikation besitzt ein höheres Abstraktionsniveau als die Implementierung
- Testfälle werden allein aus der (abstrakteren) Spezifikation abgeleitet
  - Ein vollständiger Funktionstest erfüllt daher in der Regel nicht die Minimalanforderungen einfacher Strukturtests
  - Der Funktionstest führt oft nur zu einer Zweigüberdeckungsrate von ca. 70%

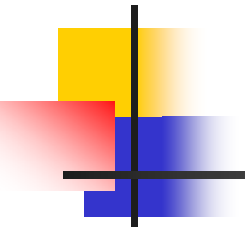
Empfehlung: Funktions- und Strukturtestverfahren miteinander kombinieren

Teststrategie

1. Testfälle nach Black-Box-Test suchen
2. Ergänzende Testfälle nach White-Box-Test (dynamische Analyse) suchen
   - nach Zweigüberdeckung
   - evtl. zusätzliche Testfälle gemäß Bedingungsüberdeckung
   - evtl. teilweise Pfadüberdeckung
3. Zusätzlich intuitive Testfälle suchen

Aber: Zitat von Dijkstra

- "Program testing can be used to show the presence of bugs, but never to show their absence!"



Teststrategie

Ziel des Testens

- Ziel soll nicht sein, die Fehlerfreiheit nachzuweisen,
- sondern: so viele Fehler wie möglich zu finden
  - Psychologie: Prüfer muss motiviert werden, Fehler zu finden !

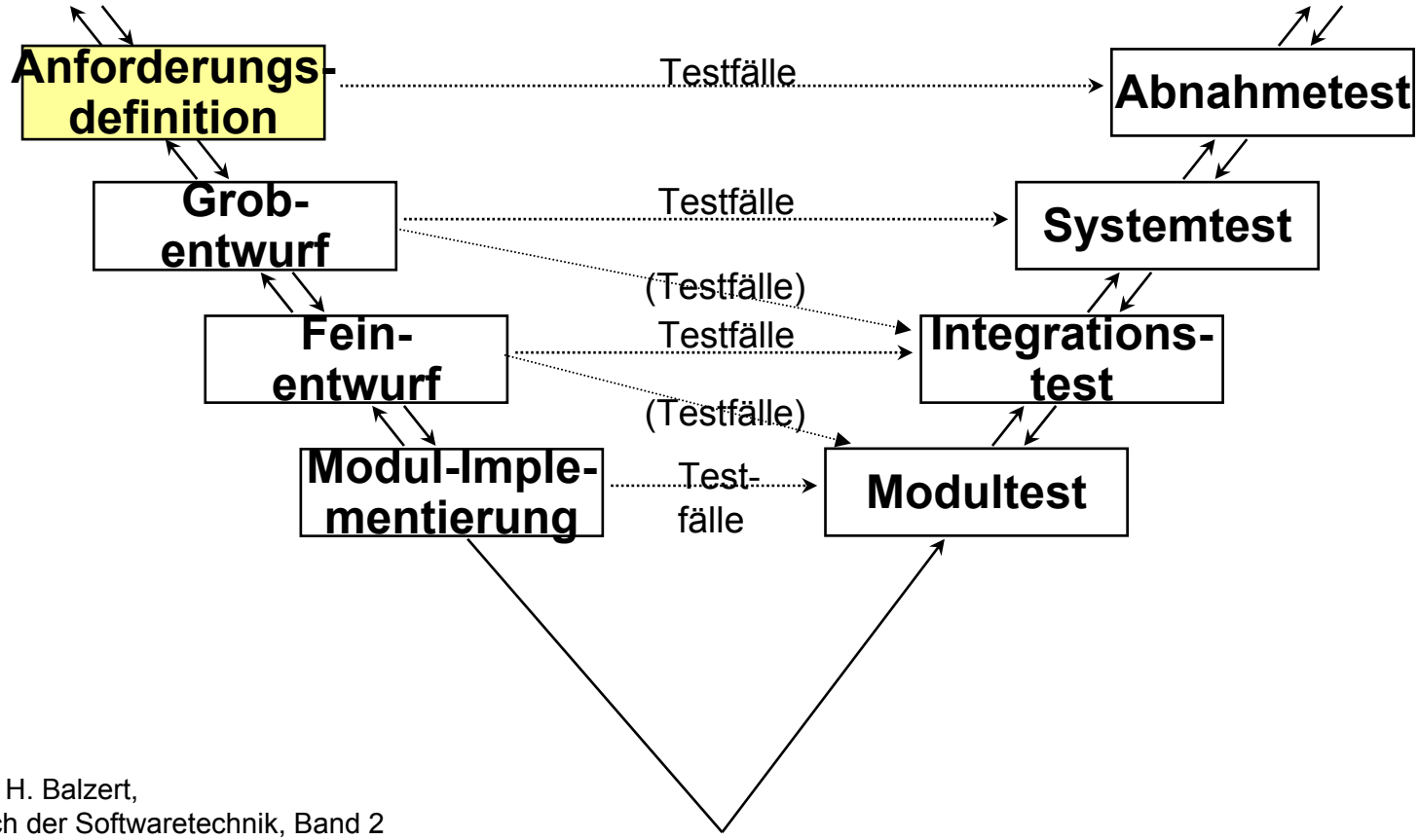
Teststrategie

Ablage von Testdaten

- Tests müssen reproduzierbar sein, z.B. auch nach einer Programmänderung muss die Funktion erneuert getestet werden.

  => Eingabe- und Ausgabedaten müssen aufgehoben werden
- evtl. Testautomatisierung mit Hilfe von Testwerkzeugen.
  - Bei Erstellung des Testplans:
    - Ein- / Ausgabedaten werden vor dem 1. Test hergeleitet und in Testdateien zu Funktionen gespeichert.
  - Beim Testlauf:
    - Der "Driver" liest die abgespeicherten Eingabedaten, gibt diese an die Funktion weiter und nimmt die Ausgabedaten zurück. Daten, die aus Datenbanken gelesen werden, Datenbankzustände, globale Variablen, Attribute der Klasseninstanzen müssen auch gesetzt werden.
  - Nach Testlauf:
    - Der "Driver" liest die abgespeicherte Soll-Ausgabedaten von der Datenbank, vergleicht diese mit den Ist-Ausgabedaten und erstellt Fehlermeldung. Daten, die in Datenbanken geschrieben werden, Datenbankzustände, globale Variablen, Attribute der Klasseninstanzen können auch Ausgabedaten sein.

Einordnung im Phasenmodell: V-Modell



Quelle: H. Balzert,
Lehrbuch der Softwaretechnik, Band 2

Die Testspezifikation sollte jeweils an den Phasenenden erfolgen!

Was ist der Modultest?

Test eines einzelnen Teilstücks der Software - eines Moduls:

- Test einzelner Module:
  - Funktionen, Klassen
    - lokales Testen von globalen Funktionen / Funktionen in Klassen
    - Testen von Funktionen mit Aufruf von Funktionen der gleichen Klasse
  - Ketten von durch z. B. Assoziationen / gegenseitigen Aufrufen verbundenen Klassen
    - Testen von Funktionen mit Aufruf von Funktionen anderer Klasseninstanzen
  - Komponenten (= eigenständige SW-Einheit mit definierter Verantwortlichkeit; sie ist über eine definierten Schnittstelle von außen nutzbar; sie besteht aus mehreren Klassen. – siehe später in dieser Vorlesung)
    - Aufruf der Komponente über die Schnittstelle

      – Test sehr schwierig,

      da innere Struktur der Komponente wesentlich komplexer als Klasse

Test einzelner Funktionen

Jede Funktion wird einzeln für sich getestet! Aber wie?

Simuliere die Umgebung des Moduls durch Test-Module, die aufrufende und aufgerufene Module ersetzen!

⇨ **"Drivers" und "Stubs"**

Test einzelner Funktionen

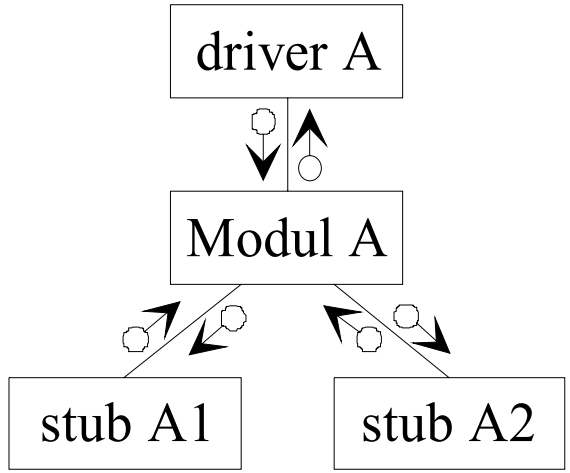
- **"Driver's"** sind Testmodule,

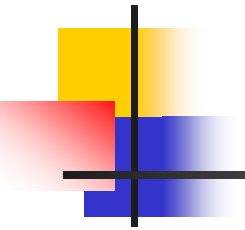
  die das **Aufruf-Verhalten** einer echten Komponente simulieren, d. h.
  - den zu testenden Modul mit Parametern versorgen
  - Ergebnisse entgegennehmen
  - Ergebnisse prüfen bzw. den Benutzer bei der Prüfung unterstützen

- **"Stub's"** sind Testmodule,

  die das **Antwort-Verhalten** einer echten Komponente (mehr oder weniger) simulieren, d. h.
  - Parameter vom getesteten Modul entgegennehmen
  - Ergebnisparameter (die evtl. "falsche" Werte besitzen können) nach oben zurückgeben

- bei der Simulation den Zustand berücksichtigen!
  - globale Variablen
  - Ein-/Ausgaben (Bildschirm, Datei)
  - ...





Test von Funktionen in Klassen

Test von einzelnen Funktionen in Klassen:

- Was müssen wir beim Test von Funktionen von Klassen beachten (zusätzlich zum Test von globalen Funktionen)?
- Die Funktion hat evtl. auch Attribute der Klasseninstanz als Eingabe-/Ausgabeparameter

Test von Aufrufhierarchien aus mehreren Funktionen

- Es werden einzeln getestete Funktionen zu Aufrufhierarchien (auch über Klassengrenzen hinweg) zusammengeführt.
  - innerhalb einer Klasseninstanz: Teil des Klassentests
  - Über Klasseninstanzgrenzen hinweg: Kettentests

    (Hier können Attributwerte der anderen Klasseninstanzen auch Ein/Ausgabewerte sein.)
- Auch wieder: Verwendung von "drivers" und "stubs"

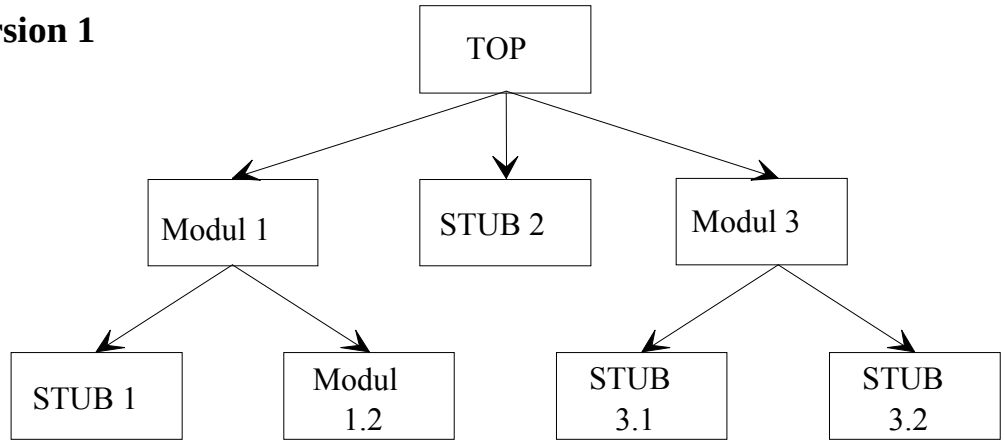
- 2 Vorgehensweisen (oder Kombination der beiden Vorgehensweisen):
  - **Top-Down-Vorgehensweise**
  - **Bottom-Up-Vorgehensweise**

Top-Down-Vorgehensweise

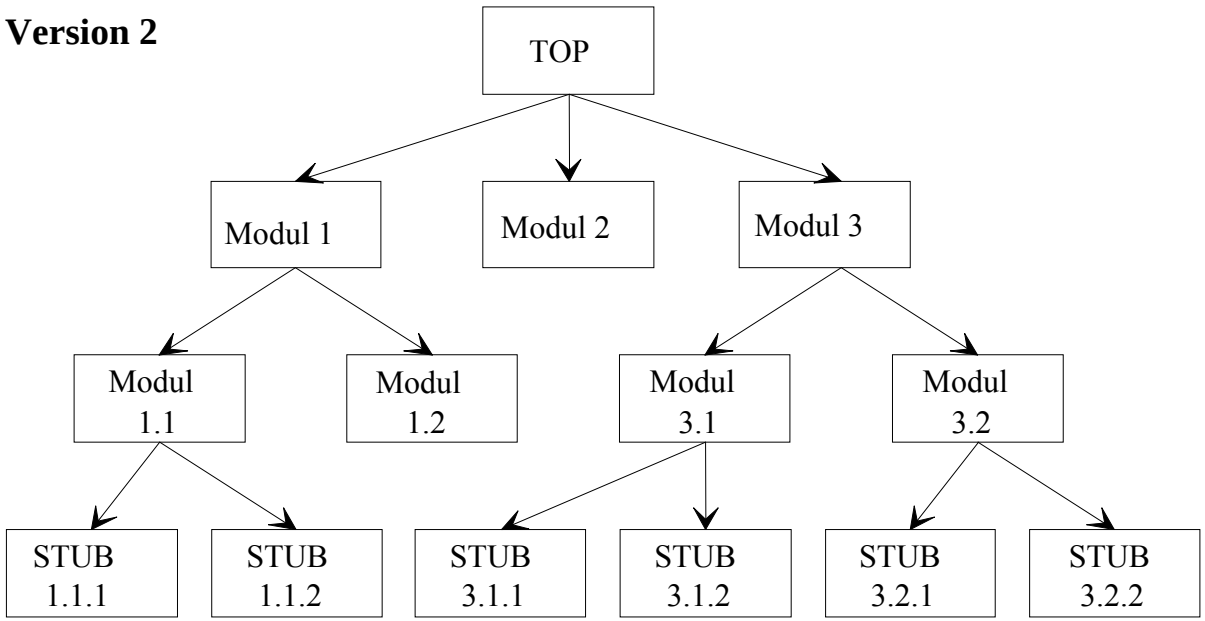
- Zuerst Implementierung, Montierung und Test der obersten Schicht,
- dann immer wieder Ersetzen der darunter liegenden Stubs durch Funktionen.
- Die noch nicht hinzugefügten Funktionen werden durch Stubs simuliert.

Top-Down-Vorgehensweise

Version 1



Version 2

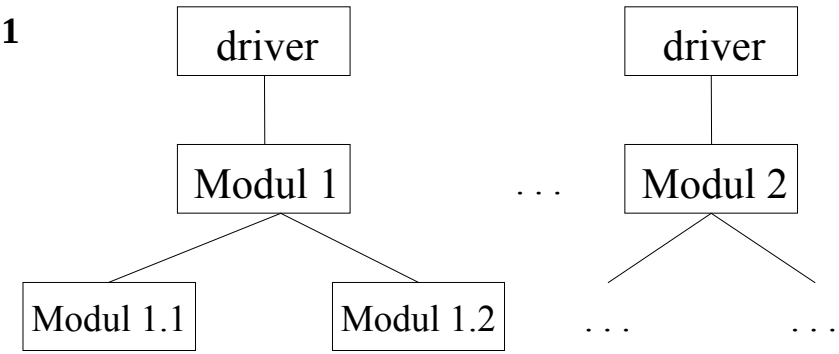


Bottom-Up-Vorgehensweise

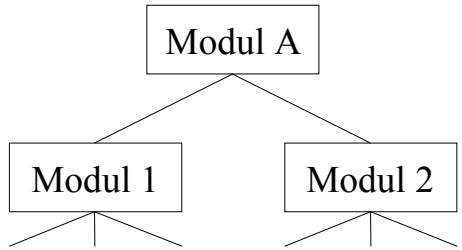
- Zuerst werden die untersten Schichten implementiert und dann schrittweise nach oben entwickelt und getestet
- Die darüber liegenden Module werden durch "driver" simuliert.

Bottom-Up-Vorgehensweise

Version 1



Version 2



Vergleich: Bottom-Up-Vorgehensweise ⇔Top-Down-Vorgehensweise:

- Vorteil Bottom-Up-Vorgehensweise:

  Ergebnisse die auf oberster montierter Schicht zu sehen sind, sind richtig, da das System darunter schon fertig ist. Es kommen keine stabs zu Anwendung.
- Nachteil Bottom-Up-Vorgehensweise:

  Der Benutzer sieht erst in den späten Phasen der Entwicklung etwas.

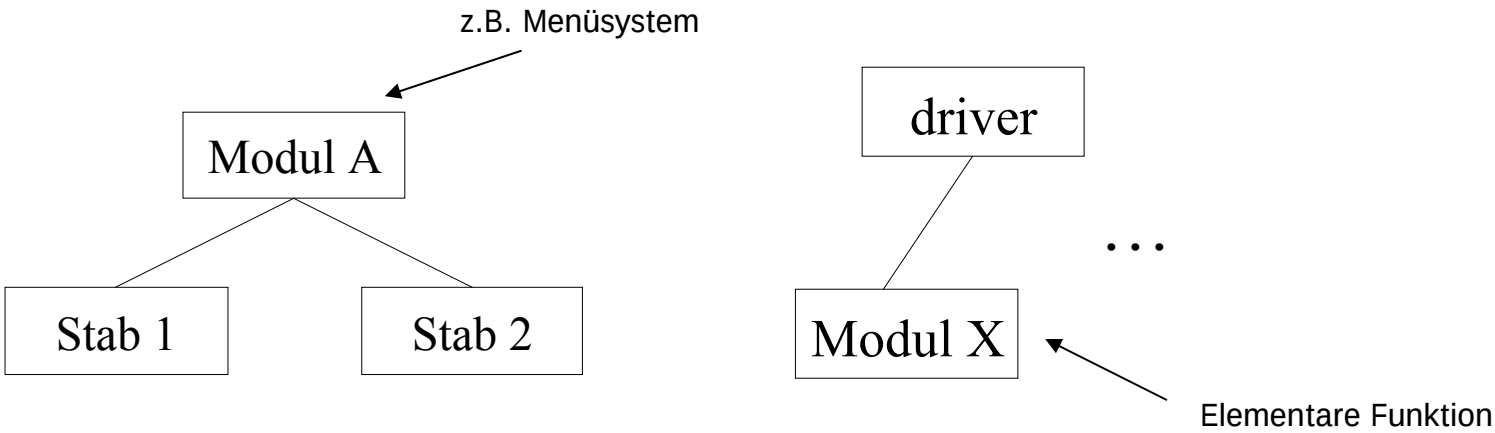
- Nachteil Top-Down-Vorgehensweise (Bei Erstellung des Gesamtsystems):

  Ergebnisse auf oberster Schicht sind nicht richtig
- Vorteil Top-Down-Vorgehensweise (Bei Erstellung des Gesamtsystems):

  Der Benutzer sieht schon zu Beginn der Integration etwas.

Kombination von TOP-DOWN- und BOTTOM-UP-Vorgehensweise

- Es kommen drivers und stubs zum Einsatz



z. B. Meet-in-the-middle-Vorgehensweise:

- von der Spitze nach unten
- von ganz unten nach oben
- Treffen irgendwo in der Mitte

Fragen

Wenn es sich bei den Funktionen nicht um globale, sondern um in Klassen definierte Funktionen handelt: Was müssen wir beachten?

- Die rufende und die aufgerufenen Funktionen haben evtl. auch Attribute der **verschiedenen** Klasseninstanzen als Eingabe-/Ausgabeparameter

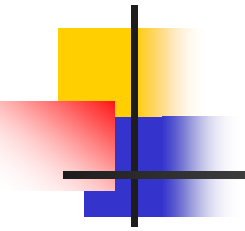
Wie stellen wir in der UML Hierarchien dar?

- z. B. als Sequenzdiagramm

Test von Komponenten

- Test von Komponenten ähnlich wie Klassentest
  - Es werden die Funktionen der Schnittstelle der Komponente getestet.
  - Da innere Struktur einer Komponente aus mehreren Klassen besteht

    => Test wesentlich komplexer und somit viel schwieriger.
  - Vorgehen:
    1. Die internen Teile der Komponente werden mit Funktions-, Klassen- und Kettentests ausgetestet.
    2. Die einzelnen Schnittstellenfunktionen der Komponente werden ausgetestet.



Integrationstest

Das Design zerlegt das System in Systemkomponenten!

- ⇨ Das Zusammenbauen von Systemkomponenten zu Teilsystemen (Sub-Systemen) heißt "Integration"
- ⇨ Der Test, ob mehrere Systemkomponenten fehlerfrei zusammenwirken heißt "Integrationstest"

Integrationstest – die Idee

- Ein komplexes System besteht aus (sehr) vielen Systemkomponenten, die zu unterschiedlichen Zeitpunkten fertig werden
- Das Lokalisieren von Fehlern in einem solchen System ist sehr schwer

Teste die Funktion von Teilsystemen und erweitere die Teilsysteme systematisch

- ⇨ führe wiederholte Tests auf dem inkrementell wachsenden Teilsystem durch
- ⇨ mache Änderungen an nur wenigen Stellen ⇨ neu entdeckte Fehler hängen mit der letzten Änderung zusammen

⇨ "Inkrementelle Integration" statt "Big Bang-Integration"

Integrationstest

Was ist ein Integrationstest?

- Prüfung, ob mehrere Module (auch externe Teilsysteme) fehlerfrei zusammenwirken
- Test von Teilen des Gesamtsystems
- Fokus auf das Zusammenspiel der Systemkomponenten

Wie führe ich einen Integrationstest durch?

- durch ein Test- bzw. Integrations-Team
- Zusammenbau des Teilsystems und Ansteuerung mit Drivers und Stubs

Was wird in einem Integrationstest getestet?

- Ein Verbund aus
  - bereits einzeln getesteten Systemkomponenten oder
  - bereits integrierten und getesteten Teilsystemen (SW-Integration)
  - auch unter Einbeziehung von externen Systemen und Hardware (System-Integration)
- Die verwendeten Module hängen von deren Verfügbarkeit zum Zeitpunkt des Tests, sowie der gewählten Integrationsstrategie ab
- Integrationsmethoden: Top-Down oder Bottom-Up

Ein Integrationstest ist ein Test für ein Teilsystem

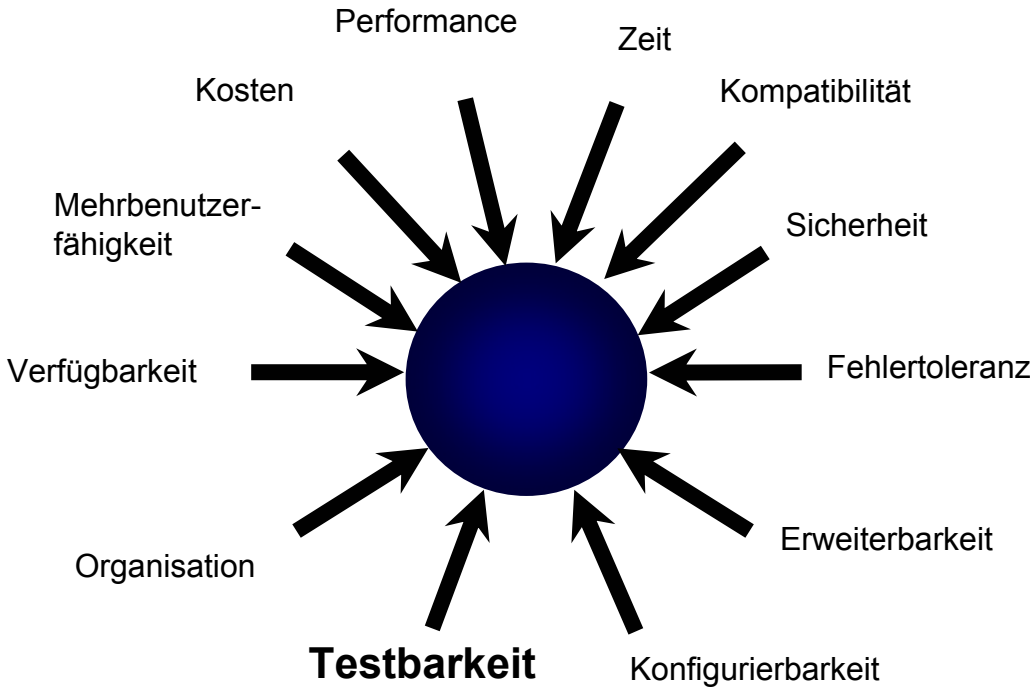
⇨ sowohl White-Box als auch Black-Box-Tests

⇨ mit Fokus auf die Interaktion der integrierten Komponenten

Systemtest, Abnahmetest

Anforderungen an das Gesamt-Software-System müssen getestet werden

- Qualitätsanforderungen
- geschäftliche Anforderungen
- wesentliche funktionale Anforderungen



In Anforderungsdefinition sind die grundlegenden Eigenschaften des Systems festgelegt!

⇨ Der Test dieser Eigenschaften wird im "Systemtest / Abnahmetest" durchgeführt

Unterschied zwischen Systemtest und Abnahmetest

Was ist ein Abnahmetest?

- Prüfung, ob erstelltes System = gewünschtes System
- die Abnahmetests sind Bestandteil des Vertrags
- Nach erfolgreicher Abnahme wird bezahlt

Wie führe ich einen Abnahmetest durch?

- Zusammen mit dem Auftragsgeber
- Mit formeller Protokollierung

Was wird in einem Abnahmetest getestet?

- Alles was spezifiziert wurde:
- Funktionstests
- Leistungstests (z. B. Antwortzeiten bei Normallast)
- Benutzbarkeitstests (z. B. einheitliche Benutzerschnittstelle)
- Volumentests (z. B. maximale Datenmengen)
- Stresstests (z. B. maximale Anforderung von Prozessorleistung)
- Robustheitstests (Prüfen der Reaktion bei unzulässigen Eingaben)

Was ist ein Systemtest?

- Prüfung, ob erstelltes System = (intern & extern) versprochenes System
- Test des integrierten Gesamtsystems
- Alles, was beim Abnahmetest getestet wird - aber noch ausführlicher

Wie führe ich einen Systemtest durch?

- ohne den Auftraggeber,

  mit Entwicklern und Qualitätssicherung in der echten Zielumgebung

Was wird in einem Systemtest getestet?

- Funktionstests, Leistungstests, Benutzbarkeitstests, Volumentests, Stresstests, Robustheitstests (ausführlicher als beim Abnahmetest)
- außerdem Sicherheit, Zuverlässigkeit, Wartbarkeit, Dokumentation, ...
- auch Installation, Datenkonvertierung, Start und Initialisierung
- auch Betriebsfunktionen wie Backup und Recovery
- auch Provozieren von Abstürzen, um Lauffähigkeit und Robustheit zu prüfen
  (Datei defekt, Plattenausfall, Datenbank korrupt, ...)

Unterschied zwischen Systemtest und Abnahmetest

Was wird in einem Systemtest getestet? Fortsetzung

- auch Berücksichtigung von Zuständen des Systems (Datenbank etc.).
- auch Berücksichtigung von Schnittstellen nach außen (andere IT-Systeme. Peripheriegeräte wie Drucker etc.).

Der Abnahmetest ist ein Blackbox-Test für das Gesamtsystem

⇨ aus Auftraggeber-Sicht

⇨ mit manuell erzeugten Testfällen

Der Systemtest ist ein Blackbox-Test für das Gesamtsystem

⇨ aus Auftragnehmer-Sicht

⇨ mit manuell erzeugten Testfällen

Abnahme- und Systemtest – Leistungs-(Performance-)Test

- Bedingungen
  - “Normale Last”
  - Messen Sie Antwortzeiten und Kapazität
- Selbstverständliche Forderungen:
  - Sofortige Antwort bei Tastendruck
  - 1-2 Sekunden für leichte Fragen
  - Ein Bescheid, wenn es länger dauert
  - Batch Jobs über Nacht fertig

Abnahme- und Systemtest - Volumentest

- Ziele
  - Maximale Datenmengen
  - Mehr als Maximum
  - Randomgenerierte oder alte Daten
  - Versuchen Sie, den ganzen Platz zu belegen
- Beobachtungen
  - Verliert das System Daten?
  - Verfälscht das System Daten?
  - Stoppt das System?
  - Braucht es enorme Zeit?

Abnahme- und Systemtest – Stresstest

- Ziele
  - Maximale Input-Geschwindigkeit
  - Input an allen Linien
  - Dasselbe Kommando an allen Linien
  - Alarm an allen Linien
  - Komplizierte Funktionen
  - Mehr als Maximalbelastung falls physisch möglich
  - Nicht alle Systemressourcen zugänglich
- Reaktionen
  - Werden Inputs vergessen?
  - Bricht das System zusammen?
  - Braucht es zu lange Zeit?
  - Werden Daten verfälscht?
- Ein ungeheuer wichtiger Test bei Realzeitsystemen!

Abnahme- und Systemtest - Benutzbarkeitstest

- Evaluation durch Experten (Ergonomen)
- Standard für Benutzerschnittstelle
- Messen der Lernkurve
- Beurteilung der Voraussetzungen

Abnahme- und Systemtest - Sicherheitstest

- Ziel: Die Zugangskontrolle des Systems zu durchbrechen
- Kontrollieren Sie die Zugangsprivilegien für jede Benutzerklasse und jede Funktion
- Möglicherweise destruktiver Test
- Kontrolle von Backup und Recovery Funktionen
- Paralleler Zugriff von mehreren Terminals
- Virus Check
- Arbeitsumfeld (Organisation)

Abnahme- und Systemtest – Weitere Tests

- Abhängig von den erwarteten und erwünschten Systemeigenschaften
  - Kunde
    - Erweiterungsfähigkeit
    - Integrationsfähigkeit
    - Zuverlässigkeit
    - Robustheit, Fehlertoleranz
  - Intern
    - Wartungsfreundlichkeit
    - Erweiterungsfähigkeit
    - Portabilität

Test-Gesamtplanung

- Erstellung (weiterer) Testpläne an den jeweiligen Phasenenden (V-Modell)
- Vor Beginn des eigentlichen destruktiven Testens, d. h. insbesondere vor schrittweiser Montage der Software:

  Erstellung Montage- und Test-Vorgehensplan

Abnahmetest - Beispiel

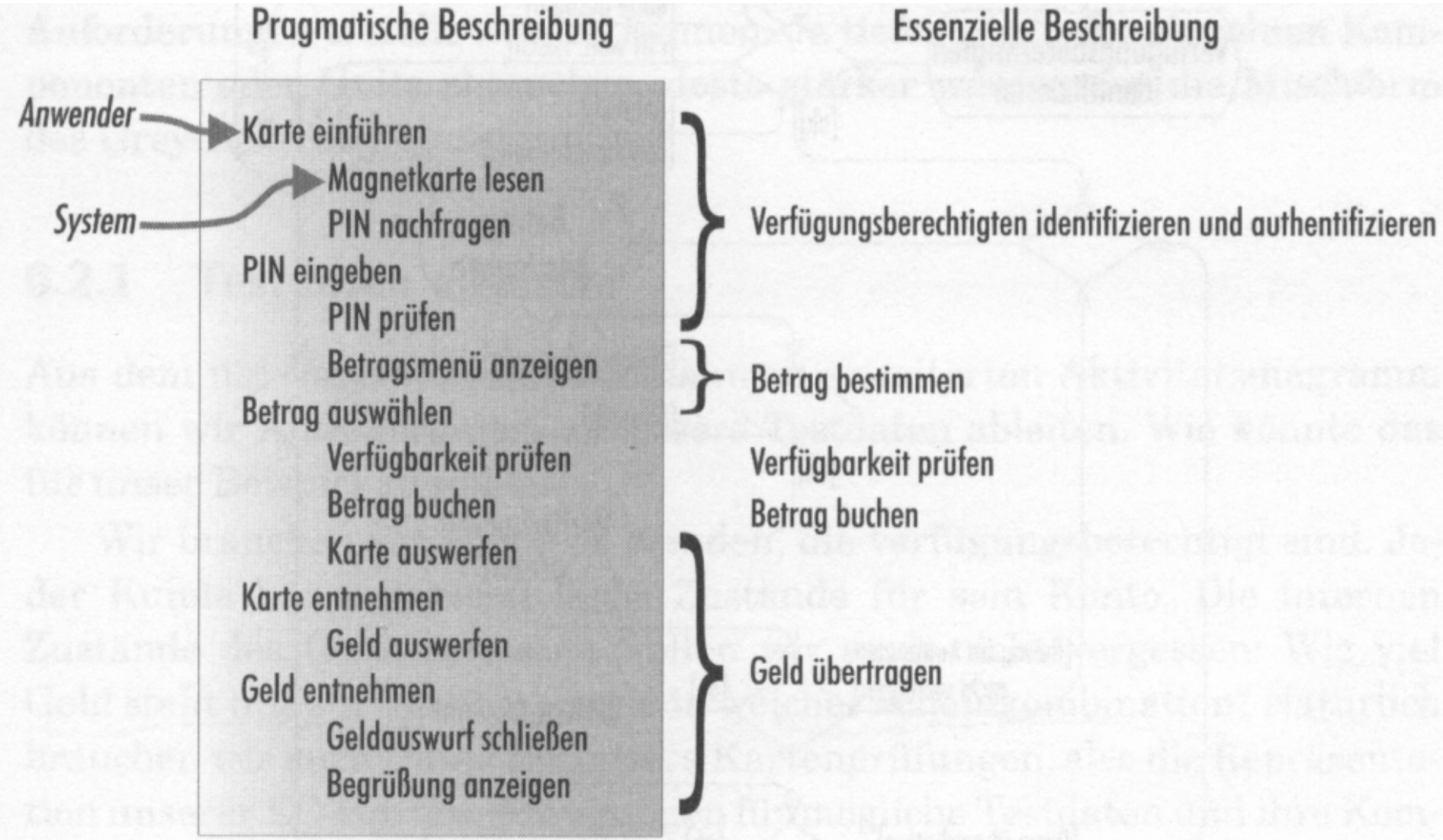
- Testen von Anwendungsszenarien (kennen Sie schon)
  - Durchlaufen aller Action-Steps (~ Anweisungsüberdeckung)
  - Durchlaufen aller Zweige im Aktivitätsdiagramm (~Zweigüberdeckung)
  - Durchlaufen aller Pfade im Aktivitätsdiagramm (~Pfadüberdeckung)
  - Bedingungsüberdeckung (Jeder Teil der Bedingung im Aktivitätsdiagramm ist mindestens einmal true und einmal false)
  - Einteilung der Bildschirmeingaben in Äquivalenzklassen (Äquivalenzklassentest / Grenzwertanalyse)

Bemerkung:

- Es handelt sich hier um einen Black-Box-Test (Man schaut nicht in den Kasten hinein, sondern kennt nur die Definition der Oberfläche.)
- Trotzdem Zweig-, Pfad-, Bedingungs-Tests.
- Historisch haben sich diese Begriffe ihren Ursprung im Test von Programmen.
- Wir haben diese Begriffe hier verwendet für das Durchlaufen von Abläufen!

Abnahmetest
- Beispiel Geldautomat - Test ableiten aus Use Case

[Wi05]



Abnahmetest
- Aktivitätendiagramm vom Use Case

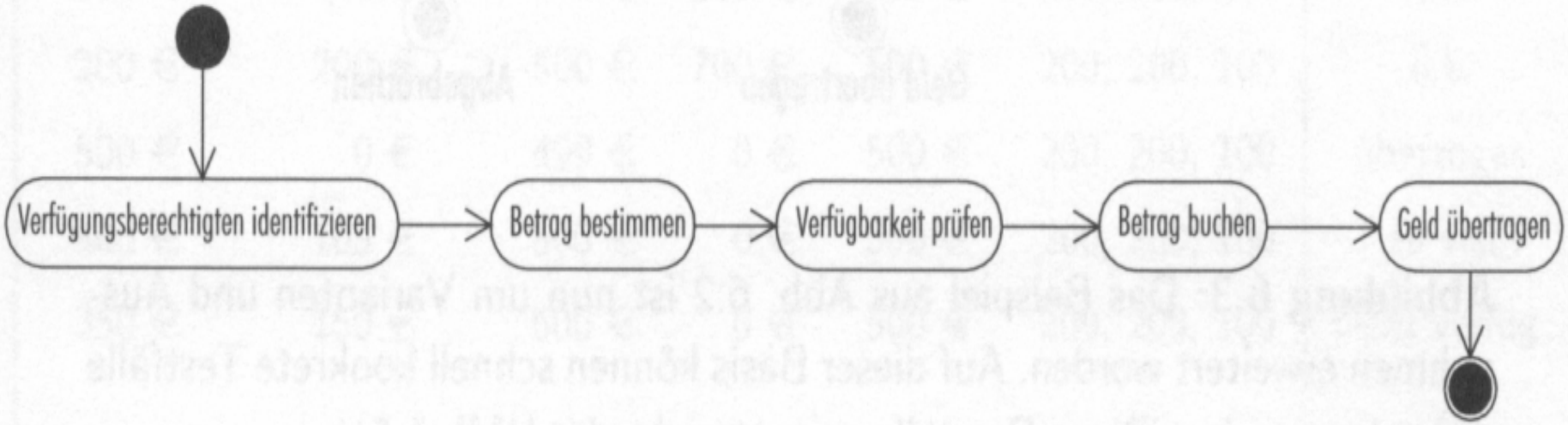
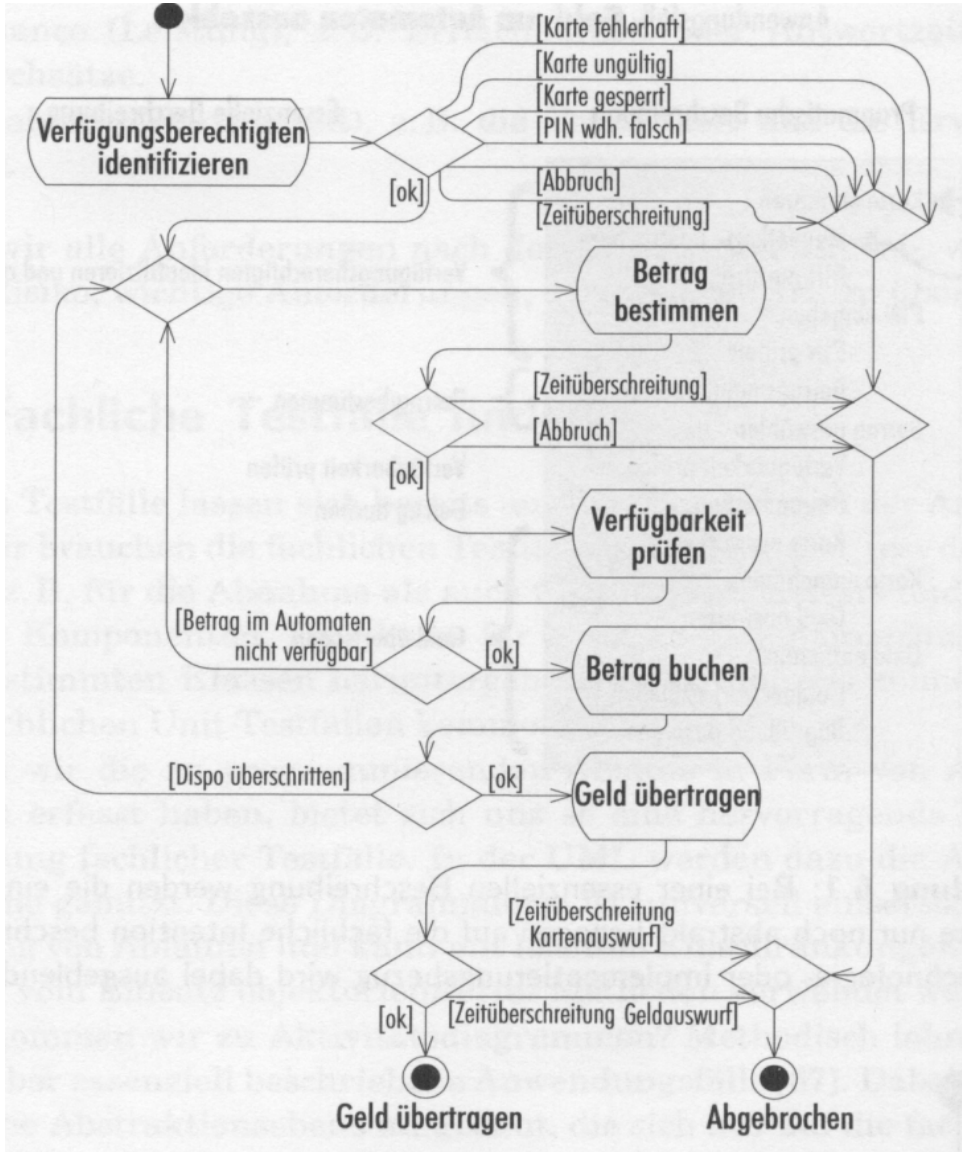


Abbildung 6.2: Beispiel eines Aktivitätsdiagramms auf Basis essenziell beschriebener Schritte eines Anwendungsfalls. Es stellt in einem ersten Schritt nur den Kontrollfluss ohne Ausnahmen dar.

Abnahmetest
- Ablauf des Use-Case incl. Ausnahmen



Aktivitätendiagramm mit Ausnahmen

Abnahmetest - Testdaten ableiten

Betrags-wunsch	bereits ab-gehoben	Saldo	Dispo	verfüg-bar	Stückelung	Ergebnis
500 €	0 €	500 €	0 €	500 €	200, 200, 100	o.k.
100 €	0 €	0 €	100 €	500 €	200, 200, 100	o.k.
200 €	200 €	-500 €	700 €	500 €	200, 200, 100	o.k.
500 €	0 €	499 €	0 €	500 €	200, 200, 100	überzogen
400 €	105 €	500 €	0 €	500 €	200, 200, 100	zu viel
350 €	150 €	500 €	0 €	500 €	200, 200, 100	nicht verfüg.
...	...	...	...	...	...	...

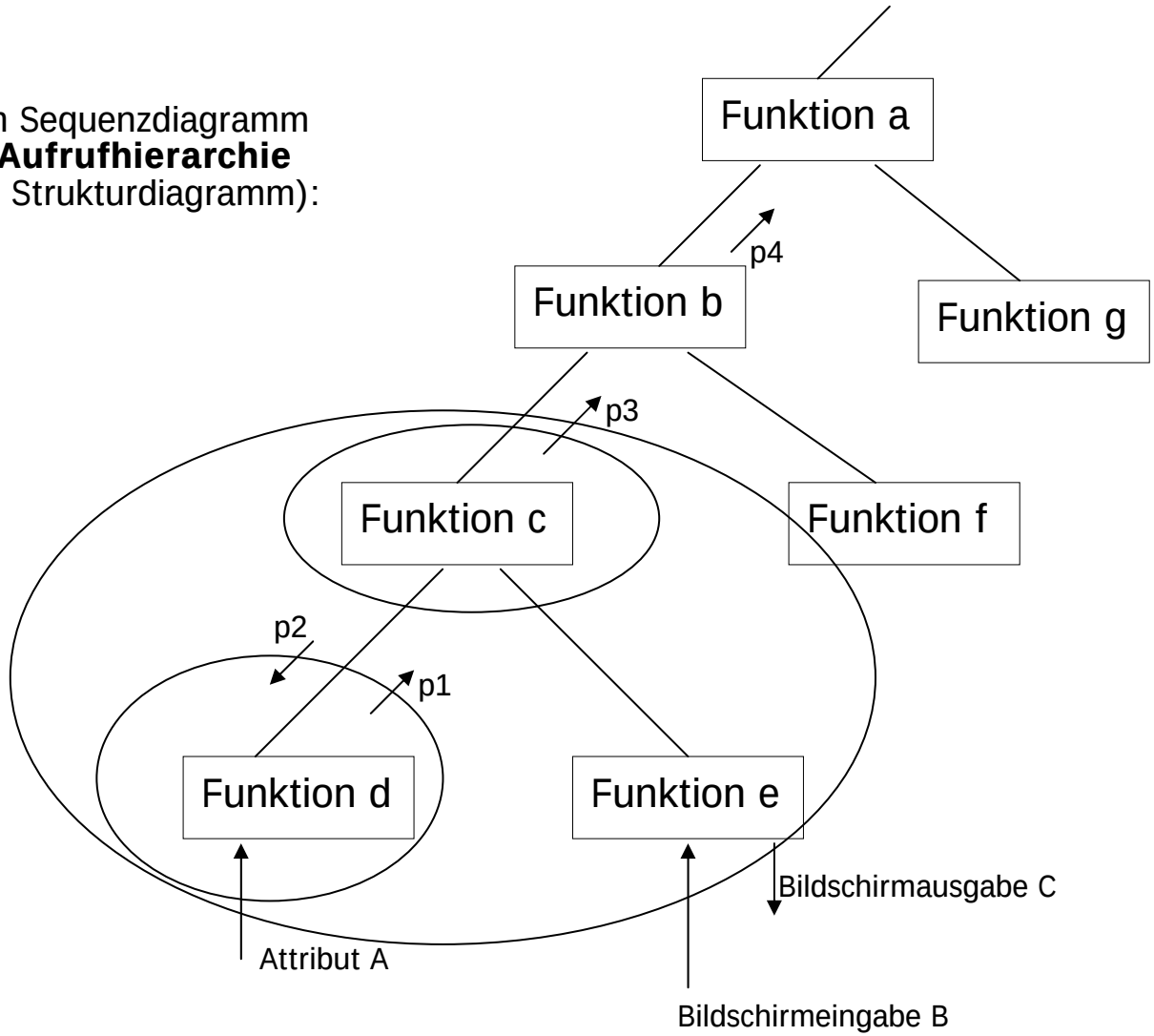
Jeder Weg im Aktivitätendiagramm zum Use-Cases ist zu durchlaufen

-> Kunden in verschiedenen Zuständen (Kto-Stand, Dispo),

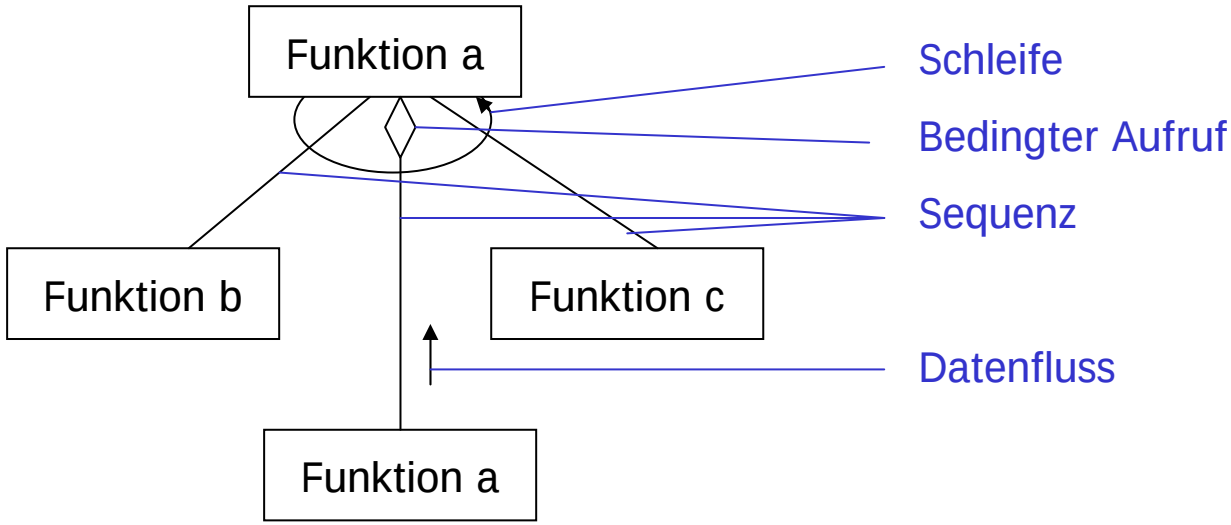
Geldautomat in versch. Zuständen

Beispiel Modul-, Integrationstest

Die aus dem Sequenzdiagramm abgeleitete **Aufrufhierarchie** (erweitertes Strukturdiagramm):



Beispiel Modul-, Integrationstest
Exkurs: Erklärung Strukturdiagramm



Beispiel Modul-, Integrationstest

2 Testfälle für Funktion d

(z. B. hergeleitet über Grenzwertanalyse + Zweigtesten + ...):

	Eingaben:		Soll-Ausgaben:	
	Parameter p2	Attribut A	Parameter p1	OK? (Nach erfolgreichen
Testfall1:	2	3,4	6,4	Test abzuhaken)
Testfall2:	2	4,8	9,6	

Beispiel Modul-, Integrationstest

1 Testfälle für Funktion c

(Funktion d und Funktion e werden durch Stubs simuliert –
Richtigkeit der Ergebnisse kann normalerweise nicht nachgeprüft werden):

	Eingaben:	Soll-Ausgaben:	
	Parameter p1 (über Stub)	Parameter p3	OK? (nur nachprüfbar, falls P3 allein abhängig von p1)
Testfall1:	8	true	

Beispiel Modul-, Integrationstest

2 Testfälle für Funktionshierarchie der Funktionen c, d, e:

	Eingaben:		Soll-Ausgaben:		
	Attribut A	Bildsch.eing. B	Parameter p3	Bildsch.ausg. C	OK?
Testfall1:	9	3	true	"9 Gramm Rum"	
Testfall2:	22	4	false	"22 Gramm Soda"	

Beispiel Modul-, Integrationstest

Wann Setzen der Attribute?

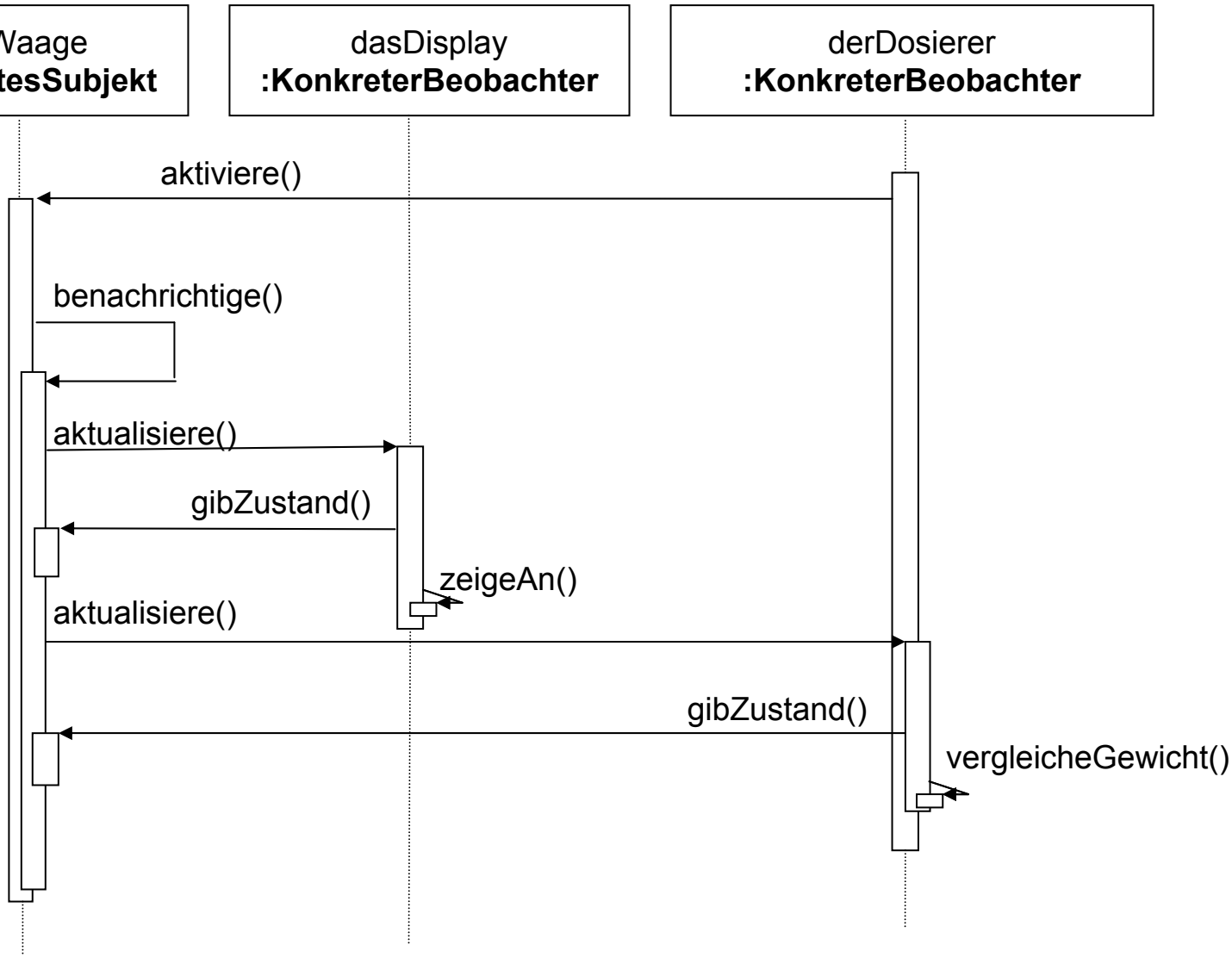
- Vor Beginn des Testens:
  - Instanziierung der Objektinstanzen
  - Setzen der benötigten Attribute
    - (falls privat: Testprogramm als friend vereinbaren)

Wo Übergeben, Annehmen der Parameter?

- Im Driver

Beispiel Modul-, Integrationstest - Aufgabe Beobachtermuster (Observer Pattern)

Interaktionen:

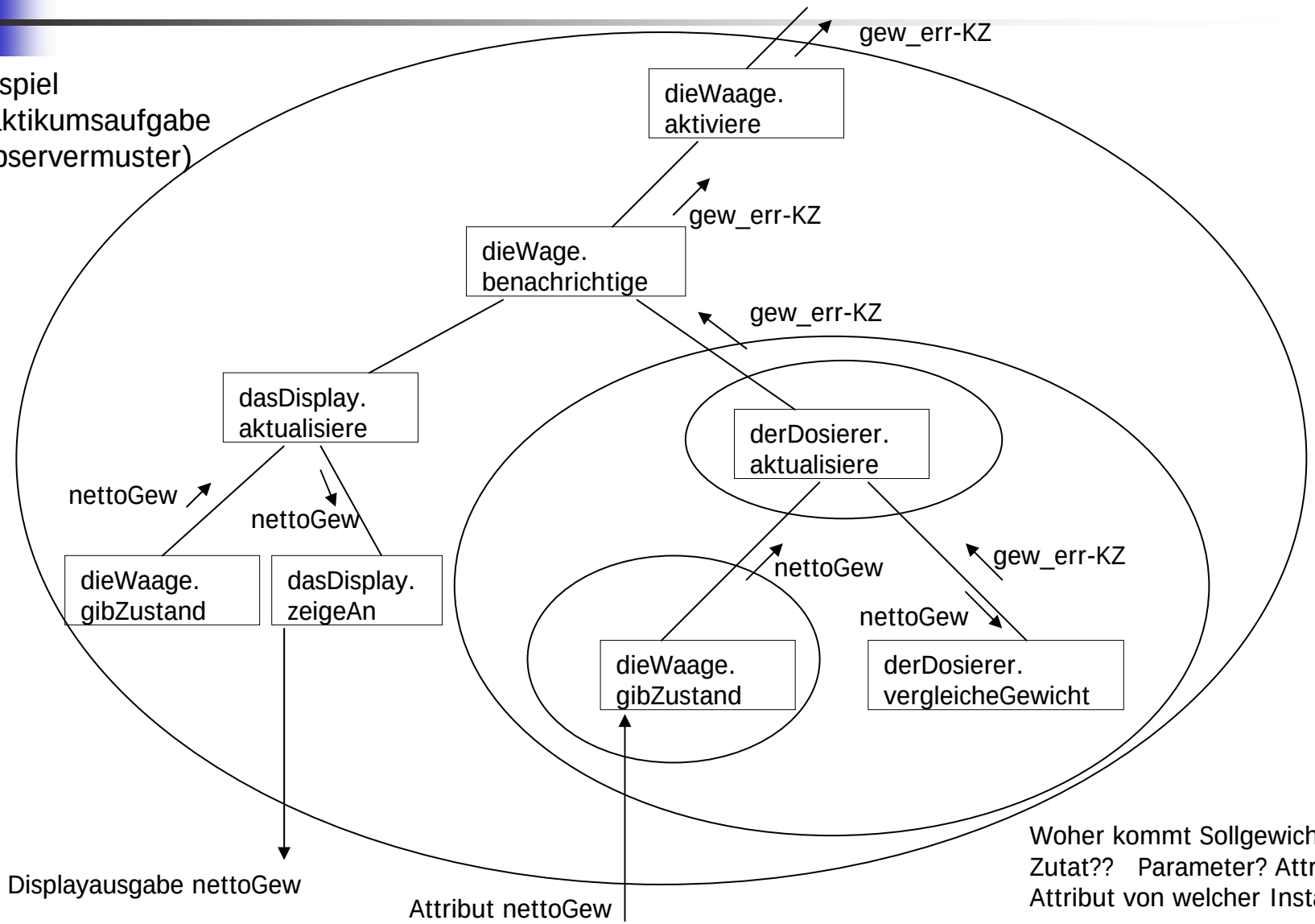


Frage: Welche Parameter werden übergeben /zurückgegeben?

Erstellen Sie das erweiterte Strukturdiagramm

Beispiel Modul-, Integrationstest

Beispiel
Praktikumsaufgabe
(Observermuster)



Woher kommt Sollgewicht von Zutat?? Parameter? Attribut? Attribut von welcher Instanz?

Beispiel Modul-, Integrationstest

Integrationstest:

Teilsysteme d, e,

werden zu größeren Teilsystem c zusammengesetzt.

(d oder e kann auch Fremdsystem sein.)

Beispiel in unserem Praktikum:

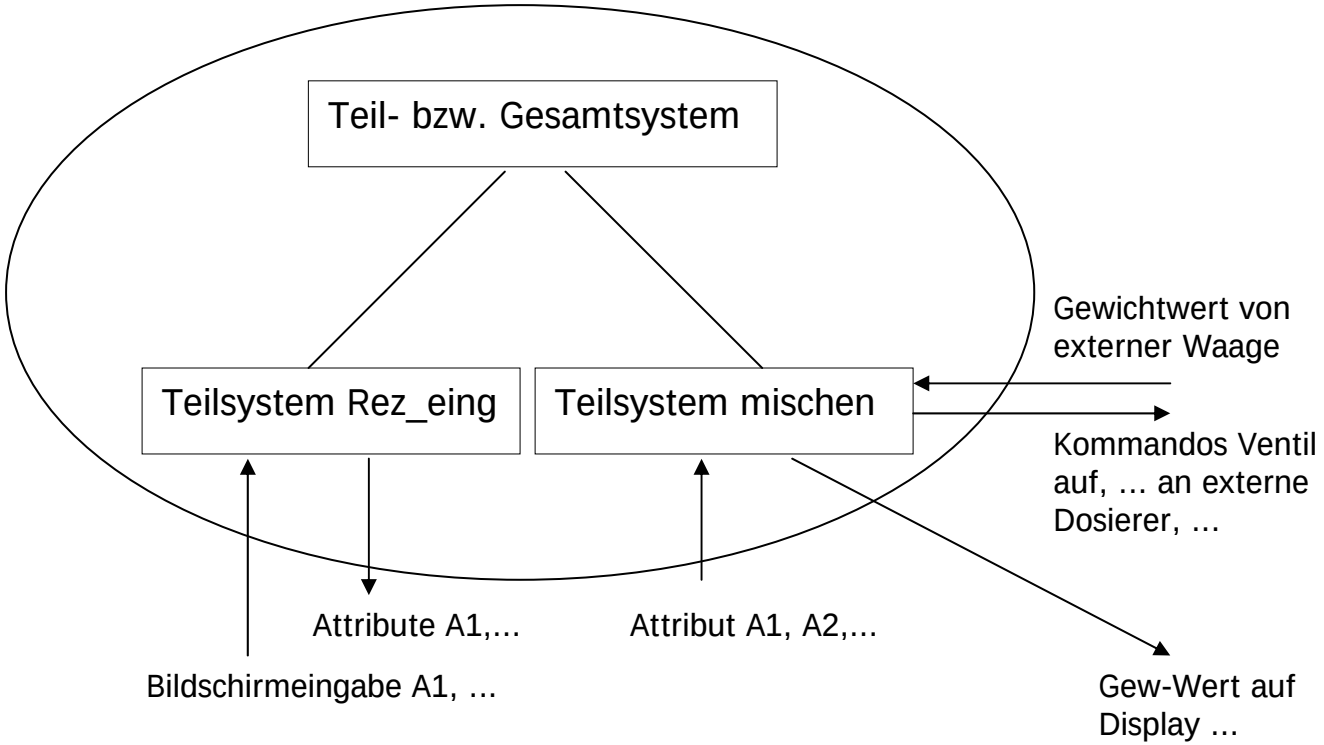
d und e sind Teilsysteme Rezept_mischen und Rezept_eingeben,

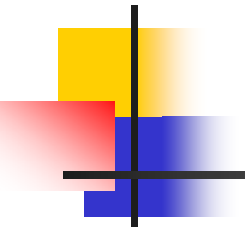
c ist Gesamtsystem

=> Hier schon Teil des Systemtest = Integrationstest der obersten Stufe

Beispiel Modul- Integrationstest

Integrationstest:





Wie schreiben wir die Tests im Praktikum?

- Wohin?
- Wie?

siehe auch Tafel

Wie schreiben wir die Tests im Praktikum?

Wollen Sie es so realisieren?
Oder: Einbindung der Test-Funktionen in Klasse (z. B. Waage)?

```
void main()
..
//Test-Driver:

void dieWaage_gibZustandTest (a)
{
   //setze Attribut dieWaage.nettoGew zu a
      ..
   //Aufruf von dieWaage.gibZustand()
      ..
   //Ausgabe Ergebnis
      Dann: manueller Vergleich mit Soll-Werten
      (Allgemein: Ergebnis kann sein:
       Rückgabe-Parameter / &-Parameter, Attribute, globale Variablen, Datei- / Datenbankfelder)
      ..
    //Zurücksetzen Attribute
}
```

Wie schreiben wir die Tests im Praktikum?

```
void aktualisiereTest(a)
{
    //setze Attribut ...
       ..
    //Aufruf von ...
       ..
    //Ausgabe Ergebnis
                ..
     //Zurücksetzen Attribute
}

void RezeptEingabeTest
{
       ..
     //Lesen aus Instanz
       ..
    //Ausgabe der Attribute von Rezept und Schritt
       ..
}

..
//instanziieren der Objekte (evtl. über Rezeptprozessor):
```

Wie schreiben wir die Tests im Praktikum?

```
// eigentlicher Test
//Test 1:
.....gibZustandTest (7);  // 7=Wert von Nettogewicht
//Test 2:
.....aktualisiereTest (7,20);  //20=Sollgewicht der Zutat
//Test 3:
.....aktualisiereTest (23,20);
..
//Test n:
...RezeptEingabeTest ();
//Test m:
.....mischeTest (Attribut1, Attribut2, ...)); //Attribute werden bei Auswahl Standardrezept schon in Programm
      gesetzt


..

//eigentliches main:
..
```

Wie schreiben wir die Tests im Praktikum?

- Finden Sie dieses Vorgehensweise zum Test großer Systeme günstig?
- => Testfunktionen auf verschiedene Klassen verteilen?

  ...
  - Wie?
  - Wohin?
  - Wie greifen wir auf protected-Attribute/-Funktionen zu?
  - Wie greifen wir auf privat-Attribute/-Funktionen zu?

- Verwendung eines Test-Tools, in dem nicht der gesamte Test in main() abgehandelt.

Testwerkzeuge

siehe auch SWT-Skript und

[JUCook]:http://junit.sourceforge.net/doc/cookstour/cookstour.htm]

verschiedene Tools für Testplanung und Testmanagement unterstützen:

- Durchführung dynamischer und statischer Tests
- Automatisierung von Testabläufen
- Automatisierung von GUI-Tests
  - Skriptsprachen um die GUI-Elemente aufzurufen und mit Werten zu versorgen
- Tools für Last- und Performancetests

Testwerkzeuge

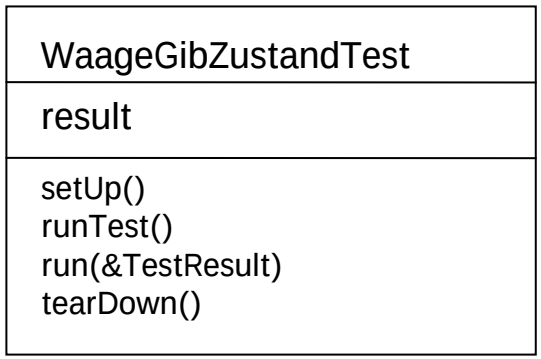
- Klassifizierung von Testwerkzeugen findet sich in [SW02, Kapitel 11].
- Auswahl von Testwerkzeugen::

- WinRunner/Mercury
- Rational Robot/IBM
- SilkTest/Segue
- QA-C++/QA-Systems
- Telelogic Tau/Telelogic

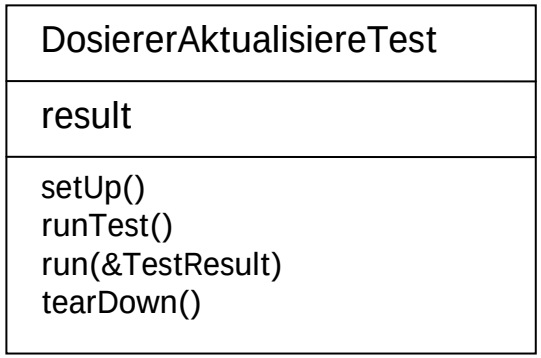
Testwerkzeug – xUnit / JUnit

- **xUnit** = Familie von Open Source Frameworks
- für verschiedene Programmiersprachen
- Java Variante: **JUnit**
- Tests in gleicher Progr-Sprache wie Anwendung
- Durchführung der Tests und die Überprüfung der Ergebnisse erfolgt automatisch.
- Jeder Testfall ist von allen anderen Testfällen unabhängig
- ⇨ bei einem auftretenden Fehler muss nicht der vollständige Testlauf abgebrochen werden.
- Die Testergebnisse werden aufgezeichnet.
- Testergebnisse werden mit Hilfe von Zusicherungen (Aussagen, die nach dem Lauf des Programms wahr sein müssen) überprüft

Testwerkzeug – xUnit / JUnit – prinzipielle Vorgehensweise

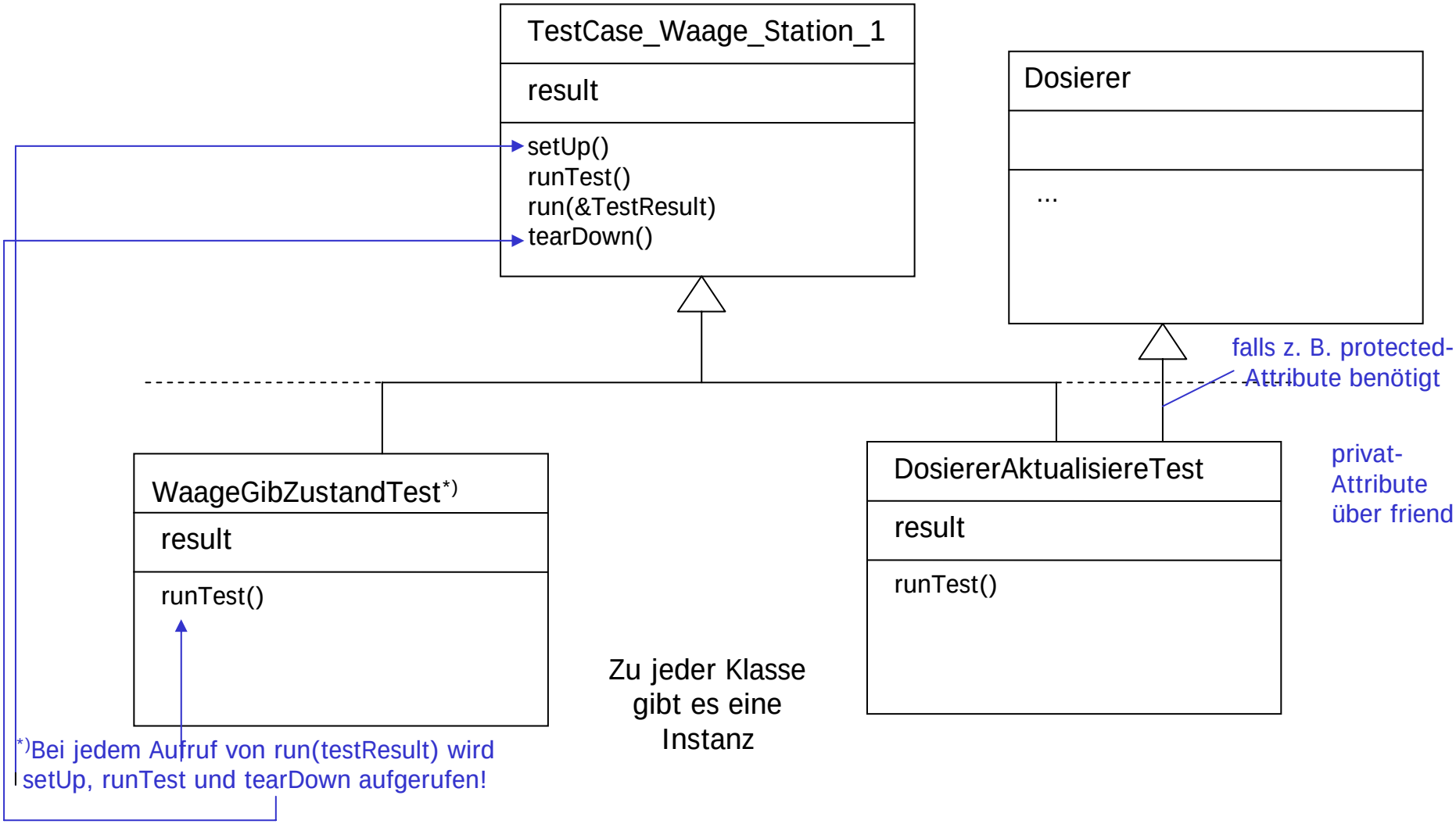



```
run(result)
{
  ... setUp(); //z. B. Zustand nettoGew auf 3 setzen
  ... runTest();
  ... tearDown(); //z. B. Zustand nettoGew wieder auf 0 setzen
  ...
}
 runTest()
{   ...
     .. assert (dieWaage.gibZustand() == 3);
     ...
}
```




```
...
runTest()
{ ...
   .. assert (derDosierer.aktualisiere (7,20) == false);
   .. assert (derDosierer.aktualisiere (23,20) == true);
}
```

Testwerkzeug – xUnit / JUnit – prinzipielle Vorgehensweise



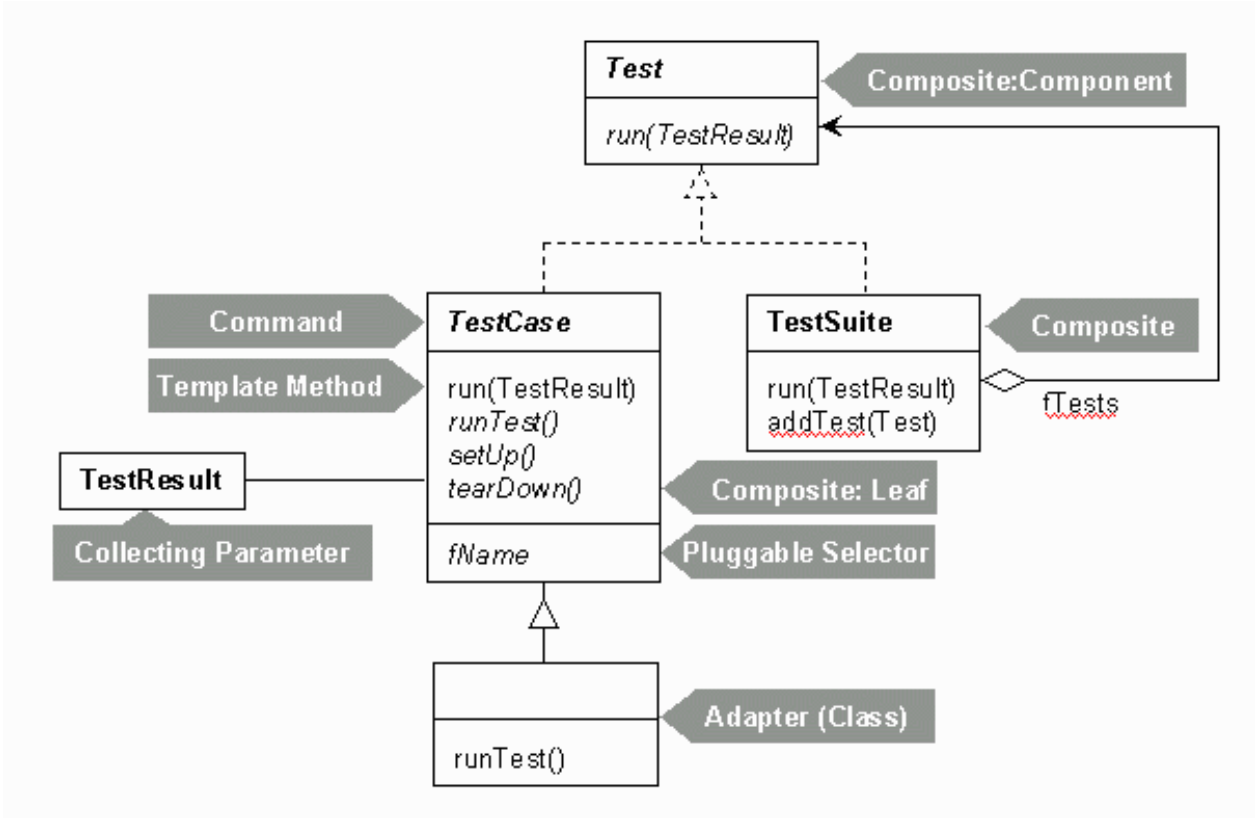
Testwerkzeug – xUnit / JUnit – prinzipielle Vorgehensweise

- Aufruf der einzelnen Testfälle von einer Test-Suite aus,
- die eine Sammlung von
  - Testfällen bzw.
  - Test-Suites enthält

Testwerkzeug – xUnit / JUnit

Struktur von xUnit, JUnit

- Klassendiagramm von xUnit (aus [JUcook])



Testwerkzeug – xUnit / JUnit

- mehrere Entwurfsmuster

  (Namen oder Klasssennamen von Entwurfsmuster: Grau hinterlegte Pfeile)
- Über Composite-Pattern ergibt sich Baumstruktur aus Testfällen und Testsuiten.
- Das abstrakte Interface Test wird vererbt an die Klassen TestCase und TestSuite.
- setUp() wird vor jedem Test aufgerufen. Sie führt alle Initialisierungen (z. B. der Instanzvariablen) durch.
- tearDown() sorgt nach jedem Testfall für einen definierten Endzustand.
- runTest() ruft den Testfall auf
- run (TestResult) übergibt das Ergebnis des Tests an die Klasse TestResult, in der die Ergebnisse gespeichert werden.
- Klasse TestSuite: Fügt über die Methode
- addTest(Test) Testfall in TestSuite ein (Aggregation).
  - ⇨ Eine TestSuite enthält mehrere Exemplare von TestCase und TestSuite
- zusätzliche Tools zur Erzeugung von Testdaten etc.

Kontrollfragen Test und Integration

- Wann ist ein Programm fehlerfrei?
- Was bedeutet es, wenn alle Tests fehlerfrei laufen
- Ist es normalerweise möglich alle möglichen Testfälle laufen zu lassen?
- Welche Prinzipien zur Auswahl von Testfällen kennen Sie?
- Wie erreichen wir, dass die Zuverlässigkeit nach dem Test möglichst hoch ist?
- Wie unterscheiden sich Black-Box-Tests und White-Box-Tests
- Welche Unterschiede es gibt zwischen Anweisungs-, Zweig-, Pfad- und Bedingungsüberdeckung
- Was halten Sie von der Aussage "Sie haben das Programm doch getestet – wie kann dann so ein Fehler auftreten?"
- Warum muss man in der Architektur, im Design,... schon an Test denken?
- Warum gibt es im V-Modell 4 Testphasen? Wie unterscheiden sie sich?

Können Sie jetzt Tests spezifizieren und anwenden?