Teil VII

Testen

7.1 Einführung

- Verifikation $\leftrightarrow$ Validation
  - Validation: entspricht System Kundenwünschen?
    (richtiges System?)
  - Verifikation: entspricht System der Spezifikation? (System richtig?)
- Testen $\leftrightarrow$ Debugging
  - Testen: Fehler feststellen
  - Debugging: Fehler finden und beseitigen
- Testen größtenteils unabhängig von Programmiersprache
- Test-Kosten z.B. 30-50% der Gesamtkosten
- Testen kann Fehler aufzeigen, aber keine Fehlerfreiheit

Bemerkungen zum Testen

- neben (dynamischem) Testen: statische Verifikation
  (z.B. Code-Inspektion) aufwändig, aber effektiv
- Testen integrieren in Implementierungsphase (und vorherige Phasen)
- Testen braucht Zeit
- psychologisch besser: Tester $\neq$ Entwickler
- beim Korrigieren von Fehlern entstehen oft neue ($\rightarrow$ Regressionstest)
- guter Entwurf (Zerlegung in überschaubare, unabhängige Einheiten,
  Schichtenarchitektur,...) erleichtert Testen
- viele Fehler in selten ausgeführtem Code
  (z.B. Sonderfall-Behandlung; denn: Normalfall meist gut verstanden)

Klassifikation nach der Größe des Testsystems

1) Testen einer (Basis-)Einheit

   klassisch: Prozedur, Modul; OO: Klasse

   - der Einfachheit halber durch Entwickler
   - keine aufwändigen Testberichte

2) Integrationstest

   - testet Zusammenspiel mehrerer Einheiten
   - z.T. gleiche Testdaten wie bei Testen einer Einheit

3) Systemtest

   Test des Gesamtsystems (inkl. Umgebung)

Klassifikation von Tests nach Zielrichtung

Regressionstest

- nach Fehlerkorrektur bzw. Änderung überprüfen, ob (bisherige) Funktionalität erhalten geblieben

Operationstest

- Systemtest im normalen Ablauf

Full-Scale-Test

- testet System an Grenzen der Anforderungen

Performance-Test bzw. Kapazitätstest

- überprüft Laufzeit bzw. Speicherplatzverbrauch ggfs. für einzelne Use-cases
- erfordert Instrumentierung

Klassifikation von Tests nach Zielrichtung (2)

Überlast-Test

- testet System jenseits der Anforderungen
- wichtig: “sanfte” Reaktion bei Überforderung
  z.B. geordnetes Herunterfahren statt Absturz

Vergleichstest

- bei extremen Anforderungen an Zuverlässigkeit (z.B. Kraftwerk)
- ≥ 2 unabhängige Implementierungen der gleichen Spezifikation
- Ergebnisse werden zur Laufzeit (oder beim Testen) verglichen
- bei Abweichung: Fehlermeldung und/oder Mehrheitsentscheid
- Vor.: Spezifikation richtig!

Klassifikation von Tests nach Zielrichtung (3)

Anforderungsbasierter Test

- überprüft einzelne Anforderung

Ergonomietest

- testet Mensch-Maschine-Schnittstelle
  (z.B. Menüs logisch und verständlich?)

Test der Dokumentation

- Vergleich von Systemverhalten und dessen Beschreibung im Handbuch
- Überprüfung der Dokumentation auf Lesbarkeit und Verständlichkeit

Akzeptierungstest

- durch Auftraggeber (α-Test)
- bei “Standard-Software” (z.B. Compiler): β-Test durch Pilot-Kunden

Test-Strategien

bottom-up:

- Reihenfolge: von kleineren Bausteinen zu größeren (inkrementell)
- z.B. bei OO:

  Reihenfolge: Klasse, Paket, Use-case, Teilsystem, System
- Vorteil: keine (bzw. wenige) Teststümpfe
- Nachteil:
  - Behebung von Fehlern in oberen Komponenten kann
    - umfangreiche Änderungen in unteren Komponenten erzwingen
    - dadurch vorherige Tests wertlos machen
  - bis zum Schluss kein lauffähiger Prototyp

  → Entwurfsfehler spät gefunden

Test-Strategien (Fortsetzung)

top-down:

- zu obersten Komponenten schrittweise weitere hinzunehmen
- erfordert viele Test-Stümpfe (aufwändig)
- früh vorzeigbarer Prototyp ($\rightarrow$ Kunden, Manager)
- Entwurfsfehler früh gefunden
- Varianten: depth-first oder breadth-first

Test-Strategien (Fortsetzung 2)

Kompromiss:

- Kombination von bottom-up- (für untere Komponten) und top-down-Vorgehen (für obere Komponenten)
- denkbar: anfangs top-down; Übergang zu bottom-up (für betrachtetes Teilsystem), wenn Stumpf zu aufwändig

Äquivalenz-Teilung

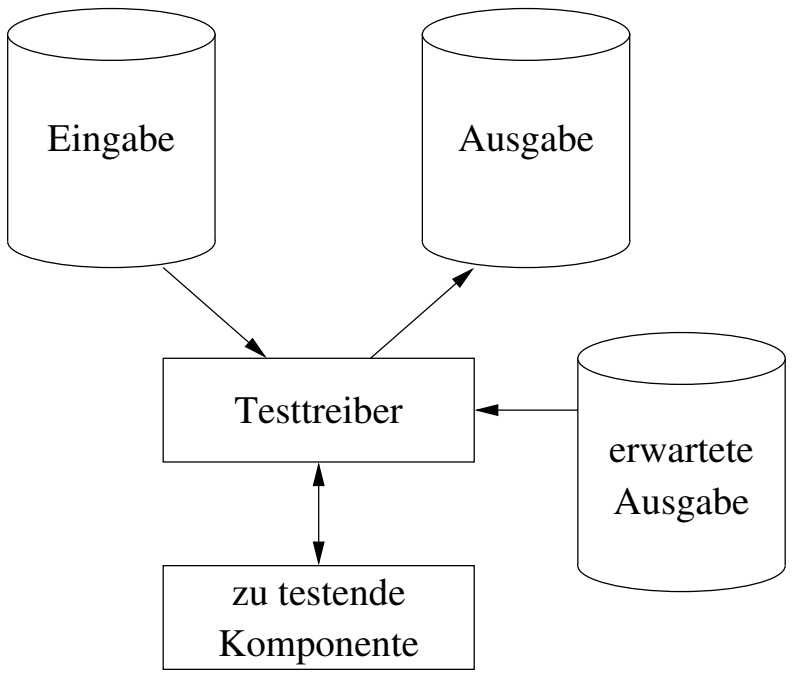
- Ansatz zur Erzeugung von Testfällen
- Testfall: Testeingabe (Testdaten) mit zugehöriger, erwarteter Ausgabe
- Idee: jeder Test soll mögliche zusätzliche Fehler aufdecken
- ein guter Test findet mit hoher Wahrscheinlichkeit einen (zusätzlichen) Fehler
- keine Redundanz bei Testfällen
- Ziel: finde Fehler mit minimalem Aufwand
- Äquivalenz-Teilung reduziert mögliche Testfälle auf einen Fall pro mögliches Verhalten (inkl. Fehlersituationen)

Beispiele: Äquivalenz-Teilung

- bei Stack: leer, einelementig, mehr als ein Element
- bei 1..50: teste z.B. 0, 1, 2, 27, 49, 50, 51
- bei Listen: leere Liste, einelementige Liste, längere Liste
- bei Überweisung:
  - Normalfall,
  - BLZ vergessen, falsche Zeichen in BLZ, unbekannte BLZ,
  - Kontonr. vergessen, falsche Zeichen in Kontonr., unbekannte Kontonr.,
  - Name vergessen,
  - Betrag über Limit,
  - Saldo unzureichend,
  - Zielkonto == Quellkonto,

Automatisches Testen

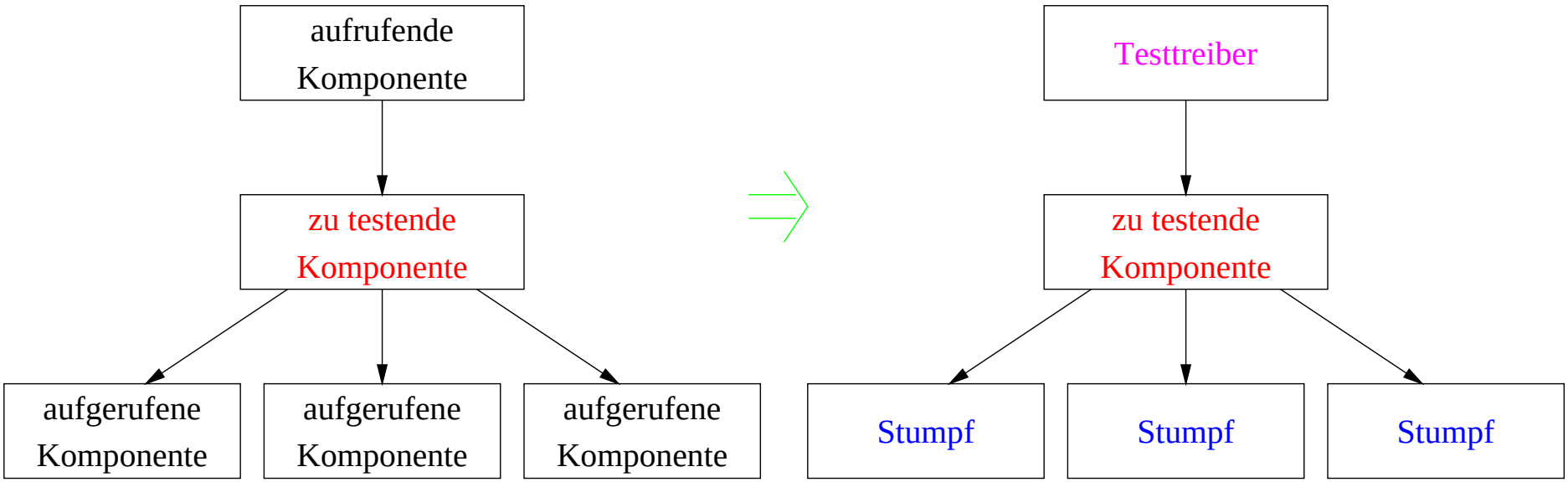
- Senkung des Testaufwands durch Automatisierung
- vor allem bei Regressionstest
- daher: alle Testfälle speichern (inkl. erwarteter Ergebnisse)
- bei automatischen (Regressions-)Tests werden erhaltene Ergebnisse mit erwarteten (bisherigen) Ergebnissen verglichen



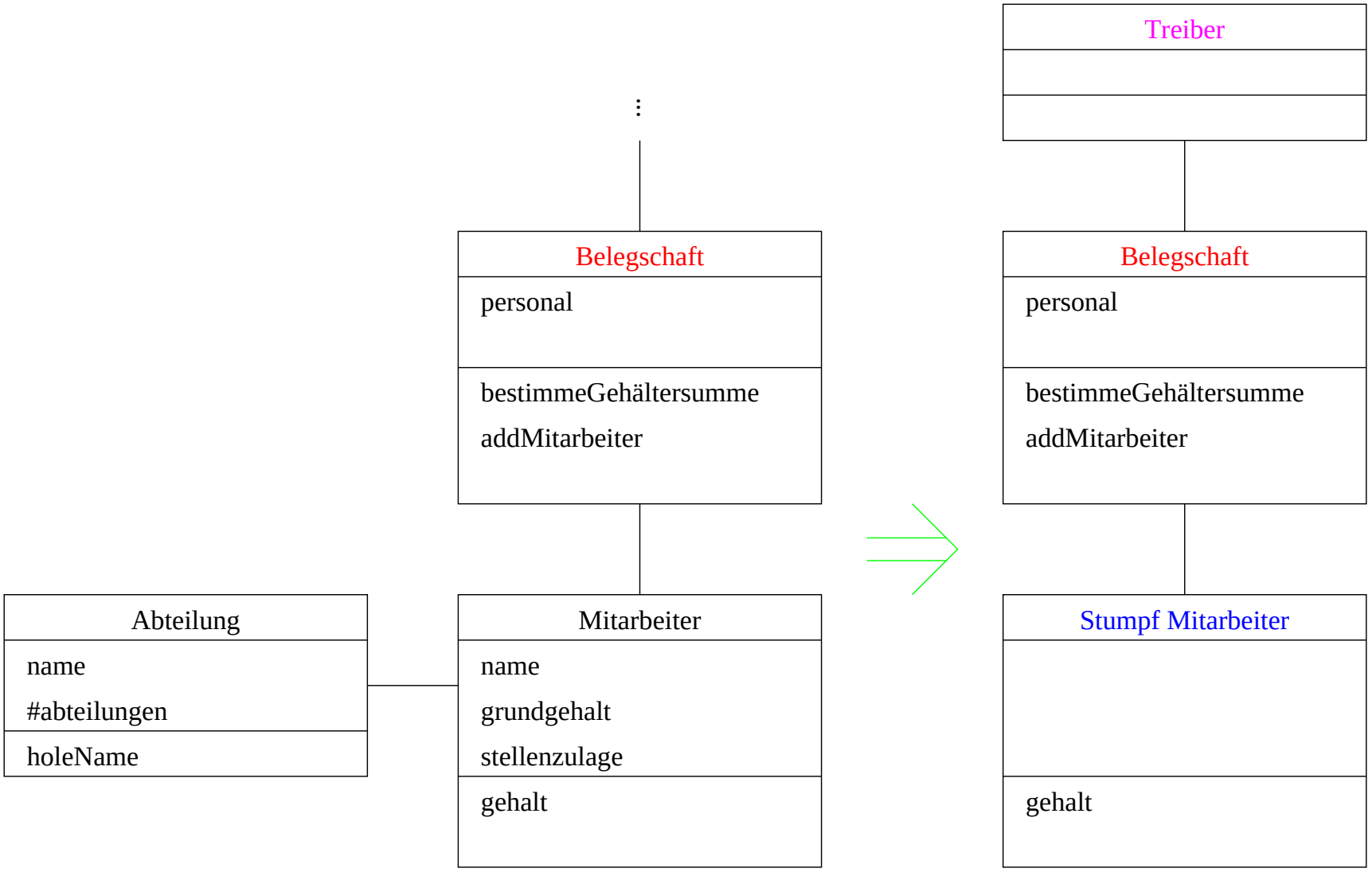
7.2 Testen einer Komponente

- z.B. Modul, Klasse
- Testtreiber rufen Komponente auf
- Stümpfe simulieren verwendete Komponenten, z.B. durch
  - Nachschlagen der Ergebnisse in Tabelle
  - durch interaktives Erfragen des Ergebnisses von Tester
    Nachteil: kein automatisches Testen
  - durch Generierung zufälliger Ergebnisse
    - Nachteil: führt oft an anderer Stelle zu Problemen
    - z.B. bei zufällig generierter statt sortierter Liste scheitern Komponenten, die von Sortierung ausgehen
  - Entwicklung eines dedizierten Simulationsstumpfs: flexibel, aber aufwändig

Beispiel: Test einer Komponente



Beispiel: Test-Treiber und -Stümpfe



Beispiel-Stumpf 1: zufälliges Ergebnis

```
public class Mitarbeiter {
    static int aufrufnr = 1;

    public Mitarbeiter(String Name, int Grundgehalt,
                       int Stellenzulage,
                       String Abteilungsname) {;}

    public int gehalt(String Abteilungsname){
      return (aufrufnr++)*7390 % 5000;}}
```

Beispiel-Stumpf 2: Ergebnis aus Tabelle

```
public class Mitarbeiter {
    static int vAufrufnr = 0;
    static int pAufrufnr = 0;
    static int vErg[] = {3720,4580,2820};
    static int pErg[] = {7480,3253,6480};

    public Mitarbeiter(String Name, int Grundgehalt,
            int Stellenzulage, String Abteilungsname) {;}

    public int gehalt(String Abtname) throws Exception{
      if (Abtname.equals("Vertrieb"))
         return( vErg[vAufrufnr++ % vErg.length]);
      if (Abtname.equals("Produktion"))
         return( pErg[pAufrufnr++ % pErg.length]);
      throw new Exception("illegaler Abtsname");}}
```

Bemerkungen zu Testtreibern und -stümpfen

- Testtreiber und -stümpfe sind Teil des Systems und können bei Wartung wiederverwendet werden
- Test-Treiber und -Stümpfe möglichst allgemein und modular, so dass leicht an andere Tests anpassbar
- wichtig: Ausgabe überprüfen (nicht nur entgegennehmen)

Erstellung von Testtreibern mit JUnit

- JUnit ist ein Framework zur Erstellung und automatischen Ausführung von Tests (s. http://www.junit.org/)
- Hierarchie von Testsuites möglich (Methode `suite()`)
- eine Testklasse ist Unterklasse von `junit.framework.TestCase` mit Methoden:
  - `setUp()`: zur Initilisierung vor eigentlichem Test
  - `tearDown()`: Finalisierung nach Test
  - `testXYZ()`: Testfall (i.allg. mehrere `XYZ`, $\rightarrow$ Reflection)
- in `testXYZ()` häufig verwendet:
  - `assertEquals(x,y)`: Test schlägt fehl, wenn `x` und `y` ungleich
  - `assertTrue(b)`: Test schlägt fehl, wenn `b` falsch
  - `fail("Text")`: Test explizit fehlgeschlagen

Beispiel: JUnit

```
public class Natural{
  protected int n;

  public Natural(int z) throws NotNaturalException{
    if (z<0)throw new NotNaturalException();
    n=z;}

  public int getVal(){return n;}

  public int div(Natural m) throws ArithmeticException{
    if (m.getVal() == 0)
      throw new ArithmeticException();
    return n / m.getVal();}
}
```

Beispiel: JUnit (2)

```
import junit.framework.*;
import junit.extensions.*;

public class DivTest extends TestCase{
  public DivTest(String name){super(name);}

  public static Test suite(){
    return new TestSuite(DivTest.class);}

  public static void main(String[] args){
    junit.textui.TestRunner.run(suite());}

  protected void setUp(){}
  protected void tearDown(){}
  ...
```

Beispiel: JUnit (3)

```
  ...
  public void testDiv(){
    try{Natural n = new Natural(7);
        Natural m = new Natural(3);
        assertEquals(2,n.div(m));}
    catch(Exception e){fail("fehlerhafte Exception");}}

  public void testDiv0(){
    try{Natural n = new Natural(4);
        Natural m = new Natural(0);  n.div(m);}
    catch(ArithmeticException e){return;}
    catch(Exception e){fail("fehlerhafte Exception");}
    fail("Arithmetic Exception erwartet");
}
```

Testen einer Basis-Einheit

- unterstes Test-Level
- klassisch: Modul, Prozedur; OO: Klasse, Methode
- (der Einfachheit halber) durch Entwickler
- (sich ergänzende) Vorgehensweisen:
  1) Glass-Box-Testing
  2) Black-Box-Testing

Glass-Box-Testing

- verwendet Wissen über Aufbau einer Einheit
  (zur Äquivalenz-Teilung)
- dadurch auch geeignet zum Testen von Sonderfall-Behandlung
- zuletzt (da durch Korrektur ggfs. Umbau)

Black-Box-Testing

- ohne Wissen über Aufbau einer Einheit
- Äquivalenz-Teilung auf Basis der Spezifikation
- Testfälle können bereits nach Systementwurf generiert werden
- auch geeignet zum Testen von größeren Systemteilen (Use-case, Teilsystem)
- weniger gut zum Testen von Sonderfällen

Zustandsbasiertes Testen

- Variante von Black-Box-Testen für OO-Klassen
- beobachtet Veränderung der Attributwerte durch Operationen
  $\rightarrow$ Lebenszyklus, Harel-Automaten
- jeden Zustand besuchen, jeder Transition folgen

Zustands-Stimulus-Matrix

Stimulus \ Zustand	s0	s1	s2
i1	ok	ok	falsch
i2	ok	langsam	Absturz

- Äquivalenz-Teilung gemäß Zustand und "repräsentativer" Eingabe
- reine Lese-Operationen ignorieren
- zur Veranschaulichung des Zustandsraum geeignete Operation bereitstellen

Glass-Box-Testing

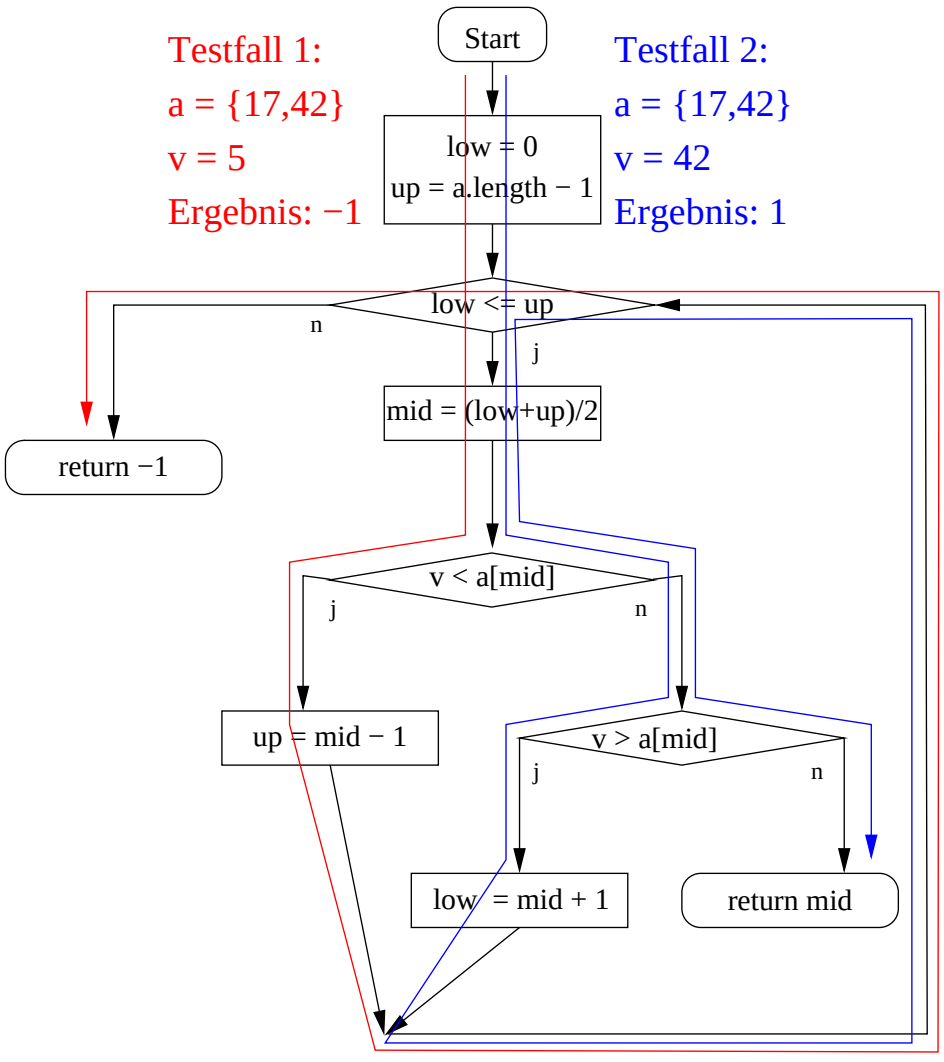
- synonym: White-Box-Testing, Strukturelles Testen
- systematisches Generieren von Testfällen anhand des Codes
- Hilfsmittel: Flussgraph
- nicht-erreichbares Maximal-Ziel: alle Pfade durchlaufen
  (i.allg. ∞-viele!)
- realistisches Ziel:
  - jeden Zweig $\geq$ 1-Mal durchlaufen
  - keine Fehlerfreiheit garantiert!

Bemerkungen zum Glass-Box-Testing

- möglich: Pfad kann nicht alleine getestet werden, sondern nur als Teilpfad eines anderen
- Testfall-Generierung kann durch Tool unterstützt werden
- Variante: alle Paare von Zweigen durchlaufen
- weitere Varianten: Schleifentesten, Datenflusstesten

Beispiel: Glass-Box-Testing

```
class SortedIntArray{
  private int a[];
  ...
  int binSearch(int v){
    int low = 0;
    int up = a.length-1;
    while (low <= up){
      int mid = (low+up)/2;
      if (v < a[mid]) up = mid - 1;
      else if (v > a[mid])
             low = mid + 1;
      else return mid;}
    return -1;
  }
}
```



#Pfade: ∞; durch Äquivalenz-Teilung: 2 Testfälle

Schleifentesten

- (einzelne) Schleife (von 0 bis n):
  - 0, 1, 2 Iterationen
  - m Iterationen, wobei $2 < m < n$
  - $n - 1$, n Iterationen
- geschachtelte Schleife:
  - bei tiefer Schachtelung ggfs. zu hohe Anzahl von Testfällen, daher:
  - Schleifen von innen nach außen betrachten
  - beim Testen einer Schleife werden Zähler der äußeren Schleifen auf Minimalwert und Zähler der inneren Schleifen auf "typischen Wert" m gesetzt
- Folge von Schleifen
  - wenn Schleifen voneinander unabhängig (bzgl. Zähler): Vorgehen wie bei einzelner Schleife
  - sonst (gleicher Zähler): Vorgehen wie bei geschachtelten Schleifen

Datenflusstesten

- $def(S) := \{X \mid S \text{ greift schreibend auf } X \text{ zu}\}$

  z.B. $S := [X = Y + 1;]$, $S := read(X);$
- $use(S) := \{X \mid S \text{ greift lesend auf } X \text{ zu}\}$

  z.B. $S := [X = X + 1;]$, $S := if\ (X > 0) \ldots$
- falls $X \in def(S) \cap use(S')$ und $X \notin def(S'')$

  für jede Anweisung S'',

  die auf einem Pfad von S zu S' durchlaufen wird,

  so bildet $[X, S, S']$ eine def-use-Kette (du-Kette)
- beim Datenflusstest wird jede du-Kette mindestens einmal durchlaufen
- hierbei wird nicht unbedingt jeder Programmzweig durchlaufen

  Beispiel: `if (X == 0) print(1) else print(2)`

Beispiel: def-use-Ketten

```
int binSearch(int v){
    int low = 0;  int up = a.length-1;
    while (low <= up){
       int mid = (low + up)/2;
       if (v < a[mid]) up = mid - 1;
       else  if (v > a[mid])
                  low = mid + 1;
             else return mid;}
    return -1;}
```

Beispiel: du-Ketten in binSearch

```
[low, low=0, while(low<=up)]
[low, low=0, mid=(low+up)/2]
[low, low=mid+1, while(low<=up)]
[low, low=mid+1, mid=(low+up)/2]
[up, up=a.length, while(low<=up)]
[up, up=a.length, mid=(low+up)/2]
[up, up=mid-1, while(low<=up)]
[up, up=mid-1, mid=(low+up)/2]
[mid, mid=(low+up)/2, if(v<a[mid])]
[mid, mid=(low+up)/2, up=mid-1]
[mid, mid=(low+up)/2, if(v>a[mid])]
[mid, mid=(low+up)/2, low=mid+1]
[mid, mid=(low+up)/2, return mid]
```

- die beiden Pfaden durchlaufen alle du-Ketten außer Nr. 8

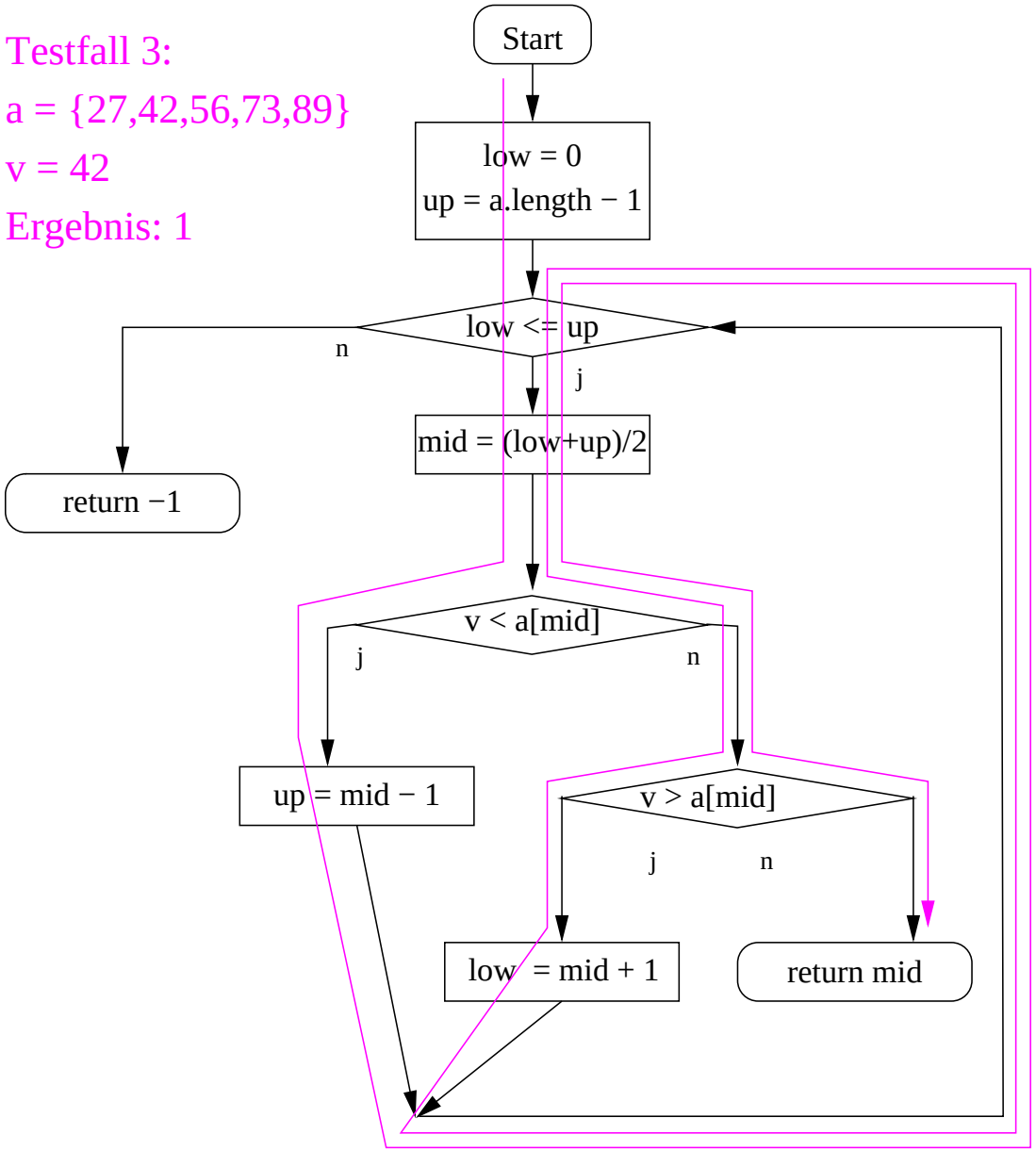
Beispiel: du-Ketten-Überdeckung für binSearch

Testfall 3:

a = {27,42,56,73,89}

v = 42

Ergebnis: 1



Zweigüberdeckung $\leftrightarrow$ du-Kettenüberdeckung

- Durchlaufen aller du-Ketten erfordert meist mehr Pfade als Durchlaufen aller Zweige

Beispiel:

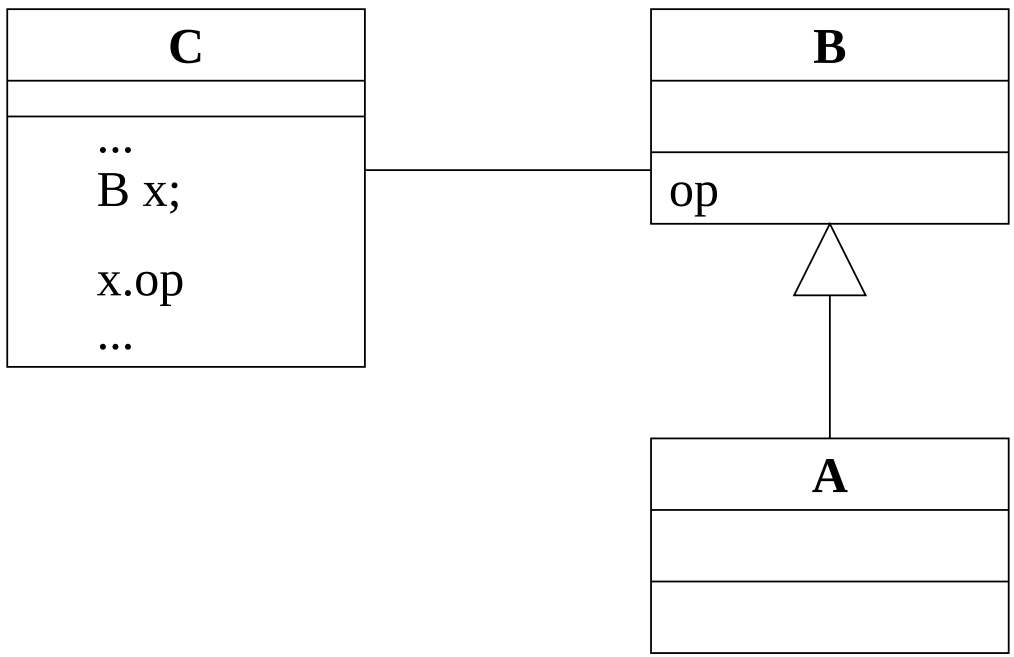
```
if (Y > 0) X = 1;
else X = 2;
if (Y < 5) Y := X;
else Y = -X;
```

- Durchlaufen der 4 du-Ketten erfordert hier 4 Pfade
- zur Zweigüberdeckung reichen 2 Pfade

7.3 Testen objektorientierter Software

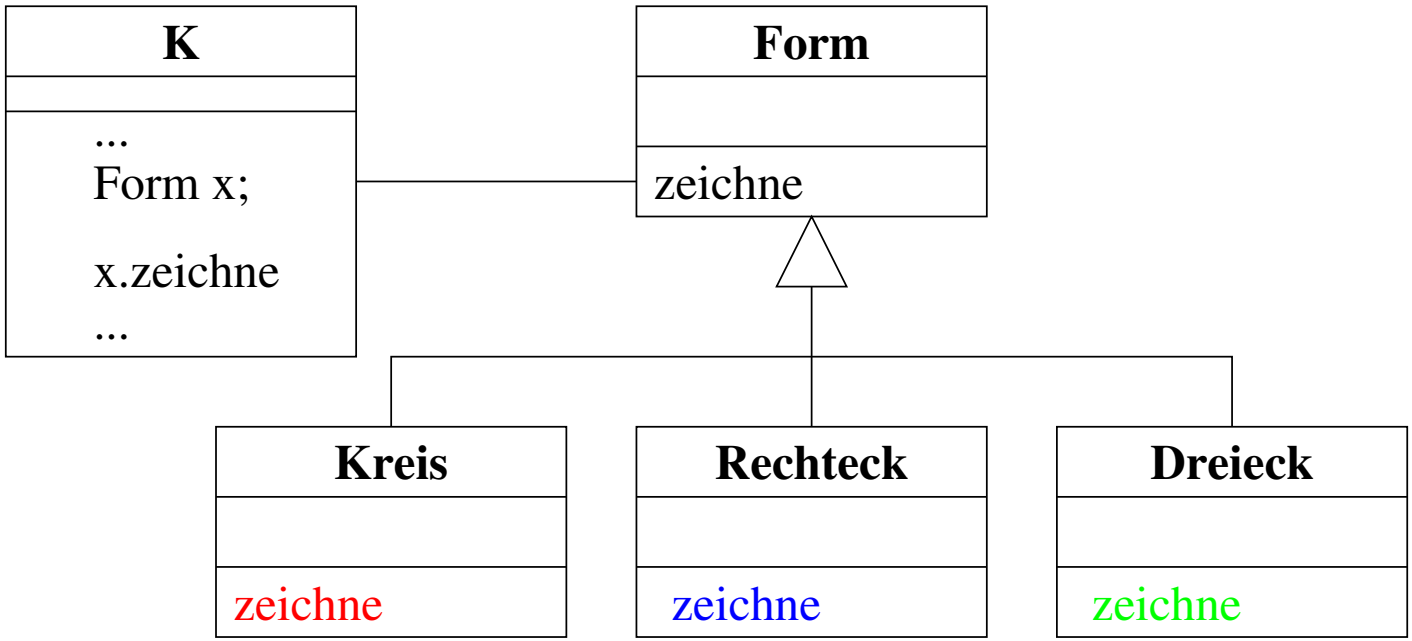
Testen und Vererbung (1)

- beim Hinzufügen einer Unterklasse *A* von *B* braucht eine Klasse *C*, die *B* benutzt, nicht erneut überprüft zu werden,
  sofern *A* keine Operation von *B* überschreibt!



Testen und Vererbung (2)

- Überschreiben von Operationen erhöht Testaufwand



- im Bsp.: *K* muss bei Hinzufügen von Dreieck (teilweise) überprüft werden
- Problem: im Code von *K* ist dies nicht erkennbar (kein Verweis auf Dreieck)

Fazit

- bei Überschreiben ist OO-Testen schwieriger als in Nicht-OO-Sprachen
- gleiche Testpfade, jedoch aufgerufene Methode aus Code nicht ersichtlich
- im Beispiel: Testfälle für alle drei Formen
- Integrationstest besonders wichtig bei OO

Testen und Vererbung (3)

- bei Test der Unterklasse müssen Oberklassen-Operationen und -Attribute mitberücksichtigt werden, denn z.B.:
  - Unterklasse kann Werte der Oberklassenattribute erzeugen, die in Oberklasse bisher nicht auftraten
    (wird vermieden, wenn Attribute private)
  - Oberklassenoperationen bewirken ggfs. eine fehlerhafte Verknüpfung von Operationen der Unterklasse
- bei Änderung der Oberklasse müssen (direkte und indirekte) Unterklassen erneut getestet werden
- daher: Vererbung reduziert Code, nicht aber Testaufwand

Beipiel: Testen und Vererbung

```
class O{
  public int f(){return 1;}
  public int g(){return 1/f();}
}
class U extends O{
  public int f(){return 0;}
}
...
O o;  U u;
o.g();        // liefert 1
u.g();        // Division durch 0
...
```

Integrationstest

- schrittweise Erweiterung von einzelnen Einheiten zu Gesamtsystem
- Test des Zusammenspiels (Schnittstellen) bereits überprüfter Einheiten
- oft durchgeführt von speziellem Test-Team
- klassisch: Module einzeln hinzunehmen (kein Big-Bang!)
- bei OO sinnvoll: Test an Use-case orientieren;
  denn Use-cases beziehen sich auf das Zusammenwirken von Einheiten
- alle Use-cases der Reihe nach einzeln testen:
  - aus interner Sicht anhand von Interaktionsdiagrammen
  - aus externer Sicht aufbauend auf Anforderungen

Integrationstest (Fortsetzung)

- Reihenfolge bei aufeinander aufbauenden Use-cases geeignet wählen
- zunächst Basis-Use-case, dann Erweiterungen
  (z.B. Fehlerbehandlung)
- Alternativen zum Use-Case-orientierten Vorgehen:
  - “bottom-up”: zuerst Klassen, die keine anderen aufrufen/verwenden;
    dann schrittweise Klassen hinzunehmen,
    die bisherige Klassen aufrufen/verwenden
    (Achtung: keine Hierarchie)
  - “top-down” weniger empfehlenswert

Systemtest

- Test des Systems in Zielumgebung
- testet (verzahntes) Zusammenwirken verschiedener Use-cases
- umfasst: Operationstest, Full-Scale-Test, Überlasttest, Anforderungs-basierte Tests, Test der Benutzerdokumentation

Testablauf

- Plan zur zeitgerechten Bereitstellung der nötigen Ressourcen (Personal, Geräte)
- Testlogbuch (für Versionen des Systems)
  mit Historie der erfolgreichen und fehlgeschlagenen Test
  (mit Angabe der Ursache und Korrektur)

→ zur Planung späterer Testwiederholungen (nach Änderung)

7.4 Weitere Aspekte

Testen von Realzeitsystemen

- Realzeitsysteme steuern oft wichtige Anlagen
  (z.B. Kraftwerke, Raketen)

→ hohe Anforderungen an Zuverlässigkeit

- Probleme:
  - Fehler oft zeitabhängig
  - Systeme oft intern parallel und nicht-deterministisch

  → Fehler schwer reproduzierbar
  - Ansatz: Instant Replay
    Aufzeichnung aller Entscheidungen und spätere Wiederholung
- Alternative zum Testen: formale Verifikation

Programmverifikation

- Grundlage: Untersuchung des Quelltextes
- umfasst weder Validation noch Überprüfung nicht-funktionaler Anforderungen
  (wie Eignung der Benutzerschnittstelle, Laufzeit)
- Vorteil: keine gegenseitige Beeinflussung von Fehlern
- effektiv; $\approx$ 60 % der Fehler typischerweise durch statische Analyse auffindbar
- oft billiger als (alleiniges) Testen
- Ansätze:
  1) Programm-Inspektion: manuelle Prüfung auf (typische) Fehler
  2) Automatische Programm-Analyse (z.B. lint für C)
  3) Formale Programmverifikation ($\rightarrow$ Formale Spezifikation)
     kann Fehlerfreiheit beweisen (heute in Nischen)

Werkzeuge beim Testen

- Debugger
- Testdatengenerator
- Kontrollfluss-Analysator
- Dateivergleicher
- Simulator für Teilsystem (inkl. Hardware)