Softwaretechnik II
Vorlesung
Sommersemester 2004,. . . , 2011

Johannes Waldmann, HTWK Leipzig

23. Mai 2011

Programme und Softwaresysteme

(Charakterisierung nach Brooks)

- *Programm:* in sich vollständig, kann von seinem Autor auf dem System benutzt werden, auf dem es entwickelt wurde
- *Komponente eines Systems:* Schnittstellen, Integration
- *marktfähiges Produkt:* Generalisierung, Tests, Dokumentation, Wartung

Software ist schwer zu entwickeln

- ist immaterielles Produkt
- unterliegt keinem Verschleiß
- nicht durch physikalische Gesetze begrenzt
- leichter und schneller änderbar als ein technisches Produkt
- hat keine Ersatzteile
- altert
- ist schwer zu vermessen

(Balzert, Band 1, S. 26 ff)

Produktivität

Ein Programmierer schafft etwa

10 (in Worten: zehn)

(vollständig getestete und dokumentierte)
Programmzeilen pro Arbeitstag.
(d. h. $\leq$ 2000 Zeilen pro Jahr)

Dieses Maß hat sich seit 30 Jahren nicht geändert.
($\Rightarrow$ Produktivitätssteigerung nur durch höhere Programmiersprachen möglich)

Inhalt

- Entwurfsmuster (= Funktionen höherer Ordnung)
- Softwarequalität
  - semantisch: Spezifizieren, Verifizieren, Testen
  - syntaktisch: „Metriken“, Code Smells
- Programmieren im Team, Werkzeuge (git)
- Refactoring, Werkzeuge (eclipse)

Material

- Balzert: Lehrbuch der Software-Technik, Band 2, Spektrum Akad. Verlag, 2000
- Andrew Hunt und David Thomas: The Pragmatic Programmer, Addison-Wesley, 2000
- Gamma, Helm, Johnson, Vlissides: Entwurfsmuster, Addison-Wesley, 1996
- Martin Fowler: Refactoring, ...
- Edsger W. Dijkstra: `http://www.cs.utexas.edu/users/EWD/`
- Joel Spolsky: `http://www.joelonsoftware.com/`
- Scott Adams: `http://dilbert.com/`
- Randall Munroe: `http://xkcd.com/`

Organisation

- Vorlesung:
  - montags, 11:15–12:45, (u: Li 415, g: Li 318)
- Übungen (Z423):
  - dienstags, 11:15–12:45 (MIB)
  - *oder* donnerstags 9:30–11:00 (INB)
  - *oder* donnerstags 11:15–12:45 (INB)
- Übungsgruppen wählen: `https://autotool.imn.htwk-leipzig.de/cgi-bin/Super.cgi`

Leistungen:

- Prüfungsvoraussetzung: regelmäßiges und erfolgreiches Bearbeiten von Übungsaufgaben
  ggf. in Gruppen (wie im Softwarepraktikum)
- Prüfung: Klausur

The Pragmatic Programmer

(Andrew Hunt and David Thomas)

1. Care about your craft.
2. Think about your work.
3. Verantwortung übernehmen. Keine faulen Ausreden (The cat ate my source code . . .) sondern Alternativen anbieten.
4. Reparaturen nicht aufschieben. (Software-Entropie.)
5. Veränderungen anregen. An das Ziel denken.
6. Qualitätsziele festlegen und prüfen.

Lernen! Lernen! Lernen!

(Pragmatic Programmer)
Das eigene *Wissens-Portfolio* pflegen:

- regelmäßig investieren
- diversifizieren
- Risiken beachten
- billig einkaufen, teuer verkaufen
- Portfolio regelmäßig überprüfen

Regelmäßig investieren

(Pragmatic Programmer)

- jedes Jahr wenigstens eine neue Sprache lernen
- jedes Quartal ein Fachbuch lesen
- auch Nicht-Fachbücher lesen
- Weiterbildungskurse belegen
- lokale Nutzergruppen besuchen (Bsp: `http://gaos.org/lug-l/`)
- verschiedene Umgebungen und Werkzeuge ausprobieren
- aktuell bleiben (Zeitschriften abonnieren, Newsgruppen lesen)
- kommunizieren

Fred Brooks: The Mythical Man Month

- Suchen Sie (google) Rezensionen zu diesem Buch.
- Was ist *Brooks' Gesetz*? (“Adding . . . ”)
- Was sagt Brooks über Prototypen? (“Plan to . . . ”)
- Welche anderen wichtigen Bücher zur Softwaretechnik werden empfohlen?

Edsger W. Dijkstra über Softwaretechnik

- Dijkstra-Archiv
  `http://www.cs.utexas.edu/users/EWD/`
- Thesen zur Softwaretechnik `http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1305.PDF`

Was macht diese Funktion?

```
public static int f (int x, int y, int z) {
 if (x <= y) {
  return z;
 } else {
  return
   f (f (x-1,y,z), f(y-1,z,x), f(z-1,x,y));
 }
}
```

- wieviele rekursive Aufrufe finden statt?
- kann man das Ergebnis vorhersagen, ohne alle rekursiven Aufrufe durchzuführen?

Beispiele („real life“)

Interfaces (Schnittstellen) im täglichen Leben:

- Batterien
- CD/DVD (Spieler/Brenner, Rohlinge,...)
- Auto(-Vermietung)
- ...

Schnittstellen und -Vererbung in der Mathematik

- Halbgruppe, Monoid, Gruppe, Ring, Körper, Vektorraum
- Halbordnung, (totale) Ordnung
  vgl. Beschreibung von `Comparable<E>`

Schnittstellen von abstrakten Datentypen

- Folge
  einfügen, löschen, zugreifen (über Index)
- Menge
  einfügen, löschen, suchen (eines Elementes)
- Abbildung
  einfügen, löschen, anwenden (für Argument)

Benutzerschnittstellen

von Softwareprodukten

- Kommandozeile
- stdin/stdout/stderr
- Dateien
- Sockets
- GUI
- API

Schnittstellen und Code-Eigenschaften

Schnittstellendesign beeinflußt

- Benutzbarkeit
  - durch Menschen
  - durch Programme (andere Komponenten)
- Konfigurierbarkeit
- Testbarkeit

Schnittstellen und Entwurf

- jede Schnittstelle drückt eine *Abstraktion* aus (verdeutlicht das Wesentliche, ignoriert das Unwesentliche)
- größere Systeme lassen sich *nur* mit Abstraktionen beschreiben (planen), herstellen, testen, benutzen
- die dabei benutzten Methoden sind *die gleichen* wie bei „kleinen“ Systemen

Schnittstellen (interfaces) in Java

Beispiel in Eclipse

- Eclipse (Window $\rightarrow$ Preference $\rightarrow$ Compiler $\rightarrow$ Compliance 6.0)
- Klasse A mit Methode main
- in A.main: `B x = new B();` , Fehler $\rightarrow$ Control-1, Klasse B anlegen
- in A.main: `x.p();`, Fehler $\rightarrow$ Control-1, Methode p anlegen
- in B: Refactor $\rightarrow$ extract interface.

Literatur zu Schnittstellen

Ken Pugh: *Interface Oriented Design*, 2006. ISBN 0-0766940-5-0. `http://www.pragmaticprogrammer.com/titles/kpiod/index.html`
enthält Beispiele:

- Pizza bestellen
- Unix devices, file descriptors
- textuelle Schnittstellen
- grafische Schnittstellen

Komponenten und Schnittstellen

jede (Software-)Komponente implementiert eine Schnittstelle, die die notwendigen Eigenschaften beschreibt.

- Unterprogramm (Funktion):
  Zusammenhang zwischen Argument und Resultat (Ein- und Ausgabedaten)
- Anweisung, Unterprogramm (Prozedur):
  Zusammenhang zwischen Programmzustand vor und nach der Ausführung (Vor- und Nachbedingung)
- Klasse:
  Objekt-Invarianten (Beziehungen zwischen Attributen)

Schrittweise Verfeinerung

- von der Spezifikation zur Implementierung
- durch fortgesetztes Einfügen von Schnittstellen für neue, kleinere Systemkomponenten
- bis schließlich die Komponenten so klein sind, daß man sie trivial implementieren kann

Beispiel: Türme von Hanoi:

```
hanoi (int k, turm x, turm y, turm z) :
  bewege Scheiben [1 .. k]
  von x nach y mithilfe von z
```

Schnittstellen (Interfaces) in Java

interface ist eine teilweise Spezifikation einer Klasse

- statische Eigenschaften sind angegeben (Typen von Methoden)
  und werden zur Übersetzungszeit geprüft
- dynamische Eigenschaften (Invarianten, Vor- und Nachbedingungen) lassen sich nicht angeben
  die Überprüfung wäre unentscheidbar
  (Gödel, Turing, Rice: jede nichttriviale Programm-Eingenschaft ist unentscheidbar, Halteproblem)

Überprüfen von Software-Eigenschaften

- dynamische Eigenschaften als statische Eigenschaften formulieren (aussagefähige Typen verwenden)
- verbleibende nicht-statische Eigenschaften:
  - in einer Spezifikationssprache formulieren und beweisen (Coq für ML, Anna für Ada)
  - in der Programmiersprache selbst formulieren und testen (Unit-Tests)
  - in einer natürlichen Sprache formulieren (Javadoc)

Abstrakte Datentypen

Beispiele (Java Collections Framework)

- Schnittstellen (abstrakte Datentypen):
  Folge, Menge, Abbildung
- Implementierungen (konkrete Datentypen):
  ...

Definition:

- ADT ist *Spezifikation* ($=$ *Signatur* und *Axiome*)
- KDT ist ein *Modell der Spezifikation*: zur Signatur passende *Algebra*, die die Axiome erfüllt

Abstrakter Datentyp Menge

Signatur:

- `Set <E> empty ()`
- `boolean null (Set <E>)`
- `Set<E> insert (Set<E>, E)`
- `boolean contains (Set<E>, E)`

explizites `this`, unveränderliche Objekte
Axiome (Beispiele):

- `null (empty ())`
- `forall E x, Set<E> s : not (null ...)`
- `forall E x, Set<E> s : contains ...`

Übung: Signatur und Axiome für Löschen, Vereinigung.

Generische Polymorphie

- Typschema:
  Typ-Ausdruck mit Typ-Variable(n) in spitzen Klammern,
  Bsp: `Set<E>, Map<K,V>`
- Typ: Instantiierung des Schemas,
  Schema-Variablen durch Typen ersetzt,
  Bsp: `Set<Integer>, Map<String,Set<String>>`

In Java: Schema-Argumente müssen Verweistypen sein,
deswegen (Auto-)Boxing für primitive Typen
(int $\rightarrow$ Integer, boolean $\rightarrow$ Boolean, char $\rightarrow$ Character,...)

Generische Polymorphie (II)

- polymorphe Klasse/Schnittstelle:

```
interface Foo<E> { ... }
```

- polymorphe Methode (Deklaration)

```
<F> void bar(List <F> xs, List<E> ys) { .. }
```

- polymorphe Methode (Aufruf)

```
p.<Integer>bar ( ..., ... )
```

nicht-statische Methoden erhalten zusätzlich die Typ-Parameter ihrer Klasse (statische nicht)
Ü: Syntax für generische Polymorphie in C#

ADTs in der Mathematik

- Relationen: Funktion, injektiv, surjektiv, bijektiv
- Ordnungsstrukturen:
  transitive Relation, Halbordnung, lineare Ordnung, Verband
- „Rechen-“Strukturen:
  Halbgruppe, Monoid, Gruppe,
  Halbring, Ring, Körper, Vektorraum

Ü: jeweils Signatur, Axiome, Implementierungen angeben

Beispiel für Spezifikation

Signatur:

- Sorten: P, G
- Funktion: `boolean I(P, G)`

Axiome:

- $\forall x \in P : \exists_{\geq 2} y \in G : I(x, y)$
- $\forall y \in G : \exists_{\geq 2} x \in P : I(x, y)$
- $\forall x, y \in P : x \neq y \Rightarrow \exists_{=1} z \in G : I(x, z) \wedge I(y, z)$
- $\forall y, z \in G : y \neq z \Rightarrow \exists_{=1} x \in P : I(x, y) \wedge I(x, z)$

Ü: gibt es Modelle mit $|P| = 2, 3, 4, \ldots$?
Ü: sind die Axiome unabhängig?
Ü: übersetze $\exists_{\geq 2}, \exists_{=1}$ in $\exists (= \exists_{\geq 1})$

Aussagenlogik (Syntax)

aussagenlogische Formel ist

- Konstante (Wahr, Falsch)
- oder Variable ($p, q, \ldots \in V$)
- oder zusammengesetzte Formel:
  $\neg F_1, F_1 \wedge F_2, F_1 \vee F_2, \ldots$

Aussagenlogik (Semantik)

Belegung ist Abbildung $b : V \to \{0, 1\}$
Wert einer Formel F unter einer Belegung b:

- $\text{wert}(\text{Wahr}, b) = 1, \text{wert}(\text{Falsch}, b) = 0,$
- für $v \in V$: $\text{wert}(v, b) = b(v)$,
- $\text{wert}(F_1 \vee F_2, b) = \max(\text{wert}(F_1, b), \text{wert}(F_2, b))$

Aussagenlogik (Eigenschaften)

Eine Formel F heißt

- allgemeingültig,
  wenn für jede Belegung b gilt: $\text{wert}(F, b) = 1$
- erfüllbar,
  wenn eine Belegung b existiert: $\text{wert}(F, b) = 1$

Prädikatenlogik (Signatur)

Eine *Signatur* besteht aus

- einer Menge von Funktions-Symbolen
- und einer Menge von Prädikat-Symbolen,

jeweils mit Stelligkeiten (einsortige Signatur) oder Typen (mehrsortige Signatur).

Prädikatenlogik (Terme)

Ein *Term* in einer Signatur ist

- eine Variable
- oder ein Funktionssymbol mit einer passenden Anzahl von Argumenten ($=$ Termen)

Prädikatenlogik (Formeln)

Eine Formel in einer Signatur ist

- ein Prädikatsymbol mit einer passenden Anzahl von Argumenten (= Termen)
- oder eine aussagenlogischer Operator mit einer passenden Anzahl von Argumenten (= Formeln)
- oder ein Quantor mit einer Variablen und einem Argument (= Formel)

Prädikatenlogik (Bindungen)

Jede Formel hat eine Baumstruktur (vgl. Ausgabe autotool)
Ein Vorkommen einer Variablen x heißt *gebunden*, falls sich auf dem Pfad vom Vorkommen zur Wurzel ein Quantor befindet, der x bindet.
(sonst heißt das Vorkommen *frei*)

Prädikatenlogik (Strukturen)

Eine Struktur zu einer Signatur besteht aus

- einem Grundbereich (Universum) U
- einer Zuordnung: k-stelliges Funktionssymbol $\rightarrow$ Funktion $U^k \rightarrow U$
- einer Zuordnung: k-stelliges Relationssymbol $\rightarrow$ Teilmenge von U^k
- einer Belegung (Abbildung Variable $\rightarrow$ Universum)

Prädikatenlogik (Semantik - Terme)

Wert eines Termes in einer Struktur, unter einer Belegung, ist ein Element des Universums

- Variable: benutze Belegung
- Funktionssymbol mit Argumenten: wende Interpretation der Funktion auf Werte der Argumente an

Prädikatenlogik (Semantik - Formeln)

Wert einer Formel in einer Struktur, unter einer Belegung, ist ein Wahrheitswert

- Prädikatsymbol mit Argumenten: wende Interpretation des Prädikatsymbols auf Werte der Argumente an
- aussagenlogische Verknüpfung (wie bei Aussagenlogik)
- Quantoren:
  $\text{wert}(\forall x.F, S, b) = \min\{\text{wert}(F, S, b[x := u]) \mid u \in U\}$

wobei $b[x := u]$ die Belegung b' ist mit:
$b'(y) = \text{wenn } y = x \text{ dann } u \text{ sonst } b(y)$.

Quantoren

Der All-Quantor (über einem endlichen Bereich) entspricht einem logischen „und “.

- $\forall x \in \{0, 1, 2, 3\} : x^2 < 10$
- $0^2 < 10 \wedge 1^2 < 10 \wedge 2^2 < 10 \wedge 3^2 < 10$
- Realisierung in C#

```
using System.Linq;
Enumerable.Range(0,3).All(x  =>  x*x < 10)
```

  - lokale Funktion, Typinferenz
  - Funktion höherer Ordnung, extension method

(Existenz-Quantor: logisches „oder“, C#: `Any`)

Definition, Motivation

Software(-Komponente) testen $=$ für bestimmte Eingaben ausführen und Resulate mit Spezifikation vergleichen

- Spezifikation $\Rightarrow$ Testfälle
- bei Fehlen einer formalen Spezifikation sind Testfälle die nächstbeste Näherung

test driven development ($=$ erst Testfälle schreiben, danach Quelltexte) bedeutet: erst spezifizieren, dann implementieren.

Tests und Schnittstellen

zur jeder Art von Schnittstelle gehört eine Art von Tests, z. B.

- Benutzerschnittstelle (Web): Click-Recorder/Replayer/Verifier (Bsp. `http://seleniumhq.org/`)
- textuelle Schnittstellen: Textvergleiche z. B. mit `diff`
- Komponentenschnittstellen (Methoden): unit tests (Java: `http://www.junit.org/`, C#: `http://www.nunit.org/`)
- Schnittstellen zwischen Anweisungen (innerhalb einer Methode): Zusicherungen (assert) (z. B. für Invarianten)

JUnit

Beispiel:

```
import static org.junit.Assert.*;
import org.junit.Test;
public class T {
    @Test
    public void t1 () {
        assertTrue(1 + 2 == 3);
    }
}
```

Eclipse: new $\rightarrow$ junit(4)test case, run as $\rightarrow$ test case

NUnit

Quelltext:

```
using NUnit.Framework;
[TestFixture] public class Test {
    [Test] public void check() {
        Assert.IsTrue (1+2 == 3);
    }
}
```

Kompilation:

```
gmcs -r:nunit.framework -t:library Test.cs
```

Ausführung:

```
nunit-console Test.dll
```

Weitere Beispiele: benutze All, Any zur Spezifikation von Halbgruppe, Gruppe.

Entwurfsmuster

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:
Entwurfsmuster (design patterns) — Elemente
wiederverwendbarer objektorientierter Software,
Addison-Wesley 1996.
liefert Muster (Vorlagen) für Gestaltung von Beziehungen
zwischen (mehreren) Klassen und Objekten, die sich in
wiederverwendbarer, flexibler Software bewährt haben.

Seminarvorträge `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss05/case/seminar/`

Beispiel zu Entwurfsmustern

aus: Gamma, Helm, Johnson, Vlissides: *Entwurfsmuster*
Beispiel: ein grafischer Editor.
Aspekte sind unter anderem:

- Dokumentstruktur
- Formatierung
- Benutzungsschnittstelle

Beispiel: Strukturmuster: Kompositum

darstellbare Elemente sind:
Buchstabe, Leerzeichen, Bild, Zeile, Spalte, Seite
Gemeinsamkeiten?
Unterschiede?

Beispiel: Verhaltensmuster: Strategie

Formatieren (Anordnen) der darstellbaren Objekte:
möglicherweise linksbündig, rechtsbündig, zentriert

Beispiel: Strukturmuster: Dekorierer

darstellbare Objekte sollen optional eine Umrahmung oder eine Scrollbar erhalten

Beispiel: Erzeugungsmuster: (abstrakte) Fabrik

Anwendung soll verschiedene GUI-Look-and-Feels ermöglichen

Beispiel: Verhaltensmuster: Befehl

Menü-Einträge, Undo-Möglichkeit

siehe auch Ereignisbehandlung in Applets

Wie Entwurfsmuster Probleme lösen

- Finden passender Objekte

  insbesondere: nicht offensichtliche Abstraktionen
- Bestimmen der Objektgranularität
- Spezifizieren von Objektschnittstellen und Objektimplementierungen

  unterscheide zwischen *Klasse* (konkreter Typ) und *Typ* (abstrakter Typ).

  programmiere auf eine Schnittstelle hin, nicht auf eine Implementierung!
- Wiederverwendungsmechanismen anwenden

  ziehe Objektkomposition der Klassenvererbung vor
- Unterscheide zw. Übersetzungs- und Laufzeit

Vorlage: Muster in der Architektur

Christopher Alexander: *The Timeless Way of Building, A Pattern Language*, Oxford Univ. Press 1979.

10. Menschen formen Gebäude und benutzen dazu Muster-Sprachen. Wer eine solche Sprache spricht, kann damit unendlich viele verschiedene einzigartige Gebäude herstellen, genauso, wie man unendlich viele verschiedene Sätze in der gewöhnlichen Sprache bilden kann.

14. . . . müssen wir tiefe Muster entdecken, die Leben erzeugen können.

15. . . . diese Muster verbreiten und verbessern, indem wir sie testen: erzeugen sie in uns das Gefühl, daß die Umgebung lebt?

16. . . . einzelne Muster verbinden zu einer Sprache für gewisse Aufgaben . . .

Strukturmuster: Kompositum

Finde wenigstens sieben (Entwurfs-)Fehler und ihre wahrscheinlichen Auswirkungen...

```
class Geo {
    int type; // Kreis, Quadrat,
    Geo teil1, teil2; // falls Teilobjekte
    int ul, ur, ol, or; // unten links, ...
    void draw () {
        if (type == 0) { ... } // Kreis
        else if (type == 1) { ... } // Quadrat
    }
}
```

Kompositum - Anwendung

so ist es richtig:

```
interface Geo {
    Box bounds ();
    Geo [] teile ();
    void draw ();
}
class Kreis implements Geo { .. }
class Neben implements Geo {
    Neben (Geo links, Geo Rechts) { .. }
}
```

Signaturen und Algebren

- (mehrsortige) Signatur Σ
  Menge von Funktionssymbolen, für jedes: Liste von Argumenttypen, Resultattyp
- A ist Σ-Algebra:
  Trägermenge und typkorrekte Zuordnung von Funktionssymbolen zu Funktionen
- Beispiel: Signatur für Vektorraum V über Körper K

Termalgebra (Bäume)

zu jeder Signatur Σ kann man die Algebra Term(Σ) konstruieren:

- Trägermenge sind alle typkorrekten Σ-Terme
- jedes Symbol $f \in \Sigma$ wird durch „sich selbst“ implementiert

anderer Name für diese Algebra: *algebraischer Datentyp*.

Algebraische Datentypen

- Listen:

```
data List a = Nil | Cons a (List a)
```

- Bäume (mit Schlüsseln in Blättern):

```
data Tree a = Leaf a
            | Branch (Tree a) (Tree a)
```

- Übung: Peano-Zahlen, Wahrheitswerte

Def: *Kompositum* = rekursiver algebraischer Datentyp

Entwurfsfragen bei Bäumen

- Knoten sind *innere* (Verzweigung) und *äußere* (Blatt).
- Die “richtige” Realisierung ist Kompositum

```
interface Tree<K>;
class Branch<K> implements Tree<K>;
class Leaf<K> implements Tree<K>;
```

- Möglichkeiten für Schlüssel: in allen Knoten, nur innen, nur außen.

Wenn Blätter keine Schlüssel haben, geht es musterfrei?

```
class Tree<K> // Branch, mit Leaf == null;
```

Jein. — betrachte Implementierung in `java.util.Map<K,V>`

Pattern Matching

```
data Tree a  = Leaf a
     | Branch ( Tree a ) ( Tree a )
```

Verarbeitung von Objekten dieses Typs:

```
leaves :: Tree a -> Int
leaves t = case t of
     Leaf k -> 1
     Branch l r -> leaves l + leaves r
```

- für jeden Konstruktur des Datentyps
  einen Zweig der Mehrfachverzweigung
- Rekursion im Typ $\rightarrow$ Rekursion im Programm

Beispiel: Scala

`http://scala-lang.org`

algebraische Datentypen:

```
abstract class Tree[A]
case class Leaf[A](key: A) extends Tree[A]
case class Branch[A]
    (left: Tree[A], right: Tree[A])
        extends Tree[A]
```

pattern matching:

```
def size[A](t: Tree[A]): Int = t match {
    case Leaf(k) => 1
    case Branch(l, r) => size(l) + size(r)
  }
```

Maybe = Nullable

Algebraischer Datentyp (Haskell):

```
data Maybe a = Nothing | Just a
```

http://hackage.haskell.org/packages/archive/base/latest/doc/html/Prelude.html#t:Maybe

Realisierung in C#: `Nullable<T>`

http://msdn.microsoft.com/en-us/library/2cf62fcy.aspx

Entweder—Oder

```
data Either a b = Left a | Right b
```

häufig benutzt für: Resultat (Right) oder Fehlermeldung (Left)

Listen (einfach verkettet)

```
data List a = Nil
            | Cons a ( List a )
```

Die Liste [1,2,3] wird realisiert als

```
Cons 1 (Cons 2 (Cons 3 Nil))
```

Implementiere:

```
null :: List a -> Bool
length :: List a -> Int
equals :: Eq a => List a -> List a -> Bool

append :: List a -> List a -> List a
reverse :: List a -> List a

merge :: Ord a => List a -> List a -> List a
```

Verifikation von funktionalen Programmen

Beweisen Sie:

```
append ist assoziativ
reverse ( reverse xs ) == xs
```

Beweis-Methoden:

- Term-Umformungen (= equational reasoning)
  Ersetzung von äquivalenten Teiltermen
- Induktion
  - Induktions-Anfang (bei Listen: `Nil`)
  - Induktions-Schritt (bei Listen: `Cons`)

d. h. genauso wie in der Mathematik

Listen in Haskell

der eingebaute Listen-Typ ist strukturgleich zu dem hier besprochenen, wird aber so notiert:

```
List a     -->  [a]
Nil        -->  []
Cons x xs  -->  x : xs
```

Funktionen zum Zugriff auf Konstruktor-Argumente:

```
head :: [a] -> a
head (x:xs) = x

tail :: [a] -> [a]
tail (x:xs) = xs
```

Vorsicht: das sind partielle Funktionen

Peano-Zahlen

```
data N = Z | S N
```

Implementieren Sie für Peano-Zahlen:

- Addition, Multiplikation, Potenz

Beweisen Sie :

- Assoziativität, Kommutatitivität, ...

Verhaltensmuster: Besucher

Prinzip:

- rekursive Datenstruktur (algebraischer Datentyp, Kompositum)

$\Rightarrow$ Rekursionsmuster für Algorithmen, die diese Struktur benutzen.

Implementierungen:

- map/fold in Haskell (funktional)
- Besucher in Java (objektorientiert)
- Select/Aggregate in C# (funktional)

Rekursion über Bäume (Beispiele)

```
data Tree a     = Leaf
  | Branch ( Tree a ) a ( Tree a )
summe :: Tree N -> N
summe t = case t of
  Leaf -> Z
  Branch l k r ->
    plus (summe l) (plus k (summe r ))
preorder :: Tree a -> List a
preorder t = case t of
  Leaf -> Nil
  Branch l k r ->
   Cons k (append (preorder l)
                  (preorder r))
```

Rekursion über Bäume (Schema)

gemeinsame Form dieser Funktionen:

```
f :: Tree a -> b
f t = case t of
  Leaf          -> ...
  Branch l k r -> ... (f l) k (f r)
```

dieses Schema ist eine Funktion höherer Ordnung:

```
fold :: ( ... ) -> ( ... ) -> ( Tree a -> b )
fold leaf branch = \ t -> case t of
  Leaf -> leaf
  Branch l k r ->
     branch (fold leaf branch l) k
            (fold leaf branch r)
summe = fold Z ( \ l k r -> plus l (plus k r ) )
```

Rekursion über Listen

```
and :: List Bool  -> Bool
and xs = case xs of
     Nil -> True ; Cons x  xs' -> x && and xs'
length :: List a -> Int
length xs = case xs of
    Nil -> 0 ; Cons x xs' ->  1 + length xs'

fold :: b -> ( a -> b -> b ) -> [a] -> b
fold nil cons xs = case xs of
    Nil -> nil
    Cons x xs' -> cons x ( fold nil cons xs' )
and = fold True (&&)
length = fold 0 ( \ x y ->  1 + y)
```

Rekursionsmuster (Prinzip)

jeden Konstruktor durch eine passende Funktion ersetzen.

```
data List a = Nil | Cons a (List a)
fold ( nil :: b ) ( cons :: a -> b -> b )
   :: List a -> b
```

Rekursionsmuster instantiieren = (Konstruktor-)Symbole interpretieren (durch Funktionen) = eine Algebra angeben.

```
length  = fold 0   ( \ _ l -> l + 1 )
reverse = fold Nil ( \ x ys ->      )
```

Rekursion über Listen (Übung)

Aufgaben:

- `append, reverse, concat, inits, tails`
  mit `fold` (d. h., ohne Rekursion)

Bezeichnungen in Haskell-Bibliotheken:
das vordefinierte Rekursionsschema über Listen ist:

```
foldr :: (a -> b -> b) -> b -> ([a] -> b)
length = foldr ( \ x y -> 1 + y ) 0
```

Beachte:

- Argument-Reihenfolge (erst cons, dann nil)
- foldr nicht mit foldl verwechseln (foldr ist das „richtige“)

Fold/Besucher in C#

fold für Listen = System.Linq.Aggregate

```
import System.Linq;
import System.Collections.Generic;

List<int> l =
    new List<int>() { 3,1,4,1,5,9 };
Console.WriteLine
    (l.Aggregate(0, (x,y) => x+y));
```

Besucher-Muster für Bäume

```
interface Tree<K> { }
class Leaf<K>   implements Tree<K> {
  Leaf(K key) { .. } }
class Branch<K> implements Tree<K> {
  Branch(Tree<K> left, Tree<K> right) { .. } }
```

für jeden Teilnehmer des Kompositums eine Methode:

```
interface Visitor<K,R> { // mit Resultattyp R
    R leaf (K x);
    R branch (R left, R right);  }
```

der Gastgeber (Baumknoten) nimmt Besucher auf:

```
interface Tree<K> {
  <R> R receive (Visitor<K,R> v) }
```

Beispiel: Baum-Besucher

Benutzung des Besuchers: Anzahl der Blätter:

```
class Trees {
  static <K> int leaves (Tree<K> t) {
    return t.receive(new Tree.Visitor<K,Integer>()
      public Integer branch
            (Integer left, Integer right) {
        return left + right;
      }
      public Integer leaf(K key) {
        return 1;
      }
    });
  }
}
```

Verhaltensmuster: Iterator

- Motivation (Streams)
- Definition Iterator
- syntaktische Formen (foreach, yield return)
- Baumdurchquerung mit Stack bzw. Queue

Unendliche Datenstrukturen

```
nats :: [ Integer ]
nats = from 0 where
     from x = x : from (x+1)

fibs :: [ Integer ]
fibs = 0 : 1
      : zipWith (+) fibs ( drop 1 fibs )
```

das ist möglich, wenn der *tail* jeder Listenzelle erst bei Bedarf erzeugt wird.
(Bedarfsauswertung, lazy evaluation)
lazy Liste = Stream = Pipeline, vgl. InputStream (Console)

Rechnen mit Streams

Unix:

```
cat stream.tex | tr -c -d aeuio | wc -m
```

Haskell:

```
sum $ take 10 $ map ( \ x -> x^3 ) $ naturals
```

C#:

```
Enumerable.Range(0,10).Select(x=>x*x*x).Sum();
```

- logische Trennung:
  Produzent $\rightarrow$ Transformator(en) $\rightarrow$ Konsument
- wegen Speichereffizienz: verschränkte Auswertung.
- gibt es bei *lazy* Datenstrukturen geschenkt, wird ansonsten durch Iterator (Enumerator) simuliert.

Iterator (Java)

```
interface Iterator<E> {
  boolean hasNext(); // liefert Status
  E next(); // schaltet weiter
}
interface Iterable<E> {
  Iterator<E> iterator();
}
```

typische Verwendung:

```
Iterator<E> it = c.iterator();
while (it.hasNext()) {
  E x = it.next (); ...
}
```

Abkürzung: `for (E x : c) { ... }`

Beispiele Iterator

- ein Iterator (bzw. Iterable), der/das die Folge der Quadrate natürlicher Zahlen liefert
- Transformation eines Iterators (map)
- Zusammenfügen zweier Iteratoren (merge)
- Anwendungen: Hamming-Folge, Mergesort

Beispiel Iterator Java

```
Iterable<Integer> nats = new Iterable<Integer>() {
  public Iterator<Integer> iterator() {
    return new Iterator<Integer>() {
      int s = 0;
      public Integer next() {
        int res = s ; s++; return res;
      }
      public boolean hasNext() { return true; }
    };
  }
};
for (int x : nats) { System.out.println(x); }
```

Aufgabe: implementiere (und benutze) eine Methode

```
static Iterable<Integer> range(int start, int count
```

soll `count` Zahlen ab `start` liefern

Enumerator (C#)

```
interface IEnumerator<E> {
  E Current; // Status
  bool MoveNext (); // Nebenwirkung
}
interface IEnumerable<E> {
  IEnumerator<E> GetEnumerator();
}
```

typische Benutzung: ...
Abkürzung: `foreach (E x in c) { ... }`

Iteratoren mit yield

```
class Range : IEnumerable<int> {
    private readonly int lo;
    private readonly int hi;
    public Range(int lo, int hi) {
        this.lo = lo; this.hi = hi;
    }
    public IEnumerator<int> GetEnumerator() {
        for (int x = lo; x < hi ; x++) {
            yield return x;
        }
        yield break;
    }
}
```

Aufgaben Iterator C#

```
IEnumerable<int> Nats () {
    for (int s = 0; true; s++) {
        yield return s;
    }
}
```

Implementiere „das merge aus mergesort“(Spezifikation?)

```
static IEnumerable<E> Merge<E>
    (IEnumerable<E> xs, IEnumerable<E> ys)
  where E : IComparable<E>
```

zunächst für unendliche Ströme, Test:

```
Merge(Nats.Select(x=>x*x),Nats.Select(x=>3*x+1)).Ta
```

Dann auch für endliche Ströme, Test:

```
Merge(new int [] {1,3,4}, new int [] {2,7,8})
```

Dann Mergesort

```
   static IEnumerable<E> Sort<E> (IEnumerable<E> xs
        where E : IComparable<E> {
```

Streams in C#: funktional, Linq

Funktional

```
IEnumerable.Range(0,10).Select(x => x^3).Sum();
```

Typ von Select? Implementierung?
Linq-Schreibweise:

```
(from x in new Range(0,10) select x*x*x).Sum();
```

Beachte: SQL-select „vom Kopf auf die Füße gestellt“.

Befehl

Beispiel:

```
interface ActionListener {
  void actionPerformed( ActionEvent e);
}
JButton b = new JButton ();
b.addActionListener (new ActionListener() {
  public void actionPerformed (ActionEvent e) { ..
} );
```

trennt Befehls-Erzeugung von -Ausführung,
ermöglicht Verarbeitung von Befehlen (auswählen, speichern, wiederholen)

Strategie

$\approx$ öfter benutzter Befehl, mit Parametern
Beispiel:

```
interface Comparator<T> { int compare (T x, Ty); }
List<Integer> xs = ...;
Collections.sort
    (xs, new Comparator<Integer>() { ... });
```

Übung:

- sortiere Strings länge-lexikografisch, ...
- wo wird Transitivität, Linearität der Relation benutzt?

Strategie (Beispiel II)

```
public class Strat extends JApplet {
  public void init () {
    JPanel p = new JPanel
       (new GridLayout(8,0)); // Strategie-Objekt
    for (int i=0; i<40; i++) {
      p.add (new JButton ());
    }
    this.getContentPane().add(p);
  }
}
```

Bemerkungen: Kompositum (Wdhlg), MVC (später)

Funktionen höherer Ordnung (I)

. . . als Erklärung für Strategie-Muster

- ```
  merge ::
       [Integer] -> [Integer] -> [Integer]
  ```
- ```
  data Ordering = LT | EQ | GT
  type Comparator a = ( a -> a -> Ordering )
  merge :: Comparator a
        -> [a] -> [a] -> [a]
  ```

OOP als Simulation für FP:

- FP: eine Funktion *f*
- OOP: ein Objekt mit Methode *f*

Funktionen höherer Ordnung (II)

- Definition

```
flip :: Comparer a -> Comparer a
flip comp = \ x y -> comp y x
```

- Anwendung:

```
mergesort comp xs
mergesert (flip comp) xs
```

- Aufgabe: `flip` in Java (für einen Comparator die Argumente vertauschen)

Muster: Interpreter (Motivation)

(Wdhlg. Iterator)

```
enum Colour { Red, Green, Blue }
class Car { int wheels; Colour colour, }
class Store {
  Collection<Car> contents;
  Iterable<Car> all ();
}
```

soweit klar, aber wie macht man das besser:

```
class Store { ...
  Iterable<Car> more_than_5_wheels ();
  Iterable<Car> red ();
  Iterable<Car> green_and_atmost_3_wheels ();
}
```

Muster: Interpreter (Realisierung)

algebraischer Datentyp (= Kompositum) für die Beschreibung von Eigenschaften

```
interface Property { }
```

Blätter (Konstanten)

```
class Has_Color { Color c }
class Has_Max_Wheels { int x }
```

Verzweigungen (Kombinatoren)

...

und Methode zur Auswertung einer (zusammengesetzten) Eigenschaft für gegebenes Datum. (Typ?)
Interpreter-Muster := Kompositum als Strategie-Objekt

Interpreter in FP

allgemein:

```
interpreter :: Programm -> Daten -> Resultat
```

Beispiel

```
data Colour = Red | Green
data Car = Car { wheels :: Integer , colour :: Colo

data Property = Colour_Is Colour
              | Max_Wheels Integer
              | And Property Property

evaluate :: Property -> Car -> Bool
```

Anwendung:

```
filter ( evaluate p ) cars
```

Query-Sprachen

DSL: domainspezifische Sprache, hier: für Datenbankabfragen

- externe DSL (Frage = String)

```
Person aPerson = (Person) session
  .createQuery("select p from Person p left join
     where p.id = :pid")
  .setParameter("pid", personId)
```

- embedded DSL (Frage = Objekt)
- typsichere embedded DSL
- (gar keine Datenbank: `http://happstack.com/`)

Hibernate Criteria Query API

```
http://www.hibernate.org/

import org.hibernate.criterion.Criterion; ...
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.between
            ("weight", minWeight, maxWeight) )
    .list();
```

Linq in C#

```
IEnumerable<Car> cars = new List<Car>()
    { new Car(){wheels = 4,
                colour = Colour.Red},
      new Car(){wheels = 3,
                colour = Colour.Blue} };
foreach (var x in from c in cars
 where c.colour == Colour.Red
 select c.wheels) {
    System.Console.WriteLine (x);
}
```

http://msdn.microsoft.com/en-us/netframework/aa904594.aspx

Datenquellen: Collections, XML, DB

Entwurfsmuster: Zustand

Zustand eines Objektes = Belegung seiner Attribute
Zustand erschwert Programm-Benutzung und -Verifikation (muß bei jedem Methodenaufruf berücksichtigt werden).
Abhilfe: Trennung in

- Zustandsobjekt (nur Daten)
- Handlungsobjekt (nur Methoden)

jede Methode bekommt Zustandsobjekt als Argument

Zustand (Beispiel)

- Zustand implizit

```
class C0 {
  private int z = 0;
  public void step () { this.z++; }
}
```

- Zustand explizit

```
class C1 {
  public int step (int z) { return z + 1; }
}
```

Zustand und Spezifikation

Für Programm-Spezifikation (und -Verifikation) muß der Zustand sowieso benannt werden,
und verschiedene Zustände brauchen verschiedene Namen (wenigstens: vorher/nachher)
also kann man sie gleich durch verschiedene Objekte repräsentieren.

Zustand in Services

- *unveränderliche* Zustandsobjekte:
- Verwendung früherer Zustandsobjekte (undo, reset, test)

wiederverwendbare Komponenten („Software als Service“)
dürfen *keinen* Zustand enthalten.
(Thread-Sicherheit, Load-Balancing usw.)
(vgl.: Unterprogramme dürfen keine globalen Variablen benutzen)

in der (reinen) funktionalen Programmierung passiert das von selbst: dort *gibt es keine Zuweisungen* (nur const-Deklarationen mit einmaliger Initialisierung).
$\Rightarrow$ Thread-Sicherheit ohne Zusatzaufwand

Dependency Injection

Martin Fowler, `http://www.martinfowler.com/articles/injection.html`

Abhängigkeiten zwischen Objekten sollen

- sichtbar und
- konfigurierbar sein (Übersetzung, Systemstart, Laufzeit)

Formen:

- Constructor injection (bevorzugt)
- Setter injection (schlecht—dadurch sieht es wie „Zustand“ aus, unnötigerweise)

Verhaltensmuster: Beobachter

zur Programmierung von Reaktionen auf Zustandsänderung von Objekten

- Subjekt: class Observable
  - anmelden: void addObserver (Observer o)
  - abmelden: void deleteObserver (Observer o)
  - Zustandsänderung: void setChanged ()
  - Benachrichtigung: void notifyObservers(...)
- Beobachter: interface Observer
  - aktualisiere: void update (...)

Objekbeziehungen sind damit konfigurierbar.

Beobachter: Beispiel (I)

```
public class Counter extends Observable {
    private int count = 0;
    public void step () { this.count ++;
        this.setChanged();
        this.notifyObservers();   }   }
public class Watcher implements Observer {
    private final int threshold;
    public void update(Observable o, Object arg) {
       if (((Counter)o).getCount() >= this.threshol
           System.out.println ("alarm");  }   }  }
public static void main(String[] args) {
    Counter c = new Counter (); Watcher w = new Wat
    c.addObserver(w); c.step(); c.step (); c.step (
```

Beobachter: Beispiel Sudoku, Semantik

- Spielfeld ist Abbildung von Position nach Zelle,
- Menge der Positionen ist $\{0, 1, 2\}^4$
- Zelle ist leer (Empty) oder besetzt (Full)
- leerer Zustand enthält Menge der noch möglichen Zahlen
- Invariante?
- Zelle C_1 beobachtet Zelle C_2, wenn C_1 und C_2 in gemeinsamer Zeile, Spalte, Block

Test: eine Sudoku-Aufgabe laden und danach Belegung der Zellen auf Konsole ausgeben.

```
git clone git://dfa.imn.htwk-leipzig.de/srv/git/ss11-st2
http://dfa.imn.htwk-leipzig.de/cgi-bin/gitweb.cgi?
p=ss11-st2.git;a=tree;f=src/kw20;hb=HEAD
```

Beobachter: Beispiel Sudoku, GUI

Plan:

- Spielfeld als JPanel (mit GridLayout) von Zellen
- Zelle ist JPanel, Inhalt:
  - leer: JButton für jede mögliche Eingabe
  - voll: JLabel mit gewählter Zahl

Hinweise:

- JPanel löschen: `removeAll()`,
  neue Komponenten einfügen: `add()`,
  danach Layout neu berechnen: `validate()`
- JPanel für die Zelle einrahmen: `setBorder()`

Model/View/Controller

(Modell/Anzeige/Steuerung)
(engl. *to control* = steuern, *nicht:* kontrollieren)

Bestandteile (Beispiel):

- Model: Counter (getCount, step)
- View: JLabel ($\leftarrow$ getCount)
- Controller: JButton ($\rightarrow$ step)

Zusammenhänge:

- Controller steuert Model
- View beobachtet Model

javax.swing und MVC

Swing benutzt vereinfachtes MVC
(M getrennt, aber V und C gemeinsam).

Literatur:

- The Swing Tutorial `http://java.sun.com/docs/books/tutorial/uiswing/`
- Guido Krüger: Handbuch der Java-Programmierung, Addison-Wesley, 2003, Kapitel 35–38

Swing: Datenmodelle

```
JSlider top = new JSlider(JSlider.HORIZONTAL, 0, 10
JSlider bot = new JSlider(JSlider.HORIZONTAL, 0, 10
bot.setModel(top.getModel());
```

Aufgabe: unterer Wert soll gleich 100 - oberer Wert sein.

Swing: Bäume

```
// Model:
class Model implements TreeModel { .. }
TreeModel m = new Model ( .. );

// View + Controller:
JTree t = new JTree (m);

// Steuerung:
t.addTreeSelectionListener(new TreeSelectionListene
   public void valueChanged(TreeSelectionEvent e) {

// Änderungen des Modells:
m.addTreeModelListener(..)
```

Anwendung, Ziele

- aktuelle Quelltexte eines Projektes sichern
- auch frühere Versionen sichern
- gleichzeitiges Arbeiten mehrere Entwickler
- . . . an unterschiedlichen Versionen (Zweigen)

Das Management bezieht sich auf *Quellen* (.c, .java, .tex, Makefile)
abgeleitete Dateien (.obj, .exe, .pdf, .class) werden daraus erzeugt, stehen aber *nicht* im Archiv

Welche Formate?

- Quellen sollen Text-Dateien sein, human-readable, mit Zeilenstruktur: ermöglicht Feststellen und Zusammenfügen von unabhängigen Änderungen
- ergibt Konflikt mit Werkzeugen (Editoren, IDEs), die Dokumente nur in Binärformat abspeichern. — Das ist sowieso *evil*, siehe Robert Brown: Readable and Open File Formats, `http://www.few.vu.nl/~feenstra/read_and_open.html`
- Programme mit grafischer Ein- und Ausgabe sollen Informationen *vollständig* von und nach Text konvertieren können
  (Bsp: UML-Modelle als XMI darstellen)

Daten und Operationen

Daten:

- Archiv (repository)
- Arbeitsbereich (sandbox)

Operationen:

- check-out: repo $\rightarrow$ sandbox
- check-in: sandbox $\rightarrow$ repo

Projekt-Organisation:

- ein zentrales Archiv (CVS, Subversion)
- mehrere dezentrale Archive (Git)

Versionierung (intern)

. . . automatische Numerierung/Benennung

- CVS: jede Datei einzeln
- SVN: gesamtes Repository
- darcs: Mengen von Patches
- git: Snapshot eines (Verzeichnis-)Objektes

Objekt-Versionierung in Git

Git verwaltet (in `.git`) eine *persistente* Sicht auf den Verzeichnisbaum (inkl. aller Änderungen)

- Objekt-Typen:
  - Datei (blob),
  - Verzeichnis (tree), mit Verweisen auf blobs und trees
  - Commit
    mit Verweisen auf tree und commits (Vorgänger)

  ```
  git cat-file [-t|-p] <hash>
  ```
- Objekte sind *unveränderlich* und durch SHA1-Hash (160 bit $=$ 40 Hex-Zeichen) identifiziert
- statt Überschreiben: neue Objekte anlegen
- jeder frühere Zustand kann wiederhergestellt werden

Versionierung (extern)

. . . mittels Tags (manuell erzeugt)
empfohlenes Schema:

- Version $=$ Liste von drei Zahlen $[x, y, z]$
- Ordung: lexikographisch. (Spezifikation?)

Änderungen bedeuten:

- x (major): inkompatible Version
- y (minor): kompatible Erweiterung
- z (patch): nur Fehlerkorrektur

Sonderformen:

- y gerade: stabil, y ungerade: Entwicklung
- z Datum

Arbeit mit Zweigen (Branches)

- Repo anlegen: `git init`
- im Haupt-Zweig (master) arbeiten:
  `git add <file>; git commit -a`
- abbiegen:
  `git branch <name>; git checkout <name>`
- dort arbeiten: `... ; git commit -a`
- zum Haupt-Zweig zurück: `git checkout master`
- dort weiterarbeiten :`... ; git commit -a`
- zum Neben-Zweig: `git checkout <name>`
- Änderung aus Haupt-Zweig übernehmen:
  `git merge master`

Übernehmen von Änderungen (Merge)

durch divergente Änderungen entsteht Zustand mit 3 Versionen einer Datei:

- gemeinsamer Start *G*
- Versionen *I*, *D* (ich, du)

Merge:

- Änderung $G \rightarrow D$ *bestimmen*
- und auf *I anwenden*,
- falls das *konfliktfrei* möglich ist.

Änderung = Folge von Editor-Befehlen (Kopieren, Einfügen, Löschen)
betrachten dabei immer ganze Zeilen

LCS

Idee: die beiden Aufgaben sind äquivalent:

- kürzeste Edit-Sequenz finden
- längste gemeinsame Teilfolge (longest common subsequence) finden

Beispiel: $y = AB\boxed{C}\boxed{AB}B\boxed{A}, z = \boxed{C}B\boxed{AB}\boxed{A}C$
für $x = CABA$ gilt $x \leq y$ und $x \leq z$,
wobei die Relation $\leq$ auf Σ^* so definiert ist:
$u \leq v$, falls man u aus v durch *Löschen* einiger Buchstaben erhält (jedoch *ohne* die Reihenfolge der übrigen Buchstaben zu ändern)
vgl. mit Ausgabe von `git diff <commit-1> <commit-2>`

Die Einbettungs-Relation

Def: $u \leq v$, falls u aus v durch Löschen von Buchstaben

- ist Halbordnung (transitiv, reflexiv, antisymmetrisch),
- ist keine totale Ordnung

Testfragen:

- Gegeben v. Für wieviele u gilt $u \leq v$?
- Effizienter Algorithmus für: Eingabe u, v, Ausgabe $u \leq v$ (Boolean)

Die Einbettungs-Relation (II)

Begriffe (für Halbordnungen):

- Kette: Menge von paarweise vergleichbaren Elementen
- Antikette: Menge von paarweise unvergleichbaren Elementen

Sätze: für $\leq$ ist

- jede echt absteigende Kette endlich (trivial)
- jede Antikette endlich (nicht trivial)

Beispiel: bestimme die Menge der $\leq$-minimalen Elemente für die Menge der Dezimaldarstellungen der Quadratzahlen (das ist eine Antikette)

Die Einbettungs-Relation (III)

Die Endlichkeit von Antiketten bezüglich Einbettung gilt für

- Listen
- Bäume (Satz von Kruskal, 1960)
- Graphen (Satz von Robertson/Seymour)
  (Beweis über insgesamt 500 Seiten über 20 Jahre, bis ca. 2000)
  vgl. Kapitel 12 in: Reinhard Diestel: Graph Theory,
  `http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/`

LCS — naiver Algorithmus (exponentiell)

cvs2/LCS.hs

LCS — wie besser?

- naiver Algorithmus arbeitet top-down:
  sehr viele rekursive Aufrufe . . .
- aber nicht viele *verschiedene* . . .
  Optimierung durch bottom-up-Reihenfolge!
- dieses Algorithmen-Entwurfs-Prinzip heißt oft *dynamische Programmierung/Optimierung*
- vgl. rekursive/iterative Berechnung der Fibonacci-Zahlen u.ä.

LCS — bottom-up (quadratisch) + Übung

```
class LCS {
  // größte Länge einer gemeinsamen Teilfolge
  static int lcs (String xs, String ys) {
    int a[][] =
        new int [ ... ] [ ... ]
    for (int i = ... ; ... ; ... ) {
      for (int j = ... ; ... ; ... ) {
        // Spezifikation:
        // a[i][j] enthält größte Länge
        // einer gemeinsamen Teilfolge
        // von xs.substring(i)
        // und ys.substring(j)
      }
    }
    return ...
  }
  @Test
  public void test1 () {
      assertEquals (4, lcs ("ABCABBA","CBABAC"));
  }
```