Softwaretechnik II
Vorlesung
Sommersemester 2004,. . . , 2011

Johannes Waldmann, HTWK Leipzig

7. Juli 2011

Einleitung

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. ≤ 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

Übung KW 11

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?

Schnittstellen

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 → Preference → Compiler → Compliance 6.0)
- Klasse A mit Methode main
- in A.main: `B x = new B();` , Fehler → Control-1, Klasse B anlegen
- in A.main: `x.p();`, Fehler → Control-1, Methode p anlegen
- in B: Refactor → 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-Eigenschaft 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 → Integer, boolean → Boolean, char → 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 $\exists (= \exists_{\geq 1})$

Logik (Wiederholung)

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:

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

Aussagenlogik (Eigenschaften)

Eine Formel F heißt

- allgemeingültig,

  wenn für jede Belegung b gilt: wert$(F, b) = 1$
- erfüllbar,

  wenn eine Belegung b existiert: 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 $\to$ Funktion $U^k \to U$
- einer Zuordnung: k-stelliges Relationssymbol $\to$ Teilmenge von U^k
- einer Belegung (Abbildung Variable $\to$ 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:
  wert$(\forall x.F, S, b) = \min\{$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`)

Tests

Definition, Motivation

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

- Spezifikation ⇒ 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 → junit(4)test case, run as → 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: allgemein

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 …

17. … verschiedene Sprachen zu einer größeren Struktur verbinden: der gemeinsamen Sprache einer Stadt.

27. In Wirklichkeit hat das Zeitlose nichts mit Sprachen zu tun. Die Sprache beschreibt nur die natürliche Ordnung der Dinge. Sie lehrt nicht, sie erinnert uns nur an das, was wir schon wissen und immer wieder neu entdecken …

Algebraische Datentypen (Komposita)

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) { ... } // Quadr
    }
}
```


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 $\mathrm{Term}(\Sigma)$ 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>
```


Benutzung Algebr. Datentypen

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 $\to$ 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, . . .

Rekursionsmuster

Verhaltensmuster: Besucher

Prinzip:

- rekursive Datenstruktur (algebraischer Datentyp, Kompositum)

⇒ 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 x
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 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,Inte
      public Integer branch
           (Integer left, Integer right) {
        return left + right;
      }
      public Integer leaf(K key) {
        return 1;
      }
    });
  }
```

```
}
```


Datenströme (Iteratoren)

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 ) $ natural
```

C#:

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

- logische Trennung:
  Produzent → Transformator(en) → 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, in
```

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`

Dann auch für endliche Ströme, Test:

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

Dann Mergesort

```
   static IEnumerable<E> Sort<E> (IEnumerabl
        where E : IComparable<E> {
        if (xs.Count() <= 1) {
            return xs;
        } else { // zwei Zeilen folgen
            ...
        }
    }
```

Test: `Sort(new int [] { 3,1,4,1,5,9})`

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, Strategie, Interpreter

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,
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-O
    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

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
     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

Zustand, DI, Beobachter, MVC

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).
⇒ 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
       if (((Counter)o).getCount() >= this.t
           System.out.println ("alarm");  }
public static void main(String[] args) {
    Counter c = new Counter (); Watcher w =
    c.addObserver(w); c.step(); c.step (); c
```


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-`
`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 (← getCount)
- Controller: JButton (→ 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
JSlider bot = new JSlider(JSlider.HORIZONTAL
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 TreeSelection
   public void valueChanged(TreeSelectionEve

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


Quelltextverwaltung

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 → sandbox
- check-in: sandbox → 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 $G \to D$ *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 $x \le y$ und $x \le z$,
wobei die Relation $\le$ auf Σ^* so definiert ist:
$u \le v$, 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: $u \le v$, falls u aus v durch Löschen von Buchstaben

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

Testfragen:

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

Die Einbettungs-Relation (II)

Begriffe (für Halbordnungen):

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

Sätze: für $\le$ ist

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

Beispiel: bestimme die Menge der $\le$-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)

```
-- | Länge einer längsten gemeinsamen Teilfo
lcs :: Eq a => [a] -> [a] -> Int
lcs [] ys = 0
lcs xs [] = 0
lcs (x : xs) (y : ys) =
  maximum
    [ if x == y then 1 + lcs xs ys
                else     lcs xs ys
    , lcs      xs  (y : ys)
    , lcs (x : xs)      ys
    ]
```


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"));
  }
}
```

Aufgaben:

- vervollständigen Sie die Methode `LCS.lcs`
- bauen Sie eine Möglichkeit ein, nicht nur die Länge einer längsten gemeinsamen Teilfolge zu bestimmen, sondern auch eine solche Folge selbst auszugeben.

  Hinweis: `int [][] a` wie oben ausrechnen und *danach* vom Ende zum Anfang durchlaufen (ohne groß zu suchen).

  damit dann die autotool-Aufgaben lösen.

Produktqualität (I)

Klassifikation der Verfahren

- Verifizieren (= Korrektheit beweisen)
  - Verifizieren
  - symbolisches Ausführen
- Testen (= Fehler erkennen)
  - statisch (z. B. Inspektion)
  - dynamisch (Programm-Ausführung)
- Analysieren (= Eigenschaften vermessen/darstellen)
  - Quelltextzeilen (gesamt, pro Methode, pro Klasse)
  - Klassen (Anzahl, Kopplung)
  - Profiling (... später mehr dazu)

Fehlermeldungen

sollen enthalten

- Systemvoraussetzungen
- Arbeitsschritte
- beobachtetes Verhalten
- erwartetes Verhalten

Verwaltung z. B. mit Bugzilla, Trac
Vgl. Seminarvortrag D. Ehricht:
`http://www.imn.htwk-leipzig.de/~waldmann/edu/ss04/se/ehricht/bugzilla.pdf`

Testen und Schnittstellen

- Test für Gesamtsystem (schließlich) oder Teile (vorher)
- Teile definiert durch Schnittstellen
- Schnittstelle ⇒ Spezifikation
- Spezifikation ⇒ Testfälle

Testen ...

- unterhalb einer Schnittstelle (unit test)
- oberhalb (mock objects) (vgl. dependency injection)

  vgl. `http://www.mockobjects.com/`

Dynamische Tests

- Testfall: Satz von Testdaten
- Testtreiber zur Ablaufsteuerung
- ggf. *instrumentiertes* Programm zur Protokollierung

Beispiele (f. Instrumentierung):

- Debugger: fügt Informationen über Zeilennummern in Objektcode ein

  `gcc -g foo.c -o foo  ;  gdb foo`
- Profiler: Code-Ausführung wird regelmäßig unterbrochen und „aktuelle Zeile“ notiert, anschließend Statistik

Programmablauf-Tests

bezieht sich auf Programm-Ablauf-Graphen (Knoten: Anweisungen, Kanten: mögliche Übergänge)

- Anweisungs-Überdeckung: jede Anweisung mindestens einmal ausgeführt
- Zweigüberdeckung: jede Kante mindestens einmal durchlaufen — Beachte: `if (X) then { A }`
- Pfadüberdeckung: jeder Weg (Kantenfolge) mindestens einmal durchlaufen — Beachte: Schleifen (haben viele Durchlaufwege)
  Variante: jede Schleife (interior) höchstens einmal
- Bedingungs-Überdeckung: jede atomare Bedingung einmal true, einmal false.

Prüfen von Testabdeckungen

mit Werkzeugunterstützung, Bsp.: *Profiler*:
mißt bei Ausführung Anzahl der Ausführungen ...

- ... jeder Anweisung (Zeile!)
- ... jeder Verzweigung (then oder else)

(genügt für welche Abdeckungen?)

Profiling durch Instrumentieren (Anreichern)

- des Quelltextes
- oder der virtuellen Maschine

Übung Profiling (C++)

Beispiel-Programm(e):
`http://www.imn.htwk-leipzig.de/~waldmann/edu/ss04/case/programme/analyze/cee/`
Aufgaben:

- Kompilieren und ausführen für Profiling:

```
g++ -pg -fprofile-arcs heap.cc -o heap
./heap > /dev/null
# welche Dateien wurden erzeugt? (ls -lrt)
gprof heap # Analyse
```

- Kompilieren und ausführen für Überdeckungsmessung:

```
g++ -ftest-coverage -fprofile-arcs heap.cc
```



```
./heap > /dev/null
# welche Dateien wurden erzeugt? (ls -lrt)
gcov heap.cc
# welche Dateien wurden erzeugt? (ls -lrt)
```

Optionen für `gcov` ausprobieren! (`-b`)

- `heap` reparieren: `check` an geeigneten Stellen aufrufen, um Fehler einzugrenzen
- `median3` analysieren: Testfälle schreiben (hinzufügen) für: Anweisungsüberdeckung, Bedingungsüberdeckung, Pfadüberdeckung

  Überdeckungseigenschaften mit `gcov` prüfen
- `median5` reparieren

Profiling (Java)

- Kommandozeile: `java -Xprof ...`
- in Eclipse: benutzt TPTP `http://www.eclipse.org/articles/Article-TPTP-Profiling-Tool/tptpProfilingArticle.html` `http://www.eclipse.org/tptp/home/documents/tutorials/profilingtool/profilingexample_32.html`
- Installation: Eclipse → Help → Update ...
- im Pool vorbereitet, benötigt aber genau diese Eclipse-Installation und java-1.5

```
export PATH=/home/waldmann/built/bin:$PATH
```



```
unset LD_LIBRARY_PATH
/home/waldmann/built/eclipse-3.2.2/eclipse
```

(für JDK-1.6: TPTP-4.4 in Eclipse-3.3 (Europa))

Dynamische Tests: Black/White

- Strukturtests (white box)
  - programmablauf-orientiert
  - datenfluß-orientiert
- Funktionale Tests (black box)
- Mischformen (unit test)

Black-Box-Tests

ohne Programmstruktur zu berücksichtigen.

- typische Eingaben (Normalbetrieb)
  alle wesentlichen (Anwendungs-)Fälle abdecken
  (Bsp: gerade und ungerade Länge einer Liste bei mergesort)
- extreme Eingaben
  sehr große, sehr kleine, fehlerhafte
- zufällige Eingaben
  durch geeigneten Generator erzeugt

während Produktentwicklung:
Testmenge ständig erweitern,
frühere Tests immer wiederholen (regression testing)

Probleme mit GUI-Tests

schwierig sind Tests, die sich nicht automatisieren lassen (GUIs: Eingaben mit Maus, Ausgaben als Grafik)
zur Unterstützung sollte jede Komponente neben der GUI-Schnittstelle bieten:

- auch eine API-Schnittstelle (für (Test)programme)
- und ein Text-Interface (Kommando-Interpreter)

Bsp: Emacs: `M-x kill-rectangle` oder `C-x R K`, usw.

Mischformen

- Testfälle für jedes Teilprodukt, z. B. jede Methode
  (d. h. Teile der Programmstruktur werden berücksichtigt)
- Durchführung kann automatisiert werden (`JUnit`)

Testen mit JUnit

Kent Beck and Erich Gamma,
`http://junit.org/index.htm`

```
import static org.junit.Assert.*;
class XTest {

   @Test
    public void testGetFoo() {
        Top f = new Top ();
        assertEquals (1, f.getFoo());
    }
}
```

`http://www-128.ibm.com/developerworks/java/library/j-junit4.html`

JUnit ist in Eclipse-IDE integriert (New → JUnit Test Case → 4.0)

JUnit und Extreme Programming

Kent Beck empfiehlt *test driven apporach*:

- *erst* alle Test-Methoden schreiben,
- *dann* eigentliche Methoden implementieren
- …bis sie die Tests bestehen (und nicht weiter!)
- Produkt-Eigenschaften, die sich nicht testen lassen, *sind nicht vorhanden*.
- zu jedem neuen Bugreport einen neuen Testfall anlegen

Testfall schreiben ist *Spezifizieren*, das geht *immer* dem Implementieren voraus. — *Testen* der Implementierung ist nur die zweitbeste Lösung (besser ist *Verifizieren*).

Delta Debugging

Andreas Zeller: *From automated Testing to Automated Debugging*, automatische Konstruktion von

- minimalen Bugreports
- Fehlerursachen (bei großen Patches)

Modell (Intervallverkleinerung, vgl. binäre Suche)

- `test : Set<Patch> -> { OK, FAIL, UNKNOWN }`
- `dd(low,high,n)=(x,y)`
  - Vorbedingung `low` $\subseteq$ `high`,
    `test(low)=OK, test(high)=FAIL`
  - Nachbedingung `low` $\subseteq$ `x` $\subseteq$ `y` $\subseteq$ `high`,
    `test(x)=OK, test (y)=FAIL,`
    `size(y - x)` „möglichst klein“

Delta Debugging (II)

```
dd(low, high, n) =
    let diff = size(high) - size(low)
        c_1, .. c_n = Partition von (high - low)
    if exists i : test (low + c_i) == FAIL
        then dd(                  )
    else if exists i : test (high - c_i) == OK
        then dd(                  )
    else if exists i : test (low + c_i) == OK
        then dd(                          )
    else if exists i : test (high - c_i) == FAIL
        then dd(                          )
    else if n < diff
        then dd(          ) else (low, high)
```

`http://www.infosun.fim.uni-passau.de/st/papers/computer2000/`

Refactoring

Herkunft

Kent Beck: *Extreme Programming*, Addison-Wesley 2000:

- Paar-Programmierung (zwei Leute, ein Rechner)
- test driven: erst Test schreiben, dann Programm implementieren
- Design nicht fixiert, sondern flexibel

Refaktoring: Definition

Martin Fowler: *Refactoring: Improving the Design of Existing Code*, A.-W. 1999,
http://www.refactoring.com/
Def: Software so ändern, daß sich

- externes Verhalten nicht ändert,
- interne Struktur verbessert.

siehe auch William C. Wake: *Refactoring Workbook*, A.-W. 2004 http://www.xp123.com/rwb/
und Stefan Buchholz: Refactoring (Seminarvortrag)
http://www.imn.htwk-leipzig.de/~waldmann/edu/ss05/se/talk/sbuchhol/

Refactoring anwenden

- mancher Code „riecht"(schlecht)

  (Liste von *smells*)
- er (oder anderer) muß geändert werden

  (Liste von *refactorings*, Werkzeugunterstützung)
- Änderungen (vorher!) durch Tests absichern

  (JUnit)

Refaktorisierungen

- Abstraktionen einführen:

  neue Schnittstelle, Klasse (Entwurfsmuster!)

  Methode, (temp.) Variable
- Abstraktionen ändern:

  Attribut/Methode bewegen (in andere Klasse)

Code Smell # 1: Duplicated Code

jede Idee sollte an *genau einer* Stelle im Code formuliert werden:
Code wird dadurch

- leichter verständlich
- leichter änderbar

Verdoppelter Quelltext (copy–paste) führt immer zu Wartungsproblemen.

Duplicated Code → Schablonen

duplizierter Code wird verhindert/entfernt durch

- *Schablonen* (beschreiben das Gemeinsame)
- mit *Parametern* (beschreiben die Unterschiede).

Beispiel dafür:

- Unterprogramm (Parameter: Daten, Resultat: Programm)
- polymorphe Klasse (Parameter: Typen, Resultat: Typ)
- Unterprogramm höherer Ordnung (Parameter: Programm, Resultat: Programm)

wenn Programme als Parameter nicht erlaubt sind (Java), dann werden sie als Methoden von Objekten versteckt (vgl. Entwurfsmuster Besucher)

Size does matter

weitere Code smells:

- lange Methode
- große Klasse
- lange Parameterliste

oft verursacht durch anderen Smell: Primitive Obession

Primitive Daten (*primitive obsession*)

Symptome: Benutzung von int, float, String ...
Ursachen:

- fehlende Klasse:

  z. B. String → FilePath, Email, URI ...
- schlecht implementiertes Fliegengewicht

  z. B. int i bedeutet x[i]
- simulierter Attributname:

  z. B. Map<String,String> m; m.get("foo");

Behebung: Klassen benutzen, Array durch Objekt ersetzen
(z. B. class M { String foo; ...})

Typsichere Aufzählungen

Definition (einfach)

```
public enum Figur { Bauer, Turm, König }
```

Definition mit Attribut (aus JLS)

```
public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(2
    Coin(int value) { this.value = value; }
    private final int value;
    public int value() { return value; }
}
```

Definition mit Methode:

```
public enum Figur {
  Bauer { int wert () { return    1; } },
  Turm  { int wert () { return    5; } },
```

```
  König { int wert () { return 1000; } };
  abstract int wert ();
}
```
Benutzung:
```
Figur f = Figur.Bauer;
Figur g = Figur.valueOf("Turm");
for (Figur h : Figur.values()) {
   System.out.println (h + ":" + h.wert());
}
```


Verwendung von Daten: Datenklumpen

Fehler: Klumpen von Daten wird immer gemeinsam benutzt
```
String infile_base; String infile_ext;
String outfile_base; String outfile_ext;

static boolean is_writable (String base, Str
```
Indikator: ähnliche, schematische Attributnamen
Lösung: Klasse definieren
```
class File { String base; String extension;

static boolean is_writable (File f);
```


Datenklumpen—Beispiel

Beispiel für Datenklumpen und -Vermeidung:
```
java.awt

Rectangle(int x, int y, int width, int heigh
Rectangle(Point p, Dimension d)
```
Vergleichen Sie die Lesbarkeit/Sicherheit von:
```
new Rectangle (20, 40, 50, 10);
new Rectangle ( new Point (20, 40)
              , new Dimension (50, 10) );
```
Vergleichen Sie:
```
java.awt.Graphics: drawRectangle(int,int,int
java.awt.Graphics2D: draw (Shape);
    class Rectangle implements Shape;
```


Aufgabe Refactoring

Würfelspiel-Simulation:
Schummelmex: zwei (mehrere) Spieler, ein Würfelbecher
Spielzug ist: aufdecken oder (verdeckt würfeln, ansehen, ansagen, weitergeben) bei Aufdecken wird vorige Ansage mit vorigem Wurf verglichen, das ergibt Verlustpunkt für den Aufdecker oder den Aufgedeckten

- Vor Refactoring:
  `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss07/case/programs/refactor/stage0/`
  Welche Code-Smells?

- Nach erstem Refactoring:
  `http://www.imn.htwk-leipzig.de/~waldmann/`

  `edu/ss07/case/programs/refactor/stage1/`
  Was wurde verbessert? Welche Smells verbleiben?

- Nach zweitem Refactoring:
  `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss07/case/programs/refactor/stage2/`
  Was wurde verbessert? Welche Smells verbleiben?

Verwendung von Daten: Data Class

Fehler:
Klasse mit Attributen, aber ohne Methoden.
```
class File { String base; String ext; }
```
Lösung:
finde typische Verwendung der Attribute in Client-Klassen, (Bsp: `f.base + "/" + f.ext`)
schreibe entsprechende Methode, verstecke Attribute (und deren Setter/Getter)
```
class File {  ...
    String toString () { ... }
}
```


Temporäre Attribute

Symptom: viele `if (null == foo)`
Ursache: Attribut hat nur während bestimmter Programmteile einen sinnvollen Wert
Abhilfe: das ist kein Attribut, sondern eine temporäre Variable.

Nichtssagende Namen

(Name drückt Absicht nicht aus)
Symptome:

- besteht aus nur einem oder zwei Zeichen
- enthält keine Vokale
- numerierte Namen (`panel1, panel2, \dots`)
- unübliche Abkürzungen
- irreführende Namen

Behebung: umbenennen, so daß Absicht deutlicher wird.
(Dazu muß diese dem Programmierer selbst klar sein!)
Werkzeugunterstützung!

Name enthält Typ

Symptome:

- Methodenname bezeichnet Typ des Arguments oder Resultats

  `class Library { addBook( Book b ); }`

- Attribut- oder Variablenname bezeichnet Typ (sog. Ungarische Notation) z. B. `char ** ppcFoo`

  siehe `http://ootips.org/hungarian-notation.html`

- (grundsätzlich) Name bezeichnet Implementierung statt Bedeutung

Behebung: umbenennen (wie vorige Folie)

Programmtext

- Kommentare

  → *don't comment bad code, rewrite it*

- komplizierte Boolesche Ausdrücke

  → umschreiben mit Verzweigungen, sinnvoll bezeichneten Hilfsvariablen

- Konstanten (*magic numbers*)

  → Namen für Konstanten, Zeichenketten externalisieren (I18N)

Größe und Komplexität

- Methode enthält zuviele Anweisungen (Zeilen)
- Klasse enthält zuviele Attribute
- Klasse enthält zuviele Methoden

Aufgabe: welche Refaktorisierungen?

Mehrfachverzweigungen

Symptom: `switch` wird verwendet

```
class C {
    int tag; int FOO = 0;
    void foo () {
        switch (this.tag) {
            case FOO: { .. }
            case 3:   { .. }
}   }   }
```

Ursache: Objekte der Klasse sind nicht ähnlich genug
Abhilfe: Kompositum-Muster

```
interface C { void foo (); }
class Foo implements C { void foo () { .. }
class Bar implements C { void foo () { .. }
```


null-Objekte

Symptom: `null` (in Java) bzw. 0 (in C++) bezeichnet ein besonderes Objekt einer Klasse, z. B. den leeren Baum oder die leere Zeichenkette
Ursache: man wollte Platz sparen oder „Kompositum" vermeiden.
Nachteil: `null` bzw. `*0` haben keine Methoden.
Abhilfe: ein extra Null-Objekt deklarieren, das wirklich zu der Klasse gehört.

Richtig refaktorisieren

- immer erst die Tests schreiben
- Code kritisch lesen (eigenen, fremden), eine Nase für Anrüchigkeiten entwickeln (und für perfekten Code).
- jede Faktorisierung hat ein Inverses.

  (neue Methode deklarieren ↔ Methode inline expandieren)

  entscheiden, welche Richtung stimmt!

- Werkzeug-Unterstützung erlernen

Aufgaben zu Refaktoring (I)

- Code Smell Cheat Sheet (Joshua Keriewsky): `http://industriallogic.com/papers/smellstorefactorings.pdf`
- Smell-Beispiele `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss05/case/rwb/` (aus Refactoring Workbook von William C. Wake `http://www.xp123.com/rwb/`)

  ch6-properties, ch6-template, ch14-ttt

Aufgaben zu Refaktoring (II)

Refactoring-Unterstützung in Eclipse:

```
package simple;

public class Cube {
    static void main (String [] argv) {
        System.out.println (3.0 + " " + 6 *
        System.out.println (5.5 + " " + 6 *
    }
}
```

extract local variable, extract method, add parameter, . . .

Aufgaben zu Refaktoring (II)

- Eclipse → Refactor → Extract Interface
- "Create Factory"
- Finde Beispiel für "Use Supertype"

Class Design

Klassen-Entwurf

- benutze Klassen! (sonst: primitive obsession)
- ordne Attribute und Methoden richtig zu

  (Refactoring: move method, usw.)
- dokumentiere Invarianten für Objekte, Kontrakte für Methoden
- stelle Beziehungen zwischen Klassen durch Interfaces dar

  (. . . Entwurfsmuster)

Mehrfachverzweigungen

Symptom: `switch` wird verwendet

```
class C {
    int tag; int FOO = 0;
    void foo () {
        switch (this.tag) {
            case FOO: { .. }
            case 3:   { .. }
}   }   }
```

Ursache: Objekte der Klasse sind nicht ähnlich genug

Abhilfe: Kompositum-Muster

```
interface C { void foo (); }
class Foo implements C { void foo () { .. }
class Bar implements C { void foo () { .. }
```


null-Objekte

Symptom: `null` (in Java) bzw. 0 (in C++) bezeichnet ein besonderes Objekt einer Klasse, z. B. den leeren Baum oder die leere Zeichenkette

Ursache: man wollte Platz sparen oder „Kompositum" vermeiden.

Nachteil: `null` bzw. `*0` haben keine Methoden.

Abhilfe: ein extra Null-Objekt deklarieren, das wirklich zu der Klasse gehört.

Das Fabrik-Muster

```
interface I {}
class C implements I {}
class D implements I {}
```

Bei Konstruktion von I-Objekten muß ein konkreter Klassenname benutzt werden.

Wie schaltet man zwischen C- und D-Erzeugung um?

Benutze Fabrik-Objekt: Implementierung von

```
interface F { I create () }
```


Immutability

(Joshua Bloch: Effective Java, Addison Wesley, 2001)

immutable = unveränderlich

Beispiele: String, Integer, BigInteger

- keine Set-Methoden
- keine überschreibbaren Methoden
- alle Attribute privat
- alle Attribute final

leichter zu entwerfen, zu implementieren, zu benutzen.

Immutability

- immutable Objekte können mehrfach benutzt werden (sharing).

  (statt Konstruktor: statische Fabrikmethode. Suche Beispiele in Java-Bibliothek)
- auch die Attribute der immutable Objekte können nachgenutzt werden (keine Kopie nötig)

  (Beispiel: negate für BigInteger)
- immutable Objekte sind sehr gute Attribute anderer Objekte:

  weil sie sich nicht ändern, kann man die Invariante des Objektes leicht garantieren

Vererbung bricht Kapselung

(Implementierungs-Vererbung: schlecht,
Schnittstellen-Vererbung: gut.)

Problem: `class B extends A` ⇒
B hängt ab von Implementations-Details von A.

⇒ wenn Implementierung von A unbekannt, dann korrekte Implementierung von B nicht möglich.

(Beispiel)
⇒ Wenn man Implementierung von A ändert, kann B kaputtgehen.

Vererbung bricht Kapselung

Joshua Bloch (Effective Java):

- design and document for in heritance
- . . . or else prohibit it

API-Beschreibung muß Teile der Implementierung dokumentieren (welche Methoden rufen sich gegenseitig auf), damit man diese sicher überschreiben kann.
(Das ist ganz furchtbar.)

statt Vererbung: benutze Komposition (Wrapper) und dann Delegation.

Testfragen

Hinweis

Das sind prüfungsähnliche Fragen aus vorigen Jahren.
Nicht alle Themen wurden jedes Jahr behandelt.

Testfragen

3 (6 Punkte) Geben Sie für die nebenstehenden Aktivitätsdiagramme jeweils die Menge aller möglichen Aktivitätsfolgen an.

Geben Sie jeweils ein Diagramm D an (das möglicherweise aus mehreren Aktivitäten besteht), für das die beiden nebenstehenden Diagramme
a) äquivalent sind, b) nicht äquivalent sind.
Begründen Sie Ihre Aussage durch Angabe der Mengen der möglichen Aktivitätsfolgen.

2 (4 Punkte) Welches dieser Diagramme gibt die Assoziation zwischen Studenten und Sitzplätzen (in diesem Raum) korrekt wieder? Begründen Sie, warum das andere falsch ist.

a) [Student] $0..1$ —— 1 [Sitzplatz]   b)

[Student] $1..*$ —— $1..*$ [Sitzplatz]

Modellieren Sie in einem Klassendiagramm diesen Zusammenhang zwischen Klassen A, B, C: jedes A-Objekt ist entweder assoziiert mit genau einem B-Objekt oder assoziiert mit genau einem C-Objekt. Benutzen Sie dazu eine zusätzliche (abstrakte) Klasse S, die in passender Beziehung zu A, B, C steht.
Welche Bestandteile gehören zum Entwurfmuster *Beobachter*, welche Methoden haben diese?

Skizzieren Sie eine typische Benutzung dieses Musters

(Reihenfolge der Methodenaufrufe).

Die Methode `setChanged()` beeinflußt die Wirkung einer anderen Methode. Welcher? Wie? Warum (zwei Gründe)?

(*Bearbeiten Sie zunächst die Aufgabe auf Seite 3, dann diese hier!*) Wie bestimmt man mit Hilfe eines Besuchers die Summe aller Schlüsselwerte eines Baumes?
Welche Bestandteile hat eine sinnvolle Fehlermeldung, wozu dienen diese jeweils (was passiert, wenn sie fehlen)?

-
-
-
-

Welche Beziehungen sind zwischen den Methoden einer Klasse wünschenswert?
Wie kann man abmessen oder ausrechnen, ob das zutrifft?

Wie kann man ein Programm ändern, um diese Beziehungen herzustellen oder zu verbessern?
Zeigen Sie an einem Beispiel, daß die von Ihnen vorgeschlagene Änderung tatsächlich die von Ihnen genannten Kennzahlen verbessert.
Die folgende Datenstruktur *Baum* mit Typparameter K soll durch das Entwurfsmuster *Kompositum* abgebildet werden.

- Verzweigungsknoten haben genau zwei Kinder, die

Bäume mit Schlüsseltyp K sind,

- Blattknoten enthalten einen Schlüssel vom Typ K und haben keine Kinder.

Geben Sie die dazu nötigen Interfaces und Klassen sowie ihre Beziehungen an.
Nennen Sie die Aufgaben eines Quelltextverwaltungssystemes (wie CVS oder SVN).

Was sind die wesentlichen Aufgaben eines Quelltextverwaltungssystems wie z. B. CVS?

In CVS wird zwischen Binär- und Textdateien unterschieden. Welche Leistungen sind nur für Textdateien verfügbar? Warum ist das so? Warum fällt die fehlende Unterstützung von Binärdateien selten ins Gewicht?
Betrachten Sie folgende Klasse:

```
public class Zeichenobjekt {
    int startx, starty, endx, endy; int brei
}
```

Wir bemerken die Code Smells *primitive obsession* und *Datenklumpen*.
Geben Sie jeweils die Symptome an sowie mögliche Auswirkungen und einen Verbesserungsvorschlag

primitive Obsession:

Datenklumpen:

Definieren Sie den Begriff *Entwurfsmuster*.

Eine Klasse enthält Funktionen, die Summe und Produkt eines Arrays von Zahlen bestimmen.

```
static int sum (int [] a) {
  int result = 0;
  for (int i = 0; i<a.length; i++) { result
```



```
  return result;
}
static int product (int [] a) {
  int result = 1;
  for (int i = 0; i<a.length; i++) { result
  return result;
}
```

Sie sollen diesen Code durch Benutzung des Strategie-Musters verbessern. Ergänzen Sie:

```
interface Folder {
  ...


}
```



```
static int fold (Folder f, int [] a) {
  int result = ...
  for (int i = 0; i<a.length; i++) {
    result = ...
  }
  return result;
}
class Sum_Folder ...                {
  ...


}
static int sum (int [] a) { return fold ( ..
class Product_Folder ...              {
```



```
  ...


}
static int product (int [] a) { return fold
```

Definieren Sie die Code-Metrik nach Halstead. (Geben Sie die Formel an und definieren Sie die darin benutzen Bezeichner o, a, A.)

Man möchte den Metrik-Wert für ein Programm dadurch verbessern, daß man Änderungen ausführt, die sich nur auf den Parameter a auswirken.

Geben Sie dafür ein Beispiel an. In welche Richtung ändert sich a, in welche Richtung ändert sich die Metrik?

Die Halstead-Metrik ist nur auf Anweisungsfolgen (z. B. Implementierungen einzelner Methoden) anwendbar.
Durch welches (objektorientierte) Refactoring kann man Metrik-Werte verbessern? (Welcher Parameter wird dabei wie geändert?)
Für die eben definieren Baumstruktur soll das Entwurfsmuster *Iterator* angeboten werden.

Geben Sie die dazu nötigen Interfaces/Klassen und ihre Beziehungen an. (Die Methoden sollen vollständig deklariert, aber nicht implementiert werden.)
Beschreiben Sie, wie man mit Hilfe eines Iterators die Summe aller Schlüsselwerte eines Baumes bestimmt.
1 (6 Punkte) Ein Softwareprojekt hat einen Umfang von 140 Personen-Stunden. 7 Entwickler arbeiten an diesem Projekt. Jeder Entwickler arbeitet 8 Stunden pro Tag. Die Entwickler sollen ihre Arbeit täglich einmal abstimmen. Jeder Kommunikationsvorgang (zwischen zwei Entwicklern) dauert 1 Stunde.
Wieviele Tage benötigt das Projekt unter diesen Szenarien:

1. Das Projekt ist nicht strukturiert (jeder Entwickler muß mit jedem kommunizieren).

2. Das Projekt ist streng hierarchisch strukturiert (als vollständiger binärer Baum: 1 großer Chef, 2 kleine Chefs, 4 Entwickler. Jeder kommuniziert nur mit seinem direkten Vorgesetzten, die Chefs entwickeln nichts)

Welche allgemeinen Folgerungen kann man für das Projektmanagement ableiten?
Inwieweit ist es zutreffend oder gerechtfertigt, daß die Chefs „nichts tun“?
Was ist der *Vertrag* eines Unterprogramms (einer Methode)? Zwischen welchen Parteien besteht das

Vertragsverhältnis?

Was ist die *Invariante* einer Klasse?

Wodurch kann man es erreichen (oder erleichtern), daß die Klasseninvariante tatsächlich erhalten bleibt?

Wir bezeichnen die Teilfolge-Relation mit $\leq$, Beispiel: $abc \leq aabbcc$; und die Längenfunktion mit $|\cdot|$, Beispiel: $|abc| = 3$.

- Spezifizieren Sie $\operatorname{lcs}(u, v)$ (= die Länge einer längsten

gemeinsamen Teilfolge)

- Bestimmen Sie $\operatorname{lcs}(babbab, ababbba)$
- Wozu werden in Quelltextverwaltungssystemem längste gemeinsame Teilfolgen bestimmt?

- (*Zusatzaufgabe*) Wir bezeichnen mit $C(w)$ das bitweise Komplement eines Wortes über $\{0, 1\}$, Beispiel $C(0010) = 1101$.

Bestimmen Sie ein Wort w über $\{0, 1\}$ der Länge 100, für das $\operatorname{lcs}(w, C(w))$ möglichst groß ist.

Definieren Sie die Kohäsionsmetrik nach Henderson-Sellers und erläutern Sie die Bedeutung der dabei vorkommenden Parameter.

Welches sind der kleinste und der größte mögliche Wert dieser Metrik?
Welcher ist anzustreben bzw. zu vermeiden und warum?
Skizzieren Sie jeweils ein Beispielprogramm, das diesen

Wert erreicht.

- kleinstmöglicher Wert

- größtmöglicher Wert

Was ist der *Vertrag* eines Unterprogramms?

Was ist die Invariante einer Klasse?

Welche Nutzen hat die Verwendung solcher Verträge in der

Software-Entwicklung?

Wie kann man sich von der Einhaltung solcher Verträge überzeugen? (Zwei Möglichkeiten — vergleichen Sie beide!)

Wieviele Testfälle werden mindestens für eine Pfadüberdeckung benötigt? (Gefragt ist nur nach der Anzahl.)
Finden Sie für das Programm P jeweils eine möglichst

kleine Menge von Testfällen für eine

- Anweisungsüberdeckung,

- Bedingungsüberdeckung.

Bei der Planung eines Softwareprojektes sollte man Meilensteine vorsehen. Nennen Sie drei wichtige Forderungen an Meilensteine.
Vervollständigen Sie: Ein Netzplan ist ein Graph mit

- Knotenmenge:

- Kantenmenge:

Definieren Sie die Begriffe

- freie Pufferzeit

- gesamte Pufferzeit

- kritischer Pfad

Warum sind die kritischen Pfade so benannt?

Was ist *primitive obsession*, welche negativen Auswirkungen hat sie, was kann man dagegen tun?
Woran erkennt man den code smell *primitive obsession*?
Wie kann man ihn beheben?
Wieso erhöht sich dadurch die statische Sicherheit des Programmes?

Im nebenstehenden Programm P sind a, b, c, d Variablen für ganze Zahlen und swap vertauscht die Inhalte der Argumente.

```
if (a < b) { s
if (c < d) { s
if (b < c) { s
```

5 (4 Punkte) Stellen Sie sich vor, ein Web-Browser (ähnlich Mozilla usw.) wäre neu zu entwickeln. Geben Sie je ein Beispiel für den Funktionsumfang eines

- horizontalen,

- vertikalen

Prototypen. Welche (Teile von) Anwendungsfällen können mit Ihren Prototypen durchgeführt werden?
Zur Qualitätssicherung werden Teilprodukte der Software-Entwicklung

- validiert

- verifiziert

Definieren Sie die beiden Begriffe.
Geben Sie zwei Beispiele für die Qualitätssicherung von Teilprodukten an, die selbst keine Programme sind.
Was ist Refactoring?
Welcher Zusammenhang besteht zwischen Testen und Refactoring? Was ist Refactoring?

Vervollständigen Sie diese Tabelle. Fügen Sie noch einen Code Smell Ihrer Wahl hinzu.

Code Smell	Auswirkung	Reparatur
duplizierter Code		
primitive obsession		
Kommentar		

Welche Code Smells stellen Sie im gegebenen Quelltext fest? Nennen Sie jeweils den *Namen*, die *Symptome*, die *Auswirkungen* (warum ist der Smell gefährlich?) und schlagen Sie eine *Verbesserung* vor.
Teilprodukte müssen während der Entwicklung verwaltet werden. Welchen Status kann ein Teilprodukt haben? Ergänzen und definieren Sie:

- geplant

- in Bearbeitung

- ??

- ??

Welche Status-Änderungen sind außer dem Normalablauf

(von oben nach unten) möglich?

-

-

Eine Klasse beginnt so:

```
class Zeichenobjekt { ...
    int typ;    ...
    static final int LINIE=1;
    static final int RECHTECK=2;
    static final int ELLIPSE=3;    ...
}
```

Welcher Code Smell wird auftreten, wenn man Objekte

dieser Klasse benutzt?

Was sollte stattdessen verwendet werden? Skizzieren Sie eine bessere Lösung.

Warum ist der Code dann besser erweiterbar? Definieren Sie die Begriffe

- Spezifizieren:

- Verifizieren:

- Testen:

Erklären Sie den Unterschied zwischen Testen und Verifizieren.
Was ist *test driven development*?
Was ist Refactoring?
Welcher Zusammenhang besteht zwischen Testen und Refactoring?
Welcher Zusammenhang besteht zwischen Entwurfsmustern und Refactoring?
7 (4 Punkte) Für $x \in \{a, b, c, d\}$ und $n \in \{1, 2, 3, 4\}$ bezeichnen wir mit x_n den Wert der Variablen x direkt *nach* Ausführung der Zeile n des Programms P, sowie mit x_0 den Wert *vor* der *ersten* Zeile.
Es gilt immer: $a_2 \geq b_2 \wedge c_2 \geq d_2$. Begründen Sie:

$a_3 \geq b_3 \wedge a_3 \geq c_3 \wedge a_3 \geq d_3$.
Lösungsweg: Unterscheiden Sie zwei Fälle. Drücken Sie für jeden Fall die Zusammenhänge zwischen a_2, b_2, c_2, d_2 und a_3, b_3, c_3, d_3 durch Gleichungen und Ungleichungen aus. (Beispiel: $d_2 = d_3$.)
Gilt immer $a_4 \geq b_4 \geq c_4 \geq d_4$? (Beweis oder Gegenbeispiel.)
Für die Baumstruktur soll zusätzlich das Entwurfsmuster *Besucher* implementiert
werden, wobei der Resultattyp R des Besuchers ein weitere Typparameter ist.
Benutzen Sie dieses Muster zum Bestimmen des größten Schlüssels eines Baumes.
Bei der Benutzung von Quelltextarchiven (z. B. CVS) kann es zu Konflikten kommen. Woran und wann (bei welcher

Aktion) erkennt wer (Server oder Client), daß ein solcher Konflikt vorliegt?

Welche Arbeitsschritte sind (von wem?) zur Lösung eines solchen Konfliktes auszuführen?

In welchen Fällen läßt sich das automatisieren? Welche

Rechnung stellt wer (Server/Client) dabei an?

Welche Rolle spielt dabei das Bestimmen einer längsten gemeinsamen Teilfolge?
Warum erschwert der Code Smell „lange Methode“ den Unit-Test?

```
public class Zeichenobjekt {
    // Merkmale eines Zeichenobjektes
    int startx, starty, endx, endy;
    int breite, hoehe; int typ;    ...
    static final int LINIE=1; static final int RECHTE
    static final int ELLIPSE=3;    ...
}
```



```
public class Zeichenflaeche {
  public Vector objekte = new Vector();
  public Zeichenobjekt zo; ...
  public void paint(Graphics g) {
      Graphics2D g2 = (Graphics2D) g;
      for (int i = 0; i < objekte.size(); i++) {
          zo = (Zeichenobjekt) objekte.elementAt(i);
          if (zo.fuellen == false) {
              switch (zo.getTyp()) {
              case Zeichenobjekt.RECHTECK:
                  int xr1 = zo.getStartx();
                  int xr2 = zo.getEndx();
                  int yr1 = zo.getStarty();
                  int yr2 = zo.getEndey();
                  if (xr1 < xr2) {
                      if (yr1 < yr2) {
```



```
                          g2.setColor(zo.getC());
                          //von links oben nach recht
                          g2.draw(new Rectangle2D.Flo
                                  (xr1, yr1, xr2 - xr
                      } else {
                          g2.setColor(zo.getC());
                          //von links unten nach rech
                          g2.draw(new Rectangle2D.Flo
                                  (xr1, yr2, xr2 - xr
                      }
                  } else ...
              case Zeichenobjekt.ELLIPSE: ...
```