Softwaretechnik II
Vorlesung
Sommersemester 2009

Johannes Waldmann, HTWK Leipzig

15. Juni 2009

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

- Programmieren im Kleinen, Werkzeuge (Eclipse)
- Programmieren im Team, Werkzeuge (CVS, Bugzilla, Trac)
- Spezifizieren, Verifizieren, Testen
- Entwurfsmuster
- Refactoring

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/`

Organisation

- Vorlesung:
  - montags, 11:15–12:45, Li 318
- Übungen (Z423):
  - dienstags, 15:30–17:00
  - *oder* donnerstags 13:45–15:15
  - *oder* donnerstags 15:30–17:00
- 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

Beschreiben Sie 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 zwischen (Teilen von) Softwareprodukten

- wo sind die Schnittstellen, was wird transportiert? (Beispiele)
- wie wird das (gewünschte) Verhalten spezifiziert, wie sicher kann man sein, daß die Spezifikation erfüllt wird?

Schnittstellen (interfaces) in Java, Beispiel in Eclipse

- Eclipse (Window → Preference → Compiler → Compliance 6.0)
- Klasse A mit Methode main

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

Schnittstellen und Verträge

wenn jemand eine Schnittstelle implementiert, dann schreibt er nicht irgendwelche Methoden mit passenden Namen, sondern erfüllt einen Vertrag:

- Implementierung soll genau das tun,
  was beschrieben wird.
- Implementierung soll nichts
  anderes, unsinniges, teures, gefährliches tun.
- Implementierung soll bescheid geben,
  wenn Auftrag nicht ausführbar ist.

(Bsp: Pizzafehlermeldung)

Design by Contract

Betrand Meyer: *Object-Oriented Software Construction*, Prentice Hall, 1997.
`http://archive.eiffel.com/doc/oosc/`
Aspekte eines Vertrages:

- Vorbedingungen
- Nachbedingungen
- Klassen-Invarianten

Schnittstellen und Tests

man überzeuge sich von

- Benutzbarkeit einer Schnittstelle (unabhängig von Implementierung)
  . . . wird das gewünschte Ergebnis durch eine Folge von Methodenaufrufen vertraglich garantiert?
- Korrekheit einer Implementierung

mögliche Argumentation:

- formal (Beweis)
- testend (beispielhaft)
  . . . benutze *Proxy*, der Vor/Nachbedingungen auswertet

Stufen von Verträgen

(nach K. Pugh)

- Typdeklarationen
- Formale Spezifikation von Vor- und Nachbedingungen
- Leistungsgarantien (für Echtzeitsysteme)
- Dienstgüte-Garantien (quality of service)

Typen als Verträge

Der Typ eines Bezeichners ist seine beste Dokumentation.
(denn der Compiler kann sie prüfen!)
Es sind Sprachen (und ihre Sprecher) arm dran, deren Typsystem ausdrucksschwach ist.

```
int a [] = { "foo", 42 }; // ??

// Mittelalter-Java:
List l = new LinkedList ();
l.add ("foo"); l.add (42);

// korrektes Java:
List<String> l = new LinkedList<String> ();
```

Arten von Schnittstellen

Was wird verwaltet?

- Schnittstellen für Daten
  (Methoden lesen/schreiben Attribute)
- Schnittstellen für Dienste
  (Methoden „arbeiten wirklich“)

Schnittstellen zum Datentransport

Adressierung:

- wahlfreier Zugriff (Festplatte)
- sequentieller Zugriff (Magnetband)

Transportmodus:

- Pull (bsp. Web-Browser)
- Push (bsp. Email)

Bsp: SAX und DOM einordnen

Schnittstellen und Zustände

- Schnittstelle *ohne* Zustand
  - Vorteil: Aufrufreihenfolge beliebig
  - Nachteil: mehr Parameter (einer?)
- Schnittstelle *mit* Zustand
  - Nachteil: Reihenfolge wichtig
  - Vorteil: weniger Parameter

Mehrfache Schnittstellen

Eine Klasse kann mehrere Schnittstellen implementieren (deren Verträge erfüllen).
Dann aber Vorsicht bei der Bezeichnung der Methoden.
. . . und beim Verwechseln von Zuständen (Bsp. Pizza/Eis)

wesentliche Bestandteile

```
public class Zahl
   implements Comparable<Zahl> {
      public int compareTo(Zahl that) { .. }
}

Zahl [] a =
   { new Zahl (3), new Zahl (1), new Zahl (4) };
Arrays.sort(a);
```

Klassen-Entwurf

- `Zahl` hat ein `private final` Attribut,
  wird im Konstruktor gesetzt
- `Zahl` implementiert `String toString()`,
  dann funktioniert

```
System.out.println(Arrays.asList(a));
```

Richtig vergleichen

das sieht clever aus, ist aber nicht allgemeingültig:

```
public int compareTo(Zahl that) {
     return this.contents - that.contents;
 }
```

(merke: *avoid clever code*)

Protokollierung mit Dekorator

Aufgabe:

- alle Aufrufe von `Zahl.compareTo` protokollieren...
- *ohne* den Quelltext dieser Klasse zu ändern!

Lösung: eine Klasse dazwischenschieben

```
class Log<E> ... {
    private final contents E;
    int compareTo(Log<E> that) { .. }
}
Log<Zahl> a [] =
    { new Log<Zahl> (new Zahl (13)), .. };
```

Diese Klasse heißt *Dekorator*, das ist ein Beispiel für ein Entwurfsmuster.

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

Aufgabe: verschiedene (auch zusammengesetzte) geometrische Objekte
naive Lösung ohne Entwurfsmuster:

```
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
}   }
```

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

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

- Bäume (mit Schlüsseln in Blättern):

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

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)

Wiederholung Kompositum

Haskell: algebraischer Datentyp

```
data List a  = Nil | Cons a (List a)
     Nil  :: List a
     Cons :: a -> List a -> List a
```

Java: Kompositum

```
interface List<A> { }
class Nil implements List<A> { }
class Cons<A> implements List<A> {
    A head; List<A> tail;
}
```

implementieren/testen:

```
public class Main {
  List<Integer> range(from,to) { .. }
  public static void main(String[] args) {
    List<Integer> l = Main.range(0,10);
  }
}
```

Wiederholung Listen

Bezeichnungen bisher:

```
data List a  = Nil | Cons a (List a)
     Nil  :: List a
     Cons :: a -> List a -> List a
```

in Haskell-Prelude schon vordefiniert:

```
data     [a] = []  |       a : [a]
```

Anwendung:

```
import Prelude hiding ( length )
length :: [a] -> Int
length l = case l of
    []     -> 0
    x : xs -> 1 + length xs
```

Operationen auf Listen

```
length :: [a] -> Int
length l = case l of
    x : xs -> 1 + length xs
    []     -> 0
sum :: [Int] -> Int
sum l = case l of
    x : xs -> x + sum xs
    [] -> 0
f ::                 ->
f l = case l of
    x : xs -> g x (f xs)
    [] -> h
```

`f` durch *Rekursionsschema* mit Parametern `g` und `h`

Rekursionsschema foldr

```
foldr :: (a -> r -> a) -> a -> [r] -> a
foldr g h l = case l of
    x : xs -> g x (foldr g h xs)
    []     -> h
sum     = foldr ( \ x y -> x + y ) 0
length  =
product =
```

Kompositum und Visitor

Definition eines Besucher-Objektes

```
interface Visitor<A,R> {
  R empty();
  R nonempty(A here, R later);
}
```

Behandlung eines Besuchers: durch jeden Teilnehmer des Kompositums

```
interface List<A> { ..
  <R> R visit (Visitor<A,R> v);
}
```

- Implementierung
- Anwendung (length, sum)

Besucher (Aufgabe)

schreibe Methoden für

- Produkt
- Minimum, Maximum
- Wert als Binärzahl, Bsp:

```
Operation.binary
    (new List<Integer>(1,1,0,1)) ==> 13
```

Desgleichen für Bäume

algebraische Datentyp:

```
data Tree k = Leaf { key  :: k }
    | Branch { left :: Tree k
             , right :: Tree k }
```

Kompositum:

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

Bäume (Aufgabe I)

Konstruktoren, toString, Testmethode

```
class Trees {
    // vollst. bin. Baum der Höhe h
    static Tree<Integer> full (int h);
}
System.out.println (Trees.full(1))
   ==> Branch{left=Leaf{key=0},right=Leaf{key=0}}
```

Besucher für Bäume (Kompositа)

(dieses Beispiel sinngemäß aus: Naftalin, Wadler: Java Generics and Collections, O'Reilly 2006.)
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 Gast nimmt Besucher auf:

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

Bäume (Aufgabe II)

Benutzung des Besuchers: Anzahl der Blätter:

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

Funktionale Programmierung in C#

foldr = 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));
```

Verhaltensmuster: Iterator

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

Unendliche Datenstrukturen

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

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) { ... }`

Beispiel Iterator Java

```
static <E extends Comparable<E>>
List<E> merge(List<E> xs, List<E> ys) {
List<E> zs = new LinkedList<E>();
Iterator<E> xi = xs.iterator();
Iterator<E> yi = ys.iterator();
        // FIXME:
while (xi.hasNext() && yi.hasNext()) {
E x = xi.next();
E y = yi.next();
if (x.compareTo(y) < 0) {
zs.add(x);
} else {
zs.add(y);
}
}
return zs;
}
```

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) { ... }`

Beispiel Enumerator (C#)

```
static IList<int>
Merge (IList<int> xs, IList<int> ys) {
    IList<int> zs = new List<int>();
    IEnumerator<int> xi = xs.GetEnumerator();
    IEnumerator<int> yi = ys.GetEnumerator();
    bool xgo = xi.MoveNext();
    bool ygo = yi.MoveNext();
    // FIXME
    while (xgo && ygo) {
        if (xi.Current < yi.Current) {
         zs.Add(xi.Current);
         xgo=xi.MoveNext();
        } else {
         zs.Add(yi.Current);
         ygo=yi.MoveNext();
        }
    }
    return zs;
```

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;
    }
}
```

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)

Muster: Interpreter (Motivation)

(Wdhlg. Iterator)

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

soweit klar, aber wie macht man das besser:

```
class Store { ...
  Iterable<Data> larger_than_5 ();
  Iterable<Data> red ();
  Iterable<Data> green_and_even ();
}
```

Muster: Interpreter (Realisierung)

algebraischer Datentyp (= Kompositum) für die Beschreibung von Eigenschaften

```
interface Property { }
```

Blätter (Konstanten)

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

Verzweigungen (Kombinatoren)

...

und Programm zur Auswertung einer (zusammengesetzten) Eigenschaft für gegebenes Datum. (Typ?)

Interpreter (Material)

- Quelltext aus der Vorlesung: `http://dfa.imn.htwk-leipzig.de/cgi-bin/cvsweb/st09/src/kw18/Store.java?rev=1.1;cvsroot=pub`
- kann insgesamt als Eclipse-Projekt importiert werden: File → New → Project → from CVS,

```
connection type: pserver
user: anonymous
host: dfa.imn.htwk-leipzig.de
port: default
path: /var/lib/cvs/pub
module: st09
```

- Zukunft *jetzt* sichern! Wahlfach WS09/10 wählen! `http://www.imn.htwk-leipzig.de/~waldmann/lehre.html`

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)

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

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

diese Zustandsobjekte sind aber unsicher

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 (II)

interaktiver Sudoku-Solver:

- jedes Feld F hat Zustand (leer oder besetzt durch Zahl)
- jede Teilmenge M (Zeile, Spalte, Block)
  hat Zustand (Menge der bereits benutzten Zahlen)
- jedes M beobachtet alle F mit $F \in M$
- (jedes F beobachtet alle M mit $F \in M$)

Fragen:

- Zustandsänderungen zurücknehmen
- zyklische Abhängigkeiten
- GUI

(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

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

CVS-Überblick

(concurrent version system)

- Server: Archiv (repository), Nutzer-Authentifizierung
  ggf. weitere Dienste (`cvsweb`)
- Client (Nutzerschnittstelle): Kommandozeile
  `cvs checkout foobar` oder grafisch (z. B. intergriert in Eclipse)

Ein Archiv (repository) besteht aus mehreren Modulen (= Verzeichnissen)
Die lokale Kopie der (Sub-)Module beim Clienten heißt Sandkasten (sandbox).

CVS-Tätigkeiten (I)

Bei Projektbeginn:

- Server-Admin:
  - Repository und Accounts anlegen (`cvs init`)
- Clienten:
  - neues Modul zu Repository hinzufügen (`cvs import`)
  - Modul in sandbox kopieren (`cvs checkout`)

CVS-Tätigkeiten (II)

während der Projektarbeit:

- Clienten:
  - vor Arbeit in sandbox: Änderungen (der anderen Programmierer) vom Server holen (`cvs update`)
  - nach Arbeit in sandbox: eigene Änderungen zum Server schicken (`cvs commit`)

Konflikte verhindern oder lösen

- ein Programmierer: editiert ein File, oder editiert es nicht.
- mehrere Programmierer:
  - strenger Ansatz: nur einer darf editieren
    beim checkout wird Datei im Repository markiert (gelockt),
    bei commit wird lock entfernt
  - nachgiebiger Ansatz (CVS): jeder darf editieren, bei commit
    prüft Server auf Konflikte
    und versucht, Änderungen zusammenzuführen (merge)

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)

Logging (I)

bei commit soll ein Kommentar angegeben werden, damit man später nachlesen kann, welche Änderungen aus welchem Grund ausgeführt wurden.

- Eclipse: textarea
- `cvs commit -m "neues Feature: --timeout"`
- `emacs -f server-start &`
  `export EDITOR=emacsclient`
  `cvs commit`
  ergibt neuen Emacs-Buffer, beenden mit `C-x #`

Logging (II)

alle Log-Messages für eine Datei:

```
cvs log foo.c
```

Die Log-Message soll den *Grund* der Änderung enthalten, denn den *Inhalt* kann man im Quelltext nachlesen:

```
cvs diff -D "1 day ago"
```

finde entsprechendes Eclipse-Kommando!

Authentifizierung

- lokal (Nutzer ist auf Server eingeloggt):

```
export CVSROOT=/var/lib/cvs/foo
cvs checkout bar
```

- remote, unsicher (Paßwort unverschlüsselt)

```
export CVSROOT=:pserver:user@host:/var/lib/cvs/f
cvs login
```

- remote, sicher

```
export CVS_RSH=ssh2
export CVSROOT=:ext:user@host:/var/lib/cvs/foo
```

Authentifizierung mit SSH/agent

- Schlüsselpaar erzeugen (ssh-keygen)
- öffentlichen Schlüssel auf Zielrechner installieren (ssh-copy-id)
- privaten Schlüssel in Agenten laden (ssh-add)

Subversion

`http://subversion.tigris.org/` — “a better CVS”

- ähnliche Kommandos, aber anderes Modell:
- Client hat Sandbox *und* lokale Kopie des Repositories
  deswegen sind weniger Server-Kontakte nötig
- “commits are atomic” (CVS: commit einer einzelnen Datei ist atomic)
- Versionsnummer bezieht sich auf Repository (nicht auf einzelne Dateien)
  in Sandbox sind Dateien verschiedener Revisionen gestattet

Subversion (II)

- Server speichert Dateien und Zusatz-Informationen in Datenbank (Berkeley DB) (CVS: im Filesystem) unterstützt auch Umbenennen usw. mit Bewahrung der History.
- Subversion läuft als standalone-Server oder als Apache2-Modul (benutzt WebDAV)
- Kommandozeilen-Client wie cvs, Grafische Clients (TortoiseSVN), Webfrontends (viewCVS/viewSVN)

Weitere Erläuterungen zu Subversion im Vortrag von Enrico Reimer (Seminar Software-Entwicklung) `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss04/se/`

Darcs

David Roundy, `http://darcs.net/`

- nicht Verwaltung von *Versionen*, sondern von *Patches*
  gestattet paralleles Arbeiten an verschiedenen Versionen
- kein zentrales Repository
  (kann jedoch vereinbart werden)

vgl. Oberseminarvortrag

Übung CVS

- ein CVS-Archiv ansehen (cvsweb-interface) `http://dfa.imn.htwk-leipzig.de/cgi-bin/cvsweb/havannah/different-applet/?cvsroot=havannah`
- ein anderes Modul aus o. g. Repository anonym auschecken (mit Eclipse):
  (Host: `dfa.imn.htwk-leipzig.de`, Pfad: `/var/lib/cvs/havannah`, Modul `demo`, Methode: `pserver`, User: `anonymous`, kein Passwort)
  Projekt als Java-Applet ausführen. . . . zeigt Verwendung von Layout-Managern.
  Applet-Fenster-Größe ändern (ziehen mit Maus).
  Noch weiter Komponenten (Buttons) und Panels (mit eigenen Managern) hinzufügen.
- ein eigenes Eclipse-Projekt als Modul zu dem gruppen-eigenen CVS-Repository hinzufügen (Team → Share)
  [Daten ggf. für laufendes Semester/Server anpassen.]
  Host: `cvs.imn.htwk-leipzig.de`, Pfad:

Datei-Status

```
cvs status ; cvs -n -q update
```

- Up-to-date:
  Datei in Sandbox und in Repository stimmen überein
- Locally modified (, added, removed):
  lokal geändert (aber noch nicht commited)
- Needs Checkout (, Patch):
  im Repository geändert (wg. unabh. commit)
- Needs Merge:
  Lokal geändert *und* in Repository geändert

CVS – Merge

- 9:00 Heinz: checkout (Revision A)
- 9:10 Klaus: checkout (Revision A)
- 9:20 Heinz: editiert ($A \rightarrow H$)
- 9:30 Klaus: editiert ($A \rightarrow K$)
- 9:40 Heinz: commit (H)
- 9:50 Klaus: commit

  `up-to-date check failed`
- 9:51 Klaus: update

  `merging differences between` A `and` H `into` K
- 9:52 Klaus: commit

Drei-Wege-Diff

benutzt Kommando `diff3` *K A H*

- changes von $A \rightarrow H$ berechnen
- . . . und auf *K* anwenden (falls das geht)

Konflikte werden in *K* (d. h. beim Clienten) markiert
und müssen vor dem nächsten commit repariert werden.

tatsächlich wird `diff3` nicht als externer Prozeß aufgerufen,
sondern als internes Unterprogramm
($\rightarrow$ unabhängig vom Prozeß-Begriff des jeweiligen OS)

Unterschiede zwischen Dateien

- welche Zeilen wurden geändert, gelöscht, hinzugefügt?
- ähnliches Problem beim Vergleich von DNS-Strängen.
- Algorithmus: Eugene Myers: *An O(ND) Difference Algorithm and its Variations*, Algorithmica Vol. 1 No. 2, 1986, pp. 251-266, `http://www.xmailserver.org/diff2.pdf`
- Implementierung (Richard Stallman, Paul Eggert et al.): `http://cvs.sourceforge.net/viewcvs.py/*checkout*/cvsgui/cvsgui/cvs-1.10/diff/analyze.c`
- siehe auch Diskussion hier: `http://c2.com/cgi/wiki?DiffAlgorithm`

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 `diff`

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 Kette endlich
- jede Antikette endlich

Beispiel: bestimme die Menge der $\leq$-minimalen Elemente für

...

Die Einbettungs-Relation (III)

Die Endlichkeit von Ketten und 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/`

Aufgaben (autotool) zu LCS

- LCS-Beispiel (das Beispiel aus Vorlesung)
- LCS-Quiz (gewürfelt - Pflicht!)
- LCS-Long (Highscore - Kür)

LCS — naiver Algorithmus (exponentiell)

cvs2/LCS.hs
top-down: sehr viele rekursive Aufrufe . . .
aber nicht viele *verschiedene* . . .
Optimierung durch bottom-up-Reihenfolge!

LCS — bottom-up (quadratisch) + Übung

```
class LCS {

    // bestimmt größte Länge einer gemeinsamen Teilfolge
    static int lcs (char [] xs, char [] ys) {
        int a[][] = new int [xs.length][ys.length];
        for (int i=0; i<xs.length; i++) {
            for (int j=0; j<ys.length; j++) {
                // Ziel:
                // a[i][j] enthält größte Länge einer ge
                // von xs[0 .. i] und ys[0 ..j]
            }
        }
        return get (a, xs.length-1, ys.length-1);
    }

    // liefert Wert aus Array oder 0, falls Indizes zu k
    static int get (int [][] a, int i, int j) {
        if ((i < 0) || (j <0)) {
            return 0;
        } else {
```

LCS – eingeschränkt linear

Suche nach einer LCS = Suchen eines kurzen Pfades von `(0,0)` nach `(xs.length-1, ys.length-1)`.
einzelne Kanten verlaufen

- nach rechts: $(i-1, j) \rightarrow (i, j)$ Buchstabe aus `xs`
- nach unten: $(i, j-1) \rightarrow (i, j)$ Buchstabe aus `ys`
- nach rechts unten (diagonal): $(i-1, j-1) \rightarrow (i, j)$ gemeinsamer Buchstabe

Optimierungen:

- Suche nur in der Nähe der Diagonalen
- Beginne Suche von beiden Endpunkten

Wenn nur $\leq D$ Abweichungen vorkommen, dann genügt es, einen Bereich der Größe $D \cdot N$ zu betrachten $\Rightarrow$ *An* $O(ND)$ *Difference Algorithm and its Variations*.

diff und LCS

Bei `diff` werden nicht einzelne *Zeichen* verglichen, sondern ganze *Zeilen*.
das gestattet/erfordert Optimierungen:

- Zeilen feststellen, die nur in einer der beiden Dateien vorkommen, und entfernen
  `diff/analyze.c:discard_confusing_lines ()`
- Zum Vergleich der Zeilen Hash-Codes benutzen
  `diff/io.c:find_and_hash_each_line ()`

siehe Quellen `http://cvs.sourceforge.net/viewcvs.py/cvsgui/cvsgui/cvs-1.10/diff/`
Aufgabe: wo sind die Quellen für die CVS-Interaktion in Eclipse?

Versionierung (I)

. . . von Quelltexten

- CVS: jede Datei einzeln
- SVN: gesamtes Repository
- darcs, git: Mengen von Patches

das ist für die *Entwickler* ganz nützlich,
aber für die *Kunden* nicht!

Versionierung (II)

empfohlenes Schema:

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

Änderungen bedeuten:

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

Sonderformen:

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

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 $\Rightarrow$ Spezifikation
- Spezifikation $\Rightarrow$ 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. Instrumetierung):

- 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

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 reverse)
- 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:

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

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 -o he
./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

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))

Code-Optimierungen

Tony Hoare first said,
and Donald Knuth famously repeated,
Premature optimization is the root of all evil.

- erste Regel für Code-Optimierung: *don't do it . . .*
- zweite Regel: *. . . yet!*

Erst korrekten Code schreiben, *dann* Ressourcenverbrauch messen (profiling),
dann eventuell kritische Stellen verbessern.

Besser ist natürlich: kritische Stellen vermeiden.
Bibliotheksfunktionen benutzen!
Die sind nämlich schon optimiert (Ü: sort, binsearch)

Kosten von Algorithmen schätzen

big-Oh-Notation zum Vergleich des Wachstums von Funktionen kennen und anwenden

- einfache Schleife
- geschachtelte Schleifen
- binäres Teilen
- (binäres) Teilen und Zusammenfügen
- Kombinatorische Explosion

(diese Liste aus Pragmatic Programmer, p. 180)

die asymptotischen Laufzeiten lassen sich durch lokale Optimierungen *nicht* ändern, also: vorher nachdenken lohnt sich

Code-Transformationen zur Optimierung

(Jon Bentley: Programming Pearls, ACM Press, 1985, 1999)

- Zeit sparen auf Kosten des Platzes:
  - Datenstrukturen anreichern (Komponenten hinzufügen)
  - Zwischenergebnisse speichern
  - Cache für häufig benutzte Objekte
- Platz sparen auf Kosten der Zeit:
  - Daten packen
  - Sprache/Interpreter (Bsp: Vektorgrafik statt Pixel)
- Schleifen-Akrobatik, Unterprogramme auflösen usw. überlassen wir mal lieber dem Compiler/der (virtuellen) Maschine

Gefährliche „Optimierungen“

Gefahr besteht immer, wenn die Programm-Struktur anders als die Denk-Struktur ist.

- anwendungsspezifische Datentypen vermieden bzw. ausgepackt $\rightarrow$ primitive obsession
  (Indikator: String und int)
- existierende Frameworks ignoriert
  (Indikatoren: kein import java.util.*; sort selbst geschrieben, XML-Dokument als String)
- Unterprogramm vermieden bzw. aufgelöst $\rightarrow$ zu lange Methode (bei 5 Zeilen ist Schluß)

(später ausführlicher bei *code smells* $\rightarrow$ Refactoring)

Code-Metriken

Welche Code-Eigenschaften kann man messen? Was sagen sie aus?

- Anzahl der Methoden pro Klasse
- Anzahl der . . . pro . . .
- textuelle Komplexität: Halstaed
- strukturelle Komplexität: McCabe
- OO-Klassenbeziehungen

Code-Metriken: Halstaed

(zitiert nach Balzert, Softwaretechnik II)

- O Anzahl aller Operatoren/Operationen (Aktionen)
- o Anzahl unterschiedlicher Operatoren/Operationen
- A Anzahl aller Operanden/Argumente (Daten)
- a Anzahl unterschiedlicher Operanden/Argumente
- ($o + a$ Größe des Vokabulars, $O + A$ Größe der Implementierung)

Programmkomplexität: $\frac{o \cdot A}{2 \cdot a}$

Code-Metriken: McCabe

(zitiert nach Balzert, Softwaretechnik II)
zyklomatische Zahl (des Ablaufgraphen $G = (V, E)$)
$|E| - |V| + 2c$ wobei $c =$ Anzahl der Zusammenhangskomponenten
(Beispiele)
Idee: durch Hinzufügen einer Schleife, Verzweigung usw. steigt dieser Wert um eins.

OO-Metriken

- Attribute bzw. Methoden pro Klasse
- Tiefe und Breite der Vererbungshierarchie
- Kopplung (zwischen Klassen) wieviele andere Klassen sind in einer Klasse bekannt? (je weniger, desto besser)
- Kohäsion (innerhalb einer Klasse): hängen die Methoden eng zusammen? (je enger, desto besser)

Kohäsion: Chidamber und Kemerer

(Anzahl der Paare von Methoden, die kein gemeinsames Attribut benutzen) − (Anzahl der Paare von Methoden, die ein gemeinsames Attribut benutzen)
bezeichnet fehlende Kohäsion, d. h. kleinere Werte sind besser.

Kohäsion: Henderson-Sellers

- M Menge der Methoden
- A Menge der Attribute
- für $a \in A$: $Z(a) =$ Menge der Methoden, die a benutzen
- z Mittelwert von $|Z(a)|$ über $a \in A$

fehlende Kohäsion: $\frac{|M|-z}{|M|-1}$
(kleinere Werte sind besser)

Code-Metriken (Eclipse)

Eclipse Code Metrics Plugin installieren und für eigenes Projekt anwenden.

- `http://eclipse-metrics.sourceforge.net/`
- Installieren in Eclipse: Help → Software Update → Find → Search for New → New (Remote/Local) site
- Projekt → Properties → Metrics → Enable, dann Projekt → Build, dann anschauen

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/current/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

- Entwurfsänderungen . . .
  verwende Entwurfsmuster!
- „kleine“ Änderungen
  - Abstraktionen ausdrücken:
    neue Schnittstelle, Klasse, Methode, (temp.) Variable
  - Attribut bewegen, 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 $\longrightarrow$ 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 $\rightarrow$ FilePath, Email, ...
- 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(25);
    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:

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, String ext
```

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 height)
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,int)
java.awt.Graphics2D: draw (Shape);
    class Rectangle implements Shape;
```

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 () { ... }
}
```

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?

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
  $\longrightarrow$ *don't comment bad code, rewrite it*
- komplizierte Boolesche Ausdrücke
  $\longrightarrow$ umschreiben mit Verzweigungen, sinnvoll bezeichneten Hilfsvariablen
- Konstanten (*magic numbers*)
  $\longrightarrow$ 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 $\leftrightarrow$ 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 * 3.0 * 3
        System.out.println (5.5 + " " + 6 * 5.5 * 5
    }
}
```

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

Aufgaben zu Refaktoring (II)

- Eclipse $\rightarrow$ Refactor $\rightarrow$ Extract Interface
- “Create Factory”
- Finde Beispiel für “Use Supertype”

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)

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: bad, Schnittstellen-Vererbung: good.)

Problem: `class B extends A` $\Rightarrow$
B hängt ab von Implementations-Details von A.

$\Rightarrow$ Wenn man nur A ändert, kann B kaputtgehen.

(Beispiel)

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.