Softwaretechnik II
Vorlesung
Sommersemester 2007

Johannes Waldmann, HTWK Leipzig

13. Juni 2007

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)

Inhalt

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

Organisation

- Vorlesung:
  - mittwochs (u), 7:30–9:00, G126
  - dienstags (g), 13:45–15:15, G121
- Übungen (Z424):
  - Do (u) 11:15–12:45 und Mi (g) 7:30–9:00
  - oder: Do (u) 13:45–15:15 und Mi (g) 9:30–11:00
- Übungsgruppen wählen: `https://autotool.imn.htwk-leipzig.de/cgi-bin/Super.cgi`
- (für CVS/Bugzilla) jeder benötigt einen Account im Linux-Pool des Fachschaftsrates

Einleitung

Programme und Softwaresysteme

(Charakterisierung nach Brooks)

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

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)

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, ...

`http://fachschaft.imn.htwk-leipzig.de/`

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 (Th 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

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

- 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

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

Beispiel: Sortieren

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 (Überblick)

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 (voriges Jahr)

`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 (Beispiel)

Strukturmuster: Fliegengewicht

- ein Teil des Zustandes wird *externalisiert*,

  d. h. weiter außen verwaltet

- dann gibt es nur noch wenige verschiedene interne Zustände

  d. h. man braucht nur wenige verschiedene Objekte

Verhaltensmuster: Interpreter

Beispiele

- reguläre Ausdrücke für Dateinamen/Pfade
- Konfigurations-Angaben für Webserver

Verhaltensmuster: Memento

- Schnappschuß eines Objektes anlegen und speichern
- später in diesen Zustand rückversetzen

Verhaltensmuster: Zustand

- Objekt enthält Zustands-Objekt
- Änderung des Zustandes ändert das Verhalten
- . . . scheinbar ändert das Objekt seine Klasse

Verhaltensmuster: Beobachter

Siehe Code-Beispiel.
Mögliche Anwendungen bei Sudoku-Beispiel

- jede Zeile (Spalte, Block) beobachtet „ihre“ Zellen
- jede GUI-Komponenten beobachtet ihre Zelle

  (typisch: Model/View/Controller)

OO-Entwurfsmuster

Jedes Muster beschreibt eine in unserer Umwelt beständig wiederkehrende Aufgabe und erläutert den Kern ihrer Lösung. Wir können die Lösung beliebig oft anwenden, aber niemals identisch wiederholen.
Ausgangspunkt/Beispiel: Model/View/Controller (für Benutzerschnittstellen in Smalltalk-80)

- Aktualisierung von entkoppelten Objekten: *Beobachter*
- hierarchische Zusammensetzung von View-Objekten: *Komposition*
- Beziehung View–Controller: *Strategie*

Musterkatalog

- Erzeugungsmuster:
  - klassenbasiert: Fabrikmethode
  - objektbasiert: abstrakte Fabrik, Erbauer, Prototyp, Singleton
- Strukturmuster:
  - klassenbasiert: Adapter
  - objektbasiert: Adapter, Brücke, Dekorierer, Fassade, Fliegengewicht, Kompositum, Proxy
- Verhaltensmuster:
  - klassenbasiert: Interpreter, Schablonenmethode
  - objektbasiert: Befehl, Beobachter, Besucher, Iterator, Memento, Strategie, Vermittler, Zustand, Zuständigkeitskette

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 verschieden 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 . . .

Entwurfsmuster (Beispiele)

Kompositum - Aufgabe

verschiedene (auch zusammengesetzte) geometrische Objekte

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

Finde wenigstens sieben (Entwurfs-)Fehler und ihre wahrscheinlichen Auswirkungen. . .
(später: *code smells* → *refactoring*)

Kompositum - Anwendung

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


Verhaltensmuster: Beobachter

- 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 (. . .)

Beobachter: Beispiel

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


Model/View/Controller

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

- Daten-Modell: Zähler(stand)
- Anzeige: Label mit Zahl oder . . .
- Steuerung: Button

MVC-Beispiel

```
public class View implements Observer {
    private Label l = new Label("undefined
    Component getComponent() { return this
    public void update(Observable o, Objec
        this.l.setText(Integer.toString(((
}   }
public class Simple_MVC_Applet extends App
    Button b = new Button ("step");
    Counter c = new Counter (); View v = n
    public void init () {
        this.add (b); this.add(v.getCompon
        c.addObserver(v);
        b.addActionListener(new ActionList
```

```
    public void actionPerformed(Acti
      c.step();
    } }); } }
```


javax.swing und MVC

Swing benutzt vereinfachtes MVC (M getrennt, aber V un
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());
```


Swing: Bäume

```
// Model:
MutableTreeNode root = Shape.binary(4);
DefaultTreeModel tm = new DefaultTreeModel

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

// Steuerung:
t.addTreeSelectionListener(new TreeSelecti
   public void valueChanged(TreeSelectionE

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


Übung zu Entwurf(smustern)

- Muster-Beispiele (aus der Vorlesung): `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss06/case/programme/design/`
- Das Spiel Sudoku: `http://www.websudoku.com/`
- Welche Klassen sollten zu einem Sudoku-GUI gehören?
- Welche Entwurfsmuster sollte man dabei verwenden?
- `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss07/case/programs/sudoku/`

PS: die eigentlichen Sudoku-Fragen sind:

- wie findet man minimale Sudoku? Gibt es eines mit 16 Vorgaben? `http://www.csse.uwa.edu.au/~gordon/sudokumin.php`
- auf welchen anderen Graphen gibt es interessante Sudoku-Aufgaben?

(gern als Diplomthema)

Übung Muster (2)

Iterator:

- schreiben Sie in `Board`:

```
for (Cell a : this) {
  for (Cell b : this) {
    if (a.connected(b)) {
      a.addObserver(b);
    }
  }
}
```

und ergänzen Sie alles nötige (Eclipse: Control-1! oft!)

- implementieren Sie hasNext/next für die Iteratorklasse

- implementieren Sie Cell.connected (durch Delegation a
Point.connected, Implementierung dort)

Trennung Modell—View/Controller

- aus Cell alles, was Darstellung/Steuerung dient, in separate Klasse bewegen
- für diese Klasse eine Schnittstelle definieren (Eclipse: Refactor → extract interface)
- eine bessere Implementierung angeben (nicht wählbare
Buttons aus Panel entfernen)

Zusatz (künstliche Intelligenz):

- wenn durch eine Auswahl in einer Zelle an anderer Stel
nur eine Möglichkeit übrigbleibt, dann diese wählen (gg

mit weiteren Folgerungen)

Hilfe zum Testen: in Main ein Brett aufbauen

```
// http://people.csse.uwa.edu.au/gordon/sudo
// beachte: Zahlen 0 bis 8 statt 1 bis 9)
private static Board board17() {
       Board b = new Board ();
       b.vorgabe(0,7, 0);
       b.vorgabe(1,0, 3);
       b.vorgabe(2,1, 1);

       b.vorgabe(3,4, 4); b.vorgabe(3,6, 3);
       b.vorgabe(4,2, 7); b.vorgabe(4,6, 2);
       b.vorgabe(5,2, 0); b.vorgabe(5,4, 8);
```



```
       b.vorgabe(6,0, 2); b.vorgabe(6,3, 3
       b.vorgabe(7,1, 4); b.vorgabe(7,3, 0
       b.vorgabe(8,3, 7); b.vorgabe(8,5, 5

       return b;
}
```


Muster-Übung 3

ausgehend von Sudoku-Version mit Model-View/Controller-Trennug:
`http://www.imn.htwk-leipzig.de/~waldmann/edu/ss07/case/programs/sudoku-mvc`

Fabrik

- für Klasse CellVCImp eine (konkrete) Fabrikklasse CellVCImpFactory erzeugen (eine Methode: construct)
- für CellVCImpFactory eine Schnittstelle (abstrakte Fabrik) CellVCFactory erzeugen (mit der einen Methode)
- Cell-Konstruktor ändern: zusätzliches Argument vom Typ abstrakte Fabrik, beim Aufruf dann eine die konkrete

Fabrik einsetzen

Singleton (die konkrete Fabrik soll Singleton sein)

- statisches Attribut anlegen, das einzige Instanz enthält (oder null);
- Methode schreiben, die dieses liefert (bei Bedarf erzeugt);
- Konstruktor privatisieren.

Muster: Besucher

Klasse für Zahlenfolgen:

```
public class Zahlenfolge {

  private final List<Integer> contents;

  // Konstruktor mit variabler Argumentzahl
  public Zahlenfolge(Integer ... xs) {
          this.contents = new LinkedList<Int
          this.contents.addAll(Arrays.asList
  }

  // TODO: wird delegiert
  public void add (int x) {  }
```



```
  // TODO: wird delegiert
  public String toString() {  }

  // TODO, soll Folge [ lo, lo + 1 .. hi -
  public static Zahlenfolge range (int lo,
}
```

testen:

```
public class Main {
  public static void main(String[] args) {
    Zahlenfolge f = Zahlenfolge.range(0, 1
    System.out.println (f);
  }
}
```


Definition eines Besucher-Objektes (inneres Interface)

```
class Zahlenfolge { ...
  public interface Visitor {
    int empty();
    int nonempty(int previous_result, int el
  }
}
```

Behandlung eines Besuchers

```
class Zahlenfolge { ...
  public int visit(Visitor v) {
    int accu = v.empty();
    for (int x : this.contents) {
      accu = v.nonempty(accu, x);
    }
```

```
    return accu;
  }
}
```

Beispiel:

```
public class Operation {
  static int summe (Zahlenfolge f) {
    return f.visit(new Zahlenfolge.Visitor
      public int empty() {
        return 0;
      }
      public int nonempty(int previous_res
        return previous_result + element;
      }
    });
```

```
  }
}
```
Aufgaben: schreibe ähnliche Methoden für

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

```
Operation.binary (new Zahlenfolge(1,1,0,1))
```


Generischer Besucher

Eigentlich soll ja auch das gehen,

`Operation.contains (new Zahlenfolge(1,5,2,`

geht aber nicht, weil der Rückgabewert bisher auf int fixie
ist.

Lösung der Rückgabetyp wird ein Parameter:

```
class Zahlenfolge { ...
  public interface Visitor<R> {
    R empty();
    R nonempty(R previous_result, int elem
  }
}
```
Behandlung eines Besuchers
```
class Zahlenfolge { ...
```



```
  public <R> R visit(Visitor<R> v) {
    R accu = v.empty();
    for (int x : this.contents) {
      accu = v.nonempty(accu, x);
    }
    return accu;
  }
}
```
Beispiel:
```
public class Operation {
  static boolean contains (Zahlenfolge f, in
    return f.visit(new Zahlenfolge.Visitor<B
      public Boolean empty() { return false;
      ...
```



```
  }
}
```


Besucher für Bäume

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

binäre Bäume mit:

- Blatt enthält Schlüssel
- innerer Knoten hat zwei Kinder (aber selbst keinen Schlüssel)

(Wdhlg. Kompositum)
```
interface Tree<K> { }
class Branch<K> implements Tree<K> {
    Tree<K> left; Tree<K> right;
}
class Leaf<K> implements Tree<K> {
```



```
    K key;
}
```
Aufgabe: 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{
```
Besucher (mit Resultattyp R, für Bäume mit Schlüsseltyp
```
interface Tree<K> {
  interface Visitor<K, R> {
    R leaf (K key);
    R branch (R left, R right);
```



```
  }
  <R> R visit (Visitor<K,R> v);
}
```
Aufgabe: implementiere visit in Branch und Leaf

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

```
    });
  }
}
```
Hinweis: am 1. 5. keine Vorlesung (Feiertag), am 2. 5. nu
eine Übung (9:30) zu Fragen zu Entwurfsmustern.

Erzeugungsmuster

Bsp: fexible, konfigurierbare GUIs

- (Konstruktor)
- Fabrikmethode
- abstrakte Fabrik
- Singleton (Einermenge)
- Prototyp

Fliegengewichte

Beispiel: Schriftzeichen

- Höhe, Breite, Bitmap, Position (Zeile, Spalte)
- intrinsischer (innerer) und extrinsischer (äußerer) Zustand
- intrinsischer im Objekt, extrinsischer in Fabrik

Vergleich Iterator/Besucher

- Iterator: benutzt als Verweis in Datenstruktur, der Benutzer des Iterators steuert den Programmablauf
- Besucher: enthält Befehl, der für jedes Element einer Struktur auszuführen ist, Struktur steuert Ablauf.

Beispiel: Summation

Veränderungen in Entwürfen vorhersehen

- unflexibel: Erzeugen eines Elements durch Nennung seiner Klasse

  flexibel: abstrakte Fabrik, Fabrikmethode, Prototyp
- unflexibel: Abhängigkeit von speziellen Operationen

  flexibel: Zuständigkeitskette, Befehl
- unflexibel: Abhängigkeit von Hard- und Softwareplattfor

  flexibel: abstrakte Fabrik, Brücke
- unflexibel: Abhängigkeit von Objektrepräsentation oder -implementierung

- unflexibel: algorithmische Abhängigkeiten
- unflexibel: enge Kopplung
- unflexibel: Implementierungs-Vererbung
- Unmöglichkeit, Klassen direkt zu ändern

Quelltextverwaltung mit CVS

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 darau 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 nich
- 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 (merg

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, dami 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

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

- remote, sicher

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


Unser CVS-Server

- Server `cvs.imn.htwk-leipzig.de`, gehört zum Linux-Pool, von der Fachschaft betreut
- für jeden Studenten wurde ein Account eingerichtet (Einzelheiten im Praktikum)

  Arbeitsgruppen: nach Projekten

  Repositories: `/cvsroot/case05_XX`

  (Gruppe feststellen: auf `chef` einloggen, dort: `groups`)
- PS: Fachschaft sucht (immer) Studenten, die bei Betreuung des Pools mithelfen und diese später übernehmen.

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: `/cvsroot/case05_XX`, `XX` = Ihre Gruppennummer

  (CVS-Zugang benutzt Account im Linux-Pool, Gruppeneinteilung beachten)

- eine Datei ändern, commit; anderer Student gleicher Gruppe: update

  was passiert bei gleichzeitigen Änderungen und unabhängigen commits?

CVS – Einzelheiten

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 ($A \to H$)
- 9:30 Klaus: editiert ($A \to K$)
- 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 $A \to H$ 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 (→ 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 $x \le y$ 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 `diff`

Aufgaben (autotool) zu LCS

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

LCS — naiver Algorithmus (exponentiell)

```
-- | Länge einer längsten gemeinsamen Teil
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
    ]
```

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 Tei
    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 ein
                // 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 z
    static int get (int [][] a, int i, int j) {
        if ((i < 0) || (j <0)) {
            return 0;
        } else {
            return a[i][j];
        }
    }

    public static void main(String[] args) {
        String xs = "ABCABBA";
        String ys = "CBABAC";
        System.out.println (lcs (xs.toCharArray(), ys
    }
}
```

Aufgaben:

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

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

  damit dann die autotool-Aufgaben lösen.

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: $(i-1,j) \to (i,j)$ Buchstabe aus `xs`
- nach unten: $(i,j-1) \to (i,j)$ Buchstabe aus `ys`
- nach rechts unten (diagonal): $(i-1,j-1) \to (i,j)$ 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.*

JDK für Eclipse wählen

(im PC-Pool)
Eclipse → Window → Preferences → Java → Installed JRE → `E:\j2sdk_nb\j2sdk1.4.2`
Vorteil: direkter Zugriff auf API-Dok! (Bezeichner im Quelltext markieren, dann rechte Maus → Open Declaration)

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?

Dynamische Optimierung, Beispiel II

(eine Aufgabe aus Jon Bentley: Programming Pearls)

- Eingabe: Zahlenfolge $[x_1, x_2, \ldots, x_n]$,
- Ausgabe: Indizes (a, b), für die $x_a + x_{a+1} + \ldots + x_b$ maximal

naiver Algorithmus:

```
int best = 0;
for (a=1; a<n; a++) {
    for (b=a; b<n; b++) {
        int sum = 0;
        for (int i=a; i<= b; i++) {
            sum += x[i];
        }
        best = max(best,sum);
    }
}
```

Laufzeit? Geht besser?

...ja, mit passenden Invarianten:

```
int best_so_far
int best_ending_here
for (int i=1; i<n; i++) {
    // INVARIANT:
    // best_so_far ist ...
    // best_ending_here ist ...
}
```


wer sagt „das war ja leicht,“
der finde bitte einen effizienten Algorithmus für die entsprechende zweidimensionale Aufgabe:

- gegeben eine Matrix $x[1..m][1..n]$,
- gesucht eine Untermatrix $x[a..b][c..d]$ mit maximaler Elementsumme

Mehr zu CVS

Keyword Expansion

in den gemanagten Dateien werden Schlüsselwörter bein `commit` durch aktuelle Daten ersetzt.
Zu Beginn: `$Key$`, danach `$Key: Value $`

```
$Id: keyword.tex,v 1.1 2005-04-18 16:59:07 waldmann
$Author: waldmann $
$Date: 2005-04-18 16:59:07 $
$Header: /var/lib/cvs/edu/edu/ss07/case/folien/cvs
$Name:  $
$RCSfile: keyword.tex,v $
$Revision: 1.1 $
$Source: /var/lib/cvs/edu/edu/ss07/case/folien/cvs
$State: Exp $
```

Das Keyword `$Log$`

. . . wird durch die Liste *aller* Log-Messages ersetzt.
Damit das als Kommentar in Quelltexten stehen kann, erhält jede Zeile den gleichen Präfix:

```
// $Log: log.tex,v $
// Revision 1.1  2005-04-18 16:59:07  waldmann
// files
//
// Revision 1.2  2004/05/10 08:34:42  waldmann
// besseres LaTeX-display
//
// Revision 1.1  2004/05/10 08:26:25  waldmann
// Vorlesung 10. 5.
//
```

Die Nützlichkeit dieses Features ist umstritten, die vielen

Log-Messages lenken vom eigentlichen Quelltext ab (der soll ja *ohne* Kenntnis der Geschichte verständlich sein).

Text- und Binär-Dateien

per Default werden gemanagte Dateien als Textdateien behandelt:

- Keyword Expansion findet statt
- Zeilenenden werden systemspezifisch übersetzt

  (DOS: CR LF, Unix: LF)

das ist für Binärdateien (Bilder, Echsen) tödlich, diese gehören normalerweise auch nicht ins CVS.
Falls es doch nötig ist, kann man Dateien als *binär* markieren, dann finden keine Ersetzungen statt.

Symbolische Revisionen (Tags)

jedes Dokument hat seine eigene Versionsnummer (revision), z. B. (dieses Dokument):
`$Revision: 1.1 $`
Es gibt also *keine* Version eines gesamten Moduls. Abhilf
symbolische Revisionen (tags).
`cvs tag -r release-1_0`
Vorsicht: im Namen sind keine Punkte erlaubt
die Revisionsnamen können bei
`diff, update, checkout` benutzt werden.

Verzweigungen (branches)

Dei Geschichte eines Dokumentes ist per Default *linear*, kann jedoch bei Bedarf zu einem Baum verzweigt werden.
übliches Vorgehen bei größeren Projekten:

- ein *main branch*
- evtl. experimentelle branches
- akzeptierte Features werden in main-branch aufgenommen
- bei jedem Release wird ein release-branch abgezweigt
- wichtige Bugfixes aus main-branch werden auf release-branches angewendet

Branches (II)

Aufgaben:

- Betrachten Sie Tags/Branches im CVS-Quelltext: `http://ccvs.cvshome.org/source/browse/ccvs/diff/` z. B. Datei `diff3.c`
- Lesen Sie Erläuterungen zu Branches im CVS-Vortrag von Thomas Preuß (Seminar Software-Entwicklung) `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss04/se/`

CVS-Benachrichtigungen

Client-Befehl: `cvs watch add` beginnt *Beobachtung* eines (Teil-)Moduls: bei jeder Aktionen (commit, add) im Repository wird Email an *watchers* versandt.
In Datei `/var/lib/cvs/case_XX/CVSROOT/notify` steht der Mailer-Aufruf (per Default auskommentiert):
`ALL mail %s -s "CVS notification"`
Beachte: das Verzeichnis CVSROOT verhält sich (z. T.) wie ein CVS-Modul, d. h.

```
cvs checkout CVSROOT
cd CVSROOT
emacs notify
cvs commit -m mail
```

CVS-Benachrichtigungen (II)

Optional: In Datei
`/var/lib/cvs/case_XX/CVSROOT/users` steht
Address-Umsetzung:
`heinz:heinz@woanders.com`
Diese Datei ist nicht von CVS gemanagt, muß also direkt erzeugt werden.

Produktqualität (analytisch)

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)

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 unterbroche 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 revers
- 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*).

Übung zum Testen

Die folgende Methode soll binäre Suche implementieren:

- wenn (Vorbedingung) $\forall k : x[k] \le x[k+1]$,
- dann (Nachbedingung) gilt für den Rückgabewert p von `binsearch(x,i)`:

  falls i in $x[..]$ vorkommt,
  dann $x[p] = i$, sonst $p = -1$.

```
public static int binsearch (int [] x, int
    int n = x.length;
    int low = 0;
    int high = n;
    while (low < high) {
```



```
        int mid = (low + high) / 2;
        if (i < x[mid]) {
            high = mid;
        } else if (i > x[mid]) {
            low = mid;
        } else {
            return mid;
        }
    }
    return -1;
}
```

Aufgaben:

- Legen Sie eine Klasse an, die `binsearch` enthält.

- Legen Sie einen JUnit-Testcase an (Eclipse: File → Ne → JUnit Test Case) mit etwa diesem Code:

```
int [] x = { 3, 4, 6, 8, 9 };
int p = binsearch (x, 4);
assertTrue (p == ???);
```

- Finden Sie Argumente, für die sich die Methode fehlerhaft verhält.
- Reparieren Sie die Methode.
- Zusatz: Vergleichen Sie mit der entsprechenden Metho aus der Java-Standard-Bibliothek (welche Klasse? welcher Name?) Sie benötigen dazu ein komplettes JD (mit Quelltexten).

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.

Datenfluß-Analyse

Ablauf-Graph wird markiert:

- in jedem Knoten (Anweisung/Deklaration):

  welche Variablen gelesen (c-use)/geschrieben(def)?
- in jeder Kante: welche Variablen wurden gelesen (p-use um Sprung-Entscheidung zu fällen?

definitionsfreier Pfad (für eine Variable v): von def(v) zu u (v), ohne dazwischenliegende def (v)
Test-Kriterien:

- all-defs: jeder geschriebene Wert (def) wird benutzt
- all-uses: jede Art der Benutzung wird getestet

Daten lokalisieren

- Abstände von Definition zu Benutzung sollen *kurz* sein (= wenige Zeilen).

  „Variablen" sollen Konstanten sein (= sich nach erster Zuweisung nicht ändern)
- *Lokalitätsprinzip*: jede Variable so „lokal wie möglich"

```
for (int i = 0; i<N; i++) { int x = a [i];
```

- Hilfsvariablen vermeiden (z. B. Java 1.5)

```
for (int x : a) { ... }
```

Globale Variablen

- globale Variablen sind *evil*.
- wenn schon, dann:

  als private Attribute mit `set/get`-Methoden, und `set` sehr sehr sparsam verwenden
- much better:

  falls ein Unterprogramm eine globale Variable liest, dan soll es diese als zusätzlichen Parameter erhalten.

erleichtert Wiederverwendung! Richtlinie:

- eine Software-Komponente sollte *keinen* impliziten Zustand haben. (d. h. nicht von Variablenbelegungen abhängen).
- falls doch Zustand nötig, dann *Zustands-Objekt* definieren und dem Anwender in die Hand geben.

Beispiel: Brettspiel:

- *nicht* eine globale Variable „das Spielbrett", und *Prozedur* `void put (Zug z)` ändert das,
- *sondern* ein Typ `Brett` und *Funktion* `Brett put (Brett b, Zug z)`

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/eclips
```

(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 ande als die Denk-Struktur ist.

- anwendungsspezifische Datentypen vermieden bzw. ausgepackt → 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 → zu lange Methode (bei 5 Zeilen ist Schluß)

(später ausführlicher bei *code smells* → 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 Methode 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

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/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)

Primitive Daten (*primitive obsession*)

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

- fehlende Klasse:

  z. B. String → 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 ersetze
(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 hei
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) );
```


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 () { ... }
}
```


Organisatorisches

- Ergebnisse der Umfragen zur Vorlesung: `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss07/umfrage/`
- Vorlesung vom 26. 6. (letzte Vorlesungswoche) wird verschoben auf Montag, den 25. 6., 13:15 Uhr, Raum L 415. (Diese Vorlesung wird eine Fragestunde: Sie frage mich. In dieser Woche dann keine Übungen.)

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

  → *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)

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` ⇒
B hängt ab von Implementations-Details von A.

⇒ 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.

Code- und Interface-Dokumentation

Code dokumentieren?

warum?

- nach innen (Implementierung)

  für Wartung, Weiterentwicklung
- nach außen (Schnittstelle)

  soll ausreichen für Benutzung

wo?

- intern (im Quelltext selbst)
- extern (separates Dokument)

(Literatur: Steve McConnell, Code Complete 2)

Abstand v. Dokumenation u. Code

. . . je größer, desto gefährlicher! (d. h. interne Dokumentation ist noch relativ sicher)
wenn möglich, externe Dokumente daraus durch Werkzeuge generieren.
warum nur unsicher: der Compiler kann zwar den Code prüfen und ausführen, aber nicht die Kommentare.
Ideal: schreibe *selbst-dokumentierenden* Code!

Selbst-dok. Code: Klassen

- Schnittstelle ist konsistente Abstraktion?
- Name beschreibt Zweck?
- Benutzung der Schnittstelle offensichtlich?
- Black Box?

Selbst-dok. Code: Methoden

- hat wohlbestimmten Zweck?
- Name beschreibt Zweck?
- ist weit genug zerlegt?

Selbst-dok. Code: Daten

(Attribute, Variablen)

- Name beschreibt Zweck?
- nur zu einem Zweck benutzt?
- Aufzählungstypen anstelle von Flags oder Zahlen?
- Benannte Konstanten anstelle magischer Zahlen?

Selbst-dok. Code: Datenorganisation

- zur Verdeutlichung zusätzliche Variablen (Konstanten)?
- Benutzungen einer Variablen stehen eng beeinander?
- Komplexe Daten nur durch Zugriffsfunktionen benutzt?

Selbst-dok. Code: Ablauf

- normaler Ausführungsweg ist deutlich?
- zusammengehörende Anweisungen stehen beeinander
- gruppenweise unabhängige Anweisungen in Unterprogramme?
- Normalfall bei Verzweigung nach if (nicht nach else)
- jede Schleife hat nur einen Zweck?
- nicht zu tief geschachtelt?
- Boolesche Ausdrücke vereinfacht?

Selbst-dok. Code: Design

- versteht man den Code?
- ist er frei von Tricks?
- Details soweit wie möglich versteckt?
- benutzte Begriffe stammen aus Anwendungsbereich und nicht aus Informatik oder Programmiersprache

Kommentare

- Wiederholung des Codes (unsinnig und gefährlich)

  (. . . debug only the code, not the comments. Comments can be terribly misleading.)
- Erklärung des Codes

  (. . . don't *document* bad code — *rewrite* it!)
- Markierung im Code (TODO, FIXME, usw. — auch von Eclipse unterstützt)
- Zusammenfassung des Codes
- Absichtserklärung

Selbst-dok. Code: Warum?

Stelle dir beim Programmieren vor, daß nach dir ein gewalttätiger Psychopath mit deinem Code arbeitet, der auch weiß, wo du wohnst.
(anonym, zitiert in McDonnell)

Schnittstellen-Dokumentation

Zur Benutzung des Codes (einer Klasse/Methode) soll Kenntnis der *Schnittstelle* ausreichen.
Diese muß den *Kontrakt* dokumentieren.
wesentlicher Teil des Kontraktes ist der Typ (Anzahl und Typen von Argumenten und Resultat)
man kann aber nicht alles (*) durch statische Typen ausdrücken, deswegen Beschreibung hinzufügen

(*) aber doch sehr viel. Wenn nicht, liegt das evtl. an: primitive obsession, Datenklumpen, lange Parameterliste

JavaDoc

```
http://java.sun.com/j2se/javadoc/
writingdoccomments/index.html
/**
 * Returns an Image object that can then be painted o
 * The url argument must specify an absolute {@link U
 * argument is a specifier that is relative to the ur
 *
 * @param  url  an absolute URL giving the base locat
 * @param  name the location of the image, relative t
 * @return      the image at the specified URL
 * @see         Image
 */
 public Image getImage(URL url, String name) { ... }
```


Übung Javadoc

- in Eclipse unterstützt (Project → Generate JavaDoc, Window → Show View → JavaDoc)
- Probieren Sie JavaDoc-Annotationen aus! (Hinweis: Tippe `@` und danach CTRL-SPACE = auto-vervollständigen),
- generierte HTML-Dateien exportieren (Export → File System) und in HTML-Browser betrachten

ähnliche Werkzeuge für andere Sprachen

Doxygen

```
http://doxygen.org/
export PATH=/home/waldmann/built/bin:$PATH

doxygen -g dox.conf # erzeugt Default-Config
emacs dox.conf # Parameter einstellen:
    # PROJECT_NAME, OPTIMIZE_OUTPUT_JAVA,
    # INPUT, FILE_PATTERNS, RECURSIVE, SOURC
doxygen dox.conf # Dokumente herstellen
```


Software-Management (KW 17)

Management: Definition

(nach Balzert: Softwaretechnik, Band II)

- alle Aktivitäten und Aufgaben,
- die von einem oder mehreren Managern durchgeführt werden,
- um die Aktivitäten von Mitarbeitern zu planen und zu kontrollieren,
- damit ein Ziel erreicht wird,
- das durch die Mitarbeiter alleine nicht erreicht werden

könnte.

Management: Aufgaben

Management ebschäftigt sich mit
Ideen, Dingen, Menschen.
und umfaßt

- Planung
- Organisation
- Personalauswahl
- Leitung
- Kontrolle

Produktivität

Ziel des Software-Managements:
hohe Produktivität der Software-Erstellung

$$\text{Produktivität} = \frac{\text{Leistung}}{\text{Aufwand}}$$

Leistungsmessung?
(Exaktheit vs. Aussagekraft)

- Quelltext-Zeilen (LOC)
- (Korrektheit, Bedienbarkeit, . . . ?)
- Verkaufs-Erlös

Aufwandsmessung?

- Personalkosten
- Materialkosten
  - Rechner (Hard- und Software)
  - Büro-Ausstattung, Verbrauchsmaterial, . . .

Software-Projekt-Eigenschaften

(nach Kent Beck: Extreme Programming)
die vier wichtigen Eigenschaften sind:
Kosten, Zeit, Qualität, Umfang

man kann drei dieser Werte vorgeben,
und daraus ergibt sich der vierte. (Aufgabe: Beispiele)

Das Management möchte am liebsten alle vier Werte vorgeben . . . Dann leidet normalerweise die Qualität.

zum XP-Ansatz (dazu später mehr) gehört:
Umfang reduzieren! Möglichst schnell einen Prototyp herstellen, der nur die allerwichtigsten Funktionen implementiert.
(20 % des Codes liefert 80 % der Funktionalität)

Qualität

- externe Qualität: vom Kunden gemessen (Endprodukt)
- interne Qualität: von Entwicklern gemessen (Quellcode)

Kent Beck: „wenn man auf *höherer* Qualität besteht, werden Projekt häufig *schneller* fertig, oder es wird in einem bestimmten Zeitraum *mehr* erledigt.“
(Bsp: Entwurfsmethoden, Code-Standards, Spezifikationen, Tests)
zu kurz gedacht ist es,

- interne Qualität abzusenken (um kurzfristig Kosten/Zeit zu sparen)
- und zu hoffen, daß externe Qualität gleich bleibt.

Planung

. . . ist Vorbereitung zukünftigen Handelns:
festlegen, *was, wie, wann, durch wen* zu tun ist.
drei Abstraktions-Ebenen

- Prozeß-Architektur (allgemein)

  welche Typen von Prozeß-Elementen gibt es, welche Schnittstellen haben sie?
- Prozeß-Modell (Firmen- oder Abteilungs-spezifisch)

  eine bestimmte Anordnung von Prozeß-Elementen
- Projektplan (projektspezifisch)

  Inkarnation eines Prozeß-Modells

Prozeß-Elemente

Ein Prozeß-Element ist gekennzeichnet durch

- Vorbedingungen (entry)
- Aufgabe (task)
- Ergebnisse (exit)
- Maße (measurement)

Elemente sind verbunden durch

- Übergabe von Produkten (input/output)
- Rückkopplungen

Prozesse und Vorgänge

Jeder Prozeß wird untergliedert in *Vorgänge*.
Ein Vorgang ist in sich abgeschlossene, identifizierbare Aktivität mit

- Namen
- erforderlicher Zeitdauer
- Zuordnung von Personal und Betriebsmitteln
- Zuordung von Kosten und Einnahmen

  (abgeleitet aus Gesamtkosten-Schätzung des Projektes

Meilensteine

Vorgänge können zu *Phasen* zusammengefaßt werden.
Zur Projekt-Überwachung werden für Beginn und Ende von Phasen (oder einzelnen Vorgängen) *Meilensteine* festgelegt.
Meilensteine sind

- überprüfbar

  (konkretes (Teil-)Produkt soll vorliegen)
- kurzfristig

  (betrifft Zeitraum von 1 . . . 4 Wochen)
- gleichverteilt

  (z. B. jeden Monat ein Meilenstein)

Netzpläne

- Knoten: Vorgänge, mit Angabe von
  - Vorgangsdauer
  - frühester/spätester Anfang/Ende-Termin
  - (Arbeitsdauer pro Mitarbeiter, Gesamtzeitraum einschl. freier Tage)

  Meilenstein auffassen als Vorgang der Dauer 0
- Kanten (Pfeile): Abhängigkeiten A $\to$ B
  - Normalfolge: Ende(A) $\leq$ Anfang(B)
  - Anfangsfolge: Anfang(A) $\leq$ Anfang (B)
  - Endfolge: Ende(A) $\leq$ Ende (B)
  - Sprungfolge: Anfang(A) $\leq$ Ende (B)

– (Überlappungen, Verzögerungen)

Aufgabe: Beispiele für die möglichen Abhängigkeiten

Planung mit Netzplänen

aus dem Netzplan werden tatsächliche Termine (und Spielräume) für die Vorgänge bestimmt,
ergibt *(vorgangsbezogenes) Gantt-Diagramm* (Balkendiagramm = Intervallgraph).

- Vorwärtsrechung:

  jeder Vorgang zum frühest möglichen Termin

  (zu dem alle Voraussetzungen erfüllt sind)

- Rückwärtsrechnung: ausgehend von Projektende

  (oder Resultat der Vorwärtsrechnung):

  für jeden Vorgang den spätest möglichen Termin

Pufferzeiten, kritische Pfade

Vor- und Rückwärtsrechnung ergibt für jeden Vorgang eine *Pufferzeit*.

- freie Pufferzeit: mögliche Verzögerung, die *keinen anderen* Vorgang verzögert
- gesamte Pufferzeit: mögliche Verzögerung, die *Projektende* ncht verzögert

Ein Vorgang ohne Pufferzeit heißt *kritisch*.
Eine Folge von abhängigen kritischen Vorgängen heißt *kritischer Pfad*.
Diese müssen vom Management besonders überwacht werden.

Scheduling-Probleme

Die Termine für die Vorgänge sind so zu planen, daß sie

- die Abhängigkeiten (siehe Netzplan) erfüllen
- mit vorgegebenen Ressourcen (Mitarbeitern, Maschine ausgeführt werden können.

Diese Aufgabe erscheint in verschiedensten Varianten (Stundenpläne, Raumpläne, Fahrpläne, Betriebssysteme, Multiprozessor-Systeme ...).

Komplexität von Scheduling-Problemen

Mathematisch gehört Ressourcen-Scheduling zu Graphentheorie/Optimierung (siehe auch entsprechende Lehrveranstaltungen)
Die algorithmische Komplexität ist gut untersucht — für die meisten interessanten Varianten gilt aber:

- die Aufgabe ist NP-vollständig

  (N: es ist ein Suchproblem, P: der Suchbaum ist polynomiell tief, d. h. exponentiell breit)

  NP ist *nicht* die Abkürzung für „nicht polynomiell“, denn die Tiefe der Suchbäume ist eben *doch* polynomiell beschränkt! (N bedeutet „nicht- deterministisch“, ohne N wäre der Baum ein Pfad)

- d. h. es gibt (*) keinen Algorithmus, der in vertretbarer (polynomialer) Zeit eine optimale Lösung findet
- d. h. man muß Näherungs-Algorithmen finden und benutzen

Eine Liste von Scheduling-Aufgaben ist:
`http://www.nada.kth.se/~viggo/problemlist/compendium.html`
Lese-Übung: Erklären Sie Unterschiede zwischen Open, Flow und Job Shop Scheduling.

(*) sehr wahrscheinlich – das ist ein “million dollar problem
`http://www.claymath.org/millennium/P_vs_NP/`

Open-Shop Scheduling

als Optimierungsproblem:

- *Eingabe*: Anzahl $m \in \mathbb{N}$ von Prozessoren, Menge J von Jobs, jedes $j \in J$ besteht aus m Operationen $o_{i,j}$, für jedes $o_{i,j}$ eine Dauer $l_{i,j}$,
- *Lösung*: ein *Open-Shop-Plan* für J, d. h. für jeden Prozessor i eine Funktion $f_i : J \to \mathbb{N}$, so daß $f_i(j) > f_i(j') \Rightarrow f_i(j) \geq f_i(j') + l_{i,j'}$ und für jedes $j \in J$: die halboffenen Intervalle $[f_i(j), f_i(j) + l_{i,j})$ sind alle disjunkt.
- *Kriterium*: möglichst geringe Gesamtlaufzeit $\max\limits_{1 \leq i \leq m, j \in J} f_i(j) + l_{i,j}$

Als Entscheidungsproblem:

zusätzliche Eingabe: eine Zahl $T \in \mathbb{N}$
Frage: gibt es einen Plan mit Gesamtlaufzeit $\leq T$?

Aufgabe zu Projektplanung

nach Balzert: Softwaretechnik, Band II

Gegeben seien folgende Vorgänge:

- Vorgang 1, Aufwand 3 MT (Mitarbeiter-Tage), fester Anfang am 25. 4. 05
- Vorgang 2, Aufwand 20 MT
- Vorgang 3, Aufwand 15 MT
- Vorgang 4, Aufwand 5 MT, Ende (fest) am 17. 6. 05

Abhängigkeiten zwischen den Vorgängen:

- V2 kann sofort nach Ende von V1 beginnen
- V3 kann erst 5 Tage nach Ende von V1 beginnen
- V4 kann erst beginnen, wenn V3 beendet ist
- V4 beginnt frühest. 5 Tage vor dem Ende von V2

- Gehen Sie zunächst davon aus, daß jedem Vorgang ein anderer Mitarbeiter zugeordnet ist. Berechnen Sie zu jedem Vorgang die frühen und die späten Termine und geben Sie die Pufferzeiten an. Hat der resultierende Netzplan einen kritischen Pfad?
- Gehen Sie nun davon aus, daß das gesamt Projekt von einem einzigen Mitarbeiter durchgeführt wird. Kann ma in diesem Fall eine termin- und kapazitätstreue Bedarfsoptimierung vornehmen? Wenn nicht, welche Möglichkeiten hat man, um das Projekt dennoch durchzuführen?

- Der Mitarbeiter hat einen Überstundensatz von 500 EUR pro Werktag und 750 EUR an Sonn- und Feiertagen. Eine Verzögerung des Endtermins kostet 500 EUR Konventionalstrafe pro Tag. Welches ist für den Auftragnehmer die kostengünstigste Lösung?

Prozeß-Modelle (KW 18)

Definition (Aufgaben)

ein Prozeßmodell beschreibt einen organisatorischen Rahmen für die Software-Erstellung:

- Reihenfolge des Arbeitsablaufs
- Definition der Teilprodukte
- Fertigstellungs-Kriterien
- notwendige Mitarbeiter-Qualifikationen
- Verantwortlichkeiten und Kompetenzen
- anzuwendende Standards, Richtlinien, Methoden und Werkzeuge

(nach Balzert, Softwaretechnik, Bd 2, LE 4)

Das einfachste Prozeßmodell

. . . ist: *code & fix* (kodieren und reparieren)

Nachteile:

- bei jeder Reparatur wird Programm umstrukturiert, das erschwert folgende Reparaturen

  → vor Kodieren ist *Entwurf* nötig

- auch gut entworfene Software wird evtl. vom Kunden nicht akzeptiert

  → vor Entwurf ist *Definition* nötig

- zum Finden von Fehlern:

  → separate *Testphase* nötig

Das Wasserfall-Modell

Entwicklung in aufeinanderfolgenden *Stufen*:

- Definition (System-Anforderungen, Software-Anforderungen, Analyse) → Produktmodell
- Entwurf → Produktarchitektur
- Implementierung (Kodieren, Testen, Betrieb) → Produkt

Resultate einzelner Stufe ist ein Dokument, das an nächs Stufe übergeben wird.
Jede Aktivität muß vollständig ausgeführt werden, bevor die nächste beginnt.
Nutzerbeteiligung nur während der Definition.

Wasserfall (Eigenschaften)

- einfach, verständlich, wenig Management-Aufwand
- nicht immer sinnvoll, jeden Schritt vollständig durchzuführen
- nicht immer sinnvoll, Schritte streng sequentiell auszuführen
- Gefahr, daß Dokumente wichtiger werden als eigentliches System
- unflexibel: durch fixierten Ablauf können Risikofaktoren nicht berücksichtigt werden

Das V-Modell

integriert *Qualitätssicherung* in Wasserfall-Modell: Teilprodukte werden

- validiert (wird *das richtige Produkt* entwickelt?)

  durch Betrachtung von Anwendungs-Szenarien

- verifiziert (wird *ein korrektes Produkt* entwickelt?)

  durch Testen

als Standard zur Software-Verarbeitung bei Bundeswehr und Behördern festgelegt, auch in Industrie angewendet

Submodelle, Rollen

gegliedert in Submodelle für:

- Systemerstellung
- Qualitätssicherung
- Konfigurationsmanagement
- Projektmanagement

für jedes Submodell gibt es diese *Rollen*:

- Manager
  legt Rahmenbedingungen fest und ist oberste Entscheidungsinstanz
- Verantwortlicher
  plant, steuert, kontrolliert Aufgaben
- Durchführende

Aktivitäten, Produkte

besteht aus Aktivitäten, deren Ziel es ist:

- ein Produkt zu erstellen
- den Zustand eines Produktes zu ändern
- den Inhalt eines Produktes zu ändern

mögliche Zustände für Produkte:

- geplant
- in Bearbeitung (beim Entwickler)
- vorgelegt (unter Konfigurationsverwaltung)
- akzeptiert (nach Qualitätssicherung)

mögliche Zustandsübergänge:

- geplant → in Bearbeitung → vorgelegt → akzeptiert
- falls *nicht akzeptiert*, dann von *vorgelegt* wieder zu *in Bearbeitung*
- von *akzeptiert* zu *in Bearbeitung* nur mit neuer Versionsnumer

V-Modell, Eigenschaften

- umfassend, detailliert festgelegt
- Anpassungen (tailoring) möglich
- gut geeignet für große Projekte
- ungeeignet/zu aufwendig für kleiner Projekte
- viele „künstliche“ Produkte, → Software-Bürokratie

Probleme mit „klassischen“ Modellen

- Auftragnehmer will Auftraggeber von prinzipieller Realisierbarkeit des Projektes überzeugen
- zu Projektbeginn sind Anforderungen meist nicht klar erkannt
- Koordination zwischen Anwender und Entwickler auch nach Definitionsphase nötig
- z. B. zur Diskussion verschiedener Lösungsmöglichkeiten
- z. B. zur Diskussion der Realisierbarkeit bestimmter Anforderungen

möglicher Ausweg: Benutzung von Prototypen

Prototypen

- Demonstrations-Prototyp — dient zur Akquisition eines Auftrags, wird dann „weggeworfen“
- Prototyp im engeren Sinne — ist provisorisches, aber lauffähiges Softwaresystem, wird parallel zur Modellierung erstellt, veranschaulicht Aspekte der Nutzerschnittstelle oder der Funktionalität
- Labormuster — zum internen Experimentieren, technisc mit späterem Produkt vergleichbar
- Pilotsystem — ist bereits der Kern des tatsächlichen Produktes, wird nach Benutzerprioritäten weiterentwick

Arten von Prototypen

- horizontaler Prototyp:

  beschränkt auf eine System-Ebene, für diese aber möglichst vollständig

- vertikaler Prototyp:

  implementiert ausgewählte Aspekte über alle Ebenen hinweg

Prototyp und Produkt

mögliche Beziehungen zwischen Prototyp und fertigem System:

- Prototypen dienen nur zur Klärung von Problemen
- Prototyp ist Teil der Produktdefinition
- Prototyp wird weiterentwickelt und damit Bestandteil de Produktes

Prototypen: Bewertung (+)

- reduziert Entwicklungsrisiko
- können in andere Prozeßmodelle integriert werden
- können durch geeignete Werkzeuge schnell hergestellt werden
- Labormuster fördern Kreativität
- Rückkopplung mit Nutzer und Auftraggeber

Prototypen: Bewertung (-)

- höherer Aufwand (*)
- Prototyp muß oft fehlende Dokumentation ersetzen
- Gefahr, daß Wegwerf-Prototyp aus Ressourcenmangel doch Teil des Endproduktes wird
- Beschränkungen und Grenzen oft nicht genau bekannt

Herstellung eines Prototyps bedeutet höherern Aufwand (*)

(*) aber beachte:

- Fred Brooks 197?: (when designing a system ...) *plan* to throw one away, you will *do* so anyhow.
- Es ist billiger, erst ein 8-Zoll-Teleskop zu bauen, und danach ein 20-Zoll-Teleskop, als nur ein 20-Zoll-Teleskop.

Evolutionäres Modell

- allmähliche und stufenweise Entwicklung, gesteuert durch Erfahrungen der Auftraggeber und Nutzer
- Neue Version bei Erweiterung, aber auch Pflege des Produktes
- Gut geeignet, wenn Auftraggeber die Anforderungen nicht komplett überblickt (*Ich kann nicht beschreiben, w ich brauche, aber ich erkenne es, wenn ich es sehe*)
- Schwerpunkt sind jeweils lauffähige (Teil-)Produkte

Aufgabe (Google): woher kommt der Spruch: *release ear release often*, und wie geht er weiter?

Eigenschaften: flexibel, aber evtl. nicht flexibel genug (späte Design-Änderungen sind schwer)

Filmtipp: Revolution OS

naTo, Karl-Liebknecht-Str., Mittwoch 11. Mai 19 Uhr: *Revolution OS*, J.T.S. Moore, USA 2001, 85 min, OF

Microsoft fürchtet GNU/Linux - und das zu Recht. Die Open-Source-Bewegung ist derzeit die größte Bedrohung für Micros und die durch Markennamen und Patente geschützte Software-Industrie. Der Film erzählt die Geschichte der Menschen, welche gegen das Software-Modell der Markenrechte rebellierten un GNU, Linux und die Open-Source-Bewegung initiierten. Und er erzäh auch (unbeabsichtigt), wie der Erfolg die Bewegung inzwischen gespalten hat - in jene, die Open-Source kommerzialisiert haben und jene, die weiterhin für Freie Software kämpfen. Zu Wort kommen Lin Torvald, Richard Stallman u.a.

`http://www.nato-leipzig.de/film_aktuell.`

`php?itemid=40163`

Software? Peopleware! (V 27. 5.)

Die vier wichtigsten Elemente des Managements

Tom de Marco: *Der Termin*, Roman über Projektmanagement, dt: Hanser, 1998

„Daß Sie einen Kurs anbieten, der diese vier Punkte:

- Personalauswahl
- Aufgabenzuordnung
- Motivation
- Teambildung

ausspart und ihn trotzdem *Projektmanagement* nennen wollen.“
„Wie sollten wir ihn denn Ihrer Meinung nach nennen?“
„Wie wäre es mit . . . *Administrivialitäten*?“
sagte Tompkins, machte kehrt und ging hinaus.

Personal-Qualifikation

(Balzert, Softwaretechnik II)

- Fähigkeit zum Abstrahieren
- Fähigkeit zur spachlichen und schriftlichen Kommunikation
- Teamfähigkeit
- Wille zum lebenslangen Lernen
- intellektuelle Flexibilität und Mobilität
- Kreativität
- Hohe Belastbarkeit (unter Streß arbeiten können. *nicht* automatische Überstunden)
- Englisch lesen und sprechen (zusätzlich zum Deutsche

- Schreibmaschine schreiben

Spezialisierung

technische Großsysteme → Arbeitsteilung

- horizontale Spezialisierung:

  Spezialisten für Definition (Analytiker), Entwurf, Implementierung, Test

- vertikale Spezialisierung:

  Spezialisten für Datenbanken, Nutzerschnittstellen, Datenstrukturen/Algorithmen

(vgl. horizontale/vertikale Prototypen)

Spezialisierung (II)

vertikal:

- verlangt vom Einzelnen mehrere Qualifikationen,
- er führt jede Tätigkeit nur selten aus,
- Teilprodukte einer Ebene müssen passen

horizontal:

- volle Nutzung der speziellen Qualifikation
- Wiederholung ähnlicher Tätigkeiten in kurzen Abständen
- Höhere Chancen für Wiederverwendung
- leichter, dem „Stand der Technik“zu folgen
- verschiedene Produktebenen müssen passen

Spezialisierung und Management

mehr Spezialisierung: höhere Qualität und Produktivität, *aber nur durch* höheren Management-Aufwand:

- ungleichmäßige Auslastung
- unflexible Einsatzmöglichkeiten

Organisations-Strukturen

- funktionsorientiert:

  für jede horizontale Spezialisierung eine Abteilung

  (z. B. Marketing, System-Analyse, Konstruktion, Vertrieb)

- projekt/markt/produkt-orientiert:

  Projektleiter zur Steuerung der Mitarbeiter aus verschiedenen Abteilungen

  (schwierig, da er keine formale Autorität besitzt)

- Kombination ergibt *Matrix-Struktur*

Rollen

- System-Analytiker (requirements engineer)
- Software-Architekt
- Implementierer/Programmierer/Algorithmen-Konstrukte
- Qualitätssicherer
- Software-Ergonom
- Anwendungsspezialist
- Software-Manager

Laufbahnen

Mitarbeiter wollen zur langfristigen Motivation Perspektiven und Aufstiegschancen,
aber nicht jeder kann und will Manager werden, deswegen sollte man bieten:

- Führungslaufbahn

  (Aufstieg: mehr Personal-Verantwortung)

- Fachlaufbahn

  (Aufstieg: mehr fachliche Verantwortung)

- Wechsel-Möglichkeiten zwischen beiden Bahnen

Diskussion

- Fundamentalkritik:

  alle Management-Theorie ist Schönfärberei, die den wahren Charakter der kapitalistischen Lohnarbeit verschleiern soll

  bei Karl Marx klingt das so: „der Arbeiter“ verkauft seine Arbeitskraft an „den Kapitalisten“ dieser eignet sich den dadurch erzeugten Mehrwert an

- Fundamental-Erwiderung:

  wer statt Markt- eine kommunistische Planwirtschaft haben möchte, der kann ja nach Kuba auswandern.

⇒ wir leben gern im Kapitalismus, und lassen uns auch

Ziele des Managments (?)

Die Firma (vertreten durch Management) will vordergründing „nur “ Geld verdienen (durch Produkte und Dienstleistungen).
wie erreicht sie das?

- Ein Wechsel der Programmiersprache bringt riesige Vorteile

  (do not program *in* a language, but *into* a language)

- Das Projekt liegt hinter Plan, Sie müssen die Produktivität erhöhen

  (wahrscheinlich sind die Kostenschätzung und der Plan falsch)

- Sie müssen die Software-Entwicklung automatisieren

  (Schwerpunkt der Arbeit ist Kommunikation zwischen Entwicklern)

- Ihre Mitarbeiter arbeiten besser, wenn Sie sie unter Druck setzen

Management by ...

- Objectives (Zielsetzung)
- Results (Ergebnis-Messung)
- Delegation (Verteilen von Aufgaben und Befugnissen)
- Participation (Mitarbeiterbeteiligung)
- Alternatives (Entwicklung und Bewertung von alternativ Lösungen)
- Exception (Delegation und Eingriff nur bei Ausnahmen)
- Motivation:

  Bedürfnisse, Interessen, Einstellungen, Ziele der Mitarbeiter erkennen und mit Unternehmenszielen verbinden

gern managen, usw. usf.

Falsche Hoffnungen

(de Marco, Lister: Die sieben falschen Hoffungen des Managements)

- Es gibt einen neuen Trick zur Produktivitätssteigerung.

  (siehe auch Fred Brooks: „no silver bullet“)

- andere Manager erreichen 100 oder 200 Prozent mehr

  (ist oft Verkaufsargument für Werkzeuge, die aber doch nur bestimmte Projektphasen betreffen)

- Sie haben den Anschluß an die Technologie verpaßt

  (Grundlagen bleiben über Jahrzehnte hinweg gleich, Produktivität ändert sich wenig)

Ziele der Mitarbeiter

Was wollen die Software-Entwickler eigentlich?

- „nur“ das Geld?
- daß das Produkt funktioniert?
- gern auf Arbeit gehen:

  interessante, intellektuell herausfordernde Aufgaben;

  Zusammenarbeit mit Gleichgesinnten?

⇒ Management muß „nur“ dafür sorgen, daß die Entwickler(teams) gute Arbeitsbedingungen haben (*management von unten*)

Arbeitsbedingungen

- technische Arbeitsbedingungen:

  eigene, große Büros (Tür, Fenster, Tisch, Sessel),

  abstellbare Telefone, Kaffeemaschine usw.

- psychologische Bedingungen: (Sicherheit und Veränderung)

  Veränderung ist entscheidende Voraussetzung für Erfolg

  Veränderungen bringen Risiken, aber auch Chancen

  Menschen können Veränderungen nur in Angriff nehmen, wenn sie sich *sicher* fühlen

Peopleware

Tom de Marco, Timothy Lister: *Peopleware*, dt: Der Fakto
Mensch im DV-Management, Hanser, 1999

- Investitionen in das „Wohlbefinden“ der Mitarbeiter nütz langfristig dem Unternehmen.
- Der Zweck von Teams liegt nicht so sehr in der Ziel-Erreichung, als in der Ausrichtung auf ein *gemeinsames* Ziel.
- *never change a winning team*

Qualitäts-Management (KW 21)

Was ist Qualität?

Ansätze:

- produktbezogen
- benutzerbezogen
- prozeßbezogen
- kosten/nutzen-bezogen

DIN ISO 9126

- Funktionalität
- Zuverlässigkeit
- Benutzbarkeit
- Effizienz
- Änderbarkeit
- Übertragbarkeit

Aufgabe: Unterpunkte zuordnen (S. 259)

Messung von Qualität

Für Entwicklungsprozeß:
Qualitäts*ziele festlegen* und ihr Erreichen *messen*.

Vorsicht mit Fremdwörtern, z. B. Software-*Metrik*.

- Mathematik: (M, ρ) heißt *metrischer Raum*, falls $\rho : M \to \mathbb{R}_{\geq 0}$ mit $\forall x, y \in M : \rho(x, y) = 0 \iff x = y$ und $\forall x, y, z \in M : \rho(x, z) \leq \rho(x, y) + \rho(y, z)$.
- Physik: *messen* kann man physikalische Größen, durch Experimente, die objektiv und wiederholbar sind.
- Software: vgl. Literatur (!)

trotzdem: *jede* Art der Messung oder Schätzung ist besse
als *gar kein* Nachdenken über Software-Qualität.

Qualitäts-Management

- konstruktive QM-Maßnahmen:
  - produktorientiert (Methoden, Sprachen)
  - prozeßorientiert (Richtlinien, Werkzeuge)
- analytische QM-Maßnahmen:
  - analysierend
  - testend

beachte: *Testen* kann nur das *Vorhandensein* von Fehlern zeigen, niemals ihre *Abwesenheit*.

Qualitäts-Sicherung

- produkt- und prozeß-abhängig
- quantitativ
- maximal konstruktiv
- frühzeitige Fehler-Entdeckung und -Behebung
- entwicklungsbegleitend, integriert
- unabhängig

Qualitätssicherung im V-Modell

V(orgehens)-Modell:

- System-Erstellung (SE)
- Qualitätssicherung (QS)
- Konfigurationsmanagement (KM)
- Projektmanagement (PM)

dabei werden

- konstruktive Qualitätsmanagement-Maßnahmen in QS festgelegt und in SE angewendet
- analytische QM-Maßnahmen in QS festgelegt und auch durchgeführt

Bugzilla

System zur Fehlerverfolgung (bug tracking).
Einzelheiten siehe Seminar-Vortrag von D. Ehricht:
`http://www.imn.htwk-leipzig.de/~waldmann/edu/ss04/se/ehricht/bugzilla.pdf`
wichtig:

- Was ist ein Bug (Status, Severity), Lebenszyklus ("A Bug's Live") (Resolutions)
- Blocking: A blockiert B ($= B$ hängt ab von A):

  erst A beheben, dann B (andersherum nicht sinnvoll möglich)
- Voting: Einbeziehung der Anwender

Übung Bugzilla

ein existierendes Bugzilla-System betrachten
`http://bugzilla.mozilla.org/`

- bug writing guidelines
- vorbildliche bug reports

Welche Unterschiede sehen Sie zwischen diesen Reports? (öffnen Sie jeden in einem neuen Fenster, damit Sie sie überhaupt vergleichen können!)

- `http://gaos.org/pipermail/lug-l/2006-June/017931.html` (am Ende)
- `http://www.haskell.org/pipermail/haskell/2006-June/018111.html`

Was fehlt trotzdem noch?

Zusammenfassung

Inhalt

- Verifizieren, Spezifizieren (Korrektheit, Termination) (Vorbedingung, Nachbedingung, Invariante, „Terminante")
- Testen (Blackbox, Unit, Whitebox) (Überdeckungen)
- Quelltextverwaltung
- Entwurfsmuster, Refactoring, Code Smells
- Klassen (Immutabilität, Vererbung bricht Kapselung)
- Dokumentation (selbsterklärender Code, Schnittstellen-Dok.)

Einordnung der Softwaretechnik

- wissenschaftliche Grundlagen

  (Mathematik, Logik, Berechenbarkeit, Algorithmik)
- Methoden, Verfahren und Werkzeuge

  (vgl. Entwurfsmuster, Refactoring-Unterstützung in Eclipse)
- . . . und der Mensch?

  (vgl: de Marco: Software? Peopleware!)

Rolle der Hochschulen

`http://www.cs.utexas.edu/users/EWD/transcriptions/EWD13xx/EWD1305.html`
It is not the task of the University to offer what society asks for, but to give what society needs.
The programmer should not ask how applicable the techniques of sound programming are, he should create a world in which they are applicable; it is his only way of delivering a high-quality design.
Machine capacities now give us room galore for making a mess of it. Developing the austere intellectual discipline of keeping things sufficiently simple is in this environment a formidable challenge, both technically and educationally.

Verhältnis von Theorie und Praxis

„Nichts ist praktischer als eine gute Theorie."
Beispiel: von C#-2.0 zu C#-3.0
`http://msdn.microsoft.com/vcsharp/future/`:

- LINQ (Language integrateg query, typsicheres eingebettetes SQL + XQuery)
- Lambda-Ausdrücke

`http://en.wikipedia.org/wiki/Typed_lambda_calculus`

- Lambda-Kalkül (Church, 1936)
- intuitionistische Logik (Brouwer, 1920)
- kartesisch abgeschlossene Kategorien (Eilenberg, Mac

Lane, 1945)

Auswertung der Umfragen zur LV

- Trennung von VL Objektorientierte Konzepte
- Trennung von VL Softwaretechnik I
- Zusammenhang mit Softwarepraktikum

Zukünftiger Bachelor/Master-Plan

Bachelor:

- wie jetzt: 3: ST I, 4: ST II + Software-Praktikum
- Objektorientierung wird aufgelöst in:
  (Java-)Programmierung (2. Sem), Funktionale und Logische Programmierung und . . .

Master:

- neues Pflichtfach: Prinzipien von Programmiersprachen
- (bisheriges Pflichtfach Compilerbau wird Wahlfach)

Autotool-Highscore

```
40 :    37686   Andreas Tharandt
35 :    40301   Mike Wedemann
33 :    37607   David Redlich
```

Buchpreise gesponsort von Siemens Leipzig.
Angebote für Praktika, Diplomarbeiten, Stellen:
`http://www.imn.htwk-leipzig.de/~waldmann/edu/diplom/`