CASE und Projektmanagement Vorlesung, Sommersemester 2005

Johannes Waldmann, HTWK Leipzig

21. Juni 2005

Einleitung

Inhalt

Vorlesung und Übung:

- Programmieren im Kleinen, Werkzeuge (Eclipse)
- Programmieren im Team, Werkzeuge (CVS, Bugzilla)
- Management von Software-Projekten
- Qualitätskontrolle, u. a. Testen und Verifizieren

Seminar:

- Entwurfsmuster (design patterns)

Material

- Balzert: Lehrbuch der Software-Technik, Bände 1 und 2, Spektrum Akad. Verlag, 2000
- Gamma, Helm, Johnson, Vlissides: Entwurfsmuster, Addison-Wesley, 1996
- Holzner: Eclipse, O'Reilly, 2004

Vergleiche auch voriges Jahr:

- Vorlesung `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss04/case/folien/case/`
- Oberseminar `http://www.imn.htwk-leipzig.de`

Organisation

Koordinaten

- Vorlesung: montags, 19:00–20:30, Li 207
- Übung: dienstags und donnerstags, PC-Pool

  Einteilung in Projektgruppen wie im Softwarepraktikum (Prof. Weicker)

- Seminar: in gerade Wochen donnerstags, 17:15–18:45, G126

  jeweils 3 Vorträge (20 min + 10 min Diskussion) pro Seminar

Einschreibungen:

- Übungsgruppen wählen:
  `http://aaron.imn.htwk-leipzig.de/~autotool/cgi-bin/autoan.cgi`

- jeder benötigt einen Account im Linux-Pool:
  `http://www.imn.htwk-leipzig.de/fschaft/nav_linuxpool.html`

Programme und Softwaresysteme

(Charakterisierung nach Brooks)

- *Programm:* in sich vollständig, kann von seinem Autor auf dem System benutzt werden, auf dem es entwicklet 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)

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)

Übung KW 11

Planung Seminar Entwurfsmuster

Erich Gamma, Richard Helm, Ralph Johnson, John Vlilssides: *Design Patterns*, Addison-Wesley 1995

Organisation: je 3 .. 4 Themen pro Seminar, je 2 . . . 3 Studenten pro Thema.

Folien/Beispiele *gemeinsam* ausarbeiten, im Vortrag lege *ich* (von Folie zu Folie wechselnd!) fest, wer spricht.

In der Übung Gruppen/Themen anmelden und Literatur(-Kopie) ausleihen. Erste Vorträge in KW 12! (Bonus für Schnell-Entschlossene.)

- Erzeugungsmuster:

  Abstrakte Fabrik, Erbauer, Fabrik, Prototyp, Singleton

- Strukturmuster:

  Adapter, Brücke, Dekorierer, Fassade, Fliegengewicht, Kompositum, Proxy

- Verhaltensmuster:

  Befehl, Beobachter, Besucher, Interpreter, Iterator, Memento, Schablonenmethode, Strategie, Vermittler, Zustand, Zuständigkeitskette

Eclipse (I)

- Homepage `http://www.eclipse.org/`

  Beschreibungen, Downloads

- Eclipse starten, Tutorial (help $\rightarrow$ cheat sheet $\rightarrow$ hello word application) ausführen

Eclipse (II)

Die folgende Methode soll binäre Suche implementieren:

- wenn (Vorbedingung) $\forall k : x[k] \leq x[k+1]$,

- 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, in
    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;
}
```

Benutzen Sie diese Methode zum Testen:

```
public static void main(String[] args) {
```

```
        int [] x = { 3, 4, 6, 8, 9 };
        int p = binsearch (x, 4);
        System.out.println(p);
}
```

Aufgaben:

- Legen Sie eine Klasse an, die beide Methoden enthält. Setzen Sie einen Unterbrechungspunkt auf den Beginn von `binsearch`. Führen Sie dann `main` aus (ab dem Unterbrechungspunkt schrittweise).
- Finden Sie Argumente, für die sich die Methode fehlerhaft verhält.
- Reparieren Sie die Methode.

- Zusatz: Vergleichen Sie mit der entsprechenden Methode aus der Java-Standard-Bibliothek (welche Klasse? welcher Name?) Sie benötigen dazu ein komplettes JDK (mit Quelltexten).

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?

Struktur von Java-Programmen

Packages

Paket $=$ Sammlung von Klassen (und Schnittstellen)

Bei Klassendeklaration: Zugehörigkeit zu Paket angeben

```
package foo;
class Bar { .. };
```

Bezeichner aus Klassen aus anderen Paketen sichtbar machen:

```
import foo.Bar; // oder:
import foo.*; // alle Klassen diese Paket
```

Sichtbarkeits-Bereiche

Bei Deklaration eines Bezeichners (Klasse, Attribut, Methode) kann man festlegen:

- private: nur in aktueller Klasse sichtbar
- (default): von allen Klassen im eigenen Paket sichtbar
- protected: wie (default) und von allen abgeleiteten Klassen (auch in anderen Paketen)
- public: von anderen Paketen aus sichtbar

Gültigkeits-Bereiche

Vorsicht: Sichtbarkeiten (visibility) nicht verwechseln mit Gültigkeiten (scope, Lebensdauer) von Variablen

- lokal: Block
- nicht-statische Attribute (Instanzvariablen): Objekt
- statische Attribute: Klasse

Schnittstellen

Eine Methode ohne Implementierung heißt *abstrakt* (Gegenteil von: *konkret*)

Eine Klasse mit wenigstens einer abstrakten Methode muß als `abstract class` deklariert sein.

Abstrakte Klassen haben keine Objekte (können nicht instantiiert werden). . .

. . . man kann aber von ihnen konkrete Klassen *ableiten* (durch Implementieren der abstrakten Methoden).

Eine abstrakte Klasse mit nur abstrakten Methoden wird kürzer als Schnittstelle (`interface`) bezeichnet.

Schnittstellen (Beispiel)

Deklaration:

```
interface List<T> { T get (int index); ..
```

Implementierungen:

```
class LinkedList<T> implements List<T>
   { T get (int index) { ... } }
class ArrayList<T> implements List<T>
   { T get (int index) { ... } }
```

Benutzung:

```
{  List<String> l = new LinkedList<String
   ... l.get (3) ...
}
```

Deklarierter Typ sollte Interface sein (nicht Klasse).

(Zum polymorphen Containern siehe Vorlesung OO.)

Eclipse verwenden: ein Applet bauen

- Eclipse, New Class, Counter, extends Applet, deklariere `Button b` und `Labe l l`
- Source, Override/Implement, init, füge Button und Label zu Zeichenfläche
- füge Attribut `int state` hinzu
- Source, Generate Getter/Setter (für state)
- in Setter: Anzeige einfügen (`l.setText`)
- eine Methode `void step ()` zum Weiterzählen schreiben
- diese Methode soll bei Button-Click aufgerufen werden

Die ActionListener-Schnittstelle

```
class Counter { ...
    class AL implements ActionListener {
        void ActionPerformed (ActionEvent
    }
    Button b = new Button ("but");
    public void init () { ...
        b.addActionListener (new AL ());
    }
}
```

Anonyme Klassen

von der Klasse AL wird nur ein Objekt hergestellt.

Kürzere Schreibweise mit *anonymer Klasse*:

```
b.addActionListener (new ActionListener (
    public void actionPerformed(ActionEve
        step ();
    }
});
```

Diskussion zur Lokalität

Die Klasse *Counter* enthält zwei Bestandteile:

- Zähler (Stand und Änderung)
- Anzeige (Ein- und Ausgabe)

Diese hängen *zu eng* zusammen.

Beweis: versuchen Sie, alles, was zur Darstellung gehört. in eine separate Klasse *Display* zu verschieben — gelingt nicht, weil `setState` eine Display-Methode aufruft.

Wer–Wen?

Man muß für Counter und Display entscheiden, *wer wen* benutzt

Falls „Display benutzt Counter“ (und andersherum nicht), dann kann Counter nicht das Display aktualisieren.

für Counter neue Methode `String form ()`, die aktuellen Zählerstand als String repräsentiert. diese wird im ActionListener aufgerufen (direkt nach `step`)

Refaktoring: Klassen teilen

- Klasse Counter umbenennen in Display (Klassennamen anclicken, dann Refactor $\rightarrow$ Rename)
- neue Klasse Counter anlegen (File $\rightarrow$ New $\rightarrow$ Class), dorthin Attribute `int state` bewegen und alle Methoden, die dazugehören

Refaktorisieren: Schnittstelle einfügen

Dem Display ist es jetzt wirklich egal, was der Counter rechnet. Schnittstelle deklarieren, die nur die benötigten Methoden enthält.

In Counter: Refactor → Extract Interface, neuer Name: Stepper, welche Methoden? (zwei Stück)

Noch eine andere Implementierung `Texter` für Stepper hinzufügen, soll der Reihe nach die Zeichenketten `"x", "xx", "xxx", ...` anzeigen. (File → New → Class, implements Stepper)

In Display deklarieren/aufrufen:

```
Stepper c = // new Counter ();
          = new Texter ();
```

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 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

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

- remote, sicher

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

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 $\rightarrow$ Share)

  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 ($A \rightarrow H$)
- 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`

Aufgaben (autotool) zu LCS

`http://aaron.imn.htwk-leipzig.de/~autotool/cgi-bin/Super.cgi`

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

LCS — naiver Algorithmus (exponentiell)

```
-- | Länge einer längsten gemeinsamen Tei
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 Te
    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 e
                // von xs[0 .. i] und ys[0 ..j]
            }
        }
        return get (a, xs.length-1, ys.length-1);
    }
```

```
    // liefert Wert aus Array oder 0, falls Indiz
    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()
    }
}
```

Aufgaben:

- vervollständigen Sie die Methode `LCS.lcs`

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

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

  damit dann die autotool-Aufgaben lösen.

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*.

JDK für Eclipse wählen

(im PC-Pool)

Eclipse $\rightarrow$ Window $\rightarrow$ Preferences $\rightarrow$ Java $\rightarrow$ Installed JRE $\rightarrow$ `E:\j2sdk_nb\j2sdk1.4.2`

Vorteil: direkter Zugriff auf API-Dok! (Bezeichner im Quelltext markieren, dann rechte Maus $\rightarrow$ 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 (KW 16)

Keyword Expansion

in den gemanagten Dateien werden Schlüsselwörter beim `commit` durch aktuelle Daten ersetzt.

Zu Beginn: `$Key$`, danach `$Key: Value $`

```
$Id: keyword.tex,v 1.1 2005/04/18 16:59:07 waldma
$Author: waldmann $
$Date: 2005/04/18 16:59:07 $
$Header: /var/lib/cvs/edu/edu/ss05/case/folien/cv
$Name:  $
$RCSfile: keyword.tex,v $
$Revision: 1.1 $
$Source: /var/lib/cvs/edu/edu/ss05/case/folien/cv
$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.
Abhilfe: 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.

Code- und Interface-Dokumentation

Code dokumentieren?

warum?

- nach innen (Implementierung)
- nach außen (Schnittstelle)

wo?

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

(Literatur: Steve McConnell, Code Complete 2)

Abstand von Dokumenation und 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-dokumentierender Code: Klassen

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

Selbst-dokumentierender Code: Methoden

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

Selbst-dokumentierender Code: Daten

(Attribute, Variablen)

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

Selbst-dokumentierender Code: Datenorganisation

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

Selbst-dokumentierender Code: Steuerung

- 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-dokumentierender 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-dokumentierende 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)

Übung 16. KW

CVS-Einzelheiten

- Keyword-Substitution ausprobieren (Keywords in Quelltext schreiben, commit/update, Änderung beobachten)

  Keywords können auch im Programmtext stehen, z. B.

```
static final version = "$Id: ueb.tex,v 1

System.out.println ( "Program foobar ver
```

  (Vor/Nachteile?)

- Quelltextspeicherung im Server ansehen: auf `cvs.imn.htwk-leipzig.de` einloggen und `less /cvsroot/case05_XX/foo/Bar.java,v`

  (commit/update, Änderung betrachten)

JavaDoc

http://java.sun.com/j2se/javadoc/writingdoccomments/index.html

Beispiel

```
/**
 * Returns an Image object that can then
 * The url argument must specify an absol
 * argument is a specifier that is relati
 *
 * @param  url  an absolute URL giving th
 * @param  name the location of the image
 * @return      the image at the specifie
 * @see         Image
 */
```

```
public Image getImage(URL url, String na
        try {
            return getImage(new URL(url,
        } catch (MalformedURLException e)
            return null;
        }
}
```

- in Eclipse unterstützt (Project $\rightarrow$ Generate JavaDoc, Window $\rightarrow$ Show View $\rightarrow$ JavaDoc)
- Probieren Sie JavaDoc-Annotationen aus! (Hinweis: Tippe @ und danach CTRL-SPACE $=$ auto-vervollständigen),
- generierte HTML-Dateien exportieren (Export $\rightarrow$

File System) und in HTML-Browser betrachten

Quelltext-Dokumentation mit Doxygen

http://doxygen.org/

Beispiel-Ausgabe: http://dfa.imn.htwk-leipzig.de/~waldmann/jam/doxygen/

Benutzen (Sun-Pool) (erst Dateien auspacken, dann verarbeiten)

```
ssh abraham
bash -login
export PATH=/home/waldmann/built/bin:$PAT
export CVS_RSH=ssh
cvs -d :ext:student@cvs:/cvsroot/case05_X

doxygen -g dox.conf # erzeugt Default-Con
emacs dox.conf # Parameter einstellen:
```

```
        # PROJECT_NAME, OPTIMIZE_OUTPUT_JAVA,
        # INPUT, FILE_PATTERNS, RECURSIVE, SO
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. Aussage-kraft)

- 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 $\rightarrow$ 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, Maschinen) 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 $\max_{1 \leq i \leq m, j \in J} f_i(j) + l_{i,j}$

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 man 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

  $\rightarrow$ vor Kodieren ist *Entwurf* nötig

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

  $\rightarrow$ vor Entwurf ist *Definition* nötig

- zum Finden von Fehlern:

  $\rightarrow$ separate *Testphase* nötig

Das Wasserfall-Modell

Entwicklung in aufeinanderfolgenden *Stufen*:

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

Resultate einzelner Stufe ist ein Dokument, das an nächste 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 $\rightarrow$ in Bearbeitung $\rightarrow$ vorgelegt $\rightarrow$ 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, $\rightarrow$ 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, technisch mit späterem Produkt vergleichbar
- Pilotsystem — ist bereits der Kern des tatsächlichen Produktes, wird nach Benutzerprioritäten weiterentwickelt

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 des 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, was ich brauche, aber ich erkenne es, wenn ich es sehe*)
- Schwerpunkt sind jeweils lauffähige (Teil-)Produkte

Aufgabe (Google): woher kommt der Spruch: *release early, 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 Microsoft und die durch Markennamen und Patente geschützte Software-Industrie. Der Film erzählt die Geschichte der Menschen, welche gegen das Software-Modell der Markenrechte rebellierten und GNU, Linux und die Open-Source-Bewegung initiierten. Und er erzählt 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 Linus Torvald, Richard Stallman u.a.

http://www.nato-leipzig.de/film_aktuell.php?itemid=40163

Seminar-Folien

oft gesucht, nie gefunden:

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

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 Deutschen)
- Schreibmaschine schreiben

Spezialisierung

technische Großsysteme $\rightarrow$ 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-Konstrukteur
- 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

Management by . . .

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

  Bedürfnisse, Interessen, Einstellungen, Ziele der

Mitarbeiter erkennen und mit Unternehmenszielen verbinden

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.

$\Rightarrow$ wir leben gern im Kapitalismus, und lassen uns auch gern managen, usw. usf.

Ziele des Managments (?)

Die Firma (vertreten durch Management) will vordergründing „nur “ Geld verdienen (durch Produkte und Dienstleistungen).

wie erreicht sie das?

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)

- 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

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?

$\Rightarrow$ 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 Faktor Mensch im DV-Management, Hanser, 1999

- Investitionen in das „Wohlbefinden“ der Mitarbeiter nützt 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 besser 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 KW 21

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

- bug writing guidelines
- vorbildliche bug reports

das lokale Bugzilla-System ausprobieren `http://cvs.imn.htwk-leipzig.de/bugzilla/index.cgi`

- eigener Account (Passwort wurde an Linux-Pool-Account gemailt, lesen mit Web-Mailer `https://tuxmail.imn.htwk-leipzig.de/`
- für eigenes Software-Projekt eigenes Produkt/Komponente anlegen, Name: `case05_XX_product`
- eine Fehlermeldung schreiben, eine Fehler-Bearbeitungs-Meldung schreiben
- bei Bedarf Zugriffsrechte einschränken (Edit

Product $\longrightarrow$ edit Group Controls)

Inspektionen

Inspektion

Definition:

- detailgenaue Betrachtung von (Teil-)Produkten

  z. B. Entwurfsdokumente, Spezifikation, Code, Dokumentation

- durch (Gruppe von) Personen ($\neq$ Autor)
- um Fehler, Standard-Verletzungen und andere Probleme festzustellen

Teilnehmer:

- Moderator, Autor, Gutachter, (Protokollführer)

andere Bezeichnungen/Bedeutungen:

- Inspektion: formale Review
- Review
- Walkthrough: abgeschwächte Review

Inspektion (II)

erfordert

- Vorbereitung (durch alle Teilnehmer)
- Moderation

jeder Inspektor erhält eine bestimmte *Rolle* (d. h. inspiziert im Hinblick auf bestimmte Kriterien: benutzer, System, Finanzen, Qualität, Service)

während der Inspektion geht es um *Fehler-Feststellung*, nicht um *Fehler-Beseitigung*, *-Diskussion* oder *-Kommentierung*.

Inspektions-Tempo: ca. 1 Seite pro Stunde, gesamt $\leq$ 2 Stunden.

Inspektion (III)

Resultat ist Protokoll mit Auflistung von

- leichten/schweren Fehlern
  (während Vorbereitung gefunden)
- Fragen an den Autor
- leichten/schweren Fehlern
  (während Sitzung gefunden)
- Verbesserungsvorschlägen

Auswirkungen:

- Produkt freigeben
- Produkt überarbeiten, neue Inspektion
- Produkt wegwerfen, neu erstellen, neue Inspektion

Kosten/Nutzen von Inspektionen

Inspektionen finden Fehler. Inspektionen kosten Zeit (und damit Geld).

empirische Daten:

- die Hälfte aller Fehler bleibt unentdeckt
- ein Sechstel aller Korrekturen ist falsch

Clean Rooms

radikaler Ansatz zur Software-Entwicklung:

der Programmierer schreibt Code *für die Inspektion* (d. h. er kompiliert und testet nicht selbst).

Produktqualität (analytisch)

Klassifikation

- Testen (= Fehler erkennen)
  - statisch (z. B. Inspektion)
  - dynamisch (Programm-Ausführung)
- Verifizieren (= Korrektheit beweisen)
  - Verifizieren
  - symbolisches Ausführen
- Analysieren (= Eigenschaften vermessen/darstellen)

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)
  - kontrollfluß-orientiert
  - datenfluß-orientiert
- Funktionale Tests (black box)

Kontrollfluß-Tests

bezieht sich auf Kontrollfluß-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

Kontrollfluß-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 use (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, dann 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)`

Übung 15. 6.: Debugging/Profiling

auf `goliath` oder `aaron` einloggen (ssh)

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 -lr
gprof heap # Analyse
```

- Kompilieren und ausführen für

Überdeckungsmessung:

```
g++ -ftest-coverage -fprofile-arcs heap.
./heap > /dev/null
# welche Dateien wurden erzeugt? (ls -lr
gcov heap.cc
# welche Dateien wurden erzeugt? (ls -lr
```

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

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

```
http://junit.org/index.htm
import junit.framework.*;
class X extends TestCase {
  static int f (int y) { ... }

  public static void testf () {
    assertEquals ("foo", 4, f(7));
  }
  public static void main (String [] argv
      junit.awtui.TestRunner.run(X.class)
  } // führt alle  test*()-Methoden  aus
}
```

JUnit und Extreme Programming

Kent Beck empfiehlt *test driven apporach*:

- *erst* alle Test-Methoden schreiben,
- *dann* eigentliche Methoden implementieren

Tests und Verifikation

Programming by contract

man testet, ob (Teil-)Produkt seine Spezifikation erfüllt: diese ist ein Kontrakt (Vertrag):

- *wenn* Vorbedingungen (an Programmzustand, Argumente) erfüllt,
- *dann* nach Arbeit des Produktes:

  Nachbedingungen (an Programmzustand, Resultat) erfüllt

eingebaut z. B. in Sprache Eiffel von Bertrand Meyer, `http://www.eiffel.com/`

Beispiel: Funktion merge (aus mergesort):

- *wenn* `xs` und `ys` jeweils aufsteigend geordnete Listen,

- *dann* gilt für `zs = merge(xs, ys)`:
  - `zs` ist aufsteigend geordnete Liste
  - `zs` ist Permutation von `append(xs, ys)`

Erfüllung des Kontrakts kann man

- *testen* (assert, JUnit, Eiffel)

- . . . oder beweisen!

Verifikation

Betrachte Aussagen der Form `{V} P {N}` für

- Aussagen V, N (Vor- und Nachbedingung)
- Programm P

„wenn V gilt, und P ausgeführt wird, gilt danach N“

Beispiel:

`{x < 0 und y > 0} x = y - x; {x > 0}`

Frage: was ist die *schwächste Vorbedingung* V, so daß `{ V } x = y - x; {x > 0}`?

Antwort (geraten): `y > x`.

Aus *tatsächlicher Vorbedingun* `x < 0 und y > 0` folgt y>x, also ist o. g. Beispiel-Aussage wahr.

Statt *raten*: *rechnen* mit Regeln.

Schlußregel: Implikation

Vorbedingung verschärfen, Nachbedingung abschwächen

```
V' => V,  N => N',  {V} P {N}
-----------------------------
        {V'} P {N'}
```

Axiom: Zuweisung

$\{ N [x := A] \} \; x = A \; \{ N \}$

Beispiel: zeige

$\{x < 0 \text{ und } y > 0\} \; x = y - x; \; \{x > 0\}$

$N = (x > 0), \; A = (y - x)$

$N [x := A] = (x > 0) [x := (y-x)]$

$= (y-x) > 0 = (y > x)$

$\{ y > x \} \; x = y - x \; \{ x > 0 \}$

$\{x < 0 \text{ und } y > 0\}$ ist eine Verschärfung der Vorbedingung $\{y > x\}$

Regel: Sequenz

```
{V} P {M}, {M} Q {N}
--------------------
   {V} P ; Q {N}
```

Aufgabe: beweise

```
{ x = A und y = B }
x = x + y; y = x - y; x = x - y;
{ x = B und y = A }
```

Regel: Verzweigung

```
{ V und         B } P { N },
{ V und nicht B } Q { N }
-------------------------
{V} if B then P else Q {N}
```

Beispiel:

```
{ }
if x < y then z = x else z = y
{ z = min(x,y) }
```

Regel: Schleifen

```
        { V und B } P { V }
-----------------------------------
{ V } while B do P { V und nicht B }
```

V heißt *Invariante* der Schleife

Beispiel: beweise `power (b,e) = bˆe`

```
static int power (int b, int e) {
  int p = 1; // B = b, E = e
  while (e > 0) { // inv: BˆE = bˆe * p
    if (odd (e)) { p = p * b; }
    b = b * b; e = e / 2; // abgerundet
  }
  return p;
}
```

Invarianten finden?

Für gegebene Schleifen sind Invarianten schwer zu finden.

Deswegen („Extreme Programming“ a la Hoare, Dijkstra):

- *erst* die Invariante hinschreiben
- *dann* den Schleifenkörper passend programmieren

Oft sind Invarianten durch Datenstrukturen vorgegeben (Baum soll heap-geordnet oder balanciert oder Suchbaum sein)

Beispiel Heap-sort. Struktur-Invariante ist:

$\forall 1 < k < n : a[\lfloor k/2 \rfloor] \geq a[k]$

Automatische Verifikation?

Automatisches Beweisen von Programm-Eigenschaften ist praktisch unmöglich.

(Satz von Gödel, Turing: Halteproblem ist nicht entscheidbar.)

D. h., der Programmierer muß mithelfen: Programm und Beweis *gleichzeitig* schreiben.

Werkzeuge können verifizieren, daß beides zueinander paßt.

Simples Beispiel: Programm und Typ-Deklarationen.

Verifikation als Wundermittel?

in sicherheitskritischen Bereichen ist Verifikation Pflicht: Schaltkreis-Herstellung, Kryptographie, Luft- und Raumfahrt

jedoch:

man verifiziert immer nur bezüglich einer Spezifikation und mit Hilfe von Werkzeugen:

- wer verifiziert die Spezifikation?
- wer verifiziert die Werkzeuge?

siehe Lehrveranstaltungen im Hauptstudium (Prof. Geser, Prof. Petermann)

Aufgaben zum Testen und Verifizieren (KW 23)

- Aufgabe zu

```
x = x + y; y = x - y; x = x - y;
```

- Aufgabe zu schnellem Potenzieren

- Wie lautet die Spezifikation für

```
static void sort (int [] a);
```

Reicht das:

```
public void testSort () {
    int [] a = { 4,2,3,1,5 };
```

```
        sort (a);
        assertMonotonSteigend (a);
    }
```

- (falls nicht schon für Eclipse installiert, teste `import junit.framework.*`)

  JUnit tiefladen (`http://junit.org/`), auspacken und `junit.jar` installieren

- Schreibe Sortier-Klasse mit Testfall und dafür nötigen Hilfsmethoden.

```
package verify;
import junit.framework.*;
public class Sort extends TestCase { ...
```

- implementiere eine Sortiermethode, die zum Zugriff auf das zu sortierende Array *nur* `swap_if_gt` benutzt (d. h. *keine* weiteren Elementvergleiche oder Zuweisungen ausführt)

```
static void swap (int [] a, int i, int j
    int h = a[i]; a[i] = a[j]; a[j] = h;
}

static void swap_if_gt
  (int [] a, int i, int j) {
    if (a[i] > a[j]) {
        swap (a, i, j);
    }
}
```

- Hinweis: zwei Schleifen. Welches sind die Invarianten?

Partielle und totale Korrektheit

- partielle Korrektheit:

  wenn Vorbedingung erfüllt und P ausgeführt wird, *dann* gilt schließlich Nachbedingung

- totale Korrektheit:

  wenn Vorbedingung gilt, *dann* ist P tatsächlich komplett ausführbar *und* es gilt schließlich die Nachbedingung

für Schleifen:

- partielle Korrektheit: „Invarianz der Invariante“
- totale Korrektheit: . . . außerdem *Termination*

Termination: Beispiel

Hält dieses Programm? Nach wievielen Schritten?

```
int main () {
  stack<int> s;
  s.push (4); // oder 5 oder ...
  for (int k = 1; ! s.empty(); k++) {
    int top = s.top (); s.pop ();
    if (top > 0) {
      for (int j=0; j<k; j++) {
        s.push (top - 1);
      }
    }
  }
}
```

Termination „von selbst“

am sichersten sind Programme, die „von selbst“ terminieren:

- ganz ohne Schleifen/Rekursion
- nur mit „einfachen“ Schleifen:

```
for (int k = 0; k < 100; k++) { .. }
```

```
Collection <T> c; for (T x : c) { .. }
```

solche Schleifen dürfen auch geschachtelt sein

...sog. *primitiv rekursive Funktionen*

Termination für Ersetzungs-Systeme

Regelmenge (z. B. $R = \{ab \to bba\}$)
definiert Relation auf Wörtern

R terminiert $\iff$ es gibt keine unendlich lange R-Ableitung ($=$ Folge von Regel-Anwendungen)

- wieviele Schritte kann man im Beispiel von $a^k b$ aus durchführen?

- terminieren diese Systeme?
  - $\{10000 \to 000011110\}$,
  - $\{aa \to bc, bb \to ac, cc \to ab\}$

- für welche p, q, r, s terminiert $a^p b^q \to b^r a^s$?

  Beispiel: $p = q = 2, r = s = 3$.

- Workshop on Termination `http://www-i2.informatik.rwth-aachen.de/WST04/`
- Matchbox `http://dfa.imn.htwk-leipzig.de/matchbox/`
- RTA list of open problems `http://www.lsv.ens-cachan.fr/rtaloop/problems/21.html`

Entwurfsmuster, Refactoring

Entwurfsmuster

Erich Gamma, Ricahrd Helm, Ralph Johnson, John Vlissides: *Entwurfsmuster (design patterns)* — Elemente wiederverwendbarer objektorientierter Software, Addison-Wesley 1994.

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/`

Vorlage: Muster in der Architektur

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

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

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

15. . . . diese Muster verbreiten und verbessern, indem wir sie testen: erzeugen sie in uns das Gefühl,

daß die Umgebung lebt?

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

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

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

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

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 Softwareplattform

  flexibel: abstrakte Fabrik, Brücke

- unflexibel: Abhängigkeit von Objektrepräsentation oder -implementierung
- unflexibel: algorithmische Abhängigkeiten
- unflexibel: enge Kopplung
- unflexibel: Implementierungs-Vererbung
- Unmöglichkeit, Klassen direkt zu ändern

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)

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,
```

Indikator: ähnliche, schematische Attributnamen

Lösung: Klasse definieren

```
class File { String base; String extensio

static boolean is_writable (File f);
```

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 () { ... }
}
```

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; ...}`)

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)

- Liste von *refactorings* (mit Beispielen):

  `http://www.refactoring.com/catalog/`

- Liste von *code smells*:

  `http://wiki.java.net/bin/view/People/SmellsToRefactorings`

- Beispiele ch6-properties, ch6-template, ch14-ttt

  `http://www.imn.htwk-leipzig.de/~waldmann/edu/current/case/rwb/`

(beides aus Refactoring Workbook von William C. Wake `http://www.xp123.com/rwb/`)

Aufgaben zu Refaktoring (II)

Refactoring-UnterstÃ¼tzung in Eclipse:

Für diesen Quelltext:

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

Diese Änderungen vornehmen (in dieser Reihenfolge). Jeweils `preview` benutzen.

- eine der Zahlen `3.0` markieren, dann: extract local

variable

- den Teilausdruck 6 * d * d markieren, dann: extract method
- in der neuen Methode: d * d markieren, dann: extract local variable

. . . hier nicht sinnvoll, aber ausprobieren:

- lokale Variable markieren, dann: inline
- einen Methodenaufruf markieren, dann: inline