CASE und Projektmanagement Vorlesung

Johannes Waldmann, HTWK Leipzig

Einleitung

Inhalt

Vorlesung und Übung:

- Programmieren im Kleinen, Werkzeuge (Eclipse)
- Spezifizieren, Verifizieren, Testen
- Entwurfsmuster, Refactoring
- Programmieren im Team, Werkzeuge (CVS, Bugzilla)
- Management von Software-Projekten

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, ...

Vergleiche auch voriges Jahr: Vorlesung `http://www.imn.htwk-leipzig.de/~waldmann/` Oberseminar `http://www.imn.htwk-leipzig.de/~waldmann/ed`

Organisation

- Vorlesung: montags, 7:30–9:00, Li 110
- Übungen (Z424):
  - unger. Freitag 11:15 und gerader Freitag 15:30
  - oder unger. Freitag 15:30 und gerader Montag 11:15

  („Seminar“→ Objektorientierte Übung)
- Übungsgruppen wählen: `https://autotool.imn.htwk-leipzig.de/cgi-bi`
- (für CVS/Bugzilla) jeder benötigt einen Account im Linux-Pool des Fachschaftsrates

Leistungen:

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

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

Software ist schwer zu entwickeln

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

(Balzert, Band 1, S. 26 ff)

Produktivität

Ein Programmierer schafft etwa

10 (in Worten: zehn)

(vollständig getestete und dokumentierte) Programmzeilen pro Arbeitstag.
(d. h. ≤ 2000 Zeilen pro Jahr)

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

The Pragmatic Programmer

(Andrew Hunt and David Thomas)

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

Lernen! Lernen! Lernen!

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

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

Regelmäßig investieren

(Pragmatic Programmer)

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

Übung KW 11

red Brooks: The Mythical Man Mon

- 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?

sger W. Dijkstra über Softwaretech

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

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?

Spezifikation und Verifikation

Programming by contract

jedes (Teil-)Produkt muß seine Spezifikation erfüllen, 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

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 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 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) = b^e`

```
static int power (int b, int e) {
  int p = 1; // B = b, E = e
  while (e > 0) { // inv: B^E = b^e * 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:

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)

ufgaben zum Testen und Verifiziere

- 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);
}
```


e richtigen Invarianten für Bubble-S

```
static void sort (int [] a) {
  int top = a.length-1;
  for (int i=top; i>1; i--) {
    for (int p=i+1; p <= top; p++) {
      dominates (a, p);
    }
    for (int j=0; j<i; j++) {
      dominates (a, j);
      swap_if_gt (a, j, j+1);
} } }

dominates(a,p) ==
```


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

Termination von Programmen ist schwer (genauer: unentscheidbar)

```
import java.util.Stack;
void dauert_lange (int start) {
  Stack<Integer> s = new Stack<Integer> ();
  s.push (start);
  for (int k = 1; ! s.empty(); k++) {
    int 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-System

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 von $a^k b$ aus?
- terminieren diese Systeme?
  - $\{10000 \to 000011110\}$,
  - $\{aa \to bc, bb \to ac, cc \to ab\}$
  - $\{101 \to 000, 000 \to 111\}$ (ist offen, gibt Preis!)
- für welche p, q, r, s terminiert $a^p b^q \to b^r a^s$?

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)

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.

Dynamische Tests: Black/White

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

Black-Box-Tests

ohne Programmstruktur zu berücksichtigen.

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

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
- ...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

Eclipse (I)

- Homepage `http://www.eclipse.org/` Beschreibungen, Downloads
- richtige Shell nehmen, Prompt setzen

  ```
  bash -login
  export PS1='\u@\h:\w$ '
  ```

  Pfade setzen:

  ```
  export PATH=/home/waldmann/built/bin:$PATH
  export MANPATH=/home/waldmann/built/man:$MA
  export LD_LIBRARY_PATH=/home/waldmann/built
  ```

  Alle `export` in File `~/.bashrc` schreiben, werden dann automatisch bei bash-Start

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, int i)
    int n = x.length;
    int low = 0;
```


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

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:

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

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

Beispiel-Programm(e):
`http://www.imn.htwk-leipzig.de/~waldmann/edu/s`
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
```


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.

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

de-Transformationen zur Optimieru

(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)

Gefährliche „Optimierungen"

Gefahr besteht immer, wenn die Programm-Struktur anders 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)

Code-Metriken

Eclipse Code Metrics Plugin installieren und für eigenes Projekt anwenden.

- Suchen mit google
- Installieren in Eclipse: Help → Software Update → Find → Search for New → New (Remote/Local) site
- Projekt → Properties → Metrics → Enable, dann Projekt → Build, dann anschauen

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

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

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_ope`
- Programme mit grafischer Ein- und Ausgabe sollen Informationen *vollständig* von und nach

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/lib/
cvs login
```

- remote, sicher

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


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/cvsw`
- 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).

Entwurfsmuster, Refactoring

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/s`

Beispiel zu Entwurfsmustern

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

- Dokumentstruktur
- Formatierung
- Benutzungsschnittstelle

eispiel: Strukturmuster: Komposit

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

eispiel: Verhaltensmuster: Strateg

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

Beispiel: Strukturmuster: Dekorier

darstellbare Objekte sollen optional eine Umrahmung oder eine Scrollbar erhalten

el: Erzeugungsmuster: (abstrakte)

Anwendung soll verschiedene GUI-Look-and-Feels ermöglichen

Beispiel: Verhaltensmuster: Befeh

Menü-Einträge, Undo-Möglichkeit

siehe auch Ereignisbehandlung in Applets (Beispiel)

ster und Funktionales Programmie

einige Muster beziehen sich auf *Funktionen* (Unterprogramme, Methoden), verpacken diese aber in *Objekte*, denn...
in Java sind Unterprogramme nicht gleichberechtigt (mit Objekten), sie dürfen nicht Argument oder Resultat eines Unterprogramms sein, nicht Attribut eines Objektes.
Typische Anwendungen:

- Befehl (z. B. ActionListener),
- Strategie (z. B. Comparator).
  dazu Beispiel `java.util.Arrays.sort(...)`

Beachte Spezifikation für durch compareTo

er und Programmierung höherer Or

einige Muster simulieren (die in der meisten OO-Sprachen fehlenden) Funktionen höherer Ordnung.
Beispiel: $S(f, n) := \sum_{x=1}^{n} f(x)$, dann
$S(x \mapsto x^2, 3) = 14$
(für Mathematiker ganz selbstverständlich, für Programmierer ... hoffentlich auch)
(siehe Java-Code)

Anonyme Klassen

wenn eine Klasse *nur* dazu dient, ihre *einzige* Methode zu kapseln, dann kann sie anonym sein. Ihre Definition erfolgt im Konstruktor-Aufruf, unter Angabe der Schnittstelle:

```
interface Function { int call (int x);  }
...
System.out.println
  (compute (new Function () {
    public int call(int x) { return x*x*x; }
  }, 3));
```

... in C# gibt es dafür Delegates und Lambda-Ausdrücke

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 speicher
- 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)

Verhaltensmuster: Beobachter

- Subjekt:
  - meldeAn (Beobachter o)
  - meldeAb (Beobachter o)
  - protected benachrichtige ()
- Beobachter:
  - aktualisiere ()

Übung zu Entwurf(smustern)

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

PS: die eigentlichen Sudoku-Fragen sind:

- wie bewertet man die Schwierigkeit für den

-Entwurfsmuster@

Jedes Muster beschreibt eine in unse
beständig wiederkehrende Aufgabe u
den Kern ihrer Lösung. Wir können d
beliebig oft anwenden, aber niemals
wiederholen.
Ausgangspunkt/Beispiel: Model/View
(für Benutzerschnittstellen in Smalltal

- Aktualisierung von entkoppelten O
  Beobachter
- hierarchische Zusammensetzung
  View-Objekten: *Komposition*

Musterkatalog

- Erzeugungsmuster:
  - klassenbasiert: Fabrikmethode
  - objektbasiert: abstrakte Fabrik, Erbauer, Prototyp, Singleton
- Strukturmuster:
  - klassenbasiert: Adapter
  - objektbasiert: Adapter, Brücke, Dekorierer, Fassade, Fliegengewicht, Kompositum, Proxy
- Verhaltensmuster:
  - klassenbasiert: Interpreter,

Wie Entwurfsmuster Probleme löse

- 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

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,

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/c`

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)

erwendung von Daten: Datenklump

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, Strin
```

Indikator: ähnliche, schematische Attributnamen
Lösung: Klasse definieren

```
class File { String base; String extension; }

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 → 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

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!)

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

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 (); }
```


null-Objekte

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

Richtig refaktorisieren

- immer erst die Tests schreiben
- Code kritisch lesen (eigenen, fremden), eine Nase für Anrüchigkeiten entwickeln (und für perfekten Code).
- jede Faktorisierung hat ein Inverses.
  (neue Methode deklarieren $\leftrightarrow$ Methode inline expandieren)
  entscheiden, welche Richtung stimmt!
- Werkzeug-Unterstützung erlernen

Aufgaben zu Refaktoring (I)

- Code Smell Cheat Sheet (Joshua Keriewsky):
  `http://industriallogic.com/papers/smellstor`
- Smell-Beispiele
  `http://www.imn.htwk-leipzig.de/~waldmann/ed`
  (aus Refactoring Workbook von William C. Wake `http://www.xp123.com/rwb/`)
  ch6-properties, ch6-template, ch14-ttt

Aufgaben zu Refaktoring (II)

Refactoring-Unterstützung in Eclipse:

```
package simple;

public class Cube {
    static void main (String [] argv) {
        System.out.println (3.0 + " " + 6 * 3.
        System.out.println (5.5 + " " + 6 * 5.
    }
}
```

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

Aufgaben zu Refaktoring (II)

- Eclipse $\rightarrow$ Refactor $\rightarrow$ Extract Interface
- “Create Factory”
- Finde Beispiel für “Use Supertype”
- Würfelspiel “Schummel-Mex” (aufdecken oder erhöhen). Was ist hier alles falsch:
  `http://www.imn.htwk-leipzig.de/~waldmann/ed`

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: Datenorganisatio

- 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/writingdoccom

```
/**
 * Returns an Image object that can then be painted on the scre
 * The url argument must specify an absolute {@link URL}. The n
 * argument is a specifier that is relative to the url argument
 *
 * @param  url  an absolute URL giving the base location of the
 * @param  name the location of the image, relative to the url
 * @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, SOURCE_
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 abschäftigt 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

Aufwandsmessung?

- Personalkosten
- Materialkosten
  - Rechner (Hard- und Software)

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

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,

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)

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

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)

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

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

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 . . .).

omplexität von Scheduling-Problem

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

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

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

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 nächste Stufe übergeben wird.
Jede Aktivität muß vollständig ausgeführt werden, bevor die nächste beginnt.

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

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)

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

robleme mit „klassischen“ Modelle

- 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

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:

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

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

Software? Peopleware! (V 27. 5.)

r wichtigsten Elemente des Manag

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

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)

Spezialisierung

technische Großsysteme $\longrightarrow$ 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

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

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.

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

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

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

Bugzilla

System zur Fehlerverfolgung (bug tracking).
Einzelheiten siehe Seminar-Vortrag von D. Ehricht:
`http://www.imn.htwk-leipzig.de/~waldmann/edu/s`
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/0`

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/transcripti`
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_calc`

- 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/d`