CASE und Projektmanagement
Vorlesung, Sommersemester 2004

Johannes Waldmann, HTWK Leipzig

25. Juni 2004

Literatur

[1] Fred Brooks: *Vom Mythos des Mann-Monats*. 1975, 2003, mitp-Verlag, `http://www.mitp.de/vmi/mitp/detail/pWert/1355`

[2] Helmut Balzert: *Lehrbuch der Software-Technik,* Band 1: Software-Entwicklung, Band 2: Software-Management, -Qualitätssicherung, Unternehmensmodellierung. Spektrum Akad. Verl., 2000

[3] Perdita Stevens und Rob Pooley: *UML*, Addison-Wesley/Peaerson, 2000

[4] Heide Balzert: *UML kompakt*. Spektr. Akad. Verl., 2001

[5] Gerhard Henschel: *Die wirrsten Grafiken der Welt*.
Hoffmann und Campe, 2003
`http://www.hoca.de/cat.cfm?art_key=6350&cat_key=10&type=21&master_cat=10`

Einleitung, Use Cases (15. 3.)

Programme und Softwaresysteme

(Charakterisierung nach Brooks)

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

- *Komponente eines Systems:* Schnittstellen, Integration

- *marktfähiges Produkt:* Generalisierung, Tests, Dokumentation, Wartung

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)

Vorlesungen

- gerade Wochen:
  - Mo 13:45 – 15:15 (G 329) CASE
  - Mo 15:30 – 17:00 (HB 207) OO
  - Do 13:45 – 15:15 (G 119) CASE
- ungerade Wochen:
  - Mo 13:45 – 15:15 (Li 318) CASE ($\rightarrow$ OO)

Seminare/Praktika

TI-D und IB

- gerade Wochen:
  - Mo 17:15 – 18:45 OO
  - Di 7:30 – 9:00 CASE
- ungerade Wochen:
  - Di 7:30 – 9:00 CASE
  - Di 13:45 – 15:15 OO

OO in Z 530, CASE in Li 107

PI-D

- gerade Wochen:
  - Mo 11:15 – 12:45 OO
  - Di 9:30 – 11:00 CASE
- ungerade Wochen:
  - Mo 15:30 – 17:00 OO
  - Di 9:30 – 11:00 CASE

Organisation der Seminare

Durchführung: Frau König,
`http://www.imn.htwk-leipzig.de/~koenig/`
studentische Hilfskraft: Herr Ehricht,
`dehricht@imn.htwk-leipzig.de`

- Diskussion von Übungsaufgaben
- Kennenlernen von CASE-Werkzeugen, Anwendung für Ihr Softwarepraktikum.

Sie arbeiten in den gleichen Gruppen wie in Softwaretechnik/Softwarepraktikum, und bearbeiten das dort gewählte Thema.

Anforderungen an Produktentwicklung

- Funktionstreue
- Qualitätstreue
- Termintreue
- Kostentreue

(Balzert, Band 1, S. 34 ff)

Einordnung

Informatik:

- Theoretische, Technische, Praktische, Angewandte.

Praktische Informatik:

- . . . , Programmier-Konzepte, Softwaretechnik, . . .

Softwaretechnik:

- Software-Entwicklung
- Software-Management
- Software-Qualitäts-Management

Definition Softwaretechnik

- zielorientierte Bereitstellung und systematische Verwendung
- von Prinzipien, Methoden und Werkzeugen
- für die arbeitsteilige, ingenieurmäßige
- Entwicklung und Anwendung
- von umfangreichen Softwaresystemen

(Definition nach Balzert, Band 1)

Die UML (unified modeling language)

Modeling: designing of software applications before coding.

Grafische Sprache UML `http://www.uml.org/`

. . . specify, visualize, and document models of software systems, including their structure and design

- Structural Diagrams:

  Class D., Object D., Component D., and Deployment D.

- Behavior Diagrams:

  Use Case D., Sequence D., Activity D., Collaboration D., and Statechart D.

- Model Management Diagrams:

  Packages, Subsystems, and Models.

Diagramme f. Geschäftsprozesse (use cases)

- Knoten:
  - Akteure (Strichmännchen)
  - Geschäftsprozesse (use cases) (Ellipse) – Menge dieser Prozesse bildet das *System* (Kasten)

- Kanten:
  - *ungerichtet!* zwischen Akteur A und Prozeß G (falls A an G beteiligt ist)
  - *gerichtet* von Prozeß G nach Prozeß H, falls
    - G eine Erweiterung von H ist, $G \stackrel{\text{extend}}{\longrightarrow} H$,
    - H ein wesentlicher Bestandteil (Teilprozess) von G ist, $G \stackrel{\text{include}}{\longrightarrow} H$,

Fragen zu UCD

Übungsaufgaben zu: Pfeilrichtungen? extend oder include?

`http://www.andrew.cmu.edu/course/90-754/umlucdfaq.html`

Seminare in Woche 12

Rational Rose kennenlernen, zur Übung: Diagramme für Geschäftsprozesse (Use Cases) Ihres Softwareprojektes zeichnen.

Für jede Gruppe: bringen Sie eine kurze Beschreibung Ihres Projektthemas mit, auch mit Namen, Matrikelnummer, Email der Beteiligten.

Graphen, Aktivitäts-D. (18. 3.)

Diagramme/Graphen

Graph $G = (V, E)$ mit

- V: Menge von Knoten (Sg. vertex, Pl. vertices)
- E: Menge von Kanten (edge)

ungerichteter Graph: jede Kante ist eine Menge von zwei Knoten, $E \subseteq \binom{V}{2}$
gerichteter Graph: jede Kante ist ein geordnetes Paar von Knoten, $E \subseteq V \times V = V^2$
Wir schreiben xy für die Kante $\{x, y\}$ bzw. (x, y).

Wege, Zusammenhang

Eine Folge $(x_0, x_1, \ldots, x_n)$ von Knoten heißt *Weg* (zwischen x_0 und x_n), falls

- für $0 \leq i < n : x_i x_{i+1} \in E(G)$,
- $x_0, x_1, \ldots, x_{n-1}$ paarweise verschieden sind

(für ungerichtete Graphen)

Die Relation $x \sim_G y$ falls es in G einen Weg zwischen x und y gibt, ist eine Äquivalenzrelation auf $V(G)$.

Die Äquivalenzklassen von $\sim_G$ heißen *Zusammenhangskomponenten*.

Ein Graph heißt *zusammenhängend*, wenn er nur eine Zusammenhangskomponente besitzt.

Starker Zusammenhang

(für gerichtete Graphen)

Kanten sind gerichtet, also auch Wege.

(Die Relation $x \sim_G y$ falls es eine Weg von x nach y gibt ist keine Äquivalenz.)

Die Relation $x \equiv_G y$ falls es einen Weg von x nach y und einen Weg von y nach x gibt, *ist* eine Äquivalenzrelation.

Die Äquivalenzklassen heißen *starke Zusammenhangskomponenten* (SCC: strongly connected components)

Bäume

Ein (ungerichteter) *Baum* ist ein

- kreisfreier
- zusammenhängender

Graph.

In einem Baum gibt es zwischen je zwei Knoten genau einen Pfad.

Wenn man einen Knoten als Wurzel auszeichnet, und alle Kanten zur Wurzel hin orientiert (oder alle von der Wurzel weg), erhält man einen *gerichteten* Baum.

Gerichtete kreisfreie Graphen – DAGs

Ein DAG (directed acyclic graph) ist

- ein gerichteter Graph
- ohne gerichteten Kreis.

Satz: G ist DAG $\iff$ jede SCC hat nur ein Element.

DAGs entstehen u. a. beim Modellieren von Abhängigkeiten: $x \to y$, falls (Tätigkeit/Objekt) x eine Voraussetzung für y ist.

Falls ein Abhängigkeitsgraph einen Kreis enthält, sind die Abhängigkeiten nicht „auflösbar".

UML: Aktivitätsdiagramme

ein Aktivitätsdiagramm ist ein gerichteter Graph,

beschreibt dynamisches Verhalten eines (Teil-)Systems (z. B. eines Use Cases)

durch Angabe von *Tätigkeiten* ($=$ *Knoten*)

und ihrer (zeitlichen, unmittelbaren inhaltlichen) *Abhängigkeiten* ($=$ *Kanten*)

$A \longrightarrow B$: nachdem A beendet ist, kann B beginnen.

`http://ivs.cs.uni-magdeburg.de/~dumke/UML/18.htm`

Knoten in Aktivitätsdiagrammen

- (der Normalfall:) Tätigkeiten (in: 1, out: 1)
- Start (in: 0, out: 1), Ende (in: 1, out: 0)
- Entscheidungen
  - *decision:* (in: 1, out: ≥ 1, mit Bedingungen),
  - *merge:* (in: ≥ 1, out: 1).

  (in Rational Rose kein merge)
- Nebenläufigkeit:
  - Gabelung, *fork* (in: 1, out: ≥ 1),
  - Zusammenführung *join* (in: ≥ 1, out: 1).

  (in Rational Rose: *synchronisation bar*)

Determinismus, Synchronisation

- alle Tätigkeiten entlang *eines* Pfades (ohne Entscheidungspunkte) *müssen* ausgeführt werden, und zwar in genau der angegebenen Reihenfolge.

  Beachte: das gilt auch über fork/join hinweg!

- die gegenseitige Reihenfolge der Tätigkeiten in "benachbarten" Pfaden ist nicht vorgeschrieben.

- in jedem Entscheidungspunkt soll es (nach Auswerten aller Bedingungen) *genau eine* Fortsetzung geben

TODO: Übungsaufgaben

Richtlinien für Akt.-Diagramme

`http://www.modelingstyle.info/activityDiagram.html`

- allgemein: Start oben links, Ende nicht vergessen
- Tätigkeiten: keine Schwarzen Löcher, keine Wunder
- Entscheidungen: sollen inhaltlich mit voriger Tätigkeit zusammenhängen, evtl. im Diagramm das Karo weglassen
- Bedingungen (nach Entscheidungspunkt): sollen vollständig sein und nicht überlappen

Schwimmbahnen (swim lanes)

ein Aktivitätsdiagramm mit mehreren Akteuren:

für jeden Akteur eine Schwimmbahn vorsehen, und seine Tätigkeiten dort eintragen.

(sonstige Knoten und Pfeile wie gehabt.)

Aufgaben für 13. Woche

Notation

Ein Aktivitätsdiagramm beschreibt eine Menge (= Sprache) von erlaubten Aktivitätsfolgen (= Wörtern).

Notation: wie für reguläre Ausdrücke (Grundstudium)

- Aktivitäten: Kleinbuchstaben `a, b, c, ..`
- Nacheinander ausführen: `a b c a`
- Decision/Merge (hier *ohne* Bedingungen): `ab + cd`
- Schleife (Decision und Rückweg): hoch Stern: `(ab)^*`
- Fork/Join: Dollar: `ab $ cd`

  Kulturelle Bemerkung: diese Sprach-Operation heißt *shuffle*, Kurzzeichen: sha (der russische Buchstabe).

http://en.wikipedia.org/wiki/Formal_language

Analyse-Aufgaben

Finden Sie jeweils einen regulären Ausdruck (d. h. ohne `$`)
für die so beschriebenen Aktivitäts-Sprachen:

- L1: `ab $ bc`
- L2: `(a $ b) (b $ c)`
- L3: `bc $ a^*`
- L4: `ba^* $ ab^*`
- L5: (Bastelaufgabe) `(ab)^* $ (cd)^*`

Korrektur

automatische Korrektur durch *autotool*.

- anmelden (Bemerkungen zu Test-Email ignorieren) `http://elearning.htwk-leipzig.de/~autotool/cgi-bin/Inter.cgi` (wichtig: … Übungsgruppe wählen!)
- Aufgaben bearbeiten: `http://elearning.htwk-leipzig.de/~autotool/cgi-bin/Face.cgi`
- Highscore: `http://www.imn.htwk-leipzig.de/~autotool/scores`

Klassen (V 29. 3.)

Daten, Typen

Dinge/Aktionen des Problembereiches abbilden auf
„Dinge“/Aktionen innerhalb der Software

- Beziehungen möglichst treu widergeben
- gut strukturierte (leicht wartbare) Software erzeugen

üblich: rechnen mit *Daten*, jedes Datum hat einen *Typ*.
in OO/UML (leider) abweichende Bezeichnungen:
rechnen mit *Objekten*, jedes Objekt gehört zu einer *Klasse*

Arten von Klassen

Klasse modellieren:

- greifbare Dinge oder Stücke aus der Realität
- Rollen
- Ereignisse
- Interaktionen

(nach Stevens/Pooley [3])

Wdhlg: Objekt/Klasse

Ein *Objekt* besitzt

- Attribute ($=$ Objekte oder „einfache“ Daten,

  diese Unterscheidung in Objekt und Nicht-Objekt ist künstlich und wird z. B. in Java-1.5 weiter aufgeweicht)

- Operationen ($=$ Methoden, Unterprogramme)
- Zustand ($=$ Attribut-Werte)
- Identität (realisiert durch Hauptspeicher-Adresse)

Klasse $=$ eine Zusammenfassung gleichartiger Objekte:

- Attribute haben gleichem Namen und Typ
- Operationen sind identisch

Klassen in Diagrammen

(in Rechteck, von oben nach untern, durch Linien getrennt:)

- Name
- Attribute
- Operationen

Attribute nur hinschreiben, falls sie nicht bereits durch Aggregat/Kompositions-Linien deutlich werden.

Es ist möglich, Sichtbarkeiten zu kennzeichnen.

Getter/Setter für Attribute

Geheimnisprinzip: Attribute sollen nicht sichtbar sein,
verwende stattdessen *Operationen*:

- `get`: um Attributwert zu lesen
- `set`: um Attributwert zu setzen

Vorteil: das Objekt „bemerkt“ jeden Zugriff
und kann entsprechend reagieren

Namens-Konvention festlegen und beachten.

```
class A {   Foo get_Foo (); Bar getBar ();
  void set (x : Foo); void set (x : Bar);
}
```

Typinformation im Namen ist eigentlich schlechter Stil
(ist 1. redundant, 2. nicht prüfbar)

Klassen festlegen

Ansatz: jedes *Substantiv* aus Geschäftsprozeß- und Aktivitäts-Beschreibung ist eine Klasse.

Kandidaten für Klassen sind ungeeignet, falls sie

- *redundant* (vervielfacht) sind
- *vage* sind (es ist nicht klar, was die Klasse soll)
- *Ergenisse* oder *Operationen* bezeichnen
- zur *Metasprache* gehören (Bsp: System)
- nicht direkt zum System gehören
- einfache *Attribute* (z. B. Zahlen) sind

(nach: Stevens/Pooley [3])

CRC-Karten

als Entwurfshilfe:

- Klasse (class)
- Zuständigkeiten (responsibilities)
- Mitarbeiter(-Klassen) (collaborators)

Beziehungen zwischen Klassen

grundsätzlich: Klassen A und B sind assoziiert, falls zwischen ihren Objekten ein realer Zusammenhang besteht.

…falls B im Quelltext von A erwähnt wird

(*nicht ausreichend*: falls A und B im Quelltext von C vorkommen)

Assoziation zwischen Klassen A und B

- ein A-Objekt schickt eine Nachricht an ein B-Objekt

  d. h.: in einer A-Methode wird eine B-Methode aufgerufen

- …kann auch ein Konstruktor von B sein

- ein A-Objekt besitzt ein B-Objekt als Attribut

- ein B-Objekt ist Teil einer Nachricht, die von einem A-Objekt gesendet oder empfangen wird

  d. h.: der Typ B kommt in der Deklaration einer A-Methode vor (als Argument oder Resultat)

Benannte Assoziationen

A_{R}——B

bedeutet: es besteht eine Relation $R \subseteq A \times B$

R ist ein *Name*, der die Relation kennzeichnet

R kann auch eine *Rolle* sein, die A (in Bezug auf B) annimmt

Wdhlg.: Relationen

R heißt *Relation* zwischen Mengen X und Y, falls
$R \subseteq X \times Y$.

d. h. R ist eine Menge von geordneten Paaren
$\{(x_1, y_1), \ldots\}$ mit $x_1, \ldots \in X, y_1, \ldots \in Y$.

Umkehr-Relation: $R^- = \{(y, x) \mid (x, y) \in R\}$.

Bilder: $R(x) = \{y \mid (x, y) \in R\}$

Urbilder: $R^-(y)$

Wdhlg: Eigenschaften von Relationen

- R ist *total*: $\forall x \in X : |R(x)| \geq 1$
- R ist *surjektiv* (überdeckend): $\forall y \in Y : |R^{-}(y)| \geq 1$
- R ist Funktion (ist nach-eindeutig): $\forall x \in X : |R(x)| \leq 1$.
- R ist *injektiv* (vor-eindeutig): $\forall y \in Y : |R^{-}(y)| \leq 1$.
- R ist *bijektiv*: R und R^{-} sind totale Funktionen

Kardinalitäten

eine Assoziation in einem UML-Diagramm:

$A \text{———} B$

bezeichnet eine *Relation* R *zwischen Objekten*.

durch Angabe von Bereichen (Mengen) genauere Festlegung:

$A \overset{M}{\text{———}} B$ bedeutet: $\forall a \in A : |R(a)| \in M$

Beispiel: Brett $\overset{361}{\text{————}}$ Punkt bedeutet:

jedes Spielbrett besitzt 361 Schnittpunkte

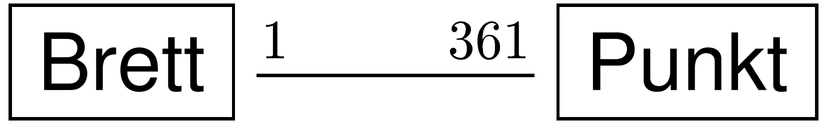
(welches Spiel ist das?)

Abkürzungen für M: Bereiche `0 .. 6`, Stern `5 .. *`

Übung: Diagramme für: R is surjektiv, injektiv, total.

Qualifizierte Assoziation

bisher:



genauere Information durch Qualifizierung:

Beachte: der *qualifier* (Koordinate) ist *kein* Attribut der Klasse (Brett).

mathematisch: Kreuzprodukt:

Komposition und Aggregation

beides: jedes A-Objekt *enthält* ein (oder mehrere) B-Objekte

Komposition ist stärkere Form: B-Objekte können nicht (sinnvoll) allein existieren

Übung: Welche Kardinalitäten sind möglich?

Navigations-Pfeile

$A \to B$, falls B in A *bekannt ist*

- Vorteil: erleichtert (ermöglicht) Programmierung

- Nachteil: bei Wiederverwendung von A muß auch B mitgenommen werden

Übung: in welchen Richtungen können Komposition und Aggregation navigierbar sein? Geben Sie Beispiele.

Aufgaben 14. Woche

Entwurf von Akt.-Diagrammen

Zeichnen Sie Aktivitätsdiagramme zu den Geschäftsprozessen Ihres Projektes.

Hinweis: ggf. weniger/umfassendere Use cases, und Einzelheiten dann in Akt.-Diagramm verschieben.

Beachten: es geht immer noch um *Tätigkeiten* (des Nutzers). Es gibt noch *keine* Daten/Objekte/Klassen.

Entwurf von Klassen-Diagrammen

Finden Sie die für Ihr Projekt benötigten Klassen.

Stellen Sie diese in einem Klassendiagramm dar.

Schnittstellen, Schablonen (V 1. 4.)

Beispiel: Zahlen

Der mathematische Zahlbegriff ist eine *Abstraktion*.

Man kann ihn verschieden *implementieren*.

(kurze ganze Zahlen, beliebig lange ganze Zahlen, exakte Brüche, gerundete Dezimalbrüche, Bitvektoren, . . .)

wichtig ist, daß man mit den implementierten Objekten *rechnen* kann.

(Addition, Negation, Multiplikation, . . .)

Konkrete und Abstrakte Datentypen

Bsp: `double` (nach IEEE-754-Norm) ist ein *konkreter* Datentyp (binäre Gleitkommazahlen mit 53 Bit Mantisse, 11 bit Exponent)

Bsp: (Java) `boolean []` , (Haskell: `[Bool]`) ist ein *konkreter* Datentyp (Array bzw. Liste von Bits)

für jeden dieser Typen kann man *Addition*,
... implementieren.

Zahl ist ein *abstrakter* Datentyp.

Schnittstellen (interfaces)

Ein abstrakter Datentyp wird durch seine Anwender-*Schnittstelle* definiert.

Diese besteht aus *Operationen* mit jeweils:

- Name
- Argument- und Resultat-Typen
- zugesicherten Eigenschaften

die Schnittstelle „Zahlen“

Operationen für Zahlen: $0, 1, (+), (-), (*), (/)$

Zusicherungen: Kommutativ-, Assoziativ-, Distributiv-Gesetze!

Die Mathematiker haben dafür (schon längst) ein sehr genaues Design:

- M mit assoziativem $+$ heißt *Halbgruppe*,
- Halbgruppe mit neutraler 0 heißt *Monoid*,
- M mit $+$ und $*$ heißt *Halb-Ring*, falls
  - $(M, +)$ kommutative Halbgruppe
  - $(M, *)$ Halbgruppe
  - Distributivgesetze gelten

`http://mathworld.wolfram.com/Semiring.html`

Weitere Implementierungen von Zahlen

weitere wichtige „Zahl“bereiche für Informatiker:

- *Wahrheitswerte* `(boolean, ||, &&)` bilden Halbring
- *Sprachen* ($=$ Mengen von Wörtern $=$ Mengen von Folgen von Ereignissen) $(2^{\Sigma^*}, \cup, \cdot)$ bilden Halbring

Vorteil: man kann damit *rechnen*.

Nutzen: bei der Spezifikation von Programm-Verhalten.

Schnittstellen sind Normen

Wichtigkeit von genormten Schnittstellen zwischen (Teil-)Systemen (und Nutzern) ist längst praktisches Allgemeingut.

- Anordnung Kupplung, Bremse, Gas.
- Straßenverkehrs-Regeln
- Durchmesser, Gewindeneigung von Schrauben
- ...

auch bei Software?

Schnittstellen in UML-Diagrammen

- Schnittstelle S wie Klasse, mit `<<interface>>`
- implementierende Klasse I mit „Anschluß-Kreis“(benannt mit S)
- benutzende Klasse N mit „Benutzungs-Pfeil“ zu Anschluß-Kreis von I

  oder, neu in UML-2.0: N hat *socket* benannt mit S

Sprachliche Unterstützung

Schnittstellen als Sprachkonstrukt vorhanden? (Wenn nein, dann durch Programmierstil simulieren.

in Programmiersprachen mit Modulsystem: Exportkontrolle von Bezeichnern.)

In Java realisiert über *abstrakte Klassen*.

- eine nur spezifizierte, aber nicht implementierte Methode heißt `abstract`
- jede Klasse mit *wenigstens* einer abstrakten Methode heißt selbst `abstract`
- ein `interface` ist eine Klasse mit *nur* abstrakten Methoden (und keinen Attributen)

Beispiel: ActionListener

```
interface ActionListener
  { void actionPerformed (ActionEvent e); }
```

```
public class Check extends Applet {
  int s = 0;  Label l = new Label ("foo");
  void change (int d) {
    s = s + d; l.setText (Integer.toString (s)); }
  }
  class AL implements ActionListener {
    int d;
    AL (int x) { d = x; }
    public void actionPerformed
      (ActionEvent e) { change (d); }
  }
  public void init () {
    add (l); l.addActionListener (new AL (1));
  }
}
```

Design-Zeit von Schnittstellen

- *vorher* (design by committee)
  - pro: gibt Teile der Systemstruktur vor
  - contra: falsche Vorgaben später schwer zu ändern
- *hinterher* (prototyping, re-factoring)
  - pro: hinterher ist man immer klüger
  - contra: bis dahin weniger Struktur
- *gar nicht*
  - (weil dann keiner mehr Zeit hat)

De-Facto-Schnittstellen

Eine Schnittstelle sollte tatsächlich *als solche* beschrieben werden.

die oft gewählte Variante *Referenz-Implementierung* ist sinnvolle Ergänzung, aber allein nicht ausreichend.

Bsp: „Die Programmiersprache XYZ ist definiert als genau das, was unser Compiler kompilieren kann.“

Dann muß man ausgehend von der Implementierung raten, was eigentlich gemeint war.

Offene Schnittstellen

Schnittstellen (für Hard- und Software) sollen allgemein zugänglich sein, damit sich ein freier Markt für Implementierungen entwickelt.

- Programmiersprachen (ISO, ANSI)
- Übertragungs-Protokolle (Internet, ...)
- Rechner-Komponenten (Sockel, Bus-Systeme)

- proprietäre Gerätetreiber (Grafik-Karten, Modems)
- proprietäre Dokumenten-Formate (für Texte, Archive)

Schablonen

Wdhlg: *abstrakter Datentyp:* Wörterbuch

($=$ *Abbildung* mit endlichem Definitionsbereich)

Operationen: leeres Buch erzeugen, Wort einfügen, Wort suchen

Übung/Wdhlg: welche Zusicherungen sollen gelten?

Implementierungen: Liste, Hashtabelle, balancierter Baum

Wörterbuch ist aber als Typ zu ungenau. Wörterbuch realisiert Abbildung von T_1 nach T_2, die wollen wir beide angeben.

wir benötigen dazu *Schablonen* (parametrisierte Klassen, generische Klassen, Templates)

Java-1.5: `interface Map<K,V>`

Klassen-Schablonen in UML

- *Schablone*: wie Klasse, aber mit Parameter-Rechteck oben rechts.

- *Instanz*: Klassensymbol (nur Name) mit `<<bind>>`-Pfeil und Belegung der Parameter.

- Parameter können Daten oder Typen sein

  (Beispiel: Größe eines Containers, Inhaltstyp eines Containers)

- die Schablone kann konkrete oder abstrakte (interface) Klassen bezeichnen

typische Verwendung: *Container* (z. B. Vektoren, Wörterbücher, Warteschlangen).

Sequenz-Diagramme (V 15. 4.)

Sequenzdiagramme

stellen zeitliche Abfolge von Interaktionen zwischen Objekten (und evtl. Akteur) dar

- Vertikale: Zeit
- Horizontale: Objekte (Akteur)

Kennzeichnung von Lebensdauer und Kontrollfluß:

- Objekt existiert: Objektlinie (vertikal)
- Objekt ist aktiv (führt Code aus): Balken (vert.)
- Nachricht/Rückkehr: unterschiedl. (horiz.) Pfeile

Erstellen und Löschen von Objekten

- statisches Objekt: Name in Kopfzeile
- dynamisch erzeugtes: Konstruktor-Aufruf zeigt auf Namen, Objektlinie beginnt dort
- Destruktor-Aufruf: zeigt auf Ende (Kreuz) der Objektlinie

Automatische Freigabe

Objekt kann freigegeben werden, sobald es nicht mehr benutzt wird (d. h. keine Methoden aufgerufen, keine Attribute gelesen werden, keine Referenzen auf das Objekt bestehen)

Dieser Zeitpunkt kann bestimmt werden:

- vom Programmierer (durch Destruktor-Aufruf)

  (fehleranfällig, Bsp: C, C++)

- vom Laufzeitsystem (durch Zeiger-Analyse)

  ($=$ garbage collection, evtl. teuer, Bsp: Java, Haskell)

- vom Compiler (durch Programm-Analyse)

  ($=$ „kostenlose“ garbage collection)

Das „Demeter-Gesetz“: *don’t talk to strangers*

propagiert von Karl Lieberherr 1989, siehe

`http://www.ccs.neu.edu/home/lieber/LoD.html`

die Demeter $=$ Göttin der Landwirtschaft, Name eines OO-Projektes von Lieberherr

Objekt O darf als Reaktion auf Nachricht $m(a_1, \ldots, a_n)$
nur Nachrichten diese Objekte senden:

- O selbst,
- Attribut-Objekte von O
- Objekte $a_1, \ldots, a_n$
- Objekte, die in m konstruiert werden

also *nicht* an Attribut-Objekte von a_i,
an Resultat-Objekte von Methodenaufrufen.

Demeter heute?

Die Absicht war, Abhängigkeiten zwischen Implementierungen von Klassen zu reduzieren. *interfaces* sind ein moderneres Mittel zum gleichen Zweck.

Beispiel `http://java.sun.com/j2se/1.5.0/docs/api/`

```
interface List<E>
    { void add(E o); E get(int i); .. }
class ArrayList<E>
     implements List<E> { .. }
List<String> ls =  new ArrayList<String> ();
         // oder:  new LinkedList<String> ();
ls.add ("foo");
System.out.println (ls.get (0));
```

grundsätzlich alle Variablen über Interface deklarieren!

Ergänzung/Zusammenfassung UML

Pakete

fassen Modell-Elemente zusammen.

Zustandsdiagramme

Beschreiben mögliche Zustände und -Übergänge *eines* Objektes.

gerichteter Graph

- Knoten: Zustände (abgerundetes Rechteck, enthält Namen)

  Start- und Endzustand kennzeichnen

- Kanten: Übergänge (beschrifteter Pfeil)

  ausgelöst durch *interne* und *externe* Ereignisse

(Diagramm ähnlich zu Aktivitätsdiagrammen.)

Mathematisches Modell: endlicher Automat (reguläre Sprache, Typ-3-Grammatik, regulärer Ausdruck)

Zustände und Design

Zustandsdiagramme sind nicht für jede Klasse nötig.

Im Gegenteil: weniger Zustand $\rightarrow$ bessere (Wieder-)Verwendbarkeit.

Falls doch Zustand nötig, dann mit Methoden für „gefahrlose“ Speicherung und Wiederherstellung.

Ausweg: benutze separate Objekte, die Zustand und -Änderungen repräsentieren.

Beispiel: Spielfeld-Belegungen, Spielzüge (auch: Liste von $\ldots$)

Checklisten

Heide Balzert: UML kompakt, S. 39 ff. enthält Checklisten für:

Use Case, Paket, Klasse, Assoziation, Attribut, (Vererbung), Kardinalität, Aggregation/Komposition, Sequenz-Diagramm, Zustandsdiagramm, Operation.

Aufgabe (20. 4.)

(Aus Balzert: Softwaretechnik Band 1, S. 426 f.)

- Jeder Lehrer kann bis zu vier Fächer unterrichten.
- Eine Klasse wird von verschiedenen Lehrern in unterschiedlichen Fächern unterrichtet.
- Jeder Klasse ist ein bestimmter Lehrer als Klassenlehrer zugeordnet.
- Jeder Lehrer ist höchstens für eine Klasse der Klassenlehrer.
- Jeder Unterrichtsstunde findet zu einer bestimmten Zeit in einem bestimmten Raum statt und wird von einem Lehrer vor einer Klasse abgehalten.

- Jede Klasse hat zwischen 30 und 35 Unterrichtsstunden.

Aufgaben (zu Modell auf Folie)

- Sind Klassen und Beziehungen korrekt?

  (Benutzen Sie Checklisten aus Balzert: UML kompakt)

- Sind Attribute sinnvoll benannt, ist das Abstraktionsniveau richtig?

Hinweis: ähnliche Aufgabe könnte in Klausur vorkommen.

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. `wincvs`)

Ein Archiv (repository) besteht aus mehreren Modulen.

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.

- wincvs: 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, z. B.

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

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/foo
cvs login
```

- remote, sicher

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

Unser CVS-Server

- Server `teeth.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, und alle Einzelkämpfer zusammen in eine Gruppe

- *wichtig:* Fachschaft sucht Studenten, die Betreuung des Pools übernehmen!

  (Linux-Admin-Kenntnisse vorausgesetzt, Einarbeitung ab sofort, Übernahme ab Herbst)

Übung 27. 4.

- ein CVS-Archiv ansehen (cvsweb-interface) `http://theo1.informatik.uni-leipzig.de/cgi-bin/cvsweb/autotool/`

  Aufgabe: welche Änderungen in Datei htwk/Inter/CASE/Activity/Language.hs?

- einen Teil dieses Archivs anonym auschecken (mit wincvs) (Methode: pserver, User: anonymous, Repository: autotool, Modul: htwk)

- im Repository (je CASE-Projektgruppe) ein neues Modul anlegen (Methode: ssh2 (dort in Settings: `C:\Prog..\Secure..\ssh2.exe` eintragen) Host: teeth, Repository: `/var/lib/cvs/CASE04_nn`)

(Zugang benutzt Account im Linux-Pool)

- diese Modul in eine sandbox auschecken, eine Datei ändern, und commit

CVS – Einzelheiten (V 29. 4.)

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

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.cs.arizona.edu/people/gene/PAPERS/diff.ps`
- Implementierung (Richard Stallman, Paul Eggert et al.): `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss04/case/cvs-1.11.15/` (diff/analyze.c)

  `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss04/case/diffutils-2.8.1/` (src/analyze.c)

LCS

Idee: die beiden Aufgaben sind äquivalent:

- kürzeste Edit-Sequenz finden
- längste gemeinsame Teilfolge (longest common subsequence) finden

Beispiel: $y = AB\boxed{C}\boxed{AB}B\boxed{A}, z = \boxed{C}B\boxed{AB}\boxed{A}C$

für $x = CABA$ gilt $x \le y$ und $x \le z$,

wobei die Relation $\le$ auf Σ^* so definiert ist:

$u \le v$, falls man v aus u durch *Löschen* einiger Buchstaben erhält (jedoch *ohne* die Reihenfolge der übrigen Buchstaben zu ändern)

vgl. mit Ausgabe von `diff`

LCS — naiver Algorithmus (exponentiell)

```
-- | Länge einer längsten gemeinsamen Teilfolge
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)

```
static<E> int lcs (E [] xs, E [] ys) {
  int a [][] = new int [xs.length][ys.length];
  // a[i][j] = lcs (xs [0 .. i], ys [0 .. j])
  for (int i=0; i < xs.length; i++) {
    for (int j=0; j < ys.length; j++) {
    int diag = ( xs[i].equals (ys[j]) ) ? 1 : 0;
    a[i][j] = Math.max (   diag + get (a, i-1, j-1),
                         Math.max (get (a, i-1, j  ),
                                   get (a, i  , j-1)));
    }
  }
  return get (a, xs.length-1, ys.length-1);
}
static int get (int [][] a, int i, int j) {
    return ((i < 0) || (j < 0)) ? 0 : a[i][j];
}
```

LCS – eingeschränkt linear

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

einzelne Kanten verlaufen (siehe Quelltext)

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

  gemeinsamer Buchstabe

Optimierungen:

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

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 ()`

CVS – Merge

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

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

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

Drei-Wege-Diff

benutzt Kommando `diff3` K A H

- changes von $A \to H$ berechnen
- …und auf K anwenden (falls das geht)

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

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

Aufgaben (autotool) zu LCS

`http://elearning.htwk-leipzig.de/~autotool/cgi-bin/Face.cgi`

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

Mehr zu CVS (10. 5.)

Keyword Expansion

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

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

```
$Id: keyword.tex,v 1.2 2004/05/10 08:34:42 waldmann Exp $
$Author: waldmann $
$Date: 2004/05/10 08:34:42 $
$Header: /var/lib/cvs/edu/edu/ss04/case/folien/acht/keyword.tex,v 1.2 2004/05/10 08:34:42 waldmann Exp $
$Name:  $
$RCSfile: keyword.tex,v $
$Revision: 1.2 $
$Source: /var/lib/cvs/edu/edu/ss04/case/folien/acht/keyword.tex,v $
$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 1.2  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: 1.2 $`

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.

Server installieren, Repository anlegen

CVS(-Server) gibt as als Source- und Binär-Paket für alle gängigen Unix-Systeme, siehe

`http://www.cvshome.org`

Unter Debian:

```
apt-get install cvs
```

richtet Server-Verzeichnis `/var/lib/cvs` ein

Repository anlegen mit

```
export CVSROOT=/var/lib/cvs/foobar
cvs init
```

alles weitere (Module anlegen, . . .) durch Clients

Bei Authentifizierung über ssh: Zugriffsrechte passend setzen (gemeinsame Gruppe mit Schreibrechten).

Das Repository

…*ist* ein Verzeichnis auf dem Server.

Jede Repository-Datei `foo.c,v` enthält

- die aktuelle Revision
- (reverse) Diffs zur Herstellung der älteren/anderen Revisionen
- Log-Nachrichten

Das Datei-Format stammt vom System RCS

Aufgabe: selbst nachschauen im Verzeichnis auf teeth

Datei-Operationen

werden von CVS nur sehr schwach unterstützt:

- neue Datei: `cvs add`
- löschen: `cvs delete`

  verschiebt in verstecktes Verzeichnis (attic), behält Revisions-Informationen
- umbenennen: im Client nicht möglich

  (bei delete/add geht History verloren!)

  im Server als normale Datei-Operation möglich

  (aber beim nächsten update wundern sich die Clients)

Subversion

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

- ähnliche Kommandos, aber anderes Modell:
- Client hat Sandbox *und* lokale Kopie des Repositories

  deswegen sind weniger Server-Kontakte nötig
- “commits are atomic” (CVS: commit einer einzelnen Datei ist atomic)
- Versionsnummer bezieht sich auf Repository (nicht auf einzelne Dateien)

  in Sandbox sind Dateien verschiedener Revisionen gestattet

Subversion (II)

- Server speichert Dateien und Zusatz-Informationen in Datenbank (Berkeley DB) (CVS: im Filesystem)

  unterstützt auch Umbenennen usw. mit Bewahrung der History.

- Subversion läuft als standalone-Server oder als Apache2-Modul (benutzt WebDAV)

- Kommandozeilen-Client wie cvs, Grafische Clients (TortoiseSVN), Webfrontends (viewCVS/viewSVN)

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

Übung 11. 5.

- Keyword Expansion ausprobieren

- CVS-Client unter Unix (mit ssh auf Sun-Pool einloggen:)
  checkout, update, commit (Syntax for `bash`:)

```
export CVS_RSH=ssh
export CVSROOT=:ext:student@teeth:/var/lib/cvs/case_XX
cvs checkout YY
export EDITOR=emacs
cvs commit
```

- CVS-Administration (mit ssh auf CVS-Server teeth einloggen:) Inhalt des Repositorys betrachten (`/var/lib/cvs/case_*/`),

  Mail-Benachrichtigung eintragen (benutze Datei `CVSROOT/users`), ausprobieren (bei commit vom

Windows-Pool aus)

Software-Management (V. 13. 5.)

Management: Definition

(nach Balzert: Softwaretechnik, Band II)

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

Management: Aufgaben

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

- Planung
- Organisation
- Personalauswahl
- Leitung
- Kontrolle

Produktivität

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

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

Leistungsmessung?
(Exaktheit vs. Aussagekraft)

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

Aufwandsmessung?

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

Software-Projekt-Eigenschaften

(nach Kent Beck: Extreme Programming)

die vier wichtigen Eigenschaften sind:
Kosten, Zeit, Qualität, Umfang

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

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

zum XP-Ansatz (dazu später mehr) gehört:

Umfang reduzieren! Möglichst schnell einen Prototyp herstellen, der nur die allerwichtigsten Funktionen implementiert.

(20 % des Codes liefert 80 % der Funktionalität)

Qualität

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

Kent Beck: „wenn man auf *höherer* Qualität besteht, werden Projekt häufig *schneller* fertig, oder es wird in einem bestimmten Zeitraum *mehr* erledigt.“

(Bsp: Entwurfsmethoden, Code-Standards, Spezifikationen, Tests)

zu kurz gedacht ist es,

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

Planung

…ist Vorbereitung zukünftigen Handelns:

festlegen, *was, wie, wann, durch wen* zu tun ist.

drei Abstraktions-Ebenen

- Prozeß-Architektur (allgemein)

  welche Typen von Prozeß-Elementen gibt es, welche Schnittstellen haben sie?

- Prozeß-Modell (Firmen- oder Abteilungs-spezifisch)

  eine bestimmte Anordnung von Prozeß-Elementen

- Projektplan (projektspezifisch)

  Inkarnation eines Prozeß-Modells

Prozeß-Elemente

Ein Prozeß-Element ist gekennzeichnet durch

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

Elemente sind verbunden durch

- Übergabe von Produkten (input/output)
- Rückkopplungen

Prozesse und Vorgänge

Jeder Prozeß wird untergliedert in *Vorgänge*.

Ein Vorgang ist in sich abgeschlossene, identifizierbare Aktivität mit

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

  (abgeleitet aus Gesamtkosten-Schätzung des Projektes)

Meilensteine

Vorgänge können zu *Phasen* zusammengefaßt werden.

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

Meilensteine sind

- überprüfbar

  (konkretes (Teil-)Produkt soll vorliegen)

- kurzfristig

  (betrifft Zeitraum von $1 \ldots 4$ 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 polynomial tief, d. h. exponentiell breit)

- 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/`

Vom Mythos des Mann-Monats

Ein naheliegender Ansatz ist es, Probleme durch Zuteilung neuer Ressourcen zu entschärfen.

Das klappt aber besonders bei der Ressource Arbeitskraft *nicht*:

Adding manpower to a late project makes it later.

(Fred Brooks, 197?, The Mythical Man-Month)

Aufgabe zu Projekplanung

aus Balzert: Softwaretechnik, Band II

Gegeben seien folgende Vorgänge:

- Vorgang 1, Aufwand 3 MT, fester Anfang am 21.10.96
- Vorgang 2, Aufwand 20 MT
- Vorgang 3, Aufwand 15 MT
- Vorgang 4, Aufwand 5 MT, festes Ende am 13.12.96

Zwischen den Vorgängen existieren folgende Abhängigkeiten:

- /2/ kann sofort nach Ende von /1/ beginnen
- /3/ kann erst 5 Tage nach Ende von /1/ beginnen
- /4/ kann erst beginnen, wenn /3/ beendet ist
- /4/ kann frühestens 5 Tage vor dem Ende von /2/ beginnen

- 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 1000 DM pro Werktag und 1500 DM an Sonn- und Feiertagen. Eine Verzögerung des Endtermins kostet 1000 DM Konventionalstrafe pro Tag. Welches ist für den

Auftragnehmer die kostengünstigste Lösung?

Prozeß-Modelle (V 24. 5.)

Aufgaben

(Balzert, Softwaretechnik, Bd 2, LE 4)

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

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) $\rightarrow$ 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

(*) 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: woher kommt der Spruch: *release early, release often*

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

Inkrementelles Modell

zunächst Anforderungen möglichst vollständig erfaßt und modelliert

weiter wie bei evolutionärem Modell

Objektorientiertes Modell

Schwerpunkt ist Wiederverwendung von

- Analyse-Modellen
- Design-Modellen
- Implementierungen

Objektorientierung ist eine (nicht die beste) Methode, polymorphen Code zu schreiben

polymorph (viel-förmig): gleicher Code, unterschiedliche Anwendungen (Typen)

angemessene Methoden sind:

- Schnittstellen (interfaces, abstrakte Klassen)
- Generische Typen, Funktionen (templates)

Nebenläufiges Modell

allem wesentlichen Entwicklungsschritte gleichzeitig beginnen

- frühes Erkennen und Eliminieren von Problemen
- optimale Zeitausnutzung
- Gefahr, daß wichtige Entscheidungen zu spät (oder nie) getroffen werden
- hoher Planungs- und Personalaufwand

Das Spiral-Modell

(ist Meta-Modell) zerlege in Teilprodukte, für jedes:

- Ziele festlegen, Randbedingungen beschreiben, Alternativen suchen
- Alternativen bewerten, Risiken abschätzen
- Prozeßmodell festlegen
- nächsten Zyklus planen

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 (V 7. 6.)

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 8. 6.

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

- das lokale Bugzilla-System ausprobieren: eigener Account, vorgegebene Testprodukt/Testkomponente

- Bugzilla verwalten: für eigenes Software-Projekt eigene Produkt/Komponente anlegen

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 ≤ 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

  Abschwächung: jede Schleife (interior) höchstens einmal
- Bedingungs-Überdeckung: jede atomate 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/`

Aufgaben:

- Kompilieren, ausführen, profilieren:

  ```
  g++ -ftest-coverage -fprofile-arcs heap.cc -o heap
  ./heap > /dev/null
  gcov heap.cc
  ```

  erzeugt `heap.cc.gcov`

  Optionen `-b` und `-c` für `gcov` ausprobieren!

- `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:

Kontrakt (Vertrag):

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

  Nachbedingungen (an Programmzustand, Resultat) erfüllt

eingebaut 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}`

Benutze Schlußregeln und Axiome

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 (schnelles Potenzieren)

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

Beweise `power (b,e) = b^e`

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:

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

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);
  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

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)

- Workshop on Termination `http://www-i2.informatik.rwth-aachen.de/WST04/`
- Matchbox `http://theo1.informatik.uni-leipzig.de/matchbox/`
- RTA list of open problems `http://www.lsv.ens-cachan.fr/rtaloop/problems/21.html`

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. Petermann)

Aufgaben zum Testen und Verifizieren (22. 6.)

Testen mit JUnit, Beispiel: Median5

- JUnit tiefladen (`http://junit.org/`), auspacken und `junit.jar` ins aktuelle Directory kopieren
- Quelltext: `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss04/case/programme/analyze/Median5.java`

  Median5 kompilieren und ausführen (Option `-cp junit.jar`)
- die anderen TestRunner ausprobieren (AWT, Swing)
- in die Methode median5 Aussagen über den erwarteten Programmzustand (nach jedem if) einfügen (assertTrue), jeweils testen

ggf. Programm korrigieren/ergänzen

- dann für jedes if: die Aussage (Zusammenhang von Vor- und Nachbedingung) beweisen

Prozeßqualität

CMM/SPICE

capability maturity model

- initial/chaotisch
- repeatable/intuitiv
- defined/qualitativ
- managed/quantitativ
- optimizing/rückgekop-pelt

software process improvement and capability determination

- incomplete/unvollständig
- performed/durchgeführt
- managed/gesteuert
- established/etabliert
- predictable/vorhersagbar
- optimizing/optimierend

Aufgabe: übertragen Sie diese Stufen auf das Erlernen eines Musikinstruments, einer Sportart, einer Sprache.

Spezifikation von Jongliermustern

Muster: Abbildung $m : \mathbb{Z} \to \mathbb{N}$

Bedeutung:

- $t =$ Abwurf-Zeitpunkt,
- $t + d(t) =$ Lande-Zeitpunkt

  genauer: nächster Abwurf-Zeitpunkt des gleichen Balls

Einfache Muster: $3, 3, 3, \ldots$, usw.

Was bedeuten gerade/ungerade Zahlen?

Was bedeuten $0, 1, 2$?

Zeit und Höhe

Muster mit verschiedenen Zeiten:

$\overline{5,3,4} = \ldots 5,3,4,5,3,4,5,3,4,\ldots$

Zahl d im Muster: Ball ist etwa $(d-1)$ Zeit-Einheiten in der Luft, 1 Zeit-Einheit in der Hand.

Flug-Höhen hängen quadratisch von den Flug-Zeiten ab:

d	3	4	5	6	7
$d-1$	2	3	4	5	6
$(d-1)^2$	4	9	16	25	36
Höhe in m	0.5	1	2	3	4

Transformationen

swap: zwei Bälle

Invariante: Quersumme

addiere/subtrahiere Periode

addiere/subtrahiere eins

Links

- Theorie:

  Joe Buhler, David Eisenbud, Ron Graham and Colin Wright: Juggling drops and descents, American Mathematical Monthly 101, (no. 6) 1994, 507 - 519
  `http://www.cecm.sfu.ca/organics/papers/buhler/`

- Applet:

  `http://jugglinglab.sourceforge.net/`

- Hilfe:

  `http://www.juggling.org/help/siteswap/`

- European Juggling Association: `http://eja.net/`

- richtige Bälle: `http://fergieprops.com/`
- richtige Keulen: `http://www.albapasser.de/`
- in Leipzig:
  `http://www.uni-leipzig.de/~jonglier/`