Softwaretechnik I
Vorlesung
Wintersemster 2011

Johannes Waldmann, HTWK Leipzig

1. Februar 2012

Gegenstand und Ziel

Modelle und Methoden der Softwareproduktion
Anwendung in

- Mini-Projekt (Verlauf dieses Semesters)
- Softwarepraktikum (4. Semester)
- Betriebspraktikum, Bachelor-Arbeit, Beruf, ...

Plan

Abschnitte der Vorlesung (ST1 + ST2) $\approx$
Schritte eines Softwareprojektes

- ST1
  - Aufnahme der Anforderungen
  - Entwurf einer Lösung
  - Ressourcenabschätzung, -Zuordnung (Mitarbeiter, Zeit)
- ST2 ($\approx$ VL Deklarative Programmierung lt. Studienordnung ab WS10 $\approx$ VL Fortgeschrittene Programmierung ab nächster StudO)
  - Realisierung der Lösung (fortgeschr. Methoden zur Implementierung)
  - Qualitätssicherung, -Verbesserung (Refactoring)

Organisation

- pro Woche eine Vorlesung (Di 12:00), eine Übung
  - Mi (u) 13:45 + Mi (g) 9:30
  - Mi (u+g) 11:15
  - Do (u+g) 9:30
  - Do (u+g) 11:15
- Prüfungszulassung:
  - regelmäßiges und erfolgreiches Bearbeiten von Übungsaufgaben (beginnt fast sofort)
  - darunter ein Mini-Projekt (wird später bekanntgegeben)
- Prüfung: Klausur (120 min, keine Hilfsmittel)

Material

- Skript `http://www.imn.htwk-leipzig.de/~waldmann/edu/ws11/st/folien/main/`
- Skript von Prof. Weicker, erhältlich bei Fachschaftsrat und `http://www.imn.htwk-leipzig.de/~waldmann/edu/ws11/st/weicker/`
- ... sowie dort angegebene Literatur, z. B.
  Uwe Kastens, Hans Kleine Büning: *Modellierung*, Hanser 2008. `http://www.hanser.de/buch.asp?isbn=978-3-446-41537-9`
- Online-Übungsaufgaben `https://autotool.imn.htwk-leipzig.de/cgi-bin/Super.cgi`
  dort auch Wahl der Übungsgruppe

Beziehungen zu anderen LV

- Diskrete Mathematik und Logik: Spezifikation von
  - Anforderungen (gewünschtes Systemverhalten)
  - Lösungen (geplante Systemstruktur)
- Grundlagen der Informatik
  Algorithmenbegriff, Berechenbarkeit
- Grundlagen der Programmierung
  Realisierung von Algorithmen in Programmen
- Algorithmen und Datenstrukturen
  Entwurf und Analyse von effizienten Algorithmen
- Theoretische Informatik
  Analyse der Komplexität von Problemen

Schrittweise Verfeinerung

- Absicht: auf sichere Weise von der Spezifikation zur Implementierung
- Methode: nicht auf einmal, sondern schrittweise.
  jeder Schritt entspricht der Unterteilung einer Komponente in Sub-Komponenten und der Spezifikation ihres Zusammenwirkens (ihrer Schnittstellen)
- schließlich sind die Komponenten so klein,
  daß ihre Implementierung trivial ist
- und damit ist das Gesamtsystem korrekt.

Das Festlegen von Schnittstellen ist der kreative Akt!
(nicht das Hacken von Programmen, in der vagen Hoffnung, daß sie was sinnvolles tun)

Spezifikation von Komponenten

- zustandslose (statische) Komponente:
  Operation ist Funktion: Eingabedaten $\to$ Ausgabedaten
- zustandsbehaftete (dynamische) Komponente:
  Operation is Aktion (Zustandsänderung),
  d. h. Funktion: Startzustand $\to$ Finalzustand

Beispiel Spezifikation (statisch)

konfliktfreie Graphenfärbung

Fragen bei der Spezifikation:

- was ist ein Graph?
- was ist eine Färbung?
- wann ist diese konfliktfrei?

Beispiel Spezifikation (dynamisch)

Kaffee-Automat

- Geld rein
- Wahl des Getränks
- Getränk raus

Fragen bei der Spezifikation:

- was ist die Zustandsmenge?
- welches sind die Aktionen?
- welche Aktionen sind erlaubt?

Mathematische Hilfsmittel

- Objekte (Daten):
  - Mengen
  - Relationen, Funktionen
  - Strukturen (Graphen, Bäume)
- Zustandsübergangssysteme:
  - (endliche) Automaten
  - Petri-Netze
- Eigenschaften:
  - Aussagenlogik
  - Prädikatenlogik

Übungen

- EWD 1305
- Modellierung 8-Damen-Problem

Vorgehensmodell Wasserfall

siehe Skript Weicker Kapitel 2

- Aktivitäten
- Schichten/Sichten
- Klammer-Struktur
- Dokumente

Übung:

- für jede Aktivität: Beispiel für Fehler, der dabei auftreten könnte. Wann wird dieser bemerkt?
- Planspiel: Anforderungsanalyse für Konferenzverwaltung

Modellierung mit Mengen und Relationen

Motivation, Plan (I)

(diese Vorlesung)

- Struktur von Objekten
  - Konstruktion von Objekt-Typen: Kreuzprodukt, Vereinigung, Folge
  - Beziehungen zwischen Objekten (Relationen, Funktionen)
  - ER-Diagramme, Objekt- und Klassendiagramme

Bemerkung: Mathematik ist aus dem 1. Semester bekannt!

Literatur: Kastens, Kleine Büning: *Modellierung*, Kap. 2

Motivation, Plan (II)

(folgende Vorlesungen)

- konkrete und abstrakte Datentypen (KKB: Kap. 3, 4)

  (prädikatenlogische Formeln und ihre Modelle)
  - (Signatur, Axiome, Algebra) (interface, ?, class)
  - Eigenschaften von Relationen (Graphen)
- Modellierung von Abläufen (KKB: Kap. 7)
  - zentraler Zustand (endliche Automaten)
  - verteilter Zustand (Petri-Netze)
  - UML: Sequenz- und Zustandsdiagramme

Mengen

- zur Beschreibung von Wertebereichen

  für Daten in: Eingabe, Rechnungsverlauf, Ausgabe
- Eine Menge ist eine Zusammenfassung von verschiedenen Objekten.

  Notation: $O \in M$: Objekt O ist Element der Menge M
- Angabe von Mengen
  - extensional (durch Hinschreiben ihrer Elemente)
    $A = \{1, 2, 4, 8\}$ Spezialfall $B = \{\ \} = \emptyset$
  - intensional (durch Angabe eines Grundbereiches und einer Bedingung)
    $A = \{a \mid a \in \mathbb{N} \wedge \exists b \in \mathbb{N} : b < 4 \wedge a = 2^b\}$

Eigenschaften von Mengen

- A ist Teilmenge von B, Notation $A \subseteq B$

  Definition $\forall x : (x \in A) \to (x \in B)$
- A ist echte Teilmenge von B, Notation $A \subset B$

  Definition $A \subseteq B \wedge B \not\subseteq A$
- A und B sind gleich, Notation $A = B$

  Definition $A \subseteq B \wedge B \subseteq A$

Übungen:

- bestimme $M =$ die Menge aller Teilmengen von $\{1, 2, 3\}$
- die Relation $\subseteq$ auf M ist eine Halbordnung, aber keine Ordnung ($=$ lineare Halbordnung)

Die Potenzmenge einer Menge

Die Potenzmenge einer Menge M
ist die Menge aller Teilmengen von M

$$\mathrm{Pow}(M) = \{N \mid N \subseteq M\}$$

Übungen

- Anzahl der Elemente von $\mathrm{Pow}(\{1, 2, 3\})$
- $\mathrm{Pow}(\emptyset)$
- $\mathrm{Pow}(\mathrm{Pow}(\emptyset))$
- $\mathrm{Pow}(\mathrm{Pow}(\mathrm{Pow}(\emptyset)))$
- $M_0 = \emptyset, M_1 = M_0 \cup \{M_0\}, M_2 = M_1 \cup \{M_1\}, \dots$

Operationen auf Mengen

- Vereinigung $A \cup B := \{x \mid \dots\}$
- Durchschnitt $A \cap B$
- Differenz $A \setminus B$

Übungen:

- stelle $A \cap B$ mittels $\cup$ und $\setminus$ dar
- für jede Menge M: die Struktur $(\mathrm{Pow}(M), \cup, \dots, \cap, \dots)$ ist ein Halbring
- $(\mathbb{N}, +, \dots, \cdot, \dots)$ ist ebenfalls ein Halbring— durch welche Formel kann man beide Strukturen unterscheiden?

Vereinigungen und interface/class

Mathematik: $I = A \cup B$
kann abgebildet werden als

```
interface I { }

class A implements I { }
class B implements I { }
```

- damit man Objekte vom deklarierten Typ I verarbeiten kann, sollte I Methoden enthalten
- das *Verhalten* (die Methoden) von Objekten beschreiben wir später, momentan interessiert uns die *Struktur*.

Das Kreuzprodukt

Das Kreuzprodukt von zwei Mengen A, B
ist die Menge aller *Paare*:

$$A \times B := \{(a, b) \mid a \in A \wedge b \in B\}$$

Beispiel

- Farben $= \{$Eichel, Grün, Rot, Schell$\}$,
- Werte $= \{7, 8, 9, 10,$ Unter, Ober, König, As$\}$,
- Karten $=$ Farbe $\times$ Wert,
- $($Grün$, 9) \in$ Karten.

Kreuzprodukte und Records

Das Kreuzprodukt von n Mengen $A_1, \dots, A_n$
ist die Menge aller *Tupel*:
$A_1 \times \ldots \times A_n := \{(a_1, \dots, a_n) \mid a_1 \in A_1 \wedge \ldots \wedge a_n \in A_n\}$

Kreuzprodukt in der Mathematik entspricht *struct* in C, *record* in Pascal, *class* in Java usw.

```
class P { final A a; final B b; final C c; }
```

Komponenten-Zugriff über *Namen* `P x; ... x.a ...`
(Mathematik: über Position).

positionelle Notation bei Aufruf des geeignet definierten Konstruktors

```
P x = new P(new A(), new B(), new C());
```

Relationen

- Eine n-stellige Relation R

  mit den Grundbereichen (Mengen) $A_1, \dots, A_n$

  ist eine Menge von Tupeln

  $R \subseteq A_1 \times \ldots \times A_n$
- Notation: $(a_1, \dots, a_n) \in R$

  oft auch als $R(a_1, \dots, a_n)$
- Notation für zweistellige Relationen: $a_1 R a_2$

  nur in Spezialfällen zu empfehlen, Bsp. $a_1 < a_2$

Relationen und Graphen

eine zweistellige Relation $R \subseteq A \times B$ ist ein Graph
$G = (V, E)$

- Knoten $V = A \cup B$
- Kanten $E = R$

beachte:

- Kanten sind gerichtet
- Schlingen sind möglich (falls $A = B$ und $(x, x) \in R$)

Übung

- zeichne Graph der Relation auf $A = \{0, 1, \dots, 6\}$

  $R = \{(x, z) \mid \exists y \in \mathbb{N} : x \cdot y = z\}$

Eigenschaften von zweistelligen Relationen

- für Relationen $R \subseteq A \times B$
  - Vorbereich (Definitionsbereich) $\operatorname{dom}(R)$
    Nachbereich (Wertebereich) $\operatorname{rng}(R)$
  - injektiv, surjektiv
- für Relationen $R \subseteq A \times A$
  - reflexiv, transitiv, symmetrisch, antisymmetrisch
  - Äquivalenz-Relation, Halbordnung, lineare Ordnung

Übungen: Begriffe und Eigenschaften im Graphen veranschaulichen

Operationen auf Relationen

- weil Relationen Mengen sind, stehen die *Mengen-Operationen* zur Verfügung
- das *Spiegelbild* einer zweistelligen Relation:

  $\overline{R} = \{(b,a) \mid (a,b) \in R\}$
- das *Produkt* von zwei zweistelligen Relationen:

  $R \subseteq A \times B, S \subseteq B \times C, (R \circ S) \subseteq A \times C$

  $R \circ S := \{(x,z) \mid \exists y \in B : (x,y) \in R \wedge (y,z) \in S\}$

Übungen: „punktfreie“ Notation von Eigenschaften

- R symmetrisch $\iff R = \overline{R}$,
- R transitiv $\iff \dots$

Graphen und Hyper-Graphen

- (gerichteter) Graph: $G = (V, E)$ mit $E \subseteq V \times V$
- Verallgemeinerung auf mehrstellige Relationen:

  Hypergraph $G = (V, E)$ mit $E =$ Menge von Hyper-Kanten, $E \subseteq V \times \ldots \times V$

  eine Hyper-Kante ist ein Tupel.

Übung: $V = \{1,2,3,4,5\}$, $E = \{(x,y,z) \mid x < y < z\}$

- Zeichnung von Hypergraphen als Graphen:

  durch zusätzliche Knoten, die den Tupeln entsprechen, und zusätzliche Kanten zu Tupel-Komponenten

Objekt- und andere Diagramme

Objektdiagramm $=$ Folge von (Hyper)graphen mit gleicher Knotenmenge, $(V, R_1, \ldots, R_n)$

- Knoten sind Objekte,
- (Hyper-)Kanten sind Tupel, die zu Relation gehören
  - Name der Relation steht an der Kante
  - oder: Name der Relation steht in Hilfsknoten

beschreibt Objekte und ihre Beziehung in *einem* Zustand des Softwaresystems

ER (Entity-Relationship)- und Klassendiagramm $=$ Spezifikation einer Menge von Objekt-Diagrammen ($=$ Menge von gewünschten Systemzuständen)

ER-Diagramme (Syntax)

ER-Diagramm ist Folge von Hypergraphen $(T, S_1, \ldots, S_n)$, gezeichnet als Graph,

- Knoten:
  - Typ-Namen (entities) (Rechteck)
  - Relations-Namen (relationships) (Raute)
- Kanten (beschriftet mit *Rolle*)
  - von Typ zu Typ
  - von Relation zu Typ
- $\forall i : |S_i| = 1$

Attribut-Knoten (Ellipse) als Notation für Kante mit vergessenem Zieltyp

ER-Diagramme (Semantik)

Objekt-Diagramm (Hypergraph) $(O, R_1, \ldots, R_n)$, *paßt zu* ER-Diagramm (Hypergraph) $(T, S_1, \ldots, S_n)$, falls:

- es gibt Typ-Abbildung $t : O \to T$,
- so daß jedes Tupel aus jeder Relation (im Objektiagramm) den richtigen Typ (im ER-Diagramm) hat:

  wenn $R_i(x_1, \ldots, x_k)$, dann $S_i(t(x_1), \ldots, t(x_k))$.

(die Relation R_i hat den deklarierten Typ S_i, vgl. Deklarationen von Methoden in Java)

ER-Diagramme (Semantik, Anzahlen)

an den Kanten im ER-Diagramm können Mengen von Zahlen notiert werden (z. B. $\{1,2,3\}$ und abkürzende Schreibweisen)
um die „paßt-zu“-Relation zu verschärfen:
Objekt-Diagramm $(O, R_1, \ldots)$ paßt zu ER-Diagramm $(T, S_1, \ldots)$, falls … und

- für jede Hyperkante $S_i(y_1, \ldots, y_k)$ mit Annotation A an Position j:

  zu jedem $x_j \in t^-(y_j)$: die Anzahl der Tupel in R_i, deren $j-$te Komponente x_j ist, ist Element von A.

Klassendiagramme

sind im wesentlichen nur eine andere Syntax für ER-Diagramme (Übung: Unterschiede),
Semantik stimmt überein, außer

- für jede Hyperkante $S_i(y_1, \ldots, y_k)$ mit Annotation A an Position j:

  Für jedes $x \in t^-(y_1) \times \ldots \times t^-(y_k)$:

  die Anzahl der Tupel $x' = (x'_1, \ldots, x'_k) \in R_i$, die mit x in allen Komponenten $\neq i$ übereinstimmen,

  ist Element von A.

Übung: Unterschied zu ER-Semantik (wird erst bei 3- und mehrstelligen Relationen deutlich)

Logik (Wiederholung)

Aussagenlogik (Syntax)

aussagenlogische Formel ist

- Konstante (Wahr, Falsch)
- oder Variable ($p, q, \ldots \in V$)
- oder zusammengesetzte Formel:

  $\neg F_1, F_1 \wedge F_2, F_1 \vee F_2, \ldots$

Übung: definiere

- Größe einer Formel
- Menge der Variablen einer Formel

Aussagenlogik (Semantik)

Belegung ist Abbildung $b : V \to \{0, 1\}$
Wert einer Formel F unter einer Belegung b:

- wert(Wahr$, b) = 1,$ wert(Falsch$, b) = 0,$
- für $v \in V$: wert$(v, b) = b(v)$,
- wert$(F_1 \vee F_2, b) = \max($wert$(F_1, b),$ wert$(F_2, b))$
- wert$(F_1 \wedge F_2, b) = \ldots$

Notation: $b \models F$ für: wert$(b, F) = 1$

Aussagenlogik (Eigenschaften)

Eine Formel F heißt

- allgemeingültig,

  wenn für jede Belegung b gilt: wert$(F, b) = 1$
- erfüllbar,

  wenn eine Belegung b existiert: wert$(F, b) = 1$

Die *Modellmenge* von F ist $\mathrm{Mod}(F) = \{b \mid b \models F\}$.
F erfüllbar $\iff \mathrm{Mod}(F) \neq \emptyset$

Prädikatenlogik (Signatur)

Eine *Signatur* besteht aus

- einer Menge von Funktions-Symbolen
- und einer Menge von Prädikat-Symbolen,

jeweils mit Stelligkeiten (einsortige Signatur) oder Typen (mehrsortige Signatur).
Beispiele:

- Signatur der *Gruppen*: Funktionssymbole: f 2-stellig, i 1-stellig, e 0-stellig, Relationssymbol „=“ 2-stellig
- Signatur der *Halbordnungen*: keine Funktionssymbole, Relationssymbole R und „=“ 2-stellig

Prädikatenlogik (Terme)

Ein *Term* in einer Signatur ist

- eine Variable
- oder ein Funktionssymbol mit einer passenden Anzahl von Argumenten ($=$ Termen)

Übungen:

- Beispiele für Terme in Signature der Gruppen, der Halbordnungen?
- Größe und Tiefe eines Terms,
- Menge der Variablen eines Terms

Prädikatenlogik (Formeln)

Eine Formel in einer Signatur ist

- ein Prädikatsymbol mit einer passenden Anzahl von Argumenten ($=$ Termen)
- oder eine aussagenlogischer Operator mit einer passenden Anzahl von Argumenten ($=$ Formeln)
- oder ein Quantor mit einer Variablen und einem Argument ($=$ Formel)

Quantor steht (im Baum) *über* der Formel, auf die er wirkt.
Deswegen: vor Formel schreiben und ggf. klammern.
Schreibweise am Ende der Formel ist irreführend:
$F(x), \forall x$ vgl. `x = 3.14 ; double x;`

Prädikatenlogik (Bindungen)

- Jede Formel hat eine Baumstruktur (vgl. Ausgabe autotool)
- Ein Vorkommen einer Variablen x heißt *gebunden*, falls sich auf dem Pfad vom Vorkommen zur Wurzel ein Quantor befindet, der x bindet.
- (…sonst heißt das Vorkommen *frei*)

Übung:

- Menge der gebundenen Variablen einer Formel,
- Menge der freien Variablen einer Formel

Prädikatenlogik (Strukturen)

Eine Struktur zu einer Signatur besteht aus

- einem Grundbereich (Universum) U
- einer Zuordnung: k-stelliges Funktionssymbol $\to$ Funktion $U^k \to U$
- einer Zuordnung: k-stelliges Relationssymbol $\to$ Teilmenge von U^k
- einer Belegung (Abbildung Variable $\to$ Universum)

Beachte: üblicherweise ist vorgeschrieben, daß dem Relationssymbol „=“die tatsächliche Gleichheit in U zugeordnet wird.

Prädikatenlogik (Semantik - Terme)

Wert eines Termes in einer Struktur, unter einer Belegung, ist ein Element des Universums

- Variable: benutze Belegung
- Funktionssymbol mit Argumenten: wende Interpretation der Funktion auf Werte der Argumente an

Prädikatenlogik (Semantik - Formeln)

Wert einer Formel in einer Struktur, unter einer Belegung, ist ein Wahrheitswert

- Prädikatsymbol mit Argumenten: wende Interpretation des Prädikatsymbols auf Werte der Argumente an
- aussagenlogische Verknüpfung (wie bei Aussagenlogik)
- Quantoren:
  wert($\forall x.F, S, b) = \min\{$wert$(F, S, b[x := u]) \mid u \in U\}$

wobei $b[x := u]$ die Belegung b' ist mit:
$b'(y) = (\text{wenn } y = x \text{ dann } u \text{ sonst } b(y))$.

Quantoren

Der All-Quantor (über einem endlichen Bereich) entspricht einem logischen „und“.

- $\forall x \in \{0, 1, 2, 3\} : x^2 < 10$
- $0^2 < 10 \wedge 1^2 < 10 \wedge 2^2 < 10 \wedge 3^2 < 10$
- Realisierung in C#

```
using System.Linq;
Enumerable.Range(0,3).All(x  =>  x*x < 10)
```

– lokale Funktion, Typinferenz
– Funktion höherer Ordnung, extension method

(Existenz-Quantor: logisches „oder“, C#: `Any`)

Abstrakte und Konkrete Datentypen

- abstrakter Datentyp:
  Signatur und Axiome (= Formel)
- konkreter Datentyp:
  zur Signature passende Struktur,
  die die Axiome erfüllt.

Beispiele:

- Gruppe , $(\mathbb{Z}, +, (x \mapsto -x), 0), (\mathbb{R} \setminus \{0\}, \cdot, (x \mapsto x^{-1}), 1)$
- `Set<E>` , `TreeSet<E>` (Axiome?)
- allgemein: Spezifikation (Schnittstelle), Implementierung

Modellierung mit Funktionen

Funktionen

- Relation $R \subseteq A_1 \times \ldots \times A_k$
  heißt an der Stelle i (mit $1 \le i \le k$) *eindeutig*, wenn für jedes $x \in A_i$:
  R enthält höchstens ein Tupel t mit $t_i = x$.
- zweistellige Relation R, die an der ersten Stelle eindeutig ist (*voreindeutig*),
  heißt (partielle) *Funktion* (aus A_1 in A_2).

Eigenschaften von Funktionen

- Definitionsbereich $\mathrm{dom}(R) = \{x \mid \exists y : (x, y) \in R\}$
- Wertebereich $\mathrm{rng}(R) = \{y \mid \exists x : (x, y) \in R\}$
- F injektiv : $\iff$ F ist *nacheindeutig*
- F ist (totale) Funktion (von A in B): ...
- F ist surjektiv (aus A auf B): ...
- F ist bijektiv: ...

Die Potenzschreibweise

- f ist eine Funktion von A in B:
  $f : A \to B$
- die Menge der Funktionen von A in B:
  $B^A = \{f \mid f : A \to B\}$
  (Basis = Wertebereich, Exponent = Definitionsbereich)
  für endliche Mengen gilt $|B^A| = |B|^{|A|}$
- Bijektion zwischen k-fachem Kreuzprodukt: A^k
  und A^K mit $|K| = k$
- Bijektion zwischen Potenzmenge von M
  und 2^M, wobei $2 = \{0, 1\}$ (Menge mit 2 Elementen)

Anzahlvergleich von Mengen

- Mengen A und B sind *gleichmächtig*, Notation $A \sim B$,
  falls eine Bijektion von A auf B existiert.
- Menge A heißt *unendlich*: $\exists B \subseteq A : (B \ne A) \wedge (B \sim A)$.
  Beispiel: $\mathbb{N}$ ist unendlich.
- Menge A heißt *abzählbar*, falls A endlich oder $A \sim \mathbb{N}$
  Bsp: Die Menge aller Programmtexte ist abzählbar.
- $\mathbb{N} \sim \mathbb{N}^2$ (konstruiere Bijektion durch *dovetailing*)
- $\mathbb{N} \not\sim 2^{\mathbb{N}}$ (Beweis: Cantors Diagonal-Argument)
  Folgerung: es gibt (sehr viele) Funktionen $f : \mathbb{N} \to \{0, 1\}$, die nicht berechenbar sind.

Modellierung mit Funktionen

Beispiel: Minimum Open-Shop Scheduling
`http://www.nada.kth.se/~viggo/wwwcompendium/node190.html`
beachte die Benutzung von

- Funktionen
- Mengen
- Logik (Quantoren)

in dieser Spezifikation.

Testklausur KW 46

Logik

für die Formel
$P(z()) \to ((\forall x : (P(x) \to P(s(x)))) \to (\forall x : P(x)))$

- zeichnen Sie den Formelbaum,

  markieren Sie für jeden Teilbaum, ob er Term oder Formel ist
- geben Sie die Signatur an
- geben Sie je eine Struktur mit einem Universum der Größe 2 an, in der die Formel
  - wahr ist,
  - falsch ist

Logik/Klassen

$P(x) : x$ ist Person, $C(x) : x$ ist Compiler, $S(x) : x$ ist Sprache, $K(x, y)$: Person x kennt Sprache y.
schreiben Sie prädikatenlogisch (ggf. Signatur erweitern) und als Klassendiagramm (soweit möglich)

- nicht alle Personen kennen eine Programmiersprache
- es gibt Personen, die zwei Sprachen kennen
- jeder Compiler übersetzt eine Quellsprache in eine Zielsprache
- der Autor jedes Compilers kennt dessen Quell- und Zielsprache
- wer Haskell kennt, kennt jede Sprache
- zu jeder Sprache gibt es einen Compiler, der Maschinencode erzeugt

Modellierung

ausgehend von Ihren Erfahrungen als Benutzer:
Geben Sie das Datenmodell an, das im *autotool* verwendet wird.

- Klassen (Entitäten)
- Relationen
- ggf. Einschränkungen (Anzahl-Constraints)

Modellierung von Zustandsübergängen

. . . durch Graphen

gerichteter Graph $G = (V, E)$, d. h. $E \subseteq V^2$

- V als Menge der Zustände eines Systems
- E als Menge der Zustandsübergänge

Beispiel: Missionare und Kannibalen.
Zustandsmenge $V \subseteq \{0, 1, 2, 3\}^6$ mit
$(L_M, L_K, B_M, B_K, R_M, R_K) \in V \iff \dots$
Zustandsübergänge:
$((L_M, L_K, B_M, B_K, R_M, R_K), (L'_M, L'_K, B'_M, B'_K, R'_M, R'_K)) \in E \iff \dots$

Wege in Graphen

gerichteter Graph $G = (V, E)$, d. h. $E \subseteq V^2$

- Knotenfolge $[v_0, v_1, \dots, v_n]$ heißt *Weg* in G, falls $\forall 0 \le i < n : (v_i, v_{i+1}) \in E$.
- Die *Länge* diese Weges ist n (= die Anzahl der Kanten)
- es existiert Weg der Länge ≥ 1 von x nach y in G
  $\iff (x, y) \in E^+$ (das heißt $\exists n : n \ge 1 \land (x, y) \in E^n$)

Kreise (Zyklen) in Graphen

- Der Weg heißt *Zyklus* (gerichteter Kreis), falls $n > 0$ und $v_0 = v_n$
- Graph ohne Zyklen heißt *azyklisch* (DAG).
- E^* (Erreichbarkeit) ist dann eine Halbordnung.
- benutze DAGs zur Modellierung von Vorrängen („x ist wichtiger als y“)

  die Halbordnung heißt deswegen auch *Präzedenz*

Ü: welche Eigenschaft von Halbordnungen gilt nicht für E^+, wenn G Zyklen enthält?

Beispiel

Minimum Storage-Time Sequencing
`http://www.nada.kth.se/~viggo/wwwcompendium/node176.html`
Formulierung mittels

- DAG
- Funktionen

  (Kanten-Gewichte, Knoten-Permutation, Indizierung)
- Summation über Mengen

  (beachte implizite Deklaration der lokalen Variablen)

Automaten

Automat $A = (\Sigma, Q, I, F, \delta)$ mit

- Q Menge der Zustände
- $I \subseteq Q$ Menge der initialen Zustände
- $F \subseteq Q$ Menge der finalen Zustände
- Σ Menge der Kantenmarkierungen (Alphabet)
- $\delta \subseteq Q \times \Sigma \times Q$ Übergangsrelation

markierter Weg in A: $(v_0, c_1, v_1, c_2, v_2, \dots, v_{n-1}, c_n, v_n)$
so daß $\forall 0 \le i < n : (v_i, c_{i+1}, v_{i+1}) \in \delta$.
Schreibweise: $v_0 \xrightarrow{w}_A v_n$ mit $w = c_1 c_2 \dots c_n$
Die *Sprache* des Automaten:
$L(A) = \{w \mid \exists i \in I : \exists f \in F : i \xrightarrow{w}_A f\}$

Determinismus

ein Automat $A = (\Sigma, Q, I, F, \delta)$ heißt *deterministisch*, wenn

- $|I| = 1$
- und zu jedem $(p, w, q) \in Q \times \Sigma^* \times Q$ höchstens ein mit w markierter Pfad von p nach q in A existiert.

zu jedem Automaten gibt es einen sprach-äquivalenten deterministischen.
nicht deterministische Automaten sind trotzdem nützlich:

- kleiner oder besser lesbar (Bsp: Automat für $\Sigma^* aba \Sigma^*$)
- effiziente Beschreibung mehrerer Möglichkeiten,
- Beschreibung nicht vorhersehbarer (Benutzer-)Aktionen.

Sprachen und Sprach-Operationen

Sprache = Menge von Wörtern, Wort = Folge von Zeichen
Bsp: $L = \{\epsilon, ab, baaaaa\}$
Operationen:

- Mengeoperationen: $L_1 \cup L_2, \dots$
- Verkettung $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$
- Potenz (iterierte Verkettung)

  $L^k = L \cdot \ldots \cdot L$ (mit k Faktoren)

  $L^* = L^0 \cup L^1 \cup L^2 \cup \dots$ (unendliche viele Summanden)

Satz: für alle Sprachen L:
(L entsteht durch $\cup, \cdot, ^*$ aus endlichen Sprachen)
$\iff$ (L ist Sprache eines endlichen Automaten)

Nebenläufige Automaten (Petri-Netze)

Ziel: Modellierung von Systemen mit verteiltem Zustand.

- Petri-Netz $P = (S, T, F)$ mit $F \subseteq S \times T \cup T \times S$
- Vorbereich einer Transition: $\mathrm{vor}(t) = \{s \mid (s, t) \in F\}$, Nachbereich $\mathrm{nach}(t) = \dots$
- Markierung von P: ist Funktion $m : S \to \mathbb{N}$
- eine Transition t *schaltet*:
  wenn alle Stellen im Vorbereich von t markiert sind,
  dann je eine Marke aus Vorbereich einziehen,
  und je eine Marke im Nachbereich austeilen

P definiert Zustandsübergangssyst. auf $\mathbb{N}^S$ mit Alphabet T.
In Spezialfällen ist die erreichbare Zustandsmenge endlich
(und damit die Sprache regulär)

UML: Zustands- und Aktivitäts-Diagramme

- Grundlage sind endliche Automaten
- Erweiterung: hierarchische Komposition

  (Verwendung von Teil-Automaten)
- Erweiterung: verteilter Zustand (*split* und *join*)

  Semantik wie bei Petri-Netzen

jedes solche Diagramm beschreibt eine Sprache
(= Menge der erlaubten Transitionsfolgen,
wobei eine Transaktion eine Aktion des Benutzers
oder des Systems ist)

UML: Sequenzdiagramme

- horizontal: Objekte (jeweils mit Zustand)
- vertikal: Zeit
- Kanten für Methodenaufruf/Rückkehr (allgemein: Kommunikation)
- Spezialfälle: Konstruktor, Destruktor

Ü: wie sieht das entsprechende Petri-Netz aus,
welche Markierungen hat es?

Die Shuffle-Operation

Automaten A_1 und A_2 nebenläufig und unabhängig voneinander
die entsprechende Sprachoperation heißt *shuffle*

- Ü: definiere $\mathrm{sh}(u, v)$, so daß z. B.

  $\mathrm{sh}(ab, cd) = \{abcd, acbd, acdb, cabd, cadb, cdab\}$
- $\mathrm{sh}(L_1, L_2) := \{w \mid \exists u \in L_1, v \in L_2 : w \in \mathrm{sh}(u, v)\}$
- Ü: bestimme $|\mathrm{sh}(u, v)|$ (Anzahl) aus $|u|, |v|$ (Längen),

  falls u und v disjunkte Alphabete haben

Übung Modellierung/Graphen

- Erreichbarkeit in ungerichteten Graphen:

  wenn E symmetrisch, dann ist E^* eine Äquivalenz-Relation
- topologisches Sortieren:

  wenn (V, E) ein DAG, dann gibt es eine Knotenreihenfolge ohne Rückwärtskanten

  (1. Aussage formulieren, 2. beweisen)
- weiteres Scheduling-Beispiel (lesen)
- Scheduling-Beispiel (Klausursitzplan) (schreiben)

- Zustandsübergänge bei Spielen/Puzzles (Bauer/Kohl/Ziege/Wolf, Umfüllen von Flüssigkeiten, Lunar Lockout)

Anforderungs-Analyse

Anforderungs-Analyse

(vgl. Skript Weicker)

- Lastenheft (Anforderungen des Kunden, Anwendungsfälle)
- Pflichtenheft (System-Modell, Plan zur Realisierung der Anforderungen)

Wiederholung Modellierung

- Klassendiagram stellt Signatur dar (dabei: Klasse = einstellige Relation)
- in dieser Signatur kann die restliche Spezifikation (als prädikatenlogische Formel) hingeschrieben werden

Übung System-Modell

Stunden- und Raumplanungssystem
(vgl. `http://stundenplan.htwk-leipzig.de:8080/ws/Berichte/Text-Listen;Studenten-Sets;name;10IN1-B?template=UNEinzelGru\&weeks=_49\&days=\&periods=3-64\&Width=0\&Height=0`)

- Akteure? Anwendungsfälle? (Anzeige- und Mutationsereignisse)
- Klassen? Relationen? Anzahl-Constraints?
- Spezifikation „konfliktfreier Stundenplan“
- Hausaufgabe: Modellierung des Zusammenhangs zwischen Planungsaufgabe (Modulplan für ein Semester) und Lösung (Stundenplan)

Software-Architekturen

Definition

- jedes Softwaresystem besteht aus Komponenten
- jede Software-Architektur ist ein Muster für die Verknüpfung von Komponenten

  („die Architektur ist das, was bleibt, wenn man die Komponenten austauscht“)

typische Architekturen/Architektur-Fragen

- Daten/Verarbeitung/Darstellung
- stand-alone oder Client/Server
- Thin Client/Rich Client
- konkrete Form der Komponenten (z. B. Java-Beans), der Kommunikation

Software-Architektur

- vlg. Skript Weicker
- Beispiel Olat `http://www.olat.org/` / OPAL

  `http://www.olat.org/website/en/html/unit_development.html`

  Ralf Rublack: OLAT-Technologie-Überblick,
  Florian Gnägi: Understanding the OLAT-Core layout
- Beispiel autotool `https://autotool.imn.htwk-leipzig.de/cgi-bin/Super.cgi` / autOlat `https://autolat.imn.htwk-leipzig.de/`

  autOlat-RPC-Protokoll `https://autolat.imn.htwk-leipzig.de/docs/protocol.pdf`

Vorlesung/Übung KW50

Vorlesung (Skript Weicker)

- programmiersprachliche Konzepte,
- Schichtenarchitektur, thin/rich clients

Übung

- autotool-RPC-Spezifikation Kommandozeilen-Client)
  - lesen `https://autolat.imn.htwk-leipzig.de/docs/protocol.pdf`
  - Sequenzdiagramm für Client
  - Client-Prototyp für Aufgabe *Zahlentheorie* → *Times–Direct, Times–Quiz*
    Server-Adresse:

`http://autolat.imn.htwk-leipzig.de/cgi-bin/autotool-0.2.0.cgi`
Java-XML-RPC-Client:
`http://ws.apache.org/xmlrpc/client.html`

- Würfelspiel
  - Hausaufgabe (Miniprojekt) Systemmodell für Client/Server (Klassendiagramm und Protokollspezifikation)

Projekt Mex: Anwendungsfälle

- Spieler meldet sich bei Spielleiter an/ab

  (minimale Benutzerverwaltung: jeder Spieler hat Namen und Paßwort, wird vorher festgelegt, wird nur geprüft, wird nicht geändert)
- Spielleiter stellt die Mitspieler zusammen (eine Teilmenge der angemeldeten Spieler), führt ein Spiel durch und verarbeitet das Resultat (Änderung der Kontostände)
- Spielleiter veröffentlicht Statistiken

Projekt Mex: Ablauf

- bis 9. Januar:

  Lasten- und Pflichten„heft“(≤ 1 Seite) sowie Systemmodell (Daten, Protokoll, Kodierung (XML-RPC)

  Abgabe über OPAL-Gruppe, als PDF-Dokument, LaTeXwird dringend empfohlen
- verbindliche Festlegung des Protokolls in VL am 10. Januar, Server-Prototyp wird vorgegeben
- bis 16. Januar: Client-Entwurf (Interfaces, Testfälle)
- bis 23. Januar: GUI-Entwurf (Freihandskizze), Kompilations-, Installations- und Bedienungsanleitung
- bis 30. Januar: Strategie-Beschreibung, Wertungsspiele

Kosten- und Aufwandsschätzungen

Überblick

die wichtigsten Fragen:

- *wie teuer* wird das Projekt?
- *wie lange* dauert das Projekt?

das Hauptproblem:

- man braucht die Antworten, *bevor* das Projekt beginnt.

die Methode:

- schätzen unter Verwendung eines Modells
- laufend und später: Modell kalibrieren

(vgl. Skript Weicker, hier nur Zusammenfassung)

Methode

- Schwierigkeit des Projektes ist bestimmt durch seine externen Schnittstellen (zum Benutzer, zu Datenquellen)

  ist erkennbar aus Lastenheft

  wird gemessen in *function points*
- Function Points werden umgerechnet in Quelltextzeilen (kLOC)

  ist abhängig von Programmiersprache
- Textgröße wird umgerechnet in Personen-Monate

  passende Personenzahl wird bestimmt

Methoden zur Kostenschätzung

Boehm, nach Endres/Rombach (Kap. 9)

- algebraische Modelle (FP, COCOMO)
- top-down (Kosten folgen aus gewünschten Produkteigenschaften)
- bottom-up (Summe der Kosten der Produkt-Komponenten)
- Expertenmeinung, • Analogieschlüsse
- price-to-win (d. h. Mitbewerber unterbieten)
- Parkinson-Gesetz (verbrauche das verfügbare Geld)
- target pricing (wieviel kann der Kunde zahlen?) – vgl. `http://timharford.com/` *Undercover Economist* zu Kaffeepreisen

Aufwand und Nutzen

- Kosten werden meist unterschätzt.
- Die ersten 90 % des Funktionsumfangs benötigen die ersten 90 % der Projektlaufzeit, die restlichten 10 % benötigen die andern 90 % der Projektlaufzeit.

  vgl. F. Lüpke-Narberhaus: Software zur Studienplatzvergabe — Länder sind sauer auf Programmierer `http://www.spiegel.de/unispiegel/studium/0,1518,804585,00.html`
- (Folgerung): 80 % der Leistung kann man für nur 20 % des Aufwands bekommen.

  (aber Vorsicht: 79 % Klausur-Punkte ergibt Note 3)

Teamgröße und Kommunikationsaufwand

Beispiel: bestimme Dauer und Kosten für $n = 1, 2, \ldots$ unter folgenden Annahmen:

- Team von n Personen als vollständiger Graph
- jede Kante entspricht 1 h Kommunikation pro Tag (8 h)
- ingesamt zu leistende Arbeit: 100 h

Folgerungen

- Brooks’ Gesetz
- Baumstruktur (Vorteile/Nachteile)?
- Schnittstellen

Lines of Code

Naturkonstante (seit Jahrzehnten):

ein Programmierer schreibt pro Tag $\boxed{}$ *Zeilen Quelltext*

. . . der im Endprodukt landet, d. h. Zeit für Entwurf, Test, Dokumentation ist inbegriffen.
Folgerung:

- Steigerung der Produktivität *nur* dadurch, daß jede einzelne Zeile mehr bedeutet:
- geeignete Abstraktionen (Unterprogramme, Bibliotheken)
- geeignete Abstraktions*mechanismen* (der Programmiersprache), z. B. Interfaces, Funktionen höherer Ordnung

Weitere Naturkonstanten (I)

Millers Gesetz (zitiert nach Endres/Rombach)

Das Kurzzeitgedächtnis verwaltet 7 (± 2) *Informationen.*

Folgerungen

- für Programmierer
- für Benutzer (d. h. für Oberflächen-Entwurf)
- für Präsentationen

Weitere Naturkonstanten (II)

Das Berufsbildungsgesetz
(Norman, nach Endres/Rombach)

Zwischen Anfänger und Meister liegen 5000 Stunden.

Beispiele

- Schule, Vollzeit-Studium
- Freizeit-Sport, -Musik

Sackmans (zweites) Gesetz

Programmierer leisten unterschiedlich viel.

(Studie von Sackman 1966, mit 12 Programmierern)

Tätigkeit (Einheit)	bester	schlechtester Wert
Programmieren (h)	2	50
Testen (h)	1	26
Laufzeit (s)	50	541
Programmgröße (LOC)	651	3287

$\Rightarrow$ nur die leistungsfähigen Programmierer einstellen

`http://www.reddit.com/r/haskell/comments/ngbbp/haskell_only_esigning_startup_closes_second_angel/`

`http://www.paulgraham.com/avg.html`

Auswertung Projekt–Systemmodell

Statistiken

- 9 von 11 Gruppen haben abgegeben
- 2 Gruppen haben offenbar keine Ahnung, um welches Spiel es geht (aber übereinstimmende Klassendiagramme)
- 4 benutzen LaTeX
- 2 benutzen Prädikatenlogik
- alle benutzen Typen

Kritik

- Begriffe nicht exakt definiert/unterschieden (Runde, Spiel)
- anwendungsspezifische Datentypen ...
  - nicht konsequent benutzt (erst `Wurf`, dann tdoch `int`)
  - gar nicht benutzt (sondern `String[]`)
- Zustände (des Servers, der Spieler) nicht exakt definitiert
- Zustandsübergänge nicht exakt definiert

Protokoll-Nachrichten

- Spieler ruft Schiedsrichter:
  - anmelden, abmelden
- Schiedsrichter ruft Spieler:
  - Durchführen des eigenen Spielzuges
    - aufdecken oder glauben
    - würfeln und ansagen
  - Spiel-Beginn und -Ende
  - Information über fremde Spielzüge

welche Zustandsänderungen finden jeweils statt?

Alternatives Protokoll

Spieler ruft Schiedsrichter (immer)

- um letzten Spielzug zu erfahren.
- und eigenen Zug auszuführen, falls nötig.

welche Zustandsänderungen finden jeweils statt?

Unterschiede zwischen beiden Protokollen?

Modellierung An/Abmeldung/Konto

unterscheide zwischen

- statischer Information (Spielername und -Paßwort)
- langfristiger veränderlicher Information (Kontostände)
- kurzlebiger Information (URLs für Callbacks)

beachte: wie wird die Registrierung fremder Callbacks verhindert?

unter diesem Aspekt die eingereichten Entwürfe durchlesen: `http://www.imn.htwk-leipzig.de/~waldmann/edu/ws11/st/mex/modell/`

Nächster Schritt: Client-Entwurf

- Datenklassen spezifizieren (Wurf, Login): Testfälle für Operationen angeben
  - compareTo (o. ä.)
  - Konversion von/zu XML-RPC-Objekt
- Datenverarbeitung spezifizieren:
  - `interface Spieler` mit Typen für die Methoden, die der Schiedsrichter später aufruft.
    (Argumente und Resultat sind dabei *nicht* XML-RPC-kodiert)
  - junit-Testfälle dafür

Abgabe (OPAL) bis Montag, 16. Januar, 13:45

Auswertung Client-Entwurf/Tests

`http://www.imn.htwk-leipzig.de/~waldmann/edu/ws11/st/mex/tests/`

- Junit4 benutzen (nicht 3.8)!
- fromXml/toXml testen!
- Spiel/Runde, Beginn/Ende?
  illegale Reihenfolgen ablehnen (zum Debugging des Servers)

weiter:

- VL KW55: Tests, Bugreports, Quelltextmanagement
- Übung KW55: Client-Programmierung
- Abgabe 23. 1.: Anleitungen: bauen, installieren, bedienen.

Übung KW 55

- Server-RPC:
  `http://nfa.imn.htwk-leipzig.de:2012/rpc`
- Benutzernamen/Passwörter im Moment: A/A, B/B ... L/L.
- Server-Status:
  `http://nfa.imn.htwk-leipzig.de:2012/log`
  im Moment: Statistiken werden bei Server-Neustart zurückgesetzt
- Server-Quellen (auch Beispiel-Client)
  `https://github.com/jwaldmann/mex`

Produktqualität

Klassifikation der Verfahren

- Verifizieren (= Korrektheit beweisen)
  - Verifizieren
  - symbolisches Ausführen
- Testen (= Fehler erkennen)
  - statisch (z. B. Inspektion)
  - dynamisch (Programm-Ausführung)
- Analysieren (= Eigenschaften vermessen/darstellen)
  - Quelltextzeilen (gesamt, pro Methode, pro Klasse)
  - Klassen (Anzahl, Kopplung)
  - Profiling

Testen: Definition, Motivation

Software(-Komponente) testen = für bestimmte Eingaben ausführen und Resulate mit Spezifikation vergleichen

- Spezifikation ⇒ Testfälle
- bei Fehlen einer formalen Spezifikation sind Testfälle die nächstbeste Näherung

test driven development (= erst Testfälle schreiben, danach Quelltexte) bedeutet: erst spezifizieren, dann implementieren.

Tests und Schnittstellen

zur jeder Art von Schnittstelle gehört eine Art von Tests, z. B.

- Benutzerschnittstelle (Web):
  Click-Recorder/Replayer/Verifier (Bsp. `http://seleniumhq.org/`)
- textuelle Schnittstellen: Textvergleiche z. B. mit `diff`
- Komponentenschnittstellen (Methoden): unit tests (Java: `http://www.junit.org/`, C#: `http://www.nunit.org/`)
- Schnittstellen zwischen Anweisungen (innerhalb einer Methode): Zusicherungen (assert) (z. B. für Invarianten)

Dynamische Tests: Black/White

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

Black-Box-Tests

ohne Programmstruktur zu berücksichtigen.

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

während Produktentwicklung:
Testmenge ständig erweitern,
frühere Tests immer wiederholen (regression testing)

Fehlermeldungen

sollen enthalten

- Systemvoraussetzungen
- Arbeitsschritte
- beobachtetes Verhalten
- erwartetes Verhalten

Verwaltung z. B. mit Bugzilla, Trac
Vgl. Seminarvortrag D. Ehricht:
`http://www.imn.htwk-leipzig.de/~waldmann/edu/ss04/se/ehricht/bugzilla.pdf`

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`)

Delta Debugging

Andreas Zeller: *From automated Testing to Automated Debugging*, automatische Konstruktion von

- minimalen Bugreports
- Fehlerursachen (bei großen Patches)

Modell (Intervallverkleinerung, vgl. binäre Suche)

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

Delta Debugging (II)

```
dd(low, high, n) =
    let diff = size(high) - size(low)
        c_1, .. c_n = Partition von (high - low)
    if exists i : test (low + c_i) == FAIL
        then dd(                   )
    else if exists i : test (high - c_i) == OK
        then dd(                   )
    else if exists i : test (low + c_i) == OK
        then dd(                           )
    else if exists i : test (high - c_i) == FAIL
        then dd(                           )
    else if n < diff
        then dd(          ) else (low, high)
```

`http://www.infosun.fim.uni-passau.de/st/papers/computer2000/`

JUnit

Beispiel:

```
import static org.junit.Assert.*;
import org.junit.Test;
public class T {
     @Test
     public void t1 () {
          assertTrue(1 + 2 == 3);
     }
}
```

Eclipse: new $\to$ junit(4)test case, run as $\to$ test case

NUnit

Quelltext:

```
using NUnit.Framework;
[TestFixture] public class Test {
     [Test] public void check() {
          Assert.IsTrue (1+2 == 3);
     }
}
```

Kompilation:
`gmcs -r:nunit.framework -t:library Test.cs`
Ausführung:
`nunit-console Test.dll`
Weitere Beispiele: benutze All, Any zur Spezifikation von Halbgruppe, Gruppe.

Testfallgenerierung: Quick/Smallcheck

automatische, typgesteuerte Erzeugung von Testfällen

```
import Test.SmallCheck
data Wurf = .. ; instance Serial Wurf ..
transitive r =
    \ u v w -> ( r u v && r v w ) <= r u w
test ( transitive ((<)::Wurf->Wurf->Bool))
```

Koen Claessen and John Hughes: *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*, In Proc. of International Conference on Functional Programming (ICFP), ACM SIGPLAN, 2000.
(ICFP 2010: most influential ICFP'00 paper award)
ähnliche Idee in `http://hackage.haskell.org/package/smallcheck-0.6`

Programmablauf-Tests

bezieht sich auf Programm-Ablauf-Graphen (Knoten: Anweisungen, Kanten: mögliche Übergänge)

- Anweisungs-Überdeckung: jede Anweisung mindestens einmal ausgeführt
- Zweigüberdeckung: jede Kante mindestens einmal durchlaufen — Beachte: `if (X) then { A }`
- Pfadüberdeckung: jeder Weg (Kantenfolge) mindestens einmal durchlaufen — Beachte: Schleifen (haben viele Durchlaufwege)
  Variante: jede Schleife (interior) höchstens einmal
- Bedingungs-Überdeckung: jede atomare Bedingung einmal true, einmal false.

Prüfen von Testabdeckungen

mit Werkzeugunterstützung, Bsp.: *Profiler*:
mißt bei Ausführung Anzahl der Ausführungen ...

- ... jeder Anweisung (Zeile!)
- ... jeder Verzweigung (then oder else)

(genügt für welche Abdeckungen?)

Profiling durch Instrumentieren (Anreichern)

- des Quelltextes
- oder der virtuellen Maschine

Beispiel: `http://www.haskell.org/haskellwiki/Haskell_program_coverage`,
ähnliche Werkzeuge für andere Sprachen/GUIs

Quelltextverwaltung

Anwendung, Ziele

- aktuelle Quelltexte eines Projektes sichern
- auch frühere Versionen sichern
- gleichzeitiges Arbeiten mehrere Entwickler
- ... an unterschiedlichen Versionen (Zweigen)

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

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)

Daten und Operationen

Daten:

- Archiv (repository)
- Arbeitsbereich (sandbox)

Operationen:

- check-out: repo $\to$ sandbox
- check-in: sandbox $\to$ repo

Projekt-Organisation:

- ein zentrales Archiv (CVS, Subversion)
- mehrere dezentrale Archive (Git)

Versionierung (intern)

... automatische Numerierung/Benennung

- CVS: jede Datei einzeln
- SVN: gesamtes Repository
- darcs: Mengen von Patches
- git: Snapshot eines (Verzeichnis-)Objektes

Objekt-Versionierung in Git

Git verwaltet (in `.git`) eine *persistente* Sicht auf den Verzeichnisbaum (inkl. aller Änderungen)

- Objekt-Typen:
  - Datei (blob),
  - Verzeichnis (tree), mit Verweisen auf blobs und trees
  - Commit
    mit Verweisen auf tree und commits (Vorgänger)

  `git cat-file [-t|-p] <hash>`
- Objekte sind *unveränderlich* und durch SHA1-Hash (160 bit $=$ 40 Hex-Zeichen) identifiziert
- statt Überschreiben: neue Objekte anlegen
- jeder frühere Zustand kann wiederhergestellt werden

Versionierung (extern)

... mittels Tags (manuell erzeugt)
empfohlenes Schema:

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

Änderungen bedeuten:

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

Sonderformen:

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

Arbeit mit Zweigen (Branches)

- Repo anlegen: `git init`
- im Haupt-Zweig (master) arbeiten:
  `git add <file>; git commit -a`
- abbiegen:
  `git branch <name>; git checkout <name>`
- dort arbeiten: `... ; git commit -a`
- zum Haupt-Zweig zurück: `git checkout master`
- dort weiterarbeiten :`... ; git commit -a`
- zum Neben-Zweig: `git checkout <name>`
- Änderung aus Haupt-Zweig übernehmen:
  `git merge master`

Übernehmen von Änderungen (Merge)

durch divergente Änderungen entsteht Zustand mit 3 Versionen einer Datei:

- gemeinsamer Start G
- Versionen I, D (ich, du)

Merge:

- Änderung $G \to D$ *bestimmen*
- und auf I *anwenden*,
- falls das *konfliktfrei* möglich ist.

Änderung $=$ Folge von Editor-Befehlen (Kopieren, Einfügen, Löschen)
betrachten dabei immer ganze Zeilen

Projekt Mex, KW 56

- Auswertung Handbücher
- Server mit echten Namen und Paßwörtern sowie Persistierung der Kontostände
- Wertungsspiele von Do. (26. 1.) abends bis Di. (31. 1.) vormittags (Auswertung in Vorlesung)
- Vorschläge für Wertung? (Punkte durch Spiele?)
- Abgabe bis Mo., 30. 1., 13:45: Beschreibung und Begründung der im Client implementierten Spielstrategie

Entwurfsmuster zur GUI-Implementierung

Begriffe, Motivation

Def: (objektorientierte) Entwurfsmuster:
Standard-Architekturbausteine auf Klassen-Ebene, ausgedrückt durch Interfaces.
Ziel ist die Trennung von Verantwortlichkeiten

- Befehl: Zuordnung von Aktion zu Bedienelement
- Model / View / Controller:
  Datenobjekt / Darstellung / Veränderung
- Kompositum: Hierarchie von GUI-Bestandteilen
- Strategie: Layout-Manager für GUI-Container

GUI-Beispiel

```
public static void main(String[] args) {
  JFrame f = new JFrame ("Demo");
  f.setDefaultCloseOperation (JFrame.EXIT_ON_CL
  JButton b = new JButton ("but");
  f.getContentPane().add (b);
  b.addActionListener (new ActionListener () {
    public void actionPerformed(ActionEvent e)
      System.out.println("click");
    } });
  f.pack(); f.setVisible(true);
}
```

vgl. `http://docs.oracle.com/javase/tutorial/uiswing/components/frame.html`

Das Befehls-Muster

```
JButton b = new JButton ("foo");
b.addActionListener (new ActionListener () {
  void actionPerformed (ActionEvent ae) {
    System.out.println ("click on foo");
} }
```

- Bedeutung: Methode des Befehlsobjektes wird ausgeführt bei Betätigung (Click) des Bedienelementes
- Realisierung: der Typ des Befehlsobjekts ist eine anonyme lokale Klasse, die das `interface ActionListener` implementiert.

Das Befehlsmuster realisiert „Unterprogramm als Datum“.
(Viele andere Muster tun das ebenso.)

Das Beobachter-Muster

ersetzt die statische Kopplung (Quelltext) von

- Objekt S (Subjekt)
- zu Beobachter-Objekten $B_1, B_2, \ldots$

durch dynamische Kopplung in der anderen Richtung
(der Beobachter meldet sich beim Subjekt an).

```
class S extends Observable {
  void m () { this.setChanged ();
              this.notifyObservers (arg);  } }
class B implements Observer {
    void update (Observable o, Object arg) { ..

S s; ... s.addObserver (new B ()); s.m ();
```

model/view/controller

Daten/Darstellung/Veränderung
Bsp: Zählerstand / Text auf Label / Button zum Erhöhen

Grundlage ist das *Beobachter-Muster*:
die Darstellung *beobachtet* die Daten, damit bei Änderung der Daten automatisch Änderung der Darstellung erfolgt. (Codebeispiel)

häufig sind View und Controller identisch:

- Slider (Schieberegler)
- Textarea für Aus- und Eingabe

Das Kompositum-Muster

eigentlich: (rekursiver, baumartiger) algebraischer Datentyp
ein Baum besteht aus :

- Wurzel,
- Liste von Bäumen (Kindern)

bei GUI-Konstruktion:

- Knotentyp ist `Component`
- manche Knoten haben Kinder:

```
class Container extends Component {
   void add (Component c) { .. }
}
```

Das Strategie-Muster

zu jedem `Container` gehört ein `LayoutManager`, dieser bestimmt die Positionen der Teilkomponenten

- bei Änderung des Rahmens (durch Bediener oder übergeordneten Container)
- bei Änderung des Inhaltes (Hinzufügen oder Entfernen von Teilkomponenten)

```
Container c =
   new JPanel (new GridLayout(2,0));
c.add (...) ; c.add (...);
c.validate ();
```

Der Layout-Manager ist ein Strategie-Objekt (ähnlich Befehls-Objekt)

Einfache Layout-Manager

- GridLayout (Rechteck-Gitter)

```
new GridLayout (int zeilen, 0);
new GridLayout (0, int spalten);
```

- BorderLayout (Rahmen)

```
Container c =
    new JPanel (new BorderLayout());
c.add ( ... , BorderLayout.NORTH);
c.add ( ... , BorderLayout.CENTER);
```

durch Schachtelung (Kompositum-Muster!) von Containern mit Managern definiert man Layouts, *ohne* eine einzige Pixelkoordinate hinzuschreiben.

Testklausur KW56

Modellierung, Zustandsübergänge

- der Mex-Server-Zustand enthält die Attribute
  - L: Menge der eingeloggten Spieler
  - S: Liste der Spieler für aktuelles Spiel
  - R: Liste der Spieler für aktuelle Runde
  - W: letzter Wurf, A: letzte Ansage.

  Beschreiben Sie die Beziehung zwischen Zustand (L, S, R, W, A) direkt vor und Zustand (L', S', R', W', A') direkt nach Aufruf der Methode `c.accept(Wurf u)`, bei der Client `c` das Resultat `boolean b` liefert. (2 Fälle!)
- Bestimmen Sie den Median der Mex-Würfe.
  (Welche Ansage ist zu genau 50 % gelogen?)

Architektur

Gesetz von Constantine: „Eine Struktur ist beständig, wenn der Zusammenhang stark und die Kopplung schwach ist.“

- belegen Sie beide Teile der Aussage durch je ein Hardware-Beispiel (Konstruktion von Autos, Häusern, …)
- welchen Einfluß auf die o.g. Parameter haben:
  - das Einführen von Schnittstellen (interface)
  - das Einführen von globalen (public) Attributen
- bewerten Sie diese Situation: eine Klasse enthält (private) Attribute a_1, a_2 und Methoden m_1, m_2.

  Dabei wird a_1 nur in m_1 benutzt und a_2 nur in m_2.

Kostenschätzung

- Wieviele Funktionspunkte (FP) hat Ihr Mex-Client? Begründen Sie Ihre Schätzung.
- Ein Produkt Q hat den doppelten Umfang eines Produktes P. Wie verhalten sich die Gesamtkosten der beiden Entwicklungsprojekte, wenn diese Berechnung angewendet wird:

  (Projektdauer in Monaten) $=$ (Produktumfang in FP)$^{0.4}$

  (Anzahl der Mitarbeiter) $=$ (Produktumfang in FP)$/150$

  Ist das Ergebnis glaubhaft?
- Stellen Sie einen Zusammenhang her zu der Aussage „ein Programmierer schreibt pro Tag … Zeilen “.
  Zu welcher Programmiersprache passen Ihre Zahlen?

Zusammenfassung

Themen

- Vorgehensmodell Wasserfall, Dokumente, Klammerstruktur
- Anforderungsanalyse, Spezifikation, Systemmodell
  - Mengen, Relationen, Funktionen (Klassen-, ER-Diagramme)
  - Aussagen- und Prädikatenlogik
  - Zustandsübergänge (Graph, Automat, Petri-Netz)
- Software-Architekturen (Kopplung, Zusammenhang)
- Kosten- und Aufwandsschätzungen (FP, LoC)
- Produktqualität (Tests)

Highscore-Aufgaben

- Lunar Lockout
- Mengen-Algebra

```
Gesucht ist ein Ausdruck (Term) mit dieser Be
    {2, {}, {3, {4}}}
Der Ausdruck soll höchstens die Größe 40 habe
Sie dürfen diese Symbole benutzen
    zweistellige : [ + , - , & ] ; einstellig
und diese vordefinierten Konstanten:
    A = {1, 2} ; B = {2, 3} ; C = {3, 4}

pow(pow (C - B ) + (B & C) - pow(A))
  +(A & B) - pow(pow(C)) - pow(C) + pow(A & C)
```

- Modell f. Spezifikation

$$\forall x.\forall y.R(x,y) \land B(x,y) \implies x = y) \land (\forall x.\forall y.\forall z.R(x,y) \lor R(y,z) \lor R(z,x)) \land (\forall x.\forall y.\forall z.B(x,y) \lor B(y,z) \lor B(z,x))$$

```
Interpretation
  { struktur = Struktur
    { universum = mkSet [ 1 , 2 , 3 , 4 , 5 ]
    , predicates = listToFM
      [ ( B
        , {(1 , 1) , (1 , 3) , (1 , 4) , (1 , 5) ,
           (2 , 2) , (2 , 3) , (2 , 4) , (2 , 5) ,
           (3 , 3) , (3 , 4) , (3 , 5) , (4 , 4) ,
           (5 , 5)}
        )
      , ( R
        , {(1 , 1) , (2 , 2) , (3 , 1) , (3 , 2) ,
           (3 , 3) , (4 , 1) , (4 , 2) , (4 , 3) ,
           (4 , 4) , (5 , 1) , (5 , 2) , (5 , 3) ,
           (5 , 5)}
        )
      ]
    , functions = listToFM [ ]
    }
  , belegung = listToFM [ ]
  }
```

Resultat:

```
45 : 5*96* : 3 Platzierungen :   1   1   1
35 : 5*59* : 2 Platzierungen :   1   1
20 : 5*11* : 1 Platzierungen :   1
20 : 5*44* : 1 Platzierungen :   1
```

Klausur

- Zulassung:
  - $\geq 50\%$ autotool
  - *und* alle 4 Projekt-Abgaben
- 120 min
- keine Hilfsmittel
- wissen und anwenden

Projekt-Auswertung

- Ziel: Softwareprojektentwicklung mit Spezifikation, Entwurf, Implementierung, Inbetriebnahme/Test . . . in (sehr) begrenzter Zeit
- Resultat (I): der olympische Gedanke
- Resultat (II):

```
5_apokalyptische_wuerfel  1174
3MPVA                     1023
```

Gewinner-Quelltexte:
`http://www.imn.htwk-leipzig.de/~waldmann/edu/ws11/st/mex/5aw/`

- Diskussion Strategie (lokal, global)