Konzepte für eine persistente Programmiersprache

Jürgen Schlegelmilch

2. Dezember 1999

Überblick

Warum noch eine Programmiersprache?

- Wieso sind Datenbanken anders?
- Was machen Programmiersprachen falsch?
- Wo liegen die Probleme, die wir lösen müssen?

Was brauchen wir denn alles?

- Datenmodell: Klassen und Beziehungen
- Verhaltensmodell: Funktionen, Ereignisse, Methoden
- Konsistenz und Persistenz

1. Warum eine neue Programmiersprache?

normale Programmiersprachen:

- Persistenz? nicht notwendig!
  - sequentielle Dateien, keinerlei Zugriffsunterstützung
  - darin nur Daten, keine Funktionalität
- Konsistenz? nicht notwendig!
  - bei Inkonsistenzen Neustart des Programms
- Anpassungen? Software-Lebenszyklus!
- Effizienz? Maschinenzyklen zählen!

Aufgaben von Datenbanksystemen

- Speichern großer Datenmengen
- über lange Zeiträume
- für mehrere Anwendungen
- für mehrere Anwender
- zuverlässig und dauerhaft
- effizient

Datenbank-Anforderungen: große Datenmengen

- Datenbankgröße $\gg$ Hauptspeichergröße
  $\Rightarrow$ Sekundärspeicher bestimmt Geschwindigkeit
  - nicht Maschinenzyklen, sondern Plattenzugriffe zählen
  - nicht peep-hole, sondern algebraische Optimierung
  - Sekundärspeicher ist segmentiert, hat Latenzzeiten
    $\Rightarrow$ Hauptspeicheralgorithmen $\neq$ Sekundärspeicheralgorithmen
- Optimierung sehr wichtig
  - algebraische Eigenschaften von Operationen
  - mengenorientierte Datenverarbeitung

Datenbank-Anforderungen: lange Zeiträume

- große Datenmengen auch bei geringen Zuwachsraten
- Datenbankschema = Modell der Realität
  Realität und/oder Modellierung ändert sich

  $\Rightarrow$ Datenbankschema anpassen (Schemaevolution)
  $\Rightarrow$ Sichten erhalten Kompatibilität

- Objekte ändern sich $\Rightarrow$ Typanpassung, Migration notwendig

Datenbank-Anforderungen: mehrere Anwendungen

- Redundanzfreiheit trotz abweichender Sichten
  - Sichtintegration
  - Normalisierung
- zentrale Instanz notwendig für
  - Persistenz: Welche Daten werden von keiner Anwendung gebraucht?
  - Konsistenz: Sicherheit auch gegen Programmfehler
- Optimierung: nicht für einzelne Anwendungen
  - physische Datenunabhängigkeit
  - deklarative Datenverarbeitung

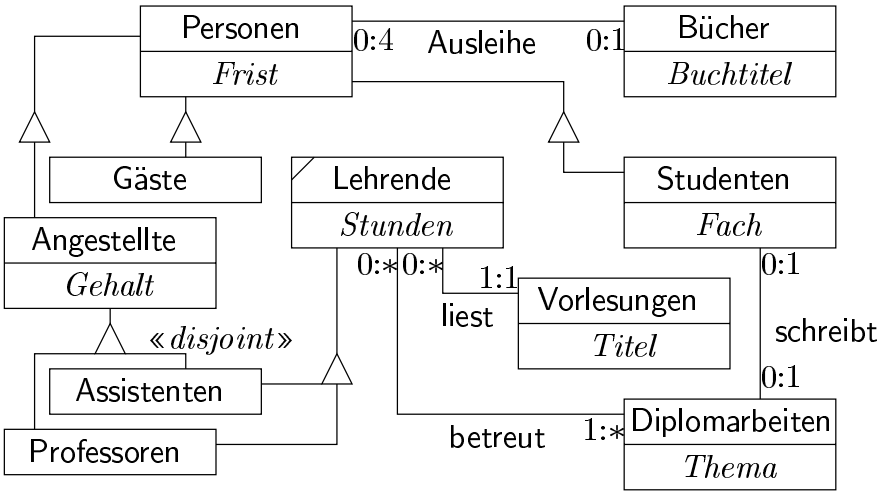
Datenbank-Anforderungen: mehrere Anwender

- transaktionsorientierte Datenverarbeitung
  - Isolation verschiedener Anwender voneinander
  - Konsistenzerhaltung unter allen Umständen
  - Rücksetzmöglichkeit im Falle von Fehlern
  - Zuverlässigkeit, Dauerhaftigkeit
- Sichten für individuelle Anpassung

Programmiersprachen-Anforderungen

- Modularisierung: Module, Klassen, abstrakte Datentypen
- Polymorphismus: Eignung für mehrere Datentypen
  - Datenstrukturen: polymorphe Variablen
  - Algorithmen: Überladen, spätes Binden
- Abstraktionsfähigkeit, „Singular Linguistic Unit“
- Orthogonalität von Syntax, Semantik

Modellierungsprobleme



- Sichtklasse **Lehrende**
- mehrere Klassifizierungsschemata
- neue Klassen $\Rightarrow$ viele Redefinitionen
- Migration: **Studenten** werden **Assistenten**
- **liest** erweitern um Raum: ternär
- Konsistenzsicherung?

2. Wir machen es besser

Datenmodell: objektrelational

- Klassen ordnen Objekten Zustände zu
- Beziehungen setzen Objekte in Relation

Grundlagen: abstrakte Datentypen und Funktionen

Verhaltensmodell: ereignisbasiert

- Objekte reagieren auf Ereignisse
- Funktionen berechnen
  - Anfrageergebnisse
  - neue Objektzustände

Datenmodell: die Basis

Grundlage: rein funktionale Programmiersprache

Typsystem: Typvariable, Typquantoren, Typhierarchie

- *keine* Subsumption; nur parametrischer Polymorphismus
- *keine* rekursiven Typen; nur Fixpunktoperator
- *keine* Referenztypen

darauf aufbauend:

- abstrakte Datentypen
- Typkonstruktoren als Monade

Datenmodell: abstrakte Datentypen

abstrakte Datentypen: mit existenz-quantifizierten Typen

- Schnittstellenbeschreibung: Tupeltyp mit Typvariable
- Axiome zur Semantik-Definition
- Implementierungstyp als Sorte des ADT
- Tupel mit Funktionen als Implementierungen der Operationen

Vereinigung, Schnitt von ADTs; Vererbung

Generator: erzeugt ADT-Wert aus Wert der Sorte und Funktionentupel

Datenmodell: Objekte und Klassen

Objekte: Werte vom Typ Old

Klassen: bilden Objekte auf Zustandswerte ab

- Zustandstyp: ADT $\Rightarrow$ Einkapselung
- Basisklassen: gespeicherte Funktion
- Sichtklassen: berechnete Funktion

Extension = Vorbereich der Zustandsfunktion = Menge der Objekte

Datenmodell: Klassen als Typen

Klassentypen: Typ Old mit Konsistenzbedingung

- Konsistenzbedingung: Objekt muß in Extension sein
- Subsumption durch Implikation der Konsistenzbedingung
- Klassentypen wegen Migration abhängig vom Datenbankzustand

Typprüfung für Klassentypen als Konsistenzproblem

- Integration in allgemeine Konsistenzprüfung
- vorübergehende Inkonsistenzen möglich

Datenmodell: Beziehungen zwischen Klassen

Beziehung subC: definiert Klassenhierarchie als GAG

- strenge Ordnungsrelation
- $\langle C, D \rangle \in \text{subC} \Rightarrow C$ Unterklasse von D
- Konsistenzbedingung: Extension Unterklasse $\subseteq$ Extension Oberklasse
- *keine* Bedingungen an Zustandstypen, *keine* Vererbung

Beziehung disj: definiert Disjunktheit von Klassen

- binäre, symmetrische Relation
- $\langle C, D \rangle \in \text{disj} \Rightarrow$ Extensionen C, D disjunkt

Datenmodell: Beziehungen

Beziehung: Relation mit Attributen beliebigen Typs

- Attribute mit Klassentypen = Rollen
  Tupel stellt Verbindung her zwischen Objekten in Rollen
- Verbindung ist ungerichtet, mehrstellig, mit Attributen
- Basisbeziehung: gespeicherte Relation
  Attribute mit Klassentypen auf oberster Ebene
- Sichtbeziehung: berechnete Relation
- keine Einkapselung, keine Methoden oder Operationen
- Konsistenzbedingung mit Strategie für Inkonsistenzen

Datenmodell: keine Referenzen

Referenzen: Speicheradressen

- schnelle Navigation im Hauptspeicher
- nur für binäre, gerichtete Beziehungen
- keine Attribute, Konsistenzbedingungen; referentielle Integrität

ungeeignet für allgemeine Beziehungen; Implementierungstechnik

Beziehungsklassen: implementieren allgemeine Beziehungen mit Referenzen

- Beziehungsklasse hat Attribute, Rollen der Beziehung
- Verbindung = Objekt der Beziehungsklasse
- 2 Referenzen pro Rolle nötig, indirekte Navigation über Beziehungsobjekt
- Probleme mit Persistenz durch Erreichbarkeit

Beziehungen: Beziehungsklassen

nicht Klassen zur Implementierung von Beziehungen

sondern aus Beziehungen abgeleitete Klassen

- Klassen: Funktionen = rechtseindeutige Relationen

$\Rightarrow$ Beziehungsrelation nach einer Rolle gruppieren

- übrige Attribute, Rollen in ADT kapseln
- gewünschte Operationen definieren

Vorteil: Integration in Klassenhierarchie

- Berücksichtigung bei (Anfrage-)Methoden
- Objekt-spezifische Sicht auf Beziehung
- Zugriff auf in Beziehung stehende Objekte wie mit Referenz

Beziehungsklassen: Beispiel

Szenario: Beziehung `arbeitet(Person,Firma,Gehalt)`

Beziehungsklassen: für jede Rolle eine möglich

- Klasse `Arbeiter` mit Attribut `Auftraggeber`,
  Typ: Menge von Paaren $\langle$`Firma`, `Gehalt`$\rangle$
- Operation *gehalt* liefert etwa Summe der Gehälter

$\Rightarrow$ automatische Berücksichtigung bei Methode *Gehalt*

- Operation *firmen* liefert Menge der Firmen aus `Auftraggeber`

$\Rightarrow$ Navigation von Arbeiter zu seinen Auftraggebern möglich

Datenmodell: Konsistenz

Spezifikation: deklarativ

- Konsistenzbedingungen für Klassen, Beziehungen
- Unterscheidung lokale, globale Bedingungen durch Reparaturmöglichkeit
  lokal: kann durch Löschen eines (inkonsistenten) Elements erfüllt werden
  global: keine Reparatur möglich ⇒ Transaktionsabbruch
- Konsistenzsicherungstrategie (nur für Beziehungen):
  bei lokalen Inkonsistenzen entweder Reparatur oder Transaktionsabbruch

Implementierung: stratifizierte Konsistenzprüfung, bis Fixpunkt erreicht
⇒ entweder alles konsistent oder Transaktionsabbruch

Konsistenz: lokale vs globale Bedingungen

Ziel: Bedingungen an Mengen erkennen, die durch Löschen erfüllbar sind

lokal: äußerster Quantor über zu prüfende Menge ist All-Quantor

- Beispiel: `alle Personen sind jünger als 120`
- Prüfung einzelner Elemente möglich
- Reparatur durch Löschen inkonsistenter Elemente möglich

global: kein Quantor, oder äußerster Quantor ist Existenz-Quantor

- Beispiel: `Es existiert eine Person mit Alter 110`
- Prüfung für gesamte Menge, keine Reparaturmöglichkeit

syntaktische Unterscheidung möglich

Konsistenz: stratifizierte Sicherung

Konsistenz durch Löschen lokal inkonsistenter Elemente

Problem: Reihenfolgeabhängigkeiten durch Bedingungen an mehrere Klassen, Beziehungen

- erst Objekte, dann Verbindungen? umgekehrt?
- erst Beziehung A, dann Beziehung B?
  erst Klasse C, dann Klasse D?

Keine allgemeine Lösung, da globale Ordnung nötig wäre

Stratifizierung: Einteilung in Schichten

- (willkürlich!) erst alle Klassen, dann alle Beziehungen
- Prüfung einer Schicht als ein Transaktionsschritt:
  Datenbankzustand $S_i \rightarrow$ Datenbankzustand S_{i+1}
- Prüfung innerhalb einer Schicht bezüglich S_i
  $\Rightarrow$ Löschungen anderer Elemente unberücksichtigt
- bei Transaktionsabbruch: $S_{i+1} = S_{i+2} = \cdots = S_0$
- Fixpunktbildung über Konsistenzsicherung
  $\Rightarrow$ alles konsistent, oder Transaktionsabbruch
- zuletzt globale Bedingungen prüfen

Datenmodell: keine Trigger

Trigger: Ereignis löst Aktion aus, falls Bedingung erfüllt

Probleme: Aktion kann beliebige Reparaturen machen

Konfluenz: Reihenfolge der Trigger egal
Terminierung: Datenbankzustand irgendwann stationär

keine allgemeine Lösung

allgemein: beliebige Reparaturen bringen immer Reihenfolgeprobleme

Datenmodell: Persistenz

Kriterien: typorthogonal, transparent, typunabhängig
(Atkinson, Morrison 1995)

Spezifikation: deklarativ

- Referenzen: Persistenz durch Erreichbarkeit
  - Objekte der Wurzelmenge sind persistent
  - von persistenten Objekten erreichbare Objekte sind persistent
- Beziehungen: rollenbasierte Persistenz
  - Rollen jeder Beziehung unterteilt in wichtige und unwichtige
  - Objekte sind persistent, wenn sie eine wichtige Rolle haben
  - Basisbeziehungen und Sichtbeziehungen gleichberechtigt
- Freispeicherverwaltungstechniken notwendig
- zusätzlich explizites Löschen möglich

Persistenz: Definition

rollenbasiert: Objekt persistent $\Longleftrightarrow$ Objekt hat wichtige Rolle

- Objekt wird gelöscht $\Rightarrow$ Verbindungen mit Objekt werden gelöscht

$\Rightarrow$ Objekte in wichtigen Rollen der Verbindungen verlieren diese

$\Rightarrow$ Objekt in wichtiger Rolle hängt ab vom Rest der Verbindung

positive Charakterisierung: was ist persistent?

- Verbindungen definieren Hyperkanten in Hypergraph H:
  Kopf: alle Objekte einer Verbindung; zeigt auf Objekt in wichtiger Rolle
- persistente Objekte = Knoten eines maximalen Untergraph H'
- persistentes Objekt liegt auf Pfad in H', der in Kreis beginnt
- Kreise entsprechen Wurzelmenge von Persistenz durch Erreichbarkeit

Persistenz: alternative Definition

negative Charakterisierung: was wird gelöscht?

- $K_i = \{\text{alle Objekte}\} \setminus \{\text{wichtige Objekte}\}$
- lösche Objekte in $K_i \Rightarrow$ Datenbankzustand $i + 1$
- Iteration, bis Datenbankzustand stationär

Beide Charakterisierungen sind äquivalent

Persistenz: Beispiele

Szenario Kontaktpflege: Alle Kunden und deren Freunde persistent

- Beziehung `ist_Kunde(Kunde,umsatz)`, wichtige Rolle: `Kunde`
- Beziehung `befreundet(A,B,seit)`, wichtige Rolle: `B`

$\Rightarrow$ Persistenz durch Erreichbarkeit

- Beziehung `ist_Kunde` ableitbar aus Klasse `Kunde`

$\Rightarrow$ klassenbasierte Persistenz

Szenario Diplomarbeiten: persistent, wenn Betreuer und Projekt bekannt

- Beziehung `betreut(DA,Betreuer,Projekt)`, wichtig: `Betreuer`, `Projekt`
- Betreuer oder Projekt werden gelöscht $\rightarrow$ Diplomarbeit wird gelöscht
- Beziehung `betreut` kann Verbund aus zwei Beziehungen sein
- Diplomarbeit soll persistent sein, wenn Student sie bearbeitet?
  Rolle `DA` wichtig in Beziehung `bearbeitet(DA,Student,seit)`

Verhaltensmodell

freie Funktionen: weder ADT noch Klasse zugeordnet

- kein Zugriff auf Element der Sorte eines ADT-Werts
- kein Zugriff auf Zustandsfunktionen von Klassen
- kann Methoden, Operationen aufrufen

Operationen: Implementierungen der Signaturen eines ADT

- manipuliert ADT-Wert als Wert der Sorte des ADT
- kein Zugriff auf Zustandsfunktionen von Klassen

Methoden: durch Funktionen *query* oder *modify* definiert

- keinen Zugriff auf Element der Sorte eines ADT-Werts
- indirekten Zugriff auf Zustandsfunktionen von Klassen

Verhaltensmodell: Transaktionen

- Zustand in funktionalen Sprachen
  - unreine Ansätze (imperative Elemente einbringen)
  - „state transformer monad“: ADT mit sequentieller Verknüpfung
  - lineare Typen: lineare Verwendung in Ausdrücken
  - Imperative Lambda Calculus: explizite Referenztypen
- Für transaktionsorientiertes Arbeiten: monadische Lösung
  - „state transformer monad“ führt implizit Zustand mit
  - „exception monad“ ermöglicht Transaktionsabbruch
  - Werte des Monad: Transaktionsschritte
  - Funktionen, Operationen, Methoden können Transaktionen sein

Der Monad Tx: Definition

- repräsentiert Berechnungen mit implizitem Zustand, potentielle Ausnahme
- Sorte: $\forall(X)\langle\langle \mathsf{DB}, \mathsf{DB}\rangle\rangle \to \langle\langle X + void, \mathsf{DB}, \mathsf{DB}\rangle\rangle$
- $\langle\langle \mathsf{DB}, \mathsf{DB}\rangle\rangle$: Zustand bei Transaktionsbeginn, aktueller Zustand
- $X + void$: Ergebnis der Berechnung, oder Transaktionsabbruch
- Operationen:
  - $unit$ erzeugt Werte von Tx= Transaktionsschritte
  - $bind$ führt zwei Transaktionsschritte sequentiell aus
  - $abort$ ist $\langle O, S\rangle \to \langle inright(void), O, O\rangle$
  - $commit$ ist $\langle O, S\rangle \to \langle inleft(void), S, S\rangle$
  - $fetch$ ist $\langle O, S\rangle \to \langle inleft(S), O, S\rangle$
  - $assign$ ist $\lambda(T : \mathsf{DB})\langle O, S\rangle \to \langle inleft(void), O, T\rangle$

Der Monad Tx: Auswertung

Transaktionen: Funktion *connect*

- stellt Verbindung zur Datenbank her, sorgt für Sperren
- sorgt für Konsistenzsicherung nach Auswertung der Transaktion
- liefert Ergebnis der Transaktion, oder Fehlerwert bei Abbruch

Subtransaktionen: Funktion *subtrans*

- führt Transaktion (inklusive Konsistenzsicherung) als Transaktionsschritt aus
- realisiert geschlossen geschachteltes Transaktionsmodell
- erfordert Unterstützung vom Datenbanksystem

Verhaltensmodell: Objekte, Methoden, Migratoren

Objekt: kann in vielen Klassen sein

- jede Klasse ordnet Objekt einen Zustandswert zu
- Gesamtzustand = Vereinigung der Teilzustände

Migratoren: schieben Objekte in Klassen, aus Klassen heraus

- Migration in Klasse: $o \mapsto T$ zu Zustandsfunktion hinzufügen
- Demigration aus Klasse: o aus Vorbereich der Zustandsfunktion löschen

Methoden: verbergen Struktur des Gesamtzustands

- Methodenaufruf sammelt Teilergebnisse von Zuständen
- Aggregatfunktion kombiniert Teilergebnisse

Methoden: Idee

- Aufruf Methode m: Zielobjekt empfängt Ereignis m (mit Argumenten)
- Objekt delegiert Ereignis an Teilzustände
- Teilzustände reagieren mit Auswertung der Operation m
  keine Operation m definiert $\Rightarrow$ kein Teilergebnis
- Methode sammelt Teilergebnisse $\langle cls, res \rangle$
  - Anfragemethode: Teilergebnisse werden aggregiert zu Gesamtergebnis
  - Änderungsmethode: res wird neuer Zustand in cls
    ($result$ ist Transaktionsschritt, Aggregation mit $bind$ von Monad Tx)
- durch Aggregation leicht erweiterbar ohne Redefinitionen

Methoden: Definition

Kapselung der Struktur von Objekten $\Rightarrow$ Meta-Methoden $query$, $modify$

Anfragen: Methode $query$ sammelt Teilergebnisse und aggregiert

- Argumente: Klasse C, Typ der Methodenargumente T, Operation m : DB $\to C \to T \to X$, Aggregatfunktion f : set$(\langle\langle cls, res : X\rangle\rangle) \to Y$
- Ergebnis: Methode mit Typ DB $\to C \to T \to Y$
  - sammelt Ergebnisse der Operationen m von Teilzuständen
  - wendet f auf Menge der Teilergebnisse an

Änderungen: Methode $modify$ konstruiert Transaktionsschritt

- Argumente: Klasse C, Typ der Methodenargumente T, Operation m : DB $\to C \to T \to X_C$ mit X_C Zustandstyp von C
- Ergebnis: Methode mit Typ $C \to T \to$ Tx
- wandelt Teilergebnisse in Transaktionsschritte, aggregiert mit $bind$

Methoden: Aggregation für Anfragen

klassisch: Aggregatfunktion für res, etwa Summation, Minimum, Maximum

Klassenkonstante: Auswahl $\langle cls, res \rangle$ mit $cls = C$
$\Rightarrow$ statisches Binden; Zustandsfunktion der Klasse C rekonstruierbar

Klassenmaximum: dito, mit $C =$ Klasse, in der Methode definiert ist

Klassenminimum: liefert Ergebnis der speziellsten Klasse des Zielobjekts
$\Rightarrow$ spätes Binden

Problem: speziellste Klasse nicht eindeutig $\Rightarrow$ Konfliktauflösung nötig

Methoden: Beispiele

$Gehalt$ mit Aggregatfunktion $\sum$ zur Summierung der Teilgehälter

- neue Unterklasse hinzufügen, Operation $gehalt$ definieren

$\Rightarrow$ wird automatisch berücksichtigt

$Extent$ mit Aggregatfunktion $\bigcup$ für Beziehungen

- Unterklasse für symmetrische Beziehungen (Konsistenzbedingung: binär)
- Unterklasse für reflexive Beziehungen (Konsistenzbedingung: binär)

keine Einordnung in Klassenhierarchie notwendig

komplexere Aggregatfunktionen möglich, etwa additive, multiplikative Anteile

Methoden: Konfliktauflösung

Idee: speziellste Klasse nicht eindeutig, aber alle liefern gleiches Ergebnis

- Mengen von Klassen betrachten, die als Minimum in Frage kommen
- falls mögliche Abweichung: Sichtklasse SC_i definieren als Schnittmenge

Mittel: Schnittklasse SC_i

- automatisch einzuordnen in Klassenhierarchie = Teilmengenverband
- Unterklasse der Basisklassen $\Rightarrow$ automatische Berücksichtigung bei spätem Binden
- Sichtklasse $\Rightarrow$ erfaßt automatisch Menge der betroffenen Objekte

Problem: welche Schnittklassen sind notwendig?

Konfliktauflösung: Algorithmus

Algorithmus: vier Schritte

1. bilde alle Mengen K_i von Klassen, mit $|K_i| > 1$
2. lösche Mengen K_i mit $\{C, D\} \subseteq K_i$ und $\langle C, D\rangle$ in disj oder subC*
3. sortiere restliche Mengen nach Teilmengenordnung $\subset$
4. durchlaufe Liste:
   (a) ersetze Teilmengen durch Schnittklassen
   $K_i' := (K_i \setminus \bigcup_{j<i \wedge \exists SC_j} K_j) \cup \{SC_j \mid j < i\}$
   (b) $C, D \in K_i'$ und $C.m \neq D.m \Rightarrow$ Konflikt erkannt
   erzeuge Schnittklasse $SC_i := \bigcap_{C \in K_i'} C$, definiere m in SC_i
   ordne SC_i in subC ein: $\forall C \in K_i$: $\langle SC_i, C\rangle \in$ subC

Konfliktauflösung: Beispiel I

Szenario: Methode $Frist$ soll verschieden sein für **Personen**, **Gäste**, **Studenten**, . . .

- Operation $Frist$ definieren in Zustandstypen dieser Klassen
- Methode $Frist$ definieren mit Aggregatfunktion

$$\lambda(E : \mathsf{set}(\langle\langle cls, res\rangle\rangle))pick(\{t.res \mid t \in E \wedge t.cls \in \min\{t.cls \mid t \in E\}\})$$

- Algorithmus prüft Mengen {**Personen**, **Gäste**}, {**Personen**, **Angestellte**}, . . .
- Konflikt {**Angestellte**, **Gäste**}:
  - definiere Klasse SC_{AG} := **Angestellte** ∩ **Gäste**
  - definiere Operation $Frist$ in Zustandstyp von SC_{AG}
  - ordne SC_{AG} in Klassenhierarchie ein

Konfliktauflösung: Beispiel II

Aufruf: Methode *Frist* für Objekt $o \in$ Personen, Angestellte, Gäste

- $o \in$ Personen, Angestellte, Gäste $\Rightarrow o \in SC_{AG}$
- Minimum-Bildung liefert $\{SC_{AG}\}$
- $t.res$ damit eindeutig

Aufruf: Methode *Frist* für Objekt $o \in$ Personen, Studenten, Gäste

- Minimum-Bildung liefert {Studenten, Gäste}
- kein Konflikt $\Rightarrow t.res$ eindeutig

Verhaltensmodell: Migration

Objekt: Wert von Old, noch kein Zustand

- Erzeugen: Transaktionsschritt $create$
- Löschen: Transaktionsschritt $destroy$, Persistenz, Konsistenz

Migration: Meta-Methode $migrator_C$: Old $\rightarrow X_C \rightarrow$ Tx

- Vorbedingung: Zielobjekt ist in allen Oberklassen von C
- $migrator_C\, o\, v$ fügt Tupel $\langle o, v \rangle$ in Zustandsfunktion von C ein

Demigration: Meta-Methode $demigrator_C : C \rightarrow$ Tx

- $migrator_C\, o$ löscht Tupel $\langle o, v \rangle$ aus Zustandsfunktion von C
- Konsistenzproblem: Klassentyp C mit Bedingung zu Extension

Demigration: Typsicherheit

Beziehungen: Konsistenzprüfung am Transaktionsende

- lokale Bedingung ⇒ inkonsistente Tupel werden entfernt
- Strategie Abbruch ⇒ Objekt darf nicht demigrieren

Funktionsargumente: werden ungültig

- kritische Funktion einer Klasse C: demigriert Objekt aus C
- Klassenhierarchie: Funktion auch kritisch für alle Unterklassen
- nach Anwendung einer kritischen Funktion
  - keine Anwendung einer Methode für C erlaubt
  - aber Methoden von Oberklassen von C
  - oder explizite Typprüfung
- alle kritischen Funktionen sind Transaktionsschritte

⇒ nur sequentieller Kontrollfluß durch $bind$

Demigration: kritische Funktionen

alle Klassen: `destroy`

Klasse mit Demigrator: der Demigrator

Klasse mit Konsistenzbedingung: Änderungsmethoden
abhängig von Bedingung ⇒ Methoden-Semantik beachten

- Axiome der Zustandstypen
- Aggregatfunktionen der Anfragemethoden

Sichtklasse: siehe Klasse mit Konsistenzbedingung

Demigration: Beispiele

Beziehung `Ausleihe(Student,Buch)`

- geeignete Strategie: Transaktionsabbruch
- Person demigriert von `Student`, hat noch Buch $\Rightarrow$ Abbruch

Beziehung `Besucher(Student,Club)`

- geeignete Strategie: Löschen
- Person demigriert von `Student` $\Rightarrow$ wird aus `Besucher` gelöscht

Funktion mit Argument vom Typ `Student`

- Funktion wendet kritische Funktion an

$\Rightarrow$ keine Methodenanwendung auf Bezeichner vom Typ `Student`

- oder: vorher explizite Typprüfung `o in Student`
- Problem: zu scharf; `destroy` ist kritisch für alle Klassen

Effizienz

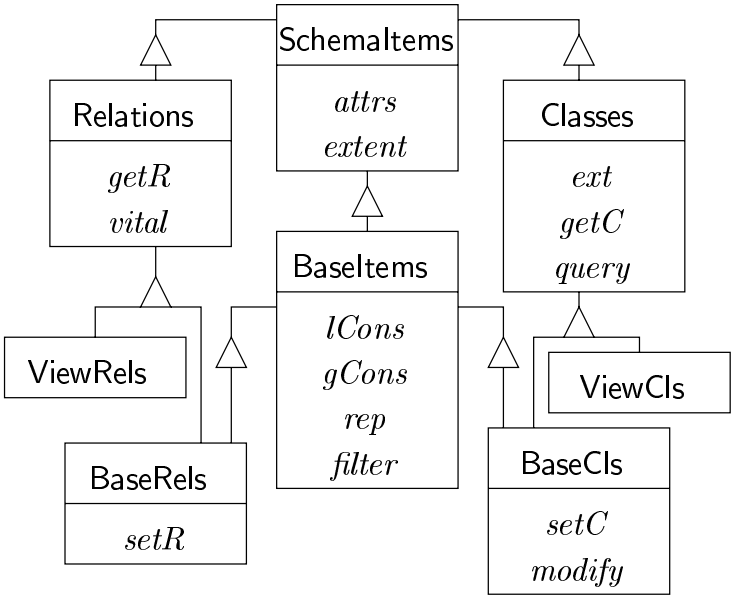
lazy evaluation: nur berechnen, was gebraucht wird

boxed values: „kleine“ Werte sofort berechnen

partial evaluation: möglichst zur Übersetzungszeit auswerten

stingy evaluation: Ressourcen-orientiert auswerten

Meta-Schema



- komplett mit vorgestellten Mitteln definiert
- definiert alle vorgestellten Mittel
- erweiterbar (symmetrische, reflexive Beziehungen)
- liefert Schema-Informationen mit Meta-Methoden

Fazit

- Datenmodell: Kombination bewährter Elemente
  - Klassen: split instance model (COCOON, OpenODB, Iris, Chimera)
  - Beziehungen (DSM, ADAM, FDM, Fibonacci)
- Verhaltensmodell: neu
  - Transaktionen als Monad-Werte
  - Operationen, Methoden, Funktionen
  - Aggregation in Methoden
  - spätes Binden trotz Datenmodell, Konfliktauflösung
  - typsichere Migration mit Reklassifikation, Korrektur von Referenzen
  - deklaratives Persistenzmodell für Beziehungen