Natürliche Persistenz als Verallgemeinerung des modularen objektorierterten Programmiermodels

Ein Schritt zu einem mächtigeren Persistenzkonzept

Luc Bläser

Programmierung mit natürlicher Persistenz

Einheitliches Programmiermodel

- Gleiche Modellierung und Verwendung von Daten, unabhängig ob persistent oder transient.

Umfassende Memory-Safety

- Volle Speicherverwaltung auch für persistente Daten, keine Dangling References, keine Memory Leaks.
- Gesicherte Konsistenz von persistenten Daten

Semantische Lücke zwischen Datenbanken und Programmiersprachen

Benötige sprach-entkoppelten APIs

Zwei separate Datenmodelle

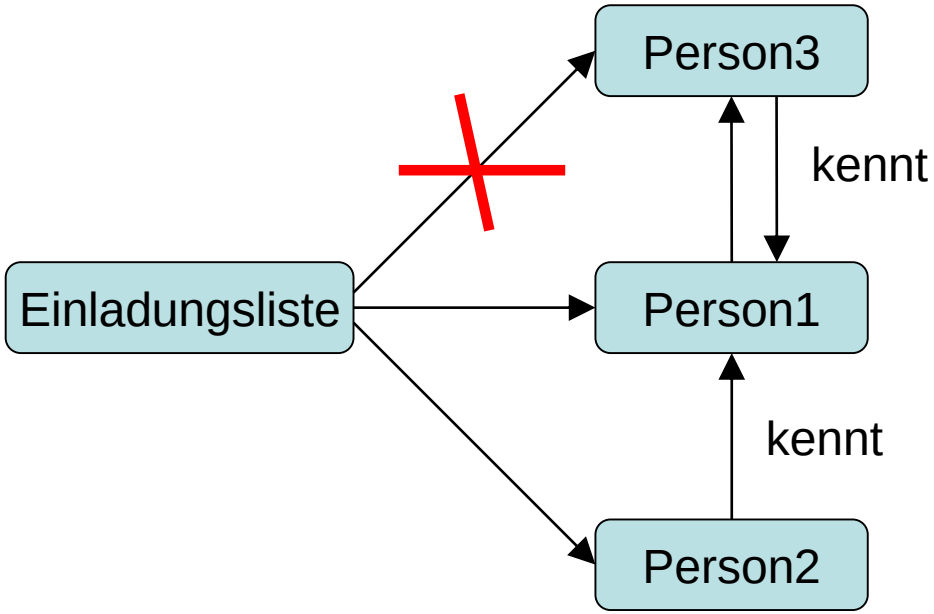
- Inkonsistenzen beim Mapping
- Eingeschränkte Programmerweiterungen
- Änderungen an Daten müssen explizit persistiert werden

Keine Memory-Safety für objektorientierte Sprachen

- Gefahr von Dangling References
- Garbage Collection von persistenten Daten erfordert Interaktion mit Laufzeitsystem

Transaktionelle Isolation nicht auf Programmierebene verfügbar

Einfaches Beispiel: Einladungsliste



Gleiches Programm wie für transiente Daten

- Ausnahme: Transaktionen zur Bestimmung von atomaren Blöcken für Fault-Tolerance

Orthogonale Persistenz

1. Operationen auf persistente Daten sehen gleich aus wie auf transiente.
   - Kein Persistenz-API (auch nicht beim Programmstart)
   - Transaktionen sind auch auf transienten Daten möglich und sinnvoll
2. Persistenz ist unabhängig vom Typ verfügbar
   - Keine bestimmtes Interfaces (oder Subtyping)
3. Persistenz bei Erreichbarkeit
   - Transitive Hülle von persistenten Objekten
   - Memory-Safe

Implizite Objektlebenszeiten (1)

Objekte haben persistente, transiente oder cache-bezogene Lebenszeiten, abhängig von der Erreichbarkeit

Referenzen haben unterschiedliche Stärken

- Attribute PERSISTENT, TRANSIENT, WEAK
- Standard ist PERSISTENT
- VAR x: {TRANSIENT} A; y: ARRAY 12 OF {WEAK} B; z: C;

default {PERSISTENT}

Module sind die Wurzel der Persistenz

- Konzeptionell unbeschränkte Lebenszeit

Implizite Objektlebenszeiten (2)

Objekt ist persistent ⇔ von Modulen über Kette von PERSISTENT-Referenzen erreichbar.

- Überlebt Neustarts und Systemfehler

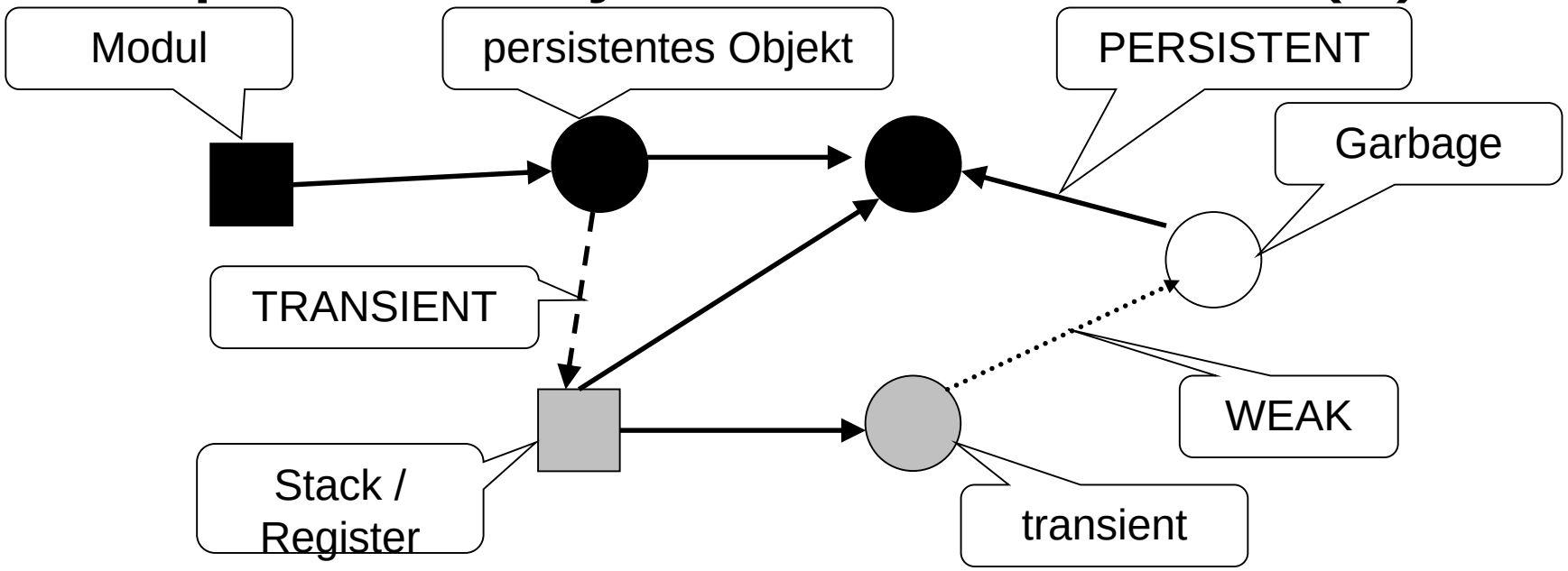
Objekt ist transient ⇔ nicht persistent und von Stack / Register über nicht-WEAK-Referenzen erreichbar

- Lebt innerhalb einer Systeminkarnation

Restliche Objekte sind Garbage

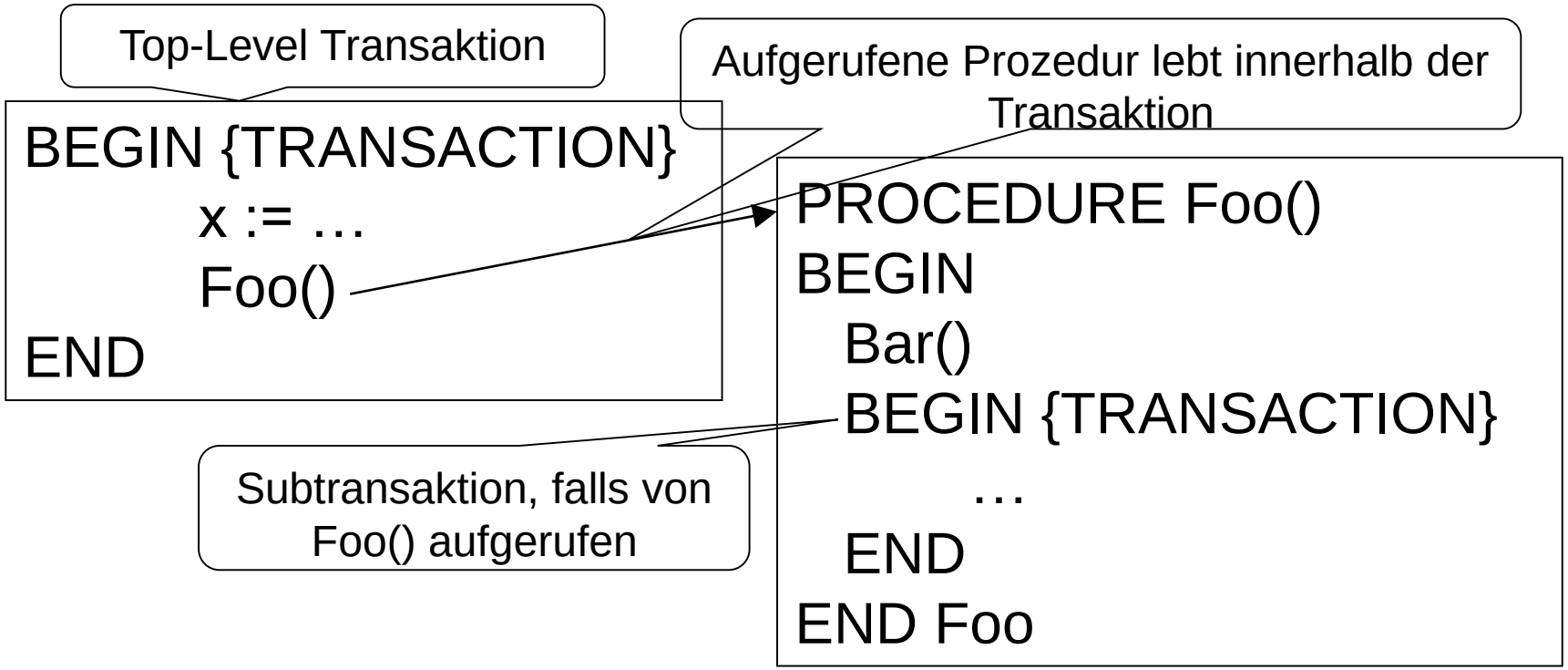
- Verspätet abgeräumt, spätenstens bei Speichernot
- Über WEAK-Referenzen als Cache zugreifbar

Implizite Objektlebenszeiten (3)



- TRANSIENT-Referenzen in persistenten Objekten werden bei Neustart auf NIL gesetzt
- WEAK-Referenzen werden auf NIL gesetzt, sobald referenziertes Garbage-Objekt gelöscht wird.
- Zuweisungs-Kompatibilität von Variablen mit unterschiedlichen Referenzattributen

Geschachtelte ACID-Transaktionen



Bei „normalem“ Verlassen einer Top-Level-Transaktionen werden Änderungen atomar und dauerhaft gespeichert.

Bei Trap oder explizitem ABORT werden Änderungen rückgängig gemacht.

Subtransationen definieren Safe-Points, die mit ABORT wiederhergestellt werden können.

Geschachtelte ACID-Transaktionen

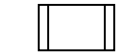
Transaktionen können nur durch Systemfehler oder explizit mit ABORT abgebrochen werden.

- Keine Starvation durch unerwartete Rollbacks des Transaktionsschedulers

Nicht-transaktionelles Verändern eines persistenten Objektes löst eine implizite Mini-Transaktion aus.

Isolation der zugegriffenen Objekte innerhalb einer Transaktion

- Gegenüber anderer Transaktionen
- Gegenüber nicht transaktionelle Zugriffe
- Azyklischer Informationsfluss bezüglich Lese- und Schreibfluss (semantische Serialisierbarkeit)



Evolution

Programänderungen invalidieren bestehende persistente Datenformate

Konsistenzsicherung im Lader mittels internen synthetisch erzeugten Schemas

Evolution zum Updaten auf neue Programmversion

- Variablen hinzufügen, umbenennen, entfernen, Typen hinzufügen, entfernen, Delegaten anpassen

Löschen aller persistenten Daten eins Moduls

System-Architektur

Compiler & Evolution

Höheres Laufzeitsystem

- Steuerung der persistenten Objektresidenz
- Lazy Object Loading
- Multi-Level Garbage Collection
- Transaktionsmanagement

Persistentes Objekt-System

- Fault-Tolerant Write-Ahead Logger
- Partitioniert, Endian-Format unabhängig

Multi-Level Garbage Collection

Transience Collector

Sammle nicht persistente Objekte in persistentem Objektsystem und mache sie transient.

- Inkrementell partitionierte Garbage Collection, blockiert nur persistentes Objektsystem während der Kollektion eines Objektes.
- Neues Modell für Concurrency mit laufenden Transaktionen und Hauptspeicher-Objektverwaltung
- Disk Garbage Collection (Mature Objekt Space) Speichere Objekte in Carriages und Trains, mit Remembered Sets und Reference Counting.
- Kompaktifizierung des Speicherbereichs
- Fault-Tolerant
- Sehr geringe Unterbrechungen mit garantierter Maximaldauer

Multi-Level Garbage Collection

Main-Memory Garbage Collection

Lösche Garbage im Hauptspeicher

- Erweiterung des klassischen Stop-And-Go-GCs
- Zurücksetzen von Weak-Referenzen
- Nur Objekte, die nicht im persistenten Objektsystem leben, dürfen aufgeräumt werden.
- Tracen von persistenten Referenzen

Beispiel Medizinalsystem (1)

Natürlichere Modellierung als mit OM

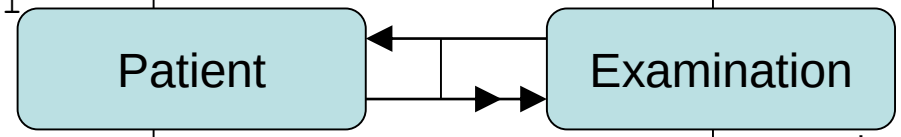
- Direkte Referenzen, Instanzen besitzen eigene Subcollections anstatt globaler Associations und Collections
- Weniger Suche in Collections / Associations nötig
- Im Gegensatz zu OM keine Dangling References oder Memory Leaks
- Einheitliches konsistentes Datenmodel
- Updates werden einfach und gesichert mit Transaktionen ausgeführt

Beispiel Medizinalanwendung (2)

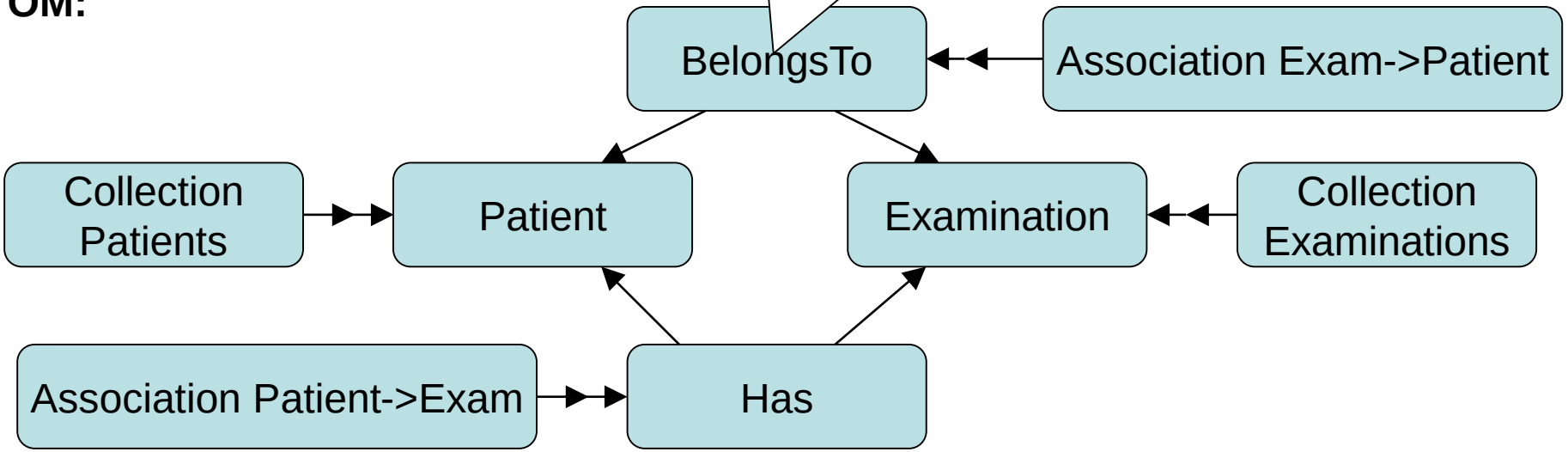
Persistente Programmiersprache

```
Patient = OBJECT
VAR
 exams: Collection;
 (*of Examination*)
END Patient
```

```
Examination = OBJECT
 patient: Patient;
END Examination
```



OM:



Beispiel Medizinalanwendung (3)

Codegrössenvergleich:

Kern mit Datenmodel, Business Logic, Model View Controller (ohne GUI, Verteilung)

- (ohne Kommentar, Tracings, Whitespaces etc.)

Original Version 61‘500 Zeichen

Mit Ausschluss von Code für Verteilung (Serailisierung, Object Identifiers, Channelling) etc., obwohl diese auch für Persistenz verwendet werden.

Neue Version 17‘100 Zeichen

~72% Einsparung

Offene Probleme

- Main-Memory Garbage Collection
  - Stop-And-Go führt zu längeren Unterbrechungen wegen De-referzierung von persistenten Identifiers, skaliert nicht für sehr hohe Hauptspeicherlast.
- Inter-Transaktionelle Concurrency
  - Bis jetzt nur serielles Scheduling, inter-transaktionelle Concurrency jedoch konzeptionell vorbereitet.
  - Brauche **Preclaiming** Two-Phase-Locking für Isolation, um unerwartete Abbrüche wegen Verzahnung auszuschliessen.
  - Low-cost Field-Analysis dafür sinnvoll. -> brauche jedoch Mehrphasenkompilation