Vorlesung Implementierung von Datenbanksystemen

7. Zugriff auf Datenbanken aus einem Programm

Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

Lehrstuhl für Informatik 6 (Datenmanagement)
Friedrich-Alexander-Universität Erlangen-Nürnberg

7.1 Relationale Datenbanken

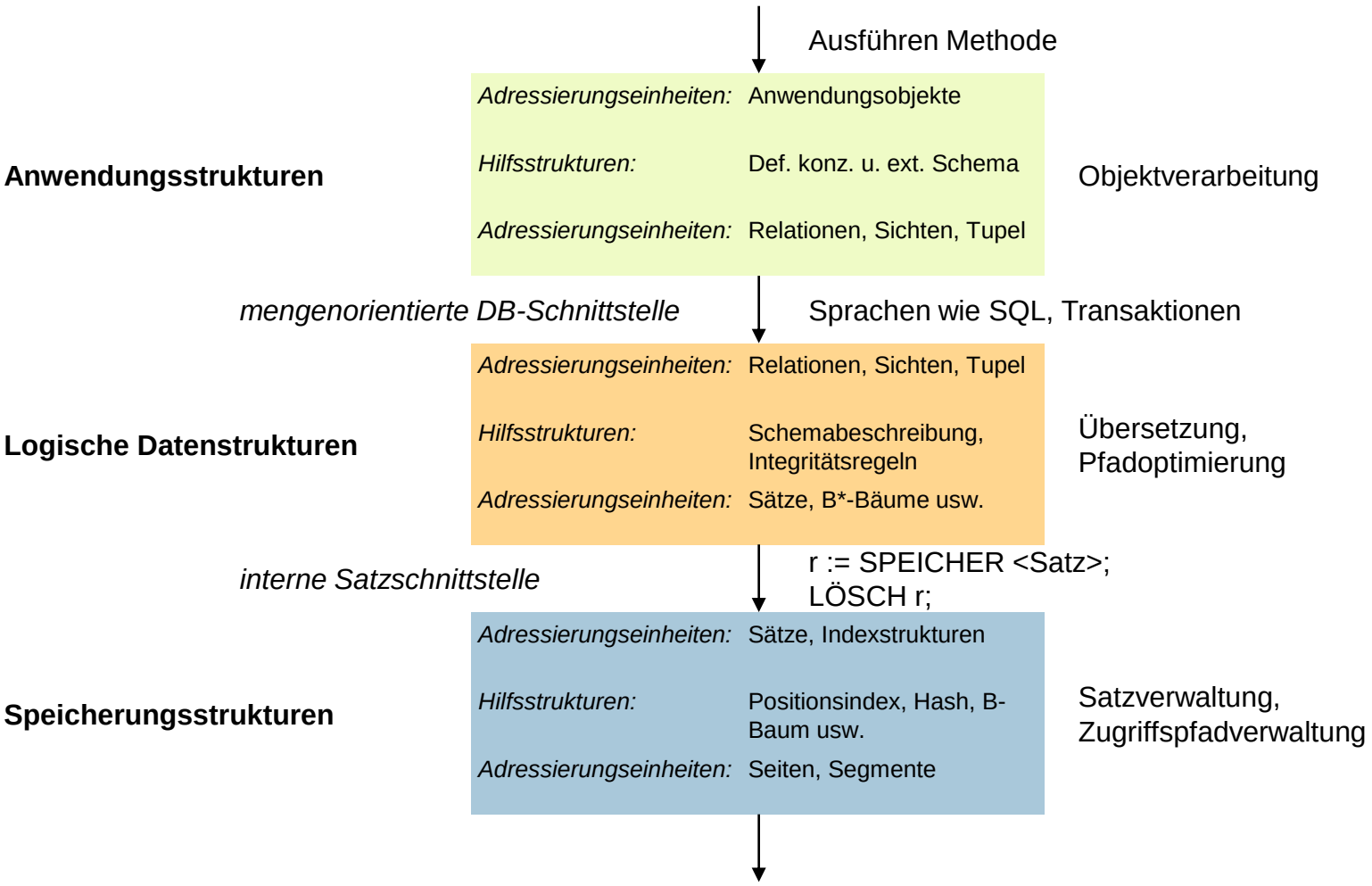
- **Bisher im Schichtenmodell erreicht:**
  Interne Satzschnittstelle
  - Benutzer (Programmierer) arbeitet beim Ablegen von Daten auf dem Hintergrundspeicher mit den "virtuellen" Objekten Segment (Datei) und Satz
  - **Segment** = Menge von Sätzen,
    Satz = variabel lange Folge von Bytes, evtl. mit Schlüssel

- **Operationen (Dienste):**
  - **Sequenzielles Lesen** aller Sätze (eines Typs, z.B. alle Kunden)
    - Reihenfolge durch Speicherungssystem festgelegt
  - **Direktzugriff über Satzadresse**
    - Wird vom Speicherungssystem vergeben
  - **Direktzugriff über Schlüssel** (= Teile der Sätze, Felder)
    - Wenn entsprechende Primär- oder Sekundär-Organisation vorhanden

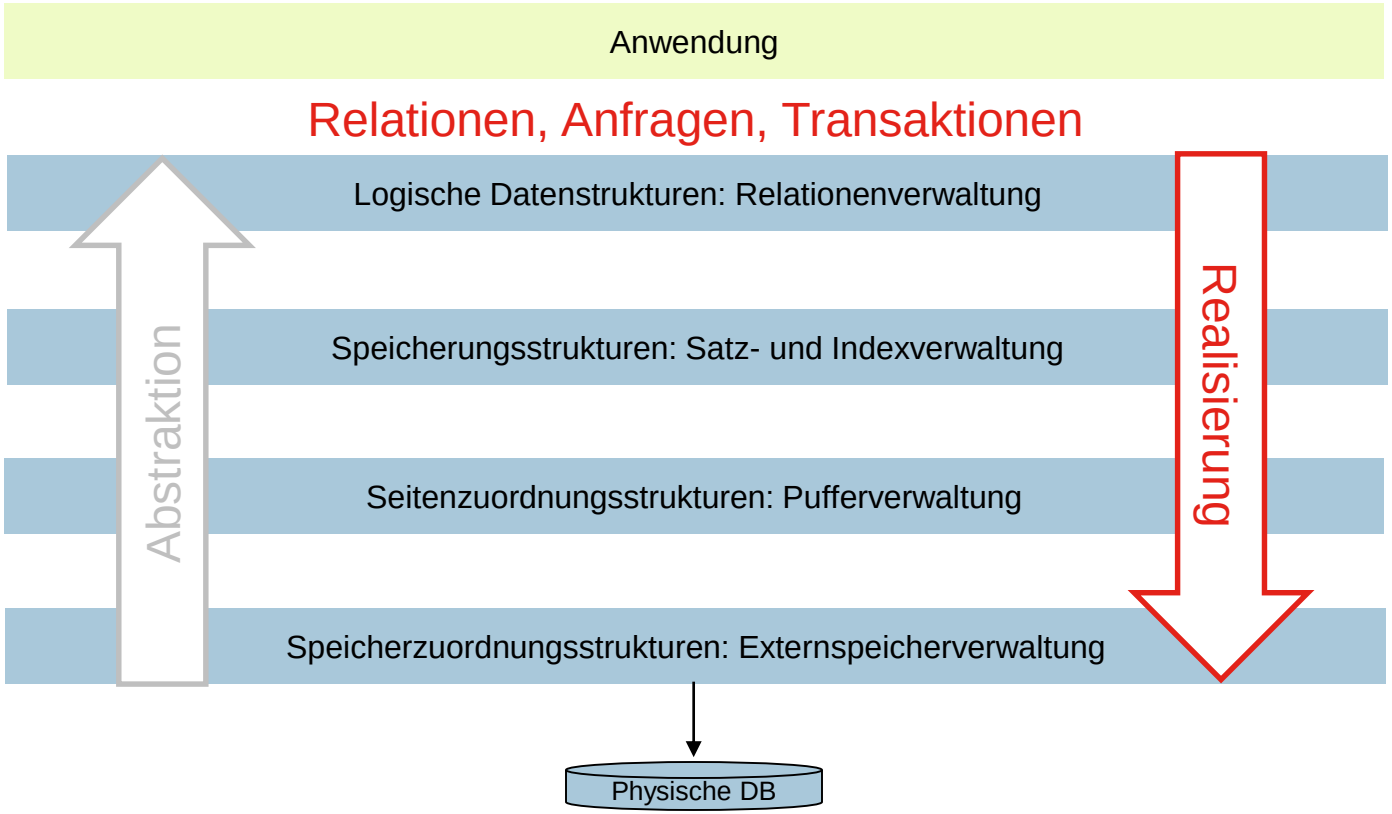
Relationale Datenbanken (2)

- **Nächster Schritt:**
  - Unabhängigkeit der Programme (Anwendungen)
    auch noch von Organisationsformen (Segmenttypen) und Indexen!
    - Sehen **nur noch Satzmengen**
    - **Direktzugriff über beliebigen Schlüssel immer möglich**
      - Speicherungssystem wählt intern Index aus oder sucht sequenziell
  - Noch mächtigere Operationen zur Auswertung:
    - Boolesche Ausdrücke zur Auswahl von Sätzen
    - Verknüpfung von Sätzen verschiedener Dateien
      (über gleiche Schlüsselwerte; Join)
- **Bedeutet weitere Abstraktion:**
  - Von Dateien zu **Relationen** (oder Klassen)
  - Von Sätzen zu **Tupeln** (oder Objekten)
  - Von Feldern (Schlüsseln) zu **Attributen**

Resultierende Schicht



Weitere Vorgehensweise



7.2 Programmzugriff

- **Ausführung von SQL-Anweisungen (Anfragen) in Programmen**
  (Hier am Beispiel von C)
  - Nicht Bestandteil der Programmiersprache
    - Durch **Vorübersetzer** (Precompiler) in normales C (hauptsächlich Funktionsaufrufe) transformiert
  - Daher spezielle Kennzeichnung im Programmtext:

    ```
    EXEC SQL <SQL-Anweisung> ;
    ```
  - Ergebnisse einer SELECT-Anweisung müssen in **Programmvariablen** abgelegt werden:
    - INTO-Klausel
    - Kennzeichnung der Variablendeklaration und -verwendung für den Vorübersetzer (Typ-Überprüfung!)
    - Programmvariablen werden zur Unterscheidung von Attributen durch vorangestellten **Doppelpunkt** gekennzeichnet.
    - Auch in der WHERE-Klausel dürfen Programmvariablen verwendet werden!

Programmzugriff (2)


```
EXEC SQL BEGIN DECLARE SECTION;
    int anr;
    char nachname[31];  /* Platz für \0 */
EXEC SQL END DECLARE SECTION;

EXEC SQL SELECT Nachname, ANr
        INTO :nachname, :anr
        FROM Personen
        WHERE PNr = 3498;
```

- Zu Beginn des Programms:

```
EXEC SQL INCLUDE SQLCA;
```

  - Einkopieren einer Struktur-Definition für die **"SQL Communication Area":** Speicherbereich u.a. für Fehlermeldungen von SQL

Cursor

- SELECT liefert i.Allg. Menge, möglicherweise sehr groß
  - Kann nicht auf einen Schlag ins Programm übernommen werden
- Lösung: Tupel für Tupel abrufen

- **Definition eines Tupelzeigers (Cursors) für eine SELECT-Anweisung:**

```
EXEC SQL DECLARE c1 CURSOR FOR
     SELECT Nachname, ANr
     FROM Personen
     WHERE Gehalt < :limit;
```

  - Wird erst beim Öffnen des Cursors ausgeführt:

```
EXEC SQL OPEN c1;
```

Cursor (2)

- Abruf der einzelnen Tupel mit FETCH-Anweisung:

```
EXEC SQL FETCH c1 INTO :nachname, :anr;
```

  - Füllt die angegebenen Programmvariablen
    (die wie gehabt in einer DECLARE SECTION aufgeführt sein müssen)
- Nach letztem Tupel
  - führt weiteres FETCH zum Fehlercode 100
    in der Komponente SQLCODE der SQLCA**.**
- Cursor schließen:

```
EXEC SQL CLOSE c1;
```

  - Anschließend ggf. neu öffnen (z.B. mit neuem Wert in **limit**)

Beispielprogramm


```
main () {
   exec sql include sqlca;
   exec sql begin declare section;
        char pnr[8];
        char nachname[31];
        int anr;
   exec sql end declare section;
   exec sql declare c1 cursor for
        select PNr, Nachname, ANr from Personen;
   exec sql open c1;
   printf ("%-3s  %-20s  %-4s\n", "PNr", "Name", "Abt");
   for (;;) {
        exec sql fetch c1 into :pnr, :nachname, :nr;
        if (sqlca.sqlcode == 100) break;
        printf ("%-3s  %-20s  %4d\n", pnr, nachname, anr);
   }
   exec sql close c1;
}
```

Einschätzung

- **Genormt**
  - Portabilität der Anwendungsprogramme
- **Relativ kompakt zu programmieren**
  - Vergleiche nächsten Abschnitt …
- **Erlaubt die Erzeugung von Zugriffsmodulen**
  - SQL-Anweisungen schon zur **Übersetzungszeit** durch Vorübersetzer an DBS übergeben
  - Analysieren, optimieren, in Zwischencode überführen, im DBS ablegen
  - Zur Laufzeit dann sehr effizient:
    Nur noch ausführen
    - Aufruf über Namen:

```
call dbs (zugriffsmodul43, parameter);
```

- **Keine SQL Injection möglich**

7.3 Unterprogrammaufruf

- **Neben Spracherweiterung**
  - Wie eben gesehen
  - Wird auch "Eingebettetes SQL" (Embedded SQL) genannt
  - Durch Vorübersetzer (precompiler) in normales C transformiert
- **Auch noch Unterprogramm-Schnittstelle:**
  - **Call-level interface (CLI)** eines DBS
    - Bzw. Application-programming Interface (API) eines DBS
  - Prinzip:

```
call DBS ("select ... from ... where ... ");
```

  - Einfacher zu realisieren für den DBS-Hersteller,
    mehr Arbeit für den DB-Anwender

Beispiel JDBC

- **Call-level Interface für die Programmiersprache Java**
- **Angelehnt an ODBC**
  - "Open Data Base Connectivity";
    Microsoft-"Standard" für Datenbank-Zugriff
  - **Aber**: "JDBC" ≠ "Java Data Base Connectivity",
    kein Akronym, sondern geschützter Name ...
  - Übersichtlicher, einfacher (Nutzung von Klassen, streng typisiert)
- **Package java.sql**

java.sql

- **java.sql.DriverManager**
  - Treiber registrieren
    - implementieren eine gemeinsame Schnittstelle (abstrakte Klasse) auf jeweils einer bestimmten Datenbank
  - Verbindungen zur Datenbank aufbauen
- **java.sql.Connection**
  - verwaltet dann bestehende Verbindung zu einer Datenbank
- **java.sql.Statement**
  - Ausführung einer SQL-Anweisung über eine bestehende Verbindung
- **java.sql.ResultSet**
  - verwaltet Ergebnis einer Anfrage in Form einer Relation
  - Zugriff auf einzelne Attribute

JDBC am Beispiel

- **Prinzipieller Ablauf einer JDBC-Datenbankanwendung:**
  - Aufbau einer Verbindung zur Datenbank
  - Senden einer SQL-Anweisung
  - Verarbeiten der Anfrageergebnisse
- **Zuerst Laden des Treibers:**
  - `Class.forName ("oracle.jdbc.driver.OracleDriver");`
- **Verbindungsaufbau:**
  - Methode **getConnection** der Klasse **java.sql.DriverManager**
  - Argumente:
    - URL mit Verbindungsmechanismus und Treiber (muss geladen sein)
    - Benutzername und Passwort

```
String url = "jdbc:oracle:thin:@rechner1:1521:db1";
Connection con =
      DriverManager.getConnection (url, "jupp", "1%Ueg&");
```

JDBC am Beispiel (2)

- **SQL-Anweisung:**

```
String anfrage = "SELECT PNr, Nachname, ANr FROM Personen";
```

  - Anmerkung: SELECT * ist hier keine gute Idee – was käme da zurück?

```
Statement anweisung = con.createStatement ();
ResultSet ergebnis = anweisung.executeQuery (anfrage);
```

- **Verarbeitung des Ergebnisses:**
  - Die meisten Programmiersprachen (so auch Java) können Mengen nicht direkt verarbeiten.
  - Deshalb **Iteration** (Schleife):
    - Nacheinander jedes einzelne Tupel der Ergebnismenge
    - Methode **next** der Klasse **ResultSet** positioniert auf das nächste Tupel.
  - Zugriff auf Spalten (Attribute einer Relation) mit typspezifischer **get-**Methode

```
while (ergebnis.next ()) {
  int pnr = ergebnis.getInt (1);
  String name = ergebnis.getString (2);
  int anr = ergebnis.getInt (3);
  System.out.println (pnr + " " + name + " " anr); }
```

JDBC am Beispiel (3)

- **Am Schluss Ressourcen freigeben:**

```
ergebnis.close ();
anweisung.close ();
```

  - Optional
    - Freigabe auch durch automatische Speicherbereinigung von Java
  - Allerdings guter Programmierstil
- **Fehler**
  - werden in JDBC als Ausnahme der Klasse **SQLException** signalisiert
  - Mit einem **try-catch**-Block abfangen
  - Details zum Fehler mit **getMessage** ermitteln

```
try {
  // Aufruf von JDBC-Anweisungen
  } catch (SQLException exc) {
          System.out.println ("SQLException: " +
                                  exc.getMessage () );
  }
```

Ausführen von Anweisungen

- **Schnittstelle "Statement"**
  - Drei **execute**-Methoden:

```
ResultSet executeQuery (String sql)
    throws SQLException;
```

  - Siehe oben
  - Lesende Anfrage (SELECT)

```
int executeUpdate (String sql)
    throws SQLException;
```

  - Ändernde Anweisung (CREATE TABLE, INSERT, DELETE, UPDATE)
  - Liefert die Anzahl der geänderten Tupel zurück

```
boolean execute (String sql)
    throws SQLException;
```

  - Für Anweisungen, die mehrere Ergebnisse liefern (gespeicherte Prozeduren)

Vorübersetzte Anfragen

- **Mehrfache Ausführung von Anfragen**
  - Evtl. mit unterschiedlichen Parametern (in Programmvariablen)
- **Schnittstelle PreparedStatement**
  - Von **Statement** abgeleitet
  - Schon beim Erzeugen SQL-Anweisung angeben:

```
PreparedStatement einfuegen =
  con.prepareStatement ("INSERT INTO Personen " +
     "VALUES (?, ?, ?, ?, ?, ?, ?)");
```

  - Wird sofort analysiert, übersetzt, optimiert, in Zwischencode überführt, ...
  - Vor Ausführung dann noch Parameterwerte (für die Fragezeichen) setzen:

```
\\ java.sql.PreparedStatement
void setBoolean (int paramIdx, boolean b)
  throws SQLException;
void setString (int paramIdx, String s)
  throws SQLException;
```

Vorübersetzte Anfragen (2)

- **Schnittstelle PreparedStatement (Forts.)**
  - Parameter von 1 an durchnummeriert

```
einfuegen.setInt (1, 8967);
einfuegen.setString (2, "Karl Numpf");
einfuegen.setString (3, "Wunsdorf");
einfuegen.setInt (4, 57);
...
```

  - Anschließend wie gewohnt ausführen

```
einfuegen.executeUpdate ();
```

  - Alle Methoden von **Statement** verfügbar, allerdings überschrieben: ohne Parameter
  - Dann Parameter neu setzen und nochmal ausführen

- **Wichtige Optimierung**
  - Erheblicher Anteil der Verarbeitung nur einmal beim Erzeugen

Gespeicherte Prozeduren

- **Weitere Optimierung**
  - Vorübersetzte Anfragen müssen immer noch bei jedem Programmlauf neu analysiert, übersetzt und optimiert werden.
  - Idee: Anfrage in der Datenbank ablegen, mit einem Namen und Parametern versehen
    - Unabhängig von den Anwendungsprogrammen
    - Die rufen die Anfrage nur noch unter ihrem Namen auf.
  - Analyse und Optimierung dann nur noch einmalig
  - "Stored Procedures"
- **Schnittstelle CallableStatement**
  - Abgeleitet von PreparedStatement
  - Objekt erzeugen über Methode **prepareCall** von **Connection**

```
CallableStatement call =
      con.prepareCall ("{ call prozedur1 }");
```

  - Sog. Escape-Syntax (geschweifte Klammern); wird von JDBC-Treiber übersetzt in die spezielle Syntax eines Datenbanksystems

Gespeicherte Prozeduren (2)

- **Parameter**
  - Wieder mit Fragezeichen als Platzhalter:

```
{ call prozedur2 (?, ?) }
{ ? = prozedur3 (?) }
```

  - Alle **set**-Methoden von **PreparedStatement** verfügbar; nutzbar für **in**-Parameter
  - Für **out**- **und inout-**Parameter vor der Ausführung JDBC-Typ festlegen:

```
void registerOutParameter (int paramIdx, int jdbcType)
   throws SQLException;
```

  - Beispiel:

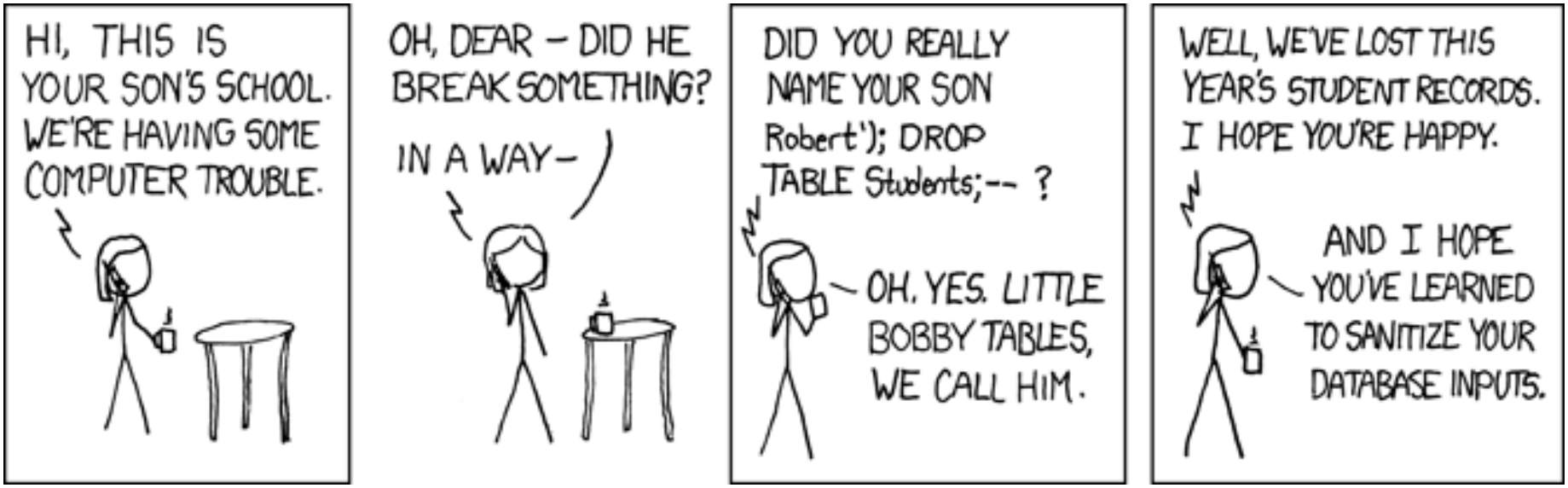
```
CallableStatement aufruf =
   con.prepareCall ("{ call Test (?, ?) }");
aufruf.setDouble (2, 42.0);
aufruf.registerOutParameter (1, java.sql.Types.VARCHAR);
aufurf.registerOutParameter (2, java.sql.Types.FLOAT);
```

  - Lesen der **out**- und **inout**-Parameter mit **get**-Methoden

Einschätzung

- **Relativ elementar**
- **Programmierer muss viel selbst machen**
- **Optimierung explizit:**
  - Direkter Aufruf oder
  - vorübersetzte Anfrage oder
  - Nutzung einer gespeicherten Prozedur
  - Wechsel nur durch Umprogrammierung!
- **Vorsicht mit SQL Injection!**
  - Wo kommt die Zeichenkette mit der SQL-Anfrage her? Direkt von der Eingabe?

SQL Injections



Quelle: http://xkcd.com/327/

7.4 Vergleich

- **Eingebettetes SQL**
  - Erweiterung der Programmiersprache: Vorübersetzer oder Extra-Compiler
  - Bequemer für Programmierer (kompakter)
  - Typüberprüfung und Optimierung schon zur Übersetzungszeit
    - Und das heißt: nur einmal
- **Unterprogrammaufruf**
  - Programmiersprache (auch "Wirtssprache" – host language – genannt) bleibt unverändert
  - Parameter für die Unterprogramme:
    alle Teile der SQL-Anweisungen einzeln
    oder SQL-Anweisung insgesamt als Zeichenkette – umständlich!
  - Typüberprüfung und Optimierung erst zur Ausführungszeit
    - Und das heißt: immer wieder

7.5 O/R-Mapping

- **Besonderheit für objektorientierte Programmiersprachen**
- **Objekte**
  - nicht selbst "zerlegen" und in Tupeln abspeichern.
  - Durch Werkzeuge erledigen lassen!
- **Konfigurationsdateien (in XML)**
  - sagen, welche Teile eines Objekts in welcher Relation gespeichert werden
- **Transfer der Daten zwischen Anwendung und Datenbank**
  - erfolgt automatisch,
  - d.h. ohne dass Anwendungscode dafür zu schreiben wäre!

Beispiel Java Persistence API

- **EJB 3.0**
- **Annotationen**

```
package  … ;
import javax.persistence.*;
@Entity
public class Personen implements java.io.Serializable
{
      private int PNr;
      private String Nachname;
      private int ANr;
      @Id
      public int getId( ) { return PNr; }
      public void setId(int PNr) { this.PNr = PNr; }
      …
}
```

Beispiel Java Persistence API (2)

- **Generiertes SQL:**

```
CREATE TABLE Personen (
     PNr            INT primary key,
     Nachname       VARCHAR(255),
     ANr            INT
);
```

  - sowie INSERT, UPDATE, DELETE usw.

- **Mehr dazu**
  - in der Vorlesung "Objektorientierte Datenbanken"