Software-Projekt. Realisierung

Skript mit didaktischen Lücken, 6. Auflage

von
Christoph Knabe
Technische Fachhochschule Berlin
Fachbereich VI (Informatik)
www.tfh-berlin.de/~knabe/

Die didaktischen Lücken sind durchnumeriert und mit mit [?*i*] gekennzeichnet, Bsp.: [?[1]]

1 ENTWURFSPHASE

Wiederholung: Welche sind die typischen Phasen in der Entwicklung eines Softwareprojekts?
[?[2]]

Def.: Entwurf ist Strukturierung des Systems aus Entwicklersicht:

- Zerlegung in Komponenten, ***Bsp***.: [?[3]]

- ihre Anordnung in Hierarchie
- ihre Spezifikation (WAS leisten sie?)
- Festlegung der Schnittstellen zwischen den Komponenten (Export, Import)

Dualität WAS vs. WIE

WIE arbeitet eine Komponente ("white box"-Sicht) wird erklärt durch [?[4]]

Bsp.: WIE arbeitet ein Telefon?

[?[5]]

Zum Analysemodell hinzukommende Aspekte

- Objektverwaltung
- (Zwischen-)Speicherung
- Effizienz

1.1 Objektverwaltung

ist [?[6]]

→ [Balzert99], p. 24

dazu folgende externe Operationen:

[Wiederholung:

intern?	[?[7]]
extern?	

]

- `erfassen()`: Geprüftes Erfassen eines neuen Objektes mit persistentem Abspeichern und Folgeoperationen
- `ändern()`: Geprüftes Ändern eines schon bestehenden persistenten Objekts mit Folgeoperationen
- `löschen()`: Geprüftes Löschen eines schon bestehenden persistenten Objekts mit Folgeoperationen
- `suchen()`: Geprüftes Suchen eines/mehrerer schon bestehender persistenter Objekte, statt Balzert: `erstelleListe()`

Im Entwurf müssen deren benötigte Varianten mit ihrer Signatur (Parametrierung) festgelegt werden, es sollten jedoch nur Objektoperationen verwendet werden, da Klassenoperationen in Java nur Stiefkinder sind (können z.B. nicht **`abstract`** sein). ***Bsp***.:

```
io_kunde.löschen()                vs.
lgApplication.löschen(io_kunde)   vs.
lgApplication.löschen(i_kundeId)
```

Varianten von `suchen()`

Von der Operation `suchen()` müssen im Entwurf alle benötigten Varianten festgelegt werden, ***Bsp***.:

```
lgApplication.kundeSuchen(i_kundeId: String): LgKunde
                                        throws LgKundeUnbekanntExc

lgApplication.kundeSuchen(i_telefonNr: String): LgKunde
                                        throws TelefonnrUnbekanntExc

lgApplication.kundenSuchen(i_namensanfang: String): LgKunde_List
```

1.2 Drei-Schichten-Architektur

→ [Balzert99], Kap. 10, p. 370 ff.

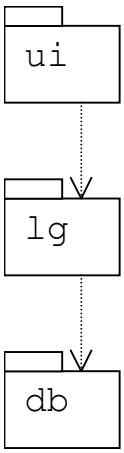
Grundlegende Entwurfsentscheidungen

- Programmiersprache: {Java, C++, C#, VB, PHP, Skriptsprachen, ...}
- UI-System: {Java/Swing, JSP, Windows, Tcl/Tk, ...}
- DB-System: {SQL, OO-DB, ...}

Entwurfsprinzipien für Komponenten

- Minimale Schnittstellen
- Austauschbarkeit ⇒ Zusammenhängendes in einer Komponente, Schnittstelle als Interface
- Testbarkeit ⇒ möglichst als API (Application Programming Interface: Unterprogrammschnittstelle)
- Wiederverwendbarkeit

Ergebnis: Strenge 3-Schichten-Architektur



User Interface: Reine Darstellung, Ereignisbehandlung, I/O;
ohne Verarbeitung

Applikationslogik (Fachkonzept) als API: Buchungen;
ohne UI-, DB-Aspekte, SQL

Datenbankzugriffe als API: Reines Speichern/Suchen; ohne Verarbeitung

Regeln

- `lg`-Schicht exportiert nur geprüfte Operationen, garantiert die Einhaltung der Geschäftsregeln.
- `ui`-Schicht darf nicht auf `db`-Schicht zugreifen, damit die Prüfungen nicht umgangen werden können.
- `ui`-Schicht sollte die Prüfungen durch `lg`-Schicht nicht duplizieren (Bsp. Plausi-Prüfungen).
- `lg`-Schicht exportiert an `ui`-Schicht nur `lg`-Objekte, für deren Erzeugung aus elementaren Daten exportiert sie spezielle, geprüfte Konstruktoroperationen, ***Bsp***.:

```
class LgDatum {
   static LgDatum fromNumeric(String i_string){…} //jjjjmmtt
   static LgDatum fromIso    (String i_string){…} //jjjj-mm-tt
   ...
}
```

1.2.1 Vom Analysemodell zum Entwurf

Aus einer in der Analyse gefundenen Klasse (***Bsp***. Konto) werden im Entwurf Klassen in allen drei Schichten.

- In der **Oberflächenschicht** ist typischerweise je Dialog eine Klasse nötig (***Bsp***. `UiKonto`, `UiKontoÜberweisung`), bei Java Server Pages (JSP) zusätzlich zur eigentlichen JSP. Außerdem Klassen für wiederkehrende Elemente von Dialogen (***Bsp***. `UiKontoList`, `UiButton`).
- In die **Logikschicht** wird die Fachklasse aus der Analyse mit allen ihren Operationen übernommen (***Bsp***. `LgKonto`), um die Objektverwaltungsoperationen sowie bei Bedarf ergänzt oder zwecks Schichtentkopplung bei top-down-Entwicklung (→ Kapitel 1.3.4) in Interface und Implementierung aufgespalten. Jede Klassenoperation (***Bsp***. `erfassen`, `suchen`) sollte dabei durch Verlagerung in ihre Ausgangsklasse zu einer Objektoperation umgewandelt werden.
- In der **Datenbankschicht** haben wir für Sprachen mit Reflection wie Java eine generische Klasse `DbObject`, die für eine beliebige Lg-Klasse deren Attribute persistent machen kann. Außerdem benötigen wir gemäß [Siedersleben] ein Interface `Pool`, das den operativen Zugriff auf den Persistenzmanager kanalisiert.

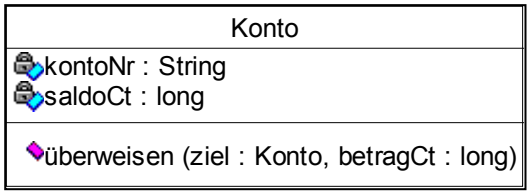
In Sprachen ohne Reflection wie C++ kann man `DbObject` nicht generisch lösen, sondern muß je Fachklasse eine Db-Klasse schreiben/generieren, die die Attribute der OOA-Klasse und die Operationen `dbInsert()`, `dbUpdate()` und `dbDelete()` enthält. Auch in Java wird dieser Ansatz häufiger gewählt („Entity Bean“), obwohl ich ihn für zu aufwändig halte.

Daneben gibt es Hilfsklassen folgender Arten:

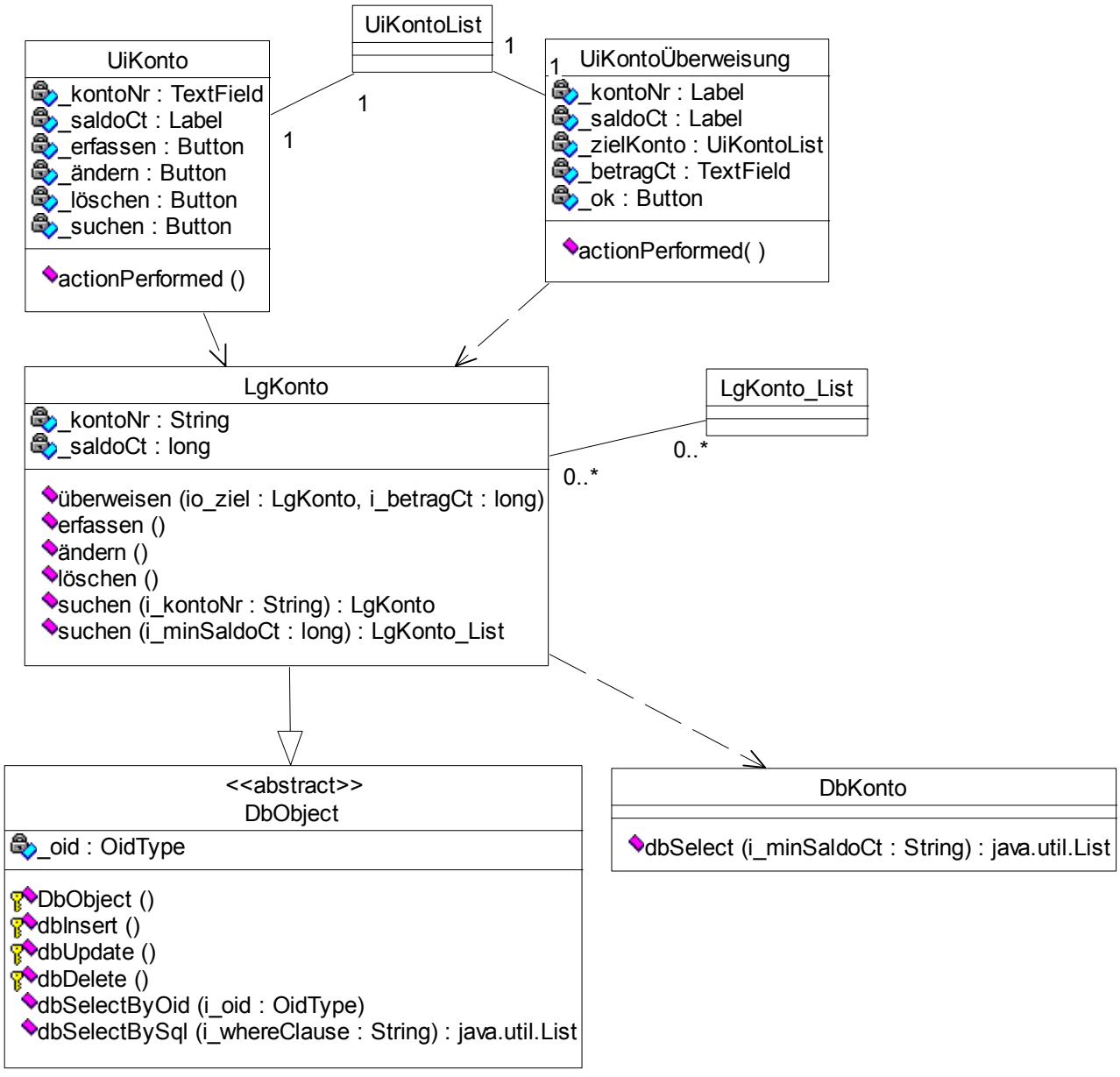
- Zur Zwischenspeicherung von jeweils einer Menge von Objekten einer Fachklasse, ***Bsp*** `LgKonto_List`.
- "Value-Klassen" ohne Persistenz zur Aufnahme von konzeptionell primitiven Werten, ***Bsp***. `LgDatum`.

Siehe dazu das folgende Bild.

OOA
.


Konto
kontoNr : String
saldoCt : long
überweisen (ziel : Konto, betragCt : long)

⇒ OOD:


UiKontoList
UiKonto
_kontoNr : TextField
_saldoCt : Label
_erfassen : Button
_ändern : Button
_löschen : Button
_suchen : Button
actionPerformed ()
UiKontoÜberweisung
_kontoNr : Label
_saldoCt : Label
_zielKonto : UiKontoList
_betragCt : TextField
_ok : Button
actionPerformed( )
1
1
1
1
LgKonto
_kontoNr : String
_saldoCt : long
überweisen (io_ziel : LgKonto, i_betragCt : long)
erfassen ()
ändern ()
löschen ()
suchen (i_kontoNr : String) : LgKonto
suchen (i_minSaldoCt : long) : LgKonto_List
LgKonto_List
0..*
0..*
<<abstract>>
DbObject
_oid : OidType
DbObject ()
dbInsert ()
dbUpdate ()
dbDelete ()
dbSelectByOid (i_oid : OidType)
dbSelectBySql (i_whereClause : String) : java.util.List
DbKonto
dbSelect (i_minSaldoCt : String) : java.util.List

1.3 Versionsverwaltung mit Subversion (SVN)

Probleme bei der Software-Wartung/Entwicklung:

- Produkt(teile) entwickeln sich, beim Kunden ältere Versionen im Einsatz, Reproduktion und Wartung dieser muß möglich sein ⇒ Versionsverwaltung nötig! Machbar in Subversion.
- Ähnliche, aber nicht gleiche Produkte (***Bsp***. Varianten für Windows, Unix, ...) ⇒ Konfigurationsverwaltung nötig! Machbar mittels Programmiertechniken, Skripten.
- Verteilte Entwicklung, ***Bsp***.: Open Source ⇒ Machbar in Subversion.

Begriffe in Subversion:

- Revision: Änderungsfolgenummer einer gesamten Filehierarchie (***Bsp***. 1329), automatische Vergabe.
- Version: Änderungszustand einer Gesamtauslieferung (***Bsp***. 4.0 oder "2000"), manuelle Vergabe.
- Verzweigung („branch"): Doppelung einer Version zum Zwecke der getrennten Weiterentwicklung,
  Bsp.: JDK 1.2α
  ↓
  JDK 1.2β
  ↓
  JDK 1.2final
  ↙ ↘
  JDK 1.2.1 JDK 1.3 α

Geschichte

- SCCS (Source Code Control System) mit Unix verbreitet, ca. 1980: Vorwärts-Diffs
- RCS (Revision Control System), ca. 1985: Rückwärts-Diffs, Sperren-Modifizieren-Freigeben
- CVS (Concurrent Versions System), ca. 1990: Kopieren-Modifizieren-Konfliktlösung-Freigeben
- SVN (Subversion), ca. 2004: Umbenennen (refactoring) unterstützt, modernere Zugänge (https)

1.3.1 Beispielszenario

Das Vorgehen bei der Verwaltung der Quelltexte eines Projekts mit Subversion kann durch folgendes Szenario (Stand Okt. 2005 im SWE-Labor) illustriert werden:

1. Download und Installation des Subversion Command Line Client ab Version 1.2.3 von subversion.tigris.org
2. `svn.exe` im PATH zugänglich machen.
3. Im SWE-Labor durch Herrn Heise ein Subversion-Projekt *projektName* mit seinen Benutzern einrichten lassen.
   Im SWE-Labor benutzen wir als Authentifizierungsmethode `https:`, als Server `pcx22` und das Repository `/work/svnroot`.
4. Während einer Internetverbindung die Anmeldeinformationen ausprobieren. Um Ihr frisch eingerichtetes leeres Projekt im Repository anzuschauen, browsen Sie in einem Web-Browser:
   `https://pcx22.tfh-berlin.de/work/svnroot/`*projektName*
   sodann Username und Passwort angeben. Es wird eine sichere Verbindung erzeugt, in der Sie den gesamten (z.Z. noch leeren) Inhalt Ihres Projekts anschauen können.
5. Den kompletten Projektinhalt aus Urquellenverzeichnis *projektName* mit *log message* in den Hauptentwicklungszweig `trunk` importieren, als *projektName* dient Ihr Projektname:
   CD *MutterverzeichnisDerUrquellen*
   DIR projektName

svn import *projektName*
`https://pcx22.tfh-berlin.de/work/svnroot/`*projektName*`/trunk`
-m "*log message*", ***Bsp***.:
```
svn import knabe1 https://pcx22.tfh-berlin.de/work/svnroot/knabe1/trunk
 -m "Projekteroeffnung"
```
Alle neu ins Repository eingetragenen Dateien werden mit Flag N protokolliert. Sie werden im Repository im Verzeichnis *projektName*`/trunk` abgelegt. Per Konvention sollte man den Hauptentwicklungszweig im Verzeichnis *projektName*`/trunk` im Repository ablegen. Nachprüfbar wie oben mit einem Browser.

6.Herstellung einer Arbeitskopie:
CD *MutterverzeichnisDerArbeitskopie*
svn checkout `https://pcx22.tfh-berlin.de/work/svnroot/`*projektName*`/trunk`, ***Bsp***.:
```
svn checkout https://pcx22.tfh-berlin.de/work/svnroot/knabe1/trunk
```
Im aktuellen Verzeichnis wird ein Unterverzeichnis `trunk/`*projektName* angelegt. Alle heruntergeladenen Dateien werden mit Flag A protokolliert. Die Internetverbindung kann beendet werden.

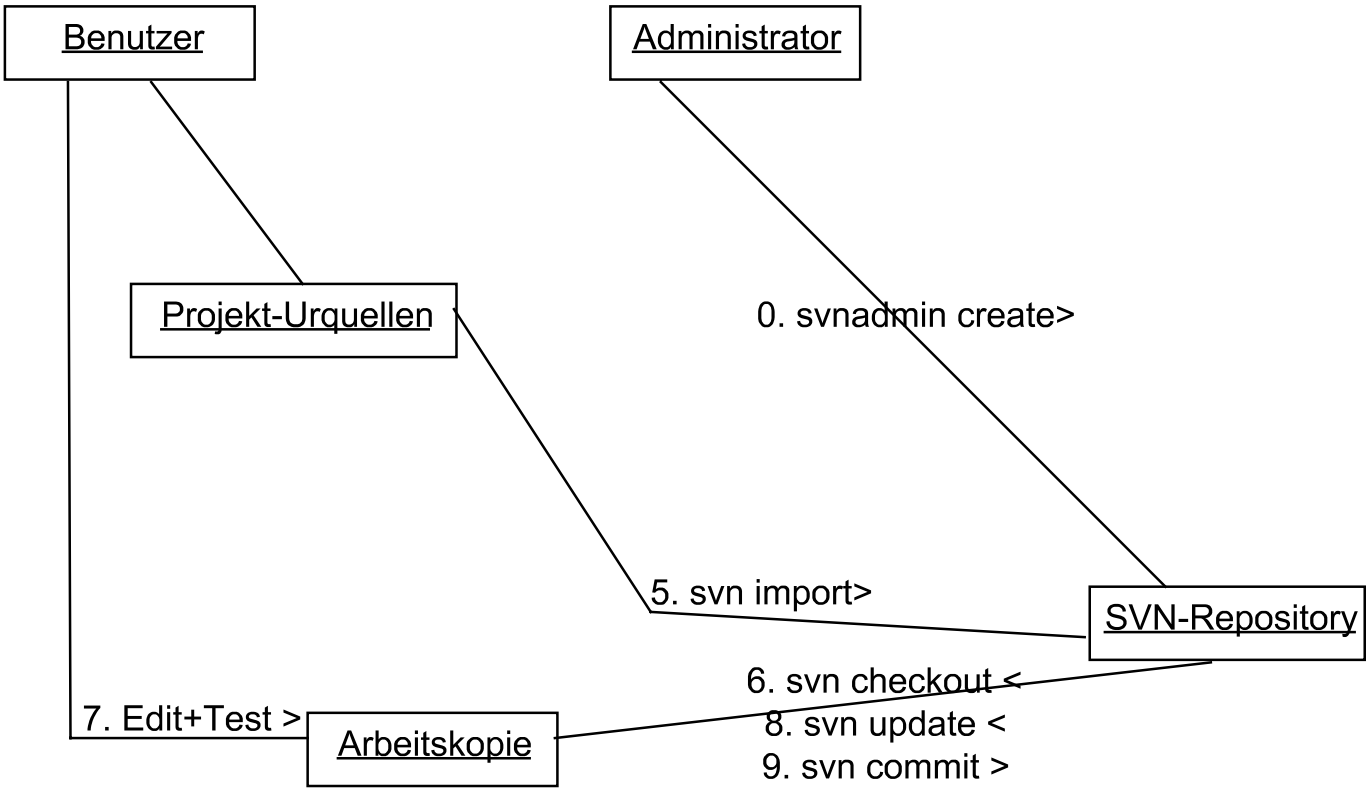
7.Bearbeitung der Arbeitskopie:
CD *projektName*`/trunk`
Editieren einiger Dateien. Test der Änderungen.

8.Aktualisierung der Arbeitskopie aus dem Repository, dabei Aufdeckung von eventuellen Konflikten:
Zunächst muß die Internetverbindung wieder hergestellt werden, sodann:
`svn update` [*Dateiname* ...]
Geänderte Dateien werden mit Flag M protokolliert.
Eventuelle Konflikte müssen durch Änderung an den soeben geladenen Dateien beseitigt werden, um dann durch ein erneutes `svn update` überprüft zu werden. Wenn Konflikte sehr unwahrscheinlich sind, kann Schritt 8 übersprungen werden.

9.Freigabe der lokalen Änderungen:
svn commit -m "*log message*" , **Bsp**.:
```
svn commit -m "Operationen durch Ausnahmen abgesichert"
```
Alle lokalen konfliktfreien Änderungen werden im Repository eingetragen, die geänderten Dateien mit alter und neuer Revisionsnr. protokolliert, sowie eventuelle Konflikte aufgedeckt. Die Internetverbindung kann beendet werden.

Bei GUI-Klienten werden alle diese Aktionen entsprechend über eine Explorer-Ansicht und Dialoge angesteuert. Siehe Links unter `http://subversion.tigris.org/project_links.html`. Auf Windows ist Tortoise `http://tortoisesvn.tigris.org/` eine bequem benutzbare Explorer-Erweiterung. Immer mehr Entwicklungsumgebungen enthalten auch von vornherein eine Möglichkeit, Subversion-Repositories wie eine lokale Verzeichnishierarchie einzubinden. Bei Eclipse wird dies durch das Plugin Subclipse http://subclipse.tigris.org/ realisiert und funktioniert sehr gut. Dennoch ist es sinnvoll, das zugrundeliegende Command-Line-Konzept zu kennen. Insbesondere bei SSL-Zugang kann es sinnvoll sein, eine Arbeitskopie zunächst mit dem Command-Line-Klienten auszuchecken, und dann erst in ein IDE-Projekt umzuwandeln.

Bild: Beispielszenario CVS-Benutzung als Datenflußdiagramm (zeichnerisch ein UML-Kollaborationsdiagramm dafür benutzt. Die Pfeile sind als Richtung der Datenübergabe zu lesen!)



1.3.2 Behandlung verteilter Änderungen bei update/commit

Bei einem `svn update` oder einem `svn commit` kann/wird es passieren, daß man selbst Änderungen vorgenommen hat und daß ein anderer Änderungen vorgenommen hat. Wie kommt Subversion damit zurecht? Subversion unterscheidet dabei verschiedene Fälle:

- Änderungen erfolgten in verschiedenen Dateien oder in derselben Datei in verschiedenen Zeilenbereichen
  ⇒
  Änderungen übernehmen (`update`: zum Client, `commit`: zum Server)
- Änderungen erfolgten in derselben Datei in demselben Zeilenbereich (Konflikt!)
  ⇒
  `commit` meldet Fehler,
  `update` übernimmt Änderungen + Konfliktmarkierung zum Client, ***Bsp***.:

```
...
  void grüßen(){
    System.out.println("Hallo, alle!");
<<<<<<< .mine
    System.out.println("Guten Abend");        }klientseitig
=======
    System.out.println("Guten Morgen");       }serverseitig, zuvor durch anderen
>>>>>>> .r2                                   }Klienten commitet.
    System.out.println("Tschüß!");
  }
...
```

  Dieser Konflikt muß vom Klienten bereinigt werden!

  Lösung: [?[8]]

  a)

  b)

c)

1.3.3 Weitere Subversion-Details

Subversion-Kommandos:

`svn status` [*dateiname*]
Zeigt Revisionsnr. + datum im Repository an. Unnötig bei GUI-Klient, da dort immer angezeigt.

`svn log` [*dateiname*]
Zeigt Änderungshistorie an, auch wenn nicht in Datei enthalten.

`svn diff` [*dateiname*]
Zeigt Unterschiede zwischen Arbeitsverzeichnis und Repository.

Subversion-Tags:

Sonderbedeutung im Quelltext, ***Bsp***.:

`//  $Log:` *Dateiname* `$`

wird durch Änderungshistorie substituiert. An den Anfang jeder Quelldatei stellen!

1.3.4 Quellen zu Subversion

Leitseite von Subversion: http://subversion.tigris.org/

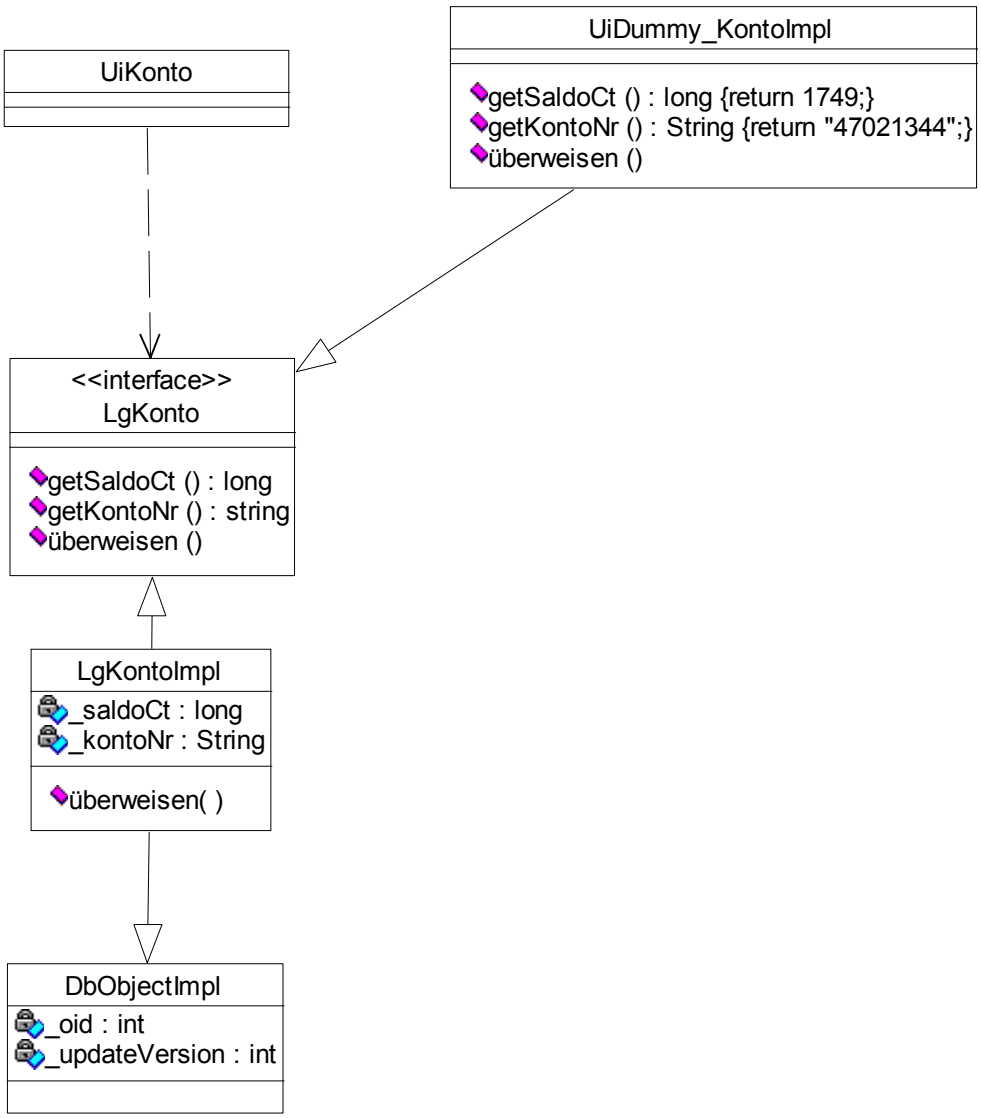
1.4 Implementierungsreihenfolgen

Verschiedene Strategien zur Reihenfolge der Implementierung möglich:

a) „bottom up“ (von unten nach oben):
Datenbankzugriffe + Dienste, dann Logik, dann Oberfläche.
Bewertung: [?[9]]

b) „top down“ (von oben nach unten):
Oberfläche, dann Logik, dann Datenhaltung + Dienste
Bewertung: [?[10]]

Bild: Entkoppelungstechniken zwischen Schichten für top-down-Entwicklung:



c) „hardest first“ (Das Schwierigste zuerst):
Bsp.: Erst CORBA, dann Rest
Bewertung: [?[11]]

d) „Durchstich zuerst“, dann weitere vertikale Komponenten:
Bsp.: Erst Objektverwaltung „Benutzer“, dann Objektverwaltung „Songlist“, dann Verbindungsverwaltung.
Bewertung: [?[12]]

1.5 Object-Relational Mapping

(§8.3 Balzert): Bei Benutzung einer objektorientierten Programmiersprache und einer relationalen Datenbank (üblich) muß die Lücke zwischen den beiden verschiedenen Konzepten überbrückt werden: OR-Mapping.

a) Objektidentität

OO: Objekt hat Attributwerte + Identität,
Bsp.: `new Kunde("Knabe") != new Kunde("Knabe")`

RDB: Inhaltsgleiche Zeilen gelten als identisch

⇒
Objektidentität muß im RDB durch künstlichen Primärschlüssel OID garantiert werden.
Bestgeeignet: Systemweit eindeutige Folgenummer, evtl. als langer String.

b) Klasse, Objekt, Attribut

Diese objektorientierten Konzepte werden wie folgt abgebildet:

OO		**RDB**
Klasse	→	Tabelle
Objekt	→	Zeile
Attribut	→	Spalte

Bsp.: Klasse

Artikel
Nummer Bezeichnung PreisCt

→ Tabelle „Artikel“

OID	Nummer	Bezeichnung	PreisCt
1	34986	Felge 28"	37,90
2	96431	Schlauch 28"	17,95
3	...		

c) Zusammengesetztes Attribut

Bsp.:

Student
MatrNr: String geburt: Datum name: String heimat: AdresseT studien: AdresseT

mit

AdresseT
plz: String ort: String strasse: String

Wie dieses abspeichern? Es gibt ja keine zusammengesetzten Spalten in einer RDB.

[?[13]]

Alternativ wie 1:1-Assoziation behandeln (siehe dort)

d) 1:*-Assoziation

Bsp.: „Ein Kunde kann viele Konten besitzen“

Wie dieses abspeichern?

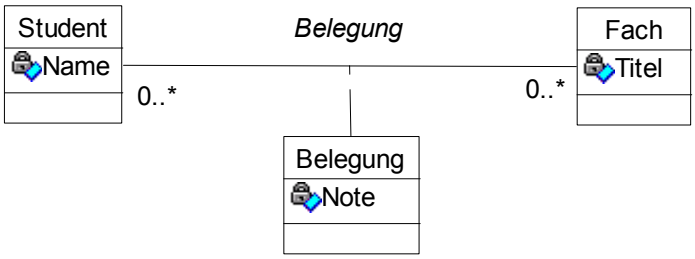
[?[14]]

e) 1:1-Assoziation

Wie 1:*-Assoziation oder zusammengesetztes Attribut.

f) *:*-Assoziation

Bsp.: „Ein Student erhält für die Belegung eines Faches eine Note“



Wie dieses abspeichern? [?[15]]

g) n-stellige Assoziation

Was ist das? [?[16]]

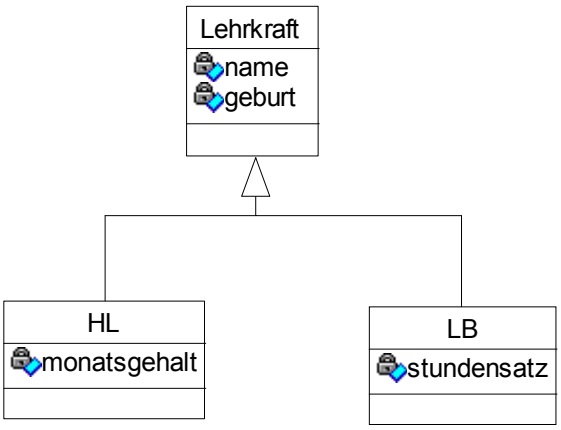
Wie dieses abspeichern? [?[17]]

h) Weitere Nebenformen → [Balzert99].

1.6 OR-Mapping der Einfachbeerbung

→ [Balzert99] p. 318 ff

Bsp.: Eine Lehrkraft an der TFH ist entweder ein HL (Hochschullehrer, hauptamtlich) oder ein LB (Lehrbeauftragter, nebenberuflich, auf Honorarbasis):



Um solch eine Vererbungshierarchie abzubilden, gibt es drei Varianten, jede mit ihren Vor- und Nachteilen:

a) Ganze Vererbungshierachie auf eine Tabelle + Klassendiskriminator

Tabelle „Lehrkraft“

OID	_Klasse	name	geburt	monatsgehalt	stundensatz
17	HL	Solymosi	27.03.1953	4.300,00	—
18	LB	Wiesner	21.11.1962	—	32,90

Bewertung: [?[18]]

Für unser Projekt gut bei Selbstprogrammierung des OR-Mapping!

b) Je konkrete Klasse eine Tabelle

Tabelle „HL“

OID	name	geburt	monatsgehalt
19	Scheschonk	11.07.1949	3.995,00

Tabelle „LB“

OID	name	geburt	stundensatz
20	Iversen	31.01.1967	32,90

Bewertung: [?[19]]

c) Je Klasse (auch abstrakte) eine Tabelle, gemeinsame OID

Tabelle „Lehrkraft“

OID	name	geburt
21	Godbersen	27.08.1950
22	Faustmann	11.04.1959

Tabelle „HL“

OID	monatsgehalt
21	4.300,00

Tabelle „LB“

OID	stundensatz
22	36,70

Bewertung: [?[20]]

1.7 Transaktion in der Datenbankschicht, Logikschicht oder Oberflächenschicht?

([Siedersleben], Kap. 9.3, Balzert p.344, p. 410)

Bsp.: Die Operation

```
quellKonto.überweisen(io_zielKonto, i_betragCt);
```

besteht geschäftsregelmäßig aus den Schritten

```
quellKonto.saldoCt   -= i_betragCt;
io_zielKonto.saldoCt += i_betragCt;
_pool.save(quellKonto);
_pool.save(zielKonto);
```

Ziel: Durchführung **ganz** oder **gar nicht**!

Transaktionen gehorchen dem **ACID-Prinzip**: Atomic, Consistent, Isolated, Durable (Datenbank-Stoff).

Zur Unterstützung von Transaktionen beim Datenbankzugriff gibt es in **JDBC** in der Klasse `java.sql.Connection` folgende Operationen:

- `commit()`: Abspeichern der bisherigen Änderungen sichtbar für Andere
- `rollback()`: Verwerfen der Änderungen seit dem letzten `commit()`, im Fehlerfall aufzurufen.

Wo Transaktion definieren?

Gemäß [Siedersleben] spezifizieren wir die Logikschicht in einer heilen Welt, d.h. als ob es nur einen Systembenutzer gäbe. Die Probleme des wechselseitigen Ausschlusses paralleler Benutzer und des Ganz-oder-Garnicht-Durchführens von Veränderungen können mit dem Transaktionskonzept gelöst werden. Es ist jedoch Sache der Oberflächenschicht oder des Applikationsservers zu entscheiden, welche Lg-Operationen zu einer Transaktion zusammengefasst werden. Bei Stapelverarbeitung wird man mehr Lg-Operationsaufrufe als bei Dialoganwendungen zu einer Transaktion zusammenfassen.

Bsp. Heile Welt in der Klasse `LgKonto`:

```
public void überweisen(
    final LgKonto io_zielKonto, final int i_betragCt
){
    //Veränderungen im Arbeitsspeicher vornehmen (eigentliche Geschäftslogik):
    this        .saldoCt -= i_betragCt;
    io_zielKonto.saldoCt += i_betragCt;

    //Veränderungen gegen Fremdänderungen prüfen; in DB isoliert (nur für sich selbst) sichern:
    _pool.save(this);
    _pool.save(io_zielKonto);
}//überweisen
```

Transaktionsabsicherung in der Oberflächenschicht:

Jede externe Operationen muss als Transaktion formuliert sein; dies macht man, indem man bei Erfolg am Ende `commit()`, andernfalls `rollback()` aufruft.
Es ist m.E. sinnvoll, diese Operationsaufrufe in die selbst geschriebenen Operationen `lgApplication.beginTransaction()` und `lgApplication.endTransaction()` zu kapseln, um auch noch andere transaktionsbegleitende Aktionen automatisch mit auszuführen.

Bsp. in der Klasse `UiKontoÜberweisung`:

```
private void actionPerformed(
    final java.awt.event.ActionEvent i_event
){
    final Object source = i_event.getSource();
    if(source==_okButton){
        final int betrag
        = LgBetrag.fromString(_betragField.getText());
        _lgApplication.beginTransaction();
        boolean ok = false;
        try {
            _quellKonto.überweisen(_zielKonto, betrag);
            ok = true; //Erfolg vermerken
        }finally{_lgApplication.endTransaction(ok);} //Veränderungen
                                                     //freigeben für andere oder verwerfen je nach ok
    }else if(source==_abbruchButton){
        _closeDialog();
    }else if(source==…){
        …
    }else{
        throw new multex.Failure(
            "Illegal event source {0}", null, source
        );
    }
}//actionPerformed
```

Bem.: Der **`finally`**-Zweig wird bei jeder Beendigung des **`try`**-Blocks durchgeführt, sei es erfolgreich oder fehlerhaft. Als Transaktionsabschluss muß `lgApplication.endTransaction` je nach dem `ok` –Wert `commit()` oder `rollback()` aufrufen.

Achtung: Es reicht nicht aus, anstatt von `ok = true;` das `commit();` aufzurufen. Es müßte dann auch vor jedem (erfolgeichen) **`return`** …`;` aufgerufen werden! Außerdem müßte jede Ausnahme abgefangen werden, um ein `rollback();` abzusetzen, sie dann aber wieder weiterzureichen. Es müßte also, wenn ansonsten keine weitere Ausnahmebehandlung benötigt wird, folgender Zweig eingefügt werden für das Rollback:

```
    }catch(Throwable ex){rollback(); throw ex;}
```

Bsp: Siehe als größeres Beispiel dazu auf meiner Homepage eine kleine Bankanwendung in 3 Schichten [Bank3Tier].

Achtung: Zwecks Redundanzvermeidung sollte sowohl die Transaktionssteuerung als auch die Meldung von Ausnahmen aus den einzelnen `actionPerformed`-Methoden herausgezogen und nur einmal zentral implementiert werden, z.B. in einer zentralen Klasse `UiAction` als Unterklasse von `javax.swing.Action` oder bei einer Struts-Webanwendung von `org.apache.struts.action.Action`!

1.8 DB-Schicht als API

folgt [Siedersleben], Scott [Ambler][1], stark vereinfacht, vgl. §10.6 Balzert.

Die Lg-Objektverwaltung erfolgt durch einfachste interne Operationen der Datenbankschicht in der Klasse `Pool`: `save`, `delete`, `find`, `findAll`, `executeQuery`:

- ```
  boolean save(DbObject io_object) throws ConcurrentUpdateExc …;
  //Saves the actual state of the object into the pool.
  //If the object was not persistent before, it gets a new OID. Its Pool is set to this Pool.
  //@return true newly inserted into the pool.
  //@throws ConcurrentUpdateExc  Another user modified this object in the Pool before you.
  //              Thus you should repeat your transaction.
  ```

- ```
  boolean delete(DbObject io_object);
  //Deletes the object from the pool.
  //Additonally marks the object as not valid by giving it the OID deletedOid.
  //@return true It was in the pool.
  ```

- ```
  DbObject find(Class i_persistentClass, Integer i_oid)
  // Returns the element of the Pool with the given Class and OID.
  throws NoPersistentClassExc, OidNotFoundExc;
  ```

- ```
  /*DbObject*/List findAll(Class i_persistentClass)
  // Returns a List with all elements of the given Class in the Pool.
  throws NoPersistentClassExc;
  ```

- ```
  /*DbObject*/List findManyByQuery(
      Class i_resultClass, Class i_queryClass, Object[] i_args
  )
  //Returns all objects of class i_resultClass from the Pool, which fulfill the query i_queryClass with the
  //given arguments.
  throws NoPersistentClassExc, NoQueryClassExc, ArgumentCountExc,
  ArgumentTypeExc;
  ```

Diese Operationen werden

- in der Db-Schicht implementiert
- von den externen Operationen der Lg-Schicht aufgerufen.

Siehe obiges Beispiel [Bank3Tier] mittels [JORA] (Java Object Relational Adapter).

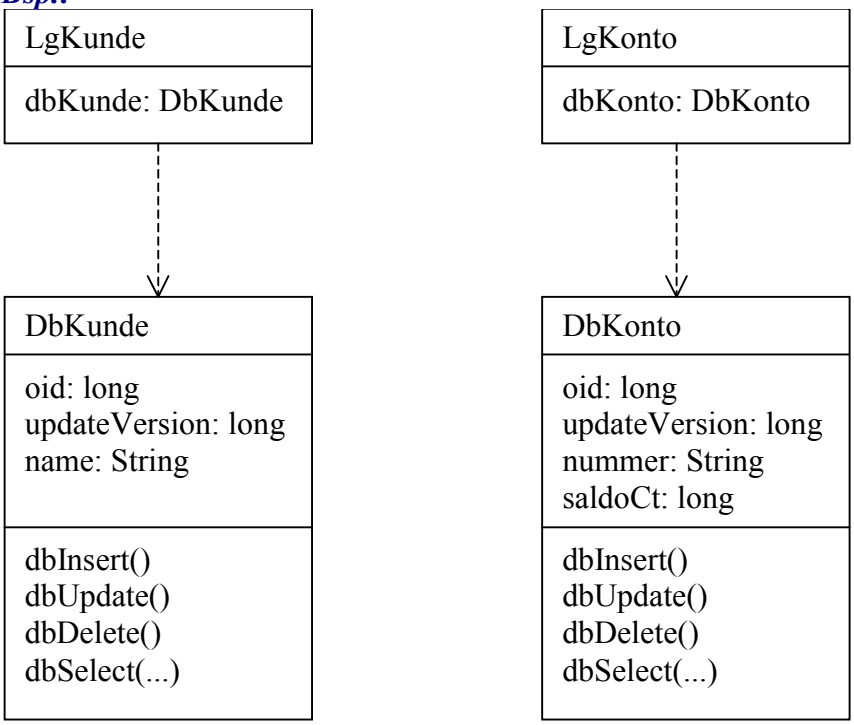
---

[1] Seine Operation retrieve() zum Laden des vollen Objektes bei einem Proxy-Objekt dient nur der Effizienz und lassen wir daher weg.
  ```

1.8.1 Ungenerischer Ansatz

Je Tabelle eine eigene Db-Klasse.

Bsp.:



Diese Operationen werden nach einem **einheitlichen Schema** implementiert; nur die Attributnamen variieren dabei.

Bsp.: Implementierung der Operation dbInsert():

```
oid = _getNewOid();     updateVersion++;
final String sql =                                          [?21]
```

```
;
final Statement stmt = _connection.createStatement();
stmt.executeUpdate(sql);
stmt.close();
```

Bewertung: [?22]

1.8.2 Generischer Ansatz

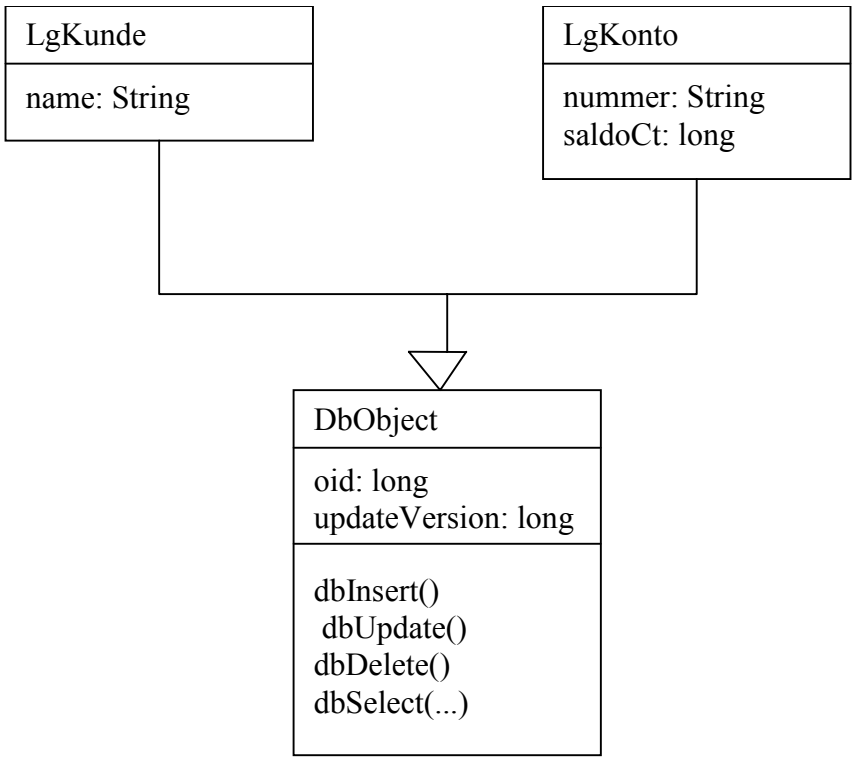
benutzt Java-Reflection: Paket `java.lang.reflect`.

Bsp.: [JORA] (Java Object Relational Adapter, klein, von mir für MulTEx instrumentiert), Hibernate (groß) oder selbst programmieren.
Überblick siehe unter [Database].

Vorgehen:

- Eine generisch arbeitende Db-Klasse für alle persistenten Lg-Klassen → Db-Tabellen
- Attribute verbleiben in den Lg-Klassen
- Db-Klasse fragt ein Lg-Objekt nach seinen Attribut(wert)en, generiert daraus eine SQL-Anweisung

Bsp.:



Diese Operationen werden in `DbObject` **generisch** implementiert; die Attributnamen und -Werte werden dabei mittels Reflection aus dem Objekt bzw. seiner Klasse ermittelt.

Bsp.: Implementierung der Operation dbInsert():

```
oid = _getNewOid();     updateVersion++;
//Prinzip:
final Class myClass = getClass();
final String className = myClass.getName();
final String tableName
= className.substring(className.lastIndexOf('.')+1);
final java.lang.reflect.Field[] fields
= myClass.getDeclaredFields();
final StringBuffer sql = new StringBuffer(
  "INSERT INTO " + tableName + " VALUES ("
);
```

[?[23]]

```
sql.append(")");
... //sql ausführen
```

Bewertung: [?[24]]

1.9 Komponentenarchitektur

Dieses Kapitel folgt [Siedersleben], Kapitel 3 und 4.

Die **Komponentenarchitektur** hat sich als ein Muster der Benutzung objektorientierter Techniken in den letzten Jahren wegen ihrer großen Flexibilität durchgesetzt.

1.9.1 Komposition

Def.: Komponente? [?[25]]

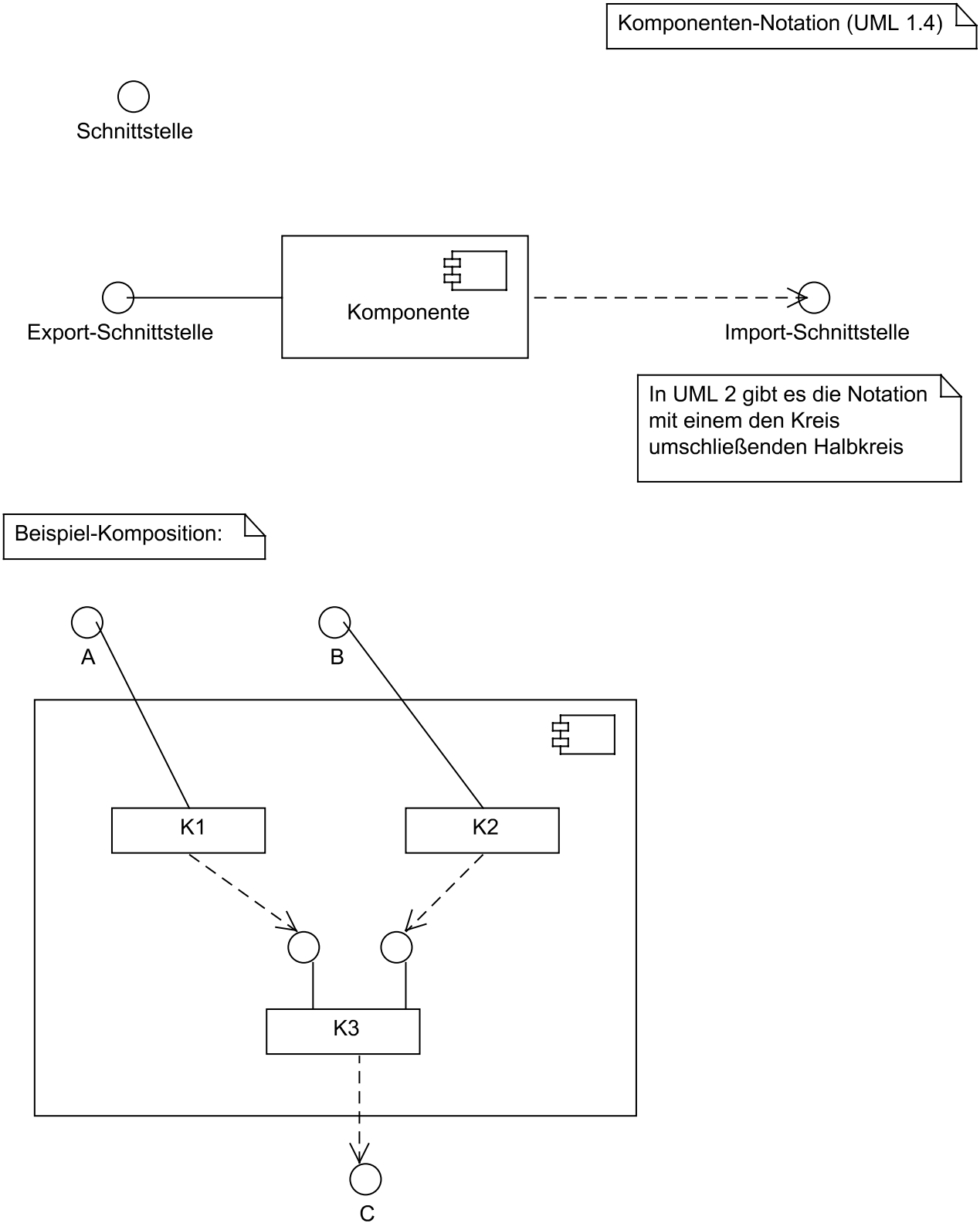
Technisch:

- **Exportiert Schnittstellen** gemäß *Vertragsmodell*, implementiert diese versteckt.
- **Importiert** (benutzt) andere *Schnittstellen***.**
- Kann zusammengestellt (**komponiert**) werden zu anderen Komponenten; über viele Stufen.

Daraus ergeben sich folgende „politische“ Eigenschaften:

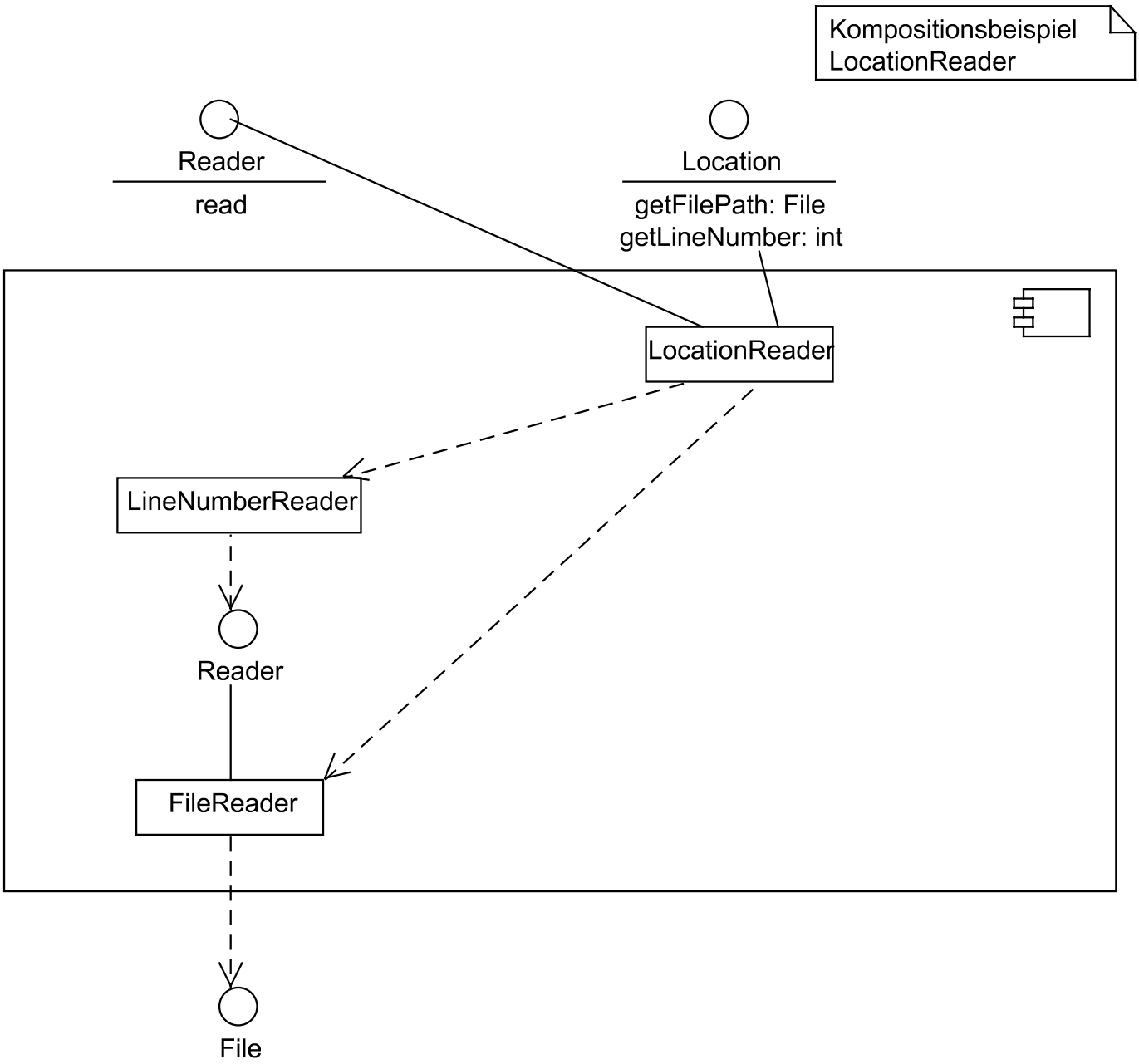
- **Ersetzbar** durch andere Komponente, die die gleichen Export-Schnittstellen implementiert.
- **Wiederverwendbar** wegen minimaler Annahmen über ihre Umgebung (nur Import-Schnittstellen).
- **Zentrale Einheit** des Entwurfs.

Notation (UML)



Bsp einer Komposition in Java

Ein `LocationReader` sei ein `Reader`, der seine aktuelle `Location` (Dateipfad und Zeilennummer) kennt:



wobei

```
import java.io.File;
import java.io.Reader;
```

und

```
interface Location {
    File getFilePath();
    int  getLineNumber();
}
```

Siehe Beispiel-Code in http://www.tfh-berlin.de/~knabe/fach/md7/LocationReader/

Def.: Ein **Kompositionsmanager** (Software) erstellt eine Komponente, indem er
- Teilkomponenten auswählt,
- sie konfiguriert,

- untereinander komponiert („dependency injection“) und
- Export-Schnittstellen befriedigt.

Im obigen Beispiel macht dies die Klasse `LocationReader`.

Richtlinie: Für freie Komponierbarkeit dürfen keine **`static`**-Variablen verwendet werden! Siehe Kap.3.1.7 static.

1.9.2 Schnittstellen und Adapter

Bemerkung:

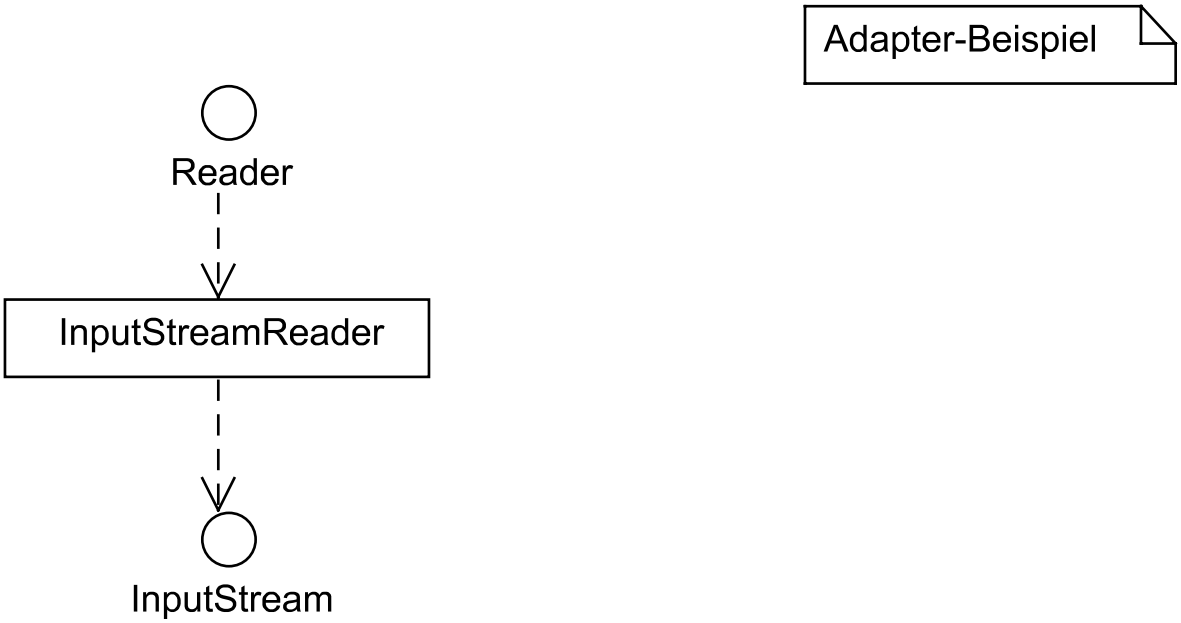
Eine *Schnittstelle* ist eine Menge von Operationsspezifikationen. Das ist allgemeiner als ein Java-**`interface`**. Im vorigen Beispiel wurden auch die Klassen `Reader` und `File` als Schnittstellen verwendet.

Arten: Export-Schnittstelle, Import-Schnittstelle, Standard-Schnittstelle.

Def.: Eine **Standardschnittstelle** ist nicht für eine bestimmte Komponente geplant, sondern für allgemeine Verwendung (ideal). Sie ist daher in allen Projekten immer im Klassenpfad.
Bsp.: `java.util.List`, `multex.Failure`

Def.: Ein **Adapter** ist eine Komponente, die eine Importschnittstelle mit einer nichtkompatiblen Exportschnittstelle verbindet.
Bsp.: `java.util.List`, `multex.Failure`

Bsp.: Oft erhält man einen `InputStream` zur Verfügung gestellt (z.B. `System.in`). Wenn man aber Zeichen statt Bytes verarbeiten will, muss man ihn mit einem `InputStreamReader` als Adapter komponieren:

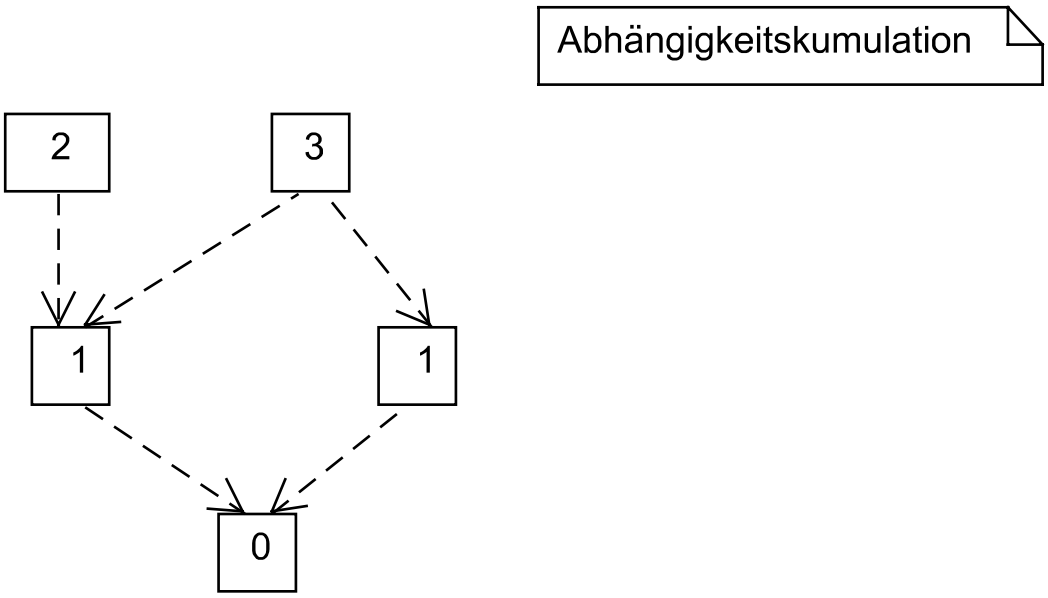


1.9.3 Software-Kategorien („Blutgruppen“)

Ziele:

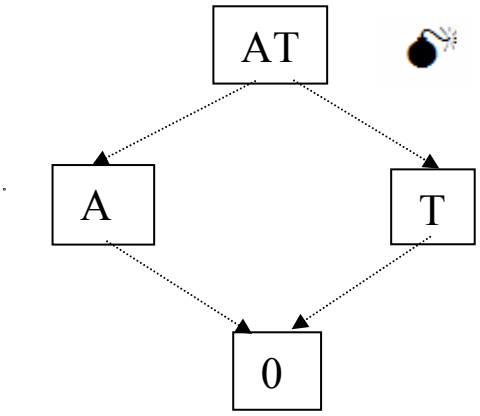
- Abhängigkeiten minimieren
- Innovation ermöglichen.

Kumulation der Komplexität bei Abhängigkeiten: (Zahl = Abhängigkeitsmaß)



Software-Kategorien:

Kat.	Beschreibung
0	Universell, steht immer zur Verfügung, ändert sich nicht
A	Anwendungsbezogen (**Bsp**. Stundenplan, FiBu), unterliegt fachlicher Innovation (**Bsp**. Steuerrecht 2005 → 2006)
T	Technikbezogen (**Bsp**. JDBC, JDO, SAX, DOM, XMLDigester), unterliegt technischer Innovation (**Bsp**. JDBC 2.0 → 3.0, Umstellung JDBC → JDO)
R	Repräsentationsbezogen, transformiert zwischen 2 Welten (Bsp. JDBC:`java.sql.ResultSet` → A-Objekt), meist schematisch ⇒ generisch lösbar.
AT	Vermischt Anwendung und Technik.  Vermeiden!



Problem: Wie Anwendung (**Bsp**. Kontaktverwaltung) unter Einsatz von Technik (**Bsp**. JDBC) schreiben, ohne Anwendung mit Technik zu mischen?

Lösung: Kommunikation über universelle Schnittstellen (Kategorie 0), notfalls Transformation mittels R-Software.

Beispiel:

```
/**Universelle Persistenzschnittstelle der Kategorie 0, kennt weder JDBC noch Kontakte*/
interface Pool {
    boolean  save(DbObject io_object);
    DbObject find(Class i_persistentClass, Integer i_oid);
    …
}
```

`LgKontakt` beerbt `DbObject`, erhält dadurch bei `save` eine OID, ist dadurch mittels `find` wieder auffindbar. **Bsp.**:

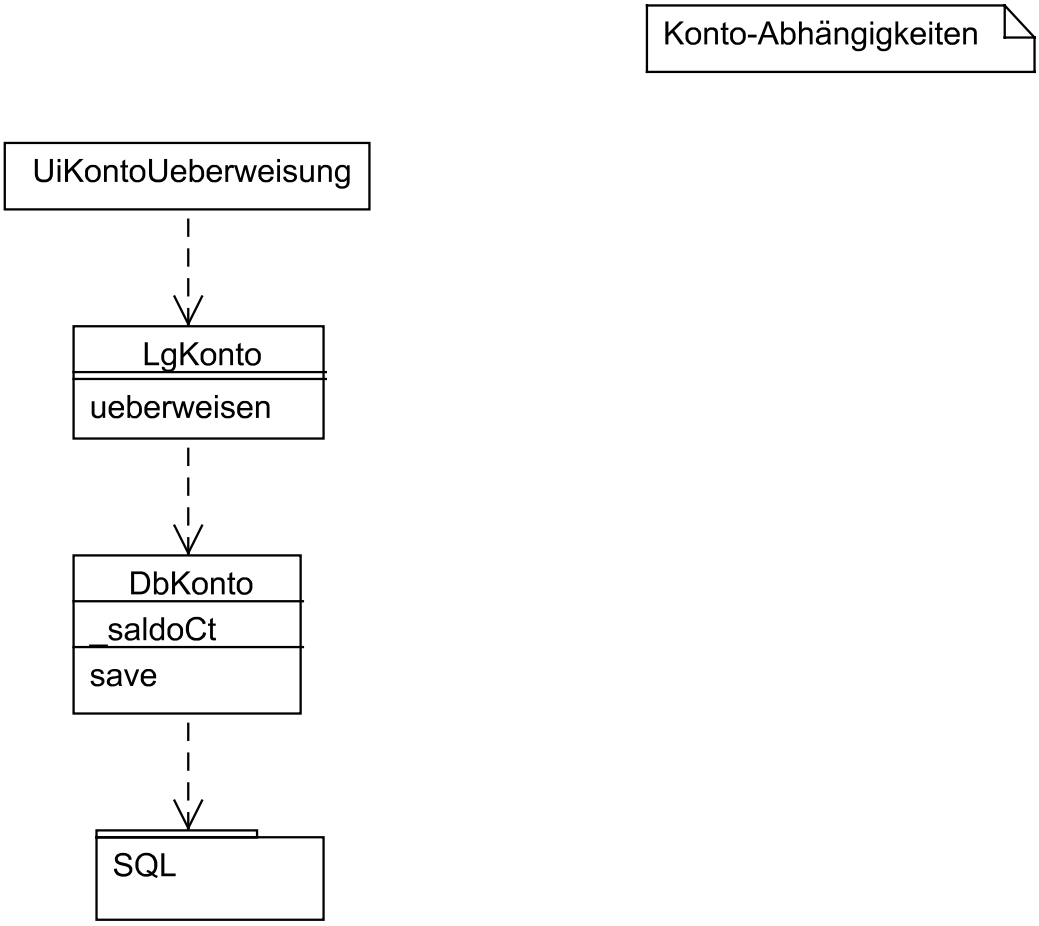
```
final LgKontakt kontakt = new LgKontakt(…);
_pool.save(kontakt);
final Integer oid = kontakt.getOid();
…
final LgKontakt kontaktKopie
= (LgKontakt)_pool.find(LgKontakt.class, oid);
```

Angestrebt:

- Alle Techniken hinter minimalen, universellen (Kategorie 0) Schnittstellen verstecken.
- Diese untergliedern in
  - **operative Schnittstelle** für häufige Verwendung (**Bsp**. `Pool.find`),
  - **administrative Schnittstelle** für seltene Verwendung (**Bsp**. `QueryDefiner` für `Pool`).

1.9.4 Testen von Komponenten („Mock Objects“)

Bsp. „Konto-Abhängigkeiten“:



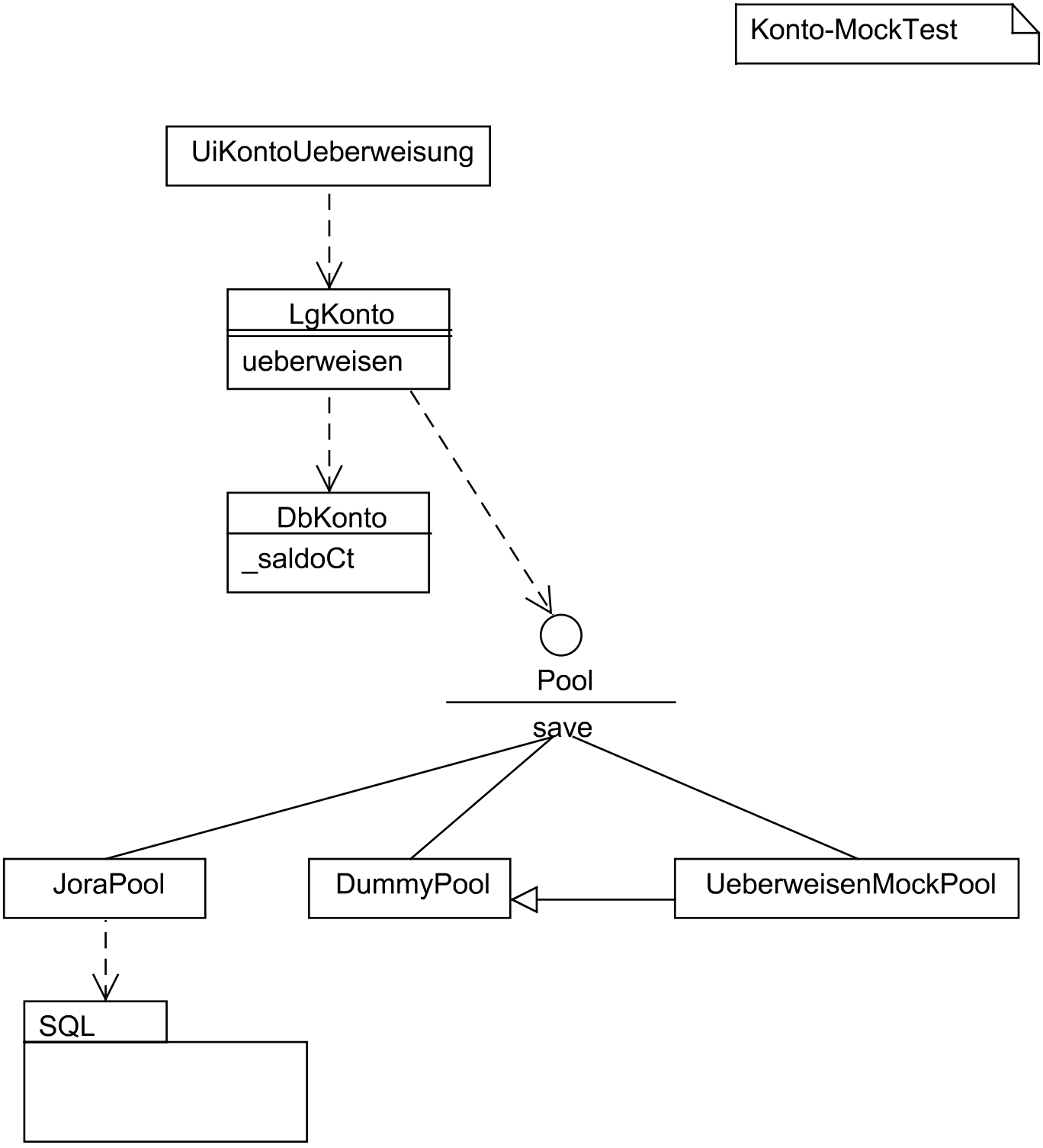
Herkömmlicher Test: *Black Box* ⇒ *Bottom Up*

Probleme:

- Erst wenn `DbKonto` funktioniert, kann `LgKonto` entwickelt und getestet werden.
- Persistenz fest „verdrahtet“ ⇒
  - benötigt reale Datenbankanbindung, oft zu langsam,
  - Komplexer Anfangszustand schwer (wieder-)herstellbar,
  - Fehlerbehandlung nicht testbar, da DB-Fehler nicht befohlen werden können.

Lösung:

- Formulieren des Testlings `LgKonto` als Komponente
- **Dependency Injection**:
  Befriedigung der Import-Schnittstelle `Pool` real oder durch Dummy- oder testfallspezifische **Mock**-Objekte:



Def.: Ein **Dummy-Objekt** implementiert alle Methoden einer Schnittstelle minimal, z.B. durch Rückgabe konstanter Werte.

Bsp.:

```
class DummyPool implements Pool {
    boolean save(DbObject io_object){return false;}
    boolean areLinked(
        DbObject i_left,
        Class    i_linkClass,
        DbObject i_right
    ){return false;}
    …
}
```

Dies reicht für viele Tests schon aus. **Bsp.**:

```
void testUeberweisen_OhneRechteExc() throws Exception {
    final Pool pool = new DummyPool();
    final LgKonto
        knabeKonto = new LgKonto(pool),
        muellerKonto = new LgKonto(pool),
    ;
    final LgKunde knabe = new LgKunde();
    try{
        knabeKonto.ueberweisen(muellerKonto, 12345, knabe);
        fail("LgKonto.OhneRechteExc expected");
    } catch ( LgKonto.OhneRechteExc expected ){
    }                    ↑
                         //Kommt, da
}           //pool.areLinked(knabe, LgVerwaltet.class, knabeKonto)
            //false liefert.
```

Def.: Ein **Mock-Objekt**
- ist eine Implementierung einer *Import-Schnittstelle*,
- die aktiv zum Test beiträgt.

Bsp.: Klasse bank3tier.lg.TestLgKontoMock.UeberweisenMockPool
→ http://www.tfh-berlin.de/~knabe/java/bank3tier/

Literatur: [Link]

1.10 Multi-Tier Exception Handling

Eine Strategie und ein Java Framework zur diagnosestarken Ausnahmebehandlung in mehrschichtigen Softwaresystemen.

Siehe [MulTEx]. Die wichtigsten Prinzipien beim Programmieren mit MulTEx sind:

- **Vertragstreue**: Eine erfolgreiche Methodenausführung soll das leisten, was ihr Name und ihre verbale Spezifikation verspricht, und zwar ganz (ACID-Prinzip).
- **Fehlbenutzung**: Vorbedingungsverletzungen werden durch geprüfte, einzeln behandelbare, Ausnahmen (endend auf `Exc`) beantwortet.
- **Versagen**: (= Nachbedingungsverletzungen) wird durch eine ungeprüfte Sammelausnahme, endend auf `Failure`, die die Versagensursache verpackt, beantwortet.
- **Diagnosestärke**: Alle ausgeworfenen Ausnahmen sollten mit den örtlich zur Verfügung stehenden Diagnoseinfos parametriert und mit menschenfreundlichem Fließtext versehen sein. Ausnahmen

immer mit den MulTEx-Diensten wie multex.Msg.report(…) melden, damit die komplette Ursachenkette angezeigt wird und bei Nachfrage auch der StackTrace.

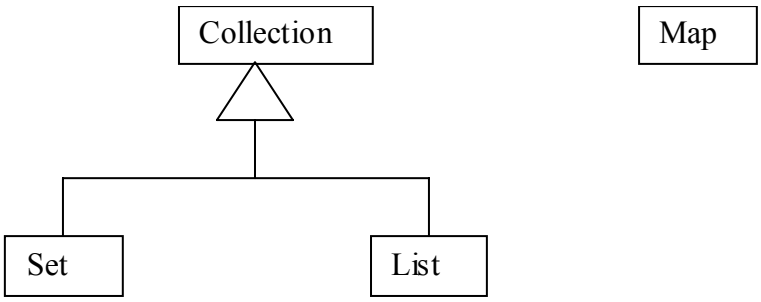
- **Zentralisierung**: Es soll idealerweise nur eine Stelle im Softwaresystem geben, wo alle Ausnahmen gemeldet werden. Dies ist typischerweise in einem generischen ActionListener o.ä. in der Oberflächenschicht, sowie in einer einheitlichen `ThreadGroup`.

1.11 Ergebnismengenübergabe

Von Db- zu Lg-Schicht und
Von Lg- zu Ui-Schicht müssen Objektmengen übergeben werden.

Dafür sinnvoll (ab JDK 1.2) das „Java Collections Framework“ in Paket `java.util`.

Interfaces:



`Collection`	Sammlung **ohne** Reihenfolge (traditionell *Bag*)
List	Sammlung **mit** Reihenfolge
`Set`	Sammlung ohne Duplikate: `x.equals(y)` verboten
`Map`	Sammlung von (Schlüssel,Wert)-Paaren

Implementierungen

- Im JDK je Interface 2, Bsp. `ArrayList`, `LinkedList`
- Weitere einkaufbar (Bsp. in POET) oder selbst definierbar.
- Auch als Verpackung eines DB-Cursors möglich.

1.11.1 Polymorphe Collection

Eine polymorphe Collection kann Elemente beliebiger Untertypen von `Object` speichern, auch gemischt!

Benutzungs-Bsp.:

```
interface Collection {  //in java.util
  void add(Object e);  //fügt ein Element hinzu
  Iterator iterator();
  ...
}
...
//Benutzung:

//Sammlung erzeugen:
final Collection coll = new ArrayList();
//Sammlung füllen:
coll.add(new Byte(1));    coll.add(new Byte(7));
...
//Sammlung auswerten:
for(final Iterator i = coll.iterator(); i.hasNext();){
  final Byte b = (Byte)  //Typspezialisierung Object → Byte nötig! Laufzeitfehlerrisiko!
  i.next();
  System.out.println(b);
}
```

Bewertung: [?[26]]

1.11.2 Generische Collection

Bei der polymorphen Collection haben sich gewisse Probleme gezeigt:

- ***Problem***: Elementetyp der Collection ist Object (polymorph)
  ⇒ alles einfügbar,
  bei Entnahme Typspezialisierung erforderlich mit Risiko von Laufzeitausnahmen.
- ***Lösung***: Generisch typsichere Collection-Klassen (wie Templates in C++),
  d.h. Elementetyp als Klassenparameter in spitzen Klammern < ... >,
  ab JDK 1.5.

Benutzungs-Bsp.:

```
interface Collection<E> { //in java.util
  void add(E e); //fügt ein Element hinzu
  Iterator<E> iterator();
  ...
}
...
//Benutzung:

//Sammlung erzeugen:
final Collection<Byte> coll = new ArrayList<Byte>();
//Sammlung füllen:
coll.add(new Byte(1));   coll.add(new Byte(7));
        // ↑ Nur Byte einfügbar
...
//Sammlung auswerten:
for(final Iterator<Byte> i = coll.iterator(); i.hasNext();){
  final Byte b =  //Entnahme ohne Typspezialisierung, Vermeidung von Laufzeitfehlern.
  i.next();
  System.out.println(b);
}
```

Bewertung: [?[27]]

Halbautomatische Generizität

- Eine typsichere Muster-Klasse mit Elementetyp ELEMENT verpackt als Wrapper eine polymorphe Collection-Klasse.
- Alle Einfügeoperationen haben den Parametertyp ELEMENT ⇒ Es können nur ELEMENT-Objekte eingefügt werden.
- Alle Entnahmeoperationen spezialisieren den Ergebnistyp nach ELEMENT. Da nur ELEMENT-Objekte in der Collection sein können, entfällt dennoch das Risiko von Laufzeitfehlern.
- Aufrufe werden 1:1 delegiert an die verpackte Collection
- Instanzierung eines typsicheren Exemplars aus der Muster-Klasse geschieht mit einem Texteditor durch globale Substitution des Platzhalters ELEMENT. Dies ist auch in einem Skript automatisierbar.

Bsp.: ELEMENT_List im Paket lib bei /~knabe/java/lib/,
angewendet auch im größeren Beispiel [Bank3Tier] als LgKunde_List.

```
public class ELEMENT_List implements java.io.Serializable{

  private final java.util.List _wrapped;

  public boolean add(ELEMENT o){
    return _wrapped.add(o);
  }
  public ELEMENT get(int index){
    return (ELEMENT)_wrapped.get(index);
  }
  ...
}
```

Noch: ELEMENT_LIST zeigen, insbesondere typsicheres `sort`, typsicherer `Comparator`

Bewertung: [?[28]]

Die folgenden Kapitel erfolgen als Tafelanschrieb.

1.12 Objektorientierte Datenbanksysteme (Balzert §8.4)

1.13 Entwurf in UML (Balzert §6.1, §6.4)

2 ENTWURFSPRINZIPIEN, ENTWURFSMUSTER, TESTSTRATEGIEN

2.1 Entwurfsprinzipien

2.2 Entwurfsmuster

Siehe [Gamma95]

2.3 Das Muster „Kompositum“

2.4 Klassifikation, Begriffe

2.5 Das Muster „Interpreter“

2.6 Das Muster „Abstract Factory“

2.7 Das Muster „Observer“

3 ANHANG

3.1 Programmierrichtlinien

Diese Richtlinien sollen helfen, gut lesbare, gut wartbare und möglichst robuste Programme zu erstellen.

3.1.1 Zentrale Prinzipien

Alles so **konstant**, sinnvoll **initialisiert**, **lokal** und **redundanzfrei** wie möglich. Schachtelungsorientierte **Einrückung**! Erläuterungen dazu siehe Vortrag [Programmierstil].

Abbrechende Fehlerprüfung: Siehe Methode `LgKonto.überweisen` im Beispiel [Bank3Tier].

3.1.2 Redundanzfreie Verwendung von Literalen

Literale (Zahlen, **Bsp**. `5`, `3.1415` und Zeichenketten, Bsp. `"EUR"`) nie direkt im Anweisungsteil einer Operation benutzen, sondern immer nur zur einmaligen Deklaration einer ihrer Bedeutung entsprechend benannten Konstanten, deren Name dann in Anweisungen mehrfach benutzt werden darf.

Gründe:

a) Die meisten Literale haben eine bestimmte Bedeutung und sind nicht zufällig so gewählt. Für die Wartbarkeit ist es erforderlich, daß bei Änderung des Wertes diese nur an einer zentralen Stelle erfolgen muss. Die zu obigen Literalen gehörigen Konstantendeklarationen könnten lauten:
Bsp.:
```
final byte plzLaenge = 5;
final float pi = 3.1415;
final String waehrungsKz = "EUR";
```

b) Eine Zeichenkette, aufgrund deren Wert eine Programmverzweigung vorgenommen wird, ist in Literalform besonders gefährlich, denn der Compiler kann die Existenz der entsprechenden Verarbeitung nicht überprüfen. Damit wird ein potentieller Fehler, z.B. bei Vertippen oder nach Umbenennung auf die Laufzeit, d.h. auf den Kunden abgewälzt. Man kann solche Fallunterscheidungen oft auch auf dem `Class`-Objekt der verarbeitenden Klasse statt auf einem String basieren lassen.
Bsp.: Statt wie in vielen Lehrbüchern bei Einsatz von JDBC zum Laden des Datenbanktreibers

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

ist die folgende Form sicherer, da bei Nichtvorhandensein der Treiber-Klasse der Compiler einen Fehler melden würde:

```
Class.forName(sun.jdbc.odbc.JdbcOdbcDriver.class.getName());
```

Bsp.: Statt in einer Swing-GUI eine button-abhängige Verzweigung durch Vergleich des Command-Strings mit einem String-Literal durchzuführen wie in:

```
void actionPerformed(final ActionEvent i_ev){
  final String command = i_ev.getActionCommand();
  if(command.equals("Drucken"){_drucken();
  }else if(command.equals("Speichern"){_speichern();
```

```
  }
```

ist es sicherer (und effizienter), die Verursacher-Objekte des Events zu vergleichen:

```
void actionPerformed(final ActionEvent i_ev){
  final Object command = i_ev.getSource();
  if(command == _druckenButton){_drucken();
  }else if(command == _speichernButton){_speichern();
  }else{throw new IllegalArgumentException(command.toString());
  }
  …
}
```

3.1.3 Redundanzfreiheit zwischen externen und internen Größen

Die notwendige Konsistenz zwischen programmexternen Größen (z.B. Konfigurationsangaben, Nationaltexte) muss zur Übersetzungszeit gesichert werden. Zu den möglichen Mitteln zählt eine Konstantengenerierung.

Bsp.: Aus der Nationaltextdatei `texte.properties`:

loginUsername = Username
loginPassword = Password

kann mit wenig Aufwand folgende Klasse generiert werden:

```
class UiKeys {
    final String loginUsername = "loginUsername";
    final String loginPassword = "loginPassword";
}
```

Die Verwendung der Konstanten aus `UiKeys` garantiert dann die Existenz des entsprechenden Nationaltexts zur Übersetzungszeit:

```
usernameField.setLabel( getResource(UiKeys.loginUsername) );
```

3.1.4 Namenskonventionen

Groß/Klein-Schreibung:

- Gliederung: Bezeichner werden durch eingestreute **Großbuchstaben** gegliedert. Bsp.: `zahlLesen`. Der Unterstrich (‘_’) ist nur für die weiter unten angegebenen Namenspräfixe reserviert.
- Typbezeichner: beginnt (nach eventuellem Namenspräfix) mit einem **Großbuchstaben**, Bsp.: `Punkt`. Alle anderen Bezeichner (Variablen, Konstanten, Methoden) beginnen mit einem **Kleinbuchstaben**., Bsp.: `final Punkt punkt, nachbarPunkt;`

  Gründe: Wir müssen aufgrund der Sprachregeln Typ und Variable des Typs mit unterschiedlichen Bezeichnern versehen. Die Konvention verhindert, daß pausenlos Synonyme eingeführt werden müssen. Als **Negativbeispiel** sei gegeben:

```
class punkt { ... }
...
final punkt ort;
```

  Viel besser ist:

```
class Punkt { ... }
...
final Punkt punkt;
```

Namenspräfixe für Parameterübergaberichtung

Die logische Parameterübergaberichtung (**in**, **out**, **in out**) jedes formalen Parameters soll dadurch ausgedrückt werden, daß jeder Parameterbezeichner mit einem der Präfixe '`i_`', '`o_`', '`io_`' versehen wird, ***Bsp***.:

```
/** Sortiert die ersten i_anzahl Elemente des Feldes io_feld */
static void sortieren(final long[] io_feld, final int i_anzahl){...}
```

Dabei ergibt sich, daß Parameter eines elementaren Typs nur den Präfix '`i_`' haben können, wohingegen bei Parametern eines Referenztyps nach der Übergaberichtung des Gezeigten (nicht der Referenz selbst) zu urteilen ist.

3.1.5 Attributzugriff

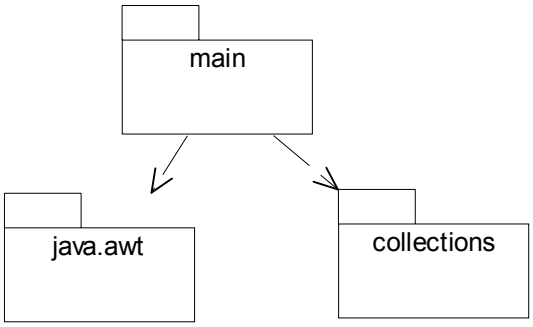
Jeder Zugriff auf ein Attribut der eigenen oder einer Oberklasse muss qualifiziert mittels **`this`** oder **`super`** erfolgen.

Bsp.:
```
class Person {
    private String name;
    String getName(){
        return this.name; //nur so erlaubt
    }
}
```

Gründe: Dies reduziert den globalen Namensraum. Am Namenspräfix **`this.`** kann der Definitionsort klassenweit statt methodenlokal erkannt werden.

Prüfung: Aktivieren Sie in Eclipse: Window > Preferences > Java > Compiler > Errors/Warnings > Unqualified access to instance field.

3.1.6 Import-Deklarationen



Import-Deklaration: Der Globalimport **`import`** `paket.*;` ist verboten.

Gründe:

a) Ohne Globalimport erzwingt der Compiler, daß an jeder Verwendungsstelle eines Typnamens dieser voll qualifiziert wird, d.h. mit Angabe des Paketnamens. Dies erhöht die Verständlichkeit des Codes und erleichtert das schnelle Auffinden der Typdefinition.

b) Der Globalimport mehrerer Pakete führt leicht zu verspäteten Namenskonflikten, ohne dass die Ursache dem Programmierer dann noch bewußt wäre. Bsp.: Im Paket `main` steht

```
import java.awt.*;
import collections.*;
```

und wird der Typ `List` aus `java.awt` benutzt. Es geht alles gut. Später kommt der Autor des Pakets `collections` auf die gute Idee, einen Typ `List` zu exportieren. Dann tritt in Paket `main` ein Namenskonflikt ein, ohne daß an `main` etwas geändert wurde.

3.1.7 static-Variablen

static-Variablen sind verboten. Demgegenüber sollten Konstanten zwecks Speicherplatzersparnis sogar **static** sein.
Bsp.: Verboten sind:

```
static final LgPerson _aktuellerBenutzer = new LgPerson(); //änderbar
static String _letzteWebseite; //ersetzbar
```

Stattdessen sollen die benötigten Zustände/Abhängigkeiten als Objektattribute einer Kontextklasse gespeichert und möglichst im Kontruktor einfrierend initialisiert werden. Erlaubt sind daher:

```
private final LgPerson _aktuellerBenutzer = new LgPerson();
private String _letzteWebseite;
```

Gründe:

Um Komponentenorientierung (siehe Kapitel 1.9) durchzusetzen, muss feststehen, von welchen anderen Komponenten eine Komponente abhängig ist. Jede Komponente wird dabei als ein Objekt dargestellt, jede Abhängigkeit als eine Objektreferenz. Globale Objekte sind, wie früher globale Variablen, verpönt. Das **Singleton**-Muster gilt daher ausdrücklich als schädlich! Siehe:

http://www.c2.com/cgi/wiki?SingletonsAreEvil
http://archive.eiffel.com/doc/manuals/technology/bmarticles/joop/globals.html

Methoden:

static-Methoden sind unproblematisch, solange sie nicht auf **static**–Variablen zugreifen. Eine **static**-Methode darf daher nur auf Variablen zugreifen, die ihr als Parameter übergeben wurden. Dann ist sie eine reine Utility-Methode.

3.1.8 Typspezialisierung (type cast)

Eine explizite Typspezialisierung mittels „type cast“ verschiebt die Typprüfung vom Übersetzungszeitpunkt auf den der Programmausführung (d.h. beim Kunden). ***Daher***:

- möglichst ersetzen durch Aufruf einer abstrakten Methode! ***Bsp***. `.toString()`
- unvermeidliche Typspezialisierung an Ort mit dem größten Wissen über Erfolg konzentrieren, d.i. meist in der Klasse, zu der man hinspezialisiert oder im Fall einer typsicheren generischen List in dieser.
  Bsp.: `ELEMENT_List` in http://www.tfh-berlin.de/~knabe/java/lib/

3.1.9 Steuerkonstrukte und geschweifte Klammern

Jeder Rumpf eines Steuerkonstruktes (**if**, **else**, **while**, **for**, **do**, **switch**) ist mit geschweiften Klammern zu versehen, auch wenn er aus nur einer Anweisung besteht.

Gründe: Die häufig verwendete eingerückte Schreibweise ohne geschweifte Klammern lässt nachträglich eingefügte Anweisungen fälschlicherweise als Teil des gesteuerten Rumpfes erscheinen. Ein korrektes Einfügen hingegen würde ein Einfügen an zwei auseinander liegenden Stellen erfordern. ***Bsp***.:

```
for(int i = 0; i<N; i++)
    values[i]++;
    System.out.println(values[i]);
System.out.println("============================");
```

Hintergrund: In Sprachen wie Ada oder Modula-2 klammern die Schlüsselwörter **if** … **end if** von sich aus. In Perl mit seiner C-ähnlichen Syntax werden immer geschweifte Klammern verlangt.

3.1.10 Steuerkonstrukte mit fehlendem else-Zweig

Alle mehrzweigigen Steuerkonstrukte (**`if`**, **`switch`**) müssen mit einem Restfall-Zweig versehen werden. Wenn der Restfall nicht erwartet wird, ist sein Betreten ein Fehler und sollte folgerichtig eine Ausnahme geworfen werden. ***Beispiele***:

```
final String command = … ;
if(command.equals(printCommand){
    print(filename);
}else if(command.equals(deleteCommand){
    delete(filename);
}else{
    throw new multex.Exc(
        "Kommando {0} unerwartet aufgetreten.", command
    );
}
```

```
final int command = … ;
switch(command){
    case printCommand:
        print(filename);
    break;
    case deleteCommand:
        delete(filename);
    break;
    default:
        throw new multex.Exc(
            "Kommando {0} unerwartet aufgetreten."
            , new Integer(command)
        );
    break;
}
```

3.1.11 Entwicklungsdokumentation

Jede Einheit (Paket, Klasse, Methode, Attribut) muss mit einem Javadoc-Kommentar versehen sein. Bei einem Paket, einer Klasse oder einem Attribut beschreibt man den Zweck, bei einer Methode spezifiziert man ihren Effekt bzw. ihre Ausnahmen. Die Paket-Doku muss in einer Datei `package.html` im zugehörigen Verzeichnis stehen.

3.2 Quellenangaben

[Ambler] Ambler, Scott W.: „Mapping Objects To Relational Databases“, Ronin International White Paper, http://www.AmbySoft.com/mappingObjects.pdf , 1998-2000.
Ausführliche Abwägung verschiedener Strategien des Object-Relational Mapping.
Ambler, Scott W.: „Design of a Robust Persistence Layer for Relational Databases“, Ronin International White Paper, http://www.AmbySoft.com/persistenceLayer.pdf , 1997-2000.
Entwurf einer Object-Relational-Mapping-Schicht und seine Begründung.

[Balzert99] Balzert, Heide: "Lehrbuch der Objektmodellierung. Analyse und Entwurf", Spektrum Akademischer Verlag, 1999, 573pp., ca. 50,- €.
Didaktisch sehr gut ausgearbeitet, dennoch erkennbar mit praktischer Erfahrung, deckt außer dem Bereich "Testen" alles Relevante für unser Software-Projekt ab, insbesondere Objektorientierte Analyse, Oberflächenentwurf, 3-Schichten-Architektur, Object-Relational Mapping.

[Bank3Tier] http://www.tfh-berlin.de/~knabe/java/bank3tier/:
Eine Demo-Java-Applikation aus dem Bankwesen (Kunde, Konto, überweisen) in 3 Schichten.

[cvshome] www.cvshome.org:
Übersicht über CVS und damit zusammenhängende Produkte.

[cvsgui] www.cvsgui.org:
Übersicht über GUI-Klienten für CVS.

[Database] www.javaskyline.com/database.html:
Übersicht über generische OR-Mapping-Tools.

[Fogel99] Fogel, Karl: „Open Source Development with CVS“, 446pp., Coriolis Group 1999 <en>, Bonn 2000 <de>.
Einführung in CVS und fortgeschrittene Benutzungsstrategien für Open-Source-Projekte.

[Gamma95] Gamma, Helm, Johnson, Vlissides: Entwurfsmuster, <de> Addison-Wesley, Bonn, 1996, <en> 1995
Das Standardwerk über Entwurfsmuster, aus dem die angeführten Muster entnommen sind.

[JORA] Knizhnik, Konstantin: Java Object Relational Adapter (JORA):
www.tfh-berlin.de/~knabe/java/jora/
Eine auf Diagnosestärke mittels [MulTEx] angepaßte Variante von JORA.

[Link] Link, Johannes: Softwaretests mit JUnit. Techniken der testgetriebenen Entwicklung, 2. überarb. Auflage, dpunkt-Verlag, Heidelberg 2005, 416pp.
Ausführliche Darstellung aus der Schule des Extreme Programming. Mit Hilfen auch für kompliziertere Fälle.

[MulTEx] Knabe, Christoph: „Ein Framework zur Ausnahmebehandlung in mehrschichtigen Softwaresystemen“, www.tfh-berlin.de/~knabe/java/multex/
MulTEx = Multi-Tier Exception Handling, Eine Strategie und ein dazu passendes Java-Framework zur diagnosestarken Ausnahmebehandlung in mehrschichtigen Softwaresystemen.

[Programmierstil]: http://www.tfh-berlin.de/~knabe/fach/pr1/Programmierstil/
Ein Vortrag zur Motivation der Richtlinien Initialisiertheit, Konstantheit, Lokalität, Redundanzfreiheit, Schachtelungsorientierte Einrückung und Abbrechende Fehlerprüfung.

[Siedersleben]: Siedersleben, Johannes: Moderne Softwarearchitektur. Umsichtig planen, robust bauen mit Quasar.
Ein hervorragend systematisches Buch aus der Praxis des qualitätsbewussten Softwarehauses sd&m.