sd&m Research

Quasar: Die sd&m Standard-architektur

Teil 2

Quasar

Johannes Siedersleben
(Hrsg.)

Quasar

Quality Software Architecture

Zusammenfassung

Dieses Papier beschreibt die Standardarchitektur betrieblicher Informationssysteme bei sd&m. Der vorliegende Teil 2 behandelt die folgenden Themen:

- Transaktionen (Kapitel 5)
- Persistenz (Kapitel 6)
- Graphische Benutzerschnittstellen (Kapitel 7)
- Verteilte Verarbeitung (Kapitel 8)

Teil 2 ist Ergebnis mehrerer Arbeitsgruppen im Rahmen des Projekts „Themenarbeit", die unsere Projekterfahrungen zusammengetragen und verdichtet haben. Wir hoffen, dass künftige Projekte auf der Basis des jetzt verfügbaren Wissens noch besser und effizienter arbeiten werden.

Johannes Siedersleben

Inhaltsverzeichnis

2. Auflage 5/03

5 Transaktionen

5.1 Einführung

Jede Transaktion ist eine Arbeitseinheit (unit of work): eine zusammengehörige Menge von Operationen auf einem gemeinsamen Datenbestand, der auf mehrere Orte verteilt sein kann und über unterschiedliche Dienste angesprochen wird. Transaktionen genügen der ACID-Eigenschaft: Atomicity, Consistency, Isolation, Durability.

Transaktionen werden über drei Basisoperationen gesteuert: *begin* startet eine Transaktion, *commit* bestätigt die Transaktion (macht also die Änderungen dauerhaft), und *rollback* macht die Transaktion ungeschehen. Jede Transaktion lebt (oder ist aktiv) von *begin* bis *commit* oder *rollback.*

Transaktionen werden von transaktionsfähigen Diensten durchgeführt. Diese nennt man Transaktionsressourcen oder einfach Ressourcen. Beispiele sind:

- Datenbankmanagementsysteme,
- Nachrichtensysteme (Message-oriented Middleware, MOM),
- Schnittstellen zu Nachbarsystemen,
- EJB-Container.

Transaktionen, die innerhalb einer Ressource stattfinden, nennt man *einfach* oder *lokal,* ressourcenübergreifende Transaktionen sind *verteilt.* Das XA-Protokoll (www.opengroup.org; [X/Open 1996]) beschreibt das Zusammenspiel von Anwendung, Transaktions- und Ressource-Managern (vgl. Abschnitt 5.2). Im Umfeld von Transaktionen gibt es neben dem XA-Protokoll zahlreiche Normen. Genannt seien hier:

- JTA (Java Transaction API, Sun). JTA ist eine Menge von Schnittstelle, die das XA-Protokoll für Java-Anwendungen festlegt. Abgesehen von wenigen Ausnahme-Klassen enthält JTA keine Implementierung.
- OTS (Object Transaction Service, OMG). OTS ist eine Menge von Schnittstellen, die das XA-Protokoll für CORBA-Anwendungen festlegt.
- JTS (Java Transaction Service). JTS ist eine JTA-Implementierung gegen OTS.

Die Fülle der Normen und Implementierungen macht die Sache für den Anwendungsprogrammierer nicht einfacher, sondern komplizierter. In Wirklichkeit ist der Transaktionsbegriff in der Praxis hoffnungslos überladen: Transaktionen gibt es bei Datenbanken, bei TP-Monitoren (z.B. CICS) und bei Applikationsservern (z.B. für EJB).

Deshalb trennen wir den Transaktionsbegriff aus Sicht der Anwendung – die *A-Transaktion* – von ihrer Implementierung durch technische Hilfsmittel, der *T-Transaktion.* A-Transaktionen lassen sich ganz unterschiedlich auf T-Transaktionen abbilden: Im einfachsten Fall entspricht jeder A-Transaktion genau eine T-Transaktion (Abbildung 1 links), aber man kann durchaus eine A-Transaktion auf mehrere Transaktionen eines EJB-Servers abbilden, die ihrerseits durch mehrere Datenbank-Transaktionen dargestellt werden (Abbildung 1 rechts).

A-Transaktionen orientieren sich ausschließlich an der Anwendung: A-Transaktionen können lang dauern (Minuten oder auch Stunden). Technische Restriktionen (mit/ohne Gedächtnis, kurze Dauer) beziehen sich nur auf T-Transaktionen. Es ist ohne weiteres möglich, A-Transaktionen mit Gedächtnis auf gedächtnislose T-Transaktionen abzubilden (indem man z.B. das Transaktionsgedächtnis in die Datenbank schreibt).

Der Rest des Kapitels befasst sich mit den folgenden Themen: Abschnitt 5.2 erläutert das XA-Protokoll für verteilte Transaktionen, Abschnitt 5.3 erklärt parallele und geschachtelte Transaktionen. Abschnitt 5.4 erläutert die Quasar-Architektur für A-Transaktionen. Diese orientiert sich an zwei Vorbildern, nämlich:

a) XA-Architektur für verteilte Transaktionen
b) JTA, und dort vor allem die Schnittstelle *UserTransaction.*

Abschnitt 5.5 erläutert die Quasar-Schnittstellen für Transaktionen im Detail. Der Wert dieser Schnittstellen liegt zum Einen in der vollständigen Unabhängigkeit von der Technik, zum Anderen in der Nähe zu JTA. Wer also eine JTA-Implementierung an der Hand hat, kann diese Quasar-konform ohne weiteres nutzen. Abschnitt 5.6 befasst sich mit Transaktionen bei Client/Server-Anwendungen, und Abschnitt 5.7 behandelt das spezielle Thema paralleler Transaktionen innerhalb einer Sitzung.

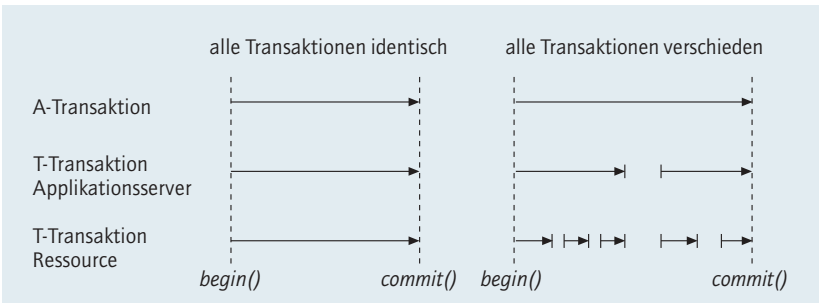


Abbildung 1:
Abbildung von
A- auf T-Transaktionen

5.2 Verteilte Transaktionen

Mehrere lokale Transaktionen $S_1, S_2, .. S_n$ kann man zu einer verteilten Transaktion T verbinden. Das bedeutet:
T ist genau dann erfolgreich, wenn alle Transaktionen $S_1, S_2, .. S_n$ erfolgreich sind. Technische Voraussetzung ist hierfür das 2-Phasen-Commit (2PC), das im XA-Protokoll definiert ist. Hier der Ablauf:

Das Anwendungsprogramm eröffnet die Transaktion gegenüber dem Transaktionsmanager *(begin).* Eine Identifikation dieser Transaktion wird als Transaktionskontext bei allen Aufrufen *(invoke)* der Anwendung an die Ressourcenverwalter mit übergeben. Durch Registrierung beim Transaktionsverwalter *(join)* mit dieser Identifikation wird der jeweilige Ressourcenverwalter Teil der Transaktion (Abbildung 2).

Die Anwendung ruft die Ressourcenverwalter über die von der jeweiligen Ressource abhängigen *invoke*-Operationen. Nach getaner Arbeit erklärt die Anwendung die Transaktion dem Transaktionsverwalter gegenüber für abgeschlossen *(commit).* Dies teilt der Transaktionsverwalter den Ressourcenverwaltern in zwei Phasen mit:

In der ersten Phase informiert der Transaktionsverwalter alle Ressourcenverwalter, dass die Transaktion abgeschlossen werden soll *(prepare).* Für die Ressourcenverwalter ist das der letzte Zeitpunkt, zu dem sie Konflikte wegen Sperren oder paralleler Zugriffe melden können. Ein Ressourcenverwalter, der eine *prepare*-Aufforderung positiv beantwortet, garantiert, dass die mit der Transaktion verbundenen Änderungen dauerhaft festgeschrieben werden und zwar sogar für den Fall eines Systemabsturzes zwischen *prepare* und *commit.* Gleichzeitig garantiert er, dass die Änderungen bei einem Abbruch der Transaktion zurückgesetzt werden können. Der eigentliche *commit* erfolgt dann in Phase 2.

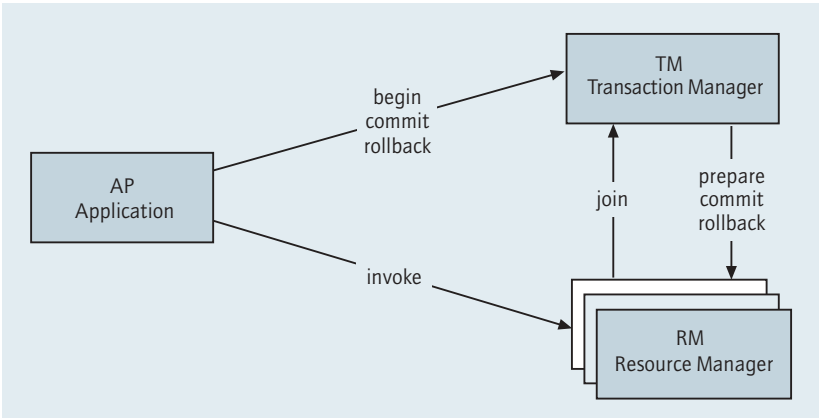
Der Ablauf von Phase 2 hängt ab von den Antworten der Ressourcenverwalter. Wenn alle Rückmeldungen positiv sind, fordert er in der Phase 2 die Ressourcenverwalter auf, die Transaktion endgültig abzuschließen *(commit).* Wenn aber auch nur eine negative Rückmeldung dabei ist, wird die gesamte Transaktion abgebrochen: Alle Ressourcenverwalter der Transaktion machen ihre Änderungen rückgängig *(rollback).*

So erfüllt der Transaktionsverwalter das *Alles-oder-Nichts*-Prinzip: Für die Anwendung selbst stellt der Abschluss der Transaktion nur einen Aufruf dar, der eine positive oder eine negative Antwort liefert. Das hinter der Fassade des Transaktionsverwalters ablaufende Protokoll bleibt ihr verborgen.

Den Abbruch einer Transaktion *(rollback)* kann natürlich auch die Anwendung selbst initiieren. In diesem Fall informiert der Transaktionsverwalter alle beteiligten Ressourcenverwalter über den Abbruch. Vier Hinweise sind an dieser Stelle angebracht:

a) Die Implementierung des 2-Phasen-Commit-Protokolls ist außerordentlich komplex. Deshalb gibt es nur wenige belastbare Implementierungen. Man sollte auf keinen Fall im Projekt eine eigene Implementierung angehen.
b) Das 2-Phasen-Commit-Protokoll koppelt die beteiligten Systeme sehr eng: Wenn z.B. auch nur eines der beteiligten Systeme nicht erreichbar ist, kann keine einzige Transaktion erfolgreich sein. Deshalb wird 2-Phasen-Commit manchmal nicht verwendet, obwohl es verfügbar wäre.
c) Es ist sorgfältig zu prüfen, ob der technische Mechanismus des 2-Phasen-Commits bereits alle Probleme löst. Oft ist es nötig, das Versagen oder die Nichterreichbarkeit eines Nachbarsystems auf Anwendungsebene zu behandeln.
d) 2-Phasen-Commit wird in der Praxis nur selten eingesetzt, da die meisten Ressourcen, die in Transaktions-Systemen eingebunden werden (Backend- oder Legacy-Systeme) kein 2PC-Protokoll unterstützen. Es gibt eine Reihe von Techniken, um eine fachlich ausreichende Konsistenz von Daten ohne Verfügbarkeit eines technischen 2PC herzustellen. Beispiel: fachliche *undo*-Operationen.

Abbildung 2: Ablauf einer XA-Transaktion



5.3 Parallele und geschachtelte Transaktionen

Parallele Transaktionen

Parallele Transaktionen bedürfen eigentlich keiner gesonderten Erwähnung, weil der Sinn von Transaktionen gerade darin besteht, dass sie parallel ablaufen und die Arbeit verschiedener Prozesse koordinieren. Allerdings ist es aus technischen Gründen schwierig, parallele Transaktionen innerhalb einer Sitzung ohne eine geeignete parallele Architektur der Anwendung (z.B. mit Hilfe von Threads) herzustellen. Aus Sicht des A-Programmierers ist diese Einschränkung hinderlich: Er will z.B. einen fachlichen Schlüssel vergeben und sichern (Transaktion 1) und parallel dazu Änderungen durchführen (Transaktion 2). Abschnitt 5.7 befasst sich mit der Frage, wie man innerhalb einer Sitzung (vgl. Kapitel 7) mehrere parallele Transaktionen verwaltet. Bis einschließlich Abschnitt 5.6 gehen wir davon aus, dass zu jedem Zeitpunkt in jeder Sitzung höchstens eine Transaktion aktiv ist.

Geschachtelte Transaktionen

Man kann Transaktionen schachteln (vgl. Abbildung 3). Die Aussage „Transaktion S ist Subtransaktion von Transaktion T" bedeutet folgendes:

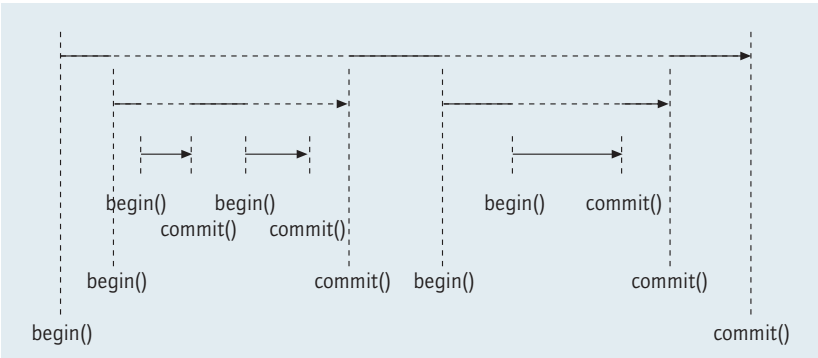
a) S beginnt nach T.
b) T kann erst nach dem Ende von S (*commit* oder *rollback*) beendet werden.
c) S sieht alle unbestätigten Änderungen von T (und alles, was T sonst sieht; vgl. Isolationsstufen).
d) T sieht nicht die unbestätigten Änderungen von S. Die bestätigten Änderungen von S werden Teil von T und werden mit T bestätigt oder verworfen.

Standardanwendung geschachtelter Transaktionen sind Was-wäre-wenn-Analysen: Sie gestatten dem Benutzer, Szenarien durchzuspielen und deren Konsequenzen zu beurteilen. Geschachtelte Transaktionen verwendet man auch, wenn man eine lange Folge von Einzelschritten nicht komplett zurücksetzen möchte, falls einer dieser Schritte fehlschlägt.

Geschachtelte Transaktionen werden eingesetzt in Situationen, in denen einzelne Teilabläufe innerhalb der Transaktion das erfolgreiche Beenden der Haupttransaktion nicht beeinflussen dürfen. Dies könnte zum Beispiel der Zugriff auf eine nicht immer erreichbare Ressource sein, die jedoch nicht entscheidend für den Ablauf ist. Sie sind auch hilfreich, um in geschachtelten Dialogen das Verwerfen eines Unterdialogs zu implementieren.

Geschachtelte Transaktionen funktionieren genauso wie Toplevel-Transaktionen: Jede Subtransaktion hält sich den Zustand der Daten zu Beginn und zum Ende der Transaktion, um die separate Rücksetzbarkeit zu gewährleisten. Der Verbrauch an Ressourcen wird also bestimmt von der Anzahl der gleichzeitig offenen Transaktionen – egal auf welcher Ebene sich diese befinden.

Abbildung 3: Geschachtelte Transaktionen



5.4 Architektur

Die Quasar-Architektur zur Transaktionsverwaltung orientiert sich an dem XA-Modell. Sie definiert eine einfache, minimale Schnittstelle für die Anwendungssteuerung, die weitgehend mit der JTA-Schnittstelle übereinstimmt.

5.4.1 Transaktionsmanager

Im Zentrum der Transaktionsverwaltung steht der Transaktionsmanager, der eine oder mehrere Transaktionsressourcen verwaltet. Neben der Einteilung in lokale und verteilte Transaktionen kann man orthogonal dazu zwei Arten der Transaktionsverwaltung unterscheiden:

a) flache Transaktionen
b) geschachtelte Transaktionen

Der Transaktionsmanager implementiert die Schnittstelle *ITransactionManager,* die nur von der A-Steuerung aufgerufen wird.

5.4.2 A-Steuerung

Die Anwendungssteuerung (A-Steuerung) vermittelt zwischen dem Anwendungskern und dem Transaktionsmanager. Sie startet und beendet Transaktionen, und sie ist auch zuständig für die Registrierung der Vor- und Nachbehandlung. Die A-Steuerung ist im Dialogbetrieb Teil des Dialogs, im Batch Teil der Batchsteuerung. Sie ist A-Software und ruft die Operationen der A-Fälle des Anwendungskerns unter der Kontrolle von A-Transaktionen auf.

5.4.3 Anwendungskern

Der Anwendungskern kennt keine Transaktionen. Er läuft gegen eine oder mehrere Ressourcen, deren wichtigste der Persistenzmanager ist (vgl. Kapitel 6). Die Transaktionsverwaltung macht keine Annahmen über die Schnittstelle zwischen Anwendungskern und Ressourcen. Diese Schnittstelle entspricht dem *invoke* des XA-Protokolls.

5.4.4 Ressourcen

Jede Ressource ist transaktionsfähig. Sie implementiert also selbst die Operationen *commit* und *rollback.* Manche Ressourcen implementieren darüberhinaus das 2-Phasen-Commit-Protokoll mit der zusätzlichen Operation *prepare* (vgl. hierzu die *XAResource*-Schnittstelle in JTA). Es ist Aufgabe des Transaktionsmanagers und der Ressourcenverwalter, die A-Transaktionen, die er in seiner Schnittstelle anbietet, auf die T-Transaktionen der Ressourcen abzubilden.

5.4.5 Konfiguration

Die Konfiguration hat die Aufgabe, A-Steuerung, Transaktionsmanager, Ressourcen und Anwendungskern zusammenzubringen. Wir betrachten zunächst den einfachen Fall mit nur einem Transaktionsmanager. Wir unterstellen, dass geeignete Fabrikklassen zur Verfügung stehen. Die Konfiguration läuft folgendermaßen ab:

a) Die Konfiguration erzeugt die nötigen Ressourcen, im allgemeinen mindestens einen Persistenzmanager. Die Ressourcen werden mit den gewünschten Einstellungen konfiguriert (beim Persistenzmanager sind das z.B. Sperrstrategie und Isolationsstufe).
b) Die Konfiguration erzeugt einen Transaktionsmanager und gibt ihm seine Ressourcen bekannt. Dies geschieht über geeignete Registrierungsmethoden. Der Transaktionsmanager wird mit den gewünschten Einstellungen (z.B. Timeout) konfiguriert.
c) Die Konfiguration erzeugt ein Exemplar des Anwendungskerns und gibt ihm – falls erforderlich – die nötigen Ressourcen bekannt.
   Anmerkung: Man kann den Anwendungskern so schreiben, dass er eine oder alle Ressourcen sieht, oder umgekehrt in der Weise, dass der Anwendungskern bei der Ressource registriert wird, die Ressourcen selbst aber für den Anwendungskern unsichtbar sind.
d) Die Konfiguration erzeugt eine A-Steuerung und gibt dieser den Transaktionsmanager und das Exemplar des Anwendungskerns bekannt.

Jetzt ist die A-Steuerung lauffähig.

5.5 Schnittstellen im Detail

Alle Schnittstellen dieses Abschnitts sind Schnittstellen aus Sicht der Anwendung. Sie befassen sich also mit A-Transaktionen. Es ist Aufgabe geeigneter T-Software, diese Schnittstellen auf die vorhandene TI-Architektur abzubilden.

5.5.1 ITransactionManager

Die Schnittstelle des Transaktionsmanagers hat zwei Ausbaustufen:
a) flache Transaktionen
b) geschachtelte Transaktionen

Wir betrachten zunächst nur flache Transaktionen: Zu jedem Zeitpunkt ist höchstens eine Transaktion aktiv. Sie wird mit *begin* gestartet, mit *commit* bestätigt und mit *rollback* zurückgesetzt. Die Methode *getTransaction* liefert – falls vorhanden – die aktuelle Transaktion; im anderen Fall liefert sie *null.* Die Methode *setRollbackOnly* bewirkt, dass die Transaktion nur noch mit *rollback* beendet werden kann.

```
public interface ITransactionManager {
    void begin();
    void commit() throws ResourceException;
    void rollback();
    void setRollbackOnly();
    ITransaction getTransaction();
}
```

Diese Schnittstelle entspricht bis auf die bei uns nicht vorgesehene Methode *setTransaction-Timeout* der *UserTransaction* von JTA. Der Aufruf von *commit, rollback* und *setRollbackOnly* ist nur konsistent, wenn eine Transaktion aktiv ist. Der Aufruf von *begin* ist nur konsistent, wenn keine Transaktion aktiv ist (Zustand *active* oder *rollbackonly,* vgl. Abbildung 4).

Die *rollback*-Methode glückt in konsistenten Situationen immer; alles andere ist ein Notfall (vgl. Kapitel 1). *commit* kann aus unterschiedlichen, Ressource-abhängigen Gründen scheitern.

Die Schnittstelle *ITransaktionManager* eignet sich auch für geschachtelte Transaktionen. Dazu bekommt die Methode *begin* eine andere Bedeutung: Falls eine Transaktion aktiv ist, wird eine neue, geschachtelte Transaktion gestartet. Auch in diesem Fall gibt es zu jedem Zeitpunkt höchstens eine aktive Transaktion, nämlich die oberste. *commit, rollback* und *getTransaction* beziehen sich immer auf diese oberste Transaktion.

Zur Erläuterung zeigen wir ein Code-Beispiel aus der A-Steuerung:

Code-Beispiel aus der A-Steuerung

```
public class MyAppControl {
    ..
    private ITransactionmanager mgr = ..    // Konfiguration
    private MyApplication app = ..          // Konfiguration
    ..
    void someMethod() {
        mgr.begin();                        // top level Transaktion
        app.useCase1( .. );                 // A-Fall 1
        mgr.begin();                        // geschachtelte Transaktion wird gestartet
        app.useCase2( .. );                 // A-Fall 2
        if ( .. )
            mgr.commit();                   // geschachtelte Transaktion wird beendet
        else
            mgr.rollback();
        if ( .. )
            mgr.commit();                   // top level Transaktion wird beendet
        else
            mgr.rollback();
    }
    ..
}
```

5.5.2 ITransaction

Die *ITransaction*-Schnittstelle braucht man nur zur Registrierung von Synchronisationen (vgl. Abschnitt 5.5.3). Jede Transaktion gehorcht dem folgenden Zustandsmodell (Abbildung 4):

Jede begonnene Transaktion ist also zunächst im Zustand *active.* Es gibt drei Möglichkeiten: *commit* befördert die Transaktion zunächst in den Zustand *committing,* dann in *comitted. rollback* verhält sich analog. *setRollbackOnly* versetzt die Transaktion in den Schwebezustand *marked_rollback,* der *rollback* als einzige Option übrig lässt.

```
public interface ITransaction {
    int getLevel();
    ITransaction getParent();
    void register(ISynchronization synchronization);
    TransactionState getState();
}
```

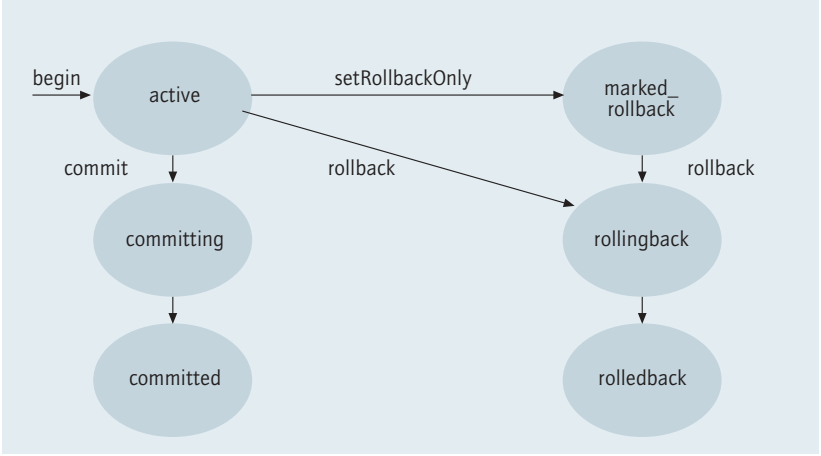
Mit der Methode *getState* wird der Status der Transaktion ermittelt. Die Methoden *getLevel* und *getParent* liefern bei geschachtelten Transaktionen die Ebene (Anzahl der Vorfahren) und die umgebende Transaktion (parent). Mit *register* werden Synchronisationen (siehe nächster Abschnitt) bei der Transaktion registriert.

5.5.3 ISynchronization

Diese Schnittstelle ist eine Call-Back-Schnittstelle, mit deren Hilfe der Ressourcenmanager spezielle Aktionen definiert, die unmittelbar vor bzw. unmittelbar nach einem *Commit* durchgeführt werden.
Die *afterCompletion*-Methode erfährt den Ausgang des *Commit* über den Status.

```
public interface ISynchronization {
    void beforeCompletion();
    void afterCompletion(int status);
}
```

Abbildung 4:
Zustandsmodell
einer Transaktion



5.6 Transaktionen im Client/Server-Betrieb

Jeder A-Fall läuft unter Kontrolle einer A-Transaktion. Die A-Fälle sind Teil des Anwendungskerns und werden von einer übergeordneten Instanz gerufen, die die A-Steuerung übernimmt (Dialog oder Batch). Bei der Koppelung von Dialogen und A-Fällen unterscheiden wir vier Typen:

Einschritt-Transaktion (Abbildung 5)
Die A-Transaktion enthält genau einen A-Fall-Aufruf.

Abbildung 5:
Einschritt-Transaktionen

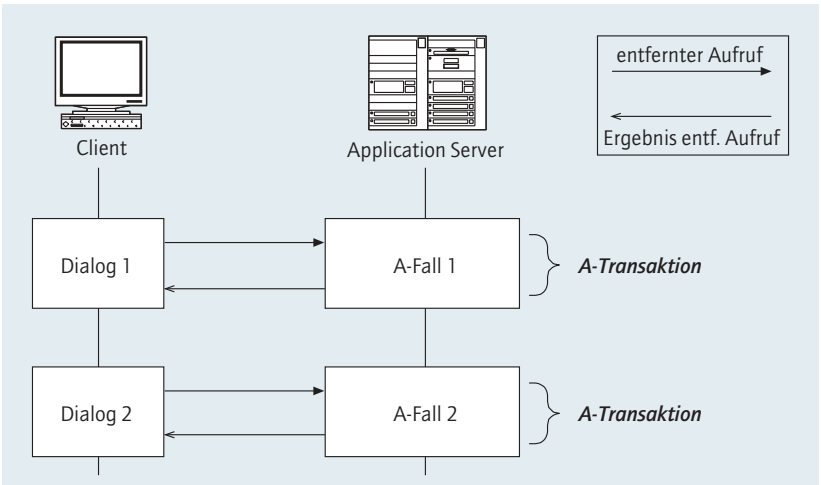
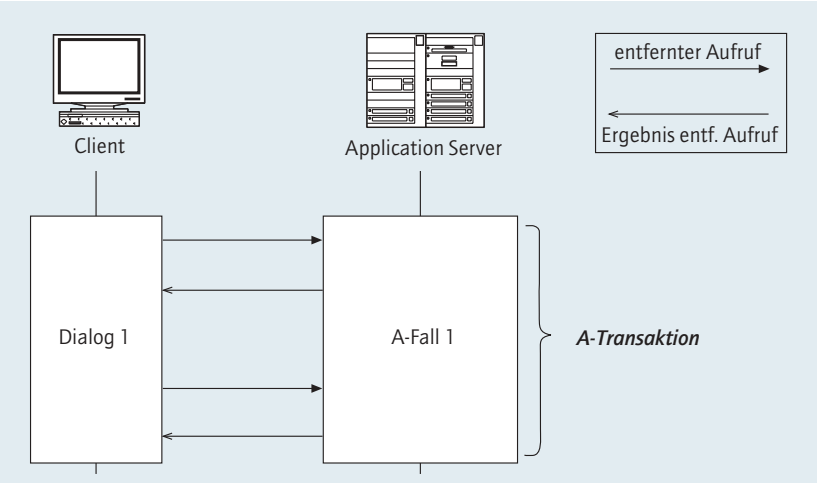


Abbildung 6:
Mehrschritt-
Transaktion



Mehrschritt-Transaktion ohne Benutzereingabe (Abbildung 6)
Die A-Transaktion enthält mehrere A-Fall-Aufrufe ohne Benutzereingabe. Die Aufrufe erfolgen also kurz hintereinander.

Mehrschritt-Transaktionen mit Benutzereingabe (Abbildung 7)
Die A-Transaktion enthält mehrere A-Fall-Aufrufe mit Benutzereingabe. Die Aufrufe erfolgen also in Abständen, die sehr groß sein können (bis hin zu Stunden). Sie finden allerdings innerhalb einer Sitzung statt – der Benutzer bleibt angemeldet.

Lange Dialoge
Lange Dialoge sind keine Transaktionen, wir erwähnen sie hier nur zur Abgrenzung. Lange Dialoge erstrecken sich über mehrere Sitzungen des Benutzers: Zwischenstände werden gespeichert und stehen dem Benutzer bei weiteren Sitzungen wieder zur Verfügung. Die ACID-Eigenschaften gelten nicht. Bei der Wiederaufnahme des Dialogs (nach Stunden, Tagen oder Wochen) wird der alte Stand wieder hergestellt und auf Konsistenz in der neuen Situation geprüft. All dies ist Sache der Anwendung.

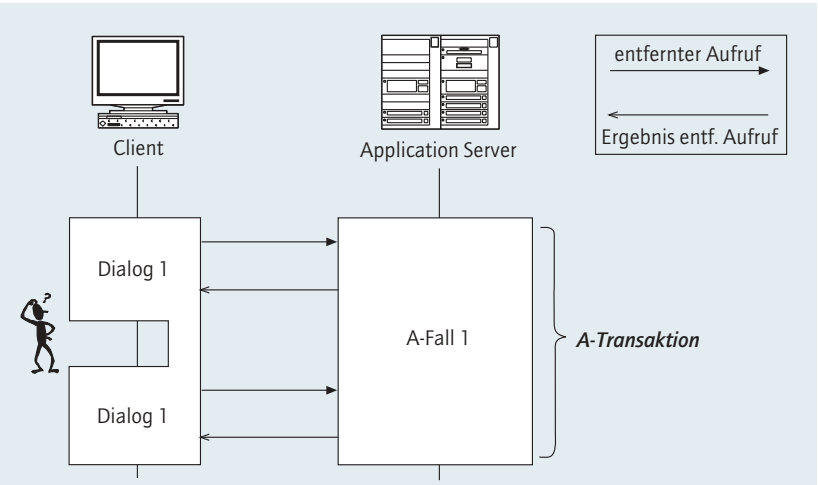


Abbildung 7:
Mehrschritt-Transaktion
mit Benutzereingabe

Bewertung
Einschritt-Transaktionen bereiten bei der Implementierung wenige Probleme, da sie sich unmittelbar auf T-Transaktionen abbilden lassen. Mehrschritt-Transaktionen besitzen in der Regel ein Gedächtnis, das irgendwo aufzubewahren ist. Dafür gibt es mehrere Möglichkeiten: Im einfachsten Fall hat auch die T-Transaktion ein Gedächtnis. Wenn dies nicht akzeptabel ist, wird man T-Software schreiben, um das Gedächtnis der A-Transaktion an einer geeigneten Stelle aufzuheben (z.B. in der Datenbank).

5.7 Parallele Transaktionen

Transaktionen sind dazu da, parallele Bearbeitung überhaupt erst zu ermöglichen. Allerdings sind in der Praxis Transaktionen fast immer mit einer Datenbankverbindung verknüpft: Innerhalb jeder Verbindung ist zu jedem Zeitpunkt immer nur eine Transaktion aktiv. Wer also innerhalb einer Sitzung parallele Transaktionen benötigt, kann dies entweder über mehrere Datenbankverbindungen tun (das ist teuer), oder er kann eine der folgenden vier Möglichkeiten verwenden:

Lösung 1:
Angabe der Transaktion bei jedem Aufruf

Man kann das Problem einfach lösen, indem man bei jedem Aufruf einer Ressource die Transaktion als zusätzliches Argument übergibt. Damit ist die Transaktion überall sichtbar; der Anwendungskern ist gezwungen, diesen zusätzlichen Parameter an die Ressourcen weiterzuleiten.

Lösung 2:
Einsteuern der Transaktion über eine transaktionsspezifische Fassade

Um die Transaktion als Extraparameter bei jedem Aufruf zu verstecken, kann man den Dienst hinter einer Fassade verbergen, die die Transaktion bei jedem Aufruf einsteuert. Die Fassade würde zu Beginn einer Transaktion einmalig erzeugt. Diese Lösung wird häufig bei verteilten Transaktionen angewendet. An der Schnittstelle zu einem entfernten System wird die ID der jeweils gültigen, verteilten Transaktion eingesteuert.

Lösung 3:
Assoziation der Transaktion mit dem Thread, in dem sie läuft.

Diese Lösung beruht darauf, dass jede Transaktion durch den Thread, in dem sie läuft, eindeutig bestimmt ist. Das heißt, dass zu jedem Zeitpunkt höchstens eine Transaktion in einem Thread laufen kann. Diese Idee wird in JTA/JTS verwirklicht: Der Transaktionsmanager kennt die Zuordnung von Thread zu Transaktion; jede Operation findet innerhalb eines Threads statt und kann somit ihrer Transaktion zugeordnet werden. Der Transaktionsmanager bietet in diesem Fall zwei zusätzliche Operationen:

```
public interface ITransactionManager {
    ..
    ITransaction suspend();
    void resume(ITransaction t)
}
```

Die Operation *suspend* unterbricht die dem aktuellen Thread zugeordnete Transaktion; die Transaktion befindet sich in der Obhut der Anwendung. Diese wird die unterbrochene Transaktion mit *resume* dem aktuellen Thread wieder zuordnen (dies kann auch ein anderer sein als der, in dem die Transaktion vorher lief).

Lösung 4:
Mehrere Transaktionsmanager, mehrere Anwendungskerne.

Diese Lösung setzt voraus, dass der Transaktionsmanager und der Anwendungskern mehrfach instanziierbar sind, was keinen Mehraufwand in der Programmierung bedeutet. Der Zusatzaufwand liegt in der Konfiguration. Die Transaktionen verschiedener Transaktionsmanager sind unabhängig voneinander.

Die gebräuchlichste Lösung ist die Nummer (3), die immer dann funktioniert, wenn verschiedene Transaktionen in verschiedenen Threads laufen.

6 Persistenz

6.1 Grundlagen

Persistenz bedeutet:

a) Daten werden über Sitzungsgrenzen hinweg gespeichert.
b) Man kann Daten nach verschiedenen Kriterien suchen und finden.
c) Man kann Daten anlegen, ändern und löschen.

Der bewusst allgemeine Begriff *Daten* umfasst sowohl Objekte im Sinn der Objektorientierung als auch Strukturen im Sinn von COBOL oder PL/1. Die folgende Betrachtung bezieht sich im Kern auf objektorientierte Sprachen wie Java, C# oder C++; alle Codebeispiele sind in Java. Die Übertragung auf nicht-objektorientierte Sprachen ist grundsätzlich möglich, erfordert aber z.T. weitergehende Überlegungen. Wir betrachten in diesem Kapitel nur Operationen innerhalb von Transaktionen (vgl. Kapitel 5).

Es ist auf der Anwendungsebene völlig unerheblich, wo die Objekte tatsächlich gespeichert werden: In einer oder mehreren Datenbanken oder – zu Testzwecken – nur im Hauptspeicher. Zentrale Komponente jeder Zugriffsschicht ist der Persistenzmanager. Er hat folgende Aufgaben:

- Er führt Abfragen aus.
- Er legt persistente Objekte an, er löscht sie, er reicht Änderungen an persistenten Objekten weiter an die Datenbank oder ein anderes Speichermedium, und er versorgt die gelesenen Objekte mit den in der Datenbank gespeicherten Eigenschaftswerten.

Besondere Beachtung verlangen Beziehungen zwischen Entitätstypen. Dabei leistet der Persistenzmanger folgendes:

- Bei referenzierten Objekten bestimmt der Persistenzmanager den Zeitpunkt des Nachlesens.
- Beim Ändern einer Beziehung ändert sich – falls vorhanden – auch die entsprechende Rückbeziehung.
- Beim Auflösen von Beziehungen zu abhängigen Objekten (Komposition) werden diese gelöscht (referentielle Integrität).

Unter dem Namen *O/R-Mapper* gibt es mehrere Produkte auf dem Markt, die die Aufgaben eines Persistenzmanagers ganz oder teilweise implementieren. Marktführer im Java-Bereich ist *TopLink (www.oracle.com).* sd&m bietet mit Quasar-Persistence einen selbstentwickelten Persistenzmanager an, der mit kommerziellen Produkten vergleichbar ist, in Projekten weiterentwickelt wird und als open source verfügbar ist *(www.openquasar.de).*

Der weitere Aufbau dieses Kapitels:
Wir zeigen zunächst, wie sich der Persistenzmanager in die Quasar-Architektur einordnet. Danach beschreiben wir die Schnittstelle des Persistenzmanagers, die die Anwendung weitgehend unabhängig von der verwendeten Persistenzmanagerimplementierung macht. Abschnitt 6.5 erläutert, welche Aufgaben ein Persistenzmanager intern leistet und wie sie gelöst werden. Dieser Abschnitt ist gedacht für Leser, die sich auch für die Interna eines Persistenzmanagers interessieren.

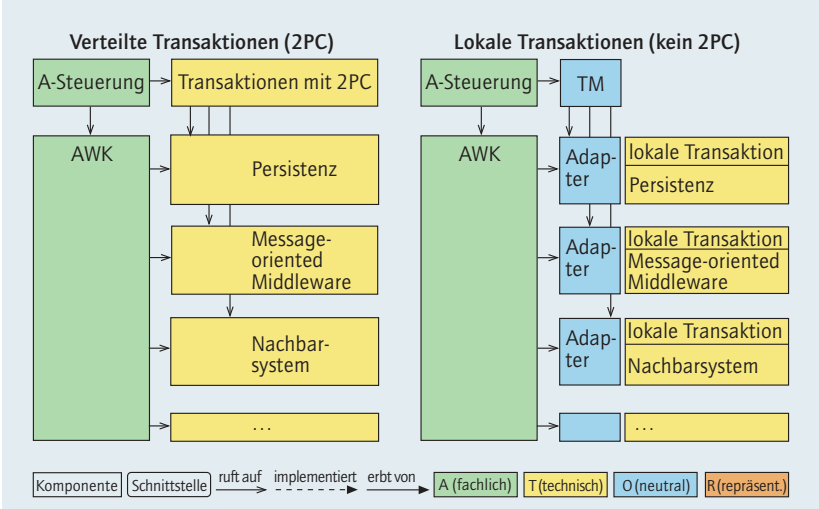
6.2 Architektur (Überblick)

Der Persistenzmanager ist ein Ressourcemanager im Sinn von Kapitel 5. Das Zusammenspiel von Ressourcemanager, Anwendungssteuerung, Transaktionsmanager und der Anwendung selbst wurde in Kapitel 5 erläutert. Ist der Persistenzmanager die einzige transaktionale Ressource, kann das Transaktionsmanager-Interface auf den Datenbanktransaktionen implementiert werden. Kontrolliert ein 2-Phasen-Commit-fähiger Transaktionsmanager die verschiedenen Ressourcemanager, so spricht die Anwendungssteuerung direkt mit diesem Transaktionsmanager (vgl. linke Seite der Abbildung 8). Steht kein 2-Phasen-Commit zur Verfügung, kann diese Logik durch Adapter vor jeder Ressource emuliert werden, beispielsweise durch eigene Protokolle und Statusverfolgung (vgl. rechte Seite der Abbildung 8).

Die Aufteilung der Schnittstellen in Transaktions- und Persistenzmanager macht immer Sinn. Dazu zwei Anmerkungen:

a) In einfachen Situationen (wenn z.B. nur ein transaktionaler Dienst vorhanden ist), kann man beide Schnittstellen in einer einzigen Komponente implementieren.
b) Wenn kein 2-Phasen-Commit zur Verfügung steht (oder wenn sich die Nutzung verbietet), kann man die in Abbildung 9 gezeigte Architektur um einen angenäherten 2-Phasen-Commit-Nachbau erweitern. Damit meinen wir einfache Verfahren, die im Fehlerfall auch manuelle Eingriffe erfordern können.

Abbildung 8:
Der Persistenzmanager als Ressourcemanager



Der Persistenzmanager besteht aus den Subkomponenten *Workspace, QueryManager*, *RelationshipManager* und *ConnectionManager* (vgl. Abbildung 9). Zusätzlich gibt es den *ModelManager,* bei denen Transformationsregeln für die Überführung von Objekten in Relationen und zurück hinterlegt sind.

Abbildung 9 zeigt die Rolle des Persistenzmanagers als Mittler zwischen der Welt der Anwendung und der Datenbank. In der Anwendung kennen wir die A-Begriffe: A-Fall, A-Komponente, A-Entitätstyp, A-Entität, A-Datentyp (der Attribute von A-Entitätstypen beschreibt) und A-Transaktionen. Bei relationalen Datenbanken kennen wir Tabellen, Spalten und Zeilen, die SQL-Datentypen und die Transaktionen der Datenbank.

Die Anwendung kann in jeder objektorientierten Programmiersprache implementiert sein; wir betrachten Java als wichtiges Beispiel. A-Entitätstypen sind in der Regel persistent. Abschnitt 6.3.3 erklärt die Kommunikation zwischen den persistenten A-Entitäten und dem Persistenzmanager. Dabei ist zu beachten, dass persistente A-Entitätstypen sowohl persistente als auch transiente Attribute haben können (etwa bei berechneten Attributen).

Die Subkomponenten in Abbildung 9 haben folgende Aufgaben:

- *Workspace:* Die Workspaces trennen die verschiedenen Transaktionen: Jede Transaktion läuft in genau einem Workspace (= Transaktionszustand, vgl. Abbildung 10). In Kapitel 5 wurden zwei Mechanismen erläutert, um dem Persistenzmanager die aktuelle Transaktion mitzuteilen: via Parameter und über den Thread-Kontext.
- *QueryManager:* Der QueryManager gestattet es, Abfragen zu formulieren und auszuführen.
- *RelationshipManager:* Der RelationshipManager verwaltet die Beziehungen konsistent. Er löscht z.B. abhängige Objekte bei der Auflösung von Beziehungen.
- *ConnectionManager:* Der ConnectionManager stellt die Verbindung zu einer oder mehreren Datenquellen her. Er ist für das Connection Pooling zuständig.
- *ModelManager:* Der ModelManager verwaltet die Transformationsregeln zur Überführung von Objekten in Relationen und zurück. Der ModelManager ist nicht Bestandteil des Persistenzmanagers, sondern wird von diesem benutzt.

Abbildung 9:
Aufbau des Persistenzmanagers

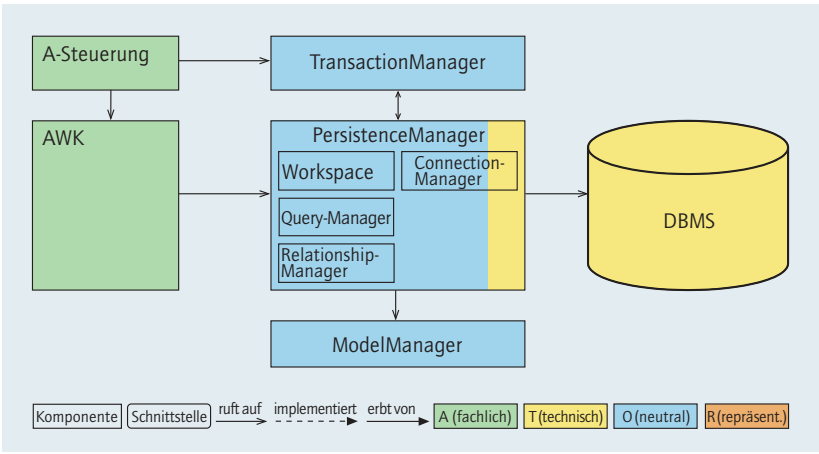
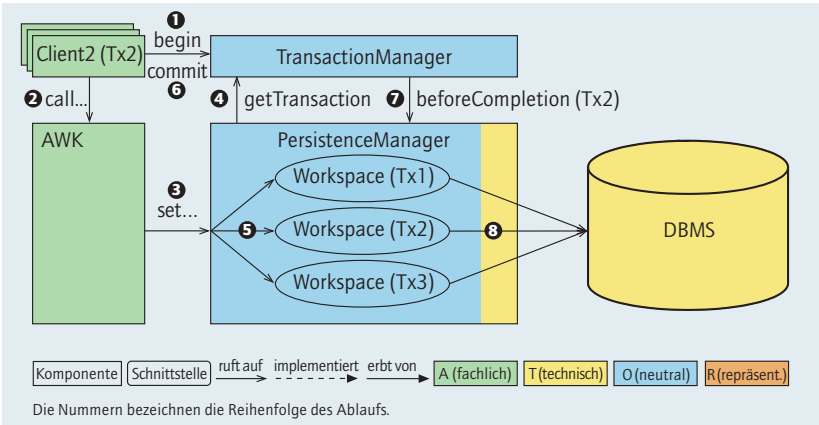


Abbildung 10:
Zuordnung von Transaktion und Workspace

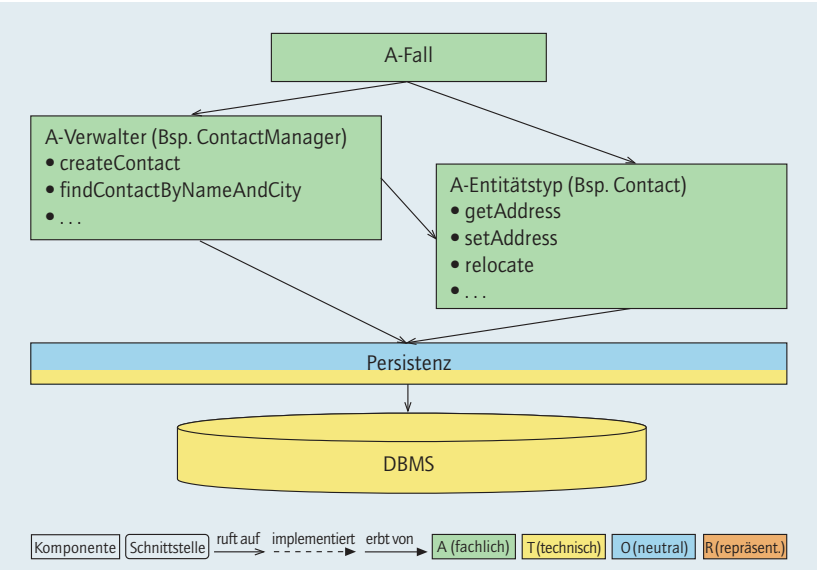


Die Nummern bezeichnen die Reihenfolge des Ablaufs.

6.3 Außensicht des Persistenzmanagers

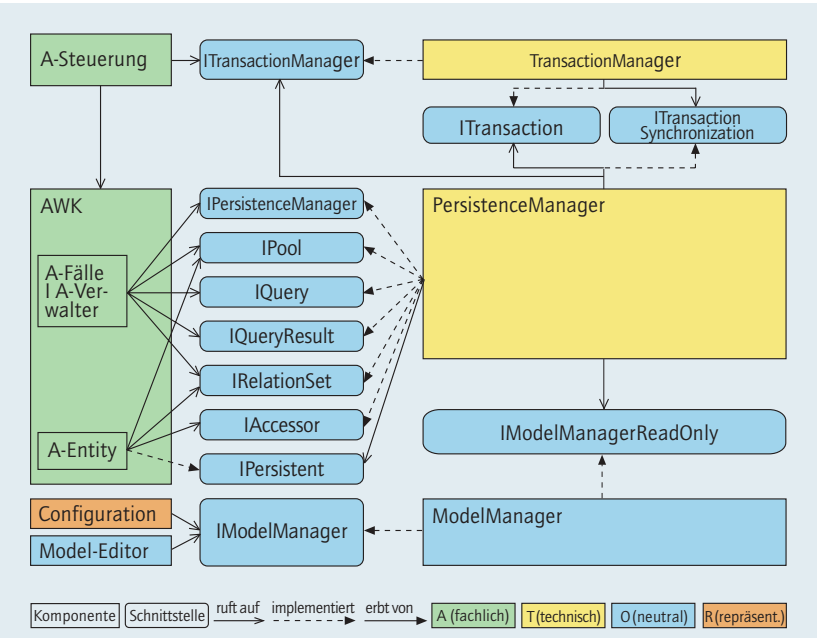
Abbildung 11 zeigt persistenz-relevante Operationen des Anwendungskerns: Erzeugen und Suchen beim A-Verwalter, Änderungsoperationen beim A-Entitätstyp.

Abbildung 11: Persistenz-relevante Operationen



Diese Operationen sprechen jeweils verschiedene Komponenten oder Hilfsobjekte des Persistenzmanagers an. Hilfsobjekte des Persistenzmanagers sind z.B. Abfrageobjekte (Interface *IQuery*) oder Behälter für Beziehungen (Interface *IRelationSet*).

Abbildung 12: Der Persistenzmanager und seine Umgebung



Hier ein Überblick der verschiedenen Schnittstellen (vgl. Abbildung 12; die Transaktionsschnittstellen sind in Kapitel 5 beschrieben):

- *IPersistenceManager* ist eine abstrakte Fabrik, die Pools produziert (vgl. Abschnitt 6.3.1).
- *IPool* dient der Verwaltung (Einfügen, Löschen, Abfragen) persistenter Objekte. Jedes persistente Objekt gehört zu genau einem Pool. Ein Pool repräsentiert ein oder mehrere Objektmodelle bzw. Objektwelten, die in einem Datenbankschema verwaltet werden.
- *IQuery* verkörpert Abfragen, die beliebig oft mit verschiedenen Argumenten ausgeführt werden. Abfragen dienen zum Lesen und Filtern persistenter Objekte. Jede Abfrage läuft über genau einen Pool.
- *IQueryResult* ist ein um die *close*-Methode erweiterter *ListIterator (java.util.ListIterator)*. Mit Hilfe von *IQueryResult* läuft man über die Ergebnisse einer ausgeführten Abfrage.
- *IRelationSet* ist eine Erweiterung von *Set (java.util.Set)*. Sie beschreibt mehrwertige Beziehungen und bietet gegenüber dem *Set* zusätzlich die Möglichkeit, die in der Beziehung enthaltenen Objekte zu sperren.
- *IAccessor* (Zugriffsobjekt) ist ein Funktionsobjekt zum Lesen und Schreiben von Eigenschaften persistenter Objekte. Die A-Entitätsklasse speichert für jedes Attribut ein solches Zugriffsobjekt und beschleunigt so die wiederholte Ausführung der *get*- und *set*-Methoden.
- *IPersistent* ist ein Callback-Interface, das die A-Entitätsklassen implementieren. Dies versetzt den Persistenzmanager in die Lage, die benötigten Informationen in den persistenten Entitäten abzulegen.
- *IModelManager* dient dazu, die Abbildung des Klassenmodells auf die Datenbank zu definieren (vgl. Abschnitt 6.4). Dies geschieht im Rahmen der Konfiguration mit Hilfe des *Model-Editors*. Der *ModelManager* besitzt eine Read-Only Schnittstelle, die der Persistenzmanager nutzt. Die *ModelManager*-Schnittstellen werden in diesem Papier nicht weiter ausgeführt.

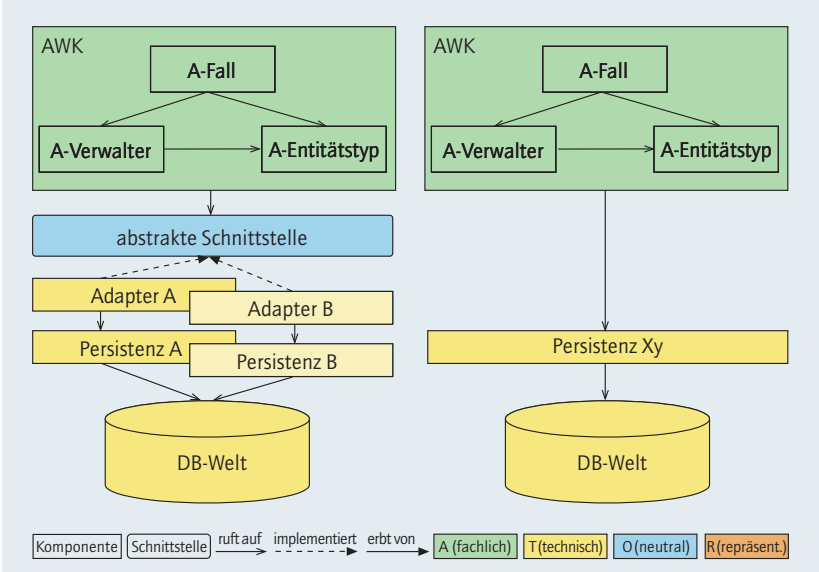
Der Anhang enthält alle Schnittstellen außer *IModelManager* in Java-Notation. Die Quasar-Schnittstelle für Persistenz minimiert die Abhängigkeiten zwischen der Anwendung und dem Persistenzmanager. Man kann sie auf zwei Arten implementieren:

a) Selbst im Projekt. Davon raten wir ab, denn die Implementierung eines Persistenzmanagers ist sehr komplex.
b) Durch einen Adapter, der auf einem am Markt erhältlichen Produkt oder bei Java auf Quasar aufsetzt. Der Aufwand für einen solchen Adapter ist gering.

Abbildung 13:
Quasar-Schnittstelle

Die Quasar-Schnittstelle entspricht im Wesentlichen der Schnittstelle gängiger Persistenzmanager bzw. O/R-Mapper. Anwendungslogik kann auf Basis dieser Schnittstelle weitgehend unabhängig von einem konkreten Persistenzmanager entwickelt und somit auf unterschiedlichen Persistenzmanagern eingesetzt werden (Abbildung 13).

Ist die Entscheidung für einen bestimmten Persistenzmanager gefallen, so wird darauf die Quasar-Schnittstelle implementiert. Sie wurde im Hinblick auf Implementierungsaufwand und Performance optimiert. Bei einem Umstieg auf einen anderen Persistenzmanager sind nur der Persistenzadapter und die Query-Adapter (R-Code, s. Abschnitt 6.3.2) neu zu implementieren. Die eigentliche Anwendungslogik bleibt weitgehend unverändert.



6.3.1 Pools

Pools konfigurieren und beschaffen

Aus Anwendungssicht bildet die Gesamtheit ihrer persistenten Entitäten eine oder mehrere Welten zusammengehöriger Objekte. Jede dieser Objektwelten ist ein Pool. Hinter einem Pool steht ein Speichermedium, in der Regel eine Datenbank. Die Anwendung sieht die Speichermedien also nicht direkt, sondern nur über Pools, von denen der Persistenzmanager mehrere gleichzeitig verwalten kann.

Jeder Pool verwaltet ein Klassenmodell, und er verfügt über ein Speichermedium, z.B. ein Datenbankschema. Bei mehreren Datenbanken kann jede Datenbank als einzelner Pool angesprochen werden, oder ein Pool kann mehrere Datenbanken kapseln. Die Erzeugung eines Pools ist Bestandteil der Konfiguration und nicht Teil der operativen Schnittstelle des Persistenzmanagers. Jeder Pool wird mit Hilfe einer Registrierungsmethode unter einem sprechenden Namen beim Persistenzmanager angemeldet. Die Registrierung enthält unter anderem die zum Öffnen der Datenbank nötigen Angaben (in Java: Datenbank-URL, -Login, -treiber).

Pools holt man sich beim Persistenzmanager über den Namen:

```
public interface IPersistenceManager {
    IPool getPool (String name);
}
```

Isolationsstufen

Jedes Speichermedium implementiert eine oder mehrere der folgenden Isolationsstufen des SQL-92 Standard: *READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ* und *SERIALIZABLE.*

Diese bedeuten folgendes:

READ_UNCOMMITTED:
Die Transaktion sieht die unbestätigten Änderungen anderer Transaktionen. Dieser Modus wird – wenn überhaupt – nur bei zeitkritischen Batches verwendet, die den Datenbestand exklusiv bearbeiten.

READ_COMMITTED:
Die Transaktion sieht nur die bestätigten Änderungen anderer Transaktionen. *READ_COMMITTED* ist der Regelfall.

REPEATABLE_READ(RR):
Die Transaktion sieht nur diejenigen Änderungen, die vor ihrem eigenen Beginn bestätigt worden sind. Alle Leseoperationen innerhalb einer *RR*-Transaktion liefern also dasselbe Ergebnis. *RR* kostet Performance. Einzige Anwendung: Auswertungsfunktionen, die parallel zu anderen Verarbeitungen laufen und eine konsistente Sicht der Daten benötigen.

SERIALIZABLE (oder *FULL_ISOLATION*):
RR plus die Verhinderung von Phantom Read. Wird in der Praxis aus Performancegründen kaum verwendet.

Die von der Implementierung abhängige Isolationsstufe kann man über den Pool abfragen:

```
interface IPool {
    ...
    IsolationLevel getIsolationLevel();
}
```

Persistente Entitäten erzeugen
Persistente Entitäten erzeugt man mit Hilfe des Pools, der dazu geeignete *make*-Methoden bereitstellt. Der Persistenzmanager entscheidet also, welche Implementierung des A-Entitätstyps tatsächlich verwendet wird.

```
interface IPool {
    ...
    IPersistent make(Class entityType);
    IPersistent make(Class entityType,
                     Object[] primaryKey)
        throws PrimaryKeyException;
}
```

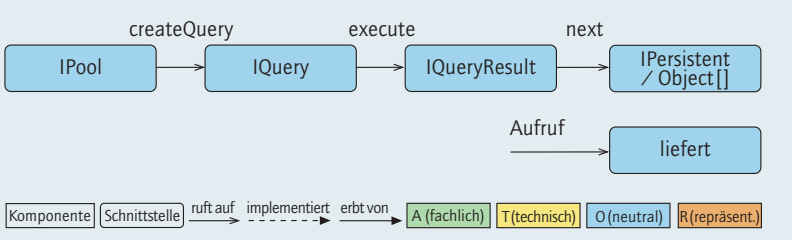
Die erste *make*-Methode setzt voraus, dass ein Primärschlüssel-Generator für den *entityType* bekannt ist. Der zweite Aufruf bietet sich an, wenn für den *entityType* Primärschlüssel individuell vergeben werden. Der Pool erzeugt grundsätzlich persistente Entitäten. Die Anwendung kann weitere Vorkehrungen treffen für den Fall, dass ein und derselbe Entitätstyp sowohl persistente als auch transiente Entitäten besitzt.

Persistente Entitäten löschen
Der Pool erzeugt persistente Entitäten, also ist er auch für das Löschen zuständig:

```
interface IPool {
    ...
    void remove(IPersistent entity);
}
```

Die *remove*-Methode entfernt die übergebene *entity* aus der persistenten Welt, aber die *entity* bleibt als Objekt bestehen. Es ist Sache der Anwendung, die *entity* auch als Objekt zu löschen (soweit die Programmiersprache das zulässt) oder die *entity* als transientes Objekt noch eine Weile am Leben zu lassen. Bei transienten Objekten sind transiente Implementierungen der *get*- und *set*-Methoden erforderlich.

Abbildung 14:
Ablauf von Abfragen



6.3.2 Abfragen

Die einfachste Suche ist die Suche über den Primärschlüssel. Dafür bietet der Pool eine geeignete Methode:

```
public interface IPool {
    ...
    IPersistent lookup(Class entityType,
                       Object[] primaryKey);
}
```

Aber im Allgemeinen ist die Sache komplizierter: Die Anwendung liest und filtert Entitäten mit Hilfe von Abfragen (Queries). Jede Abfrage läuft über genau einen Pool. Abfrageergebnisse können sein:

- Entitäten eines bestimmten Typs, die die in der Abfrage spezifizierte Bedingung erfüllen,
- Tupel von Attributwerten, die aus Attributen verschiedener Entitätstypen zusammengestellt sind,
- Tupel mit Anzahl, Summenwerten oder anderen Datenbankfunktionen,
- Tupel, die Entitäten als Elemente enthalten, oder
- Tupel, die Mengen von Entitäten enthalten, die aus Unterabfragen oder mehrwertigen Beziehungen resultieren.

Jede Abfrage wird einmal formuliert und ist dann beliebig oft mit verschiedenen Parametern ausführbar. Das Selektionskriterium ist in der Abfrage hinterlegt. Ergebnis der Ausführung ist ein Behälter mit der Ergebnisliste, den man mit einem *IQueryResult* (Erweiterung von *ListIterator* in *java.util)* ausliest. Abbildung 14 verdeutlicht den Ablauf aus Sicht der Anwendung.

Formulierung von Abfragen
Wir formulieren das Selektionskriterium und den Ergebnistyp mit Hilfe von Entitätstypen und deren Eigenschaften. Der Ergebnistyp ist entweder eine Entität oder ein Tupel. Tupel sind Arrays von Objekten und enthalten Variable beliebigen Typs: einfache Datentypen, zusammengesetzte Datentypen, Entitäten oder Behälter.

Bei der Ausführung der Abfrage wird deren syntaktische Korrektheit auf Basis der im Persistenzmanager vorhandenen Informationen geprüft. Der Pool ist unter anderem eine Fabrik für Abfragen:

```
public interface IPool {
    ...
    IQuery createQuery(Object querySpecification);
}
```

Mit der *execute*-Methode lässt sich eine Abfrage beliebig oft mit verschiedenen Argumenten ausführen. Wie bei SQL-Schnittstellen üblich werden die Argumente der Reihe nach den in der Abfrage spezifizierten Parametern zugeordnet. Die einzelnen Argumente können Attributwerte oder Entitäten sein.

```
public interface IQuery {
    IQueryResult execute(Object[] queryArguments);
}
```

Der Pool macht aus einer Abfrageformulierung *(querySpecification)* ein Abfrageobjekt. Die Abfragesprache hängt von der Implementierung ab: So unterstützen nur wenige Persistenzmanager Unterabfragen oder die Gruppierung von Tupeln. Als Abfragesprache kommen z. B. OQL oder funktionale Ausdrücke in der Programmiersprache in Betracht.

Die Formulierung der Anfrage ist notwendigerweise abhängig vom eingesetzten Persistenzmanager. Deswegen empfehlen wir, all diese Formulierungen anwendungsseitig in speziellen Query-Adaptern (R-Software) zu konzentrieren (vgl. Abbildung 15), die den eingesetzten Persistenzmanager direkt verwenden. Das Query-Adapter-Interface liefert ausführbare Abfrageobjekte *(IQuery)*.

Jede Abfrage findet innerhalb einer Transaktion statt, denn sie beeinflusst Änderungsoperationen. Gute (leider nur wenige) Persistenzmanager berücksichtigen bei der Ausführung von Abfragen den aktuellen, unbestätigten Zustand der Objekte: *What you say is what you get.* Das heißt: Die Suche nach roten Autos liefert alle Autos, die immer rot waren und es noch sind, alle roten Autos, deren Farbe innerhalb der laufenden Transaktion auf rot geändert wurde und alle roten Autos, die innerhalb der laufenden Transaktion angelegt wurden. Sie liefert NICHT die ursprünglich roten Autos, deren Farbe innerhalb der laufenden Transaktion von *rot* auf eine andere Farbe geändert wurde oder die gelöscht wurden.

Polymorphes Lesen

Jede Abfrage nimmt Bezug auf einen Entitätstyp und alle seine Spezialisierungen. Beispiel: Die Suche nach bestimmten Kunden liefert alle Privat- und alle Firmenkunden, die das Selektionskriterium erfüllen. Das Suchergebnis ist also im allgemeinen polymorph (vielgestaltig).

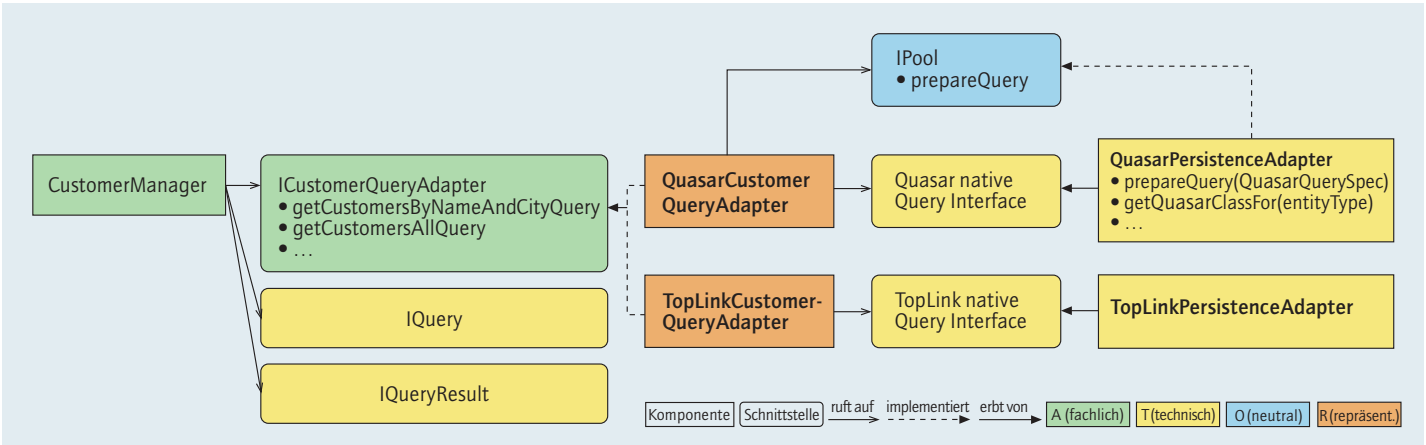
Suchergebnis als Iterator

Suchergebnisse sind beliebig lang, haben eine Reihenfolge (zufällig oder sortiert) und sind nicht änderbar. Deswegen verstecken wir sie hinter einem *ListIterator* (in *java.util*). Die Implementierung dieses Iterators im Persistenzmanager versteckt die Ladestrategie: Alles auf einmal lesen, jedes Objekt einzeln lesen (vertikales *lazy read*) oder eine Mischform. Die Iteratorimplementierung versteckt ferner Beschaffung und Freigabe von Datenbank-Ressourcen (insbesondere des Datenbank-Cursors). Die *close*-Methode schließt den Iterator explizit und macht ihn damit unbrauchbar.

```
public interface IQueryResult extends ListIterator {
    void close();
}
```

Die *next*-Methode des Iterators liefert je nach dem definierten Ergebnistyp A-Entitäten oder Tupel.

Abbildung 15:
Abfrageformulierungen in Adaptern



6.3.3 Persistente Entitätstypen

Attribute von Entitätstypen werden als Instanzvariable implementiert. Der Zugriff darauf erfolgt mit den üblichen *get*- und *set*-Methoden *(setAdresse, getAdresse)* oder durch fachliche Methoden *(einzahlen, abheben,* aber nicht *setSaldo).*

Zu jeder einwertigen Beziehung *(Konto zu Inhaber)* gibt es eine *get*-Methode *(getInhaber)* und – falls sinnvoll – eine *set*-Methode *(setInhaber).* Für jede mehrwertige Beziehung *(Bestellung zu Bestellpositionen)* schreibt man eine *get*-Methode, die einen Behälter liefert, der je nach Situation änderbar ist oder auch nicht *(getBestellpositionen).*

Wie verträgt sich all das mit dem Persistenzmanager? Hier die Probleme:

a) Wie soll der Persistenzmanager die aus der Datenbank gelesenen Daten der Anwendung zur Verfügung stellen, so dass die genannten *get/set*-Methoden funktionieren?
b) Wie erfährt der Persistenzmanager Änderungen in der Objektwelt (Erzeugen/Löschen eines Objekts, Änderungen an einem vorhandenen Objekt), so dass er sie an die Datenbank weiterleiten kann?

Es gibt zwei Mechanismen für die Kommunikation zwischen Anwendung und Persistenzmanager: *push* und *pull* .

Push

Der Persistenzmanager liest Sätze aus der Datenbank, erzeugt A-Entitäten und schiebt die Informationen aus der Datenbank in diese Objekte hinein *(push).* Umgekehrt meldet die Anwendung dem Persistenzmanager alle Änderungen der Objektwelt. Beim Push-Ansatz schauen die Entitätsklassen im Prinzip genauso aus wie im transienten Fall. Allerdings kennen sie den Pool, dem sie angehören, und melden Änderungen mit der Methode *update* dorthin.
Alle Eigenschaften stehen wie gewohnt als Instanzvariable in der Klasse (Beziehungen als änderbarer oder nicht änderbarer *Set*). Der Persistenzmanager füllt diese Variablen bei der Erzeugung der Klasse über eine geeignete Schnittstelle. In Java bietet sich *Reflection* an; allerdings sind ohne Maßnahmen am *Security Manager* nur öffentliche Variable zugänglich. Eine Alternative, die in jeder modernen Sprache funktioniert, ist das Interface *IInspectable,* das die Entitätstypen implementieren:

```
public interface IInspectable extends IPersistent {
    Object getProperty(Object property);
    void setProperty(Object property, Object value)
        throws ValueException;
}
```

Property identifiziert die Eigenschaft. Die *ValueException* wird geworfen, wenn der übergebene Wert *(value)* ungültig ist, oder wenn das angesprochene Objekt durch diesen Wert in einen ungültigen Zustand gerät. Die Anwendung informiert den Persistenzmanager über Änderungen mit Hilfe der *update*-Methode:

```
interface IPool {
    ...
    void update(IPersistent entity);
}
```

Update wird nach der Bearbeitung eines Entitätsobjekts, spätestens vor Ende der Transaktion aufgerufen. Der Persistenzmanager liest dann die aktuellen Eigenschaftswerte aus und ermittelt die Änderungen durch Vergleich. Dieser Vorgang ist jedoch nicht performant, wenn viele Eigenschaften oder mehrwertige Beziehungen zu vergleichen sind. Insbesondere hat der Persistenzmanager weniger Kontrolle über Änderungsoperationen.

Beziehungen sind als *Set* implementiert. Die typische Implementierung dieses *Set* (etwa die Bestellpositionen zu einer Bestellung) verwendet eine Abfrage *(IQueryResult),* so dass die referenzierten Objekte erst zum gewünschten Zeitpunkt geladen werden.

Pull

Der Persistenzmanager verwaltet alle Daten; die Entitätsklassen sind nur noch Hüllen, die jedes Attribut einzeln vom Persistenzmanager abholen *(pull).* Jede Änderung läuft über den Persistenzmanager, der auf diese Weise immer auf dem neuesten Stand ist. Er hat zudem die Möglichkeit, jeden Zugriff auf Berechtigung oder Validierung zu prüfen. Außerdem kann er Daten laden wann er will *(lazy* oder *eager),* und er kann referenzierte Objekte früh oder spät erzeugen. Die Entitätsklassen sehen bei der pull-Strategie anders aus als sonst: Jeder Zugriff auf eine Eigenschaft läuft über ein Funktionsobjekt, den *Accessor:*

```
interface IAccessor {
    Object getValue(IPersistent entity);
    void setValue(IPersistent entity, Object value)
        throws ValueException;
}
```

Für jeden Entitätstyp und jede Eigenschaft gibt es ein eigenes Funktionsobjekt. Die Anwendung ermittelt sie einmalig vor dem ersten Zugriff auf eine Eigenschaft beim Pool, wo die Funktionsobjekte bei der Konfiguration registriert wurden.

```
public interface IPool {
    …
    IAccessor getAccessor(Class entityType,
                          Object property);
    IAccessor getAccessor(IPersistent entity,
                          Object property);
}
```

Get- und *set*-Methoden sehen damit etwa folgendermaßen aus:

```
public class Contact implements IPersistent {
    // vgl. interface IPersistence
    private static IPool pool;
    private static IAccessor homeAddressAccessor =
        pool.getAccessor(Contact.class,
                         "homeAddress");
    public Address getHomeAddress() {
        return (Address)homeAddressAccessor
              .get Value(this);
    }
    public void setHomeAddress
    (Address homeAddress) {
        homeAddressAccessor
              .setValue(this, homeAddress);
    }
}
```

Bei dieser Implementierung belastet die Ermittlung der Zugriffsobjekte das System nur beim Anlaufen (static-Deklaration von *homeAddress-Accessor*). Alle Entscheidungen, die nur von Entitätstyp und Eigenschaft, also nicht von der konkreten Entität abhängen, werden im Zugriffsobjekt fest eingestellt und belasten das System nicht mehr als nötig. In beiden Fällen (*push* und *pull*) speichert der Persistenzmanager spätestens bei Ende der Transaktion den letzten Stand der geänderten Objekte. Der Zugriff über Funktionsobjekte *(Accessors)* macht es möglich, für Testzwecke mit sehr einfachen Pool-Implementierungen zu arbeiten. Im einfachsten Fall hält der Pool für jedes Attribut jeder Entität den Wert in einem Verzeichnis (z.B. in einer *HashMap*).

Beziehungen

Als Behälter für mehrwertige Beziehungen verwenden wir den *Set.* Die Anwendung ändert mehrwertige Beziehungen, indem sie *Set*-Elemente hinzufügt oder entfernt. Das Setzen einer mehrwertigen Beziehung wird vom Persistenzmanager in der Regel nicht unterstützt, kann aber in der Anwendung implementiert werden. *IRelationSet* bietet zusätzlich die Möglichkeit, die Elemente während des Durchlaufens zu sperren (vgl. Abschnitt 6.3.4).

```
interface IRelationSet extends Set {
    Iterator iterator(LockMode lockMode);
    Iterator iterator(LockMode lockMode,
                      int timeout);
}
```

Den Umgang mit einer mehrwertigen Beziehung zeigt folgendes Beispiel:

```
public class Address implements IPersistent {
    // vgl. interface IPersistence
    private static IPool pool;
    private static IAccessor residentsAccessor =
        pool.getAccessor(Address.class, "residents");
    …
    public Set getResidents() {
        return (Set)residentsAccessor.getValue(this);
    }
    // keine setResidents-Methode;
}
```

address.getResidents() liefert alle Personen, die an einer bestimmten Adresse wohnen. Der als Ergebnis gelieferte *Set* lässt sich mit den üblichen Methoden *(add, remove, iterator)* bearbeiten.

Callback-Interface IPersistent

Jede persistente Entität kennt ihren Primärschlüssel und den Pool, zu dem sie gehört. Über das *IPersistent*-Interface lassen sich diese Informationen abfragen:

```
public interface IPersistent {
    IPool getPool();
    Object[] getPrimaryKey();
    void setPool(IPool pool);
    void setPrimaryKey(Object[] primaryKey);
}
```

Zum Erzeugen von persistenten Entitäten braucht der Persistenzmanager einen parameterlosen Konstruktor und die beiden *set*-Methoden von *IPersistence.* Kein anderer darf sie nutzen. Mit einer abstrakten Oberklasse für Entitätstypen *(Abstract-Persistent)*, die dieses Interface implementiert, kann der Persistenzadapter die Implementierung dieses Interface bereits vorgeben.

6.3.4 Sperren

Die Sperrstrategie legt fest, wie verschiedene A-Transaktionen kooperieren. Wir unterscheiden optimistisches und pessimistisches Sperren.

Beim optimistischen Sperren liest man zu Beginn der A-Transaktion ohne Datenbanksperren. Am Ende der A-Transaktion werden die Daten nochmal gelesen; durch Vergleich von Zeitstempel, Versionszähler oder schlicht der Daten selbst wird auf Kollision mit einer anderen A-Transaktion geprüft. Im Fall einer Kollision wird die A-Transaktion zurückgesetzt, im anderen Fall speichert man sämtliche Änderungen innerhalb einer kurzen T-Transaktion. Der Vorteil der optimistischen Strategie ist die gute Nutzung der Ressourcen, denn zu jedem Zeitpunkt gibt es nur wenige gleichzeitig aktive T-Transaktionen, selbst wenn die A-Transaktionen lang dauern. Der Nachteil ist natürlich die späte Kollisionserkennung, die nicht immer zumutbar ist. Beispiel: System zur Erfassung komplexer Aufträge.

Bei der pessimistischen Strategie sperrt man die gelesenen Objekte für den weiteren Zugriff anderer Benutzer. Dies verhindert Kollisionen; allerdings kann es sein, dass diese Sperren andere Transaktionen behindern. Die pessimistische Strategie belastet die Systemressourcen wesentlich stärker. Deshalb definiert man den Sperrmodus auf der Ebene von Abfragen oder sogar von einzelnen Objekten: Man kann etwa 100 Objekte zunächst ohne Sperre lesen und dann im weiteren Verlauf einige davon gezielt sperren.

Wir unterscheiden also die *Sperrstrategie* (*OPTIMISTIC* oder *PESSIMISTIC*) und den *Sperrmodus* mit den Ausprägungen *NOLOCK, REFERENCE, ACCESS, MODIFY, DELETE.*

Abbildung 16 zeigt, welche Operationen die einzelnen Sperrmodi in anderen Transaktionen blockieren.

- Bei *NOLOCK* stehen anderen Transaktionen alle Operationen offen.
- *REFERENCE* dient zum Verweisen auf ein Objekt und verhindert, dass andere Transaktionen das Objekt löschen.
- *ACCESS* dient zum Verarbeiten des Objekts ohne es zu verändern. *ACCESS* verhindert, dass andere Transaktionen das Objekt ändern.
- *MODIFY* dient zum Ändern des Objekts. *MODIFY* verhindert, dass andere Transaktionen das Objekt lesen (ACCESS).
- *DELETE* dient zum Löschen des Objekts und verhindert, dass andere Transaktionen das Objekt referenzieren (REFERENCE).

Die *createQuery*-Methode mit dem zusätzlichen Argument *Lockmode* erzeugt eine Abfrage, in der alle gelesenen Objekte gemäß *mode* gesperrt sind; *mode* = *NOLOCK* bewirkt Lesen ohne Sperre.

```
public interface IPool {
    ...
    LockStrategy getLockStrategy();
    int getLockTimeout();

    IQuery createQuery(Object queryDescription,
                       LockMode mode);
    IQuery createQuery(Object queryDescription,
                       LockMode mode,
                       int timeoutMs);

    void lock(IPersistent entity, LockMode mode);
    void lock(IPersistent entity, LockMode mode,
              int timeoutMs);
}
```

Das Interface *IRelationSet* besitzt eine *iterator*-Methode, die es gestattet, den Sperrmodus und bei Bedarf den *TimeOut* anzugeben (vgl. Abschnitt 6.3.3).

	Referenzieren	Lesen	Ändern	Löschen
REFERENCE				
ACCESS				
MODIFY				
DELETE				
NOLOCK				

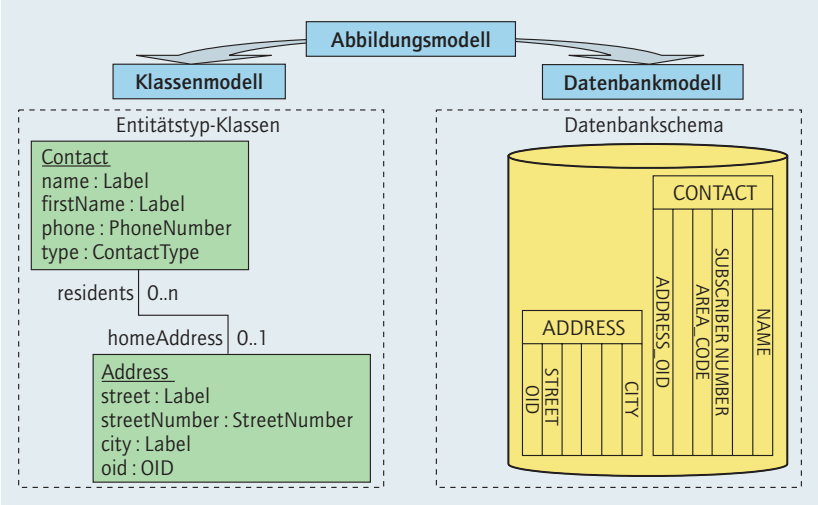
Abbildung 16: Wechselwirkungen der Sperrmodi

6.4 Außensicht des Modell-Managers

Der Modell-Manager verwaltet alle Informationen für die Transformationen von Objekten in Tabellen und Spalten der Datenbank und zurück. Nutzer dieser Information ist der Persistenzmanager. Man braucht drei Modelle (Abbildung 17):

- Das **Klassenmodell** beschreibt die Klassen mit Attributen, Beziehungen und Vererbung. Diese Information steht nur zum Teil in der Java Reflection, und deshalb beschreiben wir das Klassenmodell mit eigenen Mitteln.
- Das **Datenbankmodell** (oder Datenbankschema) beschreibt die Tabellen und Spalten der Datenbank.
- Ein **Abbildungsmodell** beschreibt, wie die Objekte und ihre Attribute auf Tabellen und Spalten abgebildet werden.

Abbildung 17:
Modelle im Überblick



Das Klassenmodell kann etwa so aussehen:

Vier Deskriptoren beschreiben das Klassenmodell: *TypeDescriptor, AttributeDescriptor, AttributeTypeDescriptor* und *RelationshipDescriptor* (Abbildung 18).

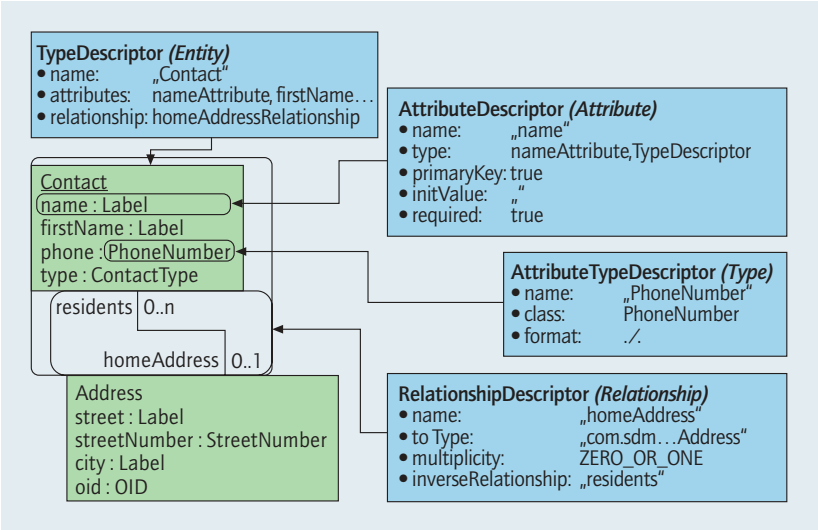
Jeder *TypeDescriptor* beschreibt eine Klasse durch den Namen, die Oberklasse und alle Attribute und Beziehungen.

Jeder *AttributeDescriptor* beschreibt ein Attribut durch den Namen, den Datentyp und ein Kennzeichen, ob das Attribut Teil des Primärschlüssels ist.

Jeder *AttributeTypeDescriptor* beschreibt, wie ein A-Datentyp in einer oder mehreren Stufen auf datenbank-nahe Datentypen abgebildet wird. Datenbank-nah sind z.B. String, Integer, BigDecimal.

Jeder *RelationshipDescriptor* beschreibt eine Beziehung durch die beteiligten Klassen und die Kardinalität.

Abbildung 18:
Beschreibung eines Klassenmodells



Ein *SimpleBaseTypeMapping* ordnet einem Entitätstyp eine Tabelle zu. Ein *SimpleSubTypeMapping* würde Attribute von abgeleiteten Klassen auf andere Tabellen abbilden und die Feststellung des konkreten Typs ermöglichen.

Ein *AttributMapping* bildet ein einfaches Attribut auf eine Spalte ab; ein *SimpleStructureAttributeMapping* bildet ein zusammengesetztes Attribut auf mehrere Spalten ab.

Ein *SimpleRelationshipMapping* verbindet die Fremdschlüsselspalte einer Beziehung mit der referenzierten Primärschlüsselspalte. Ein *SimpleBridgeRelationshipMapping* verbindet bei m:n-Beziehungen zwei Tabellen über eine Brückentabelle.

Denormalisierte oder sonstige vom Standard abweichende Abbildungen lassen sich durch zusätzliche *Mappings* realisieren.

Der Modellmanager erhält alle Modellinformationen zum Konfigurationszeitpunkt, beispielsweise in Form von XML-Dateien.

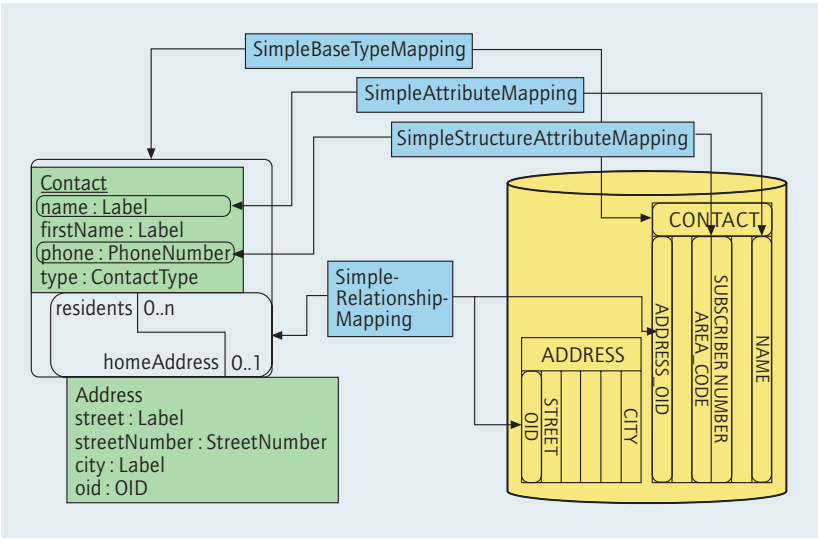
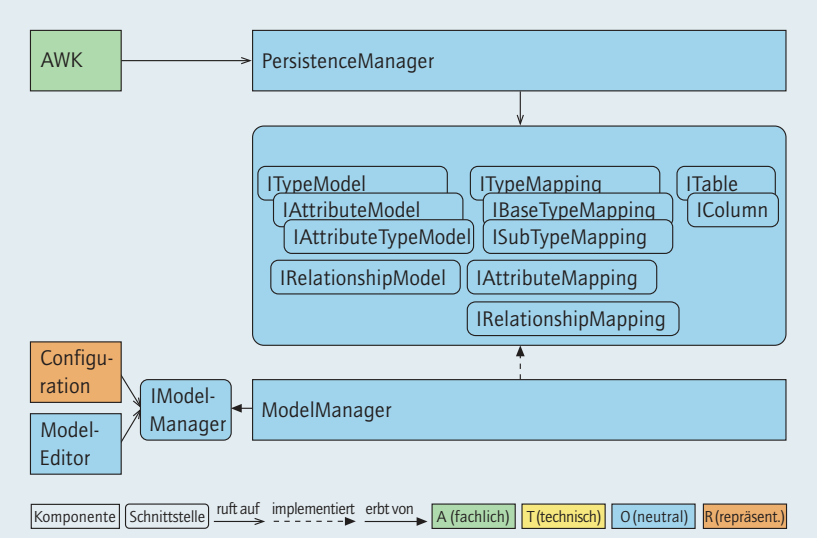


Abbildung 19:
Abbildungsmodell

Abbildung 20:
Außensicht
Modellmanager



6.5 Objektrelationale Abbildung

Nachdem die Außensicht des Persistenzmanagers und damit seine Leistungen umrissen sind, erläutert dieser Abschnitt, welche Aufgaben ein Persistenzmanager für relationale Datenbanken intern erfüllt:

- Kapselung der technischen Schnittstelle der Datenbank durch eine 0-Schnittstelle,
- Überwindung des objekt-relationalen Paradigmenbruchs, d.h. Umsetzung der Objekte und Operationen der Anwendung auf Tabellen und Abfragesprache der Datenbank sowie
- Konvertierung der Datentypen aus der Anwendung in SQL-Datentypen in der relationalen Datenbank.

In Java kapselt der JDBC-Treiber die verwendete Datenbank nur teilweise. Dem Persistenzmanager obliegen vor allem die letzten beiden Punkte.

Das Problem des O/R-Mappings wird ausführlich in der Literatur diskutiert [Keller 1998] und [Ambler 1998]. Der folgende Abschnitt beschreibt Standardlösungen, die für die Mehrzahl der Projekte tragen.

Jeder A-Entitätstyp wird beschrieben durch seine Oberklasse, seine Attribute, die Beziehungen, an denen er beteiligt ist, und seinen Schlüssel. Jedes Attribut wird durch seinen A-Datentyp gekennzeichnet.

6.5.1 A-Datentypen

Bei der Abbildung der A-Datentypen auf eine relationale Datenbank sind zwei Arten von Datentypen zu unterscheiden: Für Datentypen, deren Werte mit Standardklassen wie *String, Integer, Timestamp* dargestellt werden, kennt der Persistenzmanager Standardverfahren zur Übersetzung zwischen Anwendungs- und Datenbankwelt. Verwendet die Anwendung spezielle Datentypklassen, benötigt der Persistenzmanager über die Angabe der Klasse hinaus Informationen, welche Spalten benötigt und wie die Werte von und zur Anwendung konvertiert werden. Ein einfaches Beispiel sind zusammengesetzte Typen, deren Felder einem Standardtyp entsprechen und die demzufolge auf je eine Spalte abgebildet werden. Durch anwendungsspezifische Mapping- und Konverter-Implementierungen sind hier keine Grenzen gesetzt.

Datentypklassen, die einer Standardklasse nur Validierungsinformation (Bsp.: maximale String-Länge) hinzufügen, können eingespart werden, indem die Validierung an geeigneter Stelle modellbasiert implementiert wird. Die Validierungsinformationen werden dazu deklarativ im Objektmodell hinterlegt. Spezielle Mappings und Konverter sind nicht mehr erforderlich. Nebenbei bleibt durch modellgesteuerte Validierungen der Client frei von Validierungswissen.

6.5.2 A- und T-Schlüssel

A-Entitätstypen haben unterschiedliche Schlüssel:

- Der Schlüssel aus Sicht der Anwendung (A-Schlüssel) identifiziert die Entität nach außen. Dies kann ein einzelnes Attribut sein (z.B. Kundennummer), oder eine Kombination von Attributen.
- Objekte im Laufzeitsystem, die A-Entitäten repräsentieren, werden mit technischen Mitteln identifiziert (*ObjectId,* Speicheradresse). Beides sind T-Schlüssel.
- In der Datenbank verwenden wir Primärschlüssel, die manchmal A-Schlüssel entsprechen, oft aber nicht. Auch der Primärschlüssel ist ein T-Schlüssel.

Der Persistenzmanager gewährleistet die Identität über alle drei Ebenen (Fachlichkeit, Laufzeitsystem, Datenbank). Im einzelnen macht er folgendes:

a) Er verwaltet eine Tabelle aller persistenter Entitäten. Jede persistente Entität ist höchstens einmal aufgeführt. Es gibt also innerhalb von Transaktionsklammern keine redundanten Kopien („der letzte gewinnt").

Tabelle 1

Fachliche Schlüssel	Technische Schlüssel
Nur fachliche Daten, häufig zusammengesetzt aus mehreren Attributen bzw. Spalten.	Künstlich generiert, keine fachliche Bedeutung.
Keine zusätzlichen Primärschlüsselspalten; dafür pro Fremdschlüssel entsprechende Anzahl Spalten.	Eine zusätzliche Primärschlüsselspalte; nur eine Spalte pro Fremdschlüssel.
A- und T-Identität sind identisch.	A- und T-Identität sind verschieden.
Format kann sich mit fachlichen Erweiterungen / Änderungen verändern.	Konstantes Format über gesamten Entwicklungszeitraum.
	Zwingend erforderlich für Abhängige Objekte ohne eigene fachliche Identität.
Verwendung der im AWK generierten eindeutigen fachlichen Schlüssel.	Zusätzliche Erzeugung eindeutiger Technischer Schlüssel (kann im Persistenzmanager versteckt werden).
	Technischer Schlüssel kann mit Zusatzinformationen angereichert werden (z. B. Klassenname – damit kann bei Vererbungssituationen das Unterscheidungsmerkmal für die jeweilige Unterklasse gespart werden).
Schlüssel kann auch in fachlichen Anfragen verwendet werden. Hohe Selektivität, Indexe auf dem A-Schlüssel fachlich nutzbar.	T-Schlüssel nur bei Durchlaufen von Referenzen verwendbar (also bei Joins und Identitäts-Queries). T-Schüssel nicht für fachliche Queries nutzbar. Damit sind immer zusätzliche Indexe erforderlich.

b) Er koordiniert die Zugriffe verschiedener Transaktionen auf persistente Entitäten. Dies kann z.B. so implementiert sein, dass pro Transaktion eine Kopie der beteiligten Entitäten erstellt wird.

Hilfe zur Entscheidung zwischen technischen und fachlichen Schlüsseln liefert Tabelle 1.

6.5.3 Schlüsselerzeugung

Schlüssel werden meist fortlaufend, manchmal lückenlos generiert. Technische Schlüssel sind ein Geheimnis der Zugriffschicht. Für fachliche Schlüssel ist der Anwendungskern verantwortlich. Der Generator sollte in beiden Fällen durch ein Interface gekapselt werden. Während der Konfiguration können die so gekapselten Generatoren dem Persistenzmanager als Callback-Interface übergeben werden, damit dieser beim Anlegen persistenter Objekte ohne Zutun der Anwendung die notwendigen Schlüssel generieren kann.

Mehrere Standardverfahren generieren technische und fachliche Schlüssel:

- Schlüsselerzeugung mit Datenbankmitteln (z.B. SEQUENCE bei Oracle). Das Verfahren trägt eingeschränkt für A- und T-Schlüssel.
- Schlüsselerzeugung über Hilfstabelle (z.B. *com.sdm.quasarx.persistence.util.SimplePKGenerator,* QDI-Nummernserver Implementierung oder [Ambler 1998]). Das Verfahren trägt für beliebige A- und T-Schlüssel.
- Schlüsselerzeugung über UUID: Schlüssel (Universal Unique IDentifier) sind 128 Bit Strings, die entstehen durch Hashing der Identifikationsnummer der Ethernet-Karte sowie des Datums und der Uhrzeit. Damit ist kein Datenbankzugriff notwendig. Voraussetzung ist aber das Vorhandensein einer Netzwerkkarte. Da die Uhrzeit zurückgestellt werden kann, sind Duplikate denkbar, die vermieden werden müssen. UUID trägt nur für T-Schlüssel.

Bei der Wahl des Algorithmus sollten Optimierungsaspekte wie Streuung der Schlüsselwerte (bessere Indizes) sowie Rückgabe von nicht mehr benötigten Werten berücksichtigt werden.

6.5.4 Attribute

Der Persistenzmanager benötigt eine Vorschrift, die ihm sagt, wie die Werte zwischen Datenbank und Anwendung konvertiert werden. Falls die verwendeten Spalten nicht in der Haupttabelle des Entitätstyps liegen, sind außerdem Informationen über die verbindenden Schlüsselspalten notwendig.

Werden die Werte des Attributs durch eine Standardklasse dargestellt, reicht zur Ablage des Attributs eine einzige Spalte. Den passenden Spaltentyp leitet der Persistenzmanager vom Standard-Datentyp ab. Die Konvertierung von Werten zwischen Datenbank und Anwendung ist kein Problem.

Bei Attributen, deren Werte in der Anwendung durch spezielle Klassen dargestellt werden, ist ein geeignetes *AttributMapping* (vgl. Abschnitt 6.4) anzugeben. Dieses liefert die Spalten, in denen die Werte des Attributs gespeichert werden, sowie die Konvertiervorschrift, mit der aus Spaltenwerten Attributwerte werden und umgekehrt.

Ein Sonderfall ist die Komposition von Attributen aus einfacheren Attributen. Hier wird die Abbildung auf Felder und entsprechende Spalten der untergeordneten Attribute heruntergebrochen.

Der Persistenzmanager macht Annahmen über die Implementierung von A-Datentypen, etwa in dieser Form:

Jede A-Datentypklasse leistet folgendes:

a) Sie ist serialisierbar.
b) Sie implementiert *equals* und *hashCode* nach den Java-Regeln.
c) Sie besitzt für jedes Feld eine öffentliche *get*-Methode.
d) Sie besitzt einen Konstruktor, der ein voll verwendbares Exemplar erzeugt (jedes Feld ist gefüllt).

Falls das Datenbankschema frei erstellt werden kann und keine Denormalisierung erforderlich ist, reicht es, die Spalten aller Attribute eines Entitätstyps in einer eigenen Tabelle zusammenzufassen. Soll ein bestehendes Datenbankschema eingebunden werden, so können die Attributwerte derselben Entität auf verschiedene Tabellen verteilt sein. Deshalb sollte, sofern auf ein bestehendes Datenbankschema aufgesetzt wird, darauf geachtet werden, dass der Persistenzmanager das Verteilen der Attribute einer Klasse auf verschiedene Tabellen unterstützt. Alle Tabellen einer Entität müssen über Schlüssel miteinander verbunden sein. In der Regel wird in jeder Tabelle der Primärschlüssel der Entität wiederholt.

Nullwerte in der Datenbank sind ein besonderes Problem. Falls diese von der Anwendung benötigt werden, muss der Konverter entsprechende Vorkehrungen treffen.

Bei der Abbildung von Attributen, deren Typ ein zusammengesetzter A-Datentyp ist, gilt es, die untergeordneten Attribute abzubilden. Einfache Attribute werden auf je eine Spalte abgebildet. Bei hierarchisch strukturierten Datentypen wird die Aufteilung ggf. rekursiv bis hinunter zu einfachen Datentypen fortgesetzt. Die Spalten eines zusammengesetzten Attributs liegen üblicherweise in der Tabelle der Entität. Auch hier kann ein vorhandenes Datenbankschema dazu zwingen, die Attributwerte aus verschiedenen Tabellen zu kombinieren.

6.5.5 1:1- und 1:0..1-Beziehungen

Einwertige Beziehungen sind auf zwei Arten abbildbar: Wenn der referenzierte Entitätstyp eine eigene Tabelle besitzt, verknüpft man beide Tabellen über einen Fremdschlüssel. In Ausnahmefällen kann man die Spalten des referenzierten Entitätstyps mit in die referenzierende Tabelle aufnehmen, um den JOIN über den Fremdschlüssel einzusparen.

Die Informationen über Tabellen und Schlüssel stehen dem Persistenzmanager im Modell zur Verfügung. Bei Bedarf können abweichend vom Primärschlüssel auch andere Eigenschaften bzw. Spalten für die Beziehung verwendet werden.

Da jede Fremdschlüsselbeziehung bei einwertigen Beziehungen symmetrisch ist, kann man auch auf der referenzierten Seite den Schlüssel des Ausgangsobjekts speichern, also die Fremdschlüsselbeziehung auf die andere Seite verschieben. Die Schlüsselseite sollte im Modell spezifiziert werden. Falls die Anwendung beide Richtungen zur Navigation nutzt, sind Hin- und Rückbeziehung im Persistenzmanager synchron zu halten.

Bei Zugriff auf eine einwertige Beziehung werden an der Schnittstelle zwischen Anwendung und Persistenzmanager Entitäten gehandelt; die Schlüssel sind Sache des Persistenzmanagers.

Die Kardinalität der Beziehung kann ohne Änderung des Datenbankmodells auf 1:n oder 1:0..n geändert werden, wenn der Fremdschlüssel auf der referenzierten Seite vorhanden ist. Wenn die Tabellen vorgegeben sind und keine Schlüsselspalten hinzugefügt werden können, ist zur Abbildung der Beziehung eine Brückentabelle mit den Schlüsselspalten beider Tabellen notwendig.

6.5.6 1:n- und 1:0..n-Beziehungen

Für jede mehrwertige Beziehung wird die Tabelle des referenzierten Typs (die n-Seite) um die Fremdschlüsselspalte(n) des referenzierenden Typs (die 1-Seite) erweitert. Wenn das Datenbankschema fest vorgegeben ist und wenn es den Fremdschlüssel nicht enthält, wird eine zusätzliche Tabelle mit Fremdschlüsseln beider Seiten hinzugefügt.

Zum Lesen und Ändern liefert der Persistenzmanager einen manipulier- und iterierbaren Behälter als *Set.* Dieser kann die Erzeugung der enthaltenen Objekte bis zum ersten Zugriff verzögern *(lazy loading).* Das Hinzufügen zur und Entfernen von Objekten aus der Beziehung protokolliert der Behälter, um die Änderungen bei Bedarf zu speichern.

Die Fremdschlüsselbeziehung in der Datenbank impliziert wie bei einwertigen Beziehungen eine Rückbeziehung, die von der Anwendung zur Navigation genutzt werden kann. Die Kardinalität der Beziehung kann ohne Schemaänderung auf einwertig geändert werden.

6.5.7 m:n- und m:0..n-Beziehungen

Die Abbildung beidseitig mehrwertiger Beziehungen benötigt eine Brückentabelle, deren Datensätze je zwei Objekte in Beziehung setzen. Die Brückentabelle enthält die Fremdschlüssel beider Entitäten. Eine solche Brückentabelle kann auch Beziehungen zwischen Objekten derselben Entität darstellen.

In den betroffenen A-Entitätsklassen sollte für m:n-Beziehungen jeweils *get*-Methoden bereitstehen, die den Wert der Beziehung bzw. Rückbeziehung in Form von manipulier- und iterierbaren Behältern (Bsp. *java.util.Set*) liefern. Die enthaltenen Objekte kann der Behälter zum geeigneten Zeitpunkt nachladen: so früh wie möglich (eager) oder so spät wie möglich (lazy). Das Hinzufügen zur und Entfernen von Objekten aus der Beziehung geschieht über den Behälter. Deshalb braucht man keine *set*-Methoden. Die Kardinalität der Beziehung kann bei Abbildung auf eine Brückentabelle theoretisch beliebig reduziert werden. Aufgrund des hohen Aufwands, den die Brückentabelle verursacht, sollte aber nach Möglichkeit die Abbildung angepasst werden.

6.5.8 Vererbung

Das Relationenmodell kennt weder Vererbung noch Polymorphie. Für die Abbildung von Vererbungsbeziehungen gibt es abhängig vom Zugriffsverhalten der Anwendung drei disjunkte Verfahren:

- Ein Vererbungsbaum – eine Tabelle,
- Eine Klasse – eine Tabelle (evtl. mit partieller Denormalisierung),
- Ein Vererbungspfad – eine Tabelle.

Alle drei Verfahren stellen Grenzfälle dar, wobei allgemein verwendbare Persistenzmanager in der Regel die zweite Variante wählen. Aus Performancegründen wird bei spezialisierten Klassen, die nur wenige zusätzliche Attribute enthalten, die dafür vorgesehene Tabelle eingespart. Stattdessen werden die wenigen Spalten dieser Klasse der Tabelle der Vaterklasse hinzugefügt. Bei Objekten anderen Typs bleiben diese Spalten unberücksichtigt. Falls Vererbung nur in geringem Umfang eingesetzt wird, kann mit der ersten oder letzten Variante der Aufwand für die Abbildung einer Klasse auf mehrere Tabellen eingespart werden.

Wenn Objekte einer vererbten Klasse über mehrere Tabellen verteilt abgelegt werden, werden die Tabellen der Kindklassen über Fremdschlüssel mit der Tabelle der Oberklasse verknüpft. Um die Zugehörigkeit eines Objekts zu einer bestimmten Klasse festzustellen, benötigt man in der Tabelle der Oberklasse ein Erkennungszeichen, an dem der Typ des Objekts erkennbar ist. Dieses Erkennungszeichen kann fachlicher oder technischer Art sein. Der Persistenzmanager kann die Notwendigkeit dieses Typschlüssels völlig verbergen, indem er beispielsweise in der Tabelle der Oberklasse eine von ihm selbst generierte Typ-ID ablegt.

In nachfolgender Tabelle aus [Keller 1998] sind die verschiedenen Abbildungsverfahren bewertet. Eine ausführlichere Darstellung findet sich unter http://www.objectarchitects.de/ObjectArchitects/orpatterns/.

Pattern	Performance			Speicherplatzbedarf	Flexibilität, Handhabung	Unterstützung von Abfragen
	Einfügen/ Aktualisieren	Einzelsatzlesen	Polymorphes Lesen			
One Inheritance Tree One Table – alle Attribute der Vererbungshierarchie in einer Tabelle	+ ●	+ ●	+	–	+	+
One Class One Table – pro Klasse eine Tabelle	–	–	– ●	+	+	–
One Inheritance Path One Table – pro Vererbungspfad eine Tabelle	+	+	–	+	–	–
Objekte in BLOBs	+ ●	+ ●	●	+	–	–

+ gut, – schlecht, * irrelevant, ● situationsabhängig (siehe [Keller 1998])

6.5.9 A- und T-Integrität

Das Klassenmodell des Persistenzmanagers enthält je nach Implementierung mehr oder weniger Integritätsregeln. Die Einhaltung der Integritätsregeln sollte so früh wie möglich geprüft werden, und zwar

- attributlokale bei Änderung,
- referentielle Integrität bei Änderung und
- attribut- und objektübergreifende bei Transaktionsende.

Letztere werden in Form einer Validierungsmethode implementiert, die vom Persistenzmanager aufgerufen wird. Die Prüfung der referentiellen Integrität führt bei Pflichtbeziehungen zum Löschen abhängiger Objekte *(Cascading Delete).* Um dies zu verhindern, sind Pflichtbeziehungen ggf. vor dem Löschen neu zu füllen.

Auf welcher Ebene wird die Konsistenz der A-Entitäten und ihrer Beziehungen untereinander sichergestellt? Hier bieten sich an:

- Datenbank,
- Persistenzmanager oder
- Anwendungskern.

In fast jedem Datenbanksystem lassen sich Bedingungen formulieren, die die referentielle Integrität oder sogar mehr sicherstellen. Foreign-Key-Constraints garantieren beispielsweise, dass Pflichtbeziehungen gefüllt sind und Fremdschlüssel nicht ins Leere zeigen. Trigger können bei Beziehungen eingesetzt werden, um ggf. referenzierte Objekte zu löschen. Die so implementierte Integrität nennen wir T-Integrität.

Da der Persistenzmanager nach der Bearbeitung eines Objekts dessen Beziehungen in Form programmiersprachlicher Objekte (Objektreferenzen, Container) kennt, kann er die Einhaltung der referentiellen Integrität im Arbeitsspeicher prüfen. Sofern im Objektmodell weitere Integritätsregeln kodiert sind, die sich möglicherweise nicht in der Datenbank abbilden lassen, muss der Persistenzmanager deren Einhaltung sicherstellen.

Über die aus dem Objektmodell herauslesbaren Bedingungen hinaus sind Plausibilitätsprüfungen üblich, die sich über mehrere Attribute derselben oder gar mehrerer A-Entitäten erstrecken. Um hierfür sprechende Fehlermeldungen zu liefern, sollten diese Prüfungen im Anwendungskern in den A-Entitätstypen oder A-Verwaltern stattfinden.

Manche Persistenzmanager bieten die Möglichkeit, allgemeine Regeln geringer Komplexität (Bsp. *mwstProzent* < 25), in ihrem Objektmodell zu formulieren. Beispielsweise kann man beim Quasar Persistenzmanager Regeln, die sich nur auf einen A-Datentyp beziehen, direkt im Objektmodell formulieren. Darüberhinaus können Regeln, die die Konsistenz eines Objekts bestimmen, in der persistenten Klasse kodiert werden.

Die Möglichkeiten der Datenbanken zur Definition von Integritätsregeln sind normalerweise proprietär. Deshalb ist ein Persistenzmanager, der die Prüfung der Integritätsregeln (teilweise) an das Datenbanksystem delegiert, ein Stück weit abhängig vom Datenbanksystem. Andererseits können manche Aspekte vom Datenbanksystem wegen der Nähe zu den Daten schneller geprüft werden. Je nach Umgebung (weitere Anwendungen auf derselben Datenbank oder im selben Zuständigkeitsbereich, weitere Anwendungen mit demselben Persistenzmanager) ist es vorteilhaft, die Integritätsregeln im Datenbankschema oder im Objektmodell zu hinterlegen.

Wird ein Teil der Integritätsregeln im Datenbankschema implementiert, sollte dies durch den Persistenzmanager selbst entsprechend dem Objektmodell erfolgen. So kann eine Verteilung von Wissen auf Datenbankschema und Objektmodell vermieden werden.

Vorhandene Datenbankschemata, etwa von andersartigen Nachbarsystemen, enthalten häufig bereits T-Integritäten. Diese müssen berücksichtigt werden, z.B. über kontrollierte Schreibreihenfolge zur Berücksichtigung von Foreign-Key-Constraints (vgl. Abschnitt 6.5.11).

Achtung: Nachbarsysteme implementieren u.U. weitere A-Integritätsbedingungen, die aus dem Datenbankschema nicht ohne weiteres ersichtlich sind. Bei schreibendem Zugriff auf diese Daten müssen alle A-Integritätsbedingungen bekannt sein, da es sonst zu Inkonsistenzen im Nachbarsystem kommen kann. Das gilt auch andersherum: Was in der eigenen Anwendung nicht im Schema deklariert ist, kann übersehen werden.

6.5.10 Projektionen

Häufig benötigt eine Anwendung keine kompletten Objekte, sondern nur eine Liste von Daten – individuell zusammengestellt aus Attributen verschiedener Entitäten. In Auswahllisten reicht es beispielsweise, wenn jede Zeile einer solchen Liste ein Objekt identifiziert, damit zu ausgewählten Zeilen bei Bedarf die entsprechenden Objekte instanziiert werden können. In Auswertungslisten kondensieren die einzelnen Zeilen häufig sogar mehrere fachliche Objekte. Die naive Implementierung solcher Listen würde alle benötigen A-Entitäten instanziieren, um eine Teilmenge der Attribute anzuzeigen. Diese Lösung hat zwei Nachteile:

1.

Um die Entitäten korrekt anzuzeigen, werden sie vollständig aus der Datenbank geladen, was in der Regel mehrere Anfragen pro Zeile nach sich zieht.

2.

Für jede Zeile der Ergebnismenge müssen alle beteiligten Objekte und ggf. dahinter liegende Objektge-

flechte konstruiert werden. Das Erzeugen von Objekten ist eine teure Operation, deren Aufwand sich nur lohnt, wenn auch mit den Objekten gearbeitet wird. Unter anderem wird die dynamische Speicherverwaltung unnötig strapaziert, wenn die meisten der angezogenen Objekte nicht weiter benötigt werden.

Besser ist es, mit JOIN oder VIEW nur die benötigten Daten aus der Datenbank zu lesen, diese dann in die in der Objektwelt bekannten Attributwerte zu konvertieren, um sie in Form dummer Datensätze weiterzureichen. Dies erspart das vielfache Instanziieren komplexer Objektgeflechte und die vielen dazu notwendigen SQL-Leseoperationen. Im Idealfall kann die ganze Liste mit einer einzigen SQL-Anweisung ermittelt werden. Die Liste von Datensätzen kann als Transfer-Datenstruktur dienen, wenn Datentypen und Container für den zwischen Client und Server verwendeten Transportmechanismus serialisierbar sind[1]. Bei den für Abfragen verwendeten Containern sind besondere Maßnahmen zur Serialisierung notwendig, wenn diese zur Bewältigung beliebig großer Ergebnismengen Lazy-Mechanismen implementieren. Eine Möglichkeit ist, erstmal eine für den Anwender ausreichend große, vom Transportvolumen her zulässige Menge an Datensätzen zu liefern – in der Hoffnung, dass der Anwender bei mehr als dieser Anzahl von Sätzen seine Suche einschränkt, anstatt durchzublättern.

Wenn die Zugriffsschicht Projektionen bzw. skalare Ergebnismengen unterstützt, wird das entsprechende Query-Interface genutzt. Ansonsten enthält ein zur Sicht passender A-Verwalter den SQL-Code mit JOINs oder Zugriff auf eine passende Datenbank-VIEW.

6.5.11 Abbildung auf vorhandene Datenbankschemata

Das Datenbankschema kann eine beliebige Form haben, eine u.U. aufwendige Abbildung auf A-Entitäten ist erforderlich. Eine Zugriffschicht kann und soll das bis zu einem gewissen Maß leisten. Mit folgenden Problemen ist zu rechnen:

1.
Konflikte zwischen den T-Integritätsbedingungen und insbesondere den A-Integritätsbedingungen des Nachbar- bzw. Altsystems und den eigenen A-Integritäts-Bedingungen (vgl. Abschnitt 6.5.9)

2.
Sperrkonflikte im parallelen Betrieb, wenn mehrere Systeme parallel schreibend auf dieselben Daten zugreifen.

3.
Das Datenbankschema ist für die Zugriffsmuster des vorhandenen Systems optimiert, die Zugriffsmuster des neuen Systems weichen voraussichtlich davon ab. Schlechte Performance (Optimierungsbedarf) kann die Folge sein.

4.
Greifen mehrere Anwendungen auf das selbe Schema zu, ist eine der Grundanforderungen an A-Komponenten verletzt – ihre Datenhoheit.

5.
Das Mapping ist aufwendig (z.B. weil das Datenbankdesign aus einer prozeduralen Anwendung stammt).

Insgesamt empfehlen wir folgendes:

1.
Unterschiede zwischen Datenbankschema und A-Entitäten werden soweit wie möglich von der Datenbankzugriffsschicht überbrückt.

2.
Systemintegration findet nicht auf der Ebene der Datenbank sondern auf Ebene der A-Fälle statt. Damit bleiben die Datenhoheit der beteiligten Systeme und die A-Integritäten gewahrt verglichen zu anderen Varianten der Integration [Linthicum 2001].

[1] Vergleiche dazu die Katalogklasse aus CAESAR und Narrow View aus [Keller 1998] und DataAccessObject + ValueObject [Alur 2001]

6.6 Performance

6.6.1 Caching

Caching lohnt sich bei oft gelesenen, aber selten geänderten Daten. Caching ist Unsinn für Daten, die nur einmal verwendet werden, z.B. Projektionsobjekte aus einer Auswahl.

Was wird im Cache gehalten?

- Schlüsselverzeichnisse, Aufzählungen (diese Objekte werden selten geändert, aber oft gelesen).
- Stammdaten. Oft werden Stammdaten nur beim Neustart der Anwendung aktualisiert.
- Metadaten, welche die Infrastruktur für eine Anwendung bereitstellen.
- Daten die zu einer Sitzung eines Benutzers gehören, die also zwischen zwei Dialogschritten aufgehoben werden.
- Workflow-Daten (Begründung wie bei der Sitzung).

Caching und Cluster

Caching ist komplex, wenn. mehrere Prozesse auf denselben Datenbestand zugreifen (Cluster): Dasselbe Objekt kann in den verschiedenen Caches in unterschiedlichen Versionen existieren.

Wenn eine Zugriffsschicht ein Cache-Kohärenz-Protokoll im Cluster anbietet, können auch änderbare Daten performant vorgehalten werden. Die Caches der Prozesse werden über das Kohärenz-Protokoll synchronisiert. *TopLink* bietet das beispielsweise seit Version 3.5.

Trotzdem sollten im Cluster nur nicht änderbare Objekte oder Objekte mit geringer Änderungswahrscheinlichkeit im Cache gehalten werden. Werden änderbare Daten im Cache aufbewahrt, ist ein optimistisches Sperrkonzept notwendig, um Konflikte zu erkennen.

6.6.2 Frühes und spätes Laden

Frühes Laden:
Mit jedem Objekt werden alle referenzierten Objekte sofort mitgeladen (man bekommt also die transitive Hülle). So wird im schlimmsten Fall die gesamte Datenbank auf einmal geladen.

Frühes Laden wird eingesetzt bei Geflechten mit kleinen Objekten, die häufig zusammen verwendet werden.

Horizontales spätes Laden:
Abhängige Objekte werden erst bei Bedarf aus der Datenbank geladen. Das kann realisiert werden über intelligente Referenzen *(lazy references),* Proxies (z.B. *ValueHolder* bei *TopLink*) oder mit *get*-Methoden, die beim ersten Aufruf die Nachlade-Query ausführen und erst dann das zugehörige Feld aus der Datenbank laden.

Vertikales spätes Laden:
Vertikales spätes Laden wird in Anfragen mit größeren Treffermengen angewandt. Die Treffer der Anfrage werden erst konstruiert, wenn auf sie zugegriffen wird. Das wird beispielsweise über einen Iterator auf der Treffermenge der Query implementiert, welcher nur die Treffer lädt, die er traversiert. Vgl. dazu den *Value-List-Handler* [Alur 2001].

Spätes Laden ist sinnvoll, wenn abhängige Objekte nur selten verwendet werden. Große abhängige Objekte, beispielsweise Bilder, lädt man erst bei Bedarf.

Die Wahl der Nachladestrategie hängt von den Zugriffsmustern ab. Oft ist eine A-Fall-spezifische Mischung aus frühem und spätem Laden sinnvoll.

6.6.3 Cluster-Lesen (intelligente Joins)

Cluster-Lesen bedeutet:
Mehrere Objekte einer oder mehrere A-Entitäten werden mit einem oder wenigen Lesebefehlen auf einmal aus der Datenbank geladen. In manchen Standardfällen kann dies eine Zugriffsschicht übernehmen, aber in der Regel ist für Optimierungen dieser Art Anwendungswissen erforderlich.

Beispiel:
Zwischen zwei Entitätstypen besteht eine 1:1 Beziehung, etwa ein Kunde mit genau einem Konto. Die beiden Entitätstypen sind in verschiedenen Tabellen gespeichert. Cluster-Lesen instanziiert den Kunden und sein Konto mit einem Join. 1:1 Beziehungen lösen die meisten Zugriffschichten über einen Join auf und erzeugen gleichzeitig beide Objekte.

6.6.4 Gespeicherte Prozeduren (Stored Procedures)

Eine gespeicherte Prozedur ist ein Programm, das in der Datenbankumgebung gespeichert und ausgeführt wird. Dies ermöglicht hohe Performance auch bei komplexen Datenbankanfragen, weil nur wenig Kommunikations-Overhead anfällt. Alle großen Anbieter relationaler Datenbanksysteme bieten Unterstützung für gespeicherte Prozeduren durch unterschiedliche Implementierungssprachen, wie z.B. PL/SQL, Java, C, COBOL oder REXX.

Gespeicherte Prozeduren reduzieren in erster Linie die Netzlast: Viele einzelne Datenbank-Operationen werden gebündelt in einer gespeicherten Prozedur abgearbeitet. Dadurch entfällt der Netzverkehr zwischen den einzelnen Operationen vollständig. Nur am Anfang und vielleicht am Ende werden Parameter und Ergebnisse über das Netzwerk transportiert. Für gespeicherte Prozeduren, die in Java implementiert werden, erreicht man mit SQLJ als Zugriffsmechanismus eine ähnlich hohe Performance wie unter C oder PL/SQL; allerdings ist das je nach Datenbank und Plattform (Windows, Unix, Host) unterschiedlich.

Anwendungsbeispiele für gespeicherten Prozeduren:

- Statische Suchanfragen, die komplexe, aufwendige SQL-Statements erfordern, oder Reporting-Anfragen, bei denen spezielle Fähigkeiten der Datenbank nötig sind.
- Datenintensive Anwendungslogik (z.B. Massendatenverarbeitung).
- Performance-kritische Anfragen sind mit gespeicherten Prozeduren möglicherweise schneller.

Eine Integration von gespeicherten Prozeduren in die Quasar-Persistenz und andere objektrelationale Tools ist gut möglich. Daher kann auch bei Verwendung von gespeicherten Prozeduren eine einheitliche Zugriffsschicht implementiert werden.

Folgende Punkte sind bei gespeicherten Prozeduren zu bedenken:

- Der Anwendungskern wird auf mehrer Ebenen der Systemarchitektur verteilt; er diffundiert in die Datenbank. Die Implementierung der Anwendung ist uneinheitlich; die Wartung komplexer.
- Debugging, Konfigurationsmanagement und Deployment sind umständlich und aufwendig.
- Die Last wird auf den Datenbankserver verlegt: Dies reduziert die Skalierbarkeit.

Anhang

Übersicht der Schnittstellen

Die Schnittstellen sind in den Abschnitten 6.3.1 bis 6.3.4 beschrieben.

Wichtig:
IPersistent und (beiBedarf *IInspectable*) sind Schnittstellen, die die Anwendung selbst implementiert. Alle anderen Schnittstellen werden von der Anwendung genutzt.

```
public interface IPersistent {
    IPool getPool();
    Object[] getPrimaryKey();
    void setPool(IPool pool);
    void setPrimaryKey(Object[] primaryKey);
    void lock (LockMode mode);
    void lock (LockMode mode, int timeoutMs);
}

public interface IInspectable extends IPersistent {
    Object getProperty(Object property);
    void setProperty(Object property, Object value)
        throws ValueException;
}

public interface IPersistenceManager {
    IPool getPool(String name);
}

public interface IAccessor {
    Object getValue(IPersistent entity);
    void setValue(IPersistent entity, Object value)
        throws ValueException;
}

public interface IRelationSet extends Set {
    Iterator iterator(LockMode lockMode);
    Iterator iterator(LockMode lockMode, int timeout);
}

public interface IQueryResult extends ListIterator {
    void close();
}

interface IQuery {
    IQueryResult execute(Object[] queryArguments);
}

public interface IPool {
    IsolationLevel getIsolationLevel();
    LockStrategy getLockStrategy();
    int getLockTimeout();

    IAccessor getAccessor(Class entityType, Object property);
    IAccessor getAccessor(IPersistent entity, Object property);

    IPersistent lookup(Class entityType, Object[] primaryKey);
    IPersistent lookup(Class entityType, Object[] primaryKey),
                       LockMode mode);
    IPersistent lookup(Class entityType, Object[] primaryKey),
                       LockMode mode, int timeoutMs);

    IQuery createQuery(Object querySpecification);

    IPersistent make(Class entityType);
    IPersistent make(Class entityType, Object[] primaryKey)
        throws PrimaryKeyException;

    void update(IPersistent entity);

    void remove(IPersistent entity);

    IQuery createQuery(Object queryDescription, LockMode mode);
    IQuery createQuery(Object queryDescription, LockMode mode,
                       int timeoutMs);
    void lock(IPersistent entity, LockMode mode);
    void lock(IPersistent entity, LockMode mode, int timeoutMs);
}
```

7 Graphische Benutzerschnittstellen

7.1 Einführung

Benutzerschnittstellen schlagen eine Brücke vom Benutzer hin zu den Funktionen der Anwendung. Sie stellen fachliche Informationen gegenüber dem Benutzer dar und ermöglichen ihm die Durchführung von fachlichen Aktionen zur Zusammenstellung und Manipulation dieser Informationen. Da moderne Benutzeroberflächen hierzu in aller Regel graphisch orientierte Darstellungsmittel und Interaktionsmetaphern verwenden, wird im allgemeinen Sprachgebrauch der Begriff GUI (Graphical User Interface) stellvertretend für den allgemeineren Begriff der Benutzerschnittstelle verwendet. Auch wir werden, diesem Sprachgebrauch folgend, den Begriff GUI verwenden, obwohl genaugenommen „UI" treffender wäre, denn für unsere Ausführungen besteht kein grundsätzlicher Unterschied zwischen graphischen und nicht-graphischen Darstellungsformen.

Die Gestaltung der Benutzerschnittstelle verursacht in Projekten einen erheblichen Anteil am Gesamtaufwand. Daran kann auch eine gute Architektur kaum etwas ändern. Trotzdem lohnt es sich, über GUI-Architektur nachzudenken, denn sie kann zur Stabilität und Flexibilität einer Anwendung beitragen. In der Benutzerschnittstelle treffen Fragen der Anwendung mit GUI-technischen Belangen aufeinander. Eine gute, den Quasar-Prinzipien Rechnung tragende Architektur hat die Aufgabe, diese beiden Aspekte so gut es geht zu trennen. Der Aufruf einer Anwendungskern-Funktion direkt im *Event-Listener* einer Swing-Schaltfläche ist zwar naheliegend, wird jedoch kaum dem Anspruch an überschaubare und wartungsfreundliche Software gerecht.

Zum Aufbau des Kapitels: Zur Vereinheitlichung der Terminologie beginnen wir mit einigen Begriffsdefinitionen. Es folgt die Einführung der Standardarchitektur für Benutzerschnittstellen, die die einzelnen Architekturkomponenten als eine Wertschöpfungskette versteht; das Zusammenspiel der Komponenten und die Verteilung von Software-Bestandteilen der unterschiedlichen Blutgruppen über diese Wertschöpfungskette sind Thema von Abschnitt 7.3.2. Vor diesem Hintergrund werden anschließend einige Spezialaspekte wie die Validierung von Benutzereingaben und die Komposition von Dialogen behandelt. Den Abschluss bilden Überlegungen zur internen Realisierung der Architekturelemente.

7.2 Definitionen

Dieser Abschnitt definiert eine Reihe von Begriffen aus Anwendersicht (A-Begriffe). Sie sind also für die Konstruktion *und* die Spezifikation gedacht.

- *Dialog*
  Ein Dialog bildet eine Einheit in der Mensch/Maschine-Kommunikation derart, dass darin für den Benutzer zusammenhängende Daten und Funktionen verfügbar sind (vgl. [Denert 1991]). Der Dialog ist die Bearbeitungseinheit des Anwenders. Jeder Dialog unterstützt einen oder mehrere A-Fälle und stellt sie über geeignete Visualisierungsmittel (z.B. ein Fenster oder eine HTML-Seite) am Bildschirm dar[1]. Die Dialog-Benutzerschnittstelle ist die Gesamtheit der Dialoge (vgl. [Siedersleben 2002]).

- *Sitzung*
  In jedem Mehrbenutzersystem ist es erforderlich, dem einzelnen angemeldeten Benutzer eine Sitzung als Kontext seiner Aktivitäten zuzuordnen[2]. Die Sitzung beginnt mit der Anmeldung des Benutzers (Login) im Anwendungssystem und endet mit seiner Abmeldung (Logout) bzw. im Fehlerfall mit der Beseitigung der Sitzung durch das Crash-Management. Jede Interaktion mit dem Benutzer geschieht unter Kontrolle einer Sitzung. Derselbe Benutzer kann ggf. mehrfach im System angemeldet sein und verfügt dann gleichzeitig über mehrere Sitzungen, die allerdings in keinem Zusammenhang zueinander stehen (so, als seien es die Sitzungen verschiedener Benutzer).

- *Dialoginstanz*
  Im normalen Sprachgebrauch wird der Begriff des Dialogs häufig in zweierlei Bedeutungen verwendet: einerseits für das Abstraktum, das ein bestimmtes Verhalten definiert, andererseits für ein konkretes Exemplar, mit dem ein Benutzer im Rahmen einer Sitzung zu einem gegebenen Zeitpunkt arbeitet. Diese unscharfe Verwendung des Begriffs ist zulässig, solange sie keine Missverständnisse provoziert. Vor allem im Zusammenhang mit der Festlegung von Kardinalitäten aber, d.h. bei der Frage, wie viele Exemplare eines Dialogs innerhalb einer Sitzung existieren können, sprechen wir – in Anlehnung an die Objektorientierung – beim einzelnen Exemplar von einer *Dialoginstanz.* Grundsätzlich sind mehrere Instanzen desselben Dialogs innerhalb einer Sitzung möglich, die unterschiedliche oder auch dieselben Daten anzeigen und bearbeiten können. Umgekehrt gehört jede Dialoginstanz zu genau einer Sitzung. *Singleton-Dialoge* sind innerhalb einer Sitzung höchstens einmal vorhanden; *Nicht-Singleton-Dialoge* können in mehreren Instanzen existieren. Beispiel: Suchdialoge sind meistens Singletons, Pflegedialoge selten.

- *Dialogkategorie*
  In der Praxis lassen sich häufig Dialoge mit gleichartiger Zweckbestimmung und gleichartigem Verhalten beobachten, die man zu Kategorien zusammenfassen kann, etwa Suchdialoge, Dialoge zur Einzelsatzpflege oder zur Mehrsatz-

[1] Dies ist eine relativ weitgefasste Definition des Begriffs „*Dialog*". Man kann – nicht anders als bei Klassen – im Extremfall unsinnig kleine oder unsinnig große Dialoge bauen.

[2] Die Sitzung ist kein originäres Konzept aus dem Bereich der Benutzerschnittstellen, sie wird durchgängig von verschiedenen Teilen einer Anwendung – so auch vom Anwendungskern – benötigt. Zwar fällt also ihre Verwaltung nicht in die Zuständigkeit der GUI, trotzdem aber ist ihre Erwähnung hier angebracht, da sie in Mehrbenutzersystemen eben auch für die Benutzerschnittstelle eine wichtige Kontextinformation darstellt.

pflege. Jedes Projekt sollte eine sinnvolle Zahl (<= 10) von Dialogkategorien definieren. Typischerweise kann man die meisten Dialoge einer Anwendung mit wenigen Kategorien beschreiben; daneben gibt es einige wenige Spezialdialoge, die in keine Kategorie passen.

- *Öffnen und Schließen eines Dialogs*
  Dialoginstanzen werden nach Bedarf als Reaktion auf Benutzeraktionen erzeugt (z.B. wenn der Benutzer aus einer Liste das zu bearbeitenden Objekt auswählt und in die Pflege verzweigt) und beendet. Das Erzeugen einer Dialoginstanz nennen wir auch *Öffnen eines Dialogs,* analog heißt das Beenden der Dialoginstanz *Schließen des Dialogs.*

- *Aktivieren und Unterbrechen eines Dialogs*
  Jede offene Dialoginstanz ist entweder *aktiv* oder *unterbrochen.* Aktive Dialoginstanzen reagieren auf Benutzereingaben in der üblichen Weise; unterbrochene Dialoginstanzen warten auf ihre Reaktivierung. Je nach System gibt es in einer Sitzung genau eine oder mindestens eine aktive Dialoginstanz sowie beliebig viele unterbrochene Dialoginstanzen. Auch bei mehreren aktiven Dialoginstanzen hat höchstens eine dieser aktiven Dialoginstanzen den Fokus, denn technisch kann der Cursor nur an einer Stelle sein. Der Benutzer kann allerdings willkürlich zwischen aktiven Dialoginstanzen wechseln.

- *Dialogwechsel*
  Der Dialogwechsel beendet oder unterbricht die aktuelle Dialoginstanz und erzeugt oder reaktiviert eine Folgedialoginstanz. Dabei kann Kontextinformation an die Folgedialoginstanz fließen.

- *Unterdialog*
  Ein spezieller Fall des Dialogwechsels ist das Öffnen eines Unterdialogs. Ein Unterdialog ist ein Dialog, der nicht allein existieren kann, sondern stets von der Existenz eines anderen, übergeordneten Dialogs abhängig ist. Der Unterdialog wird durch Initiative des übergeordneten Dialogs geöffnet und spätestens mit diesem zusammen wieder geschlossen, möglicherweise aber auch schon vorher. Die übergeordnete Dialoginstanz kann dabei unterbrochen sein oder aktiv bleiben. Beispiele: Ein Dialog zur Anzeige von Validierungsfehlern, die beim Speichern eines Bearbeitungsdialogs auftreten, ist ein Unterdialog des Bearbeitungsdialogs, denn er ist ohne den Bearbeitungsdialog sinnlos. Dies gilt unabhängig davon, ob der Fehlerdialog den Bearbeitungsdialog unterbricht oder ob beide Dialoge aktiv bleiben und so dem Benutzer den freien Fokuswechsel zwischen den Dialogen ermöglichen. Wird hingegen von einem Dialog zur Bearbeitung eines Kunden aus ein Dialog zur Pflege eines Stammdatums (z.B. der Anrede) geöffnet, so ist der Stammdatendialog kein Unterdialog, da er auch unabhängig vom Kunden-Dialog sinnvoll geöffnet werden kann.

- *Aktion*
  Aktionen finden als Reaktion auf die Handlungen eines Benutzers statt. Sie können rein technischer Natur sein (z.B. die Reaktion der GUI-Bibliothek auf die Bewegung eines Rollbalkens), den Dialogablauf betreffen (z.B. Dialogwechsel) oder auf die Fachlichkeit zugreifen (z.B. das Lesen oder Schreiben von Anwendungsdaten).

- *Standardaktion*
  Standardaktionen sind solche, die bis auf definierte Ausnahmen in jedem Dialog und in jedem Dialogzustand möglich sind und deren Auswirkung immer dieselbe ist. Beispiele: *Abbruch, Undo, Zurück zum letzten Dialog.* Jedes Projekt sollte eine sinnvolle Menge von Standardaktionen definieren.

- *Dialogschritt*
  Die Verarbeitung einer einzelnen Handlung des Benutzers, die die Durchführung einer oder mehrerer Dialogaktionen beinhaltet, nennen wir einen *Dialogschritt.* In der Regel – aber nicht notwendigerweise – ist die zugehörige visuelle Darstellung während des Dialogschritts gesperrt. Vom Benutzer wird die Dauer des Dialogschritts als *Antwortzeit* wahrgenommen.

- *Formular* (synonym auch *Maske*)
  Ein Formular ist eine typische Form (Bedienmetapher), in der sich ein Dialog sichtbar darstellt. Es gestattet die Ein- und Ausgabe von Daten in einen vorgegebenen Rahmen, der mit den gängigen Bedienelementen wie Textfeldern und Listboxen gestaltet wird. Die Aufteilung der Daten auf Formulare ist abhängig vom Medium (z.B. der Größe des Bildschirms).

- *Dokument*
  Nicht alle Daten lassen sich in den vorgefertigten Rahmen eines Formulars einpassen, für manche Arten von Daten ist eine andere Bedienmetapher, das Dokument, geeigneter. In einem Dokument überwiegen nicht wie im Formular die typischen Bedienelemente einer GUI-Bibliothek, es verwendet spezielle, auf die Art der Daten zugeschnittene Formen der Datenvisualisierung (z.B. die Matrix von Kalkulationszellen in einem Tabellenkalkulationsprogramm). *Textuelle Dokumente* legen den Schwerpunkt auf die Eingabe von Text (z.B. MS Word, Excel), *direkt-manipulative Dokumente* basieren auf der graphischen Manipulation fachlicher Objekte mit Hilfe von Maus, Zeichenbrett und anderen Eingabemedien (z.B. Rational Rose, Visio). In modernen Systemen zur Dokumentverarbeitung gehört ein Undo/Redo-Mechanismus zur Standardausstattung.

7.3 GUI-Architektur

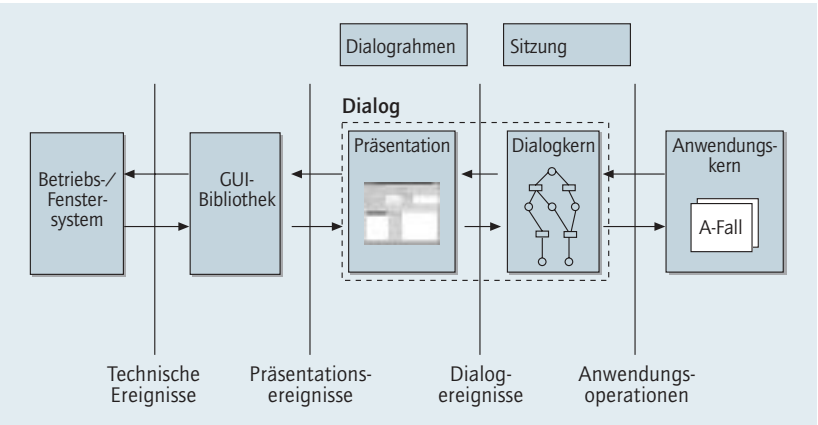
7.3.1 Komponenten einer GUI-Architektur

Ein Dialog ist stets in ein Umfeld aus technischen und anwendungsspezifischen Rahmenbedingungen eingebettet, wie es in Abbildung 21 skizziert ist.

Das direkte Umfeld eines Dialogs besteht aus verschiedenen Komponenten:

- Gegeben ist das *Betriebssystem* der Zielplattform, das bereits eine mehr oder minder komfortable Unterstützung für den GUI-Bereich anbietet (z.B. X oder Windows).
- Der GUI-Anwendungsentwickler arbeitet nicht direkt mit den Programmierschnittstellen des Betriebssystems, sondern verwendet eine *GUI-Bibliothek,* die durch geeignete – nicht selten auch betriebssystem-übergreifend einsetzbare – Abstraktionen den Entwicklungskomfort erhöht. Im Normalfall kommt eine bestehende Bibliothek zum Einsatz (z.B. AWT/Swing, Qt, HTML-Browser im Zusammenspiel mit Servlets), im Ausnahmefall liefert der Entwickler diese Bibliothek selbst. Die gängigen Umgebungen besitzen einen weitgehend einheitlichen Funktionsumfang: Scrolling, Koordination von Radio-Buttons und viele andere Funktionen gibt es schon in der Bibliothek. Trotzdem besteht immer wieder die Notwendigkeit, die vorhandene GUI-Bibliothek zu erweitern.
- Auf der anderen Seite sieht der Dialog den *Anwendungskern* mit den A-Fällen (Use Cases) als Schnittstelle. Der Dialog realisiert fachliche Aktivitäten durch Aufruf entsprechender Funktionen des Anwendungskerns; er implementiert diese also nicht noch einmal selbst. Mögliche Ausnahmen von dieser Regel werden in Abschnitt 7.4.2 behandelt.
- Ein Dialog findet in einem Mehrbenutzersystem innerhalb einer *Sitzung* eines angemeldeten Benutzers statt.
- Der *Dialograhmen* repräsentiert das System insgesamt[3]. Er übernimmt diejenigen Querschnittsaufgaben, die sich nicht dem einzelnen Dialog oder der einzelnen Sitzung zuordnen lassen. Dazu gehört z.B. das Starten und Beenden des Systems und möglicherweise auch die Verwaltung der Benutzersitzungen sowie eines Datenpools, der den Dialogen als gemeinsame Datenbasis dient (dies sind normalerweise nicht die Originaldaten – diese sind Sache des Anwendungskerns – sondern eine dialogtaugliche Form dieser Daten, z.B. Transferobjekte). Der Dialograhmen bildet das gemeinsame Dach für die Menge der vorgesehenen Dialoge eines Systems. Je nach fachlichen Anforderungen kann der Dialograhmen selbst wieder aus Teil-Dialograhmen bestehen, er selbst erhält dadurch den Charakter eines Portals (und wird sich typischerweise eines Portaldialogs bedienen, um Navigationspunkte zum Start der verschiedenen Teil-Dialograhmen anzubieten). Ein System verfügt üblicherweise nur über eine Dialograhmeninstanz.

Abbildung 21: Umfeld und Bestandteile eines Dialogs



Die Aufgaben des Dialogs untergliedern sich in zwei Bereiche, die von je einer Architekturkomponente übernommen werden[4]:

- Die *Dialogpräsentation* oder kurz *Präsentation* ist für die visuelle Darstellung von Informationen am Bildschirm und die Interaktion mit dem Benutzer zuständig. Die zentralen Fragen für diese Komponente lauten: Wie wird etwas dargestellt? Wie löst der Benutzer Aktionen aus? Dazu ist sowohl technisches Wissen über die GUI-Bibliothek als auch fachliches Wissen über die darzustellenden Informationen nötig.
- Der *Dialogkern* übernimmt Aufgaben, die nichtvisuellen, logischen Charakter haben. Seine Kernfragen lauten: Was wird dargestellt? Wie wird auf Aktionen des Benutzers reagiert? Zur Erfüllung dieser Aufgaben braucht der Dialogkern – neben technik- und anwendungsneutralem Steuerungswissen – Wissen über die konkrete Anwendung (im Sinne der geeigneten Anbindung des Anwendungskerns). Die GUI-Bibliothek sieht er nicht.

Die fünf Architekturkomponenten *Betriebssystem, GUI-Bibliothek, Präsentation, Dialogkern* und *Anwendungskern* (vgl. Abbildung 21) bilden eine Kette, die Benutzeraktionen bearbeitet. Am Beginn steht ein auslösendes Ereignis, das vom Betriebssystem wahr-

[3] Zuweilen wird für dieses Konzept der Begriff „Anwendung" verwendet. Wir vermeiden diesen Begriff hier jedoch, da „Anwendung" in Quasar vorbelegt ist für Dinge, die im Zusammenhang mit der Fachlichkeit eines Systems stehen. Der Dialograhmen hingegen ist ein technisches Konzept.

[4] Wir vermeiden hier bewusst den Begriff „Schicht". Schichten im klassischen Sinne sind einander über- bzw. untergeordnet, und höhere Schichten dürfen nur Funktionen von tieferen Schichten aufrufen, nicht jedoch umgekehrt. Für das Zusammenspiel von Präsentation und Dialogkern soll hier aber grundsätzlich eine weniger strenge Auslegung gelten. Allgemein sehen wir darin die Interaktion zweier gleichrangiger Partner. Dies schließt nicht aus, dass eine Konkretisierung der Architektur diesen weitgesteckten Rahmen zu einer Schichtung verengt, bei der die Präsentation die höhere und der Dialogkern die tiefere Schicht darstellt.

genommen wird. Das Ereignis durchwandert nacheinander die einzelnen Glieder der Kette und bewirkt schließlich die Durchführung fachlicher Funktionen im Anwendungskern und/oder Veränderungen im Dialogablauf. Die Verarbeitung eines Ereignisses in einer Komponente zieht jeweils den Rücktransport der Verarbeitungsergebnisse zur vorangehenden Komponente der Kette nach sich. Jede Station arbeitet innerhalb dieses Ablaufs nach demselben Muster: Sie verarbeitet ein erhaltenes Ereignis entweder selbst oder leitet es an die nächste Station weiter oder auch beides zugleich. Vor der Weiterleitung bereitet sie die Information des Ereignisses für die nächste Station auf. Das Ereignis abstrahiert so Schritt für Schritt von technikbezogenen Inhalten und wird mehr und mehr mit anwendungsbezogenem Gehalt angereichert. Wir sprechen daher auch von der *Wertschöpfungskette* einer Benutzerschnittstelle:

- Jede Aktion des Benutzers wird vom Betriebssystem als *technisches Ereignis* wahrgenommen. Ein technisches Ereignis ist noch ohne jede inhaltliche Interpretation, es beschreibt eine rein technische Gegebenheit wie etwa einen Mausklick mit bestimmten Bildschirmkoordinaten. Das Betriebssystem leitet das technische Ereignis an die GUI-Bibliothek weiter.
- Die GUI-Bibliothek interpretiert das technische Ereignis im Sinne der von ihr verwendeten Oberflächenmetaphern, also z.B. als Betätigung einer Schaltfläche oder als Auswahl eines Menüeintrags. Die so gedeuteten Ereignisse sind entweder *Präsentationsereignisse,* die an die Präsentation weitergeleitet werden, oder sie werden schon innerhalb der GUI-Bibliothek selbst verarbeitet (z.B. Bewegung eines Rollbalkens).
- Die Präsentation verarbeitet ein eingehendes Präsentationsereignis entweder selbst oder transformiert es zu einem *Dialogereignis*[5] und leitet dieses an den Dialogkern weiter – oder auch beides. In hochinteraktiven Anwendungen, in denen bereits der einzelne Tastendruck des Benutzers eine Veränderung in der Benutzerschnittstelle hervorrufen kann (z.B. Ausgrauen einer ‚Speichern'-Schaltfläche beim Löschen des letzten Buchstabens aus einem Textfeld), gibt es viele verschiedene Präsentationsereignisse. Die Präsentation muss besonders schnell reagieren und verarbeitet daher viele Präsentationsereignisse selbst. Nur die grundlegenden Ereignisse, die sich auf den Dialogablauf oder die Anwendung beziehen, erreichen als Dialogereignisse den Dialogkern. In HTML-basierten Anwendungen hingegen erlauben die eingeschränkten technischen Möglichkeiten keine so feinfühlige Reaktion der Oberfläche. Präsentationsereignisse werden dort in den meisten Fällen als Dialogereignisse an den Dialogkern weitergegeben. Das Dialogereignis enthält keinen Hinweis auf seine Herkunft. Es hebt die Handlungsanweisung auf die Anwendungsebene: Den Dialogkern interessiert nur, *was* sein Auftrag im inhaltlichen Sinn ist, aber nicht, wie dieser Auftrag vom Benutzer gegeben wurde. So werden z.B. die beiden Präsentationsereignisse *Schaltfläche ‚Speichern' gedrückt* wie auch *Menüeintrag ‚Speichern' gewählt* zu dem Dialogereignis *Speichern der eingegebenen Kundendaten* verdichtet.
- Der Dialogkern verarbeitet das eingehende Dialogereignis entweder selbst (indem er Veränderungen im Dialogablauf anstößt, z.B. einen Dialogwechsel) und/oder setzt es in den Aufruf von Operationen des Anwendungskerns um. Man kann sagen, dass das Ereignis in diesem letzteren Fall in eine Form transformiert wird, die überhaupt keinen Bezug mehr zur Benutzerschnittstelle enthält.

Für die beiden Architekturkomponenten *Präsentation* und *Dialogkern* definieren wir noch einige weitere Begriffe :

- *Dialogaktion*
  Dialogaktionen sind die Aktionen des Dialogkerns, mit denen er auf Dialogereignisse reagiert.

- *Dialogdaten*[6]
  Der Dialogkern verfügt über eine eigene Repräsentation der darzustellenden Datenobjekte. Dies können die fachlichen Originalobjekte sein, wie sie der Anwendungskern liefert, häufiger wird es sich aber um daraus abgeleitete Transportstrukturen handeln (vor allem dann, wenn Dialog und Anwendungskern durch Prozessgrenzen getrennt sind).

- *Dialogzustand*
  Der Dialogkern kennt in der Regel mehrere Zustände (z.B. *Eingaben bestätigt, Eingaben unbestätigt*); der Dialogkern jeder Dialoginstanz befindet sich zu jedem Zeitpunkt in genau einem dieser Zustände. Auf Basis seines aktuellen Zustands wählt der Dialogkern geeignete Dialogaktionen als Reaktion auf ein Dialogereignis aus.
  Der Dialogkern reagiert auf das Eintreffen von Dialogereignissen mit der Durchführung einer oder mehrerer Dialogaktionen, deren Auswahl er vom aktuellen Dialogzustand abhängig macht. Die Ergebnisse dieser Aktionen bestimmen den neuen Zustand des auslösenden Dialogkerns und ggf. auch der Dialogkerne anderer Dialoginstanzen (z.B. des Folgedialogs oder – beim Beenden der Sitzung – sogar aller Dialoginstanzen).

- *Dialoggedächtnis*
  Dialogdaten und Dialogzustand zusammen nennen wir das Dialoggedächtnis des Dialogkerns.

- *Präsentationsaktion*
  Präsentationsaktionen sind die Aktionen der Präsentation, mit denen sie auf Präsentationsereignisse reagiert.

[5] *Dialogereignis* ist synonym zur *virtuellen Taste* in [Denert 1991]

[6] Präziser wäre eigentlich der Begriff „Dialog*kern*daten" (in Abgrenzung von den Präsentationsdaten, s.u.). Wir ziehen jedoch den eingängigeren Begriff „Dialogdaten" vor. Analoges gilt für Dialogzustand, Dialoggedächtnis und Dialogaktion.

- *Präsentationsdaten*
  Auch die Präsentation kann über eine eigene Repräsentation der darzustellenden Daten verfügen. Das wird insbesondere dann der Fall sein, wenn die verwendete GUI-Bibliothek dies nahe legt (in Swing etwa werden allen Bedienelementen *Models* zugrunde gelegt, deren Verwendung sich vor allem für komplexere Bedienelemente wie Tabellen und Bäume unbedingt empfiehlt).

- *Präsentationszustand*
  Auch die Präsentation kann bei Bedarf eigene Zustandsinformationen verwalten. Solche Zustandsinformationen sind vor allem dann notwendig, wenn die Präsentation Präsentationsereignisse selbst verarbeitet (z.B. bei der Prüfung, ob ein Pflichtfeld gefüllt ist).

- *Präsentationsgedächtnis*
  Präsentationsdaten und Präsentationszustand bilden zusammen das Präsentationsgedächtnis.

In welchem Verhältnis steht unser Architekturmodell zum MVC-Ansatz[7] (Model-View-Controller)? Zwei grundsätzliche Aussagen sind kennzeichnend für dieses Konzept. Zum einen unterscheidet MVC drei Beteiligte in der Architektur einer Benutzerschnittstelle: Das *Model* als Verantwortlichen für die Anwendung und insbesondere die Daten, die *View* mit der Verantwortung für die sichtbare Darstellung der Daten sowie den *Controller,* der die Verarbeitung von Benutzeraktionen durchführt. Zum anderen – und dieser Punkt wird häufig unterschlagen – regelt MVC die Aktualisierung der Darstellung nach abgearbeiteter Benutzeraktion auf eine besondere Weise: Eine oder auch mehrere Views registrieren sich beim Model als *Beobachter.* Sobald sich Daten ändern, benachrichtigt das Model alle registrierten Beobachter; die Views reagieren darauf mit der Aktualisierung ihrer Darstellung.

In unserem Architekturmodell ist die View (oder auch mehrere Views, die demselben Dialog zugeordnet sind) Bestandteil der Präsentation. Controller-Funktionalität findet sich sowohl in der Präsentation als auch im Dialogkern, denn beide tragen zur Verarbeitung von Benutzeraktionen bei. Dialoggedächtnis bzw. Präsentationsgedächtnis nehmen die Rolle des Models ein. Das Beobachter-Prinzip ist eine mögliche Lösung für die Aufgabe, die sichtbare Darstellung auf die Ergebnisse der Ereignisverarbeitung anzupassen (die Präsentation registriert sich dann als Beobachter von Dialog- bzw. Präsentationsgedächtnis). Wir kommen im nachfolgenden Abschnitt unter dem Stichwort *Synchronisation* auf dieses Thema zurück.

[7] Siehe auch [Buschmann 1996]

7.3.2 Das Prinzip der Kommunikation zwischen den Architekturkomponenten

Dieser Abschnitt vertieft einen wichtigen Aspekt der im vorangegangenen Abschnitt vorgestellten GUI-Architektur: die Kommunikation innerhalb der Wertschöpfungskette. Die Kommunikation zwischen zwei benachbarten Komponenten lässt sich zu gleich bleibenden Mustern verallgemeinern, die auf alle Komponentenpaare anwendbar sind.

Als Kommunikationspartner stehen sich jeweils eine ereignisauslösende und eine ereignisverarbeitende Komponente gegenüber, etwa die Präsentation, die ein Dialogereignis erzeugt, und der Dialogkern, der auf dieses Dialogereignis reagiert.

Die Kommunikation zwischen den beiden Partnern erfolgt in zwei Schritten:

- Anstoß der Ereignisverarbeitung: Wie erfährt die ereignisverarbeitende Komponente vom Eintreten eines bestimmten Ereignisses, und wie erhält sie die Informationen, die sie zur Verarbeitung des Ereignisses benötigt?
- Anstoß der Synchronisation: Wie erfährt die ereignisauslösende Komponente vom Abschluss der Ereignisverarbeitung, und wie erhält sie das Ergebnis der Verarbeitung?

Für die Realisierung beider Schritte bestehen Gestaltungsspielräume; die beiden nachfolgenden Unterabschnitte diskutieren verschiedene Varianten. Der dritte Unterabschnitt schließlich wendet das allgemeine Kommunikationsprinzip auf die einzelnen Komponentenpaare unserer Wertschöpfungskette an und arbeitet die jeweils anwendbaren Varianten heraus.

7.3.2.1 Ereignisverarbeitung

Abbildung 22 skizziert das Zusammenspiel der beiden Partner bei der Ereignisverarbeitung in Form von Methodenaufrufen. Diese Methoden stehen für die Aufgaben, die innerhalb der Kommunikation zu bewältigen sind; sie sind lediglich als Vorlage für tatsächlich implementierbare Schnittstellen zu verstehen.

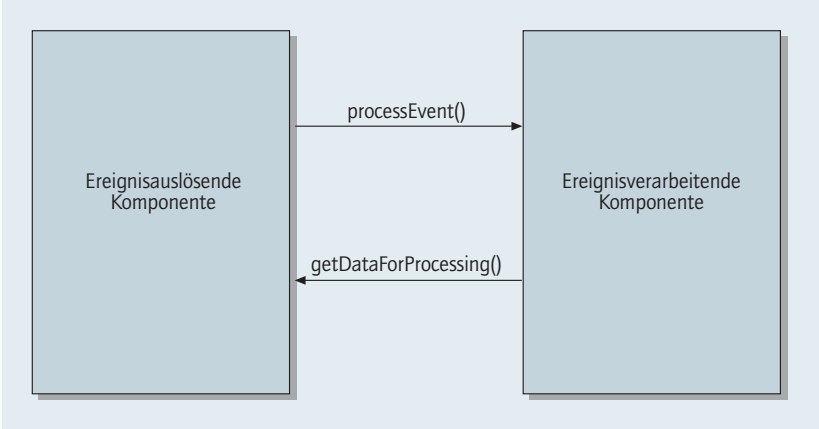


Abbildung 22: Zusammenspiel bei der Ereignisverarbeitung

Der Anstoß zur Ereignisverarbeitung kann auf zwei verschiedenen Wegen erfolgen:

- Direkter Aufruf: Die ereignisverarbeitende Komponente bietet in ihrer Schnittstelle eine Methode (oder auch mehrere Methoden) an, die die Ereignisverarbeitung des Partners anstoßen (der Name einer solchen Methode könnte dann z.B. *processEvent* lauten). Die ereignisauslösende Komponente ruft diese Methode bei Vorliegen eines Ereignisses auf.
- Registrierung: Die ereignisauslösende Komponente stellt einen Mechanismus zur Verfügung, durch den sich andere Beteiligte, die sich für ein Ereignis interessieren, bei ihr registrieren bzw. von außen registriert werden. Der Interessent muss dabei entweder selbst eine vorgegebene Schnittstelle implementieren oder ein Objekt (z.B. einen Event-Handler) bereitstellen, das diese Schnittstelle implementiert. Tritt das Ereignis ein, werden alle registrierten Interessenten von der ereignisauslösenden Komponente durch Aufruf einer entsprechenden Methode aus der Schnittstelle benachrichtigt[8]. Die ereignisverarbeitende Komponente ist ein solcher Interessent; sie registriert sich entweder selbst oder wird von einer außenstehenden Instanz registriert.

In der Variante des direkten Aufrufs weiß die ereignisauslösende Komponente, mit wem sie es zu tun hat; sie kennt die entsprechende Schnittstelle ihres Partners. Bei der Registratur-Variante hingegen gibt die ereignisauslösende Komponente selbst die Schnittstelle vor, die die Interessenten erfüllen; weitergehende Kenntnisse über ihre Kommunikationspartner benötigt sie nicht.

Im Zusammenhang mit dem Anstoß der Ereignisverarbeitung steht die Frage, auf wessen Initiative Datenänderungen, die in der ereignisauslösenden Komponente stattgefunden haben (z.B. Eingaben des Benutzers), an die ereignisverarbeitende Komponente gelangen. Auch hier gibt es zwei verschiedene Möglichkeiten:

- Die ereignisverarbeitende Komponente fordert – sofern sie im Zuge der Verarbeitung einen entsprechenden Bedarf erkennt – die auslösende Komponente auf, die aktuellen Daten bereitzustellen; dazu benutzt sie eine entsprechende Methode in der Schnittstelle der Präsentation (z.B. *getDataForProcessing*).
- Die ereignisauslösende Komponente liefert die veränderten Daten beim Anstoß der Ereignisverarbeitung mit (dann erübrigt sich die Methode *getDataForProcessing*).

Wie zeigt sich dieses allgemeine Prinzip in den Komponentenübergängen entlang der Wertschöpfungskette[9]?

1. GUI-Bibliothek als ereignisauslösende, Präsentation als ereignisverarbeitende Komponente

Bei der GUI-Bibliothek handelt es sich in der Regel um eine fremdentwickelte Komponente, z.B. Swing oder HTML in Verbindung mit Servlets. Eine Fremdkomponente kann naturgemäß keine Methoden aus der Schnittstelle der Präsentation aufrufen, da sie diese Schnittstelle nicht kennt. Die Präsentation nutzt daher den jeweiligen Registrierungsmechanismus der GUI-Bibliothek, um ihr Interesse am Erhalt von Präsentationsereignissen anzumelden. In Swing etwa registriert sie zu diesem Zweck *EventListener* bei den betroffenen Bedienelementen; benötigt sie zur Verarbeitung des Präsentationsereignisses die aktuellen Benutzereingaben, so fordert sie diese bei der GUI-Bibliothek an (z.B. über entsprechende Swing-Getter-Methoden). Im HTML/Servlet-Kontext spezialisiert sie ein Servlet, in dessen überschriebener *doGet*- bzw. *doPost*-Methode sie Zugriff auf die Ereignisparameter (d.h. die Parameter des HTTP-Requests) nehmen kann; zu diesen Parametern gehören auch die aktuellen Benutzereingaben; die GUI-Bibliothek liefert sie in diesem Fall bei jedem Request komplett mit.

[8] Wenn die eingesetzte Programmiersprache Funktionsobjekte unterstützt, kann die Registrierung auch in Form eines Funktionsobjekts erfolgen, das eine vorgegebene Signatur erfüllt; bei Eintritt des Ereignisses wird dann diese Funktion generisch aufgerufen.

[9] Den Übergang vom Betriebssystem zur GUI-Bibliothek behandeln wir hier nicht, da er für gewöhnlich nicht in den Zuständigkeitsbereich des Anwendungsentwicklers fällt.

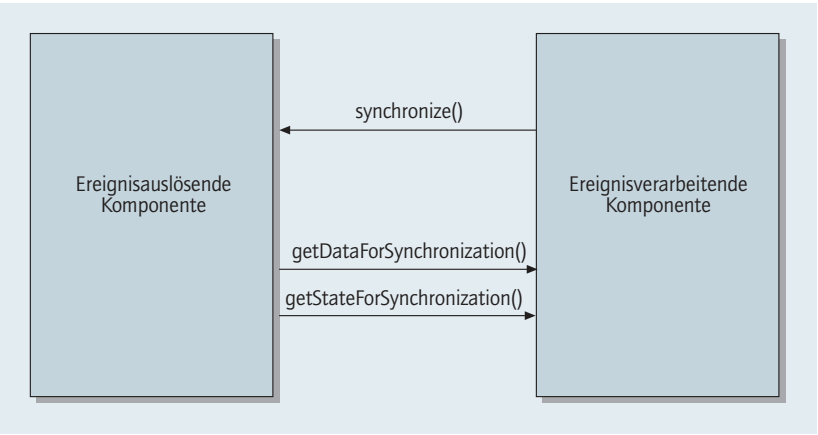
2.
Präsentation als ereignisauslösende, Dialogkern als ereignisverarbeitende Komponente

Der direktere Weg ist in dieser Konstellation die unmittelbare Meldung des Dialogereignisses durch die Präsentation an den Dialogkern (direkter Aufruf von *processEvent*), ungewöhnlicher wäre das Registrieren des Dialogkerns bei der Präsentation als Event-Listener. Typischerweise aktualisiert in beiden Fällen die Präsentation die Dialogdaten mit den Benutzereingaben, entweder auf eigene Initiative oder auf Nachfrage des Dialogkerns (letzteres ist gleichbedeutend mit dem Aufruf von *getDataForProcessing*). Eine besondere Variante dafür, dass die Präsentation die Benutzereingaben in die Datenobjekte überträgt, ist mit Smalltalk bekannt geworden: Die einzelnen Bedienelemente werden über Adapter-Objekte (in Smalltalk heißen sie Value-Models) direkt an die zugrundeliegenden Datenobjekte angekoppelt; Änderungen des Feldinhalts werden sofort automatisch in die Datenobjekte übertragen.

3.
Dialogkern als ereignisauslösende, Anwendungskern als ereignisverarbeitende Komponente

Beim Zusammenspiel von Dialogkern und Anwendungskern besteht die Ereignisverarbeitung in der Aktivierung von Anwendungsfällen. Den Anstoß dazu liefert der Dialogkern in aller Regel durch Aufruf einer oder mehrerer Methoden aus der Schnittstelle des ereignisverarbeitenden Anwendungskerns (diese Methoden sind das Äquivalent zu unserer *processEvent*-Methode). Der Dialogkern liefert die vom Anwendungskern benötigten Daten als Parameter des Methodenaufrufs mit; das umgekehrte Verfahren, bei dem sich der Anwendungskern die Daten beim Dialogkern beschaffen würde, ist nicht anwendbar, da der Anwendungskern sinnvollerweise keine Annahmen über seine mögliche Verwendung in Dialogen macht.

Abbildung 23:
Zusammenspiel bei der Synchronisation



Der abschließende Abschnitt über die Realisierungsdetails wird sich noch ausführlicher mit Verfahren der Ereignisverarbeitung auseinandersetzen.

7.3.2.2 Synchronisation

Die Verarbeitung des Ereignisses produziert Ergebnisse: Datenbestände werden aktualisiert, Komponenten verändern ihren Zustand. Damit die Darstellung am Bildschirm letztlich diese Veränderungen widerspiegelt, ist es erforderlich, die jeweils ereignisauslösende Komponente mit ihrer ereignisverarbeitenden Komponente zu synchronisieren. Auch die Synchronisation durchläuft die gesamte Wertschöpfungskette – diesmal jedoch in umgekehrter Reihenfolge – mit dem Ergebnis einer veränderten Darstellung am Bildschirm. Abbildung 23 deutet die Aufgabenverteilung zwischen zwei beteiligten Komponenten in Form von Methoden an, die wiederum lediglich als Vorlage für tatsächlich implementierbare Interfaces zu verstehen sind.

Die Initiative zum Anstoß der Synchronisation übernimmt einer der beiden Partner:

- In der ersten Variante bestimmt die ereignisauslösende Komponente den Zeitpunkt der Synchronisation. Dieser Zeitpunkt ergibt sich bei einem synchronen Ablauf durch das Warten bis zur Rückkehr der Methode, durch die die auslösende Komponente die Ereignisverarbeitung bewirkt hat (siehe die obigen Varianten zu *processEvent*). In der Schnittstelle der Präsentation erübrigt sich dann die *synchronize*-Methode. Diese Lösung funktioniert nur, wenn die Ereignisverarbeitung zum Synchronisationszeitpunkt tatsächlich abgeschlossen ist.
- In der zweiten Variante stößt die ereignisverarbeitende Komponente nach erfolgter Verarbeitung die Synchronisation selbst an. Dabei gibt es zwei Möglichkeiten: Entweder sie ruft direkt eine oder mehrere Methoden in der Schnittstelle der ereignisauslösenden Komponente auf (siehe die Methode *synchronize* in der Abbildung); sie kennt dann die ereignisauslösende Komponente direkt. Oder die ereignisauslösende Komponente registriert sich im Sinne eines EventListeners bei der ereignisverarbeitenden Komponente, um über deren Daten- und Zustandsänderungen benachrichtigt zu werden; dann kennt die ereignisverarbeitende Komponente die ereignisauslösende nicht direkt.

Die ereignisauslösende Komponente benötigt im Zuge der Synchronisation den Zugriff auf die Ergebnisse der Ereignisverarbeitung, also auf veränderte Daten und Zustände der ereignisverarbeitenden Komponente. Wiederum lässt sich dies auf zweierlei Wegen erreichen:

- Die ereignisauslösende Komponente fordert Daten und Zustandsinformationen bei der ereignisverarbeitenden Komponente an, sobald sie sie benötigt *(getDataForSynchronization, getStateForSynchronization)*.
- Die ereignisverarbeitende Komponente liefert Daten und Zustandsinformationen automatisch mit, und zwar entweder als Rückgabewert von *processEvent* (wenn die ereignisauslösende Komponente den Anstoß zur Synchronisation gibt) oder als Parameter von *synchronize* (wenn die ereignisverarbeitende Komponente den Anstoß zur Synchronisation gibt).

1.
GUI-Bibliothek als ereignisauslösende, Präsentation als ereignisverarbeitende Komponente

Wie hier hinsichtlich des Anstoßes zur Synchronisation verfahren wird, hängt von der konkreten GUI-Bibliothek ab. Swing stößt nach der Abarbeitung eines *EventListeners* nicht automatisch eine Aktualisierung der GUI an, die Präsentation muss selbst dafür sorgen, dass die Darstellung geeignet verändert und mit den aktuellen Werten versehen wird. Sie kann sich hierfür unter anderem der *Models* bedienen, die vielen Bedienelementen zugrunde liegen; Änderungen an den Models (z.B. Hinzufügen von Datenzeilen für eine Tabelle) bewirken automatisch die Anpassung der Darstellung. Ein Servlet hingegen stößt aufgrund der Request-Response-Semantik des HTTP-Protokolls stets automatisch die Aktualisierung der Anzeige nach Beendigung von *doGet* bzw. *doPost* an.

In beiden Fällen werden veränderte Daten von der Präsentation aktiv bereitgestellt, die fremdentwickelten GUI-Bibliotheken können sie in Unkenntnis der Präsentationsschnittstelle nicht explizit abfragen.

2.
Präsentation als ereignisauslösende, Dialogkern als ereignisverarbeitende Komponente

Die Präsentation kann ihre Synchronisation selbst anstoßen, nachdem die Ereignisverarbeitung des Dialogkerns abgeschlossen ist, die sie mit *processEvent* ausgelöst hat. Umgekehrt kann der Dialogkern, wenn die Ereignisverarbeitung abgeschlossen ist, die Präsentation auffordern, ihre Darstellung zu aktualisieren (entspricht dem Aufruf von *synchronize*). Eine besondere Ausprägung der zweiten Variante besteht in der Anwendung des Beobachter-Prinzips, wie bei MVC: Eine oder mehrere Views registrieren sich beim Model als Beobachter, werden von diesem automatisch über Datenänderungen benachrichtigt und aktualisieren daraufhin ihre Darstellung. Auf unsere Komponenten übertragen bedeutet das: Die Präsentation wird zum Beobachter des Dialoggedächtnisses und synchronisiert sich automatisch als Reaktion auf Datenänderungen.

Das eigentliche Aktualisieren der Bedienelemente (welcher Datenwert wird in welches Feld geschrieben etc.) kann man z.B. in der Präsentation explizit auskodieren. Es gibt aber auch eine Ausgestaltung des Beobachter-Prinzips, die auf die bereits angesprochenen Value-Models aus Smalltalk zurückgeht und bei der die Aktualisierung nicht explizit auskodiert wird: Ein Value-Model fungiert in *beiden* Richtungen als Adapter zwischen ausgewählten Bedienelementen und einem zugrundeliegenden Datenobjekt/-wert: Eingabewerte wandern nicht nur automatisch aus Feldern in Datenobjekte, Änderungen in Datenobjekten werden auch automatich in den assoziierten Feldern sichtbar. Anstatt eine Methode zu schreiben, die Werte in Felder schreibt, verbindet man hier also Felder und Daten über Adapter, die dann den Rest erledigen.

3.
Dialogkern als ereignisauslösende, Anwendungskern als ereignisverarbeitende Komponente

Hat der Dialogkern Funktionen des Anwendungskerns aufgerufen, so muss er sich danach ggf. auf veränderte Daten einstellen. Häufig wird der Anwendungskern diese veränderten Daten bereits als Rückgabewerte seiner Funktionen mitliefern, zusätzlich steht dem Dialogkern aber auch die Möglichkeit offen, die Datenlage durch zusätzliche Anwendungskern-Funktionen zu erfragen (etwa weil die ursprüngliche Anwendungskern-Funktion umfangreichere Seiteneffekte produziert haben könnte).

7.4 Spezielle Themen

7.4.1 Sitzung

In Mehrbenutzersystemen braucht man den Begriff der *Sitzung* (vgl. Abschnitt 7.2). Die Sitzung ist nun allerdings kein für Benutzerschnittstellen spezifisches Konzept, sondern sie wird von verschiedenen Komponenten, vor allem auch dem Anwendungskern selbst benötigt. Relevant ist der Sitzungsbegriff für die Benutzerschnittstelle in dreierlei Hinsicht:

- Die Sitzung ermöglicht den Zugriff auf Informationen über den betroffenen Benutzer.
- Informationen von dialogübergreifendem Charakter können sitzungsweit verfügbar gemacht werden.
- Manche GUI-Bibliotheken bringen bereits einen technisch motivierten Sitzungsbegriff mit.

Die folgenden benutzerbezogenen Informationen sind im Zusammenhang mit Benutzerschnittstellen von besonderem Interesse:

- In Benutzerprofilen können sich unter anderem Informationen über individuelle Voreinstellungen bezüglich Aussehen und Verhalten von Benutzerschnittstellen befinden, denen Präsentation und Steuerung Rechnung tragen müssen.
- Beim Umgang mit fachlichen Daten und Funktionen sind u. U. auch benutzerindividuelle Berechtigungen zu berücksichtigen.
- Häufig gehört es zur Aufgabe der Präsentation, die visuelle Darstellung in der Sprache des jeweiligen Benutzers aufzubereiten *(Lokalisierung, Internationalisierung)*. Zu diesem Zweck benötigt sie den Zugriff auf dessen Lokalisierungsinformationen.

Im Zusammenhang mit der sitzungsweit sichtbaren Ablage von Informationen definieren wir in Analogie zu Präsentation und Dialogkern die folgenden Begriffe:

- *Sitzungsdaten*
  Fachliche Datenobjekte, die in der Sitzung dialogübergreifend festgehalten werden, heißen Sitzungsdaten.

- *Sitzungszustand*
  Über fachliche Daten hinausgehende Zustandsinformationen, die in der Sitzung dialogübergreifend festgehalten werden, nennen wir zusammenfassend Sitzungszustand.

- *Sitzungsgedächtnis*
  Sitzungsdaten und Sitzungszustand bilden zusammen das Sitzungsgedächtnis.

Der Zusammenhang zwischen dem hier verwendeten Sitzungsbegriff und etwaigen technischen Sitzungsbegriffen der eingesetzten GUI-Bibliotheken lässt sich wie folgt beschreiben:

- Der hier verwendete logische Sitzungsbegriff existiert zunächst unabhängig von einem technischen Sitzungsbegriff. Die Sitzung beginnt, wie zu Beginn des Kapitels definiert, mit der Anmeldung des Benutzers und endet mit dessen Abmeldung bzw. dem Abbruch im Fehlerfall. Jegliche Aktivitäten in der Benutzerschnittstelle bis zur erfolgten Anmeldung (also insbesondere die Anzeige eines Anmeldedialogs) erfolgt in einem sitzungsfreien Raum.
- Auch wenn die GUI-Bibliothek ein technisches Sitzungskonzept mitbringt (z.B. die HTTP-Session in HTML/Servlet-basierten Kontexten), bleibt der logische Sitzungsbegriff maßgeblich für die GUI-Architektur[10]. Die logische Sitzung selbst stellt dann bei Bedarf den Zusammenhang zur technischen Sitzung her; sinnvollerweise hält sie diese als Teil ihres Sitzungszustands.

7.4.2 Validierung von Benutzereingaben

Dialoge, die der Veränderung bzw. Neuanlage von Daten dienen, müssen sicherstellen, dass das Speichern der Benutzereingaben nicht zu Inkonsistenzen im Datenbestand führt. Aus diesem Grund ist vor dem Speichern die Validierung der zu speichernden Daten erforderlich, also die Prüfung im Hinblick auf die Einhaltung formaler (z.B. Typprüfungen) und inhaltlicher (z.B. Kreuzplausibilitäten) Konsistenzbedingungen.

Ein wesentlicher Anteil dieser Prüfungen ergibt sich aus den fachlich bestimmten Zusicherungen, die die Anwendungslogik vorgibt. Prüfungen dieser Art gehören in den Anwendungskern, denn dort konzentriert sich das fachliche Wissen. Es gibt jedoch zwei Situationen, in denen Prüfungen im Dialog selbst stattfinden:

- Aus Gründen der Benutzerfreundlichkeit oder der Verbesserung von Antwortzeiten kann es erforderlich sein, Teile dieser fachlich bestimmten Prüfungen in den Dialog hineinzuziehen. Zur Absicherung der Konsistenz gilt allerdings folgende Regel: Der Anwendungskern nimmt grundsätzlich alle fachlichen Prüfungen vor, er verlässt sich nicht darauf, dass der Dialog bereits validiert hat. So wird sichergestellt, dass der Anwendungskern sowohl in verschiedenen Dialogen als auch im Batch einsetzbar ist.
  Der Dialog führt diese Validierungen lediglich *zusätzlich* noch einmal aus. Das bedeutet allerdings eine redundante Verteilung von Anwendungswissen über verschiedene Stellen der Anwendung. Konsistenz und Vollständigkeit dieses Wissens sind dann schwieriger zu kontrollieren.

[10] Möglicherweise existiert in der GUI-Bibliothek sogar bereits eine technische Sitzungsinstanz, bevor die logische Sitzung gemäß unserer Definition begonnen hat. Dies ist z.B. in Web-Anwendungen der Fall, bei denen schon mit dem ersten HTTP-Request eine HTTP-Session entsteht, ohne dass sich der Benutzer in der betroffenen Anwendung überhaupt anmelden konnte; gemäß unserem Sitzungsbegriff befinden wir uns zu diesem Zeitpunkt noch in einem sitzungsfreien Raum.

- Darüber hinaus gibt es Prüfungen, die ausschließlich der Dialog vornimmt. Ihr Ziel ist die korrekte Verwendung des Anwendungskerns. Der Anwendungskern gibt Konventionen vor, deren Einhaltung er zur Bedingung für die einwandfreie Erbringung seiner Dienste macht. Zwar muss er trotzdem auf unsachgemäße Benutzung vorbereitet sein, jedoch nur in dem Sinne, dass er eine geeignete Fehlerbehandlung auslöst (siehe dazu den nachfolgenden Abschnitt); die Erbringung des Dienstes jedoch ist bei Verletzung der Konventionen nicht mehr garantiert. In einem Reisebuchungssystem etwa ist der Aufruf einer Funktion *storniereFlugbuchung* möglicherweise nur dann zulässig, wenn die Buchung nicht bereits storniert ist und der Passagier den Flug noch nicht angetreten hat. Durch Abprüfen derartiger Bedingungen und z.B. auch durch Typprüfungen bei der Umwandlung von Transportdaten in Originaldaten stellt ein leistungsfähiger Dialog sicher, dass es gar nicht erst zur Fehlbenutzung des Anwendungskerns kommt.

Unabhängig davon, um welche der beiden Arten von Validierungen es sich handelt, sind folgende Aspekte zu beachten:

- Jedem Dialog stehen grundsätzlich zwei Wege offen, die Konsistenz von Eingaben sicherzustellen: Einerseits durch nachträgliche Prüfung bereits vorhandener Eingaben (etwa beim Abspeichern dieser Eingaben), andererseits durch präventive Validierungen. Letztere meinen eine Gestaltung und Steuerung der Benutzerschnittstelle mit dem Ziel, Fehleingaben gar nicht erst entstehen zu lassen (Beispiel: Wenn ein bis-Feld nur dann gefüllt werden darf, wenn auch das von-Feld gefüllt ist, wäre es möglich, das bis-Feld so lange für Eingaben zu sperren, bis das von-Feld einen Wert erhalten hat). Im Wortsinn sind diese Fehlerverhinderungsmaßnahmen natürlich eigentlich keine Validierungen, aber sie verwenden vergleichbares Wissen.
- Von der GUI vorgenommene Validierungen beschränken sich auf diejenigen Informationen, die den Benutzereingaben zu entnehmen sind. Weitergehender Informationsbedarf würde ohnehin den Weg durch den Anwendungskern erforderlich machen.
- Validierungen können sowohl von der Präsentation als auch vom Dialogkern vorgenommen werden. Die Präsentation operiert dabei auf Ebene von Feldinhalten (z.B. feldbezogene Typprüfungen; möglicherweise bietet sogar die verwendete GUI-Bibliothek bereits typisierbare Felder an) bzw. auf Ebene der Präsentationsdaten. Der Dialogkern führt seine Validierungen auf den Dialogdaten durch; eine geeignete Stelle für die Durchführung der Prüfungen sind die Dialogaktionen. Die Verteilung von Validierungen auf Präsentation und Dialogkern orientiert sich an folgender Regel: Syntaktische Prüfungen (z.B. Typprüfungen) werden von der Präsentation durchgeführt, Prüfungen, die Anwendungswissen erfordern, gehören in den Dialogkern.

7.4.3 Fehlerbehandlung

Der Benutzerschnittstelle einer Anwendung kommt eine besondere Rolle zu, wenn es um die Behandlung von Fehlersituationen geht. Während ein Dialog geöffnet ist, können an verschiedenen Stellen des Anwendungssystems Fehler auftreten, vor allem im Anwendungskern und auch in der GUI selbst. Diese Fehler können einerseits fachlicher Natur sein (z.B. Validierungsfehler durch Fehleingaben des Benutzers, die erfolglose Suche nach einem Datenobjekt oder das oben erwähnte Auslösen eines unzulässigen Dialogereignisses), andererseits aber auch technische Ursachen haben (z.B. der Absturz der Datenbank oder das Zusammenbrechen einer Kommunikationsverbindung). Viele dieser Fehler, vor allem solche mit fachlichem Charakter, können vom Dialog vorhergesehen und deshalb definiert behandelt werden: mit dem Übergang in einen bzw. dem Verbleib in einem definierten Zustand sowie ggf. einer Mitteilung an den Benutzer. Aber auch manche der nicht vorhergesehenen Fehler lassen möglicherweise noch eine definierte Reaktion (und sei es nur eine Meldung) zu, andere hingegen führen ggf. unweigerlich zum Crash der Anwendung[11].

Unabhängig davon, wo die Fehler entstehen, empfiehlt es sich, sie über die gesamte Anwendung hinweg einheitlich zu behandeln und sie dem Benutzer gegenüber immer in der gleichen Weise zu melden. Dies gilt gerade auch für Validierungsfehler, für die in der Praxis schnell einmal in verschiedenen Dialoge derselben Anwendung hinweg die unterschiedlichsten Darstellungsverfahren zu beobachten sind.

Hilfreich zur Schaffung dieser Einheitlichkeit kann es sein, im Dialog alle Fehler mit einem einheitlichen Datenformat zu repräsentieren, unabhängig davon, ob sie z.B. aus dem Anwendungskern oder dem Dialog selbst stammen. Hier schlagen wir folgendes Modell vor: Fehler können von einem Dialog grundsätzlich genauso behandelt werden wie die fachliche Daten, die er anzeigen soll. Der Dialogkern verwaltet Fehler dann ganz analog zu seinen Dialogdaten, nämlich als anzuzeigende Objekte, die er in einem bestimmten Format vorliegen hat. Tritt im Zusammenspiel mit dem Anwendungskern in letzterem ein Fehler auf, so wandelt der Dialogkern diesen Fehler – genau wie er es ggf. mit den fachlichen Datenobjekten tut – in das von ihm präferierte Format um. Der Präsentation kommt dann entsprechend die Aufgabe zu, die Fehlerobjekte mit den Mitteln der GUI-Bibliothek geeignet darzustellen – analog zur Anzeige der eigentlichen Dialogdaten.

[11] Vgl. auch die Diskussion *konsistenter/inkonsistenter Situationen* in Quasar Teil 1.

7.4.4 Komposition von Dialogen

Moderne GUI-Bibliotheken bieten in der Regel Konzepte an, die die hierarchische Zusammenstellung von Benutzeroberflächen aus vorgefertigten und individuell abwandelbaren Bausteinen ermöglichen. Man denke etwa an den Komponentenbegriff in AWT/Swing oder die mittels Includes realisierbare Schachtelung von JSP. In dieser Weise verstandene GUI-Bausteine beschränken sich auf den sichtbaren Teil eines Dialogs.

Man kann den Gedanken des GUI-Bausteins aber auch weiter fassen: im Sinne einer vollständig funktionierenden Einheit, die neben einer sichtbaren Darstellung auch ihr komplettes Verhalten mitbringt, also Daten und Zustände verwaltet, Ereignisse verarbeitet und u.U. sogar Anwendungskernfunktionalitäten anspricht. In den Begriffen unserer Standardarchitektur ausgedrückt heißt das: Ein solcher Baustein muss einen kompletten Ausschnitt eines Dialogs quer durch die Komponenten Präsentation und Dialogkern umfassen, der für einen bestimmten Teil der möglichen Präsentationsereignisse und korrespondierenden Dialogereignisse die Verarbeitung und Synchronisation übernimmt.
Ein kompletter Dialog lässt sich dann aus solchen Teil-Dialogen oder Dialogbausteinen hierarchisch komponieren.
Es ist klar, dass in einem solchen Szenario die Kommunikationsprotokolle erweitert werden müssen:

- Das Zusammenspiel der Präsentation und des Dialogkerns wird zum Zusammenspiel von korrespondierenden Präsentations- und Dialogkernbausteinen.
- Innerhalb der Hierarchien aus Präsentationsbausteinen bzw. Dialogkernbausteinen muss eine geeignete Arbeitsteilung zwischen den Bestandteilen sichergestellt werden: Übergeordnete Bausteine geben z.B. die Verantwortung für Teile ihrer (Dialog- bzw. Präsentations-)Daten an untergeordnete Bausteine weiter und propagieren u.U. empfangene Ereignisse an untergeordnete Knoten weiter (beim Speichern des gesamten Objekts eines Dialogs etwa müssen möglicherweise auch die Teilobjekte untergeordneter Dialogbausteine gespeichert werden).

Ein solches Konzept eröffnet neue Möglichkeiten der Wiederverwendung im Zusammenhang mit Benutzerschnittstellen. Mit ihm lassen sich Sammlungen von Bausteinen anlegen, die vollständig generischer Natur sind (z.B. ein voll parametrierbarer Suchdialog) oder auch – nicht zu spezielles – fachliches Wissen enthalten (z.B. ein Baustein zur Anzeige von Adressen).

Wir wollen an dieser Stelle keine detaillierte Diskussion der verschiedenen Aspekte führen, die hier zu bedenken sind. Mit den Quasar-Views liegen Architektur und Implementierung eines GUI-Baustein-Konzepts vor, das die beschriebene Strategie umsetzt.

7.5 Realisierungsdetails

Dieser Abschnitt arbeitet Details hinsichtlich der Realisierung der beiden Komponenten Dialogkern und Präsentation heraus.

7.5.1 Dialogkern

Die zentrale Instanz innerhalb des Dialogkerns ist die *Dialogsteuerung.* Die Dialogsteuerung erfüllt verschiedene Aufgaben:

1.
Verwaltung der Dialogdaten:
Die Dialogsteuerung verwaltet die Dialogdaten, also die im Dialog darzustellenden Datenobjekte. Dies können die fachlichen Originaldatenobjekte sein, wie der Anwendungskern sie liefert, in der Regel handelt es sich jedoch eher um eine Transportstruktur, die speziell auf die Belange der Benutzerschnittstelle zugeschnitten ist (d.h. nur die in der Benutzerschnittstelle tatsächlich benötigten Datenwerte enthält). Die Dialogdaten können entweder direkt für den Dialog erstellt worden sein – mehrere Dialoge arbeiten dann nicht auf denselben Datenobjekt-Instanzen – oder einem dialogübergreifenden Datenpool entstammen – verschiedene Dialoge können dann auf dieselben Dateninstanzen zugreifen. Die letztere Variante ermöglicht den dialogübergreifenden Einsatz des Beobachter-Prinzips aus MVC: Dialoge, die dasselbe Model (im Sinne der Datenobjekte) verwenden, können automatisch auf Datenänderungen reagieren, auch wenn sie sie selbst gar nicht bewirkt haben.

Liefert der Anwendungskern die Daten noch nicht in der Form, die der Dialogkern erwartet, so gehört die Umwandlung der Daten in beiden Richtungen ebenfalls zu den Aufgaben der Dialogsteuerung.

2.
Verwaltung des Dialogzustands:
Der Dialogkern durchläuft in Abhängigkeit von den Dialogschritten, die der Benutzer durchführt, unterschiedliche Dialogzustände. Die Verwaltung und Auswertung dieser Zustände ist Aufgabe der Dialogsteuerung. Zwei grundsätzliche Ausgestaltungsvarianten für Dialogzustände sind möglich:

- In Variante 1 werden Zustände explizit modelliert – etwa in Form einer Klasse, die einen Aufzählungstyp repräsentiert, oder als String-Konstanten. Die Dialogsteuerung verfügt dann sinnvollerweise über ein entsprechendes Attribut, in dem der aktuelle Zustand abgelegt wird.
- In Variante 2 wird der Zustand implizit als Summe der aktuell gültigen Attributwerte der Dialogsteuerung behandelt.

3.
Ereignisverarbeitung:

Die Dialogsteuerung braucht Verarbeitungswissen. Auf dieser Basis erfüllt sie die folgenden Aufgaben:

- Sie beurteilt, ob ein bestimmtes Dialogereignis im gegenwärtigen Dialogzustand überhaupt zulässig ist. Wenn etwa die Präsentation Menüeinträge auch dann aktiv lässt, wenn die dahinter stehende inhaltliche Aktion eigentlich zum gegenwärtigen Zeitpunkt gar nicht ausführbar ist, geht bei Auswahl eines solchen Menüpunkts ggf. ein ungültiges Dialogereignis bei der Dialogsteuerung ein. In diesem Fall führt sie eine geeignete Fehlerbehandlung durch (richtet z.B. eine entsprechende Meldung an den Benutzer).
- Sie wählt in Abhängigkeit von der Art des Dialogereignisses und dem aktuellen Dialogzustand geeignete Dialogaktionen aus und führt sie durch. Das gleiche Dialogereignis kann je nach Zustand zu unterschiedlichen Aktivitäten führen. Lautet das Dialogereignis etwa *Bearbeitung abbrechen,* so wird die Dialogsteuerung im Falle, dass der Benutzer Daten modifiziert hat, möglicherweise beim Benutzer rückfragen („Wollen Sie Ihre Eingaben wirklich verwerfen?"); sind die Eingaben hingegen unverändert geblieben, kann diese Nachfrage entfallen.
- Sie nimmt, falls erforderlich, einen Zustandswechsel vor.

Mögliche Dialogaktionen sind:

- eine Rückfrage beim Benutzer,
- dialogablaufbezogene Aktionen (z.B. Schließen des Dialogs, Öffnen eines Unterdialogs),
- der Aufruf einer oder mehrerer AK-Funktionalitäten (z.B. das Lesen oder Schreiben fachlicher Datenobjekte),
- die Synchronisation hinsichtlich der Ergebnisse des Anwendungskerns (Aktualisierung der Dialogdaten bzgl. veränderter Originaldaten).

Das Ereignisverarbeitungswissen lässt sich mit unterschiedlichen Mitteln darstellen, nachfolgend einige Varianten, die einzeln oder auch in Kombination eingesetzt werden können:

- *Ausprogrammiert:* Im einfachsten Fall kodiert die Dialogsteuerung das gesamte Wissen zentral in einer einzigen Funktion explizit aus. Dies empfiehlt sich jedoch höchstens in kleinen Dialogen mit wenigen Zuständen und Zustandswechseln. In größeren Dialogen besteht schnell die Gefahr, dass aufgrund der kombinatorischen Möglichkeiten tief verschachtelte if-then-else-Konstrukte entstehen, die kaum zu verstehen und noch weniger zu warten sind.
- *Zustandsautomat:* Ebenfalls zentral zusammengeführt wird das Verarbeitungswissen bei Verwendung eines *Interaktionsdiagramms (IAD)* oder eines vergleichbaren Modells, das Zustandsübergänge und auszulösende Aktionen als Folge des Eintreffens bestimmter Ereignisse bei Vorliegen bestimmter Dialogzustände beschreibt. Ein solches Modell wird entweder auskodiert oder in einer geeigneten Beschreibungssprache (z.B. XML) formuliert und bei Bedarf aus einer Konfigurationsdatei eingelesen. Die Dialogsteuerung selbst wird dadurch zu einem generisch arbeitenden Modell-Interpreter, der ohne konkretes Anwendungswissen auskommt; das Anwendungswissen steckt im Modell. Auch hier gilt allerdings: Mit steigender Anzahl von Zuständen und Zustandsübergängen leidet u.U. die Übersichtlichkeit und Handhabbarkeit des Modells. Als Faustregel kann man sagen, dass ein IAD nicht mehr als sieben Zustände haben soll. Dies lässt sich nur erreichen, wenn man Dialogzustand und Präsentationszustand auseinander hält. Request-Response-orientierte Anwendungen eignen sich besonders gut für den Einsatz von Zustandsautomaten.
- *Regelbasiert:* Bei diesem Ansatz wird das Wissen in Form von Regeln formuliert („Wenn Ereignis a eintritt und Voraussetzungen b und c vorliegen, dann führe Aktion x durch"). Ein eintreffendes Ereignis bewirkt die Anwendung einer oder mehrerer Regeln, deren Vorbedingungsteil erfüllt ist. Dieser Ansatz hat gegenüber der Zustandsmaschine vor allem dann Vorteile, wenn die Anwendung durch sehr feingranulare Regelungen geprägt ist, die in einem Zustandsautomaten die Menge der Zustände und Zustandsübergänge explodieren ließen.
- *Kommandos:* In dieser Variante zerlegt die Dialogsteuerung das Wissen in einzelne *Kommandos.* Ein Kommando ist ein Bündel aller Aktivitäten, die bei Auftreten eines bestimmten Dialogereignisses ausgeführt werden müssen. Die Dialogsteuerung wählt bei Eintreffen eines Dialogereignisses das geeignete Kommando aus und führt es durch („Dispatch"). Ein Kommando zieht ggf. den aktuellen Dialogzustand in Betracht und verzweigt abhängig davon zu unterschiedlichen Handlungssträngen.

7.5.2 Präsentation

Die Präsentation umfasst zwei wesentliche Bestandteile: die visuelle Repräsentation und die Präsentationssteuerung.

Die *visuelle Repräsentation (View)* ist das, was der Benutzer tatsächlich zu sehen bekommt. Sie wird mit den Gestaltungsmitteln der jeweils verwendeten GUI-Bibliothek konstruiert. Unter Swing hat die visuelle Repräsentation etwa die Form eines Panels, eines Fensters oder sogar einer Menge von Fenstern, gefüllt mit den benötigten Bedienelementen. In Web-Anwendungen handelt es sich um HTML-Seiten bzw. JSP, ggf. angereichert um Skript-Code in JavaScript oder vergleichbaren Skriptsprachen; andere GUI-Bibliotheken wie wingS oder Swinglets verbergen die HTML-Struktur hinter einem an Swing angelehnten, widget-orientierten Programmiermodell.

Die *Präsentationssteuerung* erfüllt mehrere Aufgaben:

1.
Bereitstellung der visuellen Repräsentation
Die Präsentationssteuerung sorgt dafür, dass eine geeignete visuelle Repräsentation vorhanden ist und modifiziert sie – sofern erforderlich – im Zeitablauf. Dazu gehört insbesondere auch das Füllen der visuellen Repräsentation mit den darzustellenden Daten. Darüber hinaus ist hier das Thema Lokalisierung anzusiedeln: Statische (z.B. Feldnebenschriften) wie auch dynamische (z.B. Feldformate, etwa für Datumsangaben) Bestandteile der Darstellung sind Gegenstand der Lokalisierung durch die Präsentationssteuerung; die Lokalisierung der Anwendungsdaten selbst fällt hingegen nicht in ihre Zuständigkeit.

2.
Verwaltung der Präsentationsdaten
Die Präsentationssteuerung kann, sofern die verwendete GUI-Bibliothek dies erfordert, die von der Dialogsteuerung erhaltenen Daten in eine geeignete Repräsentation überführen (die Präsentationsdaten). So werden z.B. in Swing den Bedienelementen sogenannte *Models* als Datenstrukturen zugrundegelegt (im Sinne des MVC-Ansatzes). Die Umwandlung der Dialogdaten in Präsentationsdaten und umgekehrt ist ebenfalls Aufgabe der Präsentationssteuerung.

3.
Verwaltung des Präsentationszustands
Besteht die Ereignisverarbeitung der Präsentation nicht ausschließlich aus dem Weiterleiten der Ereignisse an den Dialogkern, d.h. verarbeitet sie auch Ereignisse selbst, dann benötigt sie eigene Zustandsinformationen. Die Verwaltung und Auswertung dieser Zustände ist dann Aufgabe der Präsentationssteuerung.

4.
Ereignisverarbeitung
Die Präsentationssteuerung nutzt den Ereignis-Mechanismus der jeweiligen GUI-Bibliothek, um Präsentationsereignisse zu empfangen (also z.B. Event-Listener in Swing, Auswertung des HTTP-Requests im Servlet-Umfeld).

Die Verarbeitung von Präsentationsereignissen durch die Präsentationssteuerung ist analog zur Verarbeitung von Dialogereignissen durch die Dialogsteuerung und kann daher mit den gleichen Mitteln gelöst werden. Auch die Präsentationssteuerung benötigt Verarbeitungswissen, und für die Abbildung dieses Wissen stehen grundsätzlich die gleichen Realisierungsvarianten zur Verfügung wie bei der Dialogsteuerung (z.B. Kommandos oder Regeln).

Wie hoch die Ansprüche an die Ereignisverarbeitung der Präsentation sind, hängt von der Art der Anwendung ab: In hochinteraktiven Anwendungen, die differenziert auf Benutzereingaben reagieren (z.B. Ausgrauen eines Buttons, sobald aus einem Textfeld das letzte Zeichen entfernt wird), gibt es typischerweise viele Präsentationsereignisse, die direkt in der Präsentation verarbeitet werden, weil sie nur die Darstellung, nicht jedoch den grundsätzlichen Dialogablauf oder gar den Anwendungskern betreffen. In einem solchen Szenario spielt die Ereignisverarbeitung in der Präsentation eine große Rolle. Mit einem regelbasierten Ansatz ist die Komplexität der Steuerung besonders gut zu bewältigen. Request-Repsonse-orientierte HTML-Anwendungen hingegen verfügen aufgrund der eingeschränkten technischen Möglichkeiten nicht über eine derartig hohe Interaktivität. In den meisten Fällen bildet die Präsentationssteuerung dort Präsentationsereignisse direkt auf Dialogereignisse ab und leitet diese an den Dialogkern weiter.

Für die Realisierung von Dialogereignissen gibt es grundsätzlich zwei Alternativen:

- Sind sie explizit modelliert (z.B. in Form von Klassen), so gibt die Präsentationssteuerung, wenn sie die Ereignisverarbeitung im Dialogkern anstößt, jeweils eine Ereignisinstanz an die Dialogsteuerung weiter, die diese dann als Schlüssel zum Auffinden der geeigneten Aktivitäten verwendet.
- Sind sie nicht explizit modelliert, so ähnelt der Anstoß der Ereignisverarbeitung eher dem Aufruf von Anwendungskern-Funktionen durch den Dialogkern: Die Präsentationssteuerung bewirkt dann z.B. direkt die Durchführung eines bestimmten Kommandos oder den Aufruf einer Verarbeitungsfunktion.

Die Eingaben des Benutzers entnimmt die Präsentationssteuerung der visuellen Repräsentation. Für die Weiterleitung dieser Daten an den Dialogkern bestehen zwei grundsätzliche Möglichkeiten:

- Die Präsentationssteuerung kann die Benutzereingaben, nachdem sie diese der visuellen Repräsentation entnommen hat, als Anhang eines expliziten Dialogereignisses oder als Parameter eines entsprechenden Funktionsaufrufs mit an die Dialogsteuerung übertragen. Alternativ kann sie aber auch mit einer Referenz auf die Dialogdaten versehen werden und letztere dann selbst aktualisieren. In beiden Varianten stellt die Präsentationssteuerung sicher, dass alle benötigten Daten verfügbar sind. Dies kann sie im einfachen Fall dadurch erreichen, dass sie vorsorglich immer alle Eingabewerte überträgt. Soll sie jedoch gezielt nur diejenigen Eingabewerte liefern, die tatsächlich zur Verarbeitung des Ereignisses erforderlich sind, benötigt die Präsentationssteuerung Wissen darüber, welche Eingaben von einem bestimmten Ereignis betroffen sind.
- Umgekehrt kann die Dialogsteuerung im Rahmen der Durchführung eines Ablaufplans aber auch selbst den Anstoß dafür geben, dass die benötigten Eingabewerte in die Dialogdaten übertragen werden, indem sie die Präsentationssteuerung dazu auffordert. Anmerkung: Dass die Dialogsteuerung sich aktiv an die Präsentationssteuerung wendet, bedeutet nicht, dass sie deshalb auch über T-Wissen bzgl. der verwendeten GUI-Bibliothek verfügen muss. Man wird den Aufruf der entsprechenden Methode der Ereignisverarbeitung sinnvollerweise über eine technikneutrale Schnittstelle der Präsentationssteuerung vornehmen, denn es reicht für die Dialogsteuerung aus, der Präsentationssteuerung mitzuteilen, *dass* sie die Leistung erbringen soll; *wie* sie sie auf technischem Wege erbringt, ist für den Dialogkern unerheblich.

5.
Synchronisation

Die Initiative zur Synchronisation nach erfolgter Verarbeitung eines Dialogereignisses kann gemäß unserem obigen Kommunikationsmodell von der Präsentation oder vom Dialogkern ausgehen. In letzterem Fall gibt die Dialogsteuerung eine entsprechende Anweisung über eine technikneutrale Schnittstelle der Präsentationssteuerung, so dass sie selbst keine Gefahr läuft, technische Annahmen zu machen. Die Präsentationssteuerung setzt die technikneutrale Anweisung dann mit den Mitteln der GUI-Bibliothek um (Beispiel: Die Dialogsteuerungs-Anweisung „Ermögliche die Bearbeitung des Objekts" könnte sie in Swing etwa durch Freigabe der betroffenen Felder zum Editieren realisieren).

Die Synchronisation selbst bedeutet für die Präsentationssteuerung zweierlei:

- Ihre Präsentationsdaten und ihr Zustand – sofern vorhanden – werden an die Dialogdaten und den Dialogzustand angepasst, die möglicherweise infolge der Verarbeitung eines Dialogereignisses im Dialogkern verändert wurden.
- Sie aktualisiert die visuelle Repräsentation, indem sie veränderte Dialog- bzw. Präsentationsdaten sichtbar macht und dafür sorgt, dass veränderte Dialog- bzw. Präsentationszustände sich in der sichtbaren Darstellung widerspiegeln. Ein veränderter Zustand kann Auswirkungen auf die Struktur der Darstellung haben (Beispiel: In einem Suchdialog erscheint in einem zuvor leeren Bereich nach Ausführen der Suche eine Trefferliste); bestehende Bedienelemente können verändert werden (Beispiel: vorher nur lesbare Felder werden zum Editieren geöffnet); die Menge der aktuell verarbeitbaren Dialogereignisse kann sich in der Verfügbarkeit von auslösenden Bedienelementen ausdrücken (Beispiel: Schaltflächen, durch die Ereignisse ausgelöst würden, die im gegenwärtigen Dialogzustand nicht zulässig sind, werden inaktiv dargestellt oder überhaupt nicht angezeigt).

8 Verteilte Anwendungen

8.1 Einführung

Dieses Kapitel beschreibt die Quasar-Prinzipien und die Quasar-Starndardarchitektur für verteilte Systeme. Dies ist keinesfalls eine Konkurrenzveranstaltung zu Produkten wie EJB oder Normen wie CORBA, sondern wir beschreiben eine Leitlinie für den sicheren Umgang mit diesen Konzepten. Grundlage aller Überlegungen ist wie immer die Trennung der Zuständigkeiten: Wie kann man die eigentliche Anwendung von den Aspekten der Verteilung möglichst perfekt separieren?

Die neuesten Systeme sind in irgendeiner Weise verteilt. Oft denkt man dabei an Client/Server-Systeme, wo der Client mit dem entfernten Server kommuniziert. Das ist aber nur ein Spezialfall: Es ist unerheblich, ob es sich um Client- bzw. Serverkomponenten oder angebundene Nachbarsysteme handelt – jeder *prozessübergreifenden* Kommunikation liegt ein allgemeines Schema zugrunde. Darum geht es in diesem Kapitel, wobei der Schwerpunkt allerdings auf der *rechnerübergreifenden* Kommunikation liegt.

Prozess- und rechnerübergreifende Kommunikation findet immer nur über Schnittstellen statt, und Schnittstellen bestehen nur zwischen Komponenten. Wenn also eine Komponente auf mehrere Rechner zu verteilen ist, dann ist der einzige Weg die Aufteilung der Komponente in zwei Subkomponenten, die über eine definierte Schnittstelle kommunizieren.

Hinweis: Komponenten können andere Komponenten enthalten (vgl. Abschnitt 1.3.5). So enthält in Abbildung 24 die Komponente *k* die drei Subkomponenten h_1, h_2 und h_3. Dies ist die *innere* Struktur von *k;* sie ist an der Schnittstelle unsichtbar. Es ist möglich – und je nach Situation auch sinnvoll, die Subkomponenten h_1, h_2 und h_3 auf verschiedenen Rechnern laufen zu lassen.

Verteilung ist kein Selbstzweck, sondern Notwendigkeit. Kommunikation über Rechnergrenzen bedeutet immer erhöhte Komplexität und reduzierte Performance. Daher gilt der Grundsatz: So wenig Verteilung wie möglich, so viel wie nötig.

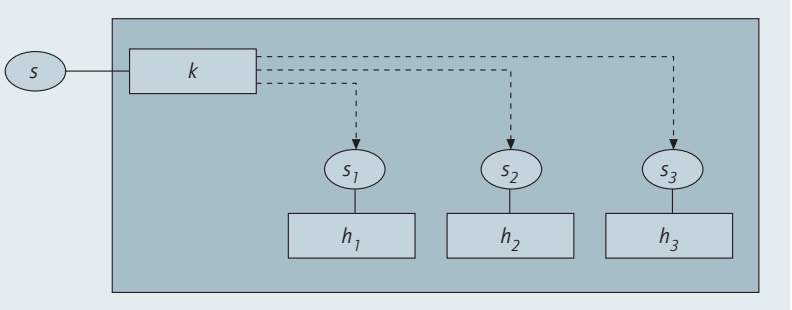


Abbildung 24: *k* als Fassade vor h_1, h_2 und h_3

8.1.1 Gründe für Verteilung

Software wird nur dann verteilt, wenn die Vorteile überwiegen. Gründe für eine Verteilung können sein:

1. Die Verteilung ist bereits vorgegeben, z.B. existieren verschiedene Nachbarsysteme auf unterschiedlichen Plattformen.
2. Eine Menge von Komponenten (z.B. Clients) nutzen eine gemeinsame und zentrale Logik.
3. Eine hohe Änderungshäufigkeit der Software verbunden mit einer kostenintensiven Distribution führt dazu, dass große Teile dieser Software zentral abgelegt werden müssen.
4. Verteilung ist nötig, um die Anwendung skalierbar zu machen. Steigt die Last und erschöpft sich dadurch die Kapazität einer Systemkomponente, dann wird die Software auf weitere Ressourcen (z.B. Serverpool) ausgelagert.
5. Die vorhandene Plattform bietet weniger an Funktionen und Leistung (z.B. spezielle Software- oder Hardware-Ausstattung) als die Anwendung braucht.
6. Eine schwache Leitungs-Infrastruktur zwingt dazu, bestimmte Verarbeitungen an bestimmten Lokationen zu platzieren. Die Entscheidung für einen Fat-Client kann z.B. bei schmaler Bandbreite sinnvoll sein, wenn dadurch die Netzlast reduziert wird.

8.1.2 Risiken der Verteilung

Jeder Fernaufruf erhöht das Risiko. Es ist wichtig, die Risiken zu kennen und sie gegen die Vorteile einer Verteilung abzuwägen:

1. Geringere Robustheit:
Bei Fernaufrufen können mehr Fehler auftreten als bei lokalen. Es sind mehr Komponenten involviert; das Netzwerk schafft zusätzliche Abhängigkeiten zur Hardware.

2. Erhöhter Realisierungsaufwand:
Bei Fernaufrufen bestimmt die verwendete Verbindungssoftware, welche Datentypen in der Schnittstelle zulässig sind. Dazu braucht man Software zur Konvertierung. Weiterhin ist das Finden und Aufrufen von entfernten Komponenten mit einem erhöhten Realisierungsaufwand verbunden.

3. Geringere Performance:
Entfernte Aufrufe sind um mehrere Größenordnungen teurer als lokale.

4. Zusätzliche Sicherheitsgesichtspunkte:
Wer darf eine Kommunikation auslösen? Kann die Verbindung abgehört werden?

5. Erhöhter Betriebsaufwand:
In einer verteilten Architektur sind viele technische Komponenten zusätzlich zu betreiben (Namensdienste, Verbindungssoftware, Netzwerke, Firewalls usw.). Dadurch wird das Gesamtsystem anfälliger für Störungen.

8.1.3 Verteilung planen

Die Software-Verteilung soll für die Entwickler und den späteren Nutzer transparent sein, aber für den Architekten ist sie es nicht: Die Performance-Nachteile schlecht geschnittener Software lassen sich auch durch geschicktes Feindesign kaum noch ausgleichen.

In frühen Entwurfsphasen sollte man sich stets fragen: „Wie reagiert mein System, wenn ich eine Komponente auf einen anderen Rechner lege?" Es ist wichtig zu prüfen, wie viele Aufrufe pro A-Fall an eine Komponente gesendet werden, und wie viele Daten dazu notwendig sind. Nur wenn eine Komponente hierfür eine schmale Schnittstelle zur Verfügung stellt, kann man sie später auch leicht über Rechnergrenzen ansprechen. Es kommt also darauf an, Verteilungsaspekte bei der Planung des Systems zu berücksichtigen, ohne eine bestimmte Verteilungsvariante in das System einzubetonieren.

Insgesamt gestalten wir Komponenten so, dass sie später im Prinzip beliebig verteilt werden können. Wir unterscheiden also zunächst nicht zwischen lokalen und entfernten Komponenten, sondern statten erst in einem zweiten Schritt die Komponenten, die entfernt laufen sollen, mit einer zusätzlichen Verteilungsschnittstelle aus. Damit hat man die Freiheit, in späteren Phasen die Verteilung nochmals ändern zu können, etwa weil neue Anforderungen andere Last erzeugen oder weil der erste Verteilungsentwurf nicht optimal war. Als angenehmer Nebeneffekt bekommt man die lokale Testbarkeit des Gesamtsystems geschenkt.

Insbesondere wird eine Komponente nicht um eine definierte Verteilungsschnittstelle herum gebaut. Auch vorgegebene Schnittstellen sind nur in einem definierten, möglichst kleinen Bereich sichtbar.

8.1.4 Verteilung kapseln

Um Verteilungstransparenz zu erreichen, wird jede Komponente *intern* so gebaut, als gäbe es keine Verteilung: Sie weiß nichts von CORBA oder RMI. Eigene Adapter (vgl. Abschnitt 1.4.2) befassen sich mit den Problemen, die aus der Verteilung resultieren. Adapter leisten folgendes:

1.
Erzeugen einer lokalen Repräsentation der entfernten Komponente.

2.
Beschaffung der Objektreferenz der entfernten Komponente. Diese Referenz wird ausschließlich in den Adaptern verwaltet und nicht in der rufenden Komponente bekannt gemacht.

3.
Fernaufruf organisieren.

4.
Transformation von Schnittstellenkategorien. Anwendungskomponenten realisieren stets A-Schnittstellen. Häufig bietet die Verbindungssoftware eigene Datentypen an (T-Schnittstelle), in welche die A-Schnittstelle der Komponente transformiert wird.

5.
Behandlung von konsistenten und inkonsistenten Situationen, die aus der Verteilung resultieren (z.B. Dienst in der angesprochenen Registratur nicht gefunden). Dabei ist zu beachten, dass Situationen, die der Adapter nicht klären kann (z.B. Nachbarsystem ist nicht verfügbar), an die aufrufende Komponente weitergereicht und dort behandelt werden.

Richtlinien für Fernaufrufe zwischen Komponenten

Bei moderner Verbindungssoftware sind Fernaufrufe meistens in ein komfortables API verpackt, so dass entfernte Kommunikationsvorgänge im Wesentlichen genauso aussehen wie lokale Prozedur- oder Methodenaufrufe. Dies darf aber nicht darüber hinwegtäuschen, dass Fernaufrufe viel komplexer und zeitaufwendiger sind als vergleichbare lokale Aufrufe.

Spätestens bei der Kommunikation zwischen zwei Rechnern fällt der Aufwand für die Durchführung des Fernaufrufs mehr ins Gewicht als die Geschwindigkeit der Kommunikation. Zwei einzelne Fernaufrufe, die jeweils einen Datenwert übertragen, verbrauchen insgesamt mehr Zeit als ein Fernaufruf, der beide Werte gleichzeitig überträgt. Deshalb gelten für effiziente Laufzeitumgebungen die folgenden Richtlinien:

1. Anzahl der Fernaufrufe minimieren:
Dazu orientieren sich die Fernaufrufe an den A-Fällen. Das Denken in A-Fällen ist der zentrale Gedanke beim Entwurf einer Verteilungsschnittstelle. Pro A-Fall gibt es genau einen Aufruf. Eine Kommunikation auf Attributebene von Objekten führt zu vielen Fernaufrufen und damit zu schlechter Performance. Auch wäre so die Innensicht der gerufenen Komponente sichtbar, was unseren Entwurfsgrundsätzen widerspricht.

2. Datenmenge reduzieren:
Die zu übertragende Datenmenge ist zu minimieren.

Die Minimierung der Anzahl der Aufrufe wirkt sich stärker auf die Performanz aus als die Reduktion der Menge der mit jedem einzelnen Aufruf übertragenen Daten. Deshalb ist es im Zweifelsfall besser, wenige Aufrufe mit großer Datenmenge durchzuführen.

8.1.5 Adapter steuern Fernaufrufe

Fernaufrufe werden von Adaptern durchgeführt. Im einfachsten Fall bietet der Adapter dafür eine A-Schnittstelle an (vgl. Abschnitt 1.4.2). Jeder Aufruf wird dann an die entfernte Komponente weitergeleitet.

Denkbar sind auch generische Ansätze: Ein Arbeitsbereich verwaltet alle Objekte lokal und erzeugt bei einer Änderungsanforderung Fernaufrufe, um die Objektgeflechte abzugleichen. Dabei kann es zu vielen und fein granularen Fernaufrufen kommen. Deshalb empfehlen wir, die Fernaufrufe zu bündeln, und dies kann der Adapter übernehmen. Egal ob und wie man bündelt – entscheidend ist, dass Verteilungsaspekte an zentraler Stelle gekapselt werden und sich nicht über den Anwendungskern verteilen.

Synchrone oder asynchrone Kommunikation

Bei einem synchronen Aufruf wird der Sender blockiert und ist passiv, bis die Rückmeldung des Empfängers eintrifft. Bei einem asynchronen Aufruf wird lediglich ein Datenpaket beim Empfänger abgegeben. Der Sender ist nicht blockiert: Er arbeitet weiter, und das Ergebnis kann zu beliebiger Zeit beim Sender ankommen.

Wir trennen zwischen Kommunikation aus Sicht der Anwendung (A-Kommunikation) und aus Sicht der Technik (T-Kommunikation):

A-Kommunikation:

Es wird festgelegt, ob die Komponente auf die Rückmeldung eines gerufenen Systems wartet oder nicht – sie kann aus Anwendungssicht erst weiterarbeiten, wenn das Ergebnis da ist. A-Kommunikation ist Sache der Anwendung.

T-Kommunikation:

Es wird festgelegt, mit welchem technischen Protokoll die A-Kommunikation stattfindet. Man kann durchaus einen aus Anwendungssicht synchronen Aufruf technisch asynchron realisieren und umgekehrt. Die T-Kommunikation ist vollständig in der Laufzeitumgebung gekapselt.

Beide Kommunikationsarten können asynchron oder synchron sein. Oft kommen sogar mehrere Formen in einem System zur Anwendung, z.B. synchrone T-Kommunikation zwischen Client und Server; asynchrone A-Kommunikation zur Versorgung von Nachbarsystemen.

Asynchrone Kommunikation mit Rückgabe (Ergebniswerten, Exceptions) beeinflusst die Struktur der rufenden Komponente: Asynchrone Aufrufe erhalten eine oder mehrere Callback-Methode(n) zur Übermittlung der Rückgabe.

Ein Beispiel verdeutlicht diesen Sachverhalt:

```
// Continuation anlegen

MessageController messageController = ..;
ContinuationManager continuationManager = ..;

Continuation continuation =
   new AbstractContinuation
   (continuationManager) {
      public void continueWithResult
      (Object result) {
         System.out.println(result);
      }
      public void continueWithException
      (Exception exception) {
         System.out.println(exception);
      }
   };
// Fernaufruf
messageController.call ("SynchronousHandler",
                        params, continuation);
```

Zunächst wird ein Continuation-Objekt mit zwei Methoden angelegt: eine für den regulären Ablauf und eine für Ausnahmen. Dem Aufruf *(call)* wird dieses Continuation-Objekt mitgegeben. Nach der Rückkehr vom Server wird eine der Continuation-Methoden ausgeführt. Es ist Aufgabe der Anwendung, den asynchronen Ablauf der Callback-Methoden mit dem restlichen Geschehen zu synchronisieren.

Transparenz von Verbindungen und das Design von Komponenten

Oft laufen mehrere Instanzen derselben Komponente gleichzeitig auf mehreren Rechnern. Mit moderner Verbindungssoftware kann man im technischen Adapter ohne Mühe sicherstellen, dass eine ausgefallene Instanz durch eine Verbindung zu einer Reserve-Instanz ersetzt wird. Damit das für den Aufrufer transparent geschieht, ist die gerufene Komponente auf ein solches Umschalten eingerichtet (vgl. Abschnitt 8.3).

Abbildung 25:
Die Transformation
von Datensichten

8.1.6 Schnittstellen zwischen verteilten Komponenten

Jede A-Komponente besitzt eine fachliche A-Schnittstelle, die fachliche Datentypen enthält. Diese Schnittstelle wird beim Design einer Komponente zuerst entworfen und fachlich spezifiziert. Soll die Komponente in eine verteilte Anwendung integriert werden, so muss die A-Schnittstelle in eine A_F- oder O-Schnittstelle transformiert werden (vgl. Abschnitt 1.4.2). Gründe für eine Transformation können sein:

1.
Die Verbindungssoftware verwendet andere Datentypen als die A-Schnittstelle der Komponente. Ein Beispiel hierfür sind die CORBA-Datentypen, die den Java-Datentypen nur ungefähr entsprechen.

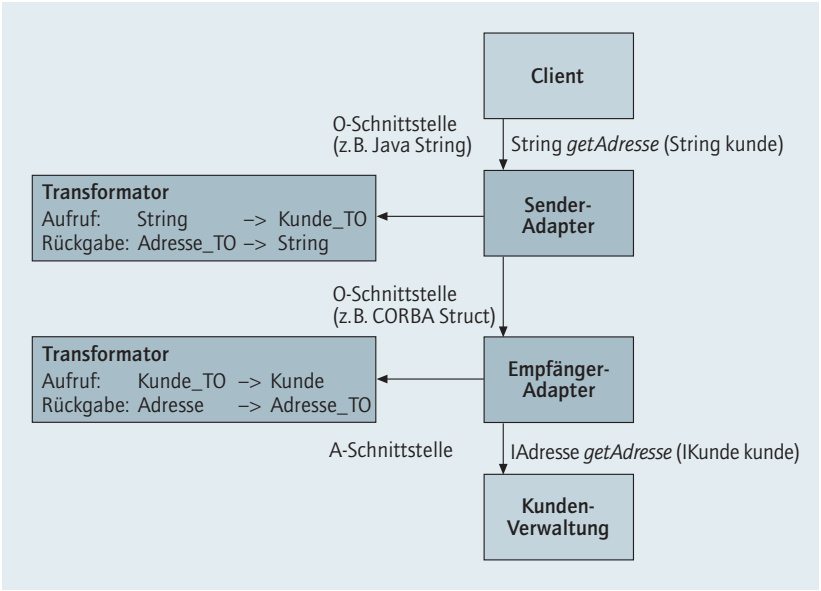
2.
Die zu übertragenden Datenstrukturen sollen so flexibel sein, dass eine Erweiterung ohne neues Übersetzen möglich ist.

3.
Die A-Fälle der entfernten Komponente erwarten andere Datentypen als die aufrufende Komponente. Das kann immer passieren (auch ohne Verteilung), besonders beim Zugriff auf Nachbarsysteme.

Der Grundsatz für die Transformation von Schnittstellen lautet: Die Transformation in bzw. aus einer externen Sicht ist immer die Aufgabe von Adaptern außerhalb der Komponente, nicht der Komponente selbst. Dabei kann es durchaus sinnvoll sein, die A-Schnittstelle einer Komponente auf mehrere externe Sichten abzubilden. Für jede Sicht ist ein eigener Adapter verantwortlich (vgl. Abbildung 25).

Wir empfehlen, die Anzahl der zwischen zwei verteilten Komponenten ausgetauschten Datentypen klein zu halten. Je mehr gemeinsame Datentypen es gibt, umso größer ist die Gefahr, dass bei Wartung bzw. Weiterentwicklung anfallende Änderungen nicht lokal bearbeitet werden, sondern alle beteiligten Komponenten betreffen.

Im einfachsten Fall würde man die Datentypen, die eine entfernte Komponente in ihrer A-Schnittstelle verwendet, auch dem lokalen Aufrufer anbieten. Dies widerspricht jedoch der Idee, die gemeinsamen Datentypen zu minimieren. Man überlegt sich besser, welche Typen tatsächlich lokal nötig sind und erzeugt diese Typen über Transformatoren in den Adaptern. Die zentrale Idee bei der Minimierung von ausgetauschten Datentypen besteht also darin, nur die von einer Komponente verwendeten Datentypen anzubieten, die der Client wirklich braucht.



Schnittstellen von Komponenten sollten stabil bleiben, auch wenn sich die Implementierung ändert. Für Verteilungsschnittstellen gilt:

- Bei Klassen- und Modul-Schnittstellen weist ein Compiler darauf hin, wenn nach einer Schnittstellenänderung die Nutzung einer Schnittstelle nicht mehr zur Definition passt. Bei Verteilungsschnittstellen treten solche Probleme meistens erst zur Laufzeit auf, da die Typsicherheit – wenn überhaupt – nur zwischen den Transformatoren gewährleistet ist. Eine mit CORBA-IDL spezifizierte Kommunikation zwischen Transformatoren wird vom IDL-Compiler geprüft. Die Transformation in den fachlichen Adaptern liegt aber außerhalb dieser Prüfung. Deren Korrektheit kann entweder durch den Compiler der eingesetzten Programmiersprache oder erst zur Laufzeit geprüft werden.
- Verteilungsschnittstellen werden nicht nur innerhalb einer einzelnen Anwendung genutzt, sondern auch über Anwendungs- oder Firmengrenzen hinweg. Eine Schnittstellenänderung kann also große organisatorische Probleme verursachen oder sie ist schlicht unmöglich.

Schnittstellen, die über Rechnergrenzen zu Verfügung stehen, sollten also ganz besonders stabil sein.

Flexibilität von Transportschnittstellen
Transportschnittstellen enthalten Datenstrukturen, die per Verbindungssoftware übertragen werden. Sie werden von den technischen Adaptern erzeugt. Jedes Projekt legt fest, wie flexibel diese Schnittstellen sind. Je flexibler eine Transportschnittstelle ist, desto besser ist sie auf unvorhergesehene Situationen (z.B. für neue Clients) vorbereitet. Auch bleibt sie stabil, wenn sich Implementierungsdetails der gerufenen Komponente ändern. Ein Beispiel: Der folgende CORBA *typedef* macht Sinn:

```
typedef string FahrzeugNummer;
```

aber dieser hier weniger:

```
typedef string<15> FahrzeugNummer;
```

Die erste Variante verkraftet auch die Änderung der Fahrzeugnummer von 15 auf 20 Stellen. Man kann des Guten aber auch zuviel tun:

```
interface Generic {

    typedef sequence<string> StringList;
    StringList execute(in string functionName,
                       in StringList parameters);
};
```

Diese Schnittstelle definiert die Methode *execute*, welche als Parameter den Namen der auszuführenden Funktion erhält, und eine Liste von Zeichenketten als Parameter. Die Resultate werden ebenfalls als Liste von Zeichenketten zurück geliefert. Diese Schnittstelle ist universell: Jedes erdenkliche verteilte System lässt sich damit realisieren. Sie ist daher auch absolut stabil, denn noch allgemeiner geht es nicht. Anderseits fehlt jede Information darüber, welche Zeichenketten für *functionName* und *parameters* zulässig sind und wie das Ergebnis zu interpretieren ist. Die Schnittstellenspezifikation wurde also nur verschoben, und irgendein anderes Dokument muss klären, was diese Schnittstelle eigentlich bedeutet. Generische Schnittstellen dieser Art sind nur zulässig, wenn sie von einem geeigneten Adapter gekapselt sind.

Wo liegt der goldene Mittelweg zwischen einer spezifischen und einer flexiblen Schnittstelle? Exakte Regeln für den goldenen Mittelweg gibt es nicht, aber als grobe Leitlinie empfehlen wir:

- Schnittstellen werden aus Sicht der Komponente von innen nach außen gestaltet; d.h. die A-Schnittstelle einer Komponente weiß nichts über einen generischen oder spezifischen Aufruf. Dafür gibt es Adapter.
- Schnittstellen sind frei von Implementierungsaspekten. So ist das Persistenzmodell von Server-Objekten ein Implementierungsaspekt und ist an der Verteilungsschnittstelle unsichtbar. In die Verteilungsschnittstelle gehören nur solche Daten, die im Rahmen eines A-Falles tatsächlich nötig sind. Auch Dinge wie Feldlängen und Datentypen der verwendeten Datenbank sind an der Schnittstelle unsichtbar.
- Absehbare Erweiterungen und Veränderungen sollen ohne Schnittstellenänderung möglich sein. Unter diesem Gesichtspunkt wird man Datenstrukturen statisch oder dynamisch auslegen; auch Mischformen sind möglich.
- Die Verteilungsschnittstelle definiert das Anwendungsprotokoll: Welche Dienste (Schnittstellen) werden unterschieden? Welche Operationen enthalten die Schnittstellen? Welche Parameter und Resultate haben diese Operationen? Welche Fehler können auftreten? So vermeidet man die Probleme universeller Schnittstellen.

Als einfaches Beispiel betrachten wir Aufzählungstypen der CORBA-IDL. Dort kann man folgende Aufzählungstypen definieren:

```
enum Jahreszeit { fruehling, sommer,
                  herbst, winter };
enum Farbe { gelb, rot, gruen, blau };
```

Die *Jahreszeit* macht einen abgeschlossenen Eindruck; eine Erweiterung ist zwar denkbar *(karneval, altweibersommer)* aber unwahrscheinlich. Die *Farbe* ist weniger stabil. Daher verwendet man anstelle des Aufzählungstyp eine für Erweiterungen offene Liste von Konstanten:

```
typedef unsigned short Farbe;
const Farbe gelb = 0;
const Farbe rot = gelb+1;
const Farbe gruen = rot+1;
const Farbe blau = gruen+1;
```

Diese Definition erlaubt es später, den Aufzählungstyp *Farbe* zu erweitern, und zwar in einer *anderen* IDL-Spezifikation, ohne Änderung der ursprünglichen Definitionen:

```
#include "Farbe.idl"
const Farbe sonnengelb = gelb+100;
const Farbe ziegelrot = sonnengelb+1;
const Farbe schlammgruen = ziegelrot+1;
```

Diese Lösung hat natürlich die Nachteile flexibler Schnittstellen: Typprüfungen zur Übersetzungszeit sind nicht möglich, und es gibt Probleme bei der Weiterentwicklung und Wartung des Systems (z.B. die Vermeidung von Überschneidungen verschiedener Farb-Spezifikationen oder die Kontrolle, ob alle Farben auch überall implementiert sind).

Wir empfehlen konkrete Transportschnittstellen, soweit sich dies nicht durch spezielle Anforderungen an die Flexibilität verbietet. Da die Transformationslogik im Adapter steckt (und nicht in der Komponente), kann man bei Bedarf mit geringem Aufwand durch neue Adapter auch weitere Datensichten für Transportschnittstellen erzeugen.

8.2 Standardarchitektur VV (Verteilte Verarbeitung)

Die hier vorgestellte Standardarchitektur *VV* beschreibt eine flexible, plattformunabhängige und generische Lösung im Sinne der losen Kopplung von Abschnitt 1.4.2. Wir beschreiben zuerst die statische Sicht, d.h. die Bausteine der Architektur und ihre Schnittstellen. Die dynamische Sicht skizziert verschiedene Systemabläufe, z.B. die Initialisierung oder Fernaufruf eines Dienstes. Danach befassen wir uns mit der Fehlerbehandlung.

Abgrenzung zu CORBA und RMI

Verbindungssoftware wie CORBA und RMI leistet folgendes:

1. Übertragung von Daten. Dafür besitzt die jeweilige Verbindungssoftware eigene Datentypen.
2. Verwaltung von Referenzen auf entfernte fachliche Objekte.
3. Dienste, die man immer braucht, z.B. : Namensdienst zum Auffinden entfernter Referenzen, Security-Dienst zur verschlüsselten Datenübertragung.

Unser Grundsatz lautet: Adapter trennen Verbindungssoftware vollständig von der Anwendung. Wir verwenden die Standards als Transportschicht zur Übertragung der Daten (Punkt 1). Zusatzdienste (Punkt 3) werden ausschließlich zur Kommunikation zwischen den Adaptern genutzt. Die *naive* Verwendung von Verbindungssoftware hätte zur Folge, dass jede Komponentenschnittstelle vom Typ T wäre. Das widerspricht der Trennung von Zuständigkeiten, und deshalb wollen wir das nicht.

Objektorientierte Verbindungssoftware (CORBA,RMI, .NET Remoting) legt es nahe, mit Referenzen auf entfernte Objekte zu arbeiten (remote references). Die Standardarchitektur *VV* verwendet nur entfernte Referenzen auf Serveradapter und ggf. auf Dienste der Verbindungssoftware (z.B. Namensdienst). Es ist Aufgabe des Client-Adapters, diese entfernten Referenzen zu verwalten.

Ein System, in dem der Aufrufer Referenzen auf fachliche Objekte einer entfernten Komponente besitzt, ist enorm komplex, und die Zuständigkeiten sind unklar: Darf der Aufrufer das Objekt direkt manipulieren oder ist ein Fernaufruf an die Komponente nötig? Wir sind der Meinung, dass dieser Ansatz wenig Sinn macht und empfehlen einen Zugriff auf entfernte Objekte nur über die A-Fälle der zugehörigen Komponente. Referenzen auf A-Fälle sollten hinter einer A-Fall-Fassade verborgen werden. Dann wird lediglich diese Fassade als entfernte Referenz in der Registratur (z.B. CORBA Naming Service) eingetragen.

8.2.1 Anforderungen an die Standardarchitektur VV

Der Standardarchitektur *VV* liegen die folgenden Ideen und Anforderungen zugrunde:

- Allgemeingültigkeit: Betrachtet wird die verteilte Verarbeitung in zwei oder mehreren gleichberechtigten Instanzen, bei der entfernte Schnittstellen (A-Fälle) genutzt werden. Es wird keine Kommunikationshauptrichtung unterstellt; GUI-Client und Anwendungsserver sind nur ein Sonderfall.
- Ortstransparenz: Ziel ist es, dass es für den Nutzer im Normalfall weder ersichtlich noch relevant ist, ob ein Dienst tatsächlich entfernt oder lokal zur Verfügung steht. Dies zahlt sich z.B. im Test aus, da hier oft mit lokalen Diensten gearbeitet wird, wenn die Kommunikation nicht Testgegenstand ist.
- Plattformunabhängigkeit: Es wird keine spezielle Programmiersprache, Verbindungssoftware oder ähnliches unterstellt, auch wenn die Beispiele in Java ausgeführt sind. Prinzipiell lässt sich die vorgestellte Architektur auch auf einem Host mit Cobol implementieren und ermöglicht die Integration der verschiedenen Welten.

8.2.2 Statische Sicht

Das grundlegende Merkmal der Standardarchitektur *VV* ist die Verwendung von Adaptern, die den einheitlichen Übergang in verschiedene Welten ermöglichen. Die Adapter kapseln die notwendige Transformations- und Zugriffslogik, so dass Anwendung und Verteilung getrennt sind.

Bei Bedarf wird die eigentliche Transformation der Datenmodelle über einen ausgelagerten Transformator durchgeführt. Dies ist besonders sinnvoll, wenn die Beschreibung der Transformation zur Laufzeit als Metamodell vorliegt.

Die Adapter sind hierarchisch aufgebaut. Ein Sender-Adapter besteht aus einem A-Fall-Proxy (fachliche Schicht) und einem Kommunikationssender (technische Schicht). Die technische Schicht kapselt im Wesentlichen die Verbindungssoftware. Die fachliche Schicht leistet die Anbindung an die Komponente. Jede Schicht kann Aufrufe bündeln oder aufteilen, und sie kann Daten für unterschiedliche Schnittstellenkategorien transformieren. Beispiele hierfür sind:

1.
Ein A-Fall-Proxy bildet eine A-Schnittstelle auf eine 0-Schnittstelle ab.

2.
Ein Aufruf, der eine große Datenmenge erzeugt wird von einem Kommunikationssender in mehrere Aufrufe mit kleineren Datenpaketen zerlegt.

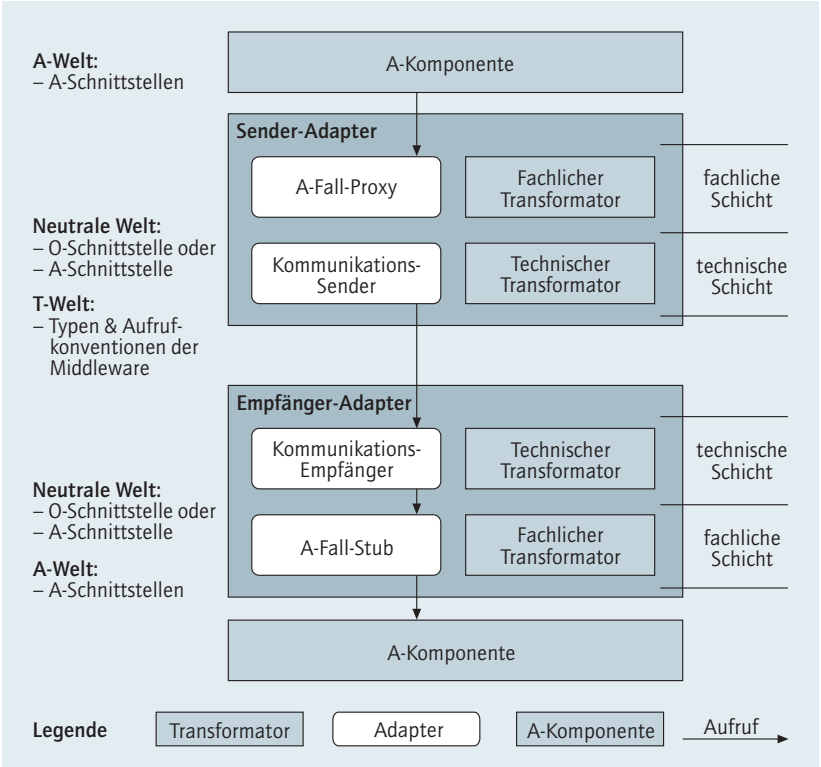
3.
Ein fachlich synchroner Aufruf wird durch den Kommunikationssender mit einer asynchron arbeitenden Verbindungssoftware abgewickelt.

Abbildung 26 zeigt insgesamt vier Adapter, die bei einem entfernten Aufruf durchlaufen werden: A-Fall-Proxy, Kommunikationssender, Kommunikationsempfänger und A-Fall-Stub. Dabei drängen sich zwei Fragen auf: Ist das performant? Ist das verwaltbar? Dazu lässt sich folgendes sagen:

Performance:
Lokal ausgeführte Transformationen spielen gegenüber dem eigentlichen Fernaufruf eine geringe Rolle. Bei sachgemäßer Programmierung hat man kaum messbare Nachteile zu erwarten. Auf der anderen Seite sind die Adapter genau die Stelle, an der man Optimierung einbauen kann (z.B. Datenkompression, Unterdrückung des Versands ungeänderter Daten), so dass die vorgeschlagene Architektur in vielen Fällen sogar einen besseren Durchsatz liefert.

Handhabung:
Die vorgeschlagenen Adapter sind nicht notwendigerweise physische Programme, möglicherweise jeweils in einer eigenen Datei, sondern logische Begriffe, die je nach Projekt und Entwicklungsumgebung sinnvoll zu organisieren sind.

Abbildung 26:
Standardarchitektur VV:
Adapter entkoppeln Welten



8.2.3 Statische Sicht – Konkretisierung

Mit den folgenden Zusatzannahmen wird die Standardarchitektur noch konkreter (vgl. Abbildung 27):

- A-Fälle werden auf dem Client durch A-Fall-Proxies repräsentiert (etwa eine Java-Adapterklasse für jeden entfernten A-Fall), die beim Deployment installiert werden. Vorteil: Für den Nutzer sehen diese Proxies genauso aus wie ein lokaler A-Fall, und er bleibt vollständig in der Programmiersprache und Anwendung des Clients.
- Die Adapter zur Kapselung der Verbindungssoftware (genannt Kommunikationssender bzw. -empfänger) stellen eine Verbindung zu einem entfernten A-Fall-Stub her.
- Zentrale Manager erzeugen A-Fall-Proxies und Kommunikationssender/empfänger. Ein separater Transformator erledigt Datenmodelltransformationen.
- Die Manager verwenden einen zentralen Verzeichnisdienst, um entfernte Dienste zu lokalisieren.

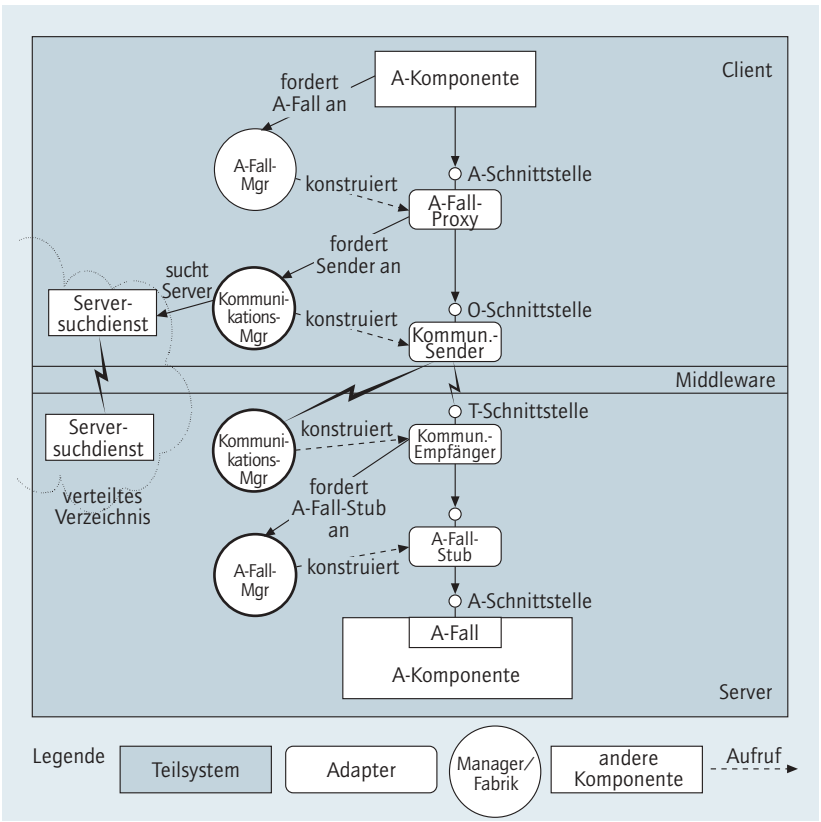
Die Standardarchitektur VV besteht aus folgenden Komponenten:

Komponente	Aufgabe	Bemerkungen
A-Fall-Manager	Erzeugt A-Fälle, A-Fall-Stubs und A-Fall-Proxies	Für den Aufrufer ist es transparent, ob ein A-Fall tatsächlich lokal ausgeführt wird oder entfernt (in letzterem Fall wird ein A-Fall-Proxy konstruiert)
A-Fall-Proxy	Stellvertreter des entfernten A-Falls auf dem Client. Wickelt den entfernten Aufruf transparent für den Nutzer ab.	
A-Fall-Stub	Stellt die Verbindung zum A-Fall auf der Empfänger-Seite her.	
Kommunikations-Manager	Verwalter für die Kommunikation über verschiedene Protokolle. Bei ihm kann ein Kommunikationskanal aus Sender und entferntem Empfänger für einen bestimmten Adapter angefordert werden.	Die Suche nach einem Server, der den gewünschten entfernten A-Fall-Stub anbietet, geschieht mit Hilfe eines verteilten Verzeichnisses, in dem sich alle Kommunikations-Manager angemeldet haben.
Kommunikations-Sender	Spezifischer Adapter zur Kapselung eines bestimmten Protokolls auf Sender-Seite. Ist dieser auf einem System installiert, so können von diesem Aufrufe über die Verbindungssoftware initiiert werden.	
Kommunikations-Empfänger	Spezifischer Adapter zur Kapselung eines bestimmten Protokolls auf Empfänger-Seite. Ist dieser auf einem System installiert, so können von diesem Aufrufe über die Verbindungssoftware angenommen werden.	
Serversuchdienst	Erlaubt das Registrieren und Finden von Servern mit bestimmten A-Fällen.	Verwendet ein verteiltes Verzeichnis (z. B. LDAP) und stellt einen einheitlichen Zugriff darauf zur Verfügung.

Anmerkungen:

- Die A-Fall-Stubs melden sich beim Systemstart bei ihrem lokalen Kommunikationsmanager unter einem symbolischen Namen an. Über diesen Namen findet sie der Client.
- Der Kommunikationsmanager trägt alle registrierten A-Fälle und alle unterstützten Protokolle in das verteilte Verzeichnis ein. Somit kann der Verzeichnisdienst die Server mit den zum Client passenden Protokollen aussuchen. Das verteilte Verzeichnis lässt sich weiter zu einem zentralen Lastverteiler ausbauen, der anhand der Last auf den potenziellen Servern, der Latenz einer Verbindung (ggf. unter Berücksichtigung von Präferenzen des Clients) einen Server aussucht.

Abbildung 27:
Standardarchitektur VV:
Aufruf eines entfernten A-Falls



8.2.4 Dynamische Sicht

Dieser Abschnitt beschreibt den Ablauf eines Fernaufrufs. Die meiste Arbeit passiert in den Adaptern. Aus Sicht des Anwendungsprogrammierers sind entfernte Aufrufe fast genauso einfach wie lokale. Er hat folgendes zu tun:

- Beschaffung eines A-Falls über einen Kommunikationsmanager,
- Füllung der Schnittstelle und Aufruf des A-Falls,
- Verarbeitung des Ergebnisses und Fehlerbehandlung.

Die gleichen Schritte müssen bei einem lokalen Aufruf durchgeführt werden. Das Wissen, dass es sich um einen Fernaufruf handelt, wird vollständig in den Adaptern verborgen.

8.2.4.1 Initialisierung und Abmelden der Kommunikationspartner

Die Initialisierung der beteiligten Server-Komponenten geschieht nach folgendem Schema:

1.
Start des Kommunikationsmanagers. Dabei werden alle installierten Sender/Empfänger registriert.

2.
Start des A-Fall-Managers. Dabei werden alle installierten A-Fälle, A-Fall-Stubs und A-Fall-Proxies registriert. A-Fall-Stubs werden beim lokalen Kommunikations-Manager unter einem symbolischen Namen registriert.

3.
Der Kommunikationsmanager meldet sich mit Hilfe der Empfänger bei der jeweiligen Verbindungssoftware an und ist somit erreichbar.

4.
Der Kommunikationsmanager meldet die von ihm unterstützten Protokolle und die über ihn erreichbaren A-Fälle im Serversuchdienst an.

Beim Abmelden einer Komponente meldet der Kommunikationsmanager alle registrierten A-Fall-Stubs ab. Es ist sinnvoll, vorher alle noch aktiven Aufrufer zu informieren, damit das spätere Abmelden nicht zu einer inkonsistenten Situation im Aufrufer führt.

8.2.4.2 Fernaufruf eines A-Falls

Die A-Komponente *a* möchte einen A-Fall einer anderen A-Komponente *b* aufrufen.

1.
a fordert beim lokalen A-Fall-Manager eine Instanz des A-Falls an.

2.
Der A-Fall-Manager findet einen A-Fall-Proxy unter dem gewünschten Namen und konstruiert diesen mit einer Referenz auf den lokalen Kommunikations-Manager.

3.
a ruft eine Methode des A-Fall-Proxies auf.

4.
Der A-Fall-Proxy fordert über den Kommunikations-Manager eine Verbindung zu seinem entfernten Partner (dem A-Fall) an.

5.
Der Kommunikations-Manager sucht und findet mit Hilfe des Server-Suchdienstes einen Server, auf dem der A-Fall installiert ist. Dazu übergibt er dem Suchdienst die Namen der von ihm unterstützten Protokolle. Er erhält vom Suchdienst den Namen des zu verwendenden Protokolls und die Protokoll-spezifische Adresse des Servers.

6.
Der Kommunikations-Manager konstruiert einen Kommunikations-Sender für den spezifischen A-Fall und den Server.

7.
Der Kommunikations-Sender stellt eine Verbindung zum entfernten Kommunikations-Manager her und fordert dort einen Kommunikations-Empfänger zu dem A-Fall an.

8.
Der entfernte Kommunikations-Manager konstruiert einen neuen Kommunikations-Empfänger und übergibt diesem die Verbindung zum Sender.

9.
Mit Hilfe des entfernten A-Fall-Managers fordert der Kommunikations-Empfänger einen A-Fall-Stub auf dem Server an.

10.
Der A-Fall-Stub erzeugt einen den gewünschten A-Fall auf dem Server.

11.
Es erfolgt die Rückmeldung bis zum Kommunikations-Manager auf dem Client. Dieser übergibt den Kommunikations-Sender mit der angehängten Verbindung bis zum entfernten A-Fall an den A-Fall-Proxy.

12.
Der A-Fall-Proxy transformiert die Aufrufparameter und ruft den Kommunikationssender auf.
13.
Der Kommunikationssender leitet der Aufruf über die Verbindungssoftware und den Empfänger-Adapter weiter.
14.
Die Methode des A-Falls der A-Komponente *b* wird ausgeführt.
15.
Das Ergebnis wird in umgekehrter Aufruf-Reihenfolge an den Aufrufer zurückgegeben.

8.2.5 Organisatorische Aspekte

Neben der technische Implementierung sind in einer verteilten Anwendung auch organisatorische Aspekte wichtig. Zur übergreifenden Kommunikation braucht man eine gemeinsame Sprache. Dazu gehören:

- ein Verzeichnis (nicht im Sinne der technischen Registratur, die zur Laufzeit genutzt wird), in der die vorhandenen Dienste mit ihren Aufrufbedingungen, Dokumentation, etc. verwaltet werden. Für jeden Dienst braucht man einen Produktverantwortlichen.
- ein gemeinsames Modell für die ausgetauschten A-Datentypen,
- ein Satz von vorgesehenen Fehlern und den Umgang damit und
- ein Namensformat für die Dienste, welches eine flexible Zuteilung von Namen erlaubt und Doppeldeutigkeiten ausschließt.

Wir empfehlen dafür den Aufbau einer eigenen organisatorischen Einheit. Dafür langt manchmal ein Teilteam, manchmal ist eine ganze Abteilung nötig.

8.3 Zustandsverwaltung

Die Zustände einer Serverkomponente kann man fachlich oder technisch betrachten:

- **A-Zustand:** Jeder Server, der aus *Anwendungssicht* Daten für Folgeaufrufe verwaltet, besitzt einen Zustand. Das können neben Daten selbst auch belegte Ressourcen (z.B. Verbindungen zu Nachbarsystemen) sein. Die meisten Server haben einen A-Zustand; Ausnahmen sind reine Berechnungsmodule (z.B. ein Zinsrechner).
- **T-Zustand:** Server haben auch *aus technischer Sicht* einen Zustand (stateful server), wenn sie auch für passive Clients Bereiche im Hauptspeicher vorhalten. Beispiel: stateful session beans (EJB).

Server ohne A-Zustand sind aus Gründen der Performance- und der Robustheit am günstigsten: Jeder Aufruf kann ohne Wiederherstellung des Kontextes ausgeführt werden, verschiedene Aufrufe desselben Clients können – bedingt durch Ausfälle oder Überlast – dynamisch auf verschiedene Server verteilt werden. Server ohne A-Zustand sind aber selten.

Viele Server besitzen aus Anwendungssicht einen Zustand, sind aber aus technischer Sicht zustandslos, weil sie ihren Zustand nicht im Hauptspeicher, sondern in einem Sekundärspeicher aufbewahren. Das sind die zweitbesten Server (also ohne T-Zustand). Damit auch ein Server mit A-Zustand bei hoher Last oder in Fehlersituationen funktioniert, kann man den A-Zustand in der Datenbank ablegen: Das betrifft neben den persistenten A-Entitäten auch Sitzungsinformationen, Zwischenergebnisse langlaufender Transaktionen und Wiederaufsetzinformationen. Alle (Applikations-) Server haben Zugriff auf diese Datenbank. Durch Einsatz von Caching lässt sich der Overhead für das Wiederherstellen minimieren (ein Serverwechsel sollte ja die Ausnahme sein).

Auch der Client kann den A-Zustands des Servers verwalten: Der Server reicht den A-Zustand nach jedem Aufruf an den Client weiter; dieser reicht sie bei der nächsten Anforderung an den Server zurück. Der Client kann diese Daten nicht interpretieren und schon gar nicht verändern.

Server mit T-Zustand haben zwei Nachteile:

a) Sie verbrauchen Ressourcen auch für Clients, die gerade nichts tun. Deshalb skalieren sie schlecht.
b) Im Falle eines Server-Ausfalls ist die Wiederherstellung des alten Zustands schwierig bis unmöglich.

Daher sollte man Server mit T-Zustand nach Möglichkeit vermeiden. Das heißt aber nur, den A-Zustand intelligent zu verwalten – aber nicht, ihn abzuschaffen.

8.3.1 Crash-Management im Server

Server mit T-Zustand brauchen Vorkehrungen für den Fall, dass sich der Client nicht mehr meldet: Ressourcen sind freizugeben, Inkonsistenzen zu bereinigen. Den Tod eines Client erkennt man über Timeout oder kontinuierliches Polling. Nach Ablauf der Frist wird die Sitzung beendet, alle belegten Ressourcen werden freigegeben und offene Transaktionen zurückgerollt. Damit dies auch für Ressourcen funktioniert, die im Anwendungscode belegt werden, können Callbacks registriert werden, die bei Sitzungsende aufgerufen werden.

8.4 Fehlerbehandlung

Bei verteilten Anwendungen gibt es zusätzliche Fehlerquellen: Netzwerke fallen aus, Server werden zur Wartung abgeschaltet, etc. Die Fehlerbehandlung der Anwendung muss sich darauf einstellen. Dabei stellt sich die Frage: Welche Fehler dringen bis in die Anwendung hoch (A-Code), welche können/sollen in technischen Schichten (T-Code) behandelt werden?

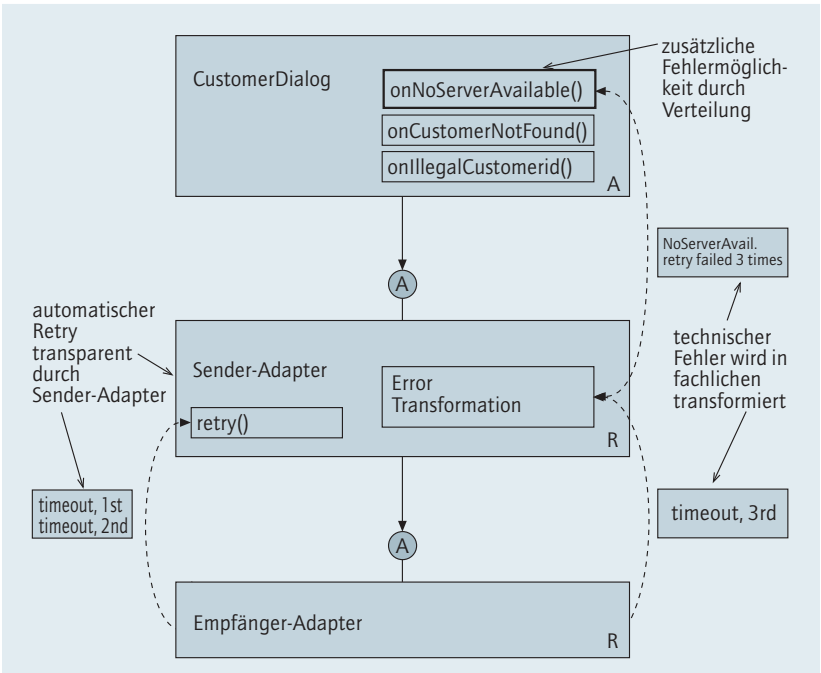


Abbildung 28: Fehlerbehandlung im Sender-Adapter

Dabei hilft die Unterscheidung von konsistenten Situationen und inkonsistenten Situationen (vgl. Abschnitt 1.5.1). Inkonsistente Situationen sind solche, auf die das Programm nicht vorbereitet ist, z.B. „Hauptspeicher voll". Fehlersituationen, auf die mit Notverfahren oder mit fachlich entscheidbaren Alternativen reagiert werden soll, sind konsistente Situationen im Sinne der Anwendung. Die Unterscheidung, ob eine bestimmte Situation als konsistent oder inkonsistent eingestuft werden soll, ist immer eine fachliche Frage und im besten Fall in der Spezifikation der Anwendung festgelegt. Es gelten folgende Regeln:

- Inkonsistente Situationen werden im technischen Code der Verbindungssoftware abgewickelt.
- Konsistente Situationen werden als fachliche Fehler an die Anwendung gemeldet und dort behandelt.

Das Entwurfsziel *Trennung der Zuständigkeiten* verbietet, dass technische Fehlercodes (z.B. *NotSerializableException*) einer konkreten Verbindungssoftware (z.B. RMI) sich durch den Anwendungskern ziehen. Technische Fehlercodes, die legale Situationen der Anwendung beschreiben, sind in fachliche Fehlercodes (bzw. Exception-Typen) zu transformieren. Die Anwendung kennt also nur von der konkreten Technik (CORBA, RMI, ...) abstrahierte Fehlercodes (vgl. Abbildung 28).

Wenn ein Fernaufruf fehl schlägt

Wir unterscheiden fünf Reaktionen auf das Scheitern eines Fernaufrufs:

- *ignore:* Den Fehler einfach übergehen. Das ist nur möglich, wenn der Aufruf optional war und für die Anwendung unwesentlich (Beispiel: Protokollierung).
- *retry:* Man wiederholt den Aufruf nach einer gewissen Wartezeit noch mal, in der Hoffnung, dass der Fehler dann nicht wieder auftritt.
- *backup:* Für das Scheitern des Fernaufrufs ist ein alternatives Verfahren vorgesehen. Das ist aufwendig; man macht das nur, wenn es der Kunde ausdrücklich fordert und bezahlt.
- *abort:* Die einfachste und daher häufige Lösung, wenn sie akzeptabel ist – man protokolliert den Fehler, führt das Notverfahren durch, beendet die Bearbeitung und setzt an der nächsten sinnvollen Stelle wieder auf.
- *delegation:* Das Problem wird dem Aufrufer mitgeteilt. Bei ihm liegt die Entscheidung über das weitere Vorgehen.

Mit Ausnahme von *backup* sind alle Reaktionen unabhängig vom fachlichen Kontext zu erledigen. Der Sender-Adapter kann die gewählte Strategie implementieren.

Literatur

[Alur 2001]
D. Alur, J. Crupi, D. Malks:
J2EE Patterns: Best Practices and Desing Strategies,
Prentice Hall, 2001.

[Ambler 1998]
S. Ambler:
Mapping Objects to Relational Databases,
AmbySoft, 1998.
www.AmbySoft.com/mappingObjects.pdf

[Buschmann 1996]
F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad,
M. Stal:
Pattern Oriented Software Architecture –
A System of Patterns. Wiley, 1996.

[Denert 1991]
E. Denert, J. Siedersleben:
Software-Engineering,
Springer Verlag, 1991.

[Keller 1998]
W. Keller, J. Coldewey:
Accessing Relational Databases: A Pattern Language,
in R. Martin, D. Riehle, F. Buschmann (Eds.):
Pattern Languages of Program Design 3,
Addison-Wesley, 1998.

[Linthicum 2001]
D. S. Linthicum:
B2B Application Integration: e-Business-Enable
Your Enterprise,
Addison-Wesley, 2001.

[Siedersleben 2002]
J. Siedersleben:
Software-Technik,
Hanser Verlag, 2002.

[X/Open 1996]
The Open Group:
Common Application Environment. Distributed
Transaction Processing: Reference Model,
Version 3, 1996.

sd&m Research

sd&m Research
Thomas-Dehler-Straße 27
81737 München
Telefon 089 638129-00
Telefax 089 638129-11

www.sdm-research.de

Quasar