

PerpleXX

# Ein Framework für Business-Objekte

Achim Demelt

Die reine Abbildung von Java-Objekten auf Tabellen in relationalen Datenbanken ist oft nicht befriedigend und ausreichend. Der vorliegende Artikel beschreibt die Konzepte für ein Framework, das mehr bietet als nur Objektpersistenz.

► Anwendungen zur Datenverarbeitung sind seit Jahren ein Schwerpunkt der Softwareentwicklung. Obwohl sich die Objektorientierung in der Programmierung weitestgehend durchgesetzt hat, erfolgt die Speicherung der Daten jedoch meist in relationalen Datenbanken.

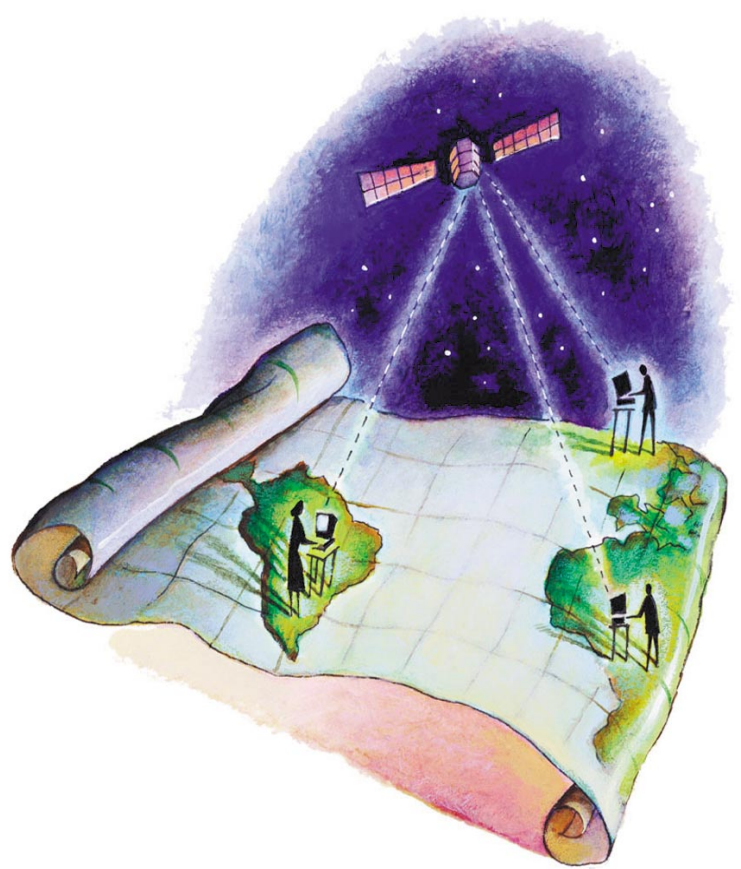
So genannte O/R Mapper versuchen den dadurch entstehenden Bruch an der Paradimgrenze zwischen objektorientierter Programmierung und relationaler Datenhaltung zu schließen. Solche Produkte sind nun seit einigen Jahren auf dem Markt und unterstützen den Entwickler bei der Speicherung von Objektnetzen in herkömmlichen Datenbanken. Im April 2002 wurde der JDO (*Java Data Objects*)-Standard – getrieben von Sun, einigen Herstellern von O/R Mappern und anderen Branchengrößen – verabschiedet, um ähnlich wie in anderen Bereichen eine Standardisierung der einzelnen Produkte zu erreichen [JDO].

Obwohl also mittlerweile ein Reifungsprozess im Markt zu erkennen ist, wird das „R“ bei O/R Mappern oftmals deutlich größer geschrieben als das „O“. Das Resultat ist eine einfache Spiegelung einzelner Klassen auf einzelne Tabellen, mit der kanonischen Abbildung von Objektbeziehungen auf Relationen.

Seitens der Produkthersteller gibt es Anstrengungen zur Adaption von Legacy-Datenbanken an Objektmodelle für neue Anwendungen. Dabei wird versucht, die direkte Kopplung der Objekte an die Tabellen zu lockern, um eine einfache Übernahme alter Datenbestände zu ermöglichen. Dies resultiert jedoch letztlich in fast allen Fällen in einer Anpassung des Objektnetzes an die zu Grunde liegende Datenbank. Auch bei Neuentwicklungen kommt man um diesen Kompromiss oft nicht herum.

## Grenzen von O/R Mappern

Als Beispiel soll das Klassendiagramm in Abb. 1 dienen. Das Business-Objekt **Fälligkeitsart** beschreibt die Termine innerhalb eines Kalenderjahres, an denen



Teile einer Zahlung fällig werden. Zum Beispiel kann eine Gebühr monatlich, halbjährlich oder jährlich bezahlt werden. Zudem hat jede Fälligkeitsart einen Namen und eine frei definierbare Liste von Synonymen (als einfache Strings).

Der übliche Ansatz zur persistenten Speicherung dieser Daten mit einem herkömmlichen O/R Mapper ist der Folgende: Die Klassen **Fälligkeitsart** und **Zeitpunkt** werden zu Business-Objekten erhoben und mit jeweils einem eigenen identifizierenden Feld (der Objekt-ID) in zwei Tabellen geschrieben. Jedes ihrer Attribute erhält dort eine passende Spalte. Eine Ausnahme bildet die Liste

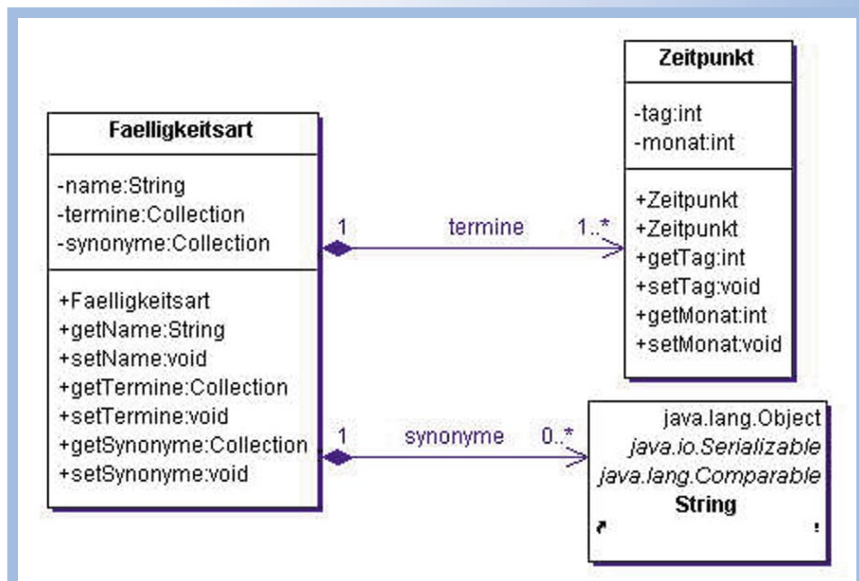


Abb. 1: Dieses Modell kann von herkömmlichen O/R Mappern nicht elegant abgebildet werden.

von Synonymen, welche in der Regel serialisiert wird und als BLOB in der Datenbank landet.

Betrachten wir uns die **Zeitpunkt**-Klasse etwas näher. Im Wesentlichen ist sie eine um das Jahr beraubte Datumsklasse. Wäre sie ein „richtiges“ Datum, hätte man sie auch mit dem üblichen `java.util.Date`-Datentyp modellieren können. Und obwohl sie kein `Date` ist, kann man mit Fug und Recht behaupten, **Zeitpunkt** sei eine Art Datentyp. Nichts rechtfertigt in diesem Kontext eine eigene, global eindeutige Objekt-ID für einen anonymen Zeitpunkt wie z. B. den 1. Juni. Der Lebenszyklus der Termine hängt voll von dem der Fälligkeitsart ab. Niemand wird alle Zeitpunkt-Objekte suchen wollen, deren Tag gleich 13 ist. Dennoch ist kein O/R Mapper in der Lage, diesen Datentyp als solchen zu behandeln. Richtig kompliziert wird die Situation, wenn ein solcher Datentyp nicht in der eigenen Anwendung modelliert wird, sondern aus einer Bibliothek eines Drittanbieters stammt. Einzige Lösung ist dann i.d.R. eine Wrapper-Klasse, die dessen Eigenschaften als Business-Objekt widerspiegelt.

Auch die Serialisierung der Synonym-Strings ist nicht befriedigend. Die Suche nach einer Fälligkeitsart, deren Synonymliste den Teilstring „\*jährl\*“ enthält, ist bei den binär abgelegten Daten nicht mehr möglich. Auch hier bestünde die Lösung in einer eigenen Synonym-Klasse, die mit einem einzigen String-Member und der obligatorischen Objekt-ID ein Business-Objekt darstellen muss.

### Constraints

Ein weiteres Manko sind die fehlenden Möglichkeiten zur Überprüfung der Konsistenz des Objektmodells. In unserem Beispiel sollte natürlich der Name der Fälligkeitsart gesetzt sein. Auf Datenbankebene kann dies mit einem **NOT NULL**-Constraint definiert werden. Wird jedoch ein Objekt ohne Namen gespeichert, so wird dies mit einer JDBC Exception quittiert, die eventuell noch tief in einer O/R Mapper spezifischen Exception verborgen ist.

Einige Produkte bieten mittlerweile eine **NOT NULL**-Prüfung auf Attributenebene. Jedoch tritt die meist nur dann in Kraft, wenn tatsächlich **set-**

**Name(null)** aufgerufen wird. Ein neu angelegtes Objekt, dessen Felder mit null vorbelegt werden, entzieht sich dieser Prüfung und beim Speichern in der Datenbank erhält man wieder die unverständliche JDBC Exception.

Möglichkeiten für „kompliziertere“ Überprüfungen werden i.d.R. kaum geboten. Die Vorgabe, dass Bezeichnungen für Fälligkeitsarten immer mit einem Großbuchstaben beginnen müssen, lässt sich mit einem einfachen regulären Ausdruck verifizieren. Der Programmierer prüft also in der **setName()**-Methode den übergebenen Parameter auf Korrektheit. Dasselbe muss er an zahlreichen anderen Stellen tun, z. B. beim Format von E-Mail-Adressen.

Andere Beispiele sind Mindest- und Maximallängen für Zeichenketten. Obwohl in Java theoretisch unbegrenzt, werden Strings in der Datenbank als in der Länge begrenzte **VARCHARs** abgelegt. Ein überlanger Wert wird ebenfalls mit einer JDBC Exception bestraft. Auch hier muss manuell an allen Stellen auf korrekte Längen geprüft werden.

Die Notwendigkeit der selbst programmierten Gültigkeitsprüfungen stellt den Entwickler vor das nächste Problem: Wo sollen die Prüfungen durchgeführt werden? Trotz bester Vorsätze findet hier oftmals eine Vermischung von Client- und Business-Logik statt. Einfache Prüfungen wie die oben vorgestellte Maximallänge werden aus Bequemlichkeit meist direkt im Texteingabefeld verankert. Andere, komplexere Prüfungen werden eher in die Geschäftsobjekte verlagert. Durch den direkten Zugriff des Clients auf die die Business-Logik implementierenden Klassen wird diese Vermischung noch verstärkt.

Eine ganz anders geartete Eigenschaft des Objektmodells wird von O/R Mappern vollkommen ignoriert: Kardinalitäten von Beziehungen. In unserem Beispiel muss eine Fälligkeitsart immer mindestens einen Termin besitzen. Obwohl diese Information im Modell in Form der „1..\*“-Kardinalität hinterlegt ist, wird kein O/R Mapper das Fehlen einer solchen Beziehung beanstanden. Wenn eine saubere Modellierung des Geschäftsmodells also nicht durch Abbildungsprobleme verhindert wird, so wird sie an anderen Stellen durch Nichtbeachtung auch nicht ermutigt. Als Konsequenz klaffen Modell und Realität oft weit auseinander.

### Ein Framework

O/R Mapper leisten also gute Dienste in Bezug auf einfache Abbildung von Objekten auf Tabellen. Insbesondere nehmen sie dem Programmierer das zeitraubende und fehleranfällige Schreiben von SQL-Anweisungen und die korrekte Bedienung der JDBC-Schnittstelle inklusive Transaktionssteuerung ab. Die Grundfähigkeiten nahezu aller Produkte sind sehr ähnlich und unterscheiden sich nur in den Schnittstellen. Dennoch sind ihre Grenzen klar zu erkennen.

Somit ist es lohnenswert, aufbauend auf der breiten Basis der Gemeinsamkeiten und den wertvollen Diensten, die O/R Mapper leisten, ein Framework zu entwickeln, das sich beim objektrelationalen Mapping mehr auf die Objekte besinnt und darüber hinaus zusätzliche Funktionalitäten bietet.

Ein solches Framework ist pleXX. Es besteht im Wesentlichen aus zwei Teilen:

- ▼ Einem Codegenerator, der modellierte Klassendiagramme analysiert und konkrete, für O/R Mapper verständliche Implementierungen erzeugt.

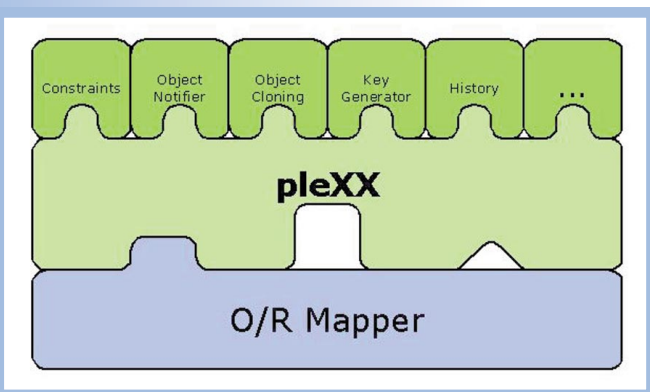


Abb. 2: Baustein-Konzept von pleXX

- ▼ Einer Laufzeitumgebung und API, die dem Programmierer eine einheitliche Schnittstelle für Objektpersistenz und Mehrwertdienste bieten.

Mit dem Framework soll die Unabhängigkeit von einem spezifischen O/R Mapper erreicht werden. Ziel ist es, den Austausch des verwendeten O/R-Produktes innerhalb eines Projekts zu ermöglichen, ohne den Anwendungscode ändern zu müssen. Ermöglicht wird dies durch den Codegenerator, der abhängig vom konkreten O/R Mapper speziell dafür angepassten Code erzeugen kann. Voraussetzung dafür ist die Einhaltung bestimmter Modellierungsvorgaben, die vom Generator analysiert werden können. Diese Vorgaben sind auch darauf ausgerichtet, eine strikte Trennung von Client- und Business-Logik zu erzwingen.

Für einen Softwaredienstleister, der zahlreiche Projekte für verschiedenste Kunden realisiert, ist das natürlich auch ein strategischer Vorteil. Abhängig von den konkreten Anforderungen und Wünschen des Auftraggebers kann ein passender O/R Mapper ausgewählt werden. Die Mitarbeiter hingegen finden über alle Projekte hinweg die gleichen Schnittstellen und Projektstrukturen vor. Der Wechsel eines Mitarbeiters von einem Projekt in ein anderes wird erleichtert.

Im Zuge der Unabhängigkeit vom O/R Mapper kann zudem ein Ausgleich der oben beschriebenen Unzulänglichkeiten im Bereich des objektrelationalen Mappings vorgenommen werden. Dies lässt dem Entwickler mehr Freiheiten beim Entwurf der Objektmodelle, ohne ständig an die konkrete Abbildung in Tabellen denken zu müssen.

Schließlich bietet das Framework weitere Möglichkeiten, die über die reine Persistenz hinaus gehen:

- ▼ Constraints: Überprüfung der Konsistenz des Objektmodells.
- ▼ Backlinks: Automatische Pflege bidirektionaler Beziehungen.
- ▼ ChangeNotifier: Benachrichtigung bei Änderung eines Objekts, z. B. zur Aktualisierung im GUI.
- ▼ Historisierung: Transparente Verarbeitung temporaler (zeitbezogener) Daten.
- ▼ Description: Benutzerverständliche Beschreibung von Objekten, was bei der Darstellung von Fehlermeldungen hilfreich ist.
- ▼ Cloning: Kopieren von Objekten und Objektnetzen.
- ▼ KeyGenerator: Flexible Vergabe von Objekt-Identifiern.

Jedes dieser Features ist natürlich optional. Alles in allem ergibt sich das Bausteinmodell aus Abb. 2. Im Folgenden wird die generelle Arbeitsweise des Frameworks vorgestellt. Eine ausführliche Beschreibung der Mehrwertdienste erfolgt in weiteren geplanten Artikeln.

## Modellierung

Die Definition des Objektmodells erfolgt im UML-Werkzeug *Together*, dessen Round-Trip-Engineering und einfache Erweiterbarkeit dank umfangreicher API sich bestens für den Einsatz eines Frameworks eignet. Anders als bei herkömmlichen Tools werden für Klassendiagramme hier nicht nur Bilder gezeichnet, aus denen Quelltext generiert wird. Vielmehr werden Diagramm und Quellcode stets synchron gehalten. Das Einfügen eines Attributs im Diagramm resultiert in der sofortigen Ergänzung der betreffenden Klasse.

Bei der Modellierung für pleXX wird unterschieden zwischen *Business Objects* (BOs) und *Data Objects* (DOs). Als Business-Objekte werden Entitäten des Geschäftsmodells bezeichnet, die die spezifischen Daten und Methoden der Fachklassen beinhalten.

Data Objects sind eher als Datentypen zu betrachten. Sie besitzen keine (persistente) Identität und ihr Lebenszyklus ist immer an den des sie beinhaltenden Business-Objekts gekoppelt. Die oben vorgestellte **Zeitpunkt**-Klasse ist ein Beispiel für ein DO. Andere Anwendungsfälle wären z. B. ein Geldbetrag, bestehend aus einem Wert und einer Währungseinheit.

BOs werden als (Java-)Interface definiert und mit dem Stereotyp `<<business object>>` gekennzeichnet. Neben den Business-Methoden werden außerdem die `get`- und `set`-Methoden für die persistenten Attribute deklariert. Die Verwendung von Interfaces gegenüber Klassen hat folgende Vorteile:

- ▼ Viele O/R Mapper erfordern die Implementierung bestimmter Interfaces um Java-Objekte überhaupt persistierbar zu machen oder um bestimmte Features zu nützen. Eine Implementierung dieser Interfaces direkt in einer Business-Objekt-Klasse würde das Objektmodell zu sehr an ein spezifisches Produkt binden. Dies übernimmt später der Generator.
- ▼ Es entkoppelt den Anwendungscode von der Implementierung der Business-Logik. Die Verwendung von Interfaces ist ein bewährtes Mittel, um den Client unabhängig von der konkreten Realisierung der ihm angebotenen Dienste zu machen.
- ▼ Der Einsatz „richtiger“ Java-Interfaces anstelle eines abstrakten Objektmodells bietet einen direkten Bezug zur eingesetzten Programmiersprache. Somit wird nicht nur eine „anonyme“ Beziehung zwischen zwei Klassen im Diagramm gezeichnet, sondern sie wird an einer konkreten Java-Collection in genau einem Interface verankert.
- ▼ Schließlich spart die Verwendung von Java-Interfaces auch unnötige Mehrarbeit. Die Präsentationslogik der Anwendung wird später mit exakt diesen Interfaces kommunizieren und daraus die nötigen Information extrahieren. Die Transformation eines abstrakten Modells in richtigen Quelltext entfällt also.

Im Gegensatz zu Business-Objekten werden Data Objects als Klassen definiert. Zum einen wird dadurch ihre Behandlung als Datentyp unterstrichen. Zum anderen stammen solche Datenobjekte oft aus Standardbibliotheken, die nicht im Quelltext vorliegen und angepasst werden können.

## Ein Beispiel

Das obige Fälligkeitsart/Zeitpunkt-Modell ist in einer leicht vereinfachten Form in Abb. 3 dargestellt. Listing 1 enthält den Quelltext dazu. Der reine Java-Quellcode ist um einige Metainformationen in Form von JavaDoc-Kommentaren angereichert. Attributbezogene Daten werden per Definition an der `get`-Methode verankert, da nicht in allen Fällen eine `set`-Methode vorhanden ist.

Auffällig sind zunächst die Ergänzungen bei der Beziehung zu den **Zeitpunkt**-Objekten. Das `@associates`-Tag definiert für die ansonsten untypi-



sierte Java-Collection die enthaltenen Objekte. Ferner werden der Beziehungstyp (`@link aggregationByValue`, also Komposition) und die Kardinalitäten definiert. Dies ist auch die Standardnotation, mit der Together seine Modellinformationen im Quelltext hinterlegt.

Des Weiteren werden mit `@mandatory` und `@regexp [A-Z][a-z]*` zwei Constraints definiert, die vorschreiben, dass der Name der Fälligkeitsart ein Pflichtfeld ist und mit einem Großbuchstaben beginnen muss, und danach Kleinbuchstaben folgen müssen. Auf weiter gehende Features wird in diesem Einführungsbeispiel verzichtet.

## Generation Gap

Nun rückt der Generator ins Blickfeld. Es handelt sich hierbei um ein in Java geschriebenes Erweiterungsmodul für Together, das direkt von der integrierten Entwicklungsumgebung aus aufgerufen werden kann. Mit Hilfe der Together API hat man vollen Zugriff auf das gesamte Objektmodell – und zwar lesend und schreibend. Dabei können auch die JavaDoc-Metainformationen abgefragt werden.

Der pleXX-Generator analysiert die modellierten Business und Data Objects und erzeugt daraus abhängig vom gewählten O/R Mapper konkrete Implementierungen. Die generierten Klassen haben die folgenden Inhalte:

- ▼ Die persistenten Attribute mit `get-` und `set-` Methoden.
- ▼ Implementierungen einiger pleXX Interfaces, unter anderem zur Transaktionssteuerung.
- ▼ Implementierungen vom O/R Mapper benötigter Interfaces.
- ▼ Implementierung weiterer pleXX Interfaces für Mehrwertdienste (z. B. Änderungsbenachrichtigung).
- ▼ Code zur Pflege bidirektionaler Beziehungen.
- ▼ Prüfung der Constraints beim Setzen der Attribute und vieles mehr.

Natürlich kann der Generator keine Implementierung der Business-Logik erzeugen. Nur Methoden in den BO Interfaces, die einem ihm bekannten Namensschema entsprechen (wie z. B. `get` und `set` für Attribute), werden implementiert. Alle anderen Methoden bleiben unberührt. Die entstehende Klasse ist dann also abstrakt.

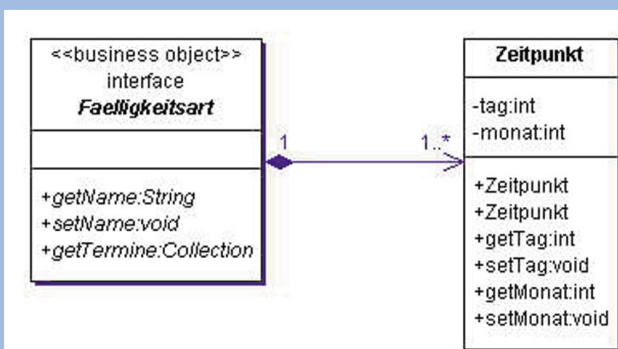


Abb. 3: BO-Modell für pleXX

```

package de.excellent.plexx.spektrum.bo;

import java.util.Collection;

/**
 * @stereotype business object
 */
public interface Faelligkeitsart {
    /**
     * @mandatory
     * @regExp [A-Z][a-z]*
     */
    String getName();

    void setName(String name);

    /**
     * @associates Zeitpunkt
     * @link aggregationByValue
     * @clientCardinality 1
     * @supplierCardinality 1..*
     */
    Collection getTermine();
}
    
```

Listing 1: Quelltext für das Business-Objekt Faelligkeitsart

Statt die fehlenden Methoden direkt im generierten Code zu ergänzen, sollte der Entwickler eine weitere Klasse erstellen, die von der generierten erbt. Dieser *Generation Gap*-Ansatz („Generationslücke“, siehe [VII98]) ist in Abb. 4 dargestellt. Generierter Code schließt hier also die Lücke zwischen zwei manuell programmierten Klassen bzw. Interfaces.

Die strikte Trennung von generiertem und manuell programmiertem Code erlaubt jederzeit eine Neugenerierung der Implementierungsklassen. Insbesondere können Änderungen im BO-Modell einfach durch einen erneuten Generatorlauf übernommen werden. Anders als bei anderen Ansätzen, wo generierter und eigener Code gemischt und durch spezielle Kommentarblöcke getrennt werden, besteht hier nicht die Gefahr, dass manuell programmierter Quelltext überschrieben wird. Ganz im Gegenteil, die generierten Klassen werden i.d.R. gar nicht in das Versionskontrollsystem eingchecked, sondern können stets neu erzeugt werden.

Neben der Implementierung des BO Interface erzeugt der Generator im vorliegenden Fall noch eine weitere Klasse. Es handelt sich dabei um eine Wrapper-Klasse für das Zeitpunkt-DO. Wie in der Einführung ja festgestellt wurde, kann ein O/R Mapper mit solch einem Datentyp nicht direkt umgehen. pleXX sorgt daher dafür, dass ihm ein entsprechendes persistentes Objekt präsentiert wird.

## Verwendung von pleXX

Betrachten wir nun, wie sich die Benutzung von pleXX zur Laufzeit gestaltet. Listing 2 zeigt eine einfache Codesequenz, die mit den Objekten aus dem Fälligkeitsartmodell arbeitet.

In den Zeilen 4 bis 9 werden Laufzeitparameter für pleXX und den O/R Mapper definiert. Statt diese wie hier im Beispiel fest zu definieren, wäre auch eine Übergabe in der Kommandozeile mit `-D` oder das Lesen aus einer Property-Datei denkbar. Hier wird die Laufzeit für die Verwendung von intelliBO von Signsoft [SS02] konfiguriert.

Nach dem Anfordern einer Transaktion (Zeile 12) wird eine Instanz einer Fälligkeitsart von der ObjectFactory erzeugt (Zeile 15). Nur diese ist dazu in der Lage, denn der Client hat nur Zugriff auf die BO Interfaces und sollte die Implementierungsklassen nie zu Gesicht bekommen. Insbesondere kann er also keine neue Instanz mittels `new` erzeugen.

Das Objekt wird in den Zeilen 18 bis 20 mit Daten befüllt und persistent gemacht (Zeile 21). Schließlich wird die Transaktion abgeschlossen.

In einer neuen Transaktion werden nun einige Fehlerfälle getestet. Zunächst wird ein ungültiger Name für eine neue Fälligkeitsart vergeben (Zeile 33). pleXX reagiert mit einer `ConstraintViolationException` um diesen Fehler anzuzeigen. Nach einer Korrektur wird versucht, die Transaktion abzuschließen. Da jedoch noch kein Zeitpunkt für die Fälligkeitsart definiert wurde, wird auch dies abgewiesen (Zeile 42). Im Objektmodell war ja definiert, dass eine Fälligkeitsart immer mindestens einen Zeitpunkt besitzen muss.

Die API gestaltet sich also sehr ähnlich wie die von reinen O/R Mappern. pleXX definiert lediglich eine Schicht eigener Interfaces, um von konkreten Produkten unabhängig zu sein.

### Fazit und Ausblick

Das pleXX-Framework bietet eine Abstraktionsschicht zwischen der Anwendung und einem konkreten O/R Mapper (welcher wiederum von einer konkreten Datenbank abstrahiert). Die Modellierungsvorgaben erzwingen eine Trennung von Client, Geschäftsmodell und Implementierung. Ein Großteil der letzteren wird vom Codegenerator erzeugt. Eine Laufzeitumgebung und API runden das Bild ab. Abbildung 5 zeigt dies schematisch.

Der Vorteil für den Entwickler liegt auf der Hand: Er kann sich voll

auf die Modellierung seines Geschäftsmodells konzentrieren und sich dabei ausschließlich in der OO-Welt bewegen. Die Einschränkungen bei der Abbildung auf relationale Datenbanken spielen hier keine Rolle. Der Einsatz des Generators spart viel Arbeit, reduziert die Fehleranfälligkeit und steigert damit die Produktivität. Zudem bindet er das Projekt nicht an ein spezifisches Produkt für die Persistenz der Objekte.

Leider kann keine vollkommene Unabhängigkeit von O/R Mappern erreicht werden. Der einzige Punkt, an dem eine Anwendung den direkten Weg zu proprietären Schnittstellen eines konkreten Produkts gehen muss, sind Queries. Jeder O/R Mapper hat seine eigene Abfragesprache, um persistente Objekte nach bestimmten Kriterien zu suchen. Das Thema hat zwar durch die in JDO definierte JDOQL (*JDO Query Language*) eine gewisse Vereinheitlichung erfahren. Es gibt aber dennoch Möglichkeiten der Erweiterung für Produkthersteller.

Auch in diesem Gebiet eine Abstrahierung in pleXX zu erzielen, hätte bedeutet, eine eigene Abfragesprache zu entwerfen bzw. eine vorhandene (z. B. JDOQL) zu parsen und in eine dem O/R Mapper verständliche Form zu überführen. Der immense Aufwand würde aber das Ergebnis nicht rechtfertigen. In vielen Objektmodellen nimmt – sobald einmal ein Einstiegspunkt gefunden ist – die *Objektnavigation* eine Vormachtstellung ein. Andere Objekte können meist durch Verfolgen von Beziehungen erreicht werden. „Richtige“ Abfragen mit Suchkriterien sind in der Minderheit.

Trotz dieser geringen Einschränkung wäre wohl die dennoch erreichte Unabhängigkeit von proprietären Produkten für viele Entscheidungsträger ein ausreichendes Argument für den Einsatz eines Frameworks wie pleXX. Wie be-

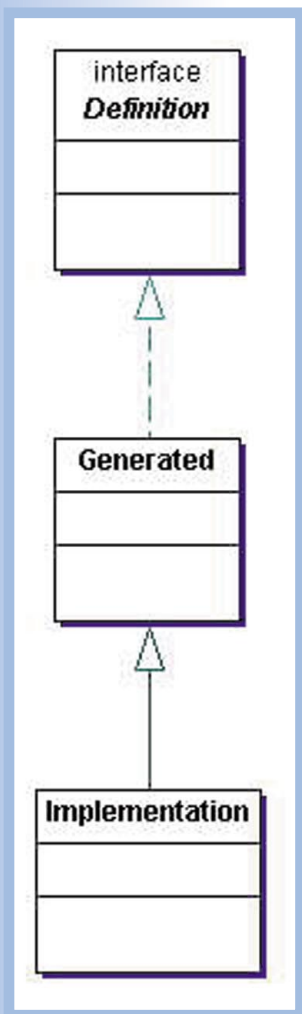


Abb. 4: Das Generation Gap Pattern

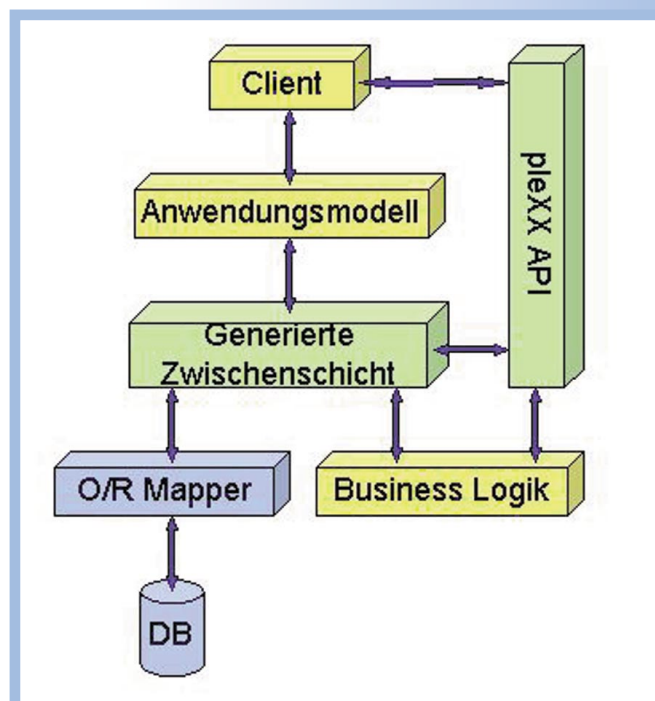


Abb. 5: Schichtenmodell bei der Verwendung des pleXX-Frameworks

reits erwähnt, bietet das Framework aber darüber hinaus zahlreiche weitere Dienste, die den Nutzen immens erhöhen.

Im nächsten Teil dieses Artikels werden die schon kurz vorgestellten Constraints näher betrachtet. Sie stellen ein mächtiges Werkzeug dar, um stets die Konsistenz des Objektmodells sicherzustellen. Ein weiterer geplanter Artikel widmet sich der Thematik temporaler Daten. Er zeigt die Modellierung und die Handhabung historisierter (zeitbezogener) Objekte und derer Relationen. Auch ohne die Verwendung des pleXX-Frameworks enthalten diese geplanten Artikel wertvolle Informationen und Konzepte, die die Qualität jedes datengetriebenen IT-Projekts spürbar verbessern können.

## Literatur und Links

- [JDO] Java Specification Request 12, <http://www.jcp.org/jsr/detail/12.jsp>
- [Signsoft] intelliBO-Produktseite, <http://www.intellibo.com/>
- [Vli98] J. Vlissides, Pattern Hatching: Design Patterns Applied, Addison Wesley, 1998



**Dipl.-Inf. (FH) Achim Demelt** ist Softwarearchitekt und Projektleiter bei der eXcellent solutions GmbH in Ulm. Einer seiner Schwerpunkte ist die Steigerung der Effizienz bei der Implementierung datengetriebener Anwendungen in Java.  
E-Mail: a.demelt@excellent.de.

```

01 public class Demo {
02     public static void main(String[] args) {
03         try {
04             // System Einstellungen setzen
05             System.setProperty("plexx.PersistenceManager",
06                 "de.excellent.plexx.impl.ibo.IntelliBOPersistenceManager");
07             System.setProperty("plexx.ObjectFactory",
08                 "de.excellent.plexx.impl.ibo.IntelliBOObjectFactory");
09             System.setProperty("plexx.ibo.driver",
10                 "org.enhydra.instantdb.jdbc.idbDriver");
11             System.setProperty("plexx.ibo.url",
12                 "jdbc:ldb:conf/ldb.properties");
13             System.setProperty("plexx.ibo.connectionInfo",
14                 "com.signsoft.ibo.dbsupport.instantdb.IDBDatabaseInfo");
15
16             // Transaktion abholen
17             Transaction tx = PersistenceManager.getInstance().
18                 createTransaction();
19
20             // Objekt erzeugen
21             Faelligkeitsart fArt =
22                 (Faelligkeitsart)ObjectFactory.getInstance().create(
23                     tx, Faelligkeitsart.class);
24
25             // Objekt befüllen und persistieren
26             fArt.setName("Halbjährlich");
27             fArt.getTermine().add(new Zeitpunkt(1, 1));
28             fArt.getTermine().add(new Zeitpunkt(1, 7));
29             tx.makePersistent(fArt);
30
31             // Transaktion abschließen
32             tx.commit();
33
34             // Neue Transaktion mit neuer Fälligkeitstyp
35             tx =
36                 PersistenceManager.getInstance().createTransaction();
37             fArt = (Faelligkeitsart)ObjectFactory.getInstance().
38                 create(tx, Faelligkeitsart.class);
39             tx.makePersistent(fArt);
40
41             // Ungültigen Namen setzen
42             try {
43                 fArt.setName("kleinbuchstabe");
44             }
45             catch (ConstraintViolationException e) {
46                 // korrigieren
47                 fArt.setName("Großbuchstabe");
48             }
49
50             // Ohne Termin Speichern ist nicht erlaubt
51             try {
52                 tx.commit();
53             }
54             catch (ConstraintViolationException e) {
55                 // korrigieren und nächster Versuch
56                 fArt.getTermine().add(new Zeitpunkt(1, 1));
57                 tx.commit();
58             }
59
60             catch (NoTxException e) {
61                 // ...
62             }
63
64             catch (TxAbortedException e) {
65                 // ...
66             }
67
68             catch (DuplicateObjectException e) {
69                 // ...
70             }
71         }
72     }
73 }

```

Listing 2: Quelltext eines pleXX-Anwendungsprogramms