

Was müssen Persistenz-Lösungen können?

Schön, dass es .NET gibt: Endlich können wir mit Objekten programmieren. Aber was tun wir, wenn wir die Objekte abspeichern wollen? Persistenz-Layer übernehmen die Aufgabe, Objektbäume in Datenbanken abzulegen und bei Bedarf wieder aufzubauen. Dabei kommen verschiedene Technologien zum Einsatz, die unterschiedliche Produkte hervorbringen. Welche sind geeignet für Ihren Zweck?

Als Entwickler von NDO, einer Persistenzlösung für .NET [1] sind wir auf die harte Tour mit all den Fragestellungen konfrontiert worden, die sich an eine Persistenzlösung stellen. Aber das ist vielleicht gut so. Denn hätten wir vorher gewusst, was uns erwartet, hätten wir möglicherweise nie mit der Entwicklung begonnen.

Im Folgenden versuchen wir, diese Fragestellungen so zu vermitteln, dass eine Checkliste daraus entsteht, die Sie zur Hand haben, wenn es darum geht, Persistenz-Lösungen anzuschaffen. Der Artikel abstrahiert von den konkreten Produkten auf dem Markt und enthält daher auch keine Wertungen. Eine Ausnahme von dieser Regel machen wir, wenn es darum geht, Erweiterungsschnittstellen zu beschreiben. Eine Liste von Herstellern finden Sie am Schluss im Anhang.

Das Thema Persistenz ist wichtig. Das zeigt sich unter anderem daran, dass Microsoft versucht hat, mit ObjectSpaces [2] selbst eine Lösung zu präsentieren, obwohl Microsoft ja seit Jahren die datenzentrierten Lösungen von ADO bzw. ADO.NET verfiel.

Eine Persistenzschicht richtig zu entwickeln ist allerdings nicht leicht, man gerät schnell in konzeptionelle Sackgassen. Das zeigt sich ebenfalls an den ObjectSpaces: sie werden in der kommenden Version von .NET (2.0) definitiv nicht enthalten sein. Der Ansatz birgt einige Schwächen, daher kann man erwarten, dass Microsoft eine völlig neue Lösung entwickeln oder zukaufen wird. Eine Lösung als Bestandteil von .NET wird daher – wenn überhaupt – nicht vor Ablauf von ca. 2-3 Jahren verfügbar sein. Es gibt jedoch bereits jetzt Persistenz-Lösungen von verschiedenen Herstellern, die sehr gut funktionieren. Sie müssen also nicht warten.

Für Ungeduldige: Bitte hier lesen!

Wenn Sie nicht viel Zeit für die Orientierung im Bereich der Persistenz-Lösungen haben, dann sollten Sie wenigstens die folgenden

fünf Punkte lesen, um eine Entscheidungsgrundlage zu erhalten:

- Datenbankstruktur
- Lazy Loading (Hollow Objects)
- Dirty-Status-Verwaltung
- Veränderung der Vererbungshierarchie
- Transaktionen

Datenbankstruktur:

Es gibt zwei Kategorien von Persistenz-Lösungen: Solche mit Objektspeichern (wie zum Beispiel Poet) und solche, deren Daten in normalen relationalen Datenbanken untergebracht werden können (Mapping Tools).

Welche für sie in Frage kommt, hängt von der IT-Strategie in Ihrem Haus bzw. beim Kunden ab. Viele große Firmen fordern explizit die Möglichkeit, Datenbestände unabhängig von der Logik der Anwendungen, aus denen sie stammen, analysieren zu können. OLAP ist hier ein wichtiges Stichwort.

Ist dies nicht gefordert, können Objektspeicher in gewissen Szenarien vorteilhaft sein. Objektrelationales Mapping beschränkt die Ausdrucksmittel, die Sie bei der objektorientierten Entwicklung haben. Sehr tiefe Klassenhierarchien mit viel Polymorphie sind schlecht in die relationale Welt abbildbar (Siehe den Abschnitt „Objektrelationales Mapping“).

In den meisten Geschäftsanwendungen aber kann man mit den Einschränkungen des Mappings leben und dadurch die Vorteile von relationalen Datenbanken nutzen.

Hier kommt dann die Frage auf, wie die Persistenzlösung auf die Datenbankstruktur einwirkt. Gibt es irgendwelche Einschränkungen? Ist ein Mapping der Objekte auf die üblichen relationalen Beziehungsmustern möglich?

Der Hätetest ist meist, ob eine vorhandene Datenbank von der Lösung verwendet werden kann. Von Mapping-Tools, welche die pubs-Datenbank des SQL-Servers nicht in die Objektwelt abbilden können, sollten Sie Abstand nehmen... Hat hier jemand

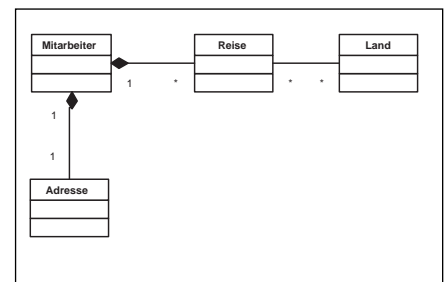


Abbildung 1 Ein einfaches Modell

gelacht?

Beziehungen und Lazy Loading

Eine Persistenzlösung soll nicht nur einzelne Objekte in Tabellen ablegen können, sondern auch die Beziehungen von Objekten zueinander.

Angenommen, Sie haben die folgende Objekthierarchie: Firma -> Mitarbeiter -> Reise -> Beleg. In Ihrem Datenbestand

Auf einen Blick

Persistenz-Lösungen können Objekte, die im Speicher aufgebaut wurden, in Datenbanken abspeichern. Mirko Matytschak zeigt, worauf Sie achten müssen, wenn Sie über die Anschaffung einer Persistenzlösung nachdenken. Er gibt auch einen Einblick in die Funktionsweise solcher Lösungen.

Der Autor

Mirko Matytschak programmiert schon seit 1977. Er leitete zehn Jahre lang die Fachredaktion der Zeitschrift „System Journal“ und betreute andere Publikationen im Entwickler-Bereich. Er veranstaltet die Advanced Developers Conference und ist als Software-Entwickler und -Berater für verschiedene Firmen tätig. Er ist maßgeblich an der Entwicklung der Persistenz-Lösung NDO beteiligt. Erreichbar ist er unter www.advanced-developers.de.

befindet sich eine Firma mit 50 Mitarbeitern, jeder der Mitarbeiter hat im Schnitt bereits 50 Reisen unternommen, auf denen im Schnitt 10 Belege anfallen. Sie wollen nun das Firmen-Objekt laden.

Natürlich sollen beim Laden eines Objekts auch die Beziehungen zu anderen Objekten wiederhergestellt werden. Das hieße aber in unserem Beispiel, dass das Laden des Firmenobjekts das Laden von 25.000 Objekten nach sich zieht. Wenn Sie nämlich die Beziehung zu den Mitarbeiter-Objekten wieder herstellen wollen, müssen Sie die Reiseobjekte laden, das aber bedeutet, dass Sie die Beleg-Objekte brauchen, etc.

Es braucht also einen Mechanismus, der diese Kette an einem wohl definierten Punkt unterbricht. Dazu bedient man sich der so genannten Hollow Objects.

Wenn Sie das Firmen-Objekt laden, dann werden statt der Mitarbeiter-Objekte Platzhalter in die Liste der Mitarbeiter-Objekte eingefügt. Erst wenn ein Zugriff auf einen der Mitarbeiter erfolgt, wird dieses Objekt nachgeladen.

Eine Persistenzlösung muss einen solchen Mechanismus aufweisen – sonst ist sie nicht zu gebrauchen. Wir werden später noch einmal zu diesem Thema zurückkommen, wenn wir beschreiben, wie das Lazy Loading verwirklicht wird.

Es sei darauf hingewiesen, dass an dieser Hürde bereits ein Großteil der Persistenz-Lösungen scheitert.

Dirty-Status-Verwaltung

Angenommen, Sie haben 25.000 Objekte geladen. Im Lauf der Zeit wurden zwei davon geändert. Nun drückt der Benutzer auf die Speichern-Taste. Werden nun zwei oder 25.000 Objekte gespeichert? Das hängt davon ab, ob der Persistenz-Layer eine Status-Verwaltung hat, die feststellen kann, ob sich ein Objekt geändert hat.

Komfortable Systeme können das automatisch – das ist allerdings alles andere als trivial zu implementieren. Einfachere Systeme erfordern es, dass der Benutzer die Objekte beim Persistenz-System als „Dirty“ anzeigt – das ist unelegant. Alle 25.000 Objekte zu speichern wäre intolerabel.

Wie sich später noch zeigen wird, gibt es einen fatalen Zusammenhang zwischen der Statusverwaltung und der Fähigkeit, jedes beliebige Objekt speichern zu können. Eine saubere, vollautomatische Statusverwaltung bekommen Sie nur für Klassen, die Sie selbst geschrieben haben.

Veränderung der Klassenhierarchie

Wer Persistenz-Lösungen einsetzt, will unbeschwert vom Aspekt der Persistenz seine fachlichen Anforderungen in Klassen gießen. Daher ist es keine gute Praxis, wenn

eine Persistenzlösung es erfordert, dass persistente Klassen von einer Basisklasse abgeleitet werden. Wir werden später noch erklären, warum dennoch einige Hersteller diesen Weg gehen. Für den Aufbau Ihrer Klassenhierarchien ist es jedoch eine Einschränkung, die unwillkürlich zu Problemen führt. Der Königsweg führt hier über die Implementierung von Interfaces.

Transaktionen

Objekte kommen selten alleine daher. Fast immer stehen sie in Beziehung zu anderen Objekten. Gibt es in diesen Beziehungen eine Änderung, dann müssen meist mehrere Datensätze geändert und in die Datenbank zurück geschrieben werden. Diese Operationen müssen nach außen hin „atomar“ erscheinen, es darf also keine Änderung oder Fehlerbedingung dazwischen kommen.

Mit dem Konzept der Transaktion können solche zusammengesetzten Operationen atomar durchgeführt werden. Während eine Transaktion stattfindet, können die beteiligten Tabellen und Datensätze von keiner anderen Transaktion angefasst werden; tritt ein Fehler bei einer Teiloperation auf, wird die gesamte Transaktion rückgängig gemacht.

Persistenz-Layer sollten Transaktionen unterstützen. Aber nicht nur das: Sie haben im Speicher eine Menge an Objekten vorliegen, die an einer Transaktion teilnehmen. Wenn Sie eine Transaktion abbrechen, sollten diese Objekte und ihre Beziehungen zueinander auf den gleichen Stand zurückgebracht werden können, wie zuvor.

Die Entwicklung dieses Features kostet die Hersteller von Persistenz-Lösungen einiges an Hirnschmalz. Aber es zeigt sich, dass die ernst zu nehmenden Kandidaten am Markt das Problem erkannt und gelöst haben.

An die Ungeduldigen nun ein letztes Wort: Danke für die Aufmerksamkeit und Tschüss. Bevor Sie aber gehen, lassen Sie sich noch folgenden Tipp ans Herz legen: Achten Sie bei Mapping-Tools darauf, welche Datenbanken unterstützt werden. Die Unterstützung mehrerer Datenbanken (SQL Server, Oracle, MySQL, eventuell Access) kann ein Hinweis dafür sein, dass die Lösung eine saubere Architektur aufweist.

Die Frage ist hier: Können Adapter für weitere Datenbanken selbst geschrieben werden? Für freiberufliche Entwickler ist es essentiell, wenn sie für verschiedene Kunden unterschiedliche Datenbanken ansprechen können. Es wäre doch traurig, wenn das nicht mit ein und demselben Mapping-Tool geht.

Für den Rest der Leser geben wir im Folgenden einen etwas tieferen Einblick, der es ermöglicht, zu verstehen, warum bes-

timtme Dinge so und nicht anders gelöst werden.

Objektrelationales Mapping

Die Zuordnung von Klassen- und Objektstrukturen zu Tabellen und Feldern in Datenbanken ist eine große Herausforderung für die Entwickler von Persistenz-Frameworks. So lange keine Vererbungshierarchien und polymorphe Konstrukte hinzukommen, ist das Thema noch relativ übersichtlich. Hier lassen sich die Beziehungen zwischen den Klassen direkt auf Datenbank-Konstrukte abbilden.

Abbildung 1 zeigt ein Modell mit relativ einfachen Beziehungen zwischen den Klassen. Die Beziehung zwischen Mitarbeiter und Adresse ist 1:1, ein Mitarbeiter hat eine Adresse und umgekehrt. Abbildung 2 zeigt, wie diese Beziehung in einer relationalen Datenbank abgebildet werden kann.

Die Tabellen *Mitarbeiter* und *Adresse* erhalten eine ID-Spalte, die jede Zeile der Tabelle eindeutig kennzeichnet. Solche IDs nennt man Primärschlüssel (Primary Key).

Die Zuordnung einer Adresse zu einem Mitarbeiter geschieht nun dadurch, dass die Mitarbeiter-Tabelle eine Spalte erhält, in ein Wert steht, der einem ID-Wert der Tabelle *Adresse* entspricht. Solche Werte nennt man Fremdschlüssel (Foreign Key). Die Adresse eines Mitarbeiters erhält man nun durch die Abfrage:

```
SELECT * FROM Adresse WHERE ID = 4711
```

Dabei kommt der Wert 4711 aus der Fremdschlüsselspalte *IDAdresse* einer Zeile der Mitarbeiter-Tabelle. Es ist gleichzeitig der Primärschlüssel des Adress-Datensatzes, der zu dem Mitarbeiter-Datensatz passt.

Die Beziehung zwischen Mitarbeiter und Reise wird anders abgebildet, da sie eine 1:n-Beziehung ist. Hier erhält die Tabelle für die Reisen eine Spalte, in der eine ID zu einer Mitarbeiter-Zeile steht. Man erhält dann alle Reisen zu einem Mitarbeiter mit der Abfrage:

```
SELECT * FROM Reise WHERE  
Reise.IDMitarbeiter = 4712
```

Hierbei ist 4712 der Primary Key der Mitarbeiter-Zeile, zu der die Reisen ermittelt werden sollen. Die Beziehung zwischen Reise und Land ist eine n:m-Beziehung. Eine solche Beziehung lässt sich nicht ohne eine Zwischentabelle abbilden, die zwischen Reise und Land vermittelt. Diese Tabelle enthält zwei Spalten, die jeweils Fremdschlüssel auf die Reise- und Land-Tabellen enthalten (siehe Abbildung 2). Man erhält dann alle Länder, in die eine Reise gegangen ist, mit der Abfrage:

```
SELECT Land.* FROM reLLandReise, Land  
WHERE reLLandReise.IDReise = 4711 AND  
reLLandReise.IDLand = Land.ID
```

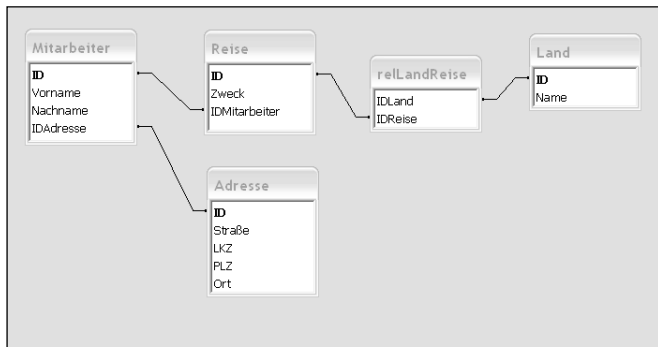


Abbildung 2 Abbildung von Klassenbeziehungen auf Tabellen

Umgekehrt erhält man alle Reisen, die in ein bestimmtes Land führten mit der Abfrage:

```
SELECT Reise.* FROM reLandReise,
Land WHERE reLandReise.IDReise =
Reise.ID AND reLandReise.IDLand =
4713
```

Das Modell in Abbildung 1 zeigt außerdem zwei Beziehungstypen. Die Beziehungen zwischen Mitarbeiter und Reise sowie zwischen Mitarbeiter und Adresse sind sogenannte Komposit-Beziehungen (Composite). Hierbei ist die Lebensdauer der Komposit-Bestandteile an die Lebensdauer des Eigners gekoppelt. Das bedeutet: Wird ein Mitarbeiter aus dem System gelöscht, müssen auch sämtliche Adressen und Reisen, die ihm zugeordnet sind, aus dem System gelöscht werden. Dies lässt sich in Datenbanken mit fortgeschriebenen Löschungen (Cascading Deletes) erzielen.

Da das Löschen von Objekten in Objektsystemen sehr oft mit bestimmten Vorgängen in der Business-Logik verknüpft ist, ist es keine gute Idee, wenn Persistenz-Frameworks die Cascading Deletes der Datenbanken einsetzen.

Anders funktioniert die Beziehung zwischen Reise und Land. Wird eine Reise aus dem System genommen, macht es durchaus Sinn, die Länder, durch die die Reise geführt hat, im System zu belassen. Solche Beziehungen nennt man Aggregat. Gute Persistenz-Lösungen unterscheiden zwischen Aggregaten und Kompositen.

Die Beispiele gehen übrigens implizit von einer bestimmten Einschränkung aus: Dass nämlich alle Felder einer bestimmten Klasse in genau einer Tabelle der Datenbank abgespeichert werden. Es gibt Persistenz-Lösungen, die das Verteilen von Klassen auf mehrere Tabellen erlauben. Selbst wenn das möglich ist, ist es nicht ratsam, von dieser Möglichkeit Gebrauch zu machen – und zwar nicht nur, weil es schrecklich langsam ist. Besser ist es, man ändert die Datenbankstruktur, oder, wenn das nicht erlaubt ist, schreibt man persistente Klassen, die die Tabellen exakt abbilden und formuliert seine Geschäftsklassen als Wrapper

für solche persistenten Klassen.

Abbildung 3 zeigt ein polymorphes Szenario: Während die Reise nur Kostenpunkte sieht, sind dieselben in unterschiedlichen Klassen mit verschiedener Logik implementiert. In der Literatur werden meist drei verschiedene Modelle beschrieben, wie man solche Szenarien auf relationale Datenbanken abbilden kann.

In zwei von diesen Modellen wird für jede Subklasse eine eigene Tabelle angelegt. Im ersten Modell enthält die Tabelle der Subklasse nur Spalten für die Felder, die in der Subklasse dazukommen. Im zweiten Modell enthalten die Tabellen für die Subklassen auch sämtliche Spalten für die persistenten Felder der Basisklassen. Im dritten Modell landen alle Klassen zusammen mit der Basisklasse in einer einzigen Tabelle. Dadurch lassen sich Beziehungen wie in Abbildung 3 sehr leicht abbilden, es entstehen jedoch aufgeblähte Tabellen mit extrem viel Null-Werten.

Das erste Modell ist ziemlich ineffizient, das zweite

ein brauchbarer Kompromiss, was Effizienz und Aufbau der Datenbank anbetrifft. Oft lässt sich das zweite und das dritte Modell wahlweise implementieren, nämlich dann, wenn das Persistenz-Framework das Mapping mehrerer Klassen auf eine Tabelle erlaubt.

Abbildung 4 zeigt, dass die Verhältnisse noch komplizierter werden, wenn Klassen Beziehungen von ihren Basisklassen erben. Die Beziehungen zwischen den Klassen werden dann über Zwischentabellen abgebildet, in denen neben den Fremdschlüsseln auch noch ein Type Code steht, der dem Persistenz-Framework zeigt, in welcher Tabelle die Objekte abgelegt werden (Abbildung 5).

Will man zum Beispiel alle Kostenpunkte einer Reise erhalten, dann ist für jede Subklasse von *Kostenpunkt* eine

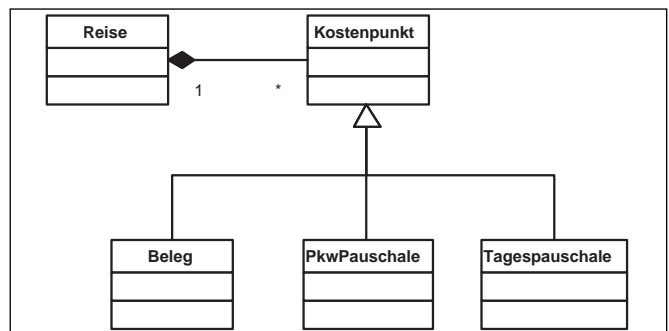


Abbildung 3 Ein polymorphes Szenario

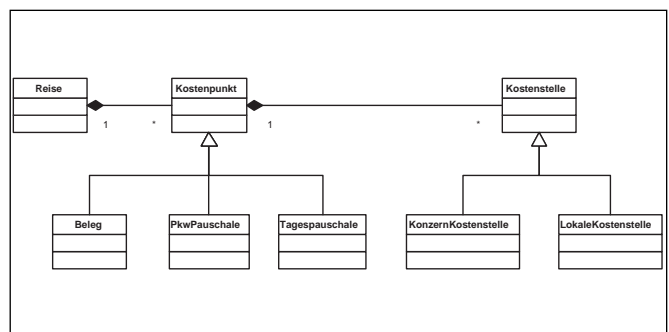


Abbildung 4 Vererbung von Beziehungen

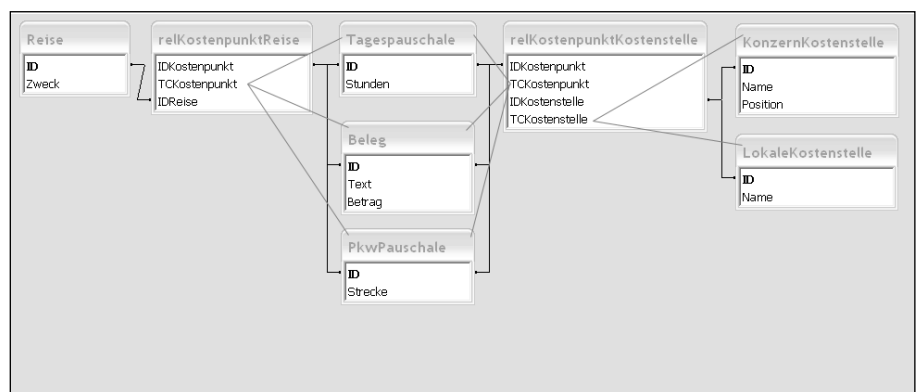


Abbildung 5 Darstellung polymorpher Beziehungen in einer relationalen Datenbank

Teilabfrage nötig. Will man jedoch alle Mitarbeiter ermitteln, die eine Reise unternommen haben, in der Belege angefallen sind, die in einer Kostenstelle mit einem bestimmten Namen abgerechnet wurden, dann lässt sich das in einer einzigen Abfrage abbilden, die allerdings schon etwas komplexer werden kann:

```
select distinct Mitarbeiter.* from
Mitarbeiter, Reise,
relKostenpunktReise,
relKostenpunktKostenstelle, Lokaleks,
KonzernKs
where Reise.IDMitarbeiter =
Mitarbeiter.ID and
relKostenpunktReise.IDReise =
Reise.ID and
relKostenpunktReise.IDKostenpunkt =
relKostenpunktKostenstelle.
IDKostenpunkt and
relKostenpunktReise.TCKostenpunkt =
relKostenpunktKostenstelle.
TCKostenpunkt and
(relKostenpunktKostenstelle.
IDKostenstelle = Lokaleks.ID
and relKostenpunktKostenstelle.
TCKostenstelle = 2
and Lokaleks.Name = 'ks3'
or relKostenpunktKostenstelle.
IDKostenstelle = KonzernKs.ID and
relKostenpunktKostenstelle.
TCKostenstelle = 4
and KonzernKs.Name = 'ks3');
```

Spätestens bei solchen Abfragen wird der normale Programmierer das Handtuch werfen und es einem Persistenz-Framework überlassen, das Mapping zwischen seinen Klassen und einer relationalen Datenbank vorzunehmen.

Noch ein Detail, das zum Thema Beziehungen wichtig ist: 1:n- oder n:m-Beziehungen werden normalerweise durch Container-Klassen verwirklicht. In .NET bietet sich dafür die Klasse *ArrayList* an. Sie hat nur einen Nachteil: Man kann nicht erkennen, welche Datentypen in dem Container stecken. Persistenz-Lösungen müssen dies jedoch wissen. Die Lösung ist – wie immer – ein Attribut. In diesem kann man nicht nur festlegen, welcher Datentyp in der *ArrayList* steckt, sondern auch, ob es sich um eine Komposit- oder Aggregatsbeziehung handelt.

Außerdem kann eine Klasse verschiedene Beziehungen zu ein und derselben anderen Klasse haben. Der Ultra-Spezialfall dieses Szenarios ist die Beziehung der Klasse *Mitarbeiter* zu sich selbst, einmal in der Rolle der Untergebenen und einmal in der Rolle des Chefs. Es kann also durchaus sinnvoll sein, einen unverwechselbaren Namen für Beziehungen zu vergeben. Auch dies lässt sich schön in einem Attribut verpacken.

Was den Zieltyp einer Beziehung anbetrifft, so wird mit .NET 2.0 Abhilfe geschaffen. Dort wird es parametrisierte Container geben:

```
collection <Mitarbeiter> c;
```

deklariert eine Collection, die Objekte vom Typ *Mitarbeiter* hält. Diese Collection implementiert dann automatisch das Interface *ICollection<Mitarbeiter>*. Sie funktioniert genauso, wie die *ArrayList*s in .NET 1.x, nur dass Sie beim Auslesen von Objekten keine Konvertierung auf den Typ *Mitarbeiter* mehr benötigen.

Objektidentitäten

Angenommen, Sie haben zwei Objekte a und b einer bestimmten Klasse vorliegen. Sie schreiben nun den Ausdruck

```
(a == b)
```

Nun stellt sich die Frage, was der Ausdruck eigentlich bedeutet. Was trifft zu, wenn der Ausdruck *true* ist?

- Es handelt sich um das selbe Objekt?

- Alle Mitglieder der beiden Objekte sind gleich?

Die Antwort ist: Kommt darauf an. Je nachdem, ob Sie die Funktion *Equals()*, die alle Klassen von *System.Object* erben, überschrieben haben oder nicht. Ist sie nicht überschrieben, wird getestet, ob es das selbe Objekt ist. Bei der Klasse *System.String* zum Beispiel wurde *Equals()* überschrieben. Hier kommt auch dann *true* zurück, wenn Sie zwei verschiedene Objekte vorliegen haben, deren Inhalt gleich ist. Im ersten Fall sind die Objekte identisch, im zweiten Fall sind sie gleich.

Betrachten wir den ersten Fall näher. Wie prüft man, ob zwei Objekte identisch sind? Ganz einfach: Hinter den Symbolen a und b verbirgt sich auf den meisten Systemen ein Zeiger auf den Speicherbereich, der ein Objekt repräsentiert. Sind die beiden Zeiger gleich, handelt es sich um den gleichen Speicherbereich, ergo um das selbe Objekt.

Diese Logik reicht für Persistenzsysteme nicht aus. Sie können ein Objekt in verschiedenen Instanzen einer Anwendung laden, vielleicht sogar auf unterschiedlichen Maschinen. Damit existiert es mehrfach, es ist aber dennoch das selbe Objekt. Die Identifikation über die Speicheradresse ist hierbei nicht besonders hilfreich. Persistente Objekte müssen jedoch eindeutig gekennzeichnet sein. Nach einer Änderung des Objekts muss eine ganz bestimmte Datenzeile in der Datenbank geändert werden – und diese gilt es zu finden.

In Datenbanken kennt man das Problem schon lange und richtet deshalb für alle Tabellen ID-Spalten ein. Meist sind das Integer-Spalten, wobei beim Einfügen eines neuen Datensatzes automatisch ein neuer Wert vergeben wird.

Können diese Datenbank-IDs für die Identifizierung von Objekten verwendet werden? Leider reichen auch sie nicht aus. Um ein Objekt eindeutig identifizieren zu

können, benötigt man zusätzlich die Typinformation. Zusammen mit der Datenbank-ID ergibt sich damit ein Schlüssel, mit dem Sie jedes Objekt eindeutig identifizieren können. Es wird Sie daher nicht erstaunen, wenn die meisten Persistenz-Systeme eine Kombination aus Typangabe und Datenbank-ID zur Kennzeichnung von Objekten verwenden. Ausnahmen bestätigen die Regel.

Im Fall von Objektspeichern müssen Sie sich nicht weiter um die Objekt-IDs (Oids) kümmern. Bei Mapping-Produkten aber ist die Behandlung der Oids ein zentrales Thema. Es muss eine Abbildung auf vorhandene Primärschlüssel in der Datenbank hergestellt werden können.

Die meisten Datenbanken unterstützen automatisch generierte IDs, die nach dem Speichern eines neu angelegten Datensatzes sofort wieder zurück gelesen werden können. Diese Vorgehensweise ist deshalb ziemlich günstig, weil bei konkurrierendem Zugriff nie eine ID mehrfach vergeben werden kann.

Es gibt jedoch auch Datenbanken, zum Beispiel die Oracle-Datenbank, die keine automatisch erzeugten IDs unterstützen. Man kann sie zwar mit Hilfe von Triggern erzeugen, es ist jedoch nicht ohne Kopfstände möglich, die ID eines eben erzeugten Datensatzes wieder zurück zu lesen.

Die Strategie in diesen Fällen ist, dass man in der Datenbank eine so genannte Sequence anlegt, das ist ein Wert, der bei jedem Zugriff um einen bestimmten Betrag hochgezählt wird. Der Client liest nun den aktuellen Wert der Sequence mit einer Abfrage ein und verfügt dann lokal über eine bestimmte Anzahl von IDs. Legt man die Sequence so an, dass sie jeweils um den Wert n hoch gezählt wird, dann hat man mit einem Zugriff n IDs zur freien Vergabe. In der Praxis haben sich Inkrement-Werte zwischen 10 und 100 bewährt.

Eine andere Möglichkeit ist die Verwendung von GUIDs als Primary Key, die ohne Mithilfe der Datenbank erstellt werden können. Allerdings ist die Sortierung von Daten mit Hilfe von GUIDs eine Idee langsamer, als mit Hilfe von Integer-Werten.

Wie auch immer die Strategie aussieht, mit der die Datenbank zu ihren IDs kommt, ein Mapping-Tool muss drei Dinge beherrschen:

- Unterstützung automatischer IDs, wenn die Datenbank diese erstellen kann und diese abfragbar sind
- Unterstützung selbst generierter IDs
- Unterstützung verschiedener Datentypen, wobei im Grunde Integer, Guid, String und Raw Bytes ausreichen – letztere benötigt man eigentlich nur, wenn es keine native Guid-Unterstützung in der Datenbank gibt.

Attribute und Mapping

Nehmen wir einmal folgende Klasse:

```
class Mitarbeiter
{
    string vorname;
    string nachname;
    ....
}
```

Objekte dieser Klasse sollen vom Persistenz-Layer gespeichert werden. Es stellen sich nun mehrere Fragen:

- Kann der Persistenz-Layer Objekte dieser Klasse überhaupt speichern?
- Woher weiß der Persistenz-Layer, welche Felder der Klasse gespeichert werden sollen?
- Wohin sollen diese Daten gespeichert werden?

Markierung persistenter Klassen

Nicht alle Persistenz-Lösungen können alle Objekte speichern. Das hängt vom Konzept ab. Es gibt Lösungen, die schlichtweg alle Objekte speichern können.

Das ist gar nicht so schwer, denn ein Objekt kann mit Hilfe von Reflection untersucht werden. Legt man den Algorithmus rekursiv aus, können komplexe Teilobjekte analysiert werden.

Es spielt dabei keine Rolle, ob die Objekte aus eigenen Klassen oder aus fremden Bibliotheken kommen.

Diese Lösung hat aber Nachteile. Erstens ist Reflection langsam. Man könnte meinen, dass auf jeden Fall die Datenbank das engere Nadelöhr darstellt. Unsere Messungen haben aber ergeben, dass sich die Reflection bei einer großen Menge an Objekten in der Laufzeit unangenehm bemerkbar macht.

Der zweite Nachteil dieser Lösung liegt in der Dirty-State-Verwaltung. Sie können nicht erkennen, ob ein Objekt geändert wurde – es sei denn, Sie haben es dem Persistenz-Layer explizit gemeldet.

Es gibt Persistenz-Lösungen, die hier einen anderen Weg beschreiten. Sie erfordern es vom Benutzer, die Klassen, die für die Persistenz in Frage kommen, im Quelltext mit einem Attribut zu kennzeichnen. Nach dem Kompilierlauf kommt ein so genannter „Enhancer“ zum Einsatz, der Veränderungen am IL-Code der Assemblies vornimmt.

Dadurch sind diese Lösungen in der Lage, automatisch zu erkennen, ob sich der Status eines Objekts geändert hat. Sounds cool? Ist es auch. Aber es erfordert eine Menge Entwicklungsarbeit, einen solchen Enhancer zu schreiben. Deshalb ist es nicht verwunderlich, dass es nur wenige Hersteller gibt, die mit diesem Feature aufwarten.

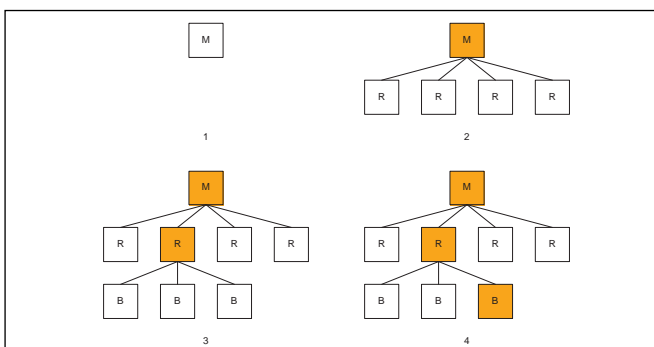


Abbildung 6 Lazy Loading von Objekten

Was sind Hollow-Objekte

Beim Lazy Loading wird von den Objekten zunächst nur die Objekt-ID geladen. Erst beim Zugriff auf persistente Felder der Klassen werden die Objekte nachgeladen. Abbildung 6 zeigt, wie das vor sich geht. Die weißen Kästchen stellen Hollow-Objekte dar, die orangenen Kästchen sind geladene Objekte. Der Beispiel-Code stammt von NDO (siehe [1]):

```
Mitarbeiter.QueryHelper qh = new
Mitarbeiter.QueryHelper();
Query q = pm.NewQuery(typeof(Mitarbeiter), qh.Name +
Query.Op.Like + "Müller");
Mitarbeiter m = (Mitarbeiter) q.ExecuteSingle();
// Zustand wie in Abbildung 6, links oben
Console.WriteLine(m.Vorname);
// Zustand wie in Abbildung 6, rechts oben
Reise r = (Reise) m.Reisen[1];
// Zustand unverändert
Console.WriteLine(r.Zweck);
// Zustand wie in Abbildung 6, links unten
Beleg b = (Beleg) r.Belege[2];
// Zustand unverändert
Console.WriteLine(b.Text);
// Zustand wie in Abbildung 6, rechts unten
```

Es sei an dieser Stelle darauf hingewiesen, dass eine solche Verwaltung nur für private Felder einer Klasse lückenlos funktioniert, weshalb einige Persistenz-Lösungen auch nur die Speicherung privater Felder zulassen.

Mit dieser Einschränkung kann man gut leben, da private Felder jederzeit mit öffentlichen oder protected Properties nach außen gereicht werden können.

Eigentlich ist es ja eine Grundregel der objektorientierten Programmierung, dass der Status von Objekten privat zu halten ist. Das erleichtert unter anderem die Fehlersuche in komplexen Systemen. Vielleicht kennen Sie das Szenario, dass ein

Bug auf einen ungültigen Zustand einer Variablen zurückzuführen ist, aber niemand weiß, wo diese Variable auf den ungültigen Wert gesetzt wird. Mit privaten Variablen können Sie das Tracing in solche einem Fall auf genau eine Klasse beschränken.

Damit Sie auch Objekte von Klassen speichern können, die aus fremden Bibliotheken stammen, erlauben es die Enhancer-basierten Lösungen in der Regel, dass persistente Klassen Teilobjekte beliebiger Typen enthalten können. Sie stellen dann einen Mechanismus zur Verfügung, mit dem Sie die Elternobjekte beim Persistenz-Framework als geändert anzeigen können.

Markierung persistenter Felder

Die Reflection-basierten Systeme werfen noch eine Problemstellung auf: Welche Mitglieder sollen eigentlich gespeichert werden?

Eine gute Annäherung an eine Lösung erzielt man, indem man alle privaten Felder abspeichert.

Die Lösungen, die mit Attribut-Markierungen arbeiten, könnten zum Beispiel ein Persistenz-Attribut anbieten, mit dem die Felder, die gespeichert werden sollen, markiert werden.

Besser ist es jedoch, alle privaten Felder einer als persistent markierten Klasse als persistent zu betrachten, es sei denn, man markiert sie explizit als transient. Da meist alle privaten Felder gespeichert werden sollen, kommt man bei der Codierung mit sehr wenigen Attributmarkierungen aus.

Mapping

Objektspeicher sind für den Benutzer Black Boxes. Es ist also völlig egal, wo bestimmte Daten abgelegt werden. Beim objekt-relationalen Mapping (OR-Mapping) ist das anders. Oft müssen vorhandene Datenbank-Strukturen verwendet werden.

Es muss also irgendwo niedergelegt sein, in welcher Tabelle und in welcher Spalte bestimmte Informationen landen sollen. Und hier kommt eine laute Warnung: *Lassen Sie die Finger von Produkten, die mit Attributen im Quellcode festlegen, wohin die Daten gespeichert werden.*

Diese Informationen haben in Ihrem Anwendungs-Code nichts zu suchen. Sie überfrachten Ihren Code mit Informationen, die nichts mit der Business-Logik zu tun haben. Außerdem müssen Sie nach jeder Umbenennung einer Spalte oder Tabelle Ihre Anwendung rekompilieren.

Ernst zu nehmende Ansätze gehen daher den Weg einer externen Mapping-Datei, die am besten im gleichen Verzeichnis liegt, wie Ihre Anwendung. Meist sind es Xml-Dateien, die leicht per Hand oder mit einem visuellen Mapping-Tool geändert werden

können. Lesen Sie hierzu auch den Abschnitt „Objektrelationales Mapping“.

Abfragesprachen

Es ist offensichtlich, dass SQL keine passende Abfragesprache für Persistenz-Lösungen ist. Sie wollen ja nicht mit Tabellen, Spalten und Zeilen hantieren, sondern mit Objekten. Statt umständlich über Joins verwandte Zeilen zu erfragen, wollen Sie simpel über Relationen von Objekt zu Objekt navigieren.

Nehmen wir als Beispiel das Szenario in *Abbildung 1*. Die Mitarbeiter-Klasse verfügt über ein `IList`-Feld, das die Reise-Objekte enthält. Wenn Sie über dieses Feld in die Reiseobjekte navigieren, sollten diese automatisch nachgeladen werden – Sie benötigen keine extra Abfragen dafür. Ebenso ist es mit Reisen, Ländern, Belegen etc.

Andererseits soll es möglich sein, zum Beispiel Mitarbeiter zu finden, die Reisen in bestimmte Länder vorgenommen haben. Irgendwie müssen Sie solche Bedingungen formulieren können.

Wichtig daran ist: Diese Bedingungen dürfen auf keinen Fall Spaltennamen der Datenbank enthalten – das würde ja alle Bemühungen zunichte machen, die Mapping-Informationen aus der Anwendung heraus zu halten.

Statt dessen beziehen sich Bedingungen am besten auf persistente Felder von Klassen. Die zugehörigen Spaltennamen sollten von der Persistenzschicht aufgrund der Mapping-Information berechnet werden.

Ein sinnvoller Abfragemechanismus ermöglicht es also, einen *System.Type* als gewünschten Resultattyp anzugeben und Bedingungen zu formulieren, die sich auf die persistenten Felder der Klassen beziehen. Hierbei sollte man auch über Beziehungen navigieren können, also zum Beispiel: Gib mir alle Mitarbeiter, die Reisen nach Spanien vorgenommen haben.

Leider gibt es keinen Standard zur Abfrage von Objekten. Es gab zwar einmal einen Versuch verschiedener Hersteller, unter dem Namen ODMG (Object Database Management Group) [3] verschiedene Aspekte der Persistenz zu normieren, unter anderem die so genannte OQL (Object Query Language). Diese Sprache hat sich jedoch noch nicht einmal bei den Mitgliedern der ODMG durchgesetzt. Dazu kommt, dass die ODMG ihren Standard im Java-Bereich durch JDO (Java Data Objects) ersetzt hat (siehe [4]). JDO aber verfügt über eine Java-ähnliche Abfragesprache, die unter .NET natürlich wenig sinnvoll ist.

Es ist zu erwarten, dass die Abfragesprachen der verschiedenen Produkte einen von drei Wegen beschreiten

werden:

- SQL-ähnliche Sprachen
- C#-ähnliche Sprachen
- XPath-ähnliche Sprachen

Dem Autor erscheint der SQL-ähnliche Weg am sinnvollsten, da die Where-Klauseln von SQL der gesprochenen Sprache sehr ähneln. Aber das ist letztlich Geschmacksfrage.

Wenn Sie mit einem Mapping-Tool arbeiten, dann sollte Ihnen bewusst sein, dass noch so mächtige Sprachmerkmale der Abfragesprache letztlich immer auf SQL abgebildet werden müssen; es macht daher wenig Sinn, auf Merkmale wie String-Funktionen besonderen Wert zu legen. Es müssen dafür alle in Frage kommenden Datensätze in den Speicher geladen und dort ausgewertet werden.

Statische Prüfung von Abfragen

Abfragen werden von allen Produkten, die dem Autor bekannt sind, als Strings übergeben. Dies ist jedoch eine potentielle Fehlerquelle, die viel Ärger bereiten kann. Angenommen, Sie codieren eine Abfrage nach dem folgenden Muster:

```
Query q =  
pm.NewQuery(typeof(Mitarbeiter),  
"name LIKE 'Müller'");
```

Hier bezieht sich der Ausdruck *name* auf ein persistentes Feld der Klasse `Mitarbeiter`. Nun ändert jemand die Klasse `Mitarbeiter` ab, so dass das Feld nicht mehr *name*, sondern *nachname* heißt. Ihre Abfrage ist nun falsch und wird zur Laufzeit eine Exception auslösen. Und Sie wissen: das passiert immer bei der Vorführung beim Kunden.

Um solche Show-Stopper zu vermeiden, hat man zur guten, alten SQL-Zeit extra Datenzugriffsschichten geschrieben, in denen alle Abfragen zusammengefasst wurden, damit möglichst keine Abfrage bei der Änderung eines Spaltennamens vergessen wird.

Solche Data Layers sind aber genau das, was man mit einer Persistenz-Lösung loswerden will. Was also nötig wäre, ist ein Mechanismus, der es erlaubt, die verwendeten Symbole vom Compiler überprüfen zu lassen.

Der Vorteil liegt klar auf der Hand: Man kann die Abfragen frei im Code verteilen. Sobald sich der Name eines persistenten Feldes ändert, erhält man einen Compiler-Fehler an der Stelle, an der die Abfrage formuliert wird.

Möglich wird so etwas durch die Verwendung von Enhancern. NDO generiert zum Beispiel so genannte *QueryHelper*, die als verschachtelte Klassen in den persistenten Klassen untergebracht werden:

```
Mitarbeiter.QueryHelper qh = new  
Mitarbeiter.QueryHelper();  
query q =
```

```
pm.NewQuery(typeof(Mitarbeiter),  
qh.name +  
Query.Op.Like +  
" 'Müller'");
```

Der Query-Helper erzeugt nichts anderes als die String-Repräsentation des Symbols *name*, aber mit Prüfung durch den Compiler.

Partielles Laden von Objekten

Viele Interessenten an Persistenz-Frameworks fragen nach der Möglichkeit, dass persistente Objekten zunächst nur teilweise aus der Datenbank geladen werden und bei Bedarf der Rest nachgeladen wird.

Obwohl dies technisch nicht so schwierig zu implementieren ist, nehmen die meisten Hersteller von solchen Mechanismen Abstand. Der Grund dafür ist, dass der Overhead für eine Abfrage meist größer ist, als das Laden von noch so vielen Spalten. Man spart also nichts.

Eine Ausnahme bildet das Szenario, wo Headerdaten zusammen mit einem großen Blob innerhalb einer Tabelle gespeichert werden, also zum Beispiel ein Bild zusammen mit einigen Daten, die zum Auffinden des Bildes nötig sind.

Die Mapping-Tools erlauben es in der Regel, mehrere Klassen auf ein und dieselbe Tabelle zu mappen. Man verwendet also zum Browsen durch die Bildheader eine bestimmte Klasse und zum Laden eines konkreten Bildes eine weitere Klasse. Beide Klassen sind auf die gleiche Tabelle gemappt, nur eben auf verschiedene Spalten.

Die saubere Lösung ist es jedoch, die Bilddaten in eine eigene Tabelle zu legen und zwischen den Header- und den Bilddaten eine Beziehung herzustellen.

Löschen von Objekten

Angenommen, in dem Beispiel in *Abbildung 1* wollen Sie einen Mitarbeiter löschen. Dies bedingt es, auch alle Reisen und somit alle Belege zu löschen. Außerdem müssen die Beziehungen zwischen den Reisen, die nun gelöscht werden, und den Ländern, in die die Reisen geführt haben, gelöscht werden.

Zusammen mit der Tatsache, dass diese Beziehungen bidirektional sein können und dass vor dem Löschen von Objekten eventuell noch ein Callback in die Anwendung erforderlich ist, wächst das Löschen von Objekten zu einem ganz schön komplexen Thema aus.

Es wundert daher nicht, dass die meisten Mapping-Produkte zunächst einmal sämtliche Objekte einer Hierarchie laden, bevor diese gelöscht werden. Das ist eine bittere Pille, aber offensichtlich wollen die Hersteller das Risiko vermeiden, dass Daten durch vorschnelles Löschen verloren gehen.

Eine Alternative wäre es, wenn der Benutzer explizit angeben muss, ob eine komplette Objekthierarchie direkt in der

Datenbank gelöscht werden soll. Wird dies nicht angegeben, werden sämtliche Objekte vor dem Löschen geladen. Damit das Laden der Objekte nicht zu viel Zeit verkonsumiert, kann die Technik der Prefetches verwendet werden. Um das zu verstehen, kommen wir nun noch einmal auf das Thema Lazy Loading zurück.

Lazy Loading und Prefetches

Lazy Loading ist eine Technik, in der zunächst nur die Oid von Objekten ermittelt wird, die Felder und Beziehungsvariablen der Objekte jedoch noch leer sind (siehe Kasten „Was sind Hollow-Objekte?“). Beim ersten Zugriff auf eine der Variablen bzw. Beziehungsfelder werden wie von Geisterhand die Felder des Objekts und sämtliche bezogenen Objekte als Hollow-Objekte nachgeladen.

Wir hatten weiter oben das Beispiel mit den 25.000 Objekten angegeben. Mit Lazy Loading wird verhindert, dass alle 25.000 Objekte geladen werden.

Aber angenommen, Sie hätten vor, die Gesamtreisekosten aller Mitarbeiter in einer Statistik darzustellen. Sie wissen also, dass Sie durch alle 25.000 Belege iterieren müssen. Mit Lazy Loading sind dafür 25.000 Abfragen nötig. Aber eigentlich könnte man die Datensätze für sämtliche Objekte mit drei Abfragen in den Speicher bekommen:

```
Select * from Mitarbeiter;  
Select * from Reise where  
IDMitarbeiter = 4177;  
Select * from Beleg where ...;
```

Es sollte also einen Weg geben, dem Persistenz-Framework zu sagen: Lade mir die Mitarbeiter, aber löse gleich die Beziehungen Mitarbeiter->Reise und Reise->Beleg mit auf. So etwas nennt sich Prefetch.

Nun kommen wir noch einmal zu dem Lösch-Szenario zurück. Angenommen, es wird ein Mitarbeiter gelöscht und für Reise-Objekte wurde im System ein Lösch-Callback definiert. Um den Mitarbeiter zu löschen, müssen daher erst alle Reise-Objekte, die zu ihm gehören, geladen werden, um die Callbacks ausführen zu können.

Ein Persistenz-System kann nun den Prefetch-Mechanismus verwenden, um für solche Szenarien gerüstet zu sein. Es werden einfach alle Reisen des Mitarbeiters mit einer Abfrage nachgeladen. Beachten Sie, dass dabei die Hollow-Objekte der Belege für all diese Reisen mit geladen werden. Wurde ein Lösch-Callback für die Belege definiert, so werden die Belege ebenfalls komplett geladen.

Wer hätte gedacht, dass das Löschen eines einzigen Objekts so kompliziert sein kann?

Statusverwaltung

Betrachten wir noch einmal das Lazy Loading. Wie kann überhaupt festgestellt werden, wann auf ein Feld zugegriffen wird? Mit Enhancern ist das relativ leicht zu ermitteln.

Im IL-Code von .NET gibt es gottseidank nur einen einzigen Mechanismus, der den Zugriff auf Felder ermöglicht: das ist die Anweisung *ldfld*. Die IL-Sprache ist eine Stack-Sprache. Vor der Anweisung muss daher das Objekt, auf das sich die Anweisung bezieht, auf den Stack gelegt worden sein (es muss sich ja nicht notgedrungen um *this* handeln).

Man kann nun die *ldfld*-Anweisung durch den Aufruf einer Accessor-Funktion ersetzen. Diese Funktion enthält die ursprüngliche *ldfld*-Anweisung, ruft aber vorher noch einmal eine Funktion im Persistenz-Framework auf, die sicherstellt, dass das Objekt auch wirklich geladen ist.

Wenn man es also schafft, für jedes persistente Feld ein Accessor-Paar (fürs Lesen und Schreiben) einzurichten und jeden *ldfld*- bzw. *stfld*-Befehl gegen einen Aufruf der Accessoren auszutauschen, erhält man eine lückenlose Aufzeichnung aller Zugriffe. Das setzt jedoch voraus, dass die Felder privat sind, denn sonst kann der Zugriff in Assemblies geschehen, die möglicherweise nicht mit dem Enhancer bearbeitet werden. Selbstverständlich sind auch Zugriffe via Reflection nicht feststellbar.

Die Lese-Accessoren stellen das Nachladen des Objekts sicher, die Schreib-Accessoren stellen zusätzlich sicher, dass das Objekt als „dirty“ markiert wird, und beim nächsten Abspeichern berücksichtigt wird.

Gibt es einen anderen Weg, als Enhancer, um diese Funktionen zu erzielen? Die Antwort ist: „praktisch nein“. Manche Lösungen erfordern es, dass alle persistenten Klassen von einer Basisklasse abgeleitet werden und sämtliche persistenten Felder von Datenklassen abgeleitet werden müssen. Diese Ableitungen dienen nichts anderem als der eben beschriebenen Buchhaltung. Das Problem an der Lösung ist, dass die Vererbungshierarchie nun voll im Dienst der Persistenz steht und nicht mehr im Dienst der Fachdomäne. Im Grunde sind solche Lösungen nicht brauchbar.

Ein weiterer Weg könnte ein Tool sein, das ähnlich wie Visual Studio bei den Typed Datasets vorgeht. Man beschreibt seine Klassen mit einer Art Metasprache oder mit Xml. Das Tool sorgt im Hintergrund dafür, dass eine C#-Datei generiert wird, die die Implementierung der Klasse enthält. Nachteil: Sie können nicht mehr durch den Code debuggen, den Sie selbst geschrieben haben, sondern debuggen durch den Code, der erzeugt wurde. Assemblies, die mit

einem Enhancer erweitert wurden, können hingegen mit den ursprünglichen Quelldateien gedebuggt werden.

Nur der Vollständigkeit halber möchte ich einen Lösungs-Ansatz erwähnen, der wohl kaum je Produktreife erlangen wird: Man kann sich mit Hilfe der Profiling-Schnittstellen in den JIT-Compiler der .NET-Runtime einklinken. Dadurch ist es möglich, einem Objekt Funktionalität unterzujubeln, ähnlich wie es ein Enhancer kann. Dazu muss aber die laufende Anwendung beim Kunden die Rechte eines Debuggers haben. Das ist für eine normale Geschäftsanwendung vom Sicherheitskonzept her inakzeptabel.

Das Fazit aus dieser Darstellung ist: Persistenz-Lösungen, die nicht mit Enhancern arbeiten, haben inhärente Schwächen, die sich nachteilig auf die Produktivität auswirken.

Kollisionen und Time Stamps

Die meisten Anwendungen heute laufen verteilt oder zumindest in konkurrierenden Szenarien. Es ist daher wichtig, festzustellen, ob ein Datensatz, der von einem Sachbearbeiter gerade bearbeitet wird, nicht in der Zwischenzeit von jemand anderem verändert wurde.

Die Command-Builder von ADO.NET erzeugen daher für Update- und Delete-Abfragen einen riesigen Rattenschanz an Bedingungen, der klären soll, ob ein Datensatz in der Zwischenzeit verändert wurde.

Angenommen, die Tabelle Mitarbeiter enthielte die Felder *Vorname* und *Nachname*. Ein Update-Befehl in ADO.NET lautet dann folgendermaßen:

```
UPDATE Mitarbeiter SET Nachname =  
@Nachname, Vorname = @Vorname, WHERE  
(ID = @Original_ID) AND (Vorname =  
@Original_Vorname OR  
@Original_Vorname IS NULL AND Vorname  
IS NULL) AND (Nachname =  
@Original_Nachname OR  
@Original_Nachname IS NULL AND  
Nachname IS NULL)
```

Sie können sich vorstellen, wie die Bedingungs-Strings aussehen, wenn die Tabelle 25 Spalten hat. Um solche Monster zu vermeiden, sollte man in den Tabellen so genannte TimeStamp-Spalten anlegen. In diesen Spalten muss nicht notwendigerweise eine Zeitangabe gespeichert sein. Es reicht, einen unverwechselbaren Code anzugeben, zum Beispiel eine GUID. Dieser Code wird für jeden Update eines Datensatzes neu erzeugt.

Wenn nun jemand einen Datensatz lädt, erhält er den TimeStamp des letzten Updates. Wenn nun jemand anderer den Datensatz in der Zwischenzeit ändert, dann ändert sich der Time Stamp. Nun reduziert sich der Bedingungsstring auf den Zustand

einer einzigen Spalte:

```
UPDATE Mitarbeiter SET Nachname =  
@Nachname, Vorname = @Vorname, WHERE  
(ID = @Original_ID) AND (TimeStamp =  
@Original_TimeStamp)
```

Wurde nun der Datensatz in der Zwischenzeit geändert, dann schlägt das Update fehl.

Wie auch immer solche Kollisionszustände entdeckt werden: Es sollte ein Mechanismus vorhanden sein, mit dem sich Kollisionen entdecken lassen. Elegant ist es, wenn die Anwendung beim Persistenz-Framework einen Callback registrieren kann, der aufgerufen wird, wenn eine Kollision passiert. Damit kann die Anwendung reagieren und dem Benutzer eine Korrektur ermöglichen.

Ein Nachteil dieser Lösung sollte nicht verschwiegen werden: Vorhandene Datenbanken müssen in ihrer Struktur verändert werden – es muss eine TimeStamp-Spalte dazukommen. Meist lässt sich eine solche Änderungen jedoch sehr gut mit Datenbank-Administratoren absprechen: Erstens ist die Änderung systematisch (in jeder Fachtabelle genau eine Spalte); zweitens lässt sie sich gut vermitteln, weil sie die Integrität der Datenbestände zu verbessern hilft.

Versionierung

Anwendungen ändern sich im Lauf der Zeit, es kommt Funktionalität dazu. Davon sind die Fachklassen und ihre Beziehungen zueinander betroffen. Dadurch ändert sich auch die Struktur der Datenbank (Schema). Es stellt sich also nach jeder Änderung das Problem, den vorhandenen Datenbestand in die neue Struktur zu bringen.

Hier zeigen Objektspeicher einen großen Vorteil. Sie unterstützen nämlich meist die Versionierung der Schemata. Daten, die im alten Schema abgespeichert wurden, verbleiben darin. Daten, die mit dem neuen Schema abgespeichert werden, landen dort. Das macht Abfragen zwar nicht gerade effizienter, aber es wird dadurch möglich, die Daten im laufenden Betrieb auf das neue Schema zu migrieren.

Mapping-Tools unterstützen in der Regel keine Versionierung, da es keine Unterstützung durch die relationalen Datenbanken gibt. Natürlich könnten Hersteller von Mapping-Tools nun selbst Hand anlegen. Jeder halbwegs brauchbare Ansatz führt jedoch zu Datenbank-Strukturen, mit denen eine Analyse der Datenbestände mit SQL oder OLAP-Werkzeugen nicht mehr möglich ist. Damit wird das Mapping als solches ad absurdum geführt.

Um Missverständnisse zu vermeiden: Die Schwierigkeiten entstehen nicht dadurch, dass die eine oder andere Fachtabelle eine Spalte mehr bekommt.

Probleme bereiten die Beziehungen zwischen den Tabellen. Es werden unter Umständen neue Zwischentabellen nötig, die die Tabellen auf eine andere Weise verknüpfen. Es reicht also nicht, nur einzelne Tabellen zu versionieren, es muss das gesamte Schema versioniert werden.

Die Lösung beim Einsatz von Mapping-Tools ist meist die harte Tour: Man erzeugt die neue Struktur mit neuen Tabellennamen. Dann überträgt man mit passenden Abfragen alle Daten aus dem alten Schema ins neue. Danach können die alten Bestände gelöscht werden. Während dieses Vorgangs ist ein Zugriff auf die Datenbestände nicht möglich. Es empfiehlt sich also in größeren Firmen, solch ein Upgrade am Wochenende vorzunehmen...

Bei lokalen Datenbanken ohne konkurrierendem Zugriff lohnt es sich, ein Upgrade-Programm zu schreiben, welches das Upgrade automatisiert, ähnlich dem Vorgang, der stattfindet, wenn man einmal wieder die neueste Version der T-Online-Software aufspielt.

Ein weiterer Nachteil der fehlenden Versionierung ist auch, dass sämtliche Anwendungen, die auf einem Datenbestand basieren, gleichzeitig an die neue Struktur angepasst werden müssen. Ist dies nicht möglich, müssen in zwei verschiedenen Datenbanken parallele Strukturen betrieben werden – alle wichtigen relationalen Datenbankprodukte unterstützen das Anlegen mehrerer Datenbanken. Die Herausforderung hierbei ist der Abgleich der Daten zwischen den beiden Datenbanken.

Es zeigt sich, dass die Versionierung das Killer-Kriterium ist, das die Entscheidung zwischen Objektspeichern und Mapping-Tools maßgeblich beeinflusst.

Callbacks und Erweiterungen

Viele Entwickler trauen den Persistenz-Frameworks nicht über den Weg, weil sie fürchten, in bestimmten Situationen keinen Einfluss auf den Ablauf der Dinge zu haben. Deshalb fordern auch viele Firmen den Source Code dieser Frameworks an.

Allerdings schaut sich kaum jemand den Source Code wirklich an. NDO besteht zum Beispiel aus mehr als 100.000 Zeilen Source Code. Es ist illusorisch, zu glauben, dass man schnell einmal durch Änderungen an bestimmten Stellen das Verhalten des Systems ändern kann.

Wichtiger ist es, dass ein Framework definierte Schnittstellen zur Benachrichtigung über bestimmte Ereignisse aufweist, und es ermöglicht, sich durch Erweiterungen in das System einzuklinken.

Das Minimum sind Callbacks, die zu den folgenden Zeitpunkten ausgeführt werden:

- nach dem Laden eines Objekts
- vor dem Speichern eines Objekts
- vor dem Löschen eines Objekts

Solche Callbacks werden normalerweise dadurch realisiert, dass die persistenten Klassen ein Interface implementieren. So landen die Callbacks innerhalb einer Klasse und können deren private Felder verändern.

So lässt sich zum Beispiel ziemlich leicht das Verschlüsseln bestimmter Felder verwirklichen. Die unverschlüsselte Information steht in einem transienten Feld. Vor dem Abspeichern wird sie verschlüsselt in ein persistentes Feld übertragen. Nach dem Lesen wird das persistente Feld entschlüsselt und die Information wiederum in das transiente Feld übertragen.

Darüber hinaus sollte es einen Callback geben, der im Fall einer Kollision benachrichtigt wird. Hierzu bieten sich Events an, die der Persistenz-Manager auslöst.

Bei Mapping-Tools sollte es einen Callback geben, der einen Connection-String abfragt. Connection-Strings enthalten Passwörter, die man normalerweise nicht im Klartext in eine Konfigurations- oder Mapping-Datei schreiben möchte. Man schreibt stattdessen Synonyme in die Konfigurations- oder Mapping-Datei. Der Callback gibt dann das Synonym an und erhält den entsprechenden Connection-String als Rückgabewert.

Die Mapping-Tools sollten außerdem Erweiterungsmechanismen aufweisen, die es ermöglichen, auf SQL-Ebene (bzw. auf der Ebene von ADO.NET) auf die Vorgänge einzuwirken. Wegen der unterschiedlichen technischen Ansätze der Produkte lassen sich solche Erweiterungen schlecht systematisiert darstellen.

Um dennoch einen Eindruck davon zu vermitteln, was möglich ist, beschreiben wir im Folgenden drei Erweiterungsmechanismen aus NDO zu den Themen:

- * Anbindung des Produkts an Datenbanken, deren Anbindung im Lieferumfang nicht enthalten ist
- * Anbindung von Datenquellen, die keine relationalen Datenbanken sind
- * Ein API für die Manipulation von Mapping-Informationen

Anbindung an neue Datenbanken

Meist unterscheiden sich die Zugriffe auf die Datenbanken nur in leichten Unterschieden des SQL-Dialekts, zum Beispiel dem Sonderzeichen zur Markierung von Variablennamen. NDO kapselt diese Unterschiede in so genannten NDO-Providern.

Um nun die Anbindung von NDO an einen speziellen ADO.NET-Provider zu bewerkstelligen, schreibt man eine Klasse, die das Interface *IProvider* implementiert.

Diesen Provider meldet man bei der Klasse *ProviderFactory* an. Beispiele gibt es für Oracle und MySql im Source Code. So ist es möglich, ohne großen Aufwand Datenbanken an NDO anzubinden, für die es vom Hersteller keine Provider gibt.

Anbindung beliebiger Datenquellen

Wer Datenquellen anbinden will, die überhaupt nicht auf SQL basieren, der kann sich mit zwei Klassen behelfen, die die Interfaces *IPersistenceHandler* und *IMappingTableHandler* implementieren. Man kann nun festlegen, dass bestimmte persistente Klassen diese neuen Handler benutzen.

Die Funktionen, die die Handler implementieren müssen, sind für ADO.NET-Entwickler sehr vertraut. Es handelt sich dabei um das Abspeichern von Änderungen in *DataTables* bzw. das Laden von Datensätzen in *DataRows*. Die Initialisierung dieser Klassen ist wohl die größere Herausforderung, weil man dazu die Mapping-Informationen auswerten muss.

Um einfach nur Einfluss auf die Vorgänge auf ADO.NET-Ebene zu nehmen, kann man sich Hüllklassen für die vorhandenen NDO-Handler schreiben und die eigentliche Arbeit an die ursprünglichen Handler delegieren. So lassen sich auch Protokoll-Funktionen implementieren.

Ein API für Mapping-Informationen

Bei NDO liegen die Mapping-Informationen in einer Xml-Datei. Es gibt eine Objektstruktur in NDO, mit der man diese Informationen auslesen und manipulieren kann. Man kann sich während des Programmlaufs vom *PersistenceManager* den gegenwärtigen Mapping-Objektbaum übergeben lassen.

Damit lassen sich sogar eigene Abfragesprachen schreiben, da NDO auch die Möglichkeit bietet, Objekte mit Hilfe von reinen SQL-Statements abzufragen. Zur datenbankunabhängigen Generierung der

SQL Statements kann man wiederum die Provider benutzen, auf die man über die *ProviderFactory* Zugriff hat.

Fazit: Mit den richtigen Erweiterungsschnittstellen wird ein Persistenz-System flexibel, ohne dass man gleich auf die Quellen zurückgreifen muss.

Die Checkliste:

Nun die am Anfang des Artikels versprochene Checkliste. In ihr werden noch einmal alle Punkte, die im Artikel näher besprochen wurden, übersichtlich zusammenfasst.

Datenbankstruktur bei Mapping-Tools – können vorhandene Datenbanken verwendet werden?

- ✓ Unterscheidung von Kompositen und Aggregaten?
- ✓ Wird Lazy Loading unterstützt?
- ✓ Gibt es eine Dirty-Status-Verwaltung?
- ✓ Funktioniert die Lösung ohne Ableitung von bestimmten Basisklassen?
- ✓ Werden Transaktionen unterstützt?
- ✓ Objekt-IDs: Werden automatische Datenbank-IDs unterstützt?
- ✓ Object-IDs: Werden selbst generierte Datenbank-IDs unterstützt – Integer, String, Guid bzw. Raw?
- ✓ Werden alle Beziehungen unterstützt? (1:1, 1:n, n:m – uni- und bidirektional)
- ✓ Wird Polymorphie unterstützt? Werden polymorphe Beziehungen unterstützt?
- ✓ Werden die Mapping-Informationen unabhängig vom Quelltext erfasst?
- ✓ Welche Datenbanken werden unterstützt?
- ✓ Ist die Abfragesprache einfach? Ist sie mächtig genug?
- ✓ Abstrahiert die Abfragesprache von Tabellen- und Spaltennamen der Datenbank?
- ✓ Gibt es eine Symbolprüfung zur Kompilierzeit für die Abfragesprache?
- ✓ Werden Kollisionen erkannt und per Event etc. angezeigt?
- ✓ Werden Callbacks unterstützt?
- ✓ Wie sieht es mit Erweiterungsschnittstellen aus?

Fazit

Dieser Artikel hat gezeigt, dass Persistenz-Lösungen nicht trivial sind: Sowohl für die Entwickler, die solche Lösungen schreiben, als auch für Sie, die Sie solche Lösungen einsetzen wollen und für sich die passende finden wollen. Die Checkliste in diesem Artikel mag dafür hilfreich sein.

Die Auseinandersetzung mit Persistenz-Lösungen lohnt sich jedoch, da sie die Produktivität bei der Entwicklung von datenzentrierten Anwendungen extrem steigern.

Es gäbe noch eine Menge mehr über Persistenz-Lösungen zu schreiben. Dieser Artikel sollte jedoch genügen, damit Sie selbst die Produktbeschreibungen zwischen den Zeilen lesen und Produkte evaluieren können. Dazu haben wir ein paar Links zu Persistenzprodukten angegeben – ohne Anspruch auf Vollständigkeit.

Bei der Recherche werden Sie schnell feststellen, dass Produkte, die über die wichtigsten Kern-Features verfügen, ihren Preis haben.

Wenn Ihnen bei Ihren Recherchen ein brauchbares Tool über den Weg läuft, das nicht in der Liste steht, zögern Sie nicht, uns ein E-Mail zu schreiben (mirkom@advanced-developers.de). Auch mit Fragen und Anregungen zum Thema müssen Sie sich nicht zurückhalten.

Literaturverweise

- [1] .NET Data Objects (NDO): www.netdataobjects.de.
- [2] ObjectSpaces: msdn.microsoft.com/data/objectspaces.aspx
- [3] Object Database Management Group (ODMG): www.odmg.org
- [4] JDO: java.sun.com/products/jdo