Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften

Verteilte Systeme und Informationssysteme

Diplomarbeit

Entwicklung eines Systems zur dynamischen Runtime-Persistenzcodeerzeugung

Uwe König
Student der Informatik (Diplom)
mail@uwekoenig.net
Matrikelnummer 4914111

Erstgutachter: **Prof. Dr. Norbert Ritter**
Zweitgutachter: **Dr. Axel Schmolitzky**

Inhaltsverzeichnis

1 Einführung und Gliederung

Die Beschäftigung mit der Persistenz von Daten innerhalb der Softwareentwicklung gewinnt zunehmend an Bedeutung. Dabei trat bereits früh die Erkenntnis ein, dass vorherrschende Paradigmen wie das der Objektorientierung in der Softwareentwicklung sowie das des Relationenkalküls in der Datenbanktechnik Probleme aufwerfen. Dazu zählen unter anderem die Abbildung der unterschiedlichen Strukturiertheit der Daten aufeinander. Mit zunehmender Komplexität der zu entwickelnden Software wird auch die Umsetzung von Persistenz aufwändiger und erfordert systematische Lösungsmöglichkeiten.
Während lange Zeit die Abbildung von Objekten auf Relationstupel manuell umgesetzt wurde, ist diese Vorgehensweise bei komplexen Klassenmodellen ineffizient. Die Verteilung der relevanten Codeteile über den gesamten Code, die hohe Fehleranfälligkeit und die geringe Varianz des Persistenzcodes führen zu hohem Aufwand bei Wartung sowie Anpassung des Codes an neue Anforderungen. Um dieser Problematik zu begegnen, wurden im Laufe der Zeit verschiedene Lösungsansätze entwickelt. Unter diesen haben insbesondere die objekt-relationalen Mapper in der Softwareentwicklung eine hohe Akzeptanz und entsprechende Verbreitung gefunden. Zwar automatisieren diese Mapper den eigentlichen Abbildungsvorgang weitgehend, allerdings bleibt die Problematik der Begrenzung von Aufwand und Kosten durch neue Anforderungen an ein Softwaresystem bestehen.

Die Problematik neuer Anforderungen an bestehende Software und an die einer Software zugrundeliegende Architektur bilden den zweiten Ausgangspunkt dieser Arbeit. In der Softwareentwicklung kann man die Kostenfaktoren einteilen in Kosten zur initialen Erstellung eines Softwaresystems sowie Kosten für dessen Wartung und Erweiterung. Während für die erste Kategorie praktikable Abschätzungsverfahren (siehe [MCC06]) wie zum Beispiel COCOMO II existieren, sind letztere bedeutend schwieriger abzuschätzen. Für die Gesamtkosten einer Anwendung über deren gesamten Lebenszyklus stellt dieser Anteil einen unbekannten Faktor dar. Moderne Softwarearchitekturen räumen dieser Problematik hohen Stellenwert ein. Insbesondere die Flexibilität, Wiederverwendbarkeit, Austauschbarkeit und Entkopplung einzelner Teilmodule nehmen im Bereich der Softwarearchitektur eine besonders wichtige Rolle ein. Als Lösungsansatz finden gegenwärtig sogenannte Plugin-Architekturen viel

Aufmerksamkeit. Gemeinsames Ziel aller Ansätze in diesem Bereich ist es, die Komplexität von großen Softwaresystemen zu reduzieren und ihre Wartung und Erweiterung zu erleichtern. Bei Plugin-Architekturen können einem einfachen Basissystem zusätzliche Module in Form von Plugins hinzugefügt werden. Dadurch ähnelt die Architektur einem Baukastensystem. Eine Anwendung kann aus den vorhandenen Bauteilen komponiert werden. Nicht vorhandene oder zusätzliche Funktionalität wird in Form neuer Plugins entwickelt, die die Funktionalität des Basissystems und diejenige bestehender Plugins nutzen kann.

Während die gegenwärtig verfügbaren OR-Mapper in ihrer Kernfunktionalität, der Abbildung von Objekten auf Relationen, relativ ausgereift sind, weisen sie in Bezug auf Flexibilität gegenüber der sie umgebenden Softwarearchitektur noch Mängel auf. Es muss festgestellt werden, dass die Mapper zwar mit monolithischen und Schichtenarchitekturen gut vereinbar sind, ihnen bei einer Benutzung innerhalb einer Plugin-Architektur jedoch Flexibilität bezüglich des Einsatzkontexts fehlen.
Diese Diplomarbeit hat zum Ziel, die Schnittstelle zwischen moderner Pluginarchitektur und OR-Mappern zu untersuchen. Dabei sollen einige der dort auftretenden Problematiken exemplarisch herausgearbeitet sowie Lösungsmöglichkeiten entworfen werden.
Die Arbeit ist dabei wie folgt gegliedert: Zunächst soll in Kapitel 2 die historische Entwicklung von Methodiken und Technologien zur Umsetzung von Objekt-Persistenz sowie das Spektrum vorhandener Ansätze skizziert und bewertet werden. Den Schwerpunkt bildet dabei die Speicherung von Objektdaten in relationalen Datenbanken mittels Abbildung dieser unterschiedlichen Datenstrukturiertheiten aufeinander. Ausgehend von diesem Ansatz und insbesondere der Mängel bestehender Umsetzungen wird in Kapitel 3 ein Katalog von Anforderungen entwickelt. Dieser soll in Beurteilungskriterien für objektrelationale Mapper (ORM) münden. Solche Kriterien sollen eine Abschätzung darüber ermöglichen, ob ein bestimmtes Framework für einen speziellen Einsatzzweck geeignet ist.

Für das Kriterium „Einsatz eines OR-Mappers innerhalb einer Plugin-Architektur“ und die damit verbundenen speziellen Problematiken wird dann im Hauptteil der Arbeit ein Lösungsansatz entwickelt, der auf dem Prinzip der dynamischen Erzeugung von Code zur Laufzeit basiert (Kapitel 4). Die Umsetzung des Lösungsansatzes soll in einem OR-Mapper münden, welcher prototypisch die entwickelten Lösungsansätze umsetzt. In Kapitel 5 wird die Umsetzung dynamischer Proxies behandelt, deren Code zur Laufzeit generiert wird. Kapitel 6

behandelt Ladestrategien, mit denen der Prototyp die notwendige Flexibilität erhält, die er für einen Einsatz innerhalb einer Plugin-Umgebung benötigt. Schließlich wird die Integration des Prototypen in eine Plugin-Architektur und die damit verbundenen Problematiken und Lösungsansätze in Kapitel 7 beschrieben. Die Arbeit schließt mit einer Bewertung der in der Arbeit entwickelten Prinzipien und des umgesetzten Prototypen sowie einem Ausblick.

2 OR-Abbildungstechniken in der Softwareentwicklung

Dieses Kapitel soll einen Überblick über den gegenwärtigen Stand im Bereich der OR-Mapper geben. Dabei werden die verwendeten Ansätze beschrieben und Vor– und Nachteile abgewägt.

2.1 Objekte, Relationen und Strukturbrüche

Seit längerer Zeit herrscht im Bereich der Softwareentwicklung die Sichtweise der Objektorientierung vor (vgl. [JAL05] S. 303). Es deutet nichts darauf hin, das dieses in naher Zukunft durch ein anderes Programmierparadigma verdrängt werden könnte, vor allem, da es sich als flexibel genug erwiesen hat, abweichende Programmierparadigmen zu integrieren (zum Beispiel Nachbildung prozeduraler Aufrufe durch Klassenmethoden, Integration funktionaler bzw. listenorientierter Verarbeitung (zu sehen beim Entwurf für C# 3.0 [HEJ05] oder bei F# [FSH06]).
Im Bereich der Datenspeicherung hingegen sind seit noch längerer Zeit die relationalen Datenbanksysteme (RDBMS) dominierend und haben einen hohen Reife- und Verbreitungsgrad erreicht (vgl. [SIL06] S. 37). Auch hier ist die Situation durch eine weitgehende Marktdominanz gekennzeichnet, in diesem Fall die des relationalen Datenmodells.
Werden nun beide Technologien gemeinsam benutzt, entsteht ein sogenannter Strukturbruch. Die beiden Ansätze zur Datenmodellierung sind nicht nahtlos miteinander vereinbar und benötigen Funktionalität zur Abbildung der Daten von einem Modell auf das andere (vgl. [FUS97a]). Diese Problematik wurde bereits früh erkannt und zwei Lösungsansätze wurden entwickelt: Kompensation sowie Aufhebung des Strukturbruchs.
Zur **Aufhebung des Strukturbruchs** wurden objektorientierte Datenbanksysteme (OODBMS) entwickelt (vgl. [VOS00] S. 261ff), die nicht mehr auf das relationale Datenmodell zurückgreifen. Es existieren eine ganze Reihe solcher OODBMS, jedoch haben diese Systeme zu keinem Zeitpunkt an der Marktdominanz und Verbreitung der RDBMS etwas ändern können. Eine große Rolle spielt dabei der Umstand, dass ein Wechsel des Speicherungssystems erhebliche Kosten und Aufwand verursacht. Zum anderen sind häufig bestehende Altsysteme so eng mit RDBMS verknüpft, dass ein Wechsel des Speicherungssystems eine komplette Neuentwicklung vieler Anwendungen erfordern würde.
Bei der **Kompensation des Strukturbruchs** geht es darum, einen Datenbroker zu

entwickeln, der zwischen objektorientierter Programmiersprache und relationalem Datenspeicher vermittelt. Ein solcher Datenbroker wird im Folgenden als OR-Mapper bezeichnet. Kernaufgabe eines OR-Mappers ist es, die bei der Modellierung einer Anwendungsdomäne entstehenden Entitäten, die Domänenklassen, auf relationale Datenstrukturen abzubilden.
Es existiert eine große Anzahl solcher OR-Mapper, sowohl kommerzielle als auch Open-Source-Systeme, und für so gut wie jede moderne Programmiersprache. Es deutet einiges darauf hin, dass zukünftig zwar die Marktverteilung zwischen RDBMS und OODBMS sich nicht mehr stark bewegen wird, das jedoch neu entwickelte Anwendungen, die ein RDBMS als Datenspeicher benutzen möchten, häufig auf OR-Mapper als Datenbroker setzen.
Dies hat vor allem seinen Grund in der OPH (orthogonal persistence hypothesis): „If application developers are provided with a well-implemented and well-supported orthogonally persistent programming platform, then a signifikant increase in developer productivity will ensue and operational performance will be satisfactory.“ (zitiert nach [ATK00]). Es wird später noch auf den Begriff der orthogonalen Persistenz und die mit ihm verbundenen Anforderungen an ein System zur Objektpersistenz eingegangen.
Mit der Benutzung eines OR-Mappers wird die Vorstellung verbunden, sowohl den initialen Entwicklungsprozess einer Anwendung zu beschleunigen als auch die im Laufe des Lebenszyklus einer Software anfallenden Wartungs- und Änderungsarbeiten zu reduzieren, welche sich aus entdeckten Fehlern und neuen Anforderungen ergeben. Ob dieser Effizienzgewinn realistisch eingehalten werden kann, ist bislang mangels Untersuchung nicht hinreichend bewiesen. Klar ist, dass eine Anwendung erst einen bestimmten Umfang haben muss, ehe sich der Einsatz eines OR-Mappers lohnt, da zunächst eine Einarbeitung in die Benutzung des Mappers notwendig ist. Den Vorteilen, die OR-Mapper unbestritten haben, stehen ebenso Nachteile hinsichtlich des Entwicklungsaufwands gegenüber:

- Jeder Entwickler muss zumindest über ein Mindestmaß an Kenntnissen über den eingesetzten OR-Mapper verfügen
- Mangelnde Flexibilität des OR-Mappers können dazu führen, dass Umgehungslösungen programmiert werden müssen

Es soll noch erwähnt werden, dass mit SQL 1999/2003 der SQL-Standard stark erweitert wurde, unter anderem um ein erweitertes Typsystem, getypte Tabellen und einen Vererbungsmechanismus für diese (näheres bei [TÜR03] und [TÜR06]). Dieser Standard wir von einigen sogenannten ORDBMS implementiert. Die Intention des Standards ist es, das relationale Datenmodell dem objektorientierten

anzunähern. Zwar ist dies teilweise gelungen, allerdings muss bei einer Benutzung von objektorientierter Wirtssprache und einem ORDBMS immer noch eine Abbildung der Datentypen des ORDBMS auf diejenigen der Wirtssprache vorgenommen werden, so dass die grundsätzliche Problematik mit der Erweiterung des SQL-Standards nicht gelöst wird.

2.2 Techniken zur Implementierung von OR-Abbildungen

Zur Umsetzung einer Abbildung von Klassen auf Relationen haben sich im Laufe der Zeit verschiedene technologische Ansätze entwickelt, die im folgenden erläutert werden sollen. Moderne OR-Frameworks verwenden zumeist Mischformen dieser Techniken.
Eines ist allen Ansätzen gemein: OR-Mapper bestehen aus verschiedenen Komponenten wie Anfrageverarbeitung, Cache und Transaktionsmanager. Die im weiteren erläuterten Methodiken unterscheiden sich vor allem in dem Punkt, wie ein OR-Mapper die Domänenklassen um die Eigenschaft der Persistenz erweitert. Von der jeweiligen Technik zur Erweiterung dieser Klassen hängt die Gestaltung der anderen Komponenten des OR-Mappers ab, diese sind eng miteinander verzahnt.

2.2.1 Manuelle Implementierung objektrelationaler Abbildungen

Anfangs wurden die Strukturunterschiede zwischen objektorientierten und relationalen Datenstrukturen mit manuell erstelltem Code überbrückt. Zusammenfassend können diese Ansätze mit dem Entwurfsmuster des Data-Access-Objects und des Active-Records beschrieben werden (vgl. [FOW03] S. 160ff und [NOC04] S. 33ff). Beispielsweise werden einer Klasse, die persistent gemacht werden soll, Methoden wie save() und load() hinzugefügt. Diese Methoden können dann an geeigneter Stelle im Code aufgerufen werden, um eine Instanz der Klasse zu speichern oder zu laden. Diese Vorgehensweise stößt jedoch an Grenzen, wenn die Objektmodelle komplex werden und die Anzahl der persistenten Klassen groß ist. Zudem ist der zu schreibende Persistenzcode für jede Klasse sehr ähnlich, durch die manuelle Erstellung aber fehleranfällig. Aus diesem Grund bietet sich hier Codegenerierung als Lösungsansatz an.
Dabei werden zunächst Metadaten über die zu persistierenden Klassen angelegt. Vor der Kompilierung wird dann durch einen Präprozessor der Code für diese Domänenklassen erstellt und anschließend die gesamte Anwendung kompiliert.

Dieser Ansatz lässt sich gut mit der Vorgehensweise der Model-Driven-Architecture (MDA) in der Softwareentwicklung vereinbaren. Bei der Softwareentwicklung nach MDA wird zunächst ein plattformunabhängiges Modell (platform-independet model, PIM) erstellt, welches unabhängig von konkreten Technologien ist. Dieses PIM wird nach bestimmten Transformationsvorschriften in ein PSM (platform-specific model) überführt und erst auf Basis des PSM wird der Code der Anwendung geschrieben. Diese Vorgehensweise erhöht vor allem die Portabilität von Anwendungen (vgl. [KLE03]).

Generell ist ein Vorteil der manuellen Erstellung von Persistenzcode, dass der Entwickler vorhandenes Kontextwissen über die Anwendung benutzt werden kann, um Optimierungspotenziale bei den Datenbank-Operationen voll ausschöpfen zu können. Bei einem bestehenden OR-Mapper werden für solche Optimierungen bisweilen Fähigkeiten des Mappers in nicht vorgesehener Weise benutzt, um die nötige Performanz zu erreichen. Ein solcher Gebrauch des Mappers zieht jedoch leicht Seiteneffekte nach sich, die schwer aufzufinden sind.
Nachteilig bei der vollständig manuellen Erstellung des Persistenzcodes ist die bereits in der Einführung genannte Fehleranfälligkeit sowie die weitgehende Monotonie des Codes, dessen Erstellung und Wartung durch Entwickler teuer ist.
Bei einer Codegenerierung sind allerdings ebenfalls Nachteile zu erwarten: In den meisten Fällen müssen Metadaten und manuell erstellte Codefragmente gepflegt und konsistent gehalten werden (siehe [STA05] S. 149f).

2.2.2 Code-Generierung

Unter Codegenerierung versteht man die Erzeugung von Quellcode durch ein Programm. Dieser Vorgang kann als eine parametrisierbare Transformation von Metadaten in Sourcecode betrachtet werden, die durch eine Konfiguration des Generators gesteuert wird. Tabelle 2.1 zeigt ein Beispiel mit einer einfachen Geschäftsklasse Konto:

UML-XML	C#-Code

<class name="Konto" access="public">
  <prop name="kontoNr" type="int" />
  <prop name="blz" type="int" />
  <prop name="eigner" type="Person" />
</class>
	public class Konto {
	  int kontoNr { get; set; }
	  int blz { get; set; }
	  Person eigner { get; set; }
	}

Tabelle 2.1: Beispiel für Code-Generierung aus Metadaten

Der XML-Export eines UML-Diagramms wird hierbei zu C#-Code transformiert. Dem Generator wird eine Konfiguration mitgegeben, zum Beispiel Zielsprache, Sichtbarkeit und Namensraum der Klasse sowie zu erzeugende Kommentare im Quellcode.

Ziel der Code-Generierung ist es vor allem, Änderungen am Quellcode zu vereinfachen, indem sie nur noch zentral an einer Stelle vorgenommen werden müssen. Wenn zum Beispiel einer Geschäftsklasse ein neues Attribut hinzugefügt wird, sollte dies nicht an verschiedenen Stellen im Code oder gar in verschiedenen Modellen der Software geschehen, wie im UML-Modell, im Metadatenmodell der Domänenklasse und im Quellcode.

Codeerzeugung kann zwar an verschiedenen Stellen die Entwicklung komplexer Software vereinfachen, bringt jedoch neue Probleme mit sich, die gelöst werden müssen. Es können große Mengen Quellcode erzeugt werden, deren manuelle Erzeugung fehleranfällig ist und der bei manueller Pflege statt der einfachen Neuerzeugung erheblichen Anpassungsaufwand benötigen würde. Andererseits ist offensichtlich, dass keine Anwendung komplett durch Codeerzeugung geschaffen werden kann. Generierter Code wird praktisch immer mit manuellem Code gemeinsam zu einer Anwendung kompiliert, und die Zusammenführung von generiertem und manuell erstelltem Code muss möglichst konfliktfrei und wartungsarm verlaufen. Dieses Zusammenführen ist unter Java nur mit zusätzlichen Tools effektiv machbar, wie zum Beispiel OpenArchitectureWare (siehe [OAW06]), welche vom Generator markierte geschützte Bereiche aus einem bestehenden Generat übernimmt und nicht alle Bereiche des Codes überschreibt. Bei C# bringt hier die Einführung von partiellen Klassen (vgl. [HEJ04] S. 607) unter C# 2.0 einen erheblichen Vorteil mit sich. Bei partiellen Klassen kann der Code einer Klasse über mehr als eine Datei verteilt sein, so dass man den Code in automatisch generierte und manuell erstellte Bereiche unterteilen kann. Ein Codegenerator erzeugt dann nur die ihm zugeordnete Datei, während der Entwickler nur die ihm zugeordnete Datei bearbeitet.

Codegenerierung ist insbesondere dort sinnvoll, wo im Softwareentwicklungs-prozess mehrere Zwischenstufen auf dem Weg von der abstrakten Modellierung zum fertigen Code durchlaufen werden. Ziel bei dieser Vorgehensweise ist es, Änderungen nur noch an einer zentralen Stelle vorzunehmen. Beispielsweise könnten Modelländerungen an einem UML-Modell vorgenommen werden. Durch die Benutzung eines Codegenerators sollten sich dann die Änderungen beim nächsten Generierungsvorgang im Code wiederfinden. Weitergehende Betrachtungen zu dieser Vorgehensweise finden sich bei [STA05].

Wie wird nun Codegenerierung im Zusammenhang mit OR-Mappern eingesetzt? Bei

einigen OR-Mappern wird folgende Vorgehensweise benutzt: Geschäftsklassen werden in Metadaten beschrieben oder grafisch modelliert (intern repräsentiert durch Metaobjekte). Aus diesen Metadaten wird dann Code für die Geschäftsklassen und ein passendes Datenbankschema erzeugt. Der Code muss dann noch ergänzt werden, um den generierten Klassen die Geschäftslogik hinzuzufügen (es sei denn, diese kann passend mitgeneriert werden). Auf der Basis dieses Generats kann dann die eigentliche Anwendung entwickelt werden. Änderungen an den Geschäftsklassen erfordern eine Neugenerierung und entsprechende Anpassungen bei deren Benutzung.
Codegenerierung wir vor allem bei Mappern eingesetzt, die entweder ein Objektmodell oder ein relationales Datenmodell als Ausgangsbasis für die Entwicklung einer persistenten Anwendung benutzen können. Die Vorgehensweise mit dem Objektmodell als Ausgangspunkt wird bezeichnet als forward-engineering, falls ein Datenbankschema die Ausgangsbasis der Anwendung heißt die Vorgehensweise reverse-engineering. Eine dritte Vorgehensweise ist das middle-out-engineering, bei dem aus Metadaten der persistenten Entitäten des Domänenmodells sowohl Klassen als auch ein passendes Datenbankschema generiert werden können. Näheres zu den drei Methoden bei [BAU05] S. 350 und [AMB03].

2.2.3 Ableitung von persistenten Basisklassen

Bei diesem Ansatz werden alle persistenten Klassen der Anwendungsdomäne von einer Basisklasse abgeleitet. Diese Basisklasse implementiert Methoden wie bspw. load() und store(). Ein zentraler Persistenzmanager weiß somit, dass diese Methoden auf einer Instanz einer Domänenklasse gefahrlos aufgerufen werden können.
Vorteil dieses Ansatzes ist es, dass der Entwickler der Geschäftsklassen sich nicht um die Implementierung der Persistenz kümmern muss. Außerdem stehen in den Geschäftsklassen die Methoden der persistenten Basisklasse zur Verfügung, zum Beispiel um den Status eines Objekts abzufragen und abhängig von diesem den Code kontextabhängig optimieren zu können. Ein solches API, das dem Entwickler der Geschäftsklassen zur Verfügung steht, weist sich in aller Regel als großer Vorteil aus, da oft die im OR-Mapper eingebauten Funktionen für spezielle Anforderungen nicht ausreichen.
Dieses dem Entwickler zur Verfügung gestellte API kann unterschiedlich granular gestaltet werden, je nach Sichtbarkeit der Methoden der Basisklasse (entweder public, protected- oder package-beschränkt, damit nur abgeleitete Domänenobjekte

bzw. der Persistenzmanager auf sie zugreifen kann).
Nachteilig bei diesem Ansatz ist, dass in die Klassenhierarchie der Geschäftsklassen eingegriffen wird, zum Beispiel könnten sich namensgleiche Methoden unbeabsichtigt überdecken. Eines der Ziele von OR-Mappern sollte es aber gerade sein, möglichst wenig Einfluss auf das Design der anderen Softwareschichten zu nehmen, insbesondere auf den Entwurf der Domänenklassen.

2.2.4 Benutzung von Reflection-APIs

Moderne Programmiersprachen wie Java und C# enthalten Funktionen zur RTTI (runtype type information) und ein Reflection-API. Mit diesen können zur Laufzeit nahezu alle Metadaten eines Objekts abgefragt werden, wie zum Beispiel:

- welchen Typ ein Objekt hat
- die Klassenhierarchie eines Typs
- welche Member ein Objekt hat (Methoden, Properties, Instanz- und Klassenvariablen)
- welche Interfaces ein Objekt implementiert

Außerdem ist mittels Reflection die Erzeugung neuer Instanzen einer Klasse möglich, ohne den Namen der Klasse zur Kompilierzeit zu kennen. Ebenso ist der dynamische Aufruf von Methoden eines Objekts zur Laufzeit möglich sowie das Setzen und Auslesen von einzelnen Datenfeldern.
Innerhalb eines OR-Mappers kann Reflection benutzt werden, um einzelne Datenfelder eines Objekts auszulesen und um aus der Datenbank gelesene Werte für ein neu erzeugtes Objekt zu setzen. Dabei müssen für den Mapper Metadaten angelegt werden, welche Namen und Typ der einzelnen Datenfelder einer Domänenklasse sowie deren Abbildungsvorschrift enthalten.

Reflection bietet als Vorteil eine nahezu grenzenlose Flexibilität, ohne dabei den Code zu verschachtelt werden zu lassen. Mit übersichtlichem Code sind hier sehr generische und elegante Lösungen implementierbar.
Als gravierender Nachteil erweist sich die Performance (vgl. für Java [SOS03], für DOT.NET [POB05]). Insbesondere der dynamische Aufruf von Methoden kann einen OR-Mapper zum Flaschenhals der Anwendung werden lassen.

Wie kein anderer der hier vorgestellten Ansätze spiegelt Reflection die Hauptproblematik der objektrelationalen Abbildung wider: zwischen hoher

Performanz und hoher Flexibilität muss ein zur Gesamtanwendung passender Kompromiss gefunden werden, beides gemeinsam ist nicht erreichbar.
Der bekannteste OR-Mapper, der hauptsächlich auf dem Einsatz von Reflection basiert, ist Hibernate/NHibernate. Um auf Reflection zu verzichten, kann für Hibernate die CGLib benutzt werden, welche Bytecode zur Laufzeit der Anwendung erzeugt und so einen Anteil der Reflection-Aufrufe umgehen kann (vgl. [BAU05] S. 386). Dabei wird dynamisch eine von der Domänenklasse abgeleitete Proxyklasse erzeugt, welche Memberzugriffe abfängt (siehe auch Abschnitt 5.3.3).

2.2.5 Code-Injektion

Bei der Code-Injektion wird den bereits kompilierten Domänenklassen zusätzlicher Code hinzugefügt, der dann später beim Ablauf der Anwendung die Persistenz dieser Klassen umsetzt. Zu diesem Zweck nimmt ein Postprozessor das Kompilat, analysiert es und fügt ihm eigenen Code hinzu. Um diese Ergänzung sinnvoll steuern zu können, müssen neben den persistenten Klassen auch Metadaten der Klassen gepflegt werden (es sei denn, der Code der Domänenklassen würde aus den Metadaten generiert, siehe Abschnitt 2.2.2). Diese Technologie hat auf den ersten Blick den Vorteil, die Entwicklung der Geschäftsklassen besonders wenig zu beeinflussen. Allerdings ist der Vorgang weniger transparent als beispielsweise Code-Generierung. Wie die Komponenten des OR-Mappers mit den Domänenobjekten im Laufe der Anwendung verfahren, muss sehr genau dokumentiert und dem Entwickler bewusst sein. Entsprechend ist die Ursachenforschung bei unerwartetem Verhalten und Fehlern schwierig, da die Intransparenz zu zeitaufwändigem Blackbox-Testen führt.
Unter Java ist JDO (Java Data Objects) das bekannteste Persistenz-Framework, welches diese Methodik anwendet. Für die Domänenklassen werden dabei XML-Dateien mit Metadaten gepflegt, welche wiederum vom JDO-Enhancer benutzt werden, um dem Kompilat Bytecode hinzuzufügen (siehe [JOR03] S. 10f). Beim Ablauf der Anwendung benutzt dann hauptsächlich die zentrale Instanz des JDO-Persistence-Managers diesen ergänzten Code zum Umgang mit den persistenten Objekten.

2.3 O/R-Mapper und Softwarearchitekturen

Ein OR-Mapper wird immer innerhalb der Softwarearchitektur einer Anwendung eingesetzt. Dabei stellt diese Architektur Anforderungen an ihre einzelnen Teile wie

Module und Komponenten. Diese Anforderungen unterscheiden sich stark zwischen einzelnen Software-Architekturen, sie führen zu unterschiedlichen Problematiken, die bei der Implementierung von Modulen und Komponenten berücksichtigt werden müssen.

2.3.1 Klassische Architekturen

Über einen langen Zeitraum hinweg wurde Software monolithisch konstruiert. Erweiterungen wurden in Form von neuem oder überarbeitetem Code hinzugefügt, dieser neu einkompiliert und die Software dann ausgeliefert. Diese Vorgehensweise lässt komplexe Software jedoch unübersichtlich werden und damit die Wartungskosten der Software schnell wachsen. Auch wenn diese gravierenden Nachteile nicht dadurch ausgeglichen werden: Die Anforderungen an Erweiterungsmodule innerhalb dieser Architektur sind die niedrigsten. Beim monolithischen Modell bestehen diese lediglich aus „inneren“ Anforderungen wie der korrekten Interaktion mit anderen Modulen. OR-Mapper, welche innerhalb eines solchen Softwaremonolithen ablaufen, müssen mit den geringsten konstruktionsbedingten Problemen rechnen, weil bei unpräzisen Abgrenzungen im Zweifelsfall die den OR-Mapper nutzenden Module angepasst werden und damit mangelnde Fähigkeiten des OR-Mappers kompensieren können.

2.3.2 Mehrschicht-Architekturen

In einer Schichtenarchitektur werden Softwaremodule zu einzelnen Schichten gruppiert. Diese Gruppierung erfolgt nach bestimmten Kriterien wie bspw. Model-View-Controller (vgl. [FOW03] S. 330ff). Die Schichten sind untereinander klar abgegrenzt und bei der Interaktion zwischen den Schichten muss Konsistenz bewahrt werden, um den Wartungsaufwand niedriger zu halten als beim monolithischen Ansatz. Bei Einhaltung dieser Anforderung bietet eine Schichtenarchitektur den Vorteil, dass die Implementierung einzelner Schichten mit geringem Aufwand gegen eine andere ausgetauscht werden können, insbesondere wenn Schnittstellen für die Schichten definiert werden.
Bei Datenbank-basierten Anwendungen mit Schichtenarchitektur werden die Teile des Codes, die direkt mit dem RDBMS interagieren, zu einer Datenhaltungsschicht zusammengefasst (vgl. [FOW03] S. 20). Darin enthalten sind OR-Mapper sowie externe Datenquellen mit deren Kapselung.
Diese Datenhaltungsschicht kommuniziert mit einer Anwendungsschicht, in der die

eigentliche Funktionalität für die Anwendungsdomäne umgesetzt wird. Zwischen den einzelnen Schichten herrscht das Kapselungsprinzip vor, die Anwendungsschicht sollte also kein Wissen über die Implementierungsdetails der Datenhaltungsschicht beim Anwendungsentwickler voraussetzen. Im Idealfall muss dieser also nicht wissen, was genau in der Datenhaltungsschicht geschieht und welche Technologien oder Frameworks dort eingesetzt werden. Daraus lässt sich direkt die Anforderung der Transparenz für den Anwendungsentwickler ableiten, die der OR-Mapper möglichst weitgehend umsetzen sollte.

2.3.3 Plugin-Architekturen

Der Begriff des Plugins existiert innerhalb der Softwaretechnik schon länger. Allerdings wurde früher unter einem Plugin eine kleine Erweiterung eines bestehenden größeren Softwaresystems verstanden, welches optional zum System hinzu installiert werden konnte. Ein Beispiel für ein solches System war der Webbrowser Mozilla, bei dem Plugins die Darstellung beispielsweise von Flash- oder SVG-Inhalten (Scalable Vector Graphics) übernehmen konnten.
Durch den Verbreitungsgrad der integrierten Applikationsserver unter J2EE sowie die Popularität und Dynamik der Eclipse-Plattform hat sich der Begriff der Plugin-Architektur für Softwaresysteme entwickelt. Dieser unterscheidet sich von obiger Beschreibung vor allem bezüglich der Rolle, die Plugins innerhalb des Gesamtsystems einnehmen. In Plugin-Architekturen ist ein Plugin nicht nur eine Ergänzung oder ein zusätzliches Feature – vielmehr wird die Gesamtfunktionalität einer Anwendung aus einzelnen Plugins zusammengebaut. Plugin-Architektur kann also als Konstruktionsweise zur Entwicklung großer Softwaresysteme verstanden werden.
Plugin-Architekturen sollen einer Reihe an hartnäckigen Problematiken begegnen, die sich während der Entwicklung von größeren Softwaresystemen häufig herausstellen.

Änderbarkeit und Erweiterbarkeit: bestehende Softwaresysteme werden regelmäßig mit neuen Anforderungen konfrontiert. Teilweise müssen bestehende Teile an veränderte Gegebenheiten angepasst werden, teilweise wird neue Funktionalität gefordert. Beim Entwurf eines Softwaresystems muss also davon ausgegangen werden, dass es später wachsen wird. Ein wichtiger Punkt ist dabei, in welcher Art und Weise dieses Wachstum erfolgt. Unkontrolliertes Wachstum der Software führt schnell zu Unübersichtlichkeit.
Unübersichtlichkeit: Unübersichtlichkeit der Software hat zur Folge, dass ein

Entwickler lange Einarbeitungszeit benötigt, um einen Überblick über die Anwendung zu gewinnen. Dies ist jedoch notwendig, um Änderungen und Erweiterungen vornehmen zu können. Unübersichtliche Software ist deshalb aufwändig bezüglich Wartung und Erweiterung und damit auch teuer.

Schwergewichtigkeit: Erweiterungen von Software werden nur selten von allen Benutzern benötigt. Trotzdem wird gegenwärtig in den meisten Bereichen mit Software gearbeitet, von deren Funktionalität nur ein geringer Anteil tatsächlich genutzt wird. Softwaresysteme sind also häufig für ihren Einsatzkontext überdimensioniert. Mit der Größe der Software wächst auch die Zahl der Fehler sowie die Belegung von Ressourcen.

Unorthogonaler Code: wenn der Code einer Anwendung getrennt werden kann in die Behandlung verschiedener Aspekte, kann von orthogonalem Code gesprochen werden. Bei orthogonalem Code sind beispielsweise technische Umsetzungen wie Datenbankprogrammierung von den umgesetzten Modellen der Anwendungsdomäne getrennt und haben wenig Abhängigkeiten untereinander. Solcher Code ist, wie im Schichtenmodell erläutert wurde, leichter wartbar und einzelne Implementierungen können leichter ausgetauscht werden.

Die angeführten Probleme sollen bei Plugin-Architekturen mit einer Reihe an Maßnahmen minimiert werden.

Plugin-Modularität: Erweiterung von Anwendungen erfolgt Plugin-zentriert: wenn möglich, wird eine Erweiterung als neues Plugin implementiert. Falls dies nicht möglich ist, sollte ermittelt werden, an welcher Stelle eine Erweiterungsmöglichkeit für ein solches Plugin sinnvoll ist, so dass dieses ergänzt werden kann.

Bei einem gelungenen Entwurf der Plugins bleiben Änderungen räumlich begrenzt, auf das Plugin selbst und diejenigen Plugins, die direkt mit dem zu ändernden interagieren. Änderungen, welche über diese Grenze hinaus propagiert werden müssen, deuten auf einen ungeeigneten Zuschnitt der Plugins hin, so dass dieser geändert werden kann.

Plugin-Interaktion: Die Gesamtanwendung besteht aus einem Rahmenwerk und Plugins. Das Rahmenwerk bietet dabei lediglich einfache Dienste an, zum Beispiel Dateizugriff und das Verwalten von Plugins. Die eigentliche Funktionalität der Anwendung wird in Plugins umgesetzt, diese Plugins interagieren über Mechanismen des Rahmenwerks miteinander. Beispiele für solche Mechanismen sind Extension-Points (synchron) oder Message-Channels (asynchron). Bei Extension-Points können Plugins festlegen, ob sie andere Plugins benutzen und an

welchen Stellen (Extension-Points) sie dies tun. Ebenso können Plugins auch eigene Extension-Points festlegen, die von anderen Plugins benutzt werden können. Systeme mit Message-Channel definieren einen oder mehrere Kanäle, über die verschiedene Arten von Nachrichten von Plugins verschickt werden können. Plugins können definieren, an welchen Kanälen sie auf welche Arten von Nachrichten lauschen und reagieren.

Plugin-Orthogonalität: Die zu entwickelnde Funktionalität der Gesamtanwendung kann zerlegt werden in abgegrenzte Aufgabenbereiche. Diese einzelnen Bereiche können Plugins zugeordnet und in diesen umgesetzt werden. Die Dekomposition der Gesamtfunktionalität der Anwendung sollte so vorgenommen werden, dass die Funktionalitäten der Plugins zueinander orthogonal ist und damit Funktionalität nicht auf verschiedene Plugins verteilt wird. Beispielsweise kann eine Gruppe von Plugins technische Basis-Funktionalität umsetzen, wie zum Beispiel Kommunikation mit externen Serverdiensten oder Dateizugriffe. Die eigentliche Domänenanwendung wird in anderen Plugins umgesetzt, welche diese technisch orientierten Basis-Plugins benutzen. Durch einen einheitlichen Interaktionsmechanismus zwischen den Plugins können solche Basis-Plugins bei Bedarf leicht gegen andere Implementationen ausgetauscht werden.

Erkennbares Ziel der Plugin-Architektur ist es, ein System für die im jeweiligen Kontext benötigte Funktionalität aus einzelnen Plugins 'zusammenstecken' zu können. Dadurch weist eine Plugin-Architektur im Gegensatz zu anderen deutlich größere Freiheitsgrade auf. Zum Zeitpunkt der Entwicklung ist der spätere, konkrete Einsatzkontext des Systems schwer abschätzbar. Es ist nicht bekannt, von welcher Art die Plugins sind, die zukünftig entwickelt werden und zu welchem Zweck eine bestimmte Konstellation von Plugins eingesetzt werden kann. In Abschnitt 3.5.3 wird näher erläutert, welche Folgen dies für einen in einem solchen Kontext eingesetzten OR-Mapper hat.

3 Anforderungen und Problematiken bei OR-Abbildungstechniken

In diesem Kapitel werden Anforderungen, die an OR-Mapper typischerweise gestellt werden, näher erläutert. Diese Anforderungen ergeben sich zum einen Teil aus den Problematiken, die bei jeder Abbildung von Objekten auf Relationen entstehen. Der andere Teil ergibt sich aus dem Kontext, in dem ein OR-Mapper eingesetzt werden soll. Kein OR-Mapper bildet ein Programm für sich, er ist immer nur ein Hilfsmittel für die eigentliche Anwendung (im folgenden als Mutteranwendung bezeichnet). Den Kontext eines OR-Mappers bildet sein gesamtes Umfeld an Software, mit dem er unmittelbar Schnittstellen gemeinsam hat. Dies sind in aller Regel die Mutteranwendung und das Datenbanksystem, welches zur Speicherung der Daten benutzt wird. Hinzukommen können noch weitere Datenquellen oder externe TP-Monitore.
Atkinson hat in einem Aufsatz (vgl. [ATK00]) über die Arbeit am PJama-Projekt eine Reihe an Anforderungen entwickelt, die an ein System zur Objektpersistenz gestellt werden können. Diese Anforderungen sind sehr umfangreich. Zunächst sollen diese Punkte erläutert werden. Weiterhin soll überprüft werden, inwiefern OR-Mapper diesen Kriterien genügen.
In weiteren Abschnitten dieses Kapitels werden Abbildungsproblematiken beim OR-Mapping sowie die Rolle eines OR-Mappers im Softwareentwicklungsprozess behandelt. Die gewonnenen Erkenntnisse sollen schließlich in einer Reihe an Anforderungen münden, die realistischerweise an einen OR-Mapper gestellt werden können.

3.1 Anforderungen an Objektpersistenz

Viele Umsetzungen von Objektpersistenz entwickeln sich aus Anwendungen heraus, die objektorientierte Programmierung und relationale Speicherungssysteme miteinander verbinden. Dabei entstehen typische bottom-up-Ansätze, die nach und nach mit neuen Detailanforderungen wachsen und erweitert werden. Allerdings ist hierbei die Gefahr groß, dass auch marginal erscheinende neue Anforderungen das System an seine Grenzen stoßen lassen.

Neben diesen pragmatischeren Ansätzen wurde im Bereich der Forschung auch der Begriff der 'orthogonalen Persistenz' entwickelt. Darunter wird eine sehr enge Kopplung von Programmiersprache und Persistenzmechanismus verstanden, so dass man solche Systeme auch als 'persistente Programmiersprache' bezeichnen kann.
Für Umsetzungen von orthogonaler Persistenz wurde von Malcolm Atkinson ein Anforderungskatalog formuliert, der sehr anspruchsvoll ist. Er zeigt jedoch deutlich, was die entscheidende Intention bei der Entwicklung von Objektpersistenz ist und kann deshalb auch als Maßstab für pragmatischere und weniger umfangreiche Systeme dienen. Dies gilt insbesondere, weil in einer Reihe von Forschungsprojekten Systeme entstanden sind, die den Anforderungen Atkinsons weitgehend genügen (bspw. PJama [ATK00] oder PEVM [LMG00]) und sich somit die hohen Anforderungen Atkinsons als erfüllbar herausgestellt haben. Die Performanzeinbußen, denen diese Systeme gegenüber einer nichtpersistenten Programmiersprache unterliegen, sind überraschend gering (vgl. [LMG00] S. 27ff).
Es soll im Folgenden überprüft werden, welchen Stand verfügbare OR-Mapper bezüglich der einzelnen Punkte von Atkinson erreicht haben. Natürlich können bei der Entwicklung von Systemen zur Objektpersistenz die einzelnen Punkte unterschiedlich priorisiert werden. Dennoch lohnt es sich zu betrachten, inwiefern OR-Mapper diesen Anforderungen genügen.
Die Anforderungen nach Atkinson (nach [ATK00]) sind:

1. **Othogonality**: All instances of all classes must have the full rights to persistence.
2. **Persistence Independence**: The language must be completely unchanged (syntax, semantic and core classes) so that imported programs and libraries of classes work correctly, whether they are imported in sourceform or as bytecodes.
3. **Durability**: Application programmers must be able to trust the system not to lose their data.
4. **Scalability**: Developers should not encounter limits which prevent their applications from running.
5. **Schema evolution**: Facilities to change any class definitions and to make consequent changes to instance populations must be well supported.
6. **Platform Migration**: It must be possible to move existing applications and their data onto new technology.
7. **Endurance**: the platform must be able to sustain continous operation
8. **Openness**: It must be possible to connect applications to other systems and

to interwork with other technologies.

9. **Transactional**: Transactional operation is required in order to combine durability and concurrency and to support appropriate combinations of isolation and communication.
10. **Performance**: The performance must be acceptable.

Da OR-Mapper auf RDBMSen basieren, sind **Endurance (7)**, **Transactional (9)**, **Durability (3)** und **Scalability (4)** normalerweise befriedigend umgesetzt, zumindest soweit diese Punkte vom RDBMS unterstützt wird. Diese Anforderungen können sehr gut innerhalb der Implementierung des Mappers an das RDBMS delegiert werden. Zu den weiteren Anforderungen sind etwas ausführlichere Erörterungen notwendig:

Orthogonality: die Persistierbarkeit jeder Klasse ohne spezielle Anforderungen an diese ist bei keinem verfügbaren OR-Mapper gewährleistet. Dies liegt vor allem daran, dass bei Domänenklassen für die Persistierung spezieller Attribute (beispielsweise vom Typ Thread oder Stream) erheblicher Abbildungsaufwand notwendig ist, der oft mit den Mitteln der benutzten Programmiersprache nicht bewältigt werden kann (vgl. [ATK00]). Näheres zu dieser Problematik findet sich in Abschnitt 3.2.2.

Persistence Independence: Hier stößt die Vorgehensweise vieler OR-Mapper ebenfalls an ihre Grenzen: zwar bietet der Reflection-Ansatz die Möglichkeit, auch bereits kompilierte Klassen zu persistieren. Spätestens bei Klassen, bei denen sich das Orthogonalitätsprinzip nicht umsetzen lässt, scheitert ein OR-Mapper jedoch auch hier. Unter diesen Punkt fällt auch die Anforderung, dass die Benutzung eines OR-Mapper sich möglichst wenig im Code der Anwendung niederschlagen sollte.

Schema Evolution: Bezüglich dieses Punkts sind verfügbare OR-Mapper unterschiedlich weit fortgeschritten: insbesondere die Mapper, die ein forward-engineering benutzen, erlauben Schema-Updates, allerdings sind bei ihnen die Schema-Updates auf einfache Operationen wie das Hinzufügen oder das Löschen von Attributen beschränkt. Komplexe Refactorings von Klassen würden deutlich aufwändigere Lösungen erfordern, als sie bei den Mappern implementiert sind. Einen vielversprechenden Ansatz verfolgt hier DataObjects.NET (siehe [DAT06]), welches für die zu persistierenden Klassen ein internes Objektmodell erstellt, aus diesem ein Datenbankmodell bildet, welches schließlich in einem Datenbankschema für das benutzte RDBMS mündet. Nach Änderungen am Objektmodell werden das alte und das neue Datenbankmodell verglichen, das Ergebnis dieses Vergleichs wird benutzt, um die Datenbankstruktur zu aktualisieren. Mit dieser Vorgehensweise ist

es möglich, Schema-Updates bis zu einem bestimmten Grad an Komplexität zu unterstützen.

Platform Migration: Ein OR-Mapper ist natürlich immer auf mindestens eine konkrete Technologie angewiesen, nämlich das RDBMS, welches als Speicherungssystem verwendet wird. Normalerweise erfolgen die Datenbank-Zugriffe des OR-Mappers durch einen Adapter, der die Kommunikation zwischen der untersten Schicht des Mappers und dem RDBMS übernimmt. Dieser Adapter kann auf die Eigenheiten des benutzten RDBMS eingehen, proprietäre SQL-Erweiterungen nutzen und Inkonformitäten zum SQL-Standard kompensieren.

Plattformunabhängigkeit erwirbt sich ein OR-Mapper vor allem dann, wenn er Adapter für verschiedene bestehende RDBMS nutzen kann. Auch in diesem Punkt unterscheiden sich existierende OR-Mapper untereinander erheblich. Open-Source-Lösungen wie Hibernate/NHibernate bieten hier das breiteste Spektrum an, während kommerzielle Mapper sich auf die wenigen dominierenden RDBMS beschränken.

Bei den Systemen der 'orthogonalen Persistenz' wurden Interfaces für einzelne Speicherungssysteme (wie bspw. Shore oder Platypus [HBKZ00]) entworfen. Eine derart einheitliche Benutzung von unterschiedlichen RDBMS ist leider nicht in Sicht, weshalb ein OR-Mapper lediglich eine bestmögliche Kapselung anstreben kann.

Openness: Wenn man OR-Mapper als einen Teil einer komplexen Softwarearchitektur betrachtet, kann sich der Begriff der Offenheit nur auf die Schnittstellen des OR-Mappers mit anderen Teilen des Systems beziehen. Folgende Schnittstellen sind bei OR-Mappern üblicherweise vorhanden:

- Schnittstellen zum RDMBS: für CRUD-Operationen (Create, Read, Update, Delete) sowie zum Transaktions- und Verbindungs-Management
- Schnittstellen zur Anwendungsschicht: Zum Erzeugen, Löschen und Speichern von Instanzen persistenter Klassen
- manche OR-Mapper bieten Schnittstellen hinsichtlich der Transaktionsverarbeitung, zum Beispiel zu einer externen Transaktionsverwaltung für die Koordination verteilter Transaktionen
- manche OR-Mapper bieten den Aufruf von callback-Methoden für Domänenklassen an. Diese callback-Methoden werden vom Anwendungsentwickler implementiert und werden vom Mapper vor und nach dem Auslösen bestimmter Aktionen wie Laden, Speichern oder Löschen aufgerufen. Solche callback-Methoden eignen sich gut, um bspw. vor dem Speichern eines Objekts Konsistenz-Bedingungen zu überprüfen.

Hinsichtlich der Offenheit finden sich erhebliche Unterschiede zwischen den verschiedenen OR-Mappern. Es ist sehr genau zu überprüfen, welche Anforderungen eine konkrete Anwendung an den potentiellen Mapper stellt und inwiefern diese erfüllt werden.

Performance: Die Wichtigkeit der Performanz ist je nach Anwendungsfall unterschiedlich, zumindest solange ein Schwellwert nicht überschritten wird, zum Beispiel die Reaktionszeit auf Benutzereingaben wie „Jetzt Speichern“. Aufgrund ihrer Arbeitsweise sind OR-Mapper weniger performant als reine Datenbank-API-Programmierung, da durch das Mapping zusätzlicher Aufwand entsteht und weniger optimiert werden kann.
Für die Steigerung der Performanz der Anwendung werden von OR-Mappern eigene Caches benutzt. Ein solcher Cache kann jedoch nur dann Datenzugriffe beschleunigen, wenn er effizienter arbeitet als der Cache des RDBMS. Im Gegensatz zum RDBMS-Cache, der sehr universell ausgelegt ist, kann der Cache eines OR-Mappers Kontextwissen nutzen. Solches Kontextwissen kann ein Mapper aus der Analyse der aus der Anwendung heraus vorgenommenen Zugriffe gewinnen.

Es ist erkennbar, das OR-Mapper zum gegenwärtigen Zeitpunkt orthogonale Persistenz (OP) nur ungenügend umsetzen. Dies hat zweierlei Ursachen: Zum einen, weil die OP-Systeme grundlegende Komponenten wie Speicherungssysteme zielgerichtet auf den Einsatz mit OP-Systemen neu entwickelt haben. Zusätzlich wurde bei allen OP-Systemen eine eigens entwickelte Java Virtual Machine benutzt.
Der andere Grund ist, dass OP Persistenz von Klassen viel grundlegender behandelt, als dies bei OR-Mappern der Fall ist. Dies erfordert zwar bedeutend größeren Aufwand zur Umsetzung, erzielt aber auch einen Grad an Transparenz für den Anwendungsentwickler, den OR-Mapper konstruktionsbedingt nicht erreichen können. Gleichwohl bilden die oben erläuterten Anforderungen einen Maßstab für Objektpersistenz, an dem sich auch OR-Mapper grundlegend orientieren können.

3.2 Probleme der Abbildungssemantik

Die Basisfunktionalität eines OR-Mappers besteht in der Abbildung von Datentypen der Wirtssprache auf diejenigen des genutzten Speicherungssystems. Bereits auf dieser Ebene zeigt sich jedoch, dass beide Bereiche an verschiedenen Punkten Unterschiede aufweisen, welche nur auf den ersten Blick geringfügig erscheinen. Auf einige dieser Problematiken soll im Folgenden eingegangen werden.

3.2.1 Identitätsabbildungen

Bei der Abbildung von Objekten auf Relationen besteht die Notwendigkeit, Identitäten aufeinander abzubilden. Das Identitätsmodell von Wirtssprachen wie Java oder C# ist dabei in aller Regel nicht in dieser Form bei relationalen RDBMS wiederzufinden. Eine Nachbildung solcher Identitätsmodelle ist zwar bei vielen RDBMS mit Hilfsmitteln möglich, bringt aber eine Reihe von Problemen mit sich (vgl. [CEL05] S. 36). Falls bei der Abbildung bereits vorhandene Datenbestände eingebunden werden müssen, ist diese Vorgehensweise oft überhaupt nicht mehr möglich. Aus diesem Grund verwalten OR-Mapper für persistente Objekte ein konfigurierbares Identitätsmodell und ein oder mehrere entsprechende Attribute in der jeweiligen Relation, welche für diese den Primärschlüssel bilden.
Die ID-Generierung durch den Mapper bietet auch Vorteile, wenn eine Anwendung in einem Rechnerverbund läuft. Normalerweise garantieren Wirtssprachen Objektidentität (zum Beispiel in Form einer Speicheradresse oder eines Hashcodes) nur für den Bereich eines einzelnen Rechners. Für die Persistierung muss aber die Eindeutigkeit der Id innerhalb des gesamten Rechnerverbunds gewährleistet sein.
Einzelne Techniken zur Id-Generierung finden sich u.a. bei [MAR02] S. 105ff.

3.2.2 Abbildung einzelner Datenstrukturen

Kernaufgabe eines OR-Mappers ist die Abbildung von objektorientierten auf relationale Datenstrukturen. Dabei werden Attribute einer Klasse auf Attribute einer Relation abgebildet. Dies funktioniert verhältnismäßig problemlos für einige Primitivattribute wie Integer, Double oder Char. Für andere Primitivattribute wie Boolean muss eine eigene Abbildungssemantik implementiert werden, da diese Datentypen in vielen RDBMS nicht zur Verfügung stehen (siehe [CEL05] S. 104).
Bei der Abbildung des Typs String muss entschieden werden, ob auf Seite des RDBMS der längenbegrenzte Datentyp VARCHAR, ein CHAR fester Länge oder für den Fall, dass der String erhebliche Länge annehmen können muss, ein Text-BLOB in der Datenbank benutzt wird. Bei dieser Entscheidung müssen Implikationen für

die spätere Benutzung beachtet werden. So kann ein Text-BLOB nicht bei allen RDBMS mit einem Index versehen werden.
Eine weitere Problematik ist die Abbildung von NULL-Werten und nicht gesetzten Werten, insbesondere die Frage, wie nicht gesetzte Werte in der Datenbank gespeichert werden und wie NULL-Werte beim Auslesen aus der Datenbank auf Attribute eines Objekts abgebildet werden. Zusätzliche Komplexität ergibt sich daraus, dass in Hochsprachen wie Java und C# für primitive Datentypen wie int immer auch eine entsprechende Klasse existiert (im Falle von int ist dies Integer), die NULL sein kann (was beim int zumindest in Java und bei C# 1.1 nicht möglich ist). In Fällen semantischer Mehrdeutigkeit von NULL-Werten muss der Mapper Metadaten pflegen, um solche Ambivalenzen auflösen zu können.

3.2.3 Abbildungen vernetzter Datenstrukturen

Objekte bestehen nicht ausschließlich aus Primitivattributen, sondern können Referenzen aufeinander enthalten. Dies wirft für einen OR-Mapper eigene Problematiken auf. Eine Referenz zwischen persistenten Objekten muss sich in der Datenbank wiederfinden. Eine Referenz eines persistenten Objekts auf eine transiente Instanz hingegen ist nach dem Beenden der Anwendung verloren und beim nächsten Start der Anwendung nicht wiederherstellbar. Grundsätzlich besteht die Anforderung der transzendierenden Persistenz an OR-Mapper: ein Objekt, dessen Typ als persistent markiert ist und das von persistenten Objekten referenziert wird, sollte ebenfalls persistiert werden, ohne dass der Entwickler dies manuell tun muss. Problematisch ist, dass hierfür bei einer Speicherung eines großen und komplexen Objektgraphen alle Objekte auf ihren Status hin überprüft werden müssen. Der Aufwand hierfür kann erheblich sein. Deutlich einfacher wird dies für den OR-Mapper, wenn der Entwickler selbst dafür sorgen muss, dass das referenzierte Objekt vor der Referenzierung bereits als persistent markiert wird. Somit muss bei einer Speicherung in die DB nur noch eine Liste der geänderten Objekte durchlaufen werden (Laufzeit: O(n)) und nicht mehr ein kompletter Objektgraph traversiert werden (Laufzeit: O(n log(n))). Detailliertere Ausführungen zur Abbildung von Objektreferenzen auf relationale Datenstrukturen finden sich bei Ambler ([AMB03] S. 244ff.). Möglichkeiten zur Modellierung, Abfrage und Manipulation von Graphenstrukturen von Objekten direkt in der Datenbank sind bei Celko ([CEL05] S. 681ff) erörtert.

3.2.4 Abbildung von Vererbungshierarchien

Wenn Vererbungshierarchien von Klassen persistent gemacht werden sollen, gibt es hierzu verschiedene Methoden, diese auf Relationen abzubilden. Im Vorfeld muss dabei abgeschätzt werden, wie später auf diese Klassen zugegriffen werden soll. Prinzipiell gibt es drei Ansätze, eine Vererbungshierarchie abzubilden (vgl. [AMB03] S. 231ff):

A) eine Tabelle pro Klasse
B) eine Tabelle für die Oberklasse und eine weitere pro abgeleiteter Klasse mit den zusätzlichen Attributen der abgeleiteten Klasse und einem Fremdschlüsselattribut als Verknüpfung zur Tabelle mit den Attributen der Oberklasse
C) eine gemeinsame Tabelle für alle abgeleiteten Klassen und die Oberklasse mit einem Diskriminatorattribut, welches festlegt, von welcher Klasse die entsprechende Instanz ist

Wenn die Zugriffe überwiegend auf die Instanzen einer Unterklasse erfolgen, ist Ansatz A der performanteste, da in diesen Fällen keine Joins durchgeführt werden müssen. Ansatz B ist empfehlenswert, wenn häufig auf die Instanzen der Oberklasse zugegriffen werden muss, da bei solchen Zugriffen bei Ansatz A ein Union verwendet werden muss. Abbildungstechnik C ist für gemischte Zugriffe günstig, wenn auf dem Diskriminatorattribut ein Index liegt. Dies solte möglichst ein Bitmapindex (vgl. [SIL06] S. 520ff) wegen der geringen Wertstreuung sein. Allerdings sind bei diesem Ansatz sehr viele Attribute dauerhaft mit NULL belegt und der Platzbedarf der Relation ist entsprechend hoch.

3.3 Die Rolle der O/R-Mapper im Softwareentwicklungsprozess

Im Bereich der Softwareentwicklung existieren verschiedene Vorgehensmodelle (siehe bspw. [WIN05] S. 27ff). Bestimmte Eigenschaften sind den gegenwärtig vorherrschenden Prozessmodellen gemeinsam. Aus diesen ergeben sich Anforderungen an OR-Mapper, damit diese (wie andere Werkzeuge auch) möglichst gut in den Entwicklungsprozess integriert werden können.

Anforderungsermittlung: Am Anfang jeder Entwicklung steht die Ausarbeitung der Anforderungen, die die zu erstellende Software erfüllen soll. Diese Anforderungsermittlung reicht jedoch in der Praxis noch weit bis in den Zeitraum hinein, in dem schon modelliert und programmiert wird (vgl. [WIE05] 200f).

Modellierung der Anwendungsdomäne: Für das Einsatzfeld der zu erstellenden

Software wird ein Modell erstellt. Dieses soll eine gemeinsame Verständnisbasis schaffen zwischen Softwareexperten und Experten des Arbeitsfeldes, in dem die Software eingesetzt werden soll. Das erstellte Modell ist die Ausgangsbasis, auf der der Code erstellt wird.

Iterative Vorgehensweise: Zwar ist Softwareentwicklung idealerweise in verschiedene, zeitlich sich nicht überschneidende Phasen unterteilt. In der Praxis ist dieser Anspruch jedoch nicht einzulösen (vgl. [WIE05] S. 198ff). Statt dessen kommt häufig eine iterative Vorgehensweise zum Zug, bei der bereits früh Prototypen erstellt werden, um Verständnisunterschiede zwischen Anwendern, Auftraggebern und Entwicklern möglichst frühzeitig aufzudecken und zu beseitigen. Eine iterative Vorgehensweise sollte idealerweise auch Änderungen (zumindest in begrenztem Umfang) in späten Projektphasen ermöglichen, ohne dass die bislang investierte Arbeit verworfen werden müsste.

Ich möchte diese Punkte an einem praktischen Beispiel erläutern. Dem Beispiel liegt das Prozessmodell der MDSD (Model-driven software development, modellgetriebene Softwareentwicklung) zugrunde (vgl. [STA05]). Die modellgetriebene Softwareentwicklung orientiert sich stark an der MDA (model-driven-architecture). MDA betont die Wichtigkeit der sauberen Modellierung des Bereichs, in dem die zu erstellende Software später eingesetzt wird. Auf die initiale Modellierung muss also besondere Sorgfalt verwendet werden, da Unsauberkeiten im Modell später zu Fehlern und Unwartbarkeit führen können. Das erstellte Modell dient als Ausgangsbasis für die Erstellung des Quellcodes der Anwendung.

Nach der ersten Ermittlung über das Anwendungsfeld kann mit der **Modellierung der Anwendungsdomäne** begonnen werden. Hierzu wird bei hinreichend großen Projekten eine „domain specific language“ (siehe [STA05] oder [KLA06]) entworfen, mit der spätere Modelle formuliert werden. Ist der Aufwand dafür unvertretbar, kann natürlich auch mit Standard-UML modelliert werden. Mit dem entstandenen Modell als Basis wird versucht, einen Teil des Codes automatisch zu generieren, insbesondere jene Teile, deren Implementierung weniger anspruchsvoll ist. Bei vielen Domänenobjekten sollte diese Vorgehensweise möglich sein und zumindest zur Generierung von Klassenskeletten führen, welche hauptsächlich aus der Definition von Attributen bestehen. Diesen Klassenskeletten wird später die manuell implementierte Domänenlogik hinzugefügt.

Die meisten OR-Mapper arbeiten mit Deskriptoren, also Metadaten über die Klassen, welche persistiert werden sollen. Diese Metadaten lassen sich auch aus dem erstellten Domänenmodell generieren. Dies hat den Vorteil, dass sie nicht

nach jedem Iterationszyklus erneut manuell erstellt oder angepasst werden müssen. Außerdem ist bei dieser Vorgehensweise ein OR-Mapper leichter gegen einen anderen austauschbar, hierzu muss lediglich die Generierungsvorschrift geändert werden.
Bei einer iterativen Vorgehensweise muss mit Modelländerungen zu Beginn jeder Iteration gerechnet werden. Änderungen am Domänenmodell sollten an möglichst wenig Stellen vorgenommen werden müssen, im Beispiel also nur im UML-Modell und an den manuell erstellten Anteilen der Implementierung. Aus diesem Grund muss von vornherein überlegt werden, welche Änderungen am Domänenmodell wahrscheinlich sind und ob diese Änderungen seitens des OR-Mappers umsetzbar sind. Häufig vorkommende Änderungen am Domänenmodell sind das Hinzufügen von Attributen, das Ändern von Attributtypen, die Verallgemeinerung oder die Spezialisierung von Klassen, die Aufteilung einer Klasse in mehrere sowie Änderung der Kardinalität eines Attributs (bspw. von 1:1 nach 1:n).

Zusammenfassend kann man sagen, dass im Softwareentwicklungsprozess zunehmend Flexibilität und Dynamik hinsichtlich des Entwicklungsprozessmodells gefordert werden. Dies schlägt sich auch in den Anforderungen an die benutzten Werkzeugen wieder, damit diese die Entwicklung möglichst gut unterstützen.
Für OR-Mapper bedeutet dies vor allem, dass Änderungen im Domänenmodell schnell vorgenommen werden können. Idealerweise unterstützen OR-Mapper nicht-triviale Konsistenzprüfungen, einerseits statisch auf dem Datenmodell, andererseits dynamisch zur Laufzeit. Ein Beispiel hierfür sind die aus dem relationalen Bereich bekannten Data-Domains: ein Attribut einer Relation kann einem eingeschränkten Wertebereich zugeordnet werden, zum Beispiel muss eine Postleitzahl eine fünfstellige, positive Ganzzahl sein. Wenn versucht wird, dem Attribut einen Wert außerhalb des erlaubten Bereichs zuzuweisen, muss entsprechend eine Fehlermeldung aufgeworfen werden.

3.4 Anforderungen an O/R-Mapper

Nachdem in Abschnitt 3.1 allgemeine Anforderungen an Objektpersistenz analysiert wurden, soll in den folgenden Abschnitten auf spezielle Anforderungen an OR-Mapper eingegangen werden. Dabei werden zunächst die beiden Bereiche behandelt, welche direkt mit dem Mapper interagieren: das Speicherungssystem und die Domänenanwendung. Das Kapitel schließt mit Anforderungen der Softwarearchitektur, innerhalb derer ein Mapper eingesetzt wird.

3.4.1 Anforderungen des Speicherungssystems

Relationale Datenbanksysteme haben im Laufe der Zeit sehr effiziente Zugriffsmechanismen entwickelt, die allerdings nicht am objektorientierten Datenmodell ausgerichtet sind. Damit ein OR-Mapper möglichst effizient mit den Nutzdaten der Objekte umgehen kann, muss er diese Mechanismen beim Laden und Speichern der Objekte nutzen. Dabei müssen Kompromisse eingegangen werden, die auf dem Prinzip des Speicherungssystems begründet sind. Beispielsweise ist es für die Performanz von Lesezugriffen erstrebenswert, jedes Attribut einer Relation mit einem Index zu belegen. Andererseits werden damit Einfügeoperationen verlangsamt, da alle Indizes aktualisiert werden müssen. Auch bei Abfragen über mehrere Relationen muss ein OR-Mapper entscheiden, ob ein Join notwendig ist oder ob die Daten aus einer verknüpften Relation zu einem späteren Zeitpunkt gelesen werden können. Ein Beispiel ist eine klassische Master-Detail Beziehung wie die von Kunde zu Bestellung. Werden die Daten eines bestimmten Kunden geladen, müssen zunächst nicht notwendigerweise seine gesamten Bestellungsdetails mitgeladen werden. Dies könnte auch erst dann erfolgen, wenn auf das Bestellungs-Attribut in der Klasse Kunde zugegriffen wird. Wenn allerdings alle Kunden mit mehr als fünf Bestellungen geladen werden sollen, ist ein Join notwendig. In diesem Fall kann der ohnehin anfallende zusätzliche Aufwand für den Join genutzt werden, um die Bestellungsdaten der Kunden gleich mitzulesen. Damit entfällt ein späterer Datenbankzugriff auf die Bestellungsdetails.
Bei Einfüge- und Änderungsoperationen mehrerer Tupel einer Relation bieten RDBMS prepared statements an. Der Vorgang der syntaktischen Prüfung und Erstellung eines optimalen Ausführungsplans für das SQL fällt dabei DBMS-seitig nur einmal an. Dies sollte von einem OR-Mapper genutzt werden, um entsprechende Änderungen bei einem Abschließen der Transaktion zu gruppieren und mit prepared statements durchzuführen. Bei sehr häufig vorkommenden Operationen (zum Beispiel Punktzugriffe auf einzelne Instanzen einer häufig benutzten Klasse) kann es sinnvoll sein, die entsprechenden prepared statements zu cachen und wiederzuverwenden.

Des weiteren sind RDBMS für bestimmte Einsatzzwecke ausgelegt, diesen Umstand sollten OR-Mapper hinreichend berücksichtigen. Lange Transaktionen bei gleichzeitigem parallelen Zugriff auf Daten sind ein typisches Beispiel dafür, dass ein OR-Mapper hier in eine Zwickmühle geraten kann. In GUI-Anwendungen ist

längeres Editieren in geöffneten Fenstern üblich. Bei parallelem Zugriff auf diese Daten ist jedoch die Wahrscheinlichkeit hoch, dass Änderungen nicht geschrieben werden können, wenn die dahinterliegende Datenbanktransaktion über einen Zeitraum von mehreren Minuten läuft. Um der Problematik zu entgehen, muss ein OR-Mapper eine eigene Versionierung von Datensätzen pflegen und mit einer Versions-Id oder mit Zeitstempeln arbeiten.

Nach Möglichkeit sollten OR-Mapper mit unterschiedlichen RDBMS einsetzbar sein. Problematisch ist dies deshalb, da sich die einzelnen RDBMS bezüglich der Umsetzung der SQL-Standards unterscheiden, gleiches gilt für RDBMS-spezifische Erweiterungen. Wenn OR-Mapper solche proprietären Erweiterungen nutzen möchten, müssen diese gegebenenfalls für den Einsatz mit anderen RDBMS in einer Kapselungsschicht emuliert werden.

3.4.2 Anforderungen der Domänenanwendung

Von Seiten des Entwicklers der Domänenanwendung werden vor allem zwei Anforderungen an den OR-Mapper gestellt: einerseits der Anspruch der Transparenn. Der Mapper sollte möglichst wenig Einfluss auf den Entwurf der Domänenklassen haben. Des weiteren sollten keine Seiteneffekte des OR-Mappers im laufenden Betrieb Einfluss auf das Verhalten der Domänenobjekte haben.
Die zweite Anforderung betrifft die Schnittstellen des API, über welche die Domänenlogik mit der Persistenzschicht interagiert. Es muss ein Basissatz an Funktionen zur Verfügung gestellt werden, mit denen Routineaufgaben einfach und schnell umsetzbar sind (vgl. [CWA06] S. 19ff).
Solche Aufgaben sind:

- Persistieren von transienten Objekten
- Laden von einzelnen sowie von Gruppen von persistenten Objekten
- Löschen von persistenten Objekten
- Lesezugriff auf den Zustand der gerade laufenden Transaktion
- Lesezugriff auf den Zustand von einzelnen Objekten

Mit diesen Funktionen lassen sich die meisten zur Persistierung notwendigen Aufgaben bewältigen. Erweiterte Funktionalität ist vor allem dort nötig, wo bei Anwendung des einfachen APIs die Performanz nicht ausreichend ist oder Unzulänglichkeiten des OR-Mappers kompensiert werden müssen. Ein Beispiel hierfür sind lange Transaktionen, welche vor allem bei Web- und

Entwurfsanwendungen auftreten. Unter Entwurfsanwendungen werden Anwendungen verstanden, bei denen einzelne Benutzer über einen längeren Zeitraum an Daten arbeiten und diese Daten anschließend anderen Benutzern im Speicherungssystem zentral zur Verfügung gestellt werden, wie beispielsweise im Bereich Softwaretechnik oder CAD. Diese Bearbeitung kann sich über einen längeren Zeitraum erstrecken. Eine derart lang laufende Transaktion würde mit der Benutzung von Sperren die Daten für andere Benutzer blockieren. Mit der Benutzung einer optimistischen Sperrstrategie würden hingegen über den langen Zeitraum der Transaktion viele Ressourcen belegt, was bei vielen parallel arbeitenden Anwendern zu Knappheit führen kann. Bei Webanwendungen stellt sich die Problematik ähnlich, wenn komplexe Datensätze über mehrere Seitenaufrufe hinweg bearbeitet werden und ACID-Transaktionalität benötigt wird.
Zur Lösung dieser Probleme bieten einige OR-Mapper eine Offline-Funktionalität an. Bei dieser können vernetzte Objektmengen komplett von den Kernkomponenten des OR-Mappers abgelöst werden. Sie können dann bearbeitet und zu einem späteren Zeitpunkt wieder mit dem Mapper verbunden werden, wobei die Änderungen mit anderen Änderungen in der Datenbank synchronisiert werden müssen. Es muss hierbei also eine optimistische Synchronisationsstrategie verwendet werden.

3.4.3 Anforderungen der Systemarchitektur

Bereits in Anschnitt 3.3 wurde dargelegt, dass eine monolithische Architektur nur wenige spezifische Anforderungen an einen OR-Mapper stellt und dieser lediglich den Anforderungen des Softwareentwicklungsprozesses genügen sollte. Verfügbare OR-Mapper genügen auch weitgehend den Anforderungen, die eine Mehrschichtenarchitektur an eine Datenhaltungsschicht stellt, da sie klar abgegrenzte APIs bieten und transparente Persistenz umsetzen.
Was hingegen den Einsatz von OR-Mappern innerhalb einer Plugin-Architektur betrifft, so sind erhebliche Defizite bei den OR-Mappern festzustellen. Plugin-Architekturen wie bspw. die Entwicklungsplattform Eclipse besitzen bezüglich ihrer Plugins und des Einsatzkontextes Freiheitsgrade, die gegenwärtig von OR-Mappern nicht ohne weiteres unterstützt werden. Diese Freiheitsgrade stellen einen bedeutenden Unterschied zwischen Plugin- und anderen Architekturen dar. Im Gegensatz zu einem klassischen System, dessen Einsatzspektrum abgegrenzt werden kann, ist das potentielle Einsatzspektrum bei einem Plugin-System deutlich breiter.

Grundlage für den Einsatz eines OR-Mappers innerhalb einer Plugin-Architektur ist, dass der Mapper den Plugin-Mechanismus unterstützen muss. Entweder kann der Mapper selbst als Plugin implementiert werden, oder aber er dient als ein Persistenzdienst des Systemkerns, den andere Plugins nutzen können. Der höhere Grad an Dynamik bei einer Plugin-Architektur erfordert darüber hinaus Möglichkeiten, die bei gegenwärtigen Mappern nicht vorgesehen sind. Zwei Beispiele für solche Vorgänge zeigt Tabelle 3.1:

Plugin-Vorgang	Äquivalenter Vorgang beim ORM
Installieren eines neuen Plugins zur Laufzeit	Neue persistente Klassen ohne Neustart der Gesamtanwendung
Ersetzen eines Plugins durch eine neuere Version zur Laufzeit	Schemaänderung persistenter Klassen ohne Neustart der Gesamtanwendung

Tabelle 3.1: OR-Mapper innerhalb einer Plugin-Umgebung

Bei einer Plugin-Architektur ist zum Zeitpunkt ihrer Entwicklung unklar, welche Plugins in welcher Konstellation später ablaufen und welche konkreten Anforderungen (bspw. bezüglich der Anfrageoptimierung und Ladestrategie) sich aus einem solchen Kontext für den OR-Mapper ergeben. Dies bedeutet, dass die Konfiguration des OR-Mappers soweit wie möglich zur Laufzeit anpassbar sein sollte und damit dem Kontext des Plugin-Systems angepasst werden kann.

Die Problematik soll an einem Beispiel erläutert werden. Dazu nehmen wir an, dass für betriebswirtschaftliche Anwendungen ein Plugin-Rahmenwerk existiert, welches einige grundlegende Klassen der Anwendungsdomäne als Plugins zur Verfügung stellt, darunter eine Klasse Angestellter. Des weiteren wird angenommen, dass auf Basis des Rahmenwerks Plugins für die Personalverwaltung sowie für Kostenrechnung bestehen.
Wenn nun die Klasse Angestellter eine große Menge an Nutzdaten enthält, belastet das Laden der Daten aller Mitarbeiter den Hauptspeicher des Rechners, auf dem die Anwendung läuft, ganz erheblich. Dabei kann es jedoch der Fall sein, dass nur bestimmte Teile der Nutzdaten für den jeweils aktuellen Anwendungskontext benötigt werden. Beispielsweise benutzt das Plugin zur Personalverwaltung meistens nur die Gesamtdaten eines einzelnen Mitarbeiters. Das Plugin zur Kostenrechnung hingegen braucht lediglich die Gehaltsdaten aller Mitarbeiter.
Jeder dieser beiden Anwendungskontexten benötigt unterschiedliche Nutzdaten. Der Kontext, der die Summe der Gehaltsdaten benötigt, stellt dabei den Entwickler vor das Problem, wie er effizient an diese Nutzdaten gelangen kann. Eine

Instanziierung aller Angestellter-Objekte würde große Mengen nicht benötigter Daten von der Datenbank in den Hauptspeicher bringen und diesen unnötig belasten. Eine Umgehung des Mappers mittels direkter Abfrage der Datenbank wäre ein Rückfall zum Prinzip des manuell erstellten Persistenzcodes, mit allen damit verbundenen Nachteilen (siehe Abschnitt 2.2.1).

Es ist ersichtlich, dass das Laden der Objekte vom Entwickler dem jeweiligen Kontext der Anwendung angemessen steuerbar sein sollte. Da der Entwickler diesen Kontext kennt, kann er den Ladevorgang von Objekten optimal konfigurieren. Sinnvoll wäre an dieser Stelle also, wenn die Anwendung zur Laufzeit eine dynamische Ladestrategie erlauben würde. Gegenwärtig verfügbare OR-Mapper können zwar einfache Ladestrategien wie lazy-load bei Attributen und teilweise auch bei Collections umsetzen, diese Ladestrategie wird jedoch nur initial für die gesamte Laufzeit der Anwendung festgelegt. Damit sind solche Strategien jedoch nicht flexibel genug, da zu unterschiedlichen Zeitpunkten unterschiedliches Ladeverhalten angemessen sein kann.

4 O/R-Mapper mit dynamischer Codeerzeugung

Im letzten Abschnitt wurde deutlich, dass der Einsatz eines OR-Mappers mit statischer Konfigurierbarkeit innerhalb einer Plugin-Architektur nicht flexibel genug ist. Ein Lösungsansatz für dieses Problem ist dynamische Codeerzeugung zur Laufzeit. Dabei wird der Code einer oder mehrerer Komponenten des OR-Mappers zur Laufzeit der Anwendung erzeugt, abhängig vom Einsatzkontext und vom Zustand des Gesamtsystems. Durch diese Vorgehensweise bekommt der OR-Mapper einen Grad an Dynamik, der einer Plugin-Architektur angemessen ist.
Dynamische Codeerzeugung ist ein mächtiges Werkzeug, dass neben dem Vorteil der Flexibilität auch Nachteile hat und aus diesem Grund mit Bedacht eingesetzt werden muss. Ziel ist es, dem OR-Mapper einerseits die notwendige Dynamik und Flexibilität zu verschaffen. Andererseits sollte der Anteil des zur Laufzeit entstehenden Codes so gering wie möglich gehalten werden, um den OR-Mapper nicht intransparent werden zu lassen.

4.1 Komponenten eines O/R Mappers und deren Aufgaben

Wie generell bei Softwaresystemen, so ist es auch bei OR-Mappern sinnvoll, diese als modulare Architektur zu entwerfen. Die Implementierung einzelner Module kann dann leichter gegen eine andere ausgetauscht werden und die Kopplung zwischen den Modulen wird lose gehalten. Es kann zunächst top-down ein Rohentwurf des OR-Mappers vorgenommen werden, um anschließend Teilaufgaben des Mappers den einzelnen Komponenten zuzuordnen. Umgekehrt können Aufgaben auch gruppiert werden und die Komponenten an diesen Aufgabengruppen ausgerichtet implementiert werden.
Abbildung 4.1 bietet einen Überblick über den Aufbau eines Mappers. Die Architektur vieler OR-Mapper ähnelt dieser Abbildung. Im Folgenden werden die einzelnen Komponenten und deren Aufgaben erläutert.

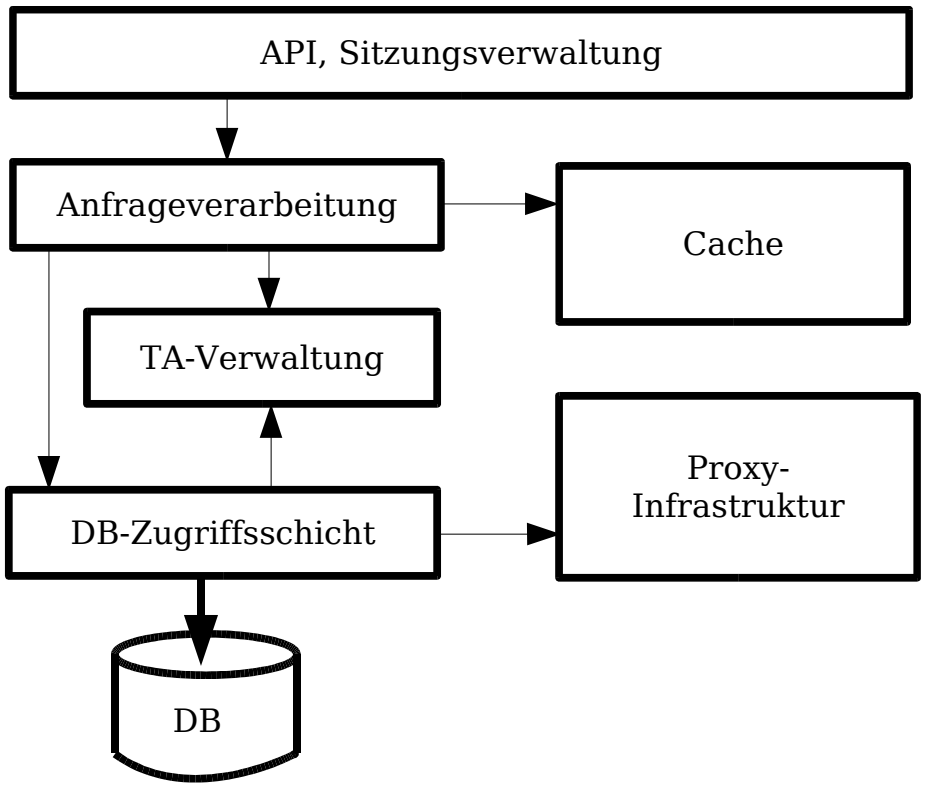


Abbildung 4.1: Komponenten eines OR-Mappers

4.1.1 API-Fassade und Sitzungsverwaltung

Zur Benutzung des OR-Mappers durch den Anwendungsentwickler muss ein API vorhanden sein. Dieses API kann einer Sitzungsklasse (im folgenden als Session bezeichnet) zugeordnet werden. Eine solche Sitzungsklasse zu instanziieren sollte nicht teuer bezüglich Zeit und Ressourcen sein. Auf einem Sitzungsobjekt können vom Entwickler Anfragen formuliert, neue Instanzen persistenter Klassen erzeugt und persistente Objekte aus der Datenbank entfernt werden.
Die Sitzungsklasse stellt dabei lediglich eine Fassade für die API dar (vgl. [GHJV95] S. 185ff und [MAR02] S. 5ff). Die durch die Aufrufe des API ausgelösten Aufgaben werden weitgehend von der Sitzungsklasse an andere Komponenten delegiert. Die Sitzungsklasse ist als zentrale Komponente des Mappers besonders gefährdet, dass ihr Aufgaben zugeteilt werden, die keiner anderen Komponente gut zugeordnet werden können. Die Umsetzung als Fassade hilft, die Sitzungskomponente übersichtlich zu halten.

4.1.2 Anfrageverarbeitung

Die an ein Sitzungsobjekt gestellten Anfragen werden von diesem an die Anfrageverarbeitung weitergereicht. Diese Komponente muss zunächst die syntaktische Korrektheit der Anfrage überprüfen und die Anfrage gegebenenfalls zurückweisen. Eine syntaktisch korrekte Abfrage wird dann in eine abstrakte Form transformiert, aus der nach der Optimierung der endgültige Ausführungsplan entsteht. Als Grundlage der Definition dieser abstrakten Form bietet sich ein Kalkül an, welches die Operatoren der relationalen Algebra enthält (Selektion, Projektion etc., siehe bspw. [SIL06] S. 46ff). Zusätzlich sind jedoch einige objektspezifische Operatoren wie die Dereferenzierung nötig, da die Abfragesprache in ihrer Syntax objektorientiert sein sollte. Auf Basis eines solchen Kalküls kann aus der Anfrage dann ein objektrelationaler Operatorbaum erstellt werden, bei dem Blätter Tupelmengen und inneren Knoten Operatoren entsprechen. Die Baumwurzel entspricht der gewünschten Ergebnismenge der Anfrage.
Auf dem objektrelationalen Operatorbaum können Optimierungen durchgeführt werden (äquivalent zu den Operationen auf einem relationalen Operatorbaum [VOS00] S. 313ff). Der resultierende Operatorbaum wird als Ausführungsplan bezeichnet. Schon an dieser Stelle kann der Cache des OR-Mappers benutzt werden (siehe Abschnitt 4.1.6), wenn Teilbäume des Ausführungsplans aus diesem bedient werden können. Vor allem bei Punkt-Abfragen nach einzelnen Objekten, deren Identität bekannt ist, ist dies sinnvoll. Sind diese Instanzen im Cache vorhanden, kann die entsprechende Unterabfrage an die Datenbank entfallen.

4.1.3 DB-Zugriffsschicht

Die Kapselung aller Datenbankzugriffe in einem eigenen Modul soll einem OR-Mapper ermöglichen, mit unterschiedlichen RDBMS und somit auch mit unterschiedlichen SQL-Dialekten zusammenzuarbeiten. Die direkte Interaktion mit dem RDBMS sollte ausschließlich auf diesen Teil des OR-Mappers beschränkt sein.
Im laufenden Betrieb des Mappers nimmt die DB-Zugriffsschicht Anfragen in der oben erläuterten Form (abstrakte objektrelationale Ausführungspläne) entgegen, diese werden dann in den SQL-Dialekt des benutzten RDBMS transformiert und das SQL-Kommando an die Datenbank gesandt.

OR-Mapper unterstützen zumeist Anfragen sowohl nach Objekten als auch nach Tupeln als Ergebnis. Von einem RDBMS wird ein Abfrageergebnis immer in Form

von Tupeln geliefert, dieses muss also noch (je nach Art der Anfrage) auf Objekte abgebildet werden. Dabei werden für die abgefragte Klasse Instanzen erzeugt, welche mit den einzelnen Datenelementen der Tupel initialisiert werden. Diese Abbildung geschieht sinnvollerweise ebenfalls in der DB-Kapselungsschicht, da Abbildungen von Datentypen zwischen Wirtssprache und Datenbank abhängig von der Implementierung der Datentypen durch das RDBMS sein können.

4.1.4 Transaktionsverwaltung

Die Transaktionsverwaltung eines Mappers muss für den Entwickler ACID-Konformität für die von ihm benutzten persistenten Objekte gewährleisten. Dazu muss eine Synchronisationsstrategie umgesetzt werden. Solche Strategien können in optimistische und pessimistische Verfahren unterschieden werden (zu den einzelnen Synchronisationsstrategien siehe [SIL06] S. 635ff). Je nach Strategie muss die Transaktionsverwaltung dabei die in Tabelle 4.1 aufgeführten Aufgaben umsetzen:

Optimistische Strategie	Pessimistische Strategie
Versionierung der Datensätze	Verwaltung von Sperren
Aufdeckung von Zugriffskonflikten	Aufdeckung von Deadlocks
Lösung von Zugriffskonflikten	Behebung von Deadlocks

Tabelle 4.1: Strategien zur Mehrbenutzersynchronisation

Die Umsetzung einer optimistischen Synchronisationsstragie ist deutlich weniger aufwändig als die einer pessimistischen, sie hat dafür den Nachteil einer höheren Konfliktrate bei stark parallelen Zugriffen. Des weiteren werden hierbei Konflikte erst beim Zurückschreiben der Änderungen in die Datenbank aufgedeckt, was für den Benutzer einer Anwendung ärgerlich ist, da er die Änderungen bereits vorgenommen hat und diese nun hinfällig sind.
Algorithmen für beide Sperrstrategien sind im Bereich der RDBMS entwickelt worden. Diese sind im Prinzip weitgehend übertragbar auf objektorientierte Datenstrukturen (siehe [AMB03] S. 299ff).
Für die Umsetzung einer Transaktionsverwaltung gibt es prinzipiell zwei Möglichkeiten: entweder werden die Transaktionen des RDBMS gekapselt oder es wird ein eigenes Transaktionssystem implementiert. Letztere Vorgehensweise kann auch Transaktionsmodelle unterstützen, welche die Transaktionsverwaltung des RDBMS nicht bietet. Der Aufwand zur Umsetzung hierfür ist allerdings hoch.

4.1.5 Proxyinfrastruktur

Eine der häufigsten Aktivitäten eines OR-Mappers ist das Auslesen von Werten persistenter Objekte sowie die Überprüfung, ob sich die Werte des Objekts geändert haben. Zur Umsetzung bieten sich zwei Techniken an: entweder das Auslesen der Werte mittels Reflection oder der Einsatz von Proxyobjekten (vgl. [GHJV95] S. 207ff). Da der massive Einsatz von Reflection die Performanz der Anwendung senken würde, wird im weiteren Verlauf dieser Arbeit die Verwendung von Proxyobjekten zur Kapselung von Domänenobjekten angenommen.
Unter dem Begriff der Proxyinfrastruktur werden im weiteren Verlauf sämtliche Teile des OR-Mappers verstanden, die sich mit der Erzeugung, Verwaltung und Entsorgung von Proxies beschäftigen. Die Effizienz der Implementierung dieser Proxyinfrastruktur ist eine Grundvoraussetzung für die Gesamteffizienz des OR-Mappers, da die Proxyobjekte sehr häufig benutzt werden (Zugriffe auf die Domänenobjekte) und ihre Beanspruchung des Hauptspeichers erheblich sein kann (wenn viele persistente Domänenobjekte instanziiert werden). Es ergeben sich daher neben der wichtigen Anforderung nach möglichst weitgehender Transparenz der Proxyinfrastruktur für den Entwickler der Domänenanwendung noch technische Anforderungen wie die nach geringem Speicherverbrauch der einzelnen Proxyobjekte. Eine nähere Erläuterung von Anforderungen an Proxies, Techniken zu deren Implementierung sowie die Vor- und Nachteile des Einsatzes von Proxies erfolgt in Kapitel 5 dieser Arbeit.

4.1.6 Cache

OR-Mapper verwenden üblicherweise einen zweistufigen Cache: jede Session hat einen eigenen Cache, daneben gibt es einen globalen Cache für alle Sessions. Ein einstufiger globaler Cache würde bei sehr unterschiedlichen Zugriffsmustern verschiedener Sessions zu schnell gefüllt werden und durch seinen hohen Datendurchsatz nutzlos werden. Ein reiner Sessioncache hingegen wäre nutzlos bei kurzlebigen Sessions, da der Cache seine Effizienz erst nach einer bestimmten Anlaufzeit erhält.
Der Cache eines OR-Mappers kann erheblich zu dessen Effizienz beitragen, da Zugriffe auf die Datenbank teuer sind und durch ihn eingespart werden können. Voraussetzung hierfür ist Kontextwissen über die zu erwartenden Datenzugriffe und eine angemessene Cache-Strategie. Hierunter wird die Systematik verstanden, nach der Objekte im Cache abgelegt und später wieder aus diesem entfernt werden. Solches Kontextwissen kann der OR-Mapper durch die Analyse der gestellten

Anfragen oder durch Konfiguration gewinnen. Häufig wiederkehrende Anfragen nach einem Objekt deuten darauf hin, dass die Anwendung keine Referenz auf das Objekt behält und dieses in den Cache gelegt werden sollte. Sinnvoller scheint jedoch die Möglichkeit, dem OR-Mapper per Konfiguration eine Cache-Strategie für bestimmte Klassen mitzuteilen, da der Anwendungsentwickler am meisten über die Umsetzung einzelner Anwendungsszenarien weiß und das Caching bei Bedarf entsprechend konfigurieren kann.
In Abhängigkeit von der Ausdrucksmächtigkeit der Anfragesprache eines OR-Mappers kann zur Umsetzung von Cachestrategien die Methode des 'semantic caching' interessant sein. Beim 'semantic caching' werden die Anfrageergebnisse gemeinsam mit dem Anfrageprädikat im Cache gespeichert. Der Cache wird dabei multidimensional innerhalb der Wertebereiche der Prädikat-Attribute partitioniert. Damit kann für folgende Anfrageprädikate bestimmt werden, ob die Anfrage komplett oder teilweise aus dem Cache bedient werden kann. Mehr zum 'semantic caching' findet sich bei [REN98] sowie [HÖP05] S. 181ff.

4.2 Potentiale für dynamische Codeerzeugung

Dynamische Codeerzeugung soll vor allem einen Zweck erfüllen: die Änderung bestimmter Konfigurationen des OR-Mappers zur Laufzeit ändern und damit auf geänderte Anforderungen des aktuellen Anwendungsszenarios reagieren. Aus diesem Grund bestehen Potentiale für dynamische Codeerzeugung besonders bei denjenigen Komponenten, welche in ihrem Verhalten und ihrer Performanz besonders konfigurationsabhängig sind. In Bezug auf den Einsatz eines OR-Mappers innerhalb einer Plugin-Architektur kommen noch die Potentiale bei Komponenten hinzu, deren Konfiguration erst zur Laufzeit des Systems vorgenommen werden kann, zum Beispiel durch das Hinzukommen neuer Plugins.
Andererseits muss berücksichtigt werden, dass eine Komponente mit hoher Komplexität durch dynamische Codeerzeugung fehleranfällig und intransparent wird. Codeerzeugung sollte also vor allem dort vorgenommen werden, wo mit wenig zu erzeugendem Code große Wirkung auf das Gesamtsystem erzielt werden kann.

Die größten Performanzkosten beim OR-Mapping entstehen durch Zugriffe auf die Datenbank. Die größten Einsparpotenziale sind also bei denjenigen Komponenten zu finden, die durch eine geeignete Umsetzung das Potential haben, solche Zugriffe unnötig zu machen. Bei der Betrachtung der angeführten Komponenten sind insbesondere der Cache und die Proxyinfrastruktur als geeignete Kandidaten für Codeerzeugung auszumachen.

Beim **Cache** kann vorhandenes Kontextwissen genutzt werden, um gezielt Domänenobjekte aus Abfrageergebnissen im Cache abzulegen, wenn klar ist, dass bald wieder darauf zugegriffen wird. Umgekehrt kann bei einmaligen Abfragen gezielt ein Caching verhindert werden, um den Cachespeicher für langlebigere Objekte freizuhalten. Andererseits könnte der Cache in Kopplung mit der Sitzungskomponente überprüfen, bei welcher Art von Abfragen er besonders gute Trefferraten erzielt. Daraus könnte der Cache eine eigene Cache-Strategie ermitteln, die je nach Anwendungskontext sehr unterschiedlich sein kann.

Bei der **Proxyinfrastruktur** kann sich Codeerzeugung deshalb lohnen, weil hier die von einer Plugin-Architektur erforderte Flexibilität umgesetzt werden kann. Wenn zur Laufzeit der Anwendung neue Plugins zur Anwendung hinzukommen und diese den OR-Mapper zur Persistierung ihrer Klassen benutzen wollen, müssen die Proxyklassen für diese Domänenklassen zur Laufzeit erzeugt und benutzt werden. Für das dynamische Erzeugen der Proxyklassen spricht außerdem, dass hier individuelle Ladestrategien umgesetzt werden können, so dass ein zusätzliches Nachladen von Nutzdaten verhindert wird.

Andererseits erscheint es übertrieben, die gesamte Infrastruktur zur Proxyverwaltung dynamisch zu erzeugen, da diese recht komplex werden kann und der enstehende Code auch nicht so homogen ist wie bei der Erzeugung des Codes für die einzelnen Proxyklassen.

Die weiteren Komponenten könnten zwar mit dynamischer Codeerzeugung an Flexibilität hinzugewinnen, jedoch stünde das Potential in keinem Verhältnis zum Aufwand und den Nachteilen.

Die **DB-Kapselungsschicht** setzt hauptsächlich eine abstrakte Anfrage in eine SQL-Abfrage um und nimmt das Mapping zwischen Ergebnistupeln und Objekten vor. Für diese Aufgaben benötigt sie keine Codeerzeugung. Es ist hier auch nicht ersichtlich, inwiefern diese Komponente durch eine Plugin-Architektur neuen Anforderungen ausgesetzt wird.

Für die **Anfrageverarbeitung** gilt ähnliches. Zwar müssen beide Komponenten beim Hinzukommen neuer persistenter Klassen ihr Verhalten ändern, die DB-Zugriffsschicht muss neue Tabellen der Datenbank einbeziehen und die

Anfrageverarbeitung muss ihre Syntaxprüfung anpassen. Dies kann aber gelöst werden, indem beide Komponenten bei einem zentralen Metadaten-Repository anfragen, ob die in der Anfrage enthaltenen Klassen persistent sind oder nicht. Insofern wäre eine dynamische Erzeugung an dieser Stelle übertrieben.

Die **Transaktionsverwaltung** bietet diesbezüglich ebenfalls wenig Potential. Neue Anforderungen entstünden lediglich dann, wenn ein neues Plugin innerhalb einer laufenden Transaktion hinzukäme, was zu erlauben nur sinnvoll wäre, wenn viele lange Transaktionen in der Gesamtanwendung benutzt würden. In diesem Fall müssten allerdings die Transaktionen des OR-Mappers von denen des RDBMS getrennt werden, was die Umsetzung einer eigenen Transaktionsverwaltung erfordern und den Rahmen dieser Arbeit sprengen würde.

Die **Sitzungsverwaltung** ist die Komponente, deren Funktionalität am schwierigsten gegenüber den anderen Komponenten abgegrenzt werden kann, da sie eine zentrale Rolle einnimmt hinsichtlich der Verknüpfung der Komponenten untereinander. Es ist empfehlenswert, sie deshalb so einfach wie möglich zu halten und ihr hauptsächlich Delegationsaufgaben zuzuweisen. Für diese Aufgaben ist jedoch nicht erkennbar, an welcher Stelle der Einsatz dynamischer Codeerzeugung lohnenswert ist.

4.3 Dynamische Codeerzeugung zur Laufzeit

Das Prinzip der Codeerzeugung wurde bereits in Abschnitt 2.2.1 sowie 2.2.4 erläutert. Bei den erläuterten Vorgehensweisen wird der Code allerdings vor dem Kompilierprozess erzeugt und dann mit anderen Modulen zur Gesamtanwendung kompiliert. Die Basis für solchen generierten Code sind Generierungsvorschriften (in Form von Templates), die aus Flexibilitätsgründen und zwecks Wiederverwendbarkeit parametrisierbar sind. Für eine Plugin-Anwendung ist die Generierung von Code vor der oder zur Kompilierzeit jedoch ungeeignet. Zum einen müsste bei einer Änderung der Generierungsvorschrift die Anwendung neu kompiliert und das Deployment erneut vorgenommen werden, zum anderen ist damit zur Laufzeit keine Dynamik möglich.

Eine Lösungsmöglichkeit ist die Codeerzeugung zur Laufzeit der Rahmenanwendung durch ebendiese. Dieser Prozess läuft folgendermaßen ab:

1. die Rahmenanwendung startet
2. die Generierungsvorschriften werden dynamisch erstellt und parametrisiert
3. der Code wird erzeugt und im Speicher kompiliert und steht anschließend zur Benutzung durch die Rahmenanwendung zur Verfügung

4. falls der erzeugte Code auf einem nichtflüchtigen Speicher abgelegt wurde und sich die Generierungsvorschriften nicht geändert haben, steht er beim nächsten Start der Anwendung sofort kompiliert zur Verfügung

Da die Klassen zur Kompilierzeit der Rahmenanwendung noch nicht vorhanden sein müssen, ist eine Benutzung der erzeugten Klassen in der Anwendung nur über Interfaces möglich.

4.3.1 Methoden dynamischer Codeerzeugung zur Laufzeit

Bei älteren Hochsprachen wie C oder C++ gibt es nur eine Möglichkeit für die Erzeugung von Code zur Laufzeit: da in diesen Programmiersprachen direkt in den Speicher geschrieben werden kann, kann zur Laufzeit ausführbarer Code in Form von Prozessorbefehlen erzeugt und über miterzeugte Einsprungadressen zur Ausführung gebracht werden. Da diese Technik extrem fehleranfällig und ebenso anfällig für Sicherheitslücken ist, wird sie in modernen Hochsprachen nicht mehr als Möglichkeit angeboten. Zudem ist der Wartungsaufwand sowie der Aufwand für Fehlerbehebung als sehr hoch einzuschätzen.

Code-DOM und Emit bei .NET

Codeerzeugung zur Laufzeit wird in modernen Programmiersprachen vor allem dadurch erleichtert, dass diese (im Gegensatz bspw. zu C) mit einem zweistufigen Kompilierprozess arbeiten. Dabei wird zunächst der Quelltext in ein Zwischenformat übersetzt (Java Bytecode oder bei DOT.NET die Intermediary Language). Zur Laufzeit des Programms wird dann dieses Zwischenformat gelesen und von einem Just-In-Time-Compiler (JIT) in die Maschinensprache des Prozessors übersetzt und dann ausgeführt.
Für den JIT-Compiler ist es prinzipiell gleichwertig, ob das Zwischenformat auf einem Festplattenspeicher vorliegt oder im Hauptspeicher erzeugt wird. Dieser Umstand wurde beim .NET-Framework genutzt und es enthält zwei Module, die es erlauben, zur Laufzeit Code zu erzeugen, diesen anschließend zu kompilieren und auszuführen: **`System.Reflection.CodeDom`** und **`System.Reflection.Emit`**.
Die Benutzung der beiden Module ist ähnlich: zunächst werden die zu erzeugenden Klassen mit ihren Membern definiert. Hierzu existieren im Namespace `System.Reflection` eine große Zahl an Hilfsklassen zur Definition von Code-Fragmenten. Die Definition umfasst den Klassennamen, Vererbungen und

deklarierte Interfaceimplementierungen sowie sämtliche Methoden-, Property-, Event- und Indexersignaturen. Diese Rümpfe müssen anschließend noch mit Code gefüllt werden, welcher die eigentliche Funktionalität umsetzt. Beim Modul CodeDOM wird dazu ein Provider für die gewünschte Programmiersprache benutzt (bspw. C# oder Visual Basic) und ein abstrakter Syntaxbaum erstellt. Dieser Syntaxbaum wird dann von einem In-Memory-Compiler in IL-Code übersetzt. Da dieser direkt auf dem Syntaxbaum arbeitet, kann er die syntaktische Prüfung überspringen. Es wird jedoch auch die Möglichkeit geboten, sogenannte Codesnippets direkt übersetzen zu lassen. Dieser Vorgang dauert entsprechend länger, ist jedoch deutlich weniger umständlich.
Beim Emit-Modul wird vom Entwickler direkt IL-Code für die Methodendefinitionen geschrieben, allerdings ist dies weniger komfortabel und sehr fehleranfällig, da IL-Code vom Abstraktionsniveau sehr nahe an Maschinensprache liegt (vgl. [BOC02]) und entsprechend klassische Konstrukte von Hochsprachen (wie Schleifen) nicht vorhanden sind. Andererseits ist die Benutzung von Emit am performantesten und für Methoden mit übersichtlicher Funktionalität benutzbar.
Eine dritte Möglichkeit der Codeerzeugung zur Laufzeit ist die Benutzung des Kommandozeilen-Kompilers. Dabei wird der Quellcode der Proxyklassen als String generiert, mithilfe des Kommandozeilen-Kompilers kompiliert und die erzeugte Assembly geladen. Allerdings ist dieses Verfahren wenig performant, da der Compiler Festplattenzugriffe benötigt und eine vollständige syntaktische Prüfung des Codes vornehmen muss.

4.3.2 Vor- und Nachteile laufzeit-dynamischer Codeerzeugung

Für den Einsatz dynamischer Codeerzeugung gilt die gleiche Bedingung wie für automatische Codegenerierung vor der Kompilierung: Erzeugung lohnt sich vor allem dann, wenn größere Mengen von homogenem und wenig komplexem Code erstellt werden müssen.
Dynamische Codeerzeugung bietet große Flexibilität als Vorteil, sollte jedoch aufgrund ihrer Nachteile nur in angemessenen Kontexten eingesetzt werden.
Nachteilig ist das erschwerte Debugging von dynamisch erzeugtem Code, dass, wenn überhaupt, nur auf dem Abstraktionsniveau von Maschinensprache oder der Zwischensprache (bei DOT.NET die Intermediate Language) möglich ist. Wenn große Mengen von komplexem Code erzeugt werden droht zudem eine hohe Intransparenz der Anwendung beim Ablauf, da der Entwickler den erzeugten Code sehr genau kennen muss, um die Verhaltensweisen des Codes nachvollziehen zu können.

Aus diesem Grund sollte dynamische Codeerzeugung nur für Klassen eingesetzt werden, die übersichtlich gehalten werden können, bei denen sich jedoch die Flexibilität der Codeerzeugung in entsprechender Performanz auszahlt. Bei den behandelten Komponenten eines OR-Mappers trifft dies vor allem auf die Proxyklassen zu, die die einzelnen Domänenklassen umhüllen.

5 Proxyinfrastruktur bei OR-Mappern

In diesem Kapitel sollen zunächst die Aufgaben ermittelt werden, welche die Proxies persistenter Objekte innerhalb eines OR-Mappers erfüllen sollen. Anschließend werden verschiedene Methoden zur Umsetzung des Proxy-Entwurfsmusters (vgl. [GHJV95] S. 207ff) auf ihre Vor- und Nachteile diesbezüglich untersucht. Schließlich wird eine Proxyinfrastruktur mit einzelnen Komponenten entworfen, welche die laufzeit-dynamische Erzeugung von Proxies umsetzt.

5.1 Aufgaben von Proxyobjekten

Die Aufgabenbereiche der Proxyobjekte lassen sich unterteilen in generelle Aufgaben, die in jedem Anwendungskontext von Proxies anfallen sowie OR-Mapper-spezifische Aufgaben.

5.1.1 Allgemeine Aufgaben von Proxies

Die grundsätzliche Aufgabe von Proxyobjekten ist es, Zugriffe und Aufrufe von Membern an das verdeckte Objekt weiterzuleiten. Dieses setzt die eigentliche Domänenfunktionalität um. Dies gilt für sämtliche Member des verdeckten Objekts, also Methoden, Properties, Indexer und Events (zu den einzelnen Membertypen siehe Anhang A). Das Proxyobjekt sollte sich dem verdeckten Objekt so ähnlich wie möglich verhalten. Dies gilt sowohl für das Verhalten zur Laufzeit der Anwendung als auch für das Verhalten zur Entwurfszeit für den Entwickler. Ziel des Proxydesigns ist, dass der Entwickler kein Hintergrundwissen über die internen Vorgänge des Proxys haben muss und sich die Proxies so transparent wie möglich verhalten. Prinzipbedingt kann jedoch diese Transparenz nicht vollkommen sein.

Neben spezifischer Proxylogik wie die für einen OR-Mapper gibt es allgemeine Funktionen, die in einem Proxy anzusiedeln hilfreich ist. Solche allgemeine Proxylogik kann vor und nach der Weiterleitung des Aufrufs an das verdeckte Objekt umgesetzt werden. Da Proxyobjekte aus verschiedenen Gründen benutzt werden können, ist es sinnvoll, die Proxylogik innerhalb der Proxyklassen erweiterbar zu halten. Damit kann verhindert werden, dass mehrere Schichten von Proxyklassen sich umhüllen. Da die einzelnen Proxyklassen in aller Regel nichts voneinander wissen, kann ein solches System von Proxyschichten zu

intransparentem Verhalten führen und die Fehlersuche erschweren.
Möglich ist eine solche Erweiterbarkeit der Proxylogik durch Events und Delegates (siehe [DEL06]). Dabei können für jeden Member wie Properties und Methodenaufrufe Prä- sowie Post-Delegates zugewiesen werden. Die Reihenfolge der Abarbeitung kann dabei vom Entwickler durch die Reihenfolge der Zuweisung der Delegates festgelegt werden. Somit ist die Proxyklasse hinreichend universell und weitere Proxyklassen sowie zusätzliche Umhüllungen sind nicht notwendig.
Dieser Ansatz ähnelt stark dem "Codeweaving" bei der aspektorientierten Programmierung (AOP, vgl. [AOS06]). Der Unterschied besteht darin, dass bei echter AOP deklarativ vorgegangen wird und bei callbacks programmatisch-funktional.
Die beschriebene Technik kann auch dazu benutzt werden, um Beschränkungen im Wertebereich einfacher Datentypen von Properties vor dem Setzen zu garantieren. Ein Beispiel: Eine Klasse Adresse hat ein Property Postleitzahl, welches nur fünfstellige positive Ganzzahlen als Werte annehmen darf. Sollte ihr nun fälschlicherweise eine sechsstellige oder eine negative Zahl zugewiesen werden, kann eine Exception ausgelöst werden. Durch solche Überprüfungen wird die Möglichkeit unterstützt, Programmfehler möglichst frühzeitig zu finden, damit sich diese nicht in den Zustand des Systems einschleichen und erst an anderer Stelle auffallen. Der Fehlerbehebungsprozess wird dadurch deutlich kürzer, weil die verursachende Stelle im Quellcode schneller dem Fehlverhalten der Anwendung zugeordnet werden kann.

5.1.2 OR-Mapper-spezifische Aufgaben von Proxies

Zu den OR-Mapper-spezifischen Aufgaben, die ein Proxy erfüllen sollte, gehört die Verwaltung von individuellen Metadaten für jede Instanz eines persistenten Objekts. Diese Metadaten werden von verschiedenen Komponenten des OR-Mappers zur Erfüllung ihrer Aufgaben benötigt. Zu diesen Metadaten gehört der Status des Objekts, anhand dessen der Mapper feststellen kann, welche SQL-Befehle bei Abschluss der laufenden Transaktion an die DB gesendet werden müssen. Tabelle 5.1 zeigt die Zustände, die ein Objekt im Laufe seines Lebenszyklus einnehmen kann:

Status	Zustand des Objekts
new	Objekt wurde neu erzeugt und ist noch nicht in der DB persistiert
persistent	Objekt ist in der DB, wurde geladen, aber nicht verändert
changed	Wie persistent, Objekt wurde jedoch geändert
removed	Objekt wurde entfernt, befindet sich aber noch in der DB

Tabelle 5.1: Zustände eines Persistenzproxys

Des weiteren muss das Proxyobjekt eine Versionsnummer oder einen Zeitstempel pflegen, um der Transaktionsverwaltung eine optimistische Synchronisationsstrategie zu erlauben. Wichtig ist dies auch für den Fall, dass externe Anwendungen ebenfalls auf die vom OR-Mapper benutzte Datenbank zugreifen. Durch eine Versionsnummer oder einen Zeitstempel kann der OR-Mapper erkennen, dass ein Datensatz von einer externen Anwendung geändert wurde und entweder versuchen, den daraus resultierenden Änderungskonflikt zu lösen oder eine Exception werfen. Falls von der Transaktionsverwaltung eine pessimistische Synchronisationsstrategie benutzt wird, muss der Proxy einen Sperrstatus enthalten und bei Schreibzugriffen entsprechend berücksichtigen. Bei der Implementierung des Prototypen wurde die Mehrbenutzersynchronisation nicht umgesetzt, es existieren aus dem Bereich der RDBMS eine Reihe an Verfahren, die hierzu benutzt werden können. Hierzu zählen optimistische Verfahren mit Versionierung, Zeitstempelverfahren sowie pessimistische Verfahren mit Sperren (siehe zum Beispiel [DAT04] S. 465ff).
Eine weitere Aufgabe von Proxyobjekten ist es, Ladestrategien zu unterstützen. Eine solche Strategie kann zum Beispiel daraus bestehen, nur bestimmte Nutzdaten des Objekts zu laden. Um dies zu unterstützen, muss ein Proxyobjekt den Ladestatus der einzelnen Nutzdaten kennen und bei Zugriffen entsprechend durch Nachladen reagieren, falls diese noch nicht geladen wurden. Konkrete Entwürfe für Ladestrategien werden in Kapitel 6 vorgestellt.

Bei der Zuteilung von Teilaufgaben eines OR-Mappers, die in den Proxyobjekten umgesetzt werden soll, muss berücksichtigt werden, dass unter Umständen sehr viele dieser Objekte von der Gesamtanwendung instanziiert werden. Entsprechend sollte der Umfang der Proxy-Instanzen klein gehalten werden.

5.2 Techniken zur Umsetzung von Proxies

Zur Umsetzung von Proxyobjekten gibt es verschiedene Ansätze mit jeweils unterschiedlichen Restriktionen für die Klassen, welche von den Proxies umhüllt

werden sollen. Im weiteren wird die Klasse der eigentliche Domänenobjekte als **O** sowie die Klasse des Proxyobjekts als **P** bezeichnet.

5.2.1 Ableitung der Klasse

Bei dieser Proxytechnik wird P von O abgeleitet. Bei der Implementierung der Domänenlogik werden vom Entwickler ausschließlich Instanzen von P benutzt. Diese müssen von einer Proxyfactory bereitgestellt werden, die ein zentraler Teil der Proxyinfrastruktur ist. Durch die Vererbungslinie können Instanzen von P durch einen Cast als O benutzt werden.
Bei der Implementierung von P müssen die Properties und Methoden überschrieben werden. Der Aufruf der Basismethode von O kann dabei zusätzlich mit Prä- und Postprozessierung umhüllt werden, diese Umhüllung setzt die Proxylogik um.
Tabelle 5.2 zeigt ein kurzes Code-Beispiel hierfür:

Domänenklasse	Proxyklasse

`public class MyClass: {`
`  public virtual doSomething() {`
`  // code goes here...`
`  }`
`}`
	`internal class MyClassProxy: MyClass {`
	`  internal doSomething() {`
	`  preProcess();`
	`  base.doSomething();`
	`  postProcess();`
	`  }`
	`}`

Tabelle 5.2: Proxying durch Ableitung von der Domänenklasse

Nachteilig bei Vererbung als Proxytechnik ist, dass P eine andere Klasse als O hat und dementsprechend der Operator `typeof()` bzw `is` in C# (entspricht einem `instanceof` in Java) nicht sinnvoll benutzt werden kann. Ein weiterer Nachteil ist, dass O seine Klassenhierarchie nicht abschließen darf, da P von O abgeleitet werden können muss. Außerdem müssen für die Umsetzung zusätzlicher Proxylogik Methoden und Properties überschrieben werden und deshalb in O als virtuell markiert sein, da ansonsten der Proxy nicht die Möglichkeit hat, bei einem Aufruf die eigenen Methoden zu benutzen und stattdessen die Methoden von O aufgerufen werden.
Vorteilhaft bei dieser Proxytechnik ist die weitgehende Erhaltung der Sichtbarkeitsbereiche von Membern. Lediglich `internal` ist nicht möglich, da sich P in aller Regel in einer anderen Assembly befindet als O. Um bestimmte Methoden von einem Zugriff von außen auszuschließen kann allerdings der Modifizierer `protected internal` verwendet werden. Mit diesem kann von der abgeleiteten

Proxyklasse aus ein Zugriff ebenso erfolgen wie von einer anderen Klasse aus der gleichen Assembly. Gegen einen Zugriff aus fremden Assemblys sind die Methoden jedoch geschützt.
Die Methode der Ableitung wird von den meisten Proxy-Implementierungen benutzt, beispielsweise der allgemein einsetzbare Dynamic Proxy (siehe [DYN06]) aus dem Castle-Projekt (siehe [CAS06]). Dieser Proxy kann auch in Verbindung mit NHibernate genutzt werden, um Teile der Reflection-Aufrufe einzusparen. Ein Beispiel für einen OR-Mapper mit Proxies, die Ableitung benutzen, ist DataObjects.NET (siehe [DAT06]).

5.2.2 Wrapper

Bei der Umsetzung eines Proxy als Wrapper hält P hält eine Referenz auf O und reicht nicht-private Member von O nach außen. Damit O bei diesem Ansatz wirklich gekapselt ist, muss sichergestellt werden, dass von außen keine Referenzen mehr auf O verweisen, die O direkt statt über P manipulieren könnten. Falls noch weitere Referenzen auf O verweisen und damit auf O zugreifen, wird die Proxylogik nicht mehr ausgeführt und daraus resultierende Fehler sind entsprechend schwierig aufzufinden. Dieser Referenzproblematik kann dadurch begegnet werden, indem bei der Proxyerzeugung statt O selbst eine Kopie von O als Kern für den Proxy benutzt wird. Ein Beispiel für einen solchen Wrapper ist in Tabelle 5.3 aufgeführt:

Domänenklasse	Proxyklasse

`public class MyClass: ICloneable {`
`  Object ICloneable.clone()`
`  { ... }`
`  public doSomething { ... }`
`}`
	`public class MyClassProxy: MyClass {`
	`  private MyClass wrappedObj;`
	
	`  public MyClassProxy(MyClass obj) {`
	`  this.wrappedObj = (MyClass)`
	`  (obj as ICloneable).clone();`
	`  }`
	
	`  public doSomething() {`
	`  preProcess();`
	`  this.wrappedObj.doSomething();`
	`  postProcess();`
	`  }`
	`}`

Tabelle 5.3: Proxying durch Umhüllung des Domänenobjekts

Nachteilig bei dieser Methode ist, dass P nicht dieselbe Klasse hat wie O, mit den

gleichen Restriktionen wie bei der Ableitung als Proxytechnik. Als zusätzlicher Nachteil erweist sich, dass P auch nicht ohne weiteres in eine Instanz von O gecastet werden kann, da die beiden in keiner gemeinsamen Vererbungslinie miteinander stehen. Zwar könnte dieses Manko teilweise kompensiert werden, indem man individuelle explizite oder implizite Casts von O nach P und umgekehrt implementiert, wie dies in C# möglich ist (siehe Anhang A und [HEJ04] S. 380f). Dadurch würde allerdings der Umgang mit Instanzen von O und P sehr unübersichtlich werden, insbesondere wenn solche Casts als implizit definiert sind, da der Entwickler sich immer bewusst sein muss, ob gerade eine Instanz von O oder P verwendet wird. Gerade dies jedoch ist eine Eigenschaft, die ein Proxy nicht haben sollte, da er möglichst transparent sein muss.
Als weiterer Nachteil lässt sich feststellen, dass Member der Sichtbarkeitsbereiche `protected` sowie `protected internal` Probleme aufwerfen, da diese nicht vom Proxy aus zugreifbar sind.
Der einzige Vorteil beim Wrapper liegt darin, dass er nicht in die Klassenhierarchie eingreift und keine entsprechenden Vorbedingungen bezüglich der Vererbungsmodifikatoren erfordert wie die in Abschnitt 5.2.1. erläuterten.

5.2.3 Interceptor

Seit Version 1.3 existiert unter Java noch eine dritte Möglichkeit, das Proxy-Entwurfsmuster umzusetzen: die Benutzung von `java.lang.reflect.Proxy` und die Implementierung des Interfaces `java.lang.reflect.InvocationHandler`. Eine Klasse, welche dieses Interface implementiert, kann Methodenaufrufe an den Proxy abfangen und dabei eigene Proxylogik umsetzen. Dazu muss ein Interface für das Objekt definiert werden, welches vom Invocationhandler verdeckt werden soll. Der Proxy kann dann gefahrlos in dieses Interface gecastet werden. Alle Methodenaufrufe an den Proxy durchlaufen die invoke-Methode des Invocationhandlers, diese Methode bekommt das Proxyobjekt, die Methode sowie die Methoden-Parameter als Parameter übergeben. Innerhalb der invoke-Methode kann die Proxylogik umgesetzt und der Methodenaufruf an das eigentliche Objekt weitergereicht werden.
Vorteil dieser Technik ist, dass eine InvocationHandler-Klasse als Proxy für mehrere Klassen dienen kann. Die bislang behandelten Proxymethoden hingegen benötigen für jede zu verdeckende Klasse eine eigene Proxyklasse. Nachteilig bei diesem Proxyansatz ist wie bei Abschnitt 5.3.2, dass ebenfalls keine externen Referenzen mehr auf das eigentliche Domänenobjekt existieren sollten, da ansonsten die

Proxylogik umgangen werden kann. In modernen Programmiersprachen wird die direkte Verwaltung von Referenzen in aller Regel dem Laufzeitsystem der Sprache überlassen, da es sich in der Vergangenheit erwiesen hat, dass insbesondere diese Teile des Codes in Anwendungen besonders fehleranfällig waren (zum Beispiel für Memory-leaks, mehrschrittige Zeiger etc.). Aus diesem Grund erweisen sich Proxy-Ansätze wie unter 5.2.2 und 5.2.3 als eher rückschrittlich bezüglich der Entwicklungstendenz von Programmiersprachen und -paradigmen.
Ebenfalls nachteilig ist, dass die Aufrufe der Methoden auf dem eigentlichen Objekt per `method.invoke()` mit später Bindung erfolgen und dementsprechend langsam sind (vgl. [SOS03]). Im speziellen Kontext dieser Arbeit spricht noch ein weiteres Argument gegen diesen Ansatz: DOT.NET verfügt über kein direktes Äquivalent zu `java.lang.reflect.Proxy`.
Unter DOT.NET könnte ein solcher Mechanismus mit der Benutzung von Delegates (siehe [DEL06]) zwar nachgebaut, die entsprechenden Proxyklassen müssten jedoch manuell generiert werden. Unter Java werden die Proxyklassen automatisch erzeugt. Da sich bei diesem Verfahren außer dem Mehrklassen-Proxying keine weiteren Vorteile ergeben und zusätzlich die Performanz der durch die Benutzung von später Bindung bei Methodenaufrufen gedrosselt würde, wird dieser Ansatz an dieser Stelle nicht weiter verfolgt.

5.3 Umsetzung der Proxies im Prototypen

Von den vorgestellten Proxytechniken bietet die Vererbung als Proxytechnik die geringsten Nachteile, da die Problematik des Referenzenmanagements entfällt und die Einschränkungen bei den Sichtbarkeitsbereichen nur gering sind. Es bleibt das grundsätzliche Problem bei der Benutzung von Proxies, dass die Proxyobjekte eine andere Klasse haben als die eigentlichen Objekte und wichtige Typoperatoren nicht korrekt benutzt werden können. Die Möglichkeit, dass P nach O gecastet werden kann, ist dabei keine Lösung.
Um Typoperatoren bei Proxies benutzbar zu machen wird im Prototypen für jede persistente Domänenklasse ein entsprechendes Domäneninterface vorausgesetzt. Unter einem Domäneninterface wird ein Interface verstanden, welches als Grundlage zur Entwicklung einer Domänenklasse dient und von dieser implementiert wird. Bei der Modellierung der Anwendungsdomäne werden zuerst Domäneninterfaces definiert und anschließend die implementierenden Domänenklassen entwickelt. Ziel dieser Vorgensweise ist, dass innerhalb der Anwendung Domänenobjekte ausschließlich über die Domäneninterfaces

angesprochen und benutzt werden.

Diese Vorgehensweise hat eine Reihe an Vorteilen:

- Das Interface bildet die Ausgangsbasis für die Implementierung der Domänenklasse. Hierdurch wird eine Trennung zwischen dem Modell und seiner technischen Umsetzung vorgenommen. Damit wird die Grundlage gelegt für eine spätere Portierung der Anwendung, entweder auf eine andere technologische Plattform oder in eine andere Programmiersprache. Andererseits wird auch die Trennung von Zuständigkeiten von Modellierung und Implementation zwischen den beteiligten Personen im Softwareentwicklungsprozess unterstützt.
- Wie auch sonst bei der Benutzung von Interfaces wird die Kopplung zwischen verschiedenen Teilen des Softwaresystems lose gehalten. Damit sind einzelne Implementierungen leichter auswechselbar. Durch das Prinzip "Inversion of Control" (vgl. [FOW04]) kann mit geringem Aufwand konfiguriert werden, welche Implementierung benutzt wird.
- Speziell unter dem Aspekt der transparenten Proxybenutzung sind Domäneninterfaces nahezu ideal. Da die Domänenobjekte ausschließlich über das Interface benutzt werden sollen, muss der Proxy entsprechend dieses Interface auch implementieren. Die Intention bei der Einführung von Interfaces, Schnittstellen und konkrete Implementation zu trennen, unterstützt also die Transparenz bei der Nutzung von Proxy-Objekten sehr gut.

Die vorgeschlagene Benutzung von Domäneninterfaces kann auch gut in gängige Prozessmodelle der Softwareentwicklung integriert werden, vor allem in Entwicklungsprozessmodelle, die sich an der Model-Driven-Architecture orientieren, wie zum Beispiel MDSD (Model Driven Software Development). Bei der MDA wird versucht, Teile des Codes der Anwendung ausgehend von einem abstrakten Modell der Anwendungsdomäne durch Generierungsvorschriften und einen Code-Generator erzeugen zu lassen. Der erzeugte Code wird mit manuell erstelltem Code zusammengebracht und kann danach kompiliert werden. Näheres zur MDA und MDSD findet sich bei [STA05].
Ein Nachteil bei der Nutzung von Domäneninterfaces muss allerdings ebenfalls erwähnt werden: die Modellierungsmächtigkeit bei Interfaces weist gegenüber derjenigen von Klassen noch immer Lücken auf. Dies gilt vor allem bei Zugriffsmodifikatoren. Unter C# beispielsweise kann ein Interface zwar als Ganzes entweder public, protected, internal oder private sein. Modifikatoren für einzelne Member des Interfaces sind allerdings nicht erlaubt (vgl. [HEJ04] S. 418), bspw. um

bei einem public-Interface den Sichtbarkeitsbereich einzelner Member einzuschränken. Die gleiche Problematik besteht unter Java, zudem sind Interfaces dort generell public und können nur Methoden als Member enthalten (vgl. [FLA02] S. 117ff), da das Konzept von Propertys (siehe Anhang A) hier fehlt.
An dieser Stelle zeigt sich, dass Interfaces bisher primär als Schnittstelle zwischen einzelnen Code-Einheiten betrachtet und benutzt werden. Ihre zweite Facette, die Trennung von Typdefinition und Typimplementierung, wird von gegenwärtigen Umsetzungen von Interfaces in modernen Programmiersprachen noch nicht genügend unterstützt.
Dieses Problem kann kompensiert werden, indem jeweils eigene Interfaces sowohl für den internen als auch für den externen Gebrauch mit unterschiedlichen Sichtbarkeitsbereichen entwickelt werden. Nachteilig ist dabei, dass die gemeinsamen Anteile beider Interfaces konsistent gehalten werden müssen. Der Aufwand hierfür hält sich bei Generierung der Interfaces aus Metadaten (wie bei [STA05] vorgeschlagen) allerdings im Rahmen.

Persistente Interfaces

Für den hier vorgeschlagenen Ansatz der Benutzung von Domäneninterfaces für die Domänenklassen spricht bei einer Umsetzung unter DOT.NET noch ein weiterer Grund: Pugh unterscheidet Interfaces in daten- bzw. serviceorientiert (siehe [PUG06] S. 32f), in Anlehnung an Ivar Jacobsens (vgl. [JAC92]). Wenn man nun auch innerhalb einer Klasse allen Membern eine solche Charakteristik zuordnet, fällt auf, dass bei einer persistenten Klasse genau die datenorientierten Member für den OR-Mapper interessant sind. Sie repräsentieren den Zustand eines Objektes und nur dieser muss persistiert werden, da angenommen werden kann, dass sich das programmatische Verhalten einer Klasse zur Laufzeit nicht ändert.
Aus dieser Perspektive betrachtet bekommt der in C# eingeführte Membertyp Property auch eine erheblich größere Bedeutung als lediglich den Vorteil einer verkürzten Notation: im Gegensatz zu Methoden ist er explizit datenorientiert. Dass Properties Member einer Interface-Definition sein können ist im Sinne Pughs konsequent und kann für einen OR-Mapper bei der Analyse einer Klasse außerordentlich nützlich sein. Die Properties eines Domäneninterfaces entsprechen den datenorientierten Membern, die Methoden hingegegen den vorgangsorientierten Membern.
Es ist überraschend, dass bei existierenden OR-Mappern persistente Interfaces kaum eine Rolle spielen. Einzig DataObjects.NET [DAT06] bietet Möglichkeiten zur

Persistierung von Interfaces. Diese sind allerdings unbefriedigend umgesetzt, da ein persistentes Interface von einem sehr umfangreichen Basisinterface abgeleitet sein muss, um vom Mapper persistiert zu werden.

5.4 Aufbau der Proxyinfrastruktur des Prototypen

Die im folgenden beschriebene Proxyinfrastruktur des OR-Mappers setzt die im vorigen Abschnitt beschriebene Proxytechnik von Vererbung und Domäneninterfaces um. Bei der Umsetzung wurde darauf geachtet, dass die Proxyinfrastruktur prinzipiell auch mit anderen Proxytechnologien (vgl. Abschnitt 5.2) benutzt werden kann.
Die wichtigsten Aufgaben der Proxyinfrastruktur sind:

- Das Analysieren von Domäneninterfaces und das Extrahieren von Metadaten
- Die zentrale Ablage und Verwaltung von Metadaten
- Das Erzeugen und Kompilieren des Codes der einzelnen Proxyklassen
- Das Erzeugen von einzelnen Proxyinstanzen

Jeder dieser Aufgaben wird einer Komponente der Proxyinfrastruktur zugeordnet, diese sind: der Type-Analyzer, die Proxy-Registry, die Proxy-Bakery sowie die Proxy-Factory. Abbildung 5.1 zeigt den Aufbau der Proxyinfrastruktur.

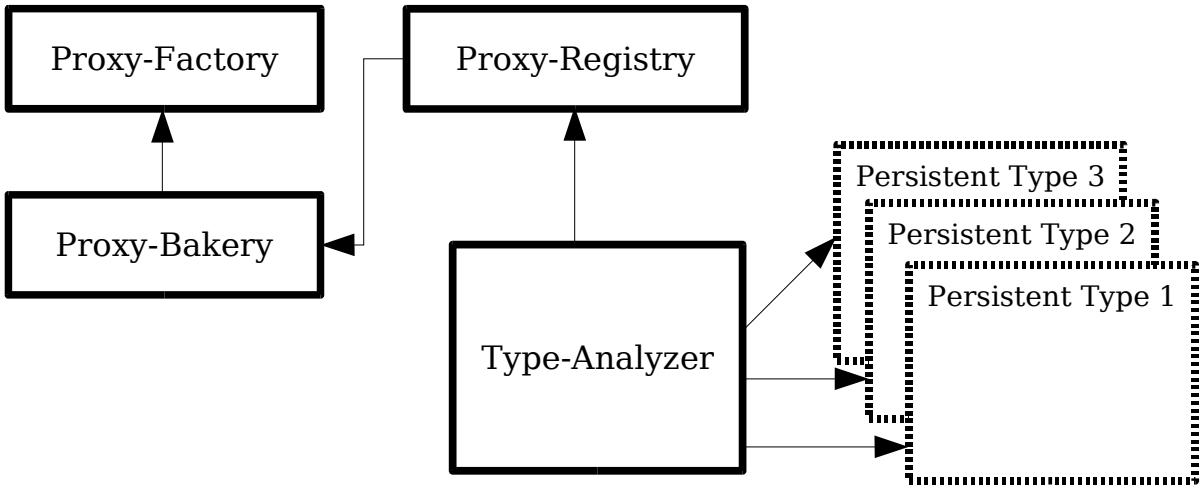


Abbildung 5.1: Proxyinfrastruktur des Prototypen

Die Zusammenarbeit der Komponenten läuft dabei folgendermaßen ab:
Ein Domäneninterface, welches vom Prototypen persistiert werden soll, wird gemeinsam mit der implementierenden Domänenklasse beim Type-Analyzer registriert. Dieser ermittelt die einzelnen Member, welche persistent sein sollen

und wie die Abbildung der Datentypen auf die Datentypen des RDBMS erfolgen soll. Bei den persistenten Properties wird unterschieden zwischen Primitivattributen wie zum Beispiel int oder String, persistenten Referenzen sowie Collections von persistenten Referenzen. Bei persistenten Referenzen sowie deren Collections muss sowohl das referenzierende als auch das referenzierte Domäneninterface als persistent markiert sein. Persistente Collections benötigen zusätzlich Angaben zu ihrer Kardinalität (siehe Kapitel 6).

Nach der Ermittlung der Metadaten werden diese in der Proxy-Registry abgelegt. Da die Metadaten auch von anderen Komponenten des Mappers benutzt werden müssen, beispielsweise um die SQL-Kommandos zu konstruieren, muss die Proxy-Registry auch von anderen Komponenten des OR-Mappers zugreifbar sein.
Die Proxy-Registry ist so implementiert, dass sie zwei Zustände einnehmen kann: offen oder versiegelt. Im offenen Zustand können neue Interfaces registriert werden, in dieser Zeit kann allerdings keine neue Session geöffnet werden, da hierfür zunächst die Metadaten konsistent und der Code für die Proxyklassen erzeugt sein muss. Nach der Registrierung von Interfaces muss die Proxy-Registry wieder geschlossen werden, bevor der Entwickler mit der Benutzung des Mappers beginnen und eine Session eröffnen kann.
Beim Schließen der Proxy-Registry werden die Metadaten der neu registrierten Domänenklassen an die Proxy-Bakery übergeben. Die Aufgabe der Proxy-Bakery ist es, für diese die entsprechenden Proxyklassen in Form von Quellcode zu erzeugen und anschließend zu kompilieren. Wenn die Kompilierung der Proxyklassen erfolgt ist, können Instanzen dieser Proxies über `System.Activator.createInstance()` erzeugt werden.
Das Schema zur Erzeugung der Proxyklassen ist gegenwärtig in der Imeplementierung der Proxy-Bakery festgelegt. Die Proxy-Bakery wird allerdings über das Interface `IProxyBakery` genutzt, damit die verwendete Implementierung leichter ausgetauscht werden kann. Somit können Benutzer des Mappers auch eigene Proxygeneratoren erstellen, um spezielle Anforderungen umzusetzen. Um eigene Proxygeneratoren zu benutzen, muss in der Konfiguration des Mappers eine `IProxyBakery` implementierende Klasse angegeben werden. Einzige Bedingung für solche Proxygeneratoren ist, dass die generierten Proxyklassen entweder das Interface IProxy (siehe Listing 5.1) oder ein davon abgeleitetes Interface implementieren, damit die anderen Komponenten des Mappers mit den Instanzen der generierten Proxyklassen umgehen können.

Die Erzeugung neuer Proxyinstanzen wird zentral in der Proxy-Factory gekapselt

und sollte ausschließlich dort erfolgen, um die korrekte Initialisierung der Proxies sicherzustellen. Die Factory kann sich bei einer Anfrage nach einer Instanz einer persistenten Klasse den entsprechenden Typ der Proxyklasse aus der Proxy-Registry holen, ihn mittels System.Activator.createInstance() instanziieren und anschließend zurückliefern. Dabei werden zwei Methoden zur Proxyinstanziierung angeboten:

- für die Erzeugung neuer Objekte, die weder in der Anwendung noch in der Datenbank existieren. Zur Erzeugung benötigen solche Proxies eine neue Id. Der Status des Proxys wird auf `ProxyState.New` gesetzt.
- für das Laden von bereits in der Datenbank, nicht aber in der Anwendung existierender Objekte. Diesen Proxies muss bei der Instanziierung ihre Id zugewiesen werden. Der Status des Proxys wird auf `ProxyState.Persistent` gesetzt.

Um die mehrfache Erzeugung von Proxies des selben Objekts zu verhindern, hält die Proxy-Factory Referenzen auf alle erzeugten Proxies. Wenn eine Proxy-Id nicht zum ersten Mal zur Erzeugung angefragt wird, so wird der bereits erzeugte Proxy mit dieser Id zurückgeliefert. Damit wird verhindert, dass mehrere Instanzen eines Objekts mit gleicher Id erzeugt werden.

5.5 Vorgehensweise bei der Proxygenerierung

Die Benutzung von Domäneninterfaces hat sich sowohl bezüglich der Transparenz bei der Benutzung von Proxies als auch innerhalb des gesamten Entwurfs- und Entwicklungsprozesses für Anwendungen als vorteilhaft herausgestellt. Aus diesem Grund werden für die Generierung der Proxyklassen durch die Proxy-Bakery die Domäneninterfaces als Ausgangsbasis genommen. Die Interfaces werden vom Type-Analyzer untersucht, um daraus Metadaten über die implementierten Domänenklassen zu gewinnen. Zur Generierung des Proxycodes werden folgende Metadaten benötigt:

- Name des Domäneninterfaces, welches der Proxy ebenfalls implementieren muss
- Name der implementierenden Domänenklasse, von der die Klasse des Proxys abgeleitet wird
- Name und Typ aller persistenten Properties
- Name und Signatur aller Methoden des Interfaces

Als Analysetechnik für die Ermittlung der Metadaten gibt es mehrere Möglichkeiten:

(a)Ermittlung der Member durch Reflection
(b)Außerhalb der Klassendefinitionen abgelegte Metadaten, beispielsweise in Form einer XML-Datei
(c)Markierung von Klassen durch Custom Attributes (siehe Anhang A, entspricht Annotationen in Java 1.5)

Für den Prototypen wurde die Möglichkeit (c) benutzt. Vorteilhaft bei (c) ist, dass sowohl der Code als auch die Persistenzkonfiguration der Klasse in einer Datei und damit an zentraler Stelle vorhanden ist. Der Code wird dadurch übersichtlicher, die Wartbarkeit wird erhöht und Refactorings erleichtert. Da es auch die Möglichkeit geben soll, einzelne Properties einer persistenten Klasse transient zu lassen, ist neben einem Custom Attribute für die Klasse zusätzlich eines für die Markierung von einzelnen Properties vorhanden.
Aus den gewonnenen Metadaten kann die Proxyklasse dynamisch generiert und kompiliert werden. Von den in Abschnitt 4.3.1 vorgestellten Techniken wird die Generierung per Code-DOM genutzt. Im Gegensatz zur Emittierung von IL-Code können Fehler im erzeugten Code einfacher gefunden werden und der Vorgang ist deutlich transparenter, da der Syntaxbaum in C#-Quellcode transformiert und der generierte Quellcode betrachtet werden kann.
Das ursprüngliche Ziel der bedarfsgesteuerten Neuerzeugung der Proxyklassen zur Laufzeit der Anwendung konnte nicht umgesetzt werden. Zwar wäre dies technisch möglich gewesen, da unter DOT.NET verschiedene Application Domains definiert werden können und diese getrennte Adress- und Namensräume besitzen. Die Kommunikation zwischen zwei Application Domains ist jedoch durch Marshaling so langsam, dass die Kommunikation zwischen Proxyobjekten und der Hauptanwendung ineffizient geworden wäre. Als Resultat werden die Proxyklassen nur einmal bei der Registrierung von Klassen und Interfaces beim Mapper erzeugt und kompiliert.

Zur internen Benutzung von Proxyobjekten durch den OR-Mapper wurde das Interface IProxy (siehe Listing 5.1) entwickelt, welches jede Proxyklasse implementieren muss. Dieses Interface stellt alle Methoden und Informationen bereit, welche die Komponenten des OR-Mappers zur Benutzung der Proxy-Objekte braucht:

- die Id des Proxys, diese dient als Primärschlüssel in der Datenbank sowie als Schlüssel in verschiedenen Listen zur Proxyverwaltung innerhalb des Mappers
- den Namen des Domäneninterfaces des Proxys, dieser dient als Schlüssel in der Proxy-Registry zur Abfrage von Metainformationen. Damit können für jeden Proxy die jeweils benötigten Metainformationen ermittelt werden.
- den Status des Proxys (siehe Abschnitt 5.1.2)
- den Gesamtladestatus des Proxies: ob der Proxy bereits geladen wurde oder noch ungeladen ist
- die Ladestati einzelner Properties

Da die Domänenklassen und ihre einzelnen Properties dem OR-Mapper zur Entwicklungszeit nicht bekannt sein können, wird eine Möglichkeit benötigt, die Werte einzelner Properties eines Proxys anhand des Property-Namens auszulesen und zu schreiben. Beim Prototypen wurde dies mit den beiden Methoden `getProp` und `setProp` umgesetzt, welche Werte des Proxys über deren Feldnamen auslesen und setzen können.

```
public interface IProxy {
	string interfaceName { get; }
	Session session { get; }
	string id { get; }
	ProxyState state { get; set; }
	ProxyLoadState loadState { get; set; }
	object getProp(string idx);
	void setProp(string idx, object obj);
	bool isLoaded(string idx);
}
```

Listing 5.1: Das Interface IProxy zur internen Benutzung durch den Mapper

6 Dynamische Ladestrategien für Proxies

Um einzelne Instanzen persistenter Klassen zu laden, kann die Anwendung Anfragen an den OR-Mapper stellen. Der OR-Mapper formuliert diese Anfrage in eine SQL-Abfrage an die Datenbank um, sollte die entsprechende Instanz nicht bereits im Cache vorliegen. Dabei müssen nicht die gesamten Daten des Objekts aus der Datenbank geladen werden – abhängig vom Anwendungskontext genügen oft nur einzelne oder eine bestimmte Gruppe von Daten.

> **Definition**: Unter einer Ladestrategie wird die Systematik verstanden, mit der abgefragte Objekte beim Auslesen aus der Datenbank mit Daten bestückt werden.

Für die Benutzung von Ladestrategien sprechen vor allem zwei Gründe: einerseits sollen keine Daten im Hauptspeicher abgelegt werden, von denen es unwahrscheinlich ist, dass sie in naher Zukunft von der Anwendung benötigt werden. Andererseits sollen die Zugriffe auf die Datenbank möglichst performant durchgeführt werden. Da ein OR-Mapper die SQL-Kommandos zum Auslesen von Daten programmatisch generiert, können diese Anfragen nicht manuell optimiert werden. Dieser Verlust an Performanz gegenüber klassischer Datenbankprogrammierung kann durch angemessene Ladestrategien zumindest zum Teil kompensiert werden.

Das vom OR-Mapper generierte SQL ist mitentscheidend für die Performanz einer Anwendung. Ladestrategien haben direkten Einfluss auf die Generierung der SQL-Abfragen durch den Mapper. Bei der folgenden Behandlung von Ladestrategien muss aus diesem Grund auch betrachtet werden, in welcher Verbindung Ladestrategien zu dem resultierenden SQL stehen und welche Konsequenzen dies für die Performanz der Datenbankabfrage hat.

RDBMS organisieren die Daten einer Relation R in Form von Speicherseiten. Für R kann aufgrund ihrer Attribute berechnet werden, wieviele Datensätze in Form von Tupeln in eine Speicherseite passen. Die Seiten für R werden durch eine Zeigerstruktur verkettet und möglichst auch physikalisch (auf dem Plattenspeicher) benachbart abgelegt, um lineare Lesevorgänge auf R schnell durchführen zu können. Gruppen von Speicherseiten werden inhaltlich zu Segmenten zusammengefasst, zum Beispiel bekommen Index-Daten ein eigenes Segment und werden getrennt von den Nutzdaten verwaltet.

Bei einer Anfrage an das RDBMS werden zunächst die Seiten ermittelt, aus denen

die angeforderten Daten gelesen werden können. Dabei kann das RDBMS Kontextwissen nutzen, um die Speicherseiten geschickt zu verwalten. Falls die benötigten Seiten im Seitencache des RDBMS liegen, können sie von dort gelesen werden. Falls nicht, müssen die Seiten vom Plattensystem gelesen werden. Insbesondere diese Zugriffe sind es, die lange dauern und die nach Möglichkeit zu minimieren sind.
Je nach Anfrage müssen die Daten hinterher noch gefiltert und sortiert werden. Falls für die Relation Indizes angelegt wurden, können diese für die Ermittlung der Speicherseiten benutzt werden. Bei umfangreichen Mehrspalten-Indizes kann sogar ein Lesen der eigentlichen Nutzdaten erspart bleiben (vgl. [LAH05] S. 49ff). Ohne Indizes müssen in jedem Fall alle Speicherseiten gelesen werden, die R zugeordnet sind. Weitergehende Ausführungen zu Verwaltungsstrategien von Speicherseiten bei RDBMS finden sich bei [SIL06] S. 461ff oder ausführlicher [HÄR01] S. 107ff.

6.1 Bestehende Ansätze für Ladestrategien

Gegenwärtig existieren bei OR-Mappern lediglich zwei Ansätze für Ladestrategien: lazy-load für Objekte sowie für einzelne Properties. Bei lazy-load auf Objektebene wird lediglich ein leerer Proxy mit seiner Id als Ergebnis einer Abfrage zurückgeliefert. Erst wenn ein Lesezugriff auf Properties dieses Proxys stattfindet, werden seine Daten aus der Datenbank ausgelesen. Als Ladestrategie ist dieser Mechanismus jedoch sehr grobgranular, weshalb ausgereiftere OR-Mapper lazy-load auf Property-Ebene anbieten. Dabei kann für jedes einzelne Property definiert werden, ob dieses erst beim tatsächlichen Lesen des Propertys durch die Anwendung aus der Datenbank geladen wird. Bei einem Schreiben eines solchen Propertys entfällt sein vorheriges Nachladen, da der alte (unbekannte) Wert ohnehin überschrieben wird.
Die Frage nach einer Ladestrategie stellt sich auch bei Collections von Referenzen auf persistente Objekte. Ein typisches Beispiel hierfür zeigt Abbildung 6.1, eine Beziehung zwischen Customer, Order und Product:

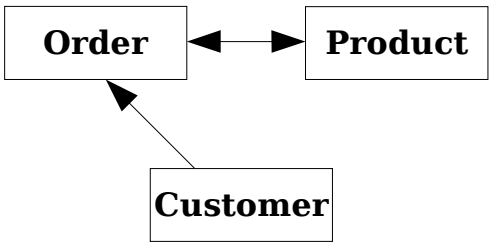


Abbildung 6.1: Domänenmodell mit Customer, Order und Product

Wenn ein bestimmter Customer aus der Datenbank gelesen wird, müssen nicht zwingend die gesamten Daten der Order-Collection mitgelesen werden, zum Beispiel wenn nur der Name des Kunden benötigt wird. Auch in diesem Fall kann die Performanz der Anfrage an die Datenbank durch den OR-Mapper mithilfe einer geeigneten Ladestrategie optimiert und der Hauptspeicher nur soweit wie nötig gefüllt werden.
Problematisch ist bei existierenden Umsetzungen von einfachen Ladestrategien wie lazy-load, dass diese beim Start der Anwendung in der Konfiguration des Mappers festgelegt werden. Sie bleiben jedoch im weiteren Verlauf der Anwendung unveränderbar. Bei ungenügender Performanz von Anfragen gibt es zwar die Möglichkeit, die Anfragen umzuformulieren, um das gewünschte Ladeverhalten zu forcieren (siehe [HOH06] S. 48). Allerdings läuft dies der Absicht entgegen, dass Anfragesprachen rein deskriptiv sein sollten. Anfragen sollten also lediglich eine Beschreibung der gewünschten Daten enthalten. Aus diesem Grunde ist es wünschenswert, für den jeweiligen Kontext der Anwendung angemessene Ladestrategien zur Laufzeit definieren zu können. Solche zur Laufzeit definierten Ladestrategien werden im folgenden als dynamische Ladestrategien bezeichnet.

Für die Entwicklung eines Mechanismus zur Umsetzung effizienter Ladestrategien muss zunächst unterschieden werden, um welche Art von Daten es sich bei einem Property einer persistenten Klasse handelt. Die im folgenden behandelten Kategorien persistenter Properties sind:

- Primitivattribute (bspw. int, double, char oder string)
- Referenzen auf persistente Objekte
- Collections von Referenzen auf persistente Objekte

In den folgenden Abschnitten sollen für diese Kategorien von Properties Teilladestrategien ermittelt werden, welche sich zu einer Gesamtladestrategie einer persistenten Klasse zusammensetzen lassen. Anschließend wird die Umsetzung des Entwurfs im Prototypen behandelt.

6.2 Ladestrategien für einfache Primitivattribute

Für ein einzelnes Primitivattribut einer persistenten Klasse gibt es beim Laden nur zwei Möglichkeiten: entweder es wird initial mitgeladen oder erst beim tatsächlichen Zugriff der Anwendung. Ob beim Laden eines Tupels der Relation ein

einzelnes Attribut weggelassen wird, macht für die Performanz der Abfrage nur in zwei Fällen einen großen Unterschied:

1. Es wird eine sehr große Menge an Objekten aus der Datenbank gelesen.
2. Bei dem Attribut handelt es sich um einen Datentyp, der sehr umfangreich sein kann und deshalb von der Datenbank in einer eigenen Datenstruktur verwaltet wird.

Ein Beispiel für solche Datentypen sind BLOBs (Binary Large Objects). Bezüglich der Performanz stellt der Datentyp BLOB ein RDBMS vor Schwierigkeiten, da seine maximale Kapazität sehr groß ist, der tatsächlich genutzte Speicherplatz jedoch stark schwanken kann. Durch ihre potentielle Größe können nur sehr kleine BLOBs in einer normalen Datenbank-Speicherseite abgelegt werden, große BLOBs werden nur durch einen Verweis in der Speicherseite referenziert (vgl. [HÄR01] S. 169ff). Dadurch besteht bei einem Zugriff auf ein BLOB-Feld das Risiko eines zusätzlich anfallenden Zugriffs auf die Speicherseiten, auf denen die Daten des BLOBs abgelegt wurden.
Der Prototyp unterstützt Primitivattribute vom Typ char, int, double und String[1]. Unter diesen kann allein der Typ String im Umfang stark variieren. String-Attribute können deshalb mit dem Custom-Attribute `LargePrimitive` markiert werden, sie werden dann in der Datenbank auf einen BLOB statt auf einen VARCHAR abgebildet.
Für die Ladestrategie einer Klasse bedeutet dies, dass das String-Attribut entweder initial oder erst bei Bedarf geladen werden kann. Für ein solches lazy-load wird dem Attribut `LargePrimitive` ein Initialisierungsparameter übergeben. Bei der Erzeugung der Proxyklassen wird berücksichtigt, ob lazy-load für das Property möglich sein soll. Die Proxy-Bakery erzeugt für jede Klasse mit lazy-load-Properties die Proxylogik, die im Proxy speichert, welche der lazy-load-Felder bereits geladen wurden. Außerdem überprüft sie bei Zugriffen auf diese Properties den Status des Feldes und läd bei Bedarf den Wert aus der Datenbank nach.

6.3 Ladestrategien für persistente Referenzen

Ein Attribut einer persistenten Klasse, welches auf eine Instanz einer persistenten Klasse zeigt, wird im folgenden als persistente Referenz bezeichnet. Persistente Referenzen müssen sich entsprechend in der Datenbank wiederfinden, um beim nächsten Ablauf der Anwendung wiederhergestellt werden zu können. Im

1In der Sprache C# ist der Typ String kein Primitivtyp, sondern eine Klasse. Durch seine Überschaubarkeit wird er im Folgenden trotzdem als Primitivtyp behandelt.

Gegensatz hierzu stehen transiente Referenzen, also Verweise auf nicht-persistente Klassen, die nach Beenden der Anwendung verloren sind und vom OR-Mapper nicht berücksichtigt werden müssen.
Für persistente Referenzen sind unterschiedliche Kardinalitäten möglich: 1:1, 1:n oder m:n. Eine effiziente Ladestrategie muss dabei die Kardinalität berücksichtigen, da sich diese im Datenbankschema wiederspiegelt. Im folgenden werden zunächst 1:1-Referenzen behandelt. Bei diesen wiederum wird zwischen inversen und nichtinversen Referenzen unterschieden. Inverse Referenzen bieten die Möglichkeit, im Klassenmodell in beide Richtungen zu navigieren. Dazu müssen die Referenzen bei beiden Instanzen konsistent sein. Bei nichtinversen Referenzen ist die Navigation im Klassenmodell nur in eine Richtung möglich.
Persistente Referenzen eines Attributs der Klasse K_A auf eine Instanz der Klasse K_B können als Fremdschlüssel eines Attributs in Relation R_A auf ein Tupel in R_B abgebildet werden. Die Schlüsselwerte entsprechen dabei sinnvollerweise der Id des persistenten Objekts (siehe Abschnitt 3.2.1). RDBMS legen für solche Fremdschlüssel-Beziehungen automatisch Indizes an, um Verbundoperationen effizient durchführen zu können. Die von RDBMS angebotenen Fremdschlüsselkaskadierungen (wie bspw. on delete cascade) dürfen für solche persistenten Referenzen nur dann benutzt werden, wenn das referenzierte Objekt ausschließlich vom referenzierenden Objekt referenziert werden kann und niemals von einem anderen Objekt referenziert wird.

Da der Wert des Fremdschlüssels klein ist und somit vom RDBMS mit den anderen Primitivattributen eines Datensatzes gemeinsam auf einer Speicherseite abgelegt werden kann, verursacht es kaum Mehrkosten, wenn beim Lesen der Daten einer Instanz von K_A die Objekt-Id der referenzierten Instanz von K_B ebenfalls mitgelesen wird. Die Ladestrategie lazy-load kann sich auf das Auslesen dieses Fremdschlüssels beschränken, in diesem Fall würde vom OR-Mapper ein leerer Proxy P_B instanziiert und nur mit der Id bestückt. Der Status des Proxy wird auf `ProxyLoadState.Unloaded` gesetzt, so dass bei Zugriffen auf den Proxy zu einem späteren Zeitpunkt seine Daten nachgeladen werden müssen. Die als eager-load bezeichnete Strategie hingegen würde die Daten der referenzierten Instanz von K_B mit auslesen, was zu einer zusätzlichen Anfrage an die Datenbank mit den entsprechenden Mehrkosten führt.

Wenn eine invers-exklusive 1:1-Beziehung zwischen zwei Klassen vorliegt, besteht auch die Möglichkeit, diese beiden Klassen auf eine Relation abzubilden (vgl. [AMB03] S. 252). Dieser Ansatz ist performanter, da mit einer SQL-Abfrage beide

Instanzen von K_A und K_B mit Daten befüllt werden können. Allerdings ist aus Modellierungssicht diese Methode fragwürdig, da die Kohäsion der Relation gering ist. Zudem ist es nicht ungewöhnlich, dass sich im Laufe des Lebenszyklus der Gesamtanwendung die Kardinalität von persistenten Referenzen zwischen Klassen ändern, zum Beispiel wenn für eine Person mehrere Adressen statt lediglich einer gespeichert werden sollen. In solchen Fällen ist der Aufwand für ein Datenbank-Refactoring deutlich höher.

6.4 Ladestrategien für Collections von persistenten Referenzen

Thema dieses Abschnitts sind Beziehungen zwischen den Klassen K_A und K_B, welche die Kardinalität 1:n und m:n haben. Eine solche Beziehung wird in einem Klassenmodell als Collection von persistenten Referenzen modelliert. Die Kardinalität 1:n kann ebenso wie eine 1:1-Beziehung als eine Fremdschlüsselbeziehung zwischen zwei Relationen der Datenbank abgebildet werden (vgl. [AMB03] S. 247). Eine Instanz von K_A hat dabei ein Property vom Datentyp Collection (bspw. List oder Set) und verweist so auf mehrere Instanzen von K_B, deren Referenzen die Collection enthält. Eine Instanz von K_B wiederum kann entweder von einer Instanz von K_A (Kardinalität 1:n) oder mehreren Instanzen von K_A (Kardinalität m:n) referenziert werden.
Ladestrategien für Collections sollen verhindern, dass Daten, deren Benutzung in naher Zukunft unwahrscheinlich ist, geladen werden. Beim initialen Ladevorgang der Collection sollten also die genau die Daten geladen werden, welche innerhalb der Anwendung mit hoher Wahrscheinlichkeit benötigt werden. Die Collection-Datentypen von Wirtssprachen unterscheiden sich in einem Punkt von den Collection-Klassen, welche Ladestrategien ermöglichen: sie kennen keinen Zustand, in dem nur Teile ihrer Nutzdaten geladen sind. Aus diesem Grund muss ein OR-Mapper, welcher Ladestrategien für Collections umsetzt, eigene Collection-Klassen umsetzen oder von den entsprechenden Klassen der Wirtssprache ableiten und deren Zugriffsmethoden überschreiben. Beim Überschreiben der Zugriffsmethoden auf die enthaltenen Elemente der Collection muss berücksichtigt werden, welche Daten bereits geladen wurden.
Eine weitere Möglichkeit ist die Implementierung von Collection-Interfaces. Diese Möglichkeit wurde für den Prototypen genutzt und eine Klasse `PersistentList` entwickelt, welche eines der DOT.NET Standard-Interfaces für Collections (`IList`) implementiert. Der Typ `PersistentList` kennt drei Ladezustände, die Tabelle 6.1 zu entnehmen sind:

CollectionLoadState	Zustand
Unloaded	Keine Daten sind geladen, die Collection ist lediglich initialisiert
IdsLoaded	Die Collection hat eine vollständige Liste der Ids derjenigen Objekte, die sie enthält. Allerdings ist unbekannt, ob die entsprechend referenzierten Objekte bereits geladen sind oder nicht
Loaded	Die Referenzen der Collection sind vollständig gesetzt, die entsprechenden Proxyobjekte also instanziiert. Allerdings ist unbekannt, ob die Proxies bereits mit Daten gefüllt wurden

Tabelle 6.1: Ladestati persistenter Collections

Insbesondere der Zustand `IdsLoaded` erfordert eine nähere Erläuterung. Einen solchen Zustand einzuführen ist deshalb sinnvoll, weil eine Reihe wichtiger und häufig benutzter Collection-Operationen keine vollständig geladene Collection benötigen, zum Beispiel `contains()` oder `count()`. Der Zustand `IdsLoaded` stellt einen Zwischenzustand dar, weil die Referenzen auf die enthaltenen Elemente der Collection zwar noch nicht gesetzt sein müssen, die Ids der einzelnen Elemente jedoch bekannt sind. Mit diesem Zustand kann verhindert werden, dass für ein `count()` auf einer großen Collection sehr viele Proxyobjekte erzeugt und ihre Daten geladen werden, obwohl dies für die Durchführung dieser Operation auf Collections nicht notwendig ist.
Damit die Collection sich für den Anwendungsentwickler transparent verhält, muss sie eine eigene Proxylogik umsetzen, die dafür sorgt, dass nach außen unbemerkt bleibt, in welchem Ladezustand sich die Collection zum Zeitpunkt eines Zugriffs befindet. Dazu werden die Nutzdaten der Collection innerhalb von `PersistentList` aufgeteilt. Es wird jeweils eine Liste für die Ids der Elemente und eine Hashmap für die eigentlichen Elemente gepflegt. Zu diesem Zweck wird in jeder Methode von IList überprüft, in welchem Zustand sich die Collection befindet und läd bei Bedarf Daten nach.

Die Proxylogik der Collection kann zusätzlich den Einsatzkontext der Collection berücksichtigen. Dabei wird unterschieden zwischen Punktzugriffen auf einzelne Elemente und Iterationszugriffen, bei welchen alle Elemente der Collection durchlaufen werden. Näheres hierzu findet sich in Abschnitt 6.6.2.

6.5 Kaskadierende Ladestrategien

Wenn in der Anwendung über persistente Referenzen auf Objekte zugegriffen wird,

deren Nutzdaten noch nicht geladen sind, müssen diese Daten aus der Datenbank nachgeladen werden. Solche Nachladevorgänge können zu Verzögerungen führen, die sich in der Anwendung für den Benutzer unangenehm bemerkbar machen. Eines der Ziele der Entwicklung von Ladestrategien ist es deshalb, ein Nachladen zu ungeeigneten Zeitpunkten zu verhindern und die Verzögerung durch den Datenbankzugriff auf einen geeigneteren Zeitpunkt zu legen.

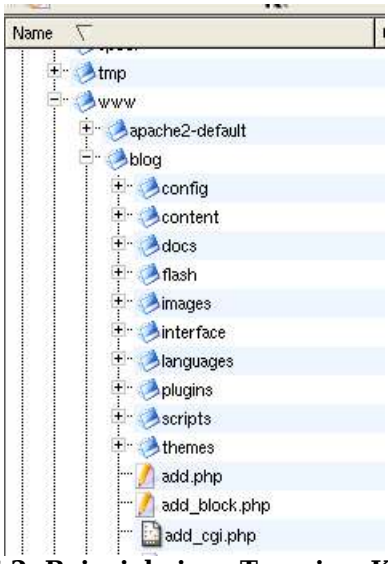


Abbildung 6.2: Beispiel einer Treeview-Komponente

Ein Beispiel hierfür ist eine Treeview-Komponente. Eine solche Komponente visualisiert eine baumartige Datenstruktur wie zum Beispiel ein Dateisystem. Während beim Öffnen eines neuen Fensters mit dem Treeview eine bestimmte Verzögerung für den Benutzer akzeptabel ist, so muss das Auf- und Zuklappen der Teilbäume oder ein mouseover-Effekt schnell vorgehen, damit die Komponente angenehm benutzbar ist. Angenommen den Fall, dass die im Treeview benutzten Daten in persistenten Objekten abgelegt sind - dann sollten beim Öffnen des Fensters auch die Objekte mitgeladen werden, die zunächst in zugeklappten Teilbäumen nicht sichtbar sind oder die erst bei einem mouseover angezeigt werden.

Um dies mithilfe von Ladestrategien umzusetzen, müssen persistente Referenzen beim Ladevorgang eines Objekts verfolgt und die referenzierten Objekte mitgeladen werden. Dieser Vorgang wird im weiteren als kaskadierendes Laden bezeichnet. Zur Analyse und Einschätzung von kaskadierenden Ladevorgängen innerhalb einer bestimmten Anwendung ist es hilfreich, ihr Modell der persistenten Domänenklassen zu betrachten. Dieses Modell kann als Grundlage für einen sogenannten Vernetzungsgraphen (im Folgenden V genannt) dienen.

Definition: Ein Vernetzungsgraph **V** besteht aus einem Tupel **<K, S, N, M>** mit:
K: Menge der Domänenklassen als Knoten
Sowie den Kanten:
S: einer Abbildung $K_A,K_B \rightarrow \mathbb{N}_0$ für die Anzahl von 1:1-Referenzen von Attributen K_A nach K_B
N: einer Abbildung $K_A,K_B \rightarrow \mathbb{N}_0$ für die Anzahl von 1:n-Referenzen von Attributen K_A nach K_B
M: einer Abbildung $K_A,K_B \rightarrow \mathbb{N}_0$ für die Anzahl von m:n-Referenzen von Attributen K_A nach K_B

Abbildung 6.3 zeigt ein Beispiel für einen solchen Vernetzungsgraphen mit neun Domänenklassen. Doppelkanten wie die von K_F nach K_I stehen dabei für mehrere persistente Referenzen auf die gleiche Klasse, bspw. die Klasse `Team` mit Referenzen auf die Klasse Person in den Properties `teamMembers` und `teamManager`.

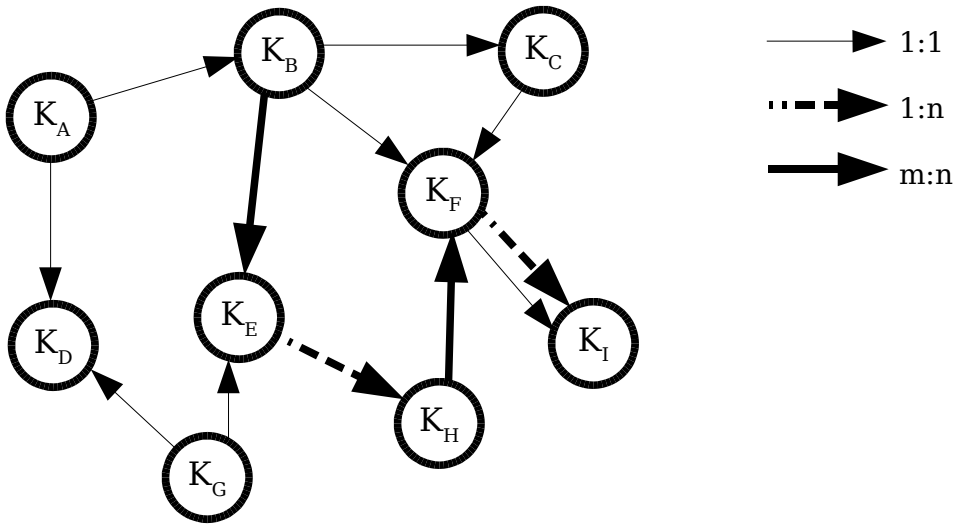


Abbildung 6.3: Beispiel für einen Vernetzungsgraph

Für die Analyse kaskadierender Ladevorgänge ist neben dem Vernetzungsgraphen die sogenannte **Kaskadierungstiefe** wichtig: unter der Kaskadierungstiefe wird die Anzahl der Schritte verstanden, über die, vom zu ladenden Objekt ausgehend, persistente Referenzen verfolgt werden und deren referenzierte Objekte mitgeladen werden. Wenn in dem Vernetzungsgraphen aus Abbildung 6.3 eine Instanz von K_B geladen wird, führt ein kaskadierender Ladevorgang mit C als Kaskadierungstiefe zu Zugriffen auf folgende Klassen:

- C=1: K_E, K_F, K_C
- C=2: K_H, K_I, K_I, K_F

Es ist offensichtlich, dass mit steigendem C und den pro Klasse K durchschnittlich ausgehenden Verbindungen s_K, n_K und m_K die Anzahl von benötigten Ladeoperationen exponentiell wächst. Dies gilt sowohl für das vollständige Laden aller Attribute als auch für das Laden der Ids. Es ergibt sich hierfür die Abschätzung:

Erforderliche einzelne Ladeoperationen für vollständiges Laden:
$LOAD(K) = (s_K + n_K + 2 \cdot m_K)^C$

mit s_K, n_K und m_K als durchschnittlichen Werten für ausgehende Kanten, gemittelt über alle Domänenklassen

Der Faktor 2 für das Laden von Collections der Kardinalität m:n ergibt sich daraus, dass beim Laden der Nutzdaten solcher Collections mittels Verbund auf zwei Relationen zugegriffen werden muss: auf die Mapping-Relation, die lediglich die Id-Zuordnungen enthält sowie auf die Relation mit den eigentlichen Nutzdaten der Elemente der Collection.

Anzumerken ist, dass Zyklen im Graphen nicht zu einer Einsparung von Ladeoperationen führen, da die jeweils referenzierten Objekte nicht die gleichen sein müssen. Selbst wenn sie es sind, so kann dies beim Laden nicht als bekannt vorausgesetzt werden, da die entsprechenden Ids erst aus den einzelnen Relationen ermittelt werden müssen.

Die bei steigender Kaskadierungstiefe schnell steigende Anzahl von ausgelösten Ladevorgängen kann zu einem Lawineneffekt führen. Dabei werden als Folge einer Ladeoperation so viele kaskadierte Ladeoperationen ausgelöst, dass der gesamte Vorgang sehr langsam wird. Ein Vernetzungsgraph bietet jedoch eine gute Möglichkeit, für einen Ladevorgang der Klasse K mit dem Kaskadierungslevel C eine Folgeabschätzung vorzunehmen. Besonders groß ist die Gefahr des Lawineneffekts, wenn sich die Kaskadierung auch auf Collections erstreckt. Hierbei ist der Vernetzungsgraph zur Wirkungsabschätzung nur eingeschränkt nützlich, da vor allem die Anzahl der in den Collections enthaltenen Elemente ausschlaggebend für die Geschwindigkeit der ausgelösten Ladeoperationen ist. Die Anzahl der Elemente einer Collection ist jedoch dem Klassenmodell nicht zu entnehmen und während der Entwicklung einer Anwendung nicht immer abschätzbar. Diese Größe stellt sich häufig erst im laufenden Betrieb der Anwendung heraus, so dass erst zu diesem Zeitpunkt mittels Ladestrategien darauf reagiert werden kann.

Da der Lawineneffekt insbesondere bei der Kaskadierung des Ladens von

Collections mit C>1 eintritt, wurde beim Prototypen auf die Möglichkeit der Kaskadierung von Collections bei Ladezugriffen verzichtet. Kaskadierendes Laden erstreckt sich dort also nur über einfache persistente Referenzen. Direkt im zu ladenden Objekt vorhandene Collection-Properties können über die behandelten Collection-Ladestrategien individuell für jede Collection oder generell innerhalb der Ladestrategie gesetzt werden.

6.6 Kontextabhängige Ladestrategien für Collections

Persistente Collections werden bei OR-Mappern auf zwei Arten verwendet:

- als persistente Referenzen der Kardinalität 1:n oder m:n
- als Ergebnis einer Bereichsanfrage gegen alle Instanzen einer persistenten Klasse K. Eine solche Anfrage liefert eine (unter Umständen leere) Liste von Instanzen von K zurück, welche die Prädikate der Anfrage erfüllen

Die bis zu diesem Punkt entwickelten Ladestrategien sahen für eine Collection lediglich drei unterschiedliche Ladeoperationen vor: nichts zu laden, die Ids der Collection-Elemente zu laden oder sämtliche Nutzdaten dieser Objekte zu laden.
Es zeigt sich, dass diese Ladestrategien zu einfach gehalten sind, da sie nicht den Verwendungszweck berücksichtigen, für den die Collection an einer bestimmten Stelle der Anwendung benutzt wird. Es existiert bei jeder Verwendung einer persistenten Collection ein Kontext, den eine optimale Ladestrategie berücksichtigen muss, um Ladeoperationen systematisch und möglichst performant durchführen zu können. Dies soll am Beispiel von zwei unterschiedlichen Einsatzkontexten verdeutlicht werden:

1. Eine Liste der Namen aller Kunden soll dem Benutzer angezeigt werden, so dass dieser sich Details einzelner Kunden und deren Bestellungen anzeigen lassen kann. Eine optimale Ladestrategie sollte zunächst nur die für die Auswahl des Kunden zur Anzeige benötigten Daten laden. Nachdem der gewünschte Kunde durch den Benutzer ausgewählt wurde, werden die restlichen Daten dieses Kunden sowie die seiner Bestellungen nachgeladen. Somit wird ein Laden der Bestellungsdaten aller Kunden vermieden.
2. Es soll die Emailadresse derjenigen Kunden ermittelt werden, welche mindestens eines von mehreren Prädikaten erfüllen. Zu diesem Zweck wird eine Schleife programmiert, welche die Liste aller Kunden durchläuft und die einzelnen Prädikate überprüft. Wird eines der Prädikate erfüllt, so wird die Emailadresse

des aktuellen Kunden einer String-Liste hinzugefügt.
Eine optimale Ladestrategie muss hierzu die zur Überprüfung der Prädikate benötigten Daten der Kunden laden. Falls sich die benötigten Daten in vom Kunden-Objekt referenzierten Objekten befinden, müssen die Daten dieser Objekte nachgeladen werden.

Es wird hier deutlich, dass die Einsatzkontexte persistenter Collections sehr unterschiedlich sein können. Da bei beiden vorgestellten Kontexten das initiale Laden sämtlicher Daten unerwünscht ist, müssen die unterschiedlichen Nachladevorgänge nach Möglichkeit genau die Daten nachladen, welche in naher Zukunft mit hoher Wahrscheinlichkeit benötigt werden. Die Steuerung solcher Nachladevorgänge wird im folgenden als kontextabhängige Ladestrategie bezeichnet.
Für die Umsetzung solcher Strategien wird zunächst ermittelt, was unter dem Kontext eines Datenzugriffs verstanden wird. Da ein solcher Kontext nahezu beliebig komplex werden kann, wird der Umfang des Kontextes auf folgende Aspekte begrenzt:

- beabsichtigter Verwendungszweck der Collection
- bislang erfolgte Datenzugriffe auf einzelne Elemente der Collection, die mit hoher Wahrscheinlichkeit für die anderen Elemente ebenfalls durchgeführt werden

Die Umsetzung zielt darauf ab, bisher erfolgte Datenzugriffe auf die Collection zu analysieren und in Abhängigkeit des Verwendungszwecks für die folgenden, zu erwartenden Collection-Zugriffe zu nutzen. Beim Verwendungszweck einer Collection kann dabei zwischen Iterationszugriffen und Navigationszugriffen unterschieden werden. Die für den Prototypen entwickelte `PersistentList` erlaubt dadurch, dass sie dass Standard-Collection-Interface `IList` des DOT.NET-Frameworks implementiert, sowohl Iterations- als auch Navigationszugriffe auf einzelne Elemente.

6.6.1 Navigationszugriffe

Bei einem Navigationszugriff wird ein Objekt (als Element der persistenten Collection) als Ausgangspunkt benutzt, um im Objektmodell über persistente Referenzattribute des Objekts zu anderen Objekten zu gelangen. Solche Zugriffe können auch über mehrere Schritte Referenzen verfolgen, um zu den gewünschten Daten zu navigieren. Navigationszugriffe sind dadurch gekennzeichnet, dass die

Richtung der Navigation im Objektmodell (dass heißt, welche Referenzen verfolgt werden und welches Element der Collection den Ausgangspunkt bildet) durch den Benutzer bestimmt wird und damit vom Mapper nicht vorhersehbar ist.
Eine geeignete Ladestrategie für solche Zugriffe muss initial nur die benötigten Daten laden, die ein Benutzer als Grundlage seiner Entscheidung zum weiteren Navigieren benötigt. Die durch die Benutzerentscheidung folgenden Nachladezugriffe können analysiert, in abstrakter Form gespeichert und bei Zugriffen im gleichen Kontext wiederverwendet werden. Somit wird das Nachladen der benötigten Daten nicht stückweise vorgenommen, sondern kann systematisch gebündelt werden. Abbildung 6.4 veranschaulicht einen Navigationszugriff auf eine Collection. Die Wahl des Benutzers fällt auf das Element E_3, in der Folge werden weitere Teile des Objektgraphen nachgeladen. Die Nachladevorgänge sind schraffiert hinterlegt.

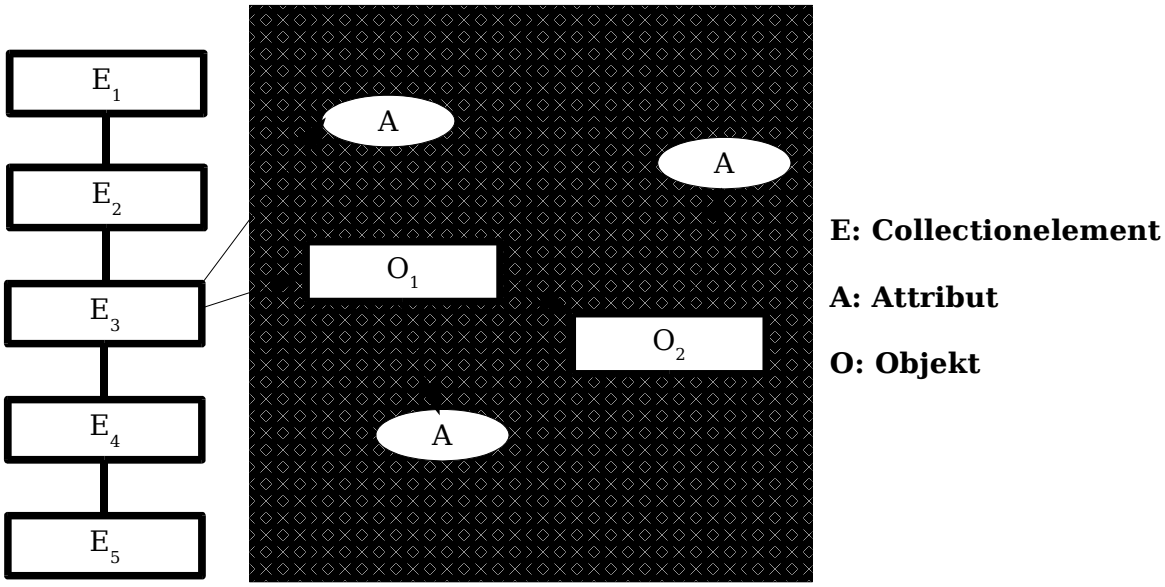


Abbildung 6.4: Beispiel für einen Navigationszugriff

Um Nachladevorgänge bei Navigationszugriffen wiederverwenden zu können, müssen die einzelnen vorgenommenen Nachladezugriffe miteinander in Verbindung gebracht werden. Das Ergebnis einer solchen Verknüpfung von Nachladevorgängen wird im folgenden Ladebaum genannt. Ladebäume dienen dazu, verkettete Nachladeoperationen zu modellieren und in abstrakter Form darzustellen. Ein Ladebaum enthält alle Nachladeoperationen, die innerhalb eines begrenzten Zeitraums (von einem bestimmten Objekt ausgehend) ausgelöst werden. Ein Ladebaum hat ein Objekt als Wurzelknoten, welches im Folgenden als Wurzelobjekt bezeichnet wird. Alle Nachladeoperationen, welche durch Zugriffe auf ungeladene Attribute (Primitivattribute, persistente Referenzen, persistente Collections) dieses Wurzelobjekts ausgelöst werden, bilden die Kindknoten des Wurzelobjekts. Somit

sind auch mehrstufige Nachladevorgänge darstellbar, also Kindknoten der Kindknoten des Wurzelknotens. Eine Navigation über n persistente Referenzen erfordert damit Nachladevorgänge der Grade 1, 2,..., n-1 und n, falls die entsprechenden Objekte nicht bereits geladen sind.
Mit Ladebäumen können die anfallenden Nachladeoperationen sehr detailliert modelliert werden. Wichtig ist beim Konzept des Ladebaums, dass leicht von seinem Wurzelknoten und dessen konkreten Verknüpfungen mit den entsprechenden Ids abstrahiert werden kann. Der Ladebaum kann hierdurch leicht übertragen werden, indem er an andere Objekte als Wurzelobjekte gebunden wird.
Abbildung 6.4 zeigt einen Ladebaum, der an E_3 gebunden ist. Wenn der Benutzer sich nach dem Betrachten der Details von E_3 dafür entscheidet, die Details von E_4 anzusehen, kann der Mapper diesen Nachladebaum an E_4 binden und alle Nachladevorgänge gebündelt vornehmen.

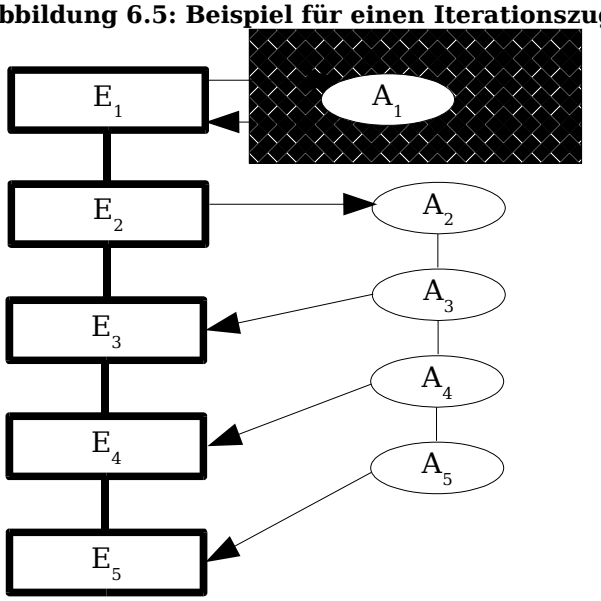
6.6.2 Iterationszugriffe

Bei einem Iterationszugriff wird eine persistente Collection in einer Schleife durchlaufen. Dabei ist zu erwarten, dass in aller Regel auf alle Elemente der Collection zugegriffen wird. Wenn die zugegriffenen Daten der Collection nur teilweise geladen sind, entsteht dabei das Problem, dass bei jedem Schleifendurchlauf für jedes Element ein oder mehrere strukturell ähnliche Nachladevorgänge ausgelöst werden. Dieses Problem ist auch als n-1-Selects-Problem bekannt (vgl. [BAU07] S. 585ff). Solche Zugriffe können auch, wie in Abschnitt 6.5.1, über die Verfolgung persistenter Referenzen zum Nachladen von Objekten führen, welche nicht Elemente der Collection ist. Im Gegensatz zu Navigationszugriffen werden solche Nachladevorgänge allerdings programmatisch und nicht durch Entscheidungen des Benutzers ausgelöst.
Eine einfache Ladestrategie für solche Schleifendurchläufe könnte darin bestehen, einen hohen Kaskadierungslevel zu verwenden. Allerdings wurde bereits gezeigt, dass bei hohem Kaskadierungslevel eine Anfrage durch den Lawineneffekt sehr viele einzelne Nachladevorgänge auslösen kann, so dass die Performanz insgesamt zu schlecht wird. Zusätzlich muss berücksichtigt werden, dass bei dieser Strategie pauschal alle Daten in unmittelbarer Umgebung des Objektgraphen mitgeladen werden und insbesondere persistente Referenzen verfolgt werden, auf die im Code der Schleife gar nicht zugegriffen wird. Aus diesen Gründen erweist sich diese Strategie als zu einfach und nicht effizient genug.
Das in Abschnitt 6.5.1 entwickelte Konzept des Ladebaums ist jedoch auch für Iterationszugriffe nutzbar. Dies gilt vor allem deshalb, weil bei

Schleifendurchläufen zu erwarten ist, dass die Nachladeoperationen sehr ähnlich und damit gut abstrahierbar sind. Um eine optimale Ladestrategie für solche Iterationen zu entwickeln, benötigt der OR-Mapper Informationen über die innerhalb der Schleife anfallenden bzw. ausgelösten Nachladevorgänge. Zu diesem Zweck bieten sich zwei Möglichkeiten an: entweder definiert der Domänenentwickler manuell einen Ladebaum, der dem Schleifencode angemessen ist. Nachteilig bei dieser Methode ist allerdings, dass bei Änderungen im Schleifencode die Ladestrategie adäquat mitgepflegt werden muss. Es kommt hinzu, dass komplexe Ladebäume zu definieren recht aufwändig sein kann.
Die zweite Möglichkeit besteht darin, dass der OR-Mapper selbst aus den während des ersten Schleifendurchlaufs vorgenommenen Zugriffen einen Ladebaum extrahiert. Hierzu muss der Mapper die vorgenommenen Nachladevorgänge analysieren, untereinander verknüpfen und zu Beginn der zweiten Iteration für alle weiteren in der Collection enthaltenen Elemente durchführen. Diese Vorgehensweise funktioniert dann besonders gut, wenn der erste Schleifendurchlauf bezüglich der Datenzugriffe repräsentativ für die folgenden Durchläufe ist. Dies ist genau dann der Fall, wenn switch- und if-Verzweigungen im Schleifenkörper nicht zu unterschiedlichen Datenzugriffen führen. Bei einer Collection als Ergebnismenge einer Anfrage an den OR-Mapper ist dies unwahrscheinlich, da es einer weiteren Filterung der Ergebnismenge gleichkäme und entsprechend in die Formulierung der Anfrageprädikate einfließen sollte.

Abbildung 6.5 verdeutlicht das Verfahren: Beim ersten Durchlauf der Schleife wird durch einen Zugriff auf das Attribut A des Collection-Elementes E_1 ein Nachladen des Wertes A_1 veranlasst. Aus diesem Zugriff konstruiert der Mapper den (farblich hinterlegten) Ladebaum. Vor dem zweiten Durchlauf der Schleife mit entsprechendem Zugriff auf E_2 wird der Ladebaum an die Elemente E_2 bis E_5 als Wurzelobjekte gebunden und damit die Attributwerte A_2 bis A_5 aus der Datenbank nachgeladen. Im letzten Schritt müssen diese Werte den einzelnen Elementen zugewiesen werden.

Abbildung 6.5: Beispiel für einen Iterationszugriff



Im Fall, dass Verzweigungen im Schleifenkörper vorgenommen werden, kann es sein, dass bestimmte Codeteile des Schleifenrumpfs erst in späteren Schleifendurchläufen prozessiert werden. Die in diesen Codeteilen vorgenommenen Nachladevorgänge werden vom beschriebenen Verfahren nicht erfasst und deshalb bei späteren Durchläufen einzeln durchgeführt. Dieser Problematik kann begegnet werden, indem inkrementell zu Beginn jeder neuen Iteration überprüft wird, ob während des letzten Durchlaufs neue Nachladevorgänge stattgefunden haben. Diese müssen dann vor Beginn des nächsten Durchlaufs für alle restlichen Elemente durchgeführt werden.

6.7 Umsetzung von Ladestrategien im Prototypen

Die bisherigen Untersuchungen von Ladestrategien für OR-Mapper sollen als Grundlage genommen werden, um die Implementierung dieser Strategien zu beschreiben. Bezüglich der Verwendung von Ladestrategien zielt die Umsetzung der Konzepte innerhalb des Prototypen dahin, einerseits die explizite Angabe von Ladestrategien für Anfragen an den Mapper zu ermöglichen. Andererseits sollte eine Nutzung des Mappers durch den Entwickler auch ohne intensive Beschäftigung mit Ladestrategien möglich sein, weshalb Ladestrategien so einfach wie möglich formulierbar sein sollten.

Zunächst stellt sich die Frage, mit welcher Granularität bzw. mit welchem Gültigkeitsbereich Ladestrategien definiert und verwaltet werden sollten. Dafür gibt es drei Möglichkeiten:

- eine Ladestrategie pro Instanz

- eine Ladestrategie pro Klasse
- eine Ladestrategie pro Anfrage / Ladevorgang

Eine Ladestrategie pro Instanz weist zwar die feinste Granularität auf, andererseits sind hierbei sowohl der Verwaltungsaufwand als auch die Speicherbenutzung hoch. Zudem geht ein direkt von der Anwendung ausgelöster Ladevorgang selten von einer einzelnen Instanz aus, nämlich nur bei expliziten Punktanfragen durch den Domänenentwickler nach einem Objekt mit einer bestimmten Id. Aus diesem Grund wird beim Prototypen eine Standard-Ladestrategie für jede persistente Klasse verwaltet. Für einzelne Anfragen kann statt dieser Standard-Ladestrategie eine explizit vom Entwickler formulierte Ladestrategie benutzt werden.
Für die Modellierung von Ladestrategien wurde die Klasse `LoadStrategy` entworfen (siehe Listing 6.1). Diese enthält Angaben zum Kaskadierungslevel und zur Standard-Ladestrategie für Collection-Attribute der Klasse. Zusätzlich ist eine Liste mit Ladestrategien für einzelne Collection-Attribute enthalten, mit welchen diese Standard-Ladestrategie überschrieben werden kann. Dies ist deshalb sinnvoll, weil die Anzahl von Elementen in einer Collection sich stark unter den einzelnen Collection-Attributen einer persistenten Klasse unterscheiden kann. Eine datenintensive Collection-Ladestrategie wie `LoadComplete` kann für sehr große Collections ungeeignet sein.

```
public class LoadStrategy {
       public int cascadingLevel;
       public defaultCollectionLoadStrategy CollectionLoadStrategy;
       public collectionLoadStrategy IDictionary<string, CollectionLoadStrategy>;
       ...
}
```

Listing 6.1: Modellierung von Ladestrategien in der Klasse LoadStrategy

Beim Laden eines Objekts wird bei Collection-Attributen überprüft, ob eine individuelle Strategie vorhanden ist, andernfalls wird die für die Anwendung festgelegte Standard-Collectionstrategie benutzt.

6.7.1 Standard-Ladestrategien für persistente Klassen

Die Standard-Ladestrategien können vom OR-Mapper im Zuge der Analyse der persistenten Klassen durch den Type-Analyzer formuliert werden. In dieser Phase (vgl. Abschnitt 5.1) analysiert der Mapper alle zu persistierenden Attribute und legt entsprechende Metadaten über die Klasse in einem zentralen Repository an. Der

Prototyp verwaltet diese Metadaten persistenter Properties in Form von Listen, jeweils eine für Primitivattribute, persistente Referenzen sowie Collection-Attribute. Damit kann schnell der Vernetzungsgrad einer persistenten Klasse mit anderen Klassen ermittelt werden und regelbasiert eine maximale Kaskadierung für die Standard-Ladestrategie bestimmt werden.
Als Standard-Ladestrategie für persistente Collection-Properties einer Klasse wird das Laden der Element-Ids gesetzt (Ladestrategie `LoadIds`). Innerhalb der Definition des Domäneninterfaces kann dieser Wert für eine Collection mittels eines Custom-Attributes überschrieben werden. Sinnvoll ist dies für sehr große Collections, auf die selten zugegriffen wird (Ladestrategie `LoadNothing`) sowie für kleine Collections, auf deren einzelne Elemente mit hoher Wahrscheinlichkeit zugegriffen wird (Ladestrategie `LoadComplete`). Ein solcher im Custom-Attribute definierter Wert wird vom Mapper in der Standard-Ladestrategie für die Klasse übernommen, deren Attribut die Collection ist.

```
public interface IOrder {
       [PersistentCollection(typeof(IProduct), typeof(Product),
       Cardinality.OneToMany,ListOccurences.Unique,
       CollectionLoadStrategy.LoadComplete)]
       IList items { get; }
       ...
}
```

Listing 6.2: Beispiel eines persistenten Interfaces mit Collection-Attribut

Primitivattribute, die mit dem Custom-Attribute `LargePrimitive` gekennzeichnet sind, können durch dessen Parametrisierung entweder frühzeitig oder erst beim Lesezugriff geladen werden. Andere Primitivattribute und die Werte persistenter Referenzen werden immer frühzeitig geladen, hier ergibt sich wenig Optimierungspotential für Ladestrategien (vgl. Abschnitte 6.1 und 6.2).

6.7.2 Aufgabenverteilung der OR-Komponenten bei Ladevorgängen

Gemäß des Paradigmas "Separation of concerns" (siehe [SOC06]) sollte bei der Umsetzung von Ladestrategien der hierfür entwickelte Code so wenig wie möglich über verschiedene Bereiche des Mappers verteilt werden. Aus diesem Grund wurden zwei eigenständige Komponenten entwickelt: der Loader sowie der Result-Mapper.
Die Umsetzung von Ladestrategien wird im Loader gekapselt. Jeder Session wird bei ihrer Initialisierung ein Loader zugeordnet. Alle Ladevorgänge innerhalb dieser

Session werden von diesem Loader durchgeführt. Er benutzt zum Laden die Datenbankzugriffskomponente und reicht die von ihr erhaltenen Daten an den Result-Mapper weiter, welcher die eigentliche Abbildung der Datentupel auf die Properties der persistenten Objekte vornimmt.
Abbildung 6.6 stellt die Architektur derjenigen Komponenten des Mappers dar, welche relevant für Ladevorgänge sind.

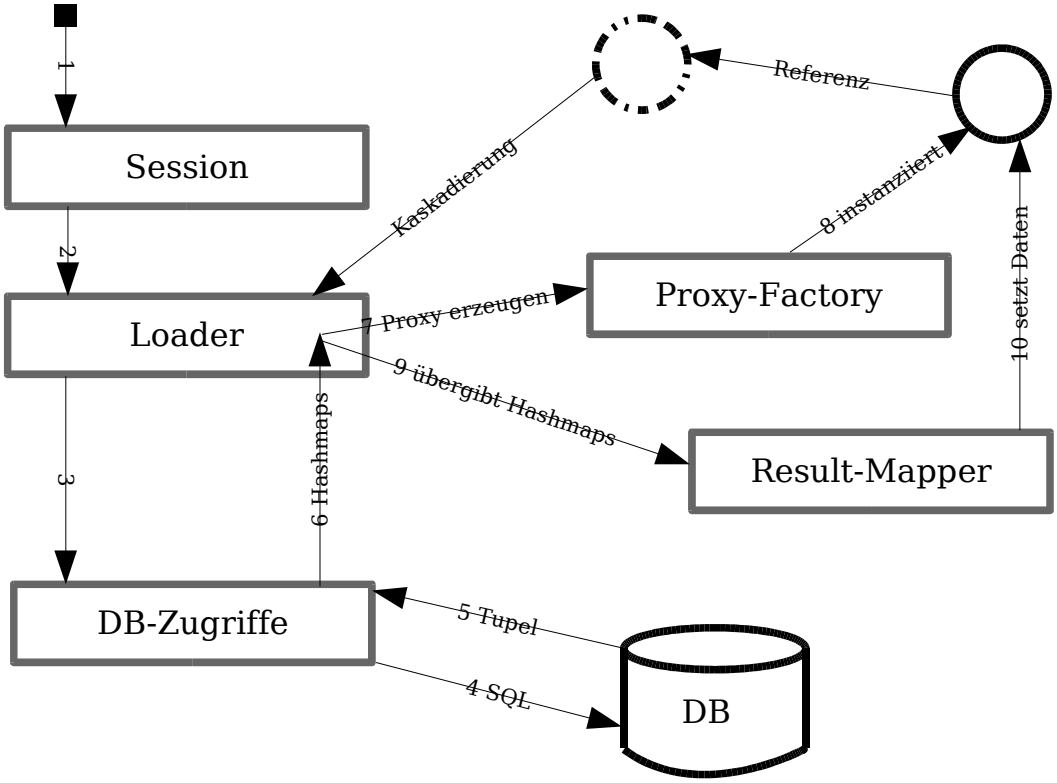


Abbildung 6.6: Ladevorgang beim Prototypen

Die Zusammenarbeit der Komponenten untereinander soll anhand einer explizit in einer Anwendung ausgelösten Punktanfrage erläutert werden. Eine solche Anfrage besteht aus dem Namen einer Klasse K und einer bestimmten Id.
Die Anfrage wird im ersten Schritt innerhalb der Anwendung an das Anfrage-API der benutzten Session gestellt **(1)** und von dieser direkt an den Loader weitergeleitet **(2)**, da eine Punktanfrage keine Prädikate enthält und deshalb kein Parsing benötigt wird. Falls vom Entwickler keine anfragespezifische Ladestrategie angegeben wurde, ermittelt der Loader die Standard-Ladestrategie der abgefragten Klasse. Diese kann aus der Proxy-Registry abgefragt werden.
Der Loader nutzt die Dienste der Datenbankzugriffskomponente **(3)**, deren API das Auslesen von einzelnen sowie einer Menge von Tupeln umfasst. Die Datenbankzugriffsschicht ermittelt die benötigten Metadaten wie den Tabellen- und die Feldernamen aus der Proxy-Registry und formuliert daraus eine SQL-Anfrage an

die Datenbank **(4)**. Die Daten werden in Form einer oder mehrerer Tupel **(5)** ausgelesen und in einer RDBMS-unabhängigen Form strukturiert. Somit werden die RDBMS-spezifischen Teile des Codes auf die Datenbankzugriffskomponente begrenzt. Jedes Tupel des Anfrageergebnisses wird in einer Hashmap gespeichert. Liefert die Datenbank mehrere Ergebnistupel zurück, dann wird eine Liste solcher Hashmaps gebildet. In dieser Form werden die Daten an den Loader zurückgeliefert **(6)**. Im nächsten Schritt fordert der Loader eine entsprechende Anzahl von Proxies der abgefragten Klasse **(7)** bei der Proxy-Factory an, welche die Proxies instanziiert **(8)**. Diese Proxies übergibt der Loader gemeinsam mit den zugehörigen Daten-Hashmaps an den Result-Mapper **(9)**.
Der Result-Mapper nimmt die eigentliche Abbildung der Daten aus den Hashmaps auf die Proxyobjekte vor **(10)**. Da in der Hashmap alle Feldnamen als Schlüsselwerte enthalten sind, muss er lediglich den Feldtypen ermitteln, also ob es sich bei einem Feld um ein Primitivattribut oder eine persistente Referenz handelt. Primitivattribute können bei den Proxies umgehend mit der Methode `IProxy.setProp(string name, object value)` gesetzt werden. Bei persistenten Referenzen muss der entsprechende Typ des Propertys aus den Metadaten ermittelt werden, da er im nächsten Schritt für die Proxyerzeugung benötigt wird. Anschließend wird ein entsprechender Proxy bei der Proxy-Factory angefordert und neu erzeugt. Sein Status wird auf `ProxyLoadState.Unloaded` gesetzt und die Referenz zugewiesen, damit ist die Verknüpfung zwischen den beiden Proxies abgeschlossen. Beendet wird der Abbildungsvorgang mit dem Setzen des Status des Proxys auf `ProxyLoadState.Loaded`. Eine Liste der neu erzeugten Proxies wird an den Loader zurückgeliefert, um nach dem Laden der Collections noch gegebenenfalls den Ladevorgang zu kaskadieren.
Nach dem Laden von Primitivattributen und persistenten Referenzen müssen noch die Collection-Attribute des Proxies geladen werden. Zunächst wird anhand der Ladestrategie überprüft, ob für die zu ladende Collection eine spezielle Collection-Ladestrategie benutzt werden soll. Falls hierfür innerhalb der Ladestrategie keine individuelle Strategie abgelegt wurde, wird der Wert von `defaultCollectionLoadStrategy` benutzt. Die Liste wird gemeinsam mit der Collection-Ladestrategie an private Methoden des Loaders delegiert. Diese wiederum nutzen die Datenbankzugriffskomponente, um je nach Collection-Ladestrategie entweder Listen von Tupeln (`LoadComplete`) oder eine Liste der Ids (`LoadIds`) der Collection-Elemente zu erhalten. Für jedes Element der Collection wird ein Proxy erzeugt. Während bei `LoadComplete` nur noch die Ids und die Proxies in der Liste gesetzt werden, muss bei `LoadComplete` die Liste der Proxies

mit den aus der Datenbank gelesenen Daten an den Result-Mapper übergeben werden. Der Result-Mapper bildet die Daten aus den Hashmaps auf die einzelnen Proxies ab. Abschließend werden Ids und Proxies als Elemente der Liste gesetzt.

Während des Ladens von Referenzen und Collections kann es sein, dass vom Loader neue Proxyobjekte erzeugt wurden. Der Loader entscheidet anhand der Ladestrategie, ob deren Daten nachgeladen werden. Dies ist dann der Fall, wenn die Kaskadierung der Ladestrategie größer als null ist. Der Loader erzeugt dann eine Kopie der für die Anfrage benutzten Ladestrategie mit einer um eins herabgesetzten Kaskadierung und stößt für die ungeladenen Proxyobjekte einen neuen Ladevorgang an. Dies wird solange fortgesetzt, bis die Kaskadierung bei null liegt. Die während des letzten Kaskadierungsschritts geladenen persistenten Referenzen werden noch als Proxyobjekte instanziiert, ihre Properties jedoch nicht mehr aus der Datenbank gelesen. Der Status dieser Proxies bleibt entsprechend auf `ProxyLoadState.Unloaded` gesetzt.

6.7.3 Umsetzung von Ladekontexten bei Collections

In Abschnitt 6.5 wurde die Berücksichtigung von Kontexten für die effiziente Durchführung von Nachladeoperationen bei persistenten Collections entworfen. Die Umsetzung der Loader-Komponente des Prototypen erlaubt für solche Nachladevorgänge die Berücksichtigung des gegenwärtigen Kontextes, in dem sich die betreffende Collection befindet. Sowohl bei Navigations- als auch bei Iterationszugriffen ist dabei das Ziel, die durch Zugriffe auf die Collection ausgelösten Nachladevorgänge zu analysieren und mithilfe von Ladebäumen bei wiederholten Zugriffen gebündelt ausführen zu können, um damit für den jeweiligen Einsatzkontext optimale Ladestrategien zu erreichen.
Um dies zu erreichen, wurde das Konzept des Ladebaums implementiert. Dabei ist ein Ladebaum eine gewöhnliche Baumstruktur, bei der zwei Knotentypen unterschieden werden. Diese speichern unterschiedliche Daten zu den einzelnen Nachladevorgängen, welche sie repräsentieren:

- Feldlade-Knoten stellen das Nachladen eines großen Primitivattributs dar: dabei wird gespeichert, um welches Primitivattribut es sich handelt (Name des persistenten Propertys) und welcher Proxy den Nachladevorgang auslöst.
- Proxylade-Knoten stehen für das Nachladen eines Objekts: dabei wird gespeichert, welcher Proxy nachgeladen wird (Klasse sowie die Id) und welcher Proxy (Id) diesen Nachladevorgang auslöst (Name der persistenten Referenz,

welche auf den nachzuladenden Proxy verweist).

Durch die Speicherung der Ids sowohl des den Ladevorgang auslösenden als auch des zu ladenden Proxys kann aus den einzelnen Nachladevorgängen ein Ladebaum konstruiert werden. Dazu werden diejenigen Nachladevorgänge als Knoten miteinander verknüpft, deren zugehörige Proxyobjekte auch im Objektgraphen durch persistente Referenzen verknüpft sind.
Bei der Implementierung des Prototypen verwaltet jeder Loader ein `LoadContext`-Objekt. Dieses enthält eine Liste der vorgenommenen Ladevorgänge, sowohl explizite als auch implizite (also Nachladevorgänge, die durch Zugriffe auf ungeladene Properties ausgelöst werden). Zusätzlich wird in der Liste gespeichert, wenn über eine persistente Collection iteriert wird und wenn diese Iteration beendet ist. Zu jedem Beginn einer Iteration wird der Hashwert der Collection gespeichert. Der Ladekontext einer Collection, über die im Code iteriert wird, bildet sich damit aus genau den Ladeoperationen der Liste, die mit den Zugriffen auf das erste Proxy-Element der Collection verknüpft werden können. Aus diesen Elementen kann ein Ladebaum für die Iteration konstruiert werden.
Abbildung 6.7 zeigt exemplarisch diese Transformation, links befindet sich die Liste der Ladeoperationen, rechts der entsprechende Ladebaum. Ein Pfeil zwischen Objekten entspricht einer persistenten Referenz, seine Beschriftung ist der Name des entsprechenden Propertys.

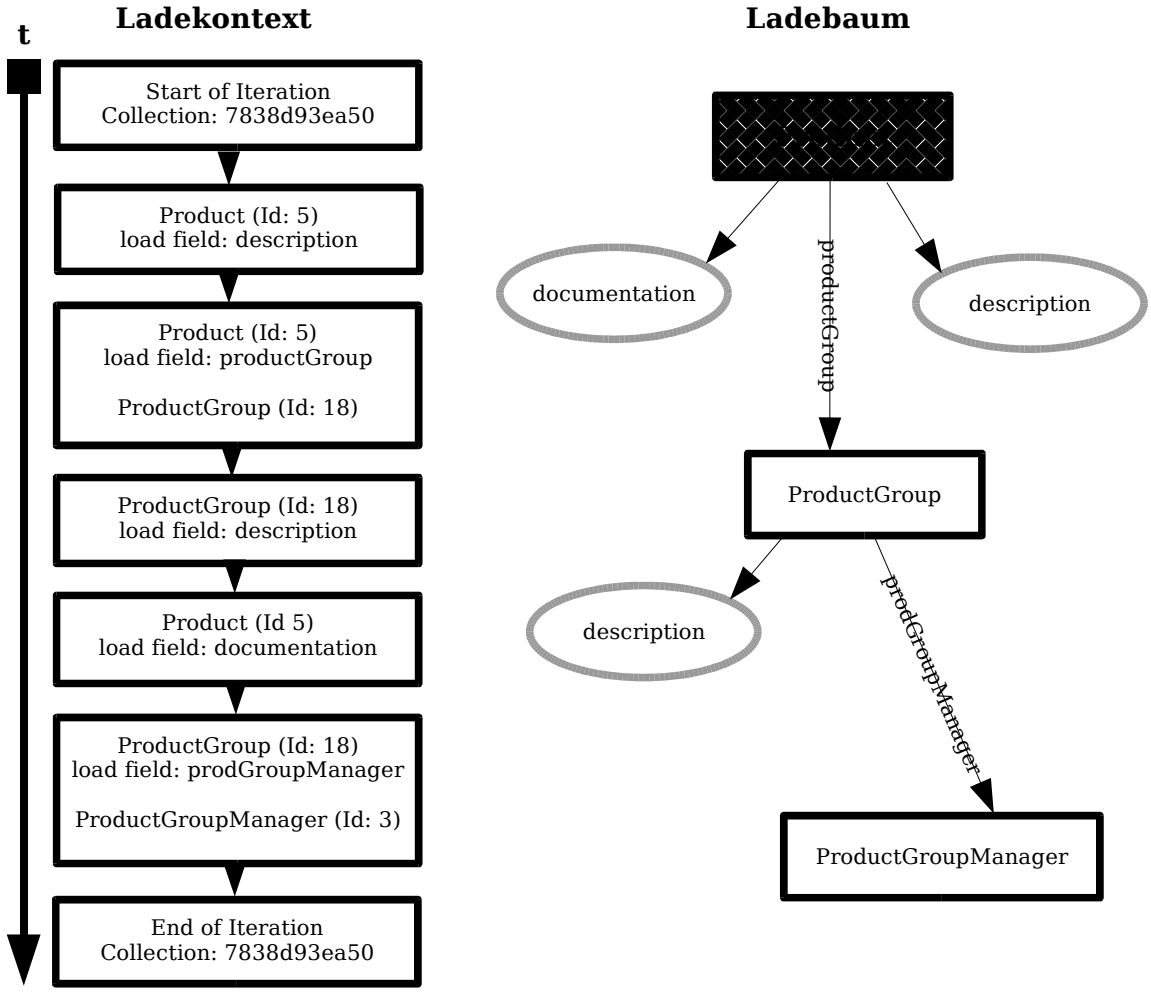


Abbildung 6.7: Konstruktion eines Ladebaums aus einem Ladekontext

Eine besondere Rolle nimmt dabei das Wurzelobjekt (in Abbildung 6.7 schraffiert dargestellt) ein. Im abgebildeten Beispiel steht es für die Bindung des Ladebaums an die Instanz der Klasse `Product` mit der Id 5. Ohne konkrete Bindung an eine Instanz ist der Ladebaum abstrakt und kann prinzipiell an jede Instanz der Klasse `Product` gebunden werden.

Da auch die Klassennamen der zu ladenden Proxies sowie die Namen der Properties, welche die Proxies miteinander verknüpfen, in den Knoten des Ladebaums gespeichert werden, ist es möglich, aus einem Ladebaum eine SQL-Abfrage zu konstruieren. Die SQL-Abfrage ist dabei zunächst unabhängig von der Id des Wurzelobjekts, da sie nur die auszulesenden Felder aller zu ladenden Objekte sowie die Verknüpfungen zwischen den Objekten durch persistente Referenzen enthält, ohne in der where-Klausel die Id des Wurzelproxys zu enthalten. Die Verknüpfungen werden in der Abfrage auf Left-Outer-Joins abgebildet. Es müssen Outer-Joins benutzt werden, weil eine Referenz auch nicht gesetzt und damit null sein kann. Durch die Verwendung von Outer-Joins liefert die SQL-Anfrage in jedem

Fall ein Tupel zurück. Die Felder nicht gesetzter Referenzen sind bei diesem Tupel alle Null. Die entstandene Abfrage kann später mit der Id eines Proxys parametrisiert werden, dieser nimmt dann die Rolle des Wurzelproxys ein. Das Abfrageergebnis liefert genau die Daten zurück, welche nach Vorgabe des Ladebaums nachgeladen werden müssen. Der Ladebaum ist hierbei ein abstraktes Zugriffsmuster, welches an ein Wurzelobjekt gebunden werden kann. Die Bindung eines Ladebaums an ein Wurzelobjekt entspricht der Parametrisierung der SQL-Abfrage mit der Id des Wurzelobjekts in der where-Klausel. Die entstandene SQL-Abfrage kann auch mit mehreren Ids parametrisiert werden, um die Werte der nachzuladenden Objekte für eine ganze Reihe von Wurzelobjekten zu erhalten. Für eine solche Bindung an mehrere Objekte kann der in-Operator von SQL in der where-Klausel benutzt werden. Bei einer Bindung an mehrere Wurzelobjekte liefert die Abfrage für jedes Wurzelobjekt ein Tupel zurück, welches die Daten aller nachzuladenden Objekte des gebundenen Ladebaums enthält.

Die Nutzung von Ladekontexten mithilfe von Ladebäumen wurde beim Prototypen wie folgt umgesetzt: Wenn über eine persistente Collection mittels einer `foreach`-Schleife iteriert wird, bildet der während des ersten Iterationsdurchlaufs entstandene Ladebaum den Kontext dieser Collection für alle folgenden Durchläufe. Vor dem Durchlaufen der zweiten Iteration wird die Liste der Nachladevorgänge seit dem Beginn der Iteration benutzt, um daraus einen Ladebaum zu konstruieren. Zu diesem Zweck wird in der Liste der Ladevorgänge des aktuellen Ladekontextes nach dem Beginn der Iteration der jeweiligen Collection gesucht und alle folgenden Nachladeoperationen, die vom ersten Element der Collection ausgelöst werden, in den Nachladebaum als Kindknoten eingehängt. Alle hinzugefügten Kindknoten werden während dieser Analysephase in einer Liste gespeichert, um überprüfen zu können, ob eine Nachladeoperation dem Kontext der gegenwärtig durchlaufenen Collection zugeordnet werden kann. Da jedes Nachladen des Grads n ein vorheriges Nachladen des Grads n-1 erfordert (also das Nachladen des Elternknotens zeitlich vor dem Nachladen des Kindknotens erfolgen muss), ergibt sich für den Algorithmus zur Konstruktion des Ladebaums eine Laufzeit von O(n) mit n als der Anzahl der Listenelemente im Ladekontext der benutzten Session. Der Pseudocode des Algorithmus sieht folgendermaßen aus:

```
Dictionary<string,LoadItem> allChildNodesById = new Dictionary<string,LoadItem>();
foreach (LoadItem loadItem in LoadContext.itemList) {
	if (loadItem.TriggerId == idOfFirstCollectionElement) {
		// direct childnode of root-proxy
		allChildNodesById[loadItem.Id] = loadItem;
	}
	else if (allChildNodesById.ContainsKey(loadItem.TriggerId)) {
		// indirect childnode
		allChildNodesById[loadItem.TriggerId].childNodes.Add(loadItem);
		allChildNodesById[loadItem.Id] = loadItem;
	}
	else {
		// no item of the current loadcontext, so skip it
		continue;
	}
}
```

Listing 6.3: Algorithmus zur Konstruktion des Ladebaums

Nach der Konstruktion des Ladebaums wird die äquivalente SQL-Abfrage erstellt. Hierzu wird der Baum von einer rekursiven Funktion durchlaufen. Jeder Proxylade-Knoten erhält einen Tabellenalias, der sich aus dem Klassennamen, der Entfernung vom Wurzelproxy und dem Propertynamen zusammensetzt und damit für die SQL-Abfrage eindeutig ist. Diese Eindeutigkeit ist notwendig, da die einzelnen Elemente des Ergebnistupels der Abfrage später den einzelnen Proxies zugeordnet werden müssen. Zusätzlich zu den weiter oben angebenen Informationen wird in jedem Knoten des Ladebaums seine Entfernung vom Wurzelproxy gespeichert, um den Tabellenalias direkt aus den im einzelnen Knoten gespeicherten Informationen heraus konstruieren zu können. Neben der durch Left-Outer-Joins verknüpften Tabellen und deren Aliase wird bei der Traversierung des Ladebaums eine Liste der auszulesenden Felder für jede Tabelle erstellt. Diese Felder müssen ebenfalls mit einem eindeutigen Alias versehen werden, um zugeordnet werden zu können. Ein Feld einer Tabelle wird dann hinzugefügt, wenn für das entsprechende Property entweder eager-load als Ladestrategie vorgesehen ist oder wenn das Feld mittels lazy-load zu einem späteren Zeitpunkt nachgeladen wurde.

Zuletzt wird die SQL-Abfrage mit den Ids aller Collection-Elemente mit Ausnahme des Ersten parametrisiert. Damit wird der aus den Zugriffen über das erste Collection-Element entstandene abstrakte Ladebaum an die Gruppe der restlichen Elemente als Wurzelobjekte gebunden.

Das Ergebnis der SQL-Abfrage besteht aus einem Tupel pro Element der Collection, jedes Tupel enthält die Attribute aller nachgeladenen Proxies mit dem jeweiligen

Element der Collection als Wurzelknoten. Jedes Tupel wird in einer Hashmap gespeichert. Die Liste aller Hashmaps wird dann dem Result-Mapper übergeben, der den Objektgraphen konstruiert und die Werte der Hashmap auf die einzelnen Properties der Objekte abbildet. Der Result-Mapper sortiert zunächst die einzelnen Schlüssel-Wert-Paare einer Hashmap nach ihrer Zugehörigkeit zu einem nachgeladenen Proxy. Diese Zuordnung wird über die Feldaliase als Schlüssel vorgenommen. Anschließend werden bei der Proxy-Factory neue Proxies angefordert und die einzelnen Propertys gesetzt. Zuletzt werden die Proxies gemäß des Ladebaums miteinander verknüpft und der entstandene, nachgeladene Objektbaum beim Wurzelproxy eingehängt. Dieser Vorgang wird für jede Hashmap wiederholt. Anschließen wird die Ablaufkontrolle an die `PersistentList` zurückgegeben und diese kann mit dem zweiten Schleifendurchlauf fortfahren.

7 Integration des Mappers in die Plugin-Umgebung

Durch die Fähigkeit, für persistente Klassen Proxycode zur Laufzeit zu erzeugen, kann der Prototyp innerhalb einer Plugin-Umgebung benutzt werden, welche die Einbindung neuer Plugins zur Laufzeit der Anwendung vorsieht. Die Möglichkeit dynamischer Ladestrategien unterstützt den Entwickler von Plugins dabei, um vom Einsatzkontext des Plugins abhängig angemessene Performanz zu erreichen sowie den Speicherverbrauch so gering wie möglich zu halten.
Die Plugin-Umgebung, innerhalb derer der Mapper eingesetzt werden soll, orientiert sich in ihrer Softwarearchitektur stark am Eclipse-Framework (vgl. [ECL06]), basiert jedoch technologisch auf der DOT.NET-Plattform. Zum Zeitpunkt dieser Arbeit befindet sich diese Plugin-Umgebung noch in der Entwicklung, weshalb die Integration des Prototypen lediglich skizzenhaft entworfen werden kann. Im Folgenden wird für die Plugin-Umgebung der Name SoNET verwendet.

7.1 Architektur der Plugin-Umgebung

SoNET besteht ähnlich wie Eclipse aus einem minimal gehaltenen Softwarekern, dem **Kernel**. Der Kernel basiert auf dem Windsor-Container (siehe [WIN06]), welcher das Prinzip "Inversion of Control" (vgl. [FOW04]) zur Verknüpfung einzelner Komponenten umsetzt. Wichtig ist, dass die Komponenten untereinander ausschließlich über Interfaces benutzt werden müssen, da deren Implementierungen in der Konfiguration des Containers festgelegt werden und somit zur Entwicklungszeit nicht bekannt sind. Die Abhängigkeiten von Komponenten untereinander wird ebenfalls in der Konfiguration festgelegt. Beim Start überprüft der Windsor-Container diese Abhängigkeiten und meldet bei entsprechenden Inkonsistenzen Fehler. Nach dem Start des Windsor-Containers können die Komponenten für die Benutzung beim Container angefordert werden. Er initialisiert dann je nach Konfiguration die Komponente als Singleton (siehe entsprechendes Entwurfsmuster bei [GHJV95] S. 127ff) oder als eine Instanz pro Anforderung und liefert diese zurück. Durch diese Vorgehensweise bleibt die Kopplung der Komponenten lose (zu den Vorteilen hierzu siehe [FOW04]).

Für den Kernel von SoNET können sogenannte Launchables festgelegt werden. Eine Launchable ist vergleichbar mit einer eigenständigen Anwendung, mit eigenen Modulen und Konfigurationen. Der Kernel überprüft beim Starten der Plattform das

Vorhandensein von Launchables und startet dann eine ausgewählte. Die eigentliche Anwendung läuft dann innerhalb dieser Launchable ab, beispielsweise in Form einer Event-Schleife bei GUI-Anwendungen.
An den Kernel können Plugins angedockt werden (siehe Abbildung 7.1). Unter SoNET, welches sich an der OSGI-Terminologie orientiert (siehe [OSG06]), heißen diese Plugins **Bundles**, die Begriffe werden im folgenden synonym verwendet. Plugins können sowohl interne Hilfsfunktionen als auch die eigentliche Funktionalität einer Anwendung für den Benutzer erfüllen. Der Kernel selbst enthält lediglich grundlegende, technische Funktionen wie die Verwaltung der Konfiguration des Gesamtsystems und das Laden der Plugins.

Jede weitergehende Funktionalität ist selbst als Plugin implementiert. Ein Beispiel hierfür ist die Plugin-Registry, in der alle installierten Plugins eingetragen sind und aus der Informationen über Plugins abgerufen werden können. Plugins können andere Plugins benutzen und somit eigene Abhängigkeiten definieren. Ein Plugin kann nur geladen werden, wenn die von ihm benutzten Plugins in der richtigen Version vorhanden sind und geladen werden konnten.
Plugins können zur Laufzeit geladen, angedockt und ihre Funktionalität aktiviert werden. Ebenso ist eine Deaktivierung zur Laufzeit möglich. Gegenüber einer monolithischen Softwarearchitektur ist bei SoNET dadurch ein hohes Maß an Dynamik zur Laufzeit möglich.

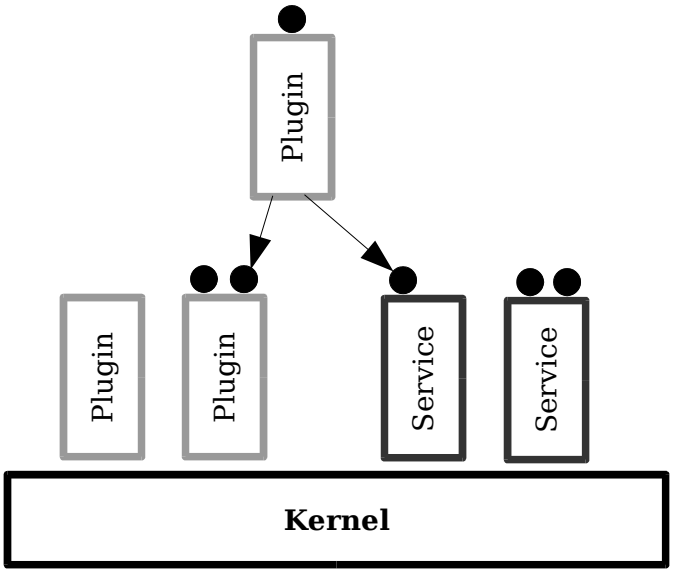


Abbildung 7.1: Architektur der SoNET-Plattform

Ein Plugin bzw. Bundle enthält Extensions und Extension-Points. Extension Points stellen dabei die Schnittstellen dar, welche Plugins für andere Plugins anbieten.

Innerhalb der Konfigurationsdatei eines Plugins wird für jeden Extension Point eine Id, ein Interface und eine implementierende Klasse angegeben. Das Interface enthält das API des Extension-Points, welches von Plugins, die sich an dieser Stelle andocken möchten, genutzt werden kann. Im weiteren wird ein Plugin, welches einen Extension-Point anbietet, als Anbieterplugin bezeichnet. Ein Plugin, das diesen Extension-Point nutzt, wird entsprechend Nachfragerplugin genannt.

Unter den verfügbaren Plugins bilden die sogenannten **Services** eine spezielle Gruppe. In Services wird Funktionalität implementiert, die unabhängig von einer konkreten Anwendungsdomäne ist und daher den Charakter eines Hilfsdienstes hat. Diese Dienste werden von den Plugins genutzt, welche die Anwendungsdomäne implementieren. Das entscheidende Kriterium für Dienste ist, dass sie in ihrer Funktionalität hinreichend abstrakt und technisch orientiert sind. Sie sollen dem Anwendungsentwickler häufig wiederkehrende Aufgaben erleichtern. Unter den Services bilden wiederum die **Predefined-Services** eine spezielle Gruppe. Predefined-Services stehen über die gesamte Laufzeit der SoNET-Plattform zur Verfügung. Die Predefined-Services müssen beim Starten der SoNET-Plattform in der Konfiguration der Launchable angegeben sein und werden am Anfang vom Container geladen und gestartet.
Einer der grundlegenden Dienste ist der Extension-Registry-Service. Dieser durchsucht nach seinem Start das Dateisystem nach installierten Plugins. Die ermittelte Menge der vorhandenen Plugins, deren Version und Konfiguration bilden die Gesamtkonfiguration des Systems. Diese wird mit der Gesamtkonfiguration des letzten Ablaufs von SoNET verglichen und daraus bei Veränderungen eine neue Konfigurationsdatei erstellt. Die entstandene Konfiguration wird von der Extension-Registry anderen Plugins über einen Extension-Point zur Verfügung gestellt. Über diesen können Informationen über einzelne Plugins und deren Konfiguration abgerufen werden.
Auf der Basis des Extension-Registry-Service arbeitet der Objectbuilder-Service, welcher das Prinzip "Inversion of Control" mittels des Windsor-Containers umsetzt. Er übernimmt die Instanziierung der Plugin-Klassen und die Verknüpfung der Plugins untereinander. Für den Konstruktor einer Nachfragerplugin-Klasse kann in der Konfiguration festgelegt werden, dass es Extension-Points eines Anbieterplugins nutzt. Die entsprechenden Instanzen der Anbieterplugins können vom Objectbuilder-Service als Parameter an den Konstruktor des Nachfragerplugins übergeben werden und anschließend in diesem benutzt werden. Dem Nachfragerplugin ist dabei nur das mit dem Extension-Point assoziierte Interface des Anbieterplugins bekannt. Dieses Interface stellt die Funktionalität

bereit, welche der Extension-Point nach außen reicht. Das Interface wird in der SoNET-Terminologie auch Contract genannt. Zum Begriff des Contracts und dem damit verbundenen Paradigma des "Design by Contract" findet sich mehr bei [PUG06] S. 18ff und [MEY97].
Weitere Predefined-Services sind der Cache-Service, der das Caching von serialisierbaren Objekten anbietet sowie der Error-Service, der den Umgang mit verschiedenen Arten von Exceptions übernimmt.

7.2 Integration des Mappers in die Plugin-Umgebung

Um den Prototypen für die Benutzung innerhalb der SoNET-Plattform vorzubereiten, muss zunächst entschieden werden, ob eine Umsetzung als Plugin, als normaler Service oder als Predefined-Service am sinnvollsten ist. Objektpersistenz ist nicht anwendungsspezifisch, sondern unabhängig von einzelnen Anwendungen und hat eine sehr allgemeine Charakteristik. In dieser Hinsicht kann Objektpersistenz als ein Aspekt (siehe hierzu [CLA05] S. 4ff) betrachtet werden, welcher unabhängig von Abstraktionsniveaus und Modellierungsfragen steht. Diese Gründe sprechen dafür, den Mapper als Service umzusetzen. Zudem sollte der Nutzer einer Anwendung nie direkt mit der Funktionalität des Mappers zu tun haben, was den Charakter des Mappers als technisch orientierter Dienst unterstreicht. Um den Mapper auch anderen Diensten und speziell anderen Predefined-Services zur Verfügung zu stellen, wird der Prototyp als Predefined-Service konzipiert. Mit dieser Entscheidung ist zudem sichergestellt, dass Objektpersistenz über die gesamte Laufzeit einer auf SoNET basierenden Anwendung benutzt werden kann und nicht damit gerechnet werden muss, dass der Dienst zur Laufzeit beendet werden kann.
Beim Entwurf eines solchen Persistenzdienstes gibt es zwei Möglichkeiten: entweder als Persistenzdienst für ein konkretes Speicherungssystem wie zum Beispiel ein bestimmtes RDBMS, oder als generischer Dienst, für den per Konfiguration festgelegt werden kann, welches Speicherungssystem benutzt werden soll. Die zweite Möglichkeit hat folgende Vorteile:

- Für die Einbindung und Nutzung neuer Speicherungssysteme muss kein neuer Dienst entwickelt werden. Zusätzlich wird die Anzahl an Predefined-Services nicht unnötig vergrößert, wenn neue Speicherungssysteme genutzt werden.
- Der Abhängigkeitsgraph der Plugins ändert sich nicht bei Nutzung eines neuen Speicherungssystems. Es ändert sich lediglich die Konfiguration des

Nachfragerplugins, welche den Persistenzdienst benutzen. Damit kann die Plattform bei Einbindung neuer Speicherungssysteme mit der Konfiguration des letzten Ablaufs benutzt werden und schneller starten.

Für den im Zuge dieser Arbeit entwickelten Prototypen wurde nur eine Implementation einer DB-Zugriffsschicht für ein RDBMS als Speicherungssystem realisiert. Seine Architektur erlaubt jedoch prinzipiell die Nutzung auch anderer Speicherungssysteme. Deshalb ist es sinnvoll, für die Integration in SoNET ein zusätzliches Interface `IPersister` zu entwickeln, welches den Persistenzdienst darstellt. Ein Plugin, welches Objektpersistenz nutzen möchte, kann per Konfiguration entscheiden, welche Implementierung von `IPersister` benutzt werden soll. Dadurch können auch andere Speicherungssysteme eingebunden werden. Beispiele hierfür sind Dateisysteme, OODBMS, XML-Datenbanken oder Verzeichnisdienste. Um sie einzubinden, muss eine Implementierung von `IStorageEngine` vorhanden sein, welche die Speicherung und das Auslesen der Daten für das Speicherungssystem vornimmt. Zusätzlich ist es sinnvoll, dass ein Standard-Speicherungssystem angegeben werden kann, welches benutzt wird, wenn kein spezifisches Speicherungssystem angegeben wird.
Je nach Persistenzdienst sind noch zusätzliche Konfigurationsangaben nötig. Für den Prototypen gehören hierzu:

- Angaben, welches RDBMS benutzt werden soll. Hiervon ist abhängig, welche Implementierung von `IStorageEngine` benutzt wird.
- Die Verbindungsdaten für den Zugang zum RDBMS.
- Die Implementierungen einzelner Komponenten des Mappers wie der Proxy-Factory und Proxy-Bakery. Werden keine speziellen Implementierungen angegeben, so werden die Standard-Implementierungen des Mappers benutzt.

Ein Nachfragerplugin, das sich an den Extension-Point `IPersister` ankoppelt, kann sich im Konstruktor eine `IPersister` implementierende Instanz als Parameter übergeben lassen. `IPersister` ist so einfach wie möglich gehalten und enthält eine mehrfach überladene Methode `createSessionFactory`. Durch diese lässt sich eine Session-Factory mit der Konfiguration erzeugen, die als Parameter übergeben wurde. Die Benutzung des Mappers mit den Klassen `SessionFactory` und `Session` ist unabhängig vom Speicherungssystem und einheitlich.
Eine exemplarische Nutzung persistenter Objekte innerhalb eines Plugins sieht aus wie folgt:

```
public class Nachfrageplugin(IPersister persister) {
       SessionFactory sf = persister.createSessionFactory(...);
       sf.registerType(typeof(IProject), typeof(Project));
       Session ses = sf.createSession();
       ses.startTransaction();
       IProject project1 = ses.createInstance<IProject>();
       // do some work with the project...
       ses.commitTransaction();
}
```

Listing 7.1: Beispiel für die Nutzung des Persistenz-Dienstes in einem Plugin

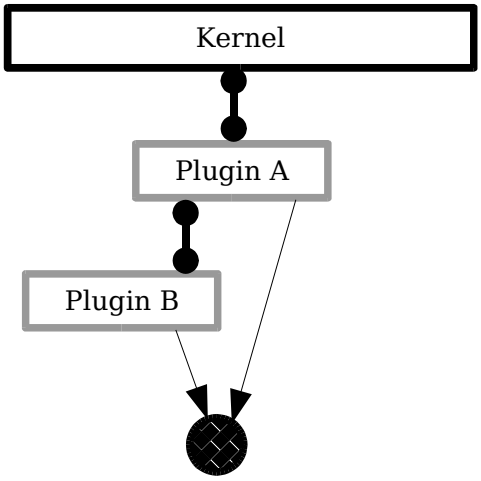
7.2.1 Nutzung persistenter Objekte durch mehrere Plugins

Bei der Nutzung persistenter Objekte lassen sich zwei Szenarien unterscheiden: der exklusive Zugriff auf Objekte durch ein Plugin (wie im obigen Beispiel) sowie deren Nutzung durch mehrere Plugins. Letzteres Szenario soll im folgenden auf mögliche Probleme und Konflikte hin untersucht werden.

Ein Nachfragerplugin des Persistenzdienstes kann auch als Anbieterplugin eigene Extension-Points anbieten und über diese persistente Objekte an andere Plugins übergeben. Dabei ist festzustellen, dass die bei dieser Konstellation entstehenden Zugriffe auf die persistenten Objekte nicht parallel sind. Da der Kontrollfluss und entsprechend die Möglichkeit, auf die betroffenen Objekte zuzugreifen, immer nur entweder beim Anbieter- oder beim Nachfragerplugin liegt, sind die Zugriffe sequentiell. Die Kontrolle über die Transaktion bleibt dabei beim Anbieterplugin, solange es die Session nicht ebenfalls an das Nachfragerplugin übergibt.
Die Zugriffe im beschriebenen Szenario sind allerdings dann nicht mehr sequentiell, wenn die Interaktion zwischen den Plugins asynchron erfolgt, zum Beispiel durch einen eigenen Thread des Nachfragerplugins. In diesem Fall sind echte Parallelzugriffe möglich und es stellen sich ähnliche Probleme wie im weiter unten beschriebenen Szenario.

Die zweite Möglichkeit ist die Benutzung von persistenten Objekten durch zwei Plugins, die keine Kenntnis davon haben, dass beide Plugins Objekte gemeinsam nutzen. Hier liegt echter Parallelzugriff vor. Abbildung 7.2 verdeutlicht beide Szenarien, der schraffierte Kreis steht hierbei für ein Objekt, auf welches zugegriffen wird und bei dem Zugriffskonflikte auftreten können.

Sequentielle Zugriffe:



Parallele Zugriffe:

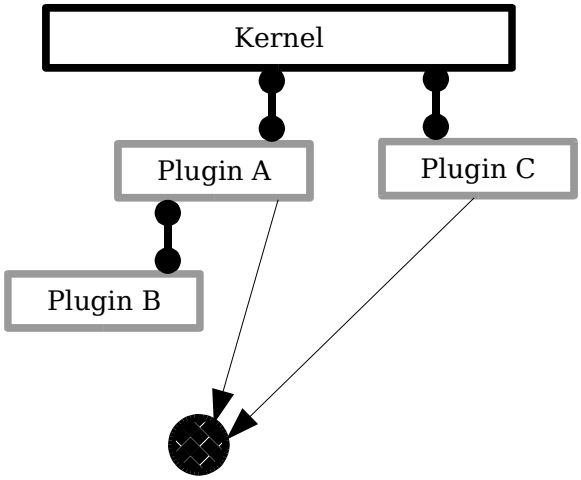


Abbildung 7.2: Objektzugriffe mehrerer Plugins

Der erste Fall wirft beim entwickelten Prototypen keine Probleme auf. Dabei startet das Anbieterplugin eine Session und eine Transaktion und übergibt gegebenenfalls die Objekte an das Nachfragerplugin. Wenn der Kontrollfluss wieder beim Anbieterplugin ist, kann die Transaktion beendet oder zurückgesetzt werden. Allerdings fehlt für das Anbieterplugin die Möglichkeit, die vor einer Übergabe an das Nachfragerplugin vorgenommenen Änderungen beizubehalten, wenn innerhalb des Nachfragerplugins eine Exception auftritt. In diesem Fall bleibt nur die Möglichkeit, die gesamte Transaktion zurückzusetzen. Für diese Problematik hat sich bei RDBMS das Konzept der Rücksetzpunkte (savepoints) innerhalb einer Transaktion als Lösungsansatz bewährt. Rücksetzpunkte speichern vorgenommene Änderungen innerhalb einer Transaktion. Die Transaktion kann bei einem Auftreten von nicht schwerwiegenden Fehlern auf einen bestimmten Rücksetzpunkt zurückgesetzt werden. Somit bleiben Teile der Verarbeitung erhalten. Näheres hierzu findet sich bei [HÄR01] S. 502f und [GRA93] S. 187ff. Rücksetzpunkte konnten im Prototypen aus Zeitgründen nicht umgesetzt werden. Dass die Umsetzung für OR-Mapper jedoch prinzipiell möglich ist, zeigt [DAT06].

Das Szenario mit parallelen Zugriffen, ohne dass die zugreifenden Plugins voneinander wissen, ist deutlich problematischer. Durch die zentral gehaltene Proxy-Factory wird zwar garantiert, dass zu einem Zeitpunkt nur eine Instanz des Proxyobjekts existieren kann. Allerdings kann dieses Objekt von verschiedenen Sessions gleichzeitig benutzt werden. Damit ist es möglich, dass sich überschneidende Zugriffe zu Problemen führen. Als Beispiel soll angenommen werden, dass sowohl eine Session S_A als auch eine Session S_B beide das Attribut A

des Objekt O ändern. Wenn nun die zu S_A und S_B gehörenden Transaktionen abgeschlossen werden sollen, muss der Schreibkonflikt aufgelöst werden, zum Beispiel, indem eine der beiden Transaktionen zurückgesetzt wird.

Für die Behandlung solcher Konflikte ist das Transaktionssystem des Mappers zuständig (siehe auch Abschnitt 4.1.4). Hierbei gibt es zwei prinzipielle Ansätze zur Umsetzung:

Delegation der Transaktionen an das RDBMS: Der Prototyp implementiert gegenwärtig kein eigenes Transaktionssystem, sondern nutzt die Transaktionen des zugrundeliegenden RDBMS. Folglich werden auftretende Zugriffs- und Versionskonflikte an dieses delegiert und müssen dort behandelt werden. Der Nachteil dieser Vorgehensweise ist, dass die Einbindung anderer Speicherungssysteme, wie sie in Abschnitt 7.2.2 behandelt wird, nicht ohne weiteres möglich ist. Der Mapper müsste hierfür die Transaktionen mehrerer Datenquellen koordinieren und für nicht-transaktionsfähige Speicherungssysteme wie zum Beispiel Dateisysteme Transaktionalität simulieren.

Eigenes einfaches Transaktionssystem: Für den Mapper könnte ein eigenes, einfach gehaltenes Transaktionssystem umgesetzt werden. Der zusätzliche Vorteil eines eigenen Transaktionssystems ist, dass die Transaktionen des Mappers auf dem RDBMS sehr kurz gehalten werden können.

Für eine Umsetzung eigener Transaktionen müsste ein Transaction-Manager entwickelt werden. Dieser wird zentral als Singleton beim Starten des Mappers angelegt. Eine Session-Factory meldet eine erzeugte Session beim Transaction-Manager an. Da die Sessions Referenzen auf alle geänderten Objekte halten, müssten die Änderungen an den Objekten mitprotokolliert werden. Wenn eine Session ihre laufende Transaktion abschließen will, wird dies beim Transaction-Manager angemeldet und dieser prüft die Änderungen auf Konflikte. Die bei der Implementierung von Transaktionssystemen anfallenden Problematiken sind beschrieben bei [HÄR01] S. 389ff oder sehr ausführlich bei [GRA93].

Auch bei einem einfachen eigenen Transaktionssystem des Mappers bleiben allerdings die oben beschriebene Probleme bei der gleichzeitigen Nutzung mehrerer Speicherungssysteme bestehen. Lösungsansätze hierzu werden im nächsten Abschnitt vorgestellt.

7.2.2 Gleichzeitige Nutzung mehrerer Speicherungssysteme

Zielsetzung bei der Einführung von `IPersister` ist, auch andere Speicherungssysteme in SoNET nutzbar zu machen. Nicht behandelt wurde bisher,

inwieweit der Prototyp den Einsatz mehrerer Speicherungssysteme gleichzeitig erlaubt.
Zunächst muss festgelegt werden, welche Komponenten des Mappers als einzelne, zentrale Instanz laufen können und welche pro Speicherungssystem verwaltet werden müssen. Da innerhalb einer Session eine Implementierung von `IStorageEngine` benutzt wird, die spezifisch auf das Speicherungssystem zugeschnitten ist, muss für jede Datenquelle eine eigene Session-Factory instanziiert werden. Die Session-Factory bekommt als Parameter die Konfiguration für das von ihr benutzte Speicherungssystem übergeben und kann die von einem Plugin bei ihr angeforderten Sessions korrekt initialisiert zurückliefern. Die Benutzung der Session-API zur Erzeugung und Abfrage von persistenten Objekten ist unabhängig von der Datenquelle einheitlich.
Die Umsetzungen der Proxy-Factory und der Proxy-Bakery sind unabhängig vom eingesetzten Speicherungssystem. Gleiches gilt für die Proxy-Registry. Deshalb können diese Komponenten zentral als Singletons gehalten werden. Für die Proxy-Factory muss dies auch so sein, da sie Referenzen auf die bereits erzeugten Proxies hält und bei einer Anforderung nur dann einen neuen Proxy erzeugt, falls noch kein Proxy mit dieser Id erzeugt wurde. Bei mehreren Instanzen der Proxy-Factory könnte es sonst zu mehreren Instanzen eines Proxys mit der gleichen Id kommen, worauf der Mapper nicht vorbereitet ist.

Bei der oben beschriebenen Konstruktion des Mappers ist die gleichzeitige Benutzung mehrerer Datenquellen möglich. Dabei existieren jedoch zwei wichtige Beschränkungen: Eine Session darf immer nur Objekte aus einer Datenquelle enthalten. Der Grund hierfür ist, dass eine Session bei ihrer Instanziierung durch die Session-Factory spezifisch für ihre Datenquelle konfiguriert wurde. Dies könnte zwar umgangen werden, indem jede einzelne persistente Klasse einem Speicherungssystem zugeordnet wird und dort persistiert wird. In diesem Fall können allerdings die Transaktionen der Session nicht mehr an ein Speicherungssystem delegiert werden. Somit müsste mit verteilten Transaktionen gearbeitet werden. Beim Abschließen der Session muss bei allen an der Session beteiligten Datenquellen mittels two-phase-commit ermittelt werden, ob diese die Transaktion beenden können. Falls dies nicht der Fall ist, wird die gesamte Transaktion abgebrochen. Näheres zum two-phase-commit und verteilten Transaktionen findet sich bei [BEL92] S. 177ff sowie [GRA93] S. 567ff.
Abbildung 7.2 zeigt einen Einsatz des Prototypen mit mehreren Datenquellen. Die kleinen Kreise stellen dabei einzelne Sessions dar. Die gestrichelte Linie zwischen zwei Sessions steht für die kritische Verbindung.

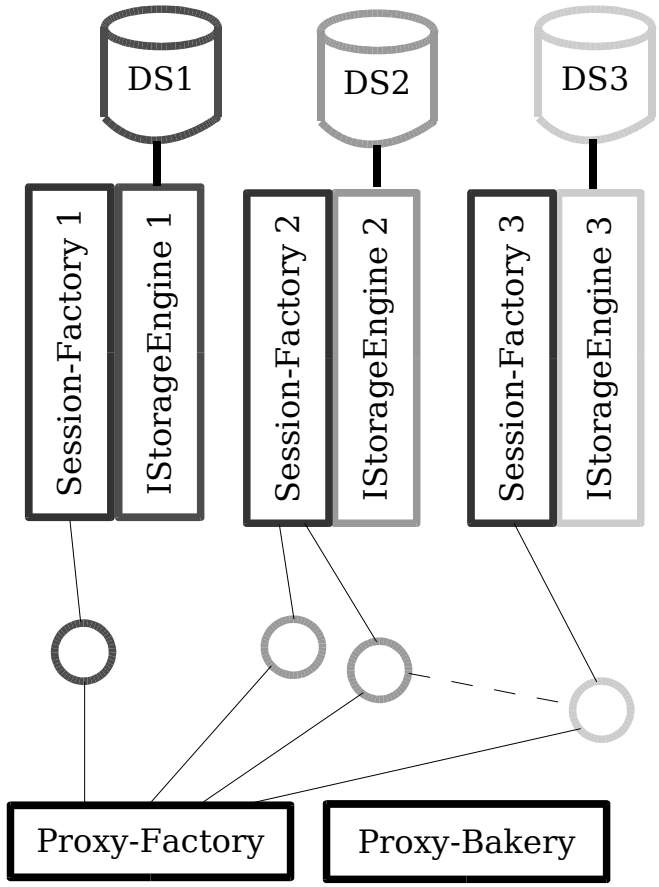


Abbildung 7.3: Einsatz des Mappers mit mehreren Datenquellen

Ein zweites Problem bei der Verwendung mehrerer Speicherungssysteme innerhalb einer Session ergibt sich daraus, dass bei der gegenwärtigen Umsetzung des Prototypen persistente Referenzen nicht angemessen behandelt werden können, wenn referenziertes und referenzierendes Objekt in unterschiedlichen Speicherungssystemen persistiert wurden. Zwar würden die Referenzen korrekt gespeichert, allerdings können die Joins in den vom Mapper erzeugten SQL-Abfragen beim Auslesen von Collections und bei der Umsetzung von Ladebäumen nicht mehr durchgeführt werden. Hierzu müsste also die gesamte Handhabung persistenter Referenzen durch den OR-Mapper erweitert werden und würde damit deutlich komplexer, als dies bisher der Fall ist. Somit kann pro Session nur eine Datenquelle benutzt werden. Zwar können mehrere Sessions von einem Plugin benutzt werden, allerdings müssen die mit den einzelnen Sessions assoziierten Objektmengen paarweise disjunkt sein.

Bei der Integration des Mappers zeigt sich, dass zwar die Architektur des Mappers angemessen ist und zur Nutzing in SoNET nur wenig angepasst werden musste. Allerdings zeigt sich auch, dass die Teile der Implementierung, welche aus

Zeitgründen noch nicht konsequent umgesetzt werden konnten, für eine sinnvolle Integration notwendig sind. Als Beispiel kann die Verwendung mehrerer Speicherungssysteme genommen werden, die somit einen Anknüpfungspunkt für weitere Arbeiten bietet.

8 Zusammenfassung und Ausblick

Ziel dieser Arbeit war die Entwicklung eines OR-Mapper-Prototypen, welcher sich in eine Plugin-Umgebung integrieren lässt. Die hierfür notwendige Flexibilität sollte durch Codeerzeugung von Proxyklassen zur Laufzeit sowie die Möglichkeit individueller Ladestrategien zur Laufzeit erreicht werden. Der umgesetzte Prototyp setzt diese beiden technischen Kriterien um.
Einige wichtige Bestandteile, über die ein OR-Mapper verfügen muss, konnten im Prototypen aus Zeitgründen nicht umgesetzt werden. Hier sind insbesondere die Mehrbenutzersynchronisation und eine OQL-ähnliche Abfragesprache für komplexere Anfragen an den Mapper zu erwähnen. Die bereits existierenden OR-Mapper beweisen zwar, dass diese Dinge prinzipiell umsetzbar sind. Es kann aber nicht ausgeschlossen werden, dass bei der Implementierung einige Komponenten des Prototypen geändert werden müssen.

Die zunächst beabsichtigte Erzeugung der Proxyklassen durch Emit, bei welcher direkter IL-Code in die Klassenrümpfe geschrieben wird, hat sich während der Entwicklung der Proxy-Bakery als zu aufwändig und intransparent herausgestellt. Ob sie einen nennenswerten Geschwindigkeitsvorteil gegenüber anderen Verfahren hat, ist fraglich und dürfte sich erst bei einer hohen Anzahl an zu erzeugenden Klassen herausstellen. Die Code-Erzeugung zur Laufzeit mittels Code-DOM hat sich hingegen als ein ausgesprochen einfach verwendbares Verfahren erwiesen. Mit seinen angebotenen Möglichkeiten ist es flexibel und leicht erweiterbar, um die erzeugten Proxys mit zusätzlichem Code auch an spezielle Bedürfnisse anzupassen.
Eine interessante Erweiterung des Ansatzes "Proxycodeerzeugung zur Laufzeit" bietet sich dadurch an, dass die Generierungsvorschrift für die Proxyklassen gegenwärtig in der Proxy-Bakery festgelegte sind. Für eine Änderung dieser Generierungsvorschrift muss deshalb eine neue Proxy-Bakery entwickelt werden. Es ist allerdings leicht vorstellbar, dass die Generierung durch Templates gesteuert wird. Templates sind leichter an spezielle Bedürfnisse und Anforderungen an die erzeugten Proxyklassen anpassbar. Da Code-DOM statt einzelner Syntax-Elemente auch ganze Codefragmente einbinden und kompilieren kann, sollte sich der Aufwand für eine solche Umsetzung in Grenzen halten. Zudem ist eine Template-basierte Generierung der Proxyklassen gut mit der Vorgehensweise der MDA und MDSD vereinbar.

Die Umsetzung von zur Laufzeit definierbarer Ladestrategien bildet den zweiten Schwerpunkt dieser Arbeit. Das Konzept der Ladebäume hat sich als gut geeignetes Mittel zur automatischen Analyse von Datenzugriffsmustern bewährt, nicht zuletzt duch den effizienten Algorithmus zur Analyse von Ladevorgängen und der Konstruktion eines entsprechenden Ladebaums. Als Konzept zur Formulierung von Ladestrategien im Code einer Anwendung ist es hingegen zu aufwändig, da für komplexe Ladebäume keine kurze Notationsform gefunden werden konnte. Allerdings ist zu überlegen, ob es nicht andere Möglichkeiten gibt, mit denen der Entwickler Ladebäume nutzen kann. Eine dieser Möglichkeiten ist die manuelle Definition von Zeitintervallen durch den Entwickler. Der in diesem Intervall entstandene Ladebaum könnte gespeichert und später an Stellen im Code mit ähnlichen Datenzugriffsmustern wiederverwendet werden.
Ein erheblich umfangreicherer Entwurf wäre die Einführung einer Lernphase für den Betrieb des OR-Mappers. In dieser Lernphase werden Datenzugriffsmuster analysiert und für einzelne Ladevorgänge die Wahrscheinlichkeiten von Nachladungen innerhalb eines kurzen Zeitraums berechnet. Die Lernphase muss bezüglich ihrer Datenzugriffe hinreichend repräsentativ für den späteren Betrieb der Anwendung sein. Nach Beenden der Lernphase könnten die gewonnenen stochastischen Daten benutzt werden, um Ladevorgänge zu optimieren, zum Beispiel um Nachladungen sofort vorzunehmen statt in einer separaten Anfrage an die Datenbank. Zusätzlich könnte der OR-Mapper Empfehlungen bezüglich der vom Entwickler definierten Ladestrategien machen. So ergibt ein Property, welches als lazy-load markiert ist, nur dann einen Sinn, wenn es in der Mehrheit der Fälle nie geladen wird. Ob dies der Fall ist, kann in der Lernphase des Mappers leicht festgestellt werden. Ebenso könnte überprüft werden, ob die Ladestrategien für persistente Collections angemessen gesetzt wurden.
Ein weiterer Anknüpfungspunkt an diese Arbeit wäre die Überprüfung, ob die Ladestrategien tatsächlich die erhoffte Verbesserung von Geschwindigkeit und Speichernutzung erbringt. Hierzu müssten eine Reihe von Benchmarks entwickelt werden. Solche Benchmarks existieren zwar für RDBMS, jedoch nicht für OR-Mapper, was aussagekräftige Geschwindigkeitsvergleiche unter OR-Mappern schwierig macht.

Beendet wurde der Hauptteil der Arbeit mit der Darstellung, wie sich der entwickelte Prototyp in die SoNET-Plattform einbinden lässt. SoNET befindet sich noch im ersten Drittel seiner Entwicklungsphase, zudem existieren bis jetzt keinerlei hinreichend komplexen Anwendungen. Aus diesem Grunde muss davon ausgegangen werden, dass einige Probleme beim Einsatz des Prototypen innerhalb

von SoNET noch nicht bekannt sind.

Mit der Konstruktionsweise von SoNET ist der Prototyp gut vereinbar. Die Änderungen, welche zur Integration vorgenommen wurden, konnten ausnahmslos schnell durchgeführt werden. Ein offener Punkt bleibt die Benutzung verschiedener Speicherungssysteme, die gegenwärtig noch den erläuterten Beschränkungen unterliegt. Anzumerken ist, dass andere OR-Mapper hier ebenfalls ihre Grenze finden. Da jedoch bestehende Systeme zur Integration unterschiedlicher Datenquellen existieren bietet sich als ein weiterer Anknüpfungspunkt hier an, die Ansätze und Techniken dieser Datenintegrationssysteme innerhalb des Prototypen umzusetzen.

Anhang A: Überblick C# und DOT.NET 2.0

Die Programmiersprache C# weist große Parallelen zu Java auf, es wird aus diesem Grunde an dieser Stelle nur auf Unterschiede zu Java bzw. neue Sprachkonstrukte bei C# eingegangen, die für diese Arbeit Relevanz haben. Näheres zu den Entwurfsprinzipien von C# findet sich bei [HEJ04], zur praktischen Anwendung bei [LIB05]. Über die interne Umsetzung von DOT.NET informiert [RIC06].

Application Domain: Neue Prozesse unter Windows zu erzeugen ist teuer bzgl. Zeit und Ressourcen. Um Prozesse und Adressräume zu entkoppeln, wurden unter NET sog. Application Domains (AD) eingeführt. Eine Anwendung läuft in einer default-AD und kann weitere ADs erzeugen. Kommunikation zwischen den ADs ist nur über Marshaling möglich und deshalb nicht besonders schnell.

Assembly: eine Assembly ist eine Menge von kompilierten Klassen und enthält zusätzlich Metadaten dieser Klassen. Sie ist vergleichbar mit einer JAR-Datei in Java.

Custom Cast: Wenn die Klassen A und B nicht in einer Vererbungslinie stehen, können sie normalerweise auch nicht zueinander gecastet werden. C# erlaubt jedoch, solche individuellen Casts in einer der Klassendefinitionen von A oder B zu schreiben. Solche Casts können als explicit oder als implicit markiert werden. Ein explicit-Cast kann im Code entsprechend nur explizit benutzt werden.

```
A a = new B();      // implicit cast is implemented
A a = (A) new B();  // explicit cast is implemented
```

Custom Attributes: Custom Attributes sind das NET-Äquivalent zu den Annotationen in Java 1.5, also individuell definierbare Metadaten, die im Code durch Reflection abgefragt werden können. Metadaten sind auf unterschiedlichen Ebenen möglich: Assembly, Typ, Interface oder einzelne Member.

Generic methods: Methoden können bei C# generisch sein, auch dann, wenn die zugehörige Klasse nicht generisch ist. Dies bringt den Vorteil von erhöhter Typsicherheit zur Lauf- und zur Entwicklungszeit, da potentielle Fehler durch Casts schon zur Entwicklungszeit ausgeschlossen werden können. Ein Beispiel für eine generische Methode ist:

Definition	Aufruf

class Session {
T newInstance<T>()
}
	Session ses = new Session();
	MyType m = ses.newInstance<MyType>();

Generic types: Es gibt seit C# 2.0 die Möglichkeit zur Definition generischer Typen. Ein gutes Beispiel bietet der Typ `Dictionary<K, V>` bei dem K dem Typ der Schlüssel und V dem Typ der Werte entspricht, die gespeichert werden sollen. Zusätzlich können bei der Definition generischer Typen Bedingungen angegeben werden, die einer oder mehrere der Typen erfüllen müssen. Als Beispiel nehmen wir ein generisches Dictionary zur Speicherung von Proxyobjekten:

```
class ProxyDictionary<K, V>
	where K: IPrimaryKey
	where V: IProxy
	{...}
```

Damit kann ProxyDictionary nur instanziiert werden, wenn der Typ K das Interface IPrimaryKey und der Typ V IProxy implementieren, so dass hier gefahrlos gecastet werden kann.
Die Umsetzung von Generics in C# 2.0 unterscheidet sich in einigen Punkten von derjenigen in Java 1.5. Einer der wichtigsten Punkte ist, dass Generics first-class-objects sind, mit entsprechend eigenen Sprachelementen in der Zwischensprache intermediate language und der Möglichkeit, bspw. die Typen K und V von ProxyDictionary per Reflection abzufragen.

Interfaces: Die Möglichkeiten zur Benutzung von Interfaces wurde in C# gegenüber Java deutlich erweitert. Es existieren zwei Möglichkeiten, ein Interface zu implementieren: direkt oder indirekt, in Java ist nur die indirekte Implementierung möglich.
Die direkte Implementierung von Interfaces ermöglicht es, Signatur-Ambivalenzen bei der Implementierung mehrerer Interfaces aufzulösen. Außerdem kann eine direkte Interfaceimplementierung nicht von einer Instanz der Klasse aufgerufen werden und „zwingt“ somit den Entwickler zur Nutzung des Objekts via Interface. Ein Beispiel hierzu:

```
public interface IDocument {
	void close();
}

public interface ISocket {
	void close();
}

public class MyClass: IDocument, ISocket {
	void IDocument.close() {
		// this method is called when the instance is used as IDocument
	}

	void ISocket.close() {
		// this method is called when the instance is used as ISocket
	}

	void close() {
		// this method is called when the instance is used as MyClass
	}
}
```

Im folgenden Codefragment werden gemäß obiger Implementierung drei verschiedene Methoden aufgerufen:

```
MyClass myInstance = new MyClass();
myInstance.close();
ISocket socket = new MyClass();
socket.close();
IDocument document = new MyClass();
document.close();
```

Iteratoren: Ein Iterator für eine Klasse kann in C# durch die Implementierung des Interfaces IEnumerator erstellt werden, welcher den yield-Operator einsetzt. Ein einfaches Beispiel hierfür sieht folgendermaßen aus:

```
public object IEnumerator enumerate() {
	for (int i=0; i < this.Count(); i++) {
		yield return this[i];
	}
}
```

Die Implementation von IEnumerator führt dazu, dass eine Instanz des Typs in einer foreach-Schleife durchlaufen werden kann.

Member ist die Menge von Klassen- und Instanzmethoden, Klassen und

Instanzvariablen, Properties, Events und Delegates einer Klasse.

Properties sind Kapselungen von privaten Instanzvariablen. Im Kompilat werden Properties als Methoden angelegt, sie dienen hauptsächlich einer kürzeren und übersichtlicheren Syntax. Beispiel:

Sichtbarkeitsbereiche werden für Klassen, Interfaces und ihre einzelnen Member definiert. Folgende Bereiche gibt es: public, private, protected, internal sowie protected internal. Public und private entsprechen denjenigen Bereichen in Java. Protected ermöglicht einen Zugriff nur aus abgeleiteteten Klassen, internal nur aus Klassen der gleichen Assembly. Protected internal entspricht einer oder-Verknüpfung von protected sowie internal.

Type-operators (is, as, typeof()): Drei besonders wichtige Typoperatoren in C#.
`obj is T` überprüft, ob die Instanz obj vom Typ T ist oder von einem von T abgeleiteten Typ. Falls T ein Interface ist, wird true zurückgegeben, wenn der Typ von obj T implementiert.
`obj as T` liefert obj nach T gecastet zurück, falls der cast vorgenommen werden kann. Falls nicht, wird null zurückgeliefert (es wird allerdings dann keine InvalidCastException ausgelöst, wie dies bei `(T) obj` geschähe).
`typeof(obj)` liefert den Typ von obj zurück. Äquivalent wäre der Aufruf `obj.getType()`.

Vererbungsmodifikatoren erfüllen die gleiche Funktion wie in Java. Abstract ist bei beiden Sprachen gleich benutzbar. Das Abschließen einer Vererbungslinie von Klassen bzw. Methoden geschieht in Java mit final, in C# mit sealed.
In Java sind sämtliche Methoden implizit als virtuell gekennzeichnet, d.h. eine abgeleitete Klasse, die eine Methode gleicher Signatur definiert, überschreibt die Methode der Elternklasse. In C# hingegen kann eine Methode optional mit virtual gekennzeichnet werden. Methoden einer Kindklasse können diese virtuellen Methoden dann explizit überschreiben (mit override) oder als neu definieren (mit new).

Verwendete Quellen:

[AMB03] Ambler, Scott W.: Agile Database Techniques, Wiley 2003

[AOS06] Aspect-Oriented Software-Development, unter http://www.aosd.net

[ATK95] Atkinson, Malcolm; Morrison, Ronald: Orthogonally Persistent Object Systems, VLDB Journal 4/3 1995

[ATK00] Atkinson, Malcolm: A Review of the Rationale and Architectures of Pjama, DCS Tech Report, University of Glasgow 2000

[BAU05] Bauer, Christian; King, Gavin: Hibernate in Action, Manning 2005

[BAU07] Bauer, Christian; King, Gavin: Java Persistence with Hibernate, Manning 2007

[BEL92] Bell, David; Grimson, Jane: Distributed Database Systems, Addison-Wesley, 1992

[BOC02] Bock, Jason: CIL Programming, apress 2002

[CAS06] The Castle-Project: http://www.castleproject.org/

[CEL05] Celko, Joe: SQL for Smarties 3rd ed., Morgan Kaufmann Publishers 2005

[CLA05] Clarke, Siobhan; Baniassad, Elisa: Aspect-oriented Analysis and Design, Pearson Education, 2005

[CWA06] Cwalina, Krzysztof; Abrams, Brad: Framework Design Guidelines, Addison-Wesley 2006

[DAT04] Date, C. J.: Introduction to Database Systems 8th ed., Pearson Education 2004

[DAT06] DataObjects.NET: http://www.x-tensive.com/Products/DataObjects.NET/

[DEL06] Delegates bei C#: http://de.wikipedia.org/wiki/Methodenzeiger

[DYN06] Castle Dynamic Proxy: http://wiki.castleproject.org/index.php/DynamicProxy

[ECL06] Eclipse-Projekt, unter http://www.eclipse.org

[FLA02] Flanagan, David: Java in a nutshell 4th ed., O'Reilly 2002

[FOW03] Fowler, Martin: Patterns of Enterprise Application Architecture, Addison-Wesley 2003

[FOW04] Fowler, Martin: Inversion of Control Containers and the Dependency Injection pattern, unter http://www.martinfowler.com/articles/injection.html

[FSH06] F#, unter http://research.microsoft.com/fsharp/fsharp.aspx

[FUS97a] Fussel, Mark: Foundations of Object Relational Mapping, 1997, unter http://www.chimu.com/publications/objectRelational/index.html

[GHJV95] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: Design Patterns, Addison-Wesley 1995

[GRA93] Gray, Jim; Reuter, Andreas: Transaction Processing, Morgan-Kaufmann Publishers, 1993

[HÄR01] Härder, Theo; Rahm, Erhard: Datenbanksysteme – Konzepte und Techniken der Implementierung, 2. Auflage, Springer 2001

[HBKZ00] He, Zhen; Blackburn, Stephen M.; Kirby, Luke; Zigman, John: Platypus – Design and Implementation of a Flexible High Performance Object Store (in Persistent Object Systems 9th international workshop POS-9, Lillehammer, September 2000, Springer)

[HEJ04] Hejlsberg, Anders; Wiltamuth, Scott; Golde, Peter: Die C# Programmiersprache, Addison-Wesley 2004

[HEJ05] Hejlsberg, Anders: The LINQ-project

(http://msdn.microsoft.com/data/ref/linq/default.aspx?pull=/library/en-us/dndotnet/html/linqprojectovw.asp)

[HÖP05] Höpfner, Hagen; Türker, Can; König-Ries, Birgitta: Mobile Datenbanken und Informationssysteme, dpunkt-Verlag 2005

[HOH06] Hohenstein, Uwe; Bartholdt, Jörg: Performante Anwendungen mit Hibernate, Java-Spektrum 4/2006, S. 44ff

[JAC92] Jacobson, Ivar; Christerson, Magnus; Jonsson, Patrik; Övergaard, Gunnar: Object-Oriented Software Engineering, Addison-Wesley, 1992

[JAL05] Jalote, Pankaj: An integrated approach to software engineering 3rd ed., Springer 2005

[JOR03] Jordan, David; Russell, Craig: Java Data Objects, O'Reilly 2003

[KLE03] Kleppe, Anneke; Warmer, Jos; Bast, Wim: MDA explained, Pearson Education 2003

[KLA06] Klar, Michael; Klar, Susanne: Einfach generieren – Generative Programmierung verständlich und praxisnah, Hanser 2006

[LAH05] Lahdenmäki, Tapio; Leach, Michael: Relational Database Index Design and the Optimizers, Wiley 2005

[LIB05] Liberty, Jesse: Programming C# 4th ed., O'Reilly 2005

[LMG00] Lewis, Brian; Mathiske, Bernd; Gafter, Neal: Architecture of the PEVM – A High-Performance Orthogonally Persistent Java Virtual Machine, SUN Microsystems, 2000 (in: Persistent Object Systems 9th international workshop POS-9, Lillehammer, September 2000, Springer)

[MAR02] Marinescu, Floyd: EJB Design Patterns, Wiley 2002

[MCC06] McConnell, Steve: Software Estimation, Microsoft Press 2006

[MEY97] Meyer, Bertrand: Object-oriented Software Construction 2nd ed., Prentice-Hall, 1997

[NOC04] Nock, Clifton: Data Access Patterns, Addison Wesley 2004

[OAW06] OpenArchitectureWare (http://www.openarchitectureware.org), Stand August 2006

[OSG06] OSGi, unter http://www.osgi.org

[POB05] Pobar, Joel: Dodge Common Performance Pitfalls to Craft Speedy Applications, MSDN-Magazine 07/2005, auch unter http://msdn.microsoft.com/msdnmag/issues/05/07/Reflection/

[PUG06] Pugh, Ken: Interface-Oriented Design, The Pragmatic Bookshelf 2006

[REN98] Ren, Qun; Dunham, Margaret H.: Semantic Caching and Query Processing, Technical Report 98-CSE-04, Department of Computer Science and Engineering, Southern Methodist University, Dallas, 1998

[RIC06] Richter, Jeffrey: CLR via C# 2nd ed., Microsoft Press 2006

[SIL06] Silberschatz, Abraham; Korth, Henry F.; Sudarshan, S.: Database System Concepts 5th edition, McGraw-Hill 2006

[SOC06] Separation of concerns, unter http://en.wikipedia.org/wiki/Separation_of_concerns

[SOS03] Sosnoskis, Dennis M.: Java programming dynamics, Part 2: Introducing reflection, unter http://www-106.ibm.com/developerworks/java/library/j-dyn0603/

[STA05] Stahl, Thomas; Völters, Markus: Modellgetriebene Softwareentwicklung, dpunkt-Verlag 2005

[TÜR03] Türker, Can: SQL:1999 & SQL:2003 – Objektrelationales SQL, SQL/J & SQL/XML, dpunkt-Verlag 2003

[TÜR06] Türker, Can; Saake, Gunter: Objektrelationale Datenbanken, dpunkt-Verlag, 2006

[VOS00] Vossen, Gottfried: Datenmodelle, Datenbanksprachen und DatenbankManagementsysteme (4. Auflage), Oldenbourg 2000

[WIE05] Wiegert, Oliver: Ein praktischer Leitfaden für eine iterative Softwareentwicklung (in: Breu, Ruth et.al: Software-Engineering, Oldenbourg 2005)

[WIN05] Winter, Mario: Methodische objektorientierte Softwareentwicklung, dpunkt-Verlag 2005

[WIN06] Windsor-Container: http://www.castleproject.org/container/index.html

Abbildungsverzeichnis:

Tabellenverzeichnis:

Listingverzeichnis: