FernUniversität in Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet Programmiersysteme

Bachelorarbeit Informatik

Implementierung eines objektorientierten Domänenmodells mit bidirektionalen, mengenwertigen Verweisen in Java

Maik Jablonski

01.11.2011

Betreuer

Prof. Dr. Friedrich Steimann
Lehrgebiet Programmiersysteme
FernUniversität in Hagen

Maik Jablonski

Implementierung eines objektorientierten Domänenmodells mit bidirektionalen, mengenwertigen Verweisen in Java

Bachelorarbeit Informatik
FernUniversität in Hagen

Bearbeitungszeitraum: 15. August 2011 - 15. November 2011

Inhaltsangabe

Die Implementierung von bidirektionalen Beziehungen zwischen Domänenobjekten ist mit objektorientierten Programmiersprachen im Vergleich zu den Möglichkeiten des relationalen Datenmodells sehr umständlich, da Verweise zwischen Objekten stets gerichtet sind. In der wissenschaftlichen Diskussion finden sich deshalb verschiedene Ansätze, Relationen in die Objektorientierung einzuführen, allerdings zu dem Preis, die effiziente und verbreitete Verwendung von Referenzen und Kollektionen für die Realisierung von mengenwertigen Verweisen durch dezidierte Relationsobjekte bzw. Spracherweiterungen zu ersetzen. In der vorliegenden Arbeit wird eine aspektorientierte Bibliothek für Java entwickelt und diskutiert, die es mit Hilfe von Annotationen erlaubt, Domänenmodelle mit bidirektionalen, mengenwertigen Verweisen auf Basis von Referenzen und Kollektionen zu implementieren.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Nutzung von unidirektionalen Verweisen zwischen Objekten ist bei der objektorientierten Entwicklung von Geschäftsanwendungen für unterschiedliche Fachdomänen einfach und effektiv: Ein Mitarbeiter referenziert seinen Arbeitsplatz, der Arbeitsplatz wiederum seine Abteilung. Zur Realisierung mengenwertiger Verweisen nutzt man die vom Sprachumfang bzw. die von Standardbibliotheken zur Verfügung gestellten Kollektionen zur Kapselung einer Menge von Verweisen: ein Mitarbeiter speichert die ihm zugeordneten Projekte z.B. in einer Liste. Auf Basis von unidirektionalen Referenzen und Kollektionen zwischen verschiedenen Objekten können komplexe Objektgraphen implementiert werden, die sich durch Navigation von Verweis zu Verweis nach definierten Kriterien abfragen und manipulieren lassen.

Während es im objektorientierten Datenmodell sehr einfach ist, auszudrücken, welcher Arbeitsplatz einem bestimmten Mitarbeiter zugeordnet ist, ist die Frage in der umgekehrten Richtung, also welcher Mitarbeiter gehört zu einem bestimmten Arbeitsplatz, nicht ohne weiteres und mitunter nur sehr umständlich zu beantworten. Der Arbeitsplatz weiß im objektorientierten Datenmodell nicht automatisch, durch welchen Mitarbeiter er referenziert wird. Dies ist ein großer Unterschied zum relationalen Datenmodell, wo die Blickrichtung, aus der eine Abfrage gestellt wird, egal ist. Die Relation zwischen Mitarbeiter und Arbeitsplatz kann sowohl vom Mitarbeiter als auch vom Arbeitsplatz gleichberechtigt befragt werden. Eine vergleichbare Möglichkeit zur Navigation bidirektionaler Verweise erhält man im objektorientierten Datenmodell nur dadurch, dass man für jeden Verweis stereotypen Programmcode

zur Synchronisierung der Verweise zwischen den zwei beteiligten Objekten einfügt (vgl. hierzu [Noble, 1997]).

Dieses relationale Defizit der Objektorientierung wurde bereits früh von [Rumbaugh, 1987] beklagt und es gibt seither zahlreiche Versuche, die Vorzüge des relationalen Datenmodells in die Objektorientierung zu integrieren. Ein großer Teil der diskutierten Entwürfe ersetzt dabei das verbreitete Paradigma zur Realisierung von mengenwertigen Verweisen mit Referenzen und Kollektionen: Relationen werden als Klassenbibliothek (z.B. bei [Osterbye, 2007]), Aspekte (z.B. bei [Pearce and Noble, 2006]) oder über Spracherweiterungen (z.B. bei [Rumbaugh et al., 1989]) in die Objektorientierung eingeführt, sind aber in dieser Form durch die Praxis weitgehend ignoriert worden. Die Gründe hierfür mögen vielfältig sein, ein zentraler Grund ist sicherlich die Effizienz der Dereferenzierung von Zeigern ([Steimann and Stadtler, 2010b]). Das verbreitete Idiom der Nutzung von Referenzen und Kollektionen zur Realisierung von Verweisen besitzt ein derart gewichtiges Momentum, auf welches Entwickler wegen langjährig erlernter Patterns sowie der Werkzeugunterstützung nicht ohne Not verzichten mögen.

Das Ziel dieser Arbeit ist es, eine Bibliothek zu entwickeln, die es ermöglicht, bidirektionale, mengenwertige Verweise in Domänenmodellen mit Java auf Basis von gewöhnlichen Referenzen und Kollektionen zu implementieren. Grundidee der Implementierung wird die Nutzung von Annotationen sein, mit denen bidirektional zu bindende Felder in Domänenklassen ausgezeichnet werden. Über aspektorientierte Programmierung werden diese Annotationen ausgewertet und die Verweise automatisch und transparent für den Entwickler synchronisiert. Der eigene Beitrag dieser Arbeit liegt in der Demonstration, dass es mit in der Praxis weit verbreiteten Technologien[1] problemlos möglich ist, bidirektionale, mengenwertige Verweise für die Entwicklung von Domänenmodellen in Java generisch zu realisieren und dabei vom Entwickler nur marginale Änderungen seiner gewohnten Implementierungsstrategien zu verlangen.

Ein Überblick über den Aufbau der Arbeit soll dem Leser das Verständnis

[1] Als Beispiel für die verbreitete Nutzung von Annotationen und aspektorientierter Programmierung sei hier das Spring-Framework (`http://www.springsource.org/`) genannt, welches mittlerweile als Quasi-Standard für die Entwicklung von Geschäftsanwendungen im Java-Bereich gilt.

der Argumentation erleichtern: Im Kapitel 2 „Problemstellung" werden ausgehend von einem einfachen Domänenmodell die möglichen Kardinalitäten und Richtungen von Beziehungen eingeführt. Anschließend werden verschiedene Ansätze zur Integration von Relationen in die Objektorientierung diskutiert, die aber die in der Praxis verbreitete Verwendung von Referenzen und Kollektionen zur Realisierung mengenwertiger Verweise vernachlässigen. Diese Lösungsansätze werden im Kapitel 3 „Verwandte Arbeiten" noch einmal auf Basis der wissenschaftlichen Diskussion vertieft. In dem Kapitel 4 „Bidirektionale Verweise mit Annotationen und Aspekten" wird eine Bibliothek konzipiert und implementiert, die mit Hilfe von aspektorientierter Programmierung und paarigen Annotationen auf bidirektional zu bindenden Feldern eine für den Java-Entwickler transparente Synchronisation von mengenwertigen Verweisen auf Basis von Referenzen und Kollektionen realisiert. Im Kapitel 5 „Ergebnisse" werden der Implementierungsaufwand und die Laufzeitkosten der entwickelten Lösung am Beispiel eines einfachen Domänenmodells im Vergleich zu einer manuellen Umsetzung diskutiert. Abschließend wird die Bibliothek in Bezug auf ihre Vor- und Nachteile bewertet. Den Abschluss bildet das Kapitel 6 „Schlussbetrachtung", das nach einer Zusammenfassung einen Ausblick auf mögliche Weiterentwicklungen der vorgestellten Bibliothek gibt.

Kapitel 2

Problemstellung

Die Analyse der fachlichen Domäne ist der erste Schritt in der Entwicklung komplexer Geschäftsanwendungen. Dabei werden Entitäten und ihre Beziehungen identifiziert: Entitäten sind eindeutig bestimmbare Objekte der Fachdomäne, über die Daten gespeichert und verarbeitet werden sollen. In objektorientierter Terminologie werden Entitäten auch Geschäfts- bzw. Domänenobjekte genannt. Die Beziehungen der Entitäten untereinander werden allgemein als Relationen und in objektorientierter Terminologie als Assoziationen bezeichnet.

2.1 Kardinalitäten von Beziehungen

Bei der Datenmodellierung ist eine zentrale Eigenschaft einer Beziehung zwischen Entitäten ihre Kardinalität, die ausdrückt, mit wie vielen Entitäten eine gegebene Entität grundsätzlich in Beziehung stehen kann. Um die verschiedenen Varianten möglicher Kardinalitäten zu erläutern, soll auf ein vereinfachtes Modell einer Geschäftsanwendung aus Mitarbeitern, Arbeitsplätzen, Abteilungen und Projekten zurück gegriffen werden, welches durch ein Entity-Relationship-Diagramm grafisch veranschaulicht wird.

1-zu-1: Jeder Mitarbeiter besetzt einen Arbeitsplatz und jeder Arbeitsplatz kann maximal von einem Mitarbeiter besetzt werden.

1-zu-N: Jeder Abteilung sind mehrere Arbeitsplätze zugeordnet bzw. ein Arbeitsplatz ist genau einer Abteilung zugeordnet.

N-zu-M: Jeder Mitarbeiter ist einem oder mehreren Projekten zugeordnet bzw. jedem Projekt sind mehrere Mitarbeiter zugeordnet.

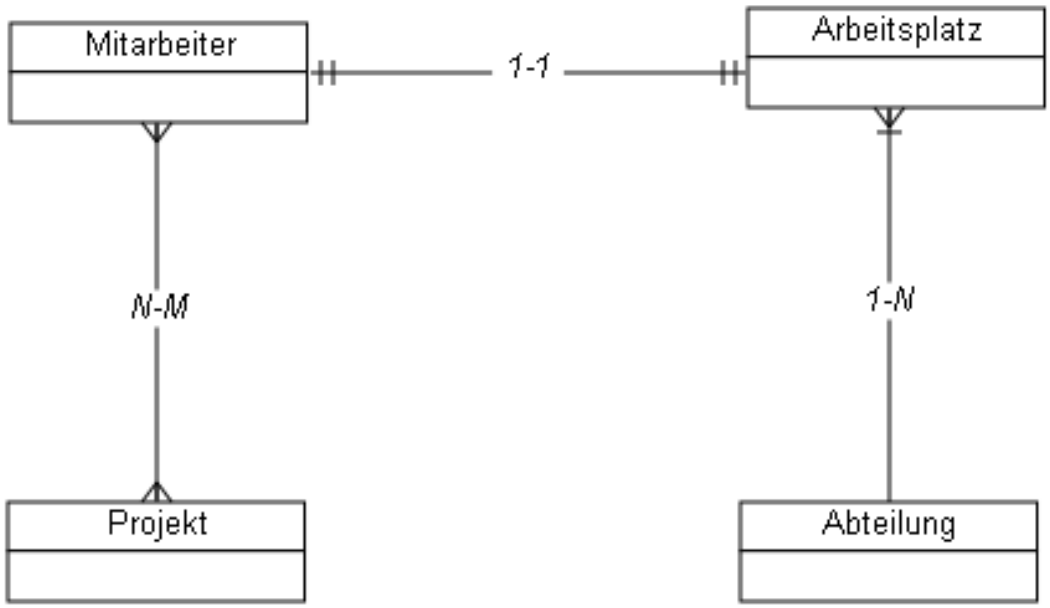

Abbildung 2.1: Entity-Relationship-Diagramm

Die Kardinalität einer Beziehung sollte grundsätzlich feststehen und ist in der Regel eine Invariante der Fachdomäne. Allerdings gibt es in der Praxis häufig genug den Fall, dass sich bei der Analyse einer Fachdomäne bestimmte Aspekte erst im späteren Projektverlauf zeigen bzw. sich unter Umständen sogar die Fachdomäne ändern kann: wenn z.B. ein Unternehmen ein Job-Sharing einführt bzw. dieses bei der Analyse der Fachdomäne nicht als möglicher Fall berücksichtigt wurde, muss die Kardinalität der Beziehung von Arbeitsplatz und Mitarbeiter angepasst werden, weil zukünftig ein Arbeitsplatz von mehreren Mitarbeitern besetzt werden kann. Auch wenn der Unterschied beim Übergang von einer Zu-1- zu einer Zu-N-Beziehung im Entity-Relationship-Diagramm nur marginal erscheint, ist der Implementierungsaufwand in der Regel signifikant, da in objektorientierten Programmiersprachen das Zugriffsprotokoll zwischen einfachen Referenzen und mengenwertigen Verweisen unterschiedlich ist und mitunter an vielen Stellen im Programmtext angepasst werden muss.

2.2 Richtungen von Beziehungen

Ein weiteres wichtiges Merkmal des Entity-Relationship-Diagramms ist die Tatsache, dass sämtliche Beziehungen ungerichtet bzw. immanent bidirektional sind, d.h. in Abhängigkeit von der Fragestellung in beide Richtungen abgefragt werden können. Im relationalen Modell macht es also keinen Unterschied, ob man wissen möchte, welcher Mitarbeiter einen bestimmten Arbeitsplatz besetzt oder welcher Arbeitsplatz von einem bestimmten Mitarbeiter besetzt wird.

Beim Übergang zum objektorientierten Design und damit beim Übergang vom Entity-Relationship-Diagramm zum Klassendiagramm der Unified Modeling Language (UML) muss die Ungerichtetheit bzw. die immanente Bidirektionalität von Beziehungen im relationalen Modell explizit modelliert werden. Assoziationen zwischen Objekten sind stets gerichtet, so dass unidirektionale von bidirektionalen Beziehungen durch entsprechende Pfeilspitzen im Diagramm unterschieden werden: eine unidirektionale Beziehung zwischen einem Mitarbeiter und einem Arbeitsplatz kann nur vom Mitarbeiter zum Arbeitsplatz navigiert werden und nicht umgekehrt. Die Ursache hierfür liegt darin, dass in der Objektorientierung letztlich Zeiger (also Verweise auf Speicheradressen) benutzt werden, um eine Beziehung von einem Objekt zu einem anderen Objekt anzugeben. Während in maschinennahen Sprachen wie C oder C++ Zeiger auf Speicheradressen frei erzeugt und benutzt werden können, sind sie in Sprachen wie Java, die auf der Abstraktion einer virtuellen Maschine aufsetzen, zwar intern vorhanden, für den Programmierer aber nur in der Form von Referenzen zugänglich, die unablösbar an Objekte innerhalb der virtuellen Maschine gebunden sind.

2.3 Bidirektionale Verweise als Problem

Die mangelnde Bidirektionalität von mengenwertigen Verweisen stellt ein fundamentales Problem beim Übergang von der Modellierung zur objektorientierten Implementierung dar: Beziehungen, die im Modell selbstverständlich bidirektional sind, werden auf Ebene der Implementierung problematisch, weil es keine einfache Lösung gibt, die Bidirektionalität automatisiert bzw. mit gerin-

gem Wartungsaufwand zu erzeugen. Die aus der wissenschaftlichen Diskussion bekannten Ansätze, die in Kapitel 3 im Detail vorgestellt werden, sind aus der Sicht der Praxis mit dem Fokus der Entwicklung von Geschäftsanwendungen unbefriedigend:

Patterns erlauben im Sinne eines Entwurfsmusters eine einfache Verständigung unter Entwicklern darüber, wie bestimmte Probleme im Rahmen einer definierten Umgebung sinnvoll gelöst werden können, belassen aber die konkrete Implementierung in den Händen der einzelnen Entwickler. Die Realisierung bidirektionaler Beziehungen ist mit den vorhandenen Sprachmitteln objektorientierter Programmiersprachen zwar über eine explizite Synchronisation bzw. durch die Nutzung dezidierter Relationsobjekte möglich, führt aber zur Duplizierung von strukturell identischem Code für jede zu implementierende Beziehung und scheidet damit als nachhaltige Lösung wegen des Aufwands und der schlechten Wartbarkeit aus.

Code-Generatoren automatisieren zwar die Erzeugung des erforderlichen Codes für die bidirektionale Synchronisierung von Beziehungen, führen aber eine zusätzliche Komplexität in den Entwicklungsprozess ein und geben die Richtung für diesen in der Regel vor: der Weg führt vom Modell zum Code und häufig nicht umgekehrt, was den Einsatz in agilen Projekten oder beim Prototyping schwer macht.

Klassenbibliotheken führen Relationen als eigenständige Relationsobjekte ein und berauben sich auf diese Weise der Möglichkeit, Zu-1-Beziehungen über die Dereferenzierung von Zeigern zu modellieren, was für den Entwickler in der Praxis ein so gängiges Idiom der Objektorientierung ist, dass offensichtlich eher auf die Vorteile der Bidirektionalität als auf die direkte Navigation von Objekt zu Objekt verzichtet wird.

Aspekte erlauben es, Relationen mit ihren Eigenschaften als eigenständige sprachliche Konstrukte in Form von Aspekten zu erfassen, während gleichzeitig auf der Ebene des Bytecodes keine oder nur geringe Unterschiede zu manuell erzeugtem Code und damit keine nennenswerten Einbußen in Bezug auf die Laufzeiteffizienz sichtbar werden. Nichtsdestotrotz verdeckt auch hier eine Bibliothek von Relationsaspekten das in

der Praxis bedeutende Idiom der unmittelbaren Navigation von Objekt zu Objekt und erfordert darüber hinaus durchgängig die Nutzung einer um die Aspektorientierung erweiterten Entwicklungsumgebung, die nicht immer realisierbar ist.

Spracherweiterungen haben es in der Praxis schwer, weil in vielen Projekten sich daraus ergebende Nischentechnologien aus technischen oder politischen Gründen grundsätzlich nicht eingesetzt werden. Solange also die Spracherweiterungen nicht den Weg in den Standard finden, werden sie nicht genutzt und solange sie nicht genutzt werden, werden sie auch kein Standard.

Es ist offenkundig, dass mit den vorhandenen Mitteln von Java keine befriedigende Lösung für das Problem der mangelnden Bidirektionalität von Verweisen erzielt werden kann. Die diskutierten Ansätze haben bislang aus den genannten Gründen keinen nennenswerten Eingang in die Praxis der Softwareentwicklung gefunden, vor allem auch deshalb, weil sie das verbreitete Idiom und die Effizienz der Objekt-zu-Objekt-Navigation vernachlässigen. Es stellt sich deshalb als Problemstellung dieser Arbeit die Frage, ob und wie es möglich ist, eine Implementierung für bidirektionale, mengenwertige Verweise für objektorientierte Domänenmodelle in Java bereit zu stellen, die Aussicht auf Praxisakzeptanz hat, weil sie das allgemeine und effiziente Idiom zur Realisierung mengenwertiger Verweise in Form von Referenzen und Kollektionen um Bidirektionalität erweitert, ohne dabei den gewohnten Entwicklungsprozess nennenswert zu verändern.

Kapitel 3

Verwandte Arbeiten

Um dem Problem der mangelnden Bidirektionalität objektorientierter Assoziationen zu begegnen, wurden in der wissenschaftlichen Diskussion zahlreiche Vorschläge entwickelt, die sich für die Strukturierung der Diskussion in die Kategorien „Patterns", „Code-Generatoren", „Klassenbibliotheken", „Aspekte" und „Spracherweiterungen" einordnen lassen.

3.1 Relationship Patterns

Für die objektorientierte Implementierung von Assoziationen werden die wesentlichen Patterns von [Noble, 1997][1] beschrieben:

Relationship as Attribute: Eine gerichtete 1-zu-1-Assoziation zwischen Objekten kann über ein Attribut realisiert werden, das jeweils eine Referenz auf das referenzierte Objekt enthält. Der Zugriff auf diese Assoziation ist sehr effizient, da eine direkte Navigation zum referenzierten Objekt über den Verweis möglich ist.

Collection Object: Eine gerichtete 1-zu-N-Assoziation kann mittels eines Container- bzw. Kollektionsobjekts realisiert werden, welches eine Menge von Verweisen auf die zu referenzierenden Objekte aufnehmen kann. Im

[1] Noble beschreibt in seiner Arbeit fünf Patterns, wovon in der vorliegenden Arbeit nur vier Patterns vorgestellt werden. Das fünfte Pattern „Active Value" beschreibt die Kapselung eines Wertes über definierte Zugriffsfunkionen, um registrierte Observer über Änderungen an dem Wert benachrichtigen zu können. Dieses Pattern spielt für die weitere Diskussion in dieser Arbeit keine Rolle.

referenzierenden Objekt wird eine Referenz auf eine Kollektion als Attribut realisiert, über die dann über diesen zwischengeschalteten Umweg zu den referenzierten Objekten navigiert werden kann.

Mutual Friends: Bidirektionale Beziehungen können durch expliziten Programmcode über konsistent zu haltende gerichtete Assoziationen realisiert werden. Dazu müssen Verweise in eine Richtung programmatisch durch entsprechende Rückverweise synchronisiert werden. Auf diese Weise wird sehr viel stereotyper Programmcode eingeführt, da die grundsätzlichen Algorithmen der Synchronisation von Referenzen immer gleich sind.

Relationship Object: Durch die Reifizierung von Assoziationen als eigenständige Klassen können komplexe, bidirektionale Assoziationen abgebildet werden. Dazu wird für jede Relation eine eigenständige Klasse gebildet, die entsprechende Tupel der verknüpften Objekte samt zusätzlicher Attribute speichern kann. [Noble and Grundy, 1995] argumentieren, dass die Einführung von expliziten Relationsobjekten in der objektorientierten Entwicklung zu schlankeren Domänenobjekten mit einer reduzierten Kopplung zwischen den Klassen führt, da sämtlicher Code, der Klassen miteinander in Beziehung setzt, in eine jeweils eigene Klasse ausgelagert wird. [Balzer et al., 2007] arbeiten auf Basis eines mathematischen Relationsbegriffs heraus, dass eigenständige Relationsobjekte die gültigen Invarianten einer Relation an einer Stelle im Programmcode zusammenführen.

Bidirektionalität lässt sich mit den vorhandenen Möglichkeiten der Objektorientierung nur manuell über die Patterns „Mutual Friends" bzw. „Relationship Object" implementieren. Insbesondere die Implementierung von „Mutual Friends" führt dabei zu sehr viel stereotypem Programmcode für die Synchronisation der beteiligten Objektreferenzen, der insbesondere bei einer Änderung der Kardinalität aufwändig in mehreren Klassen angepasst werden muss. Der Aufwand für die manuelle Implementierung und Pflege bidirektionaler Assoziationen ist so hoch, dass sogar Protagonisten eines „Domain Driven Designs"[2] wie [Evans, 2003, S. 82ff] vorschlagen, auf bidirektionale Assoziationen

[2]Das „Domain Driven Design" erklärt die unverfälschte Spiegelung der Fachlichkeit einer Domäne im Softwaredesign zum absoluten Dreh- und Angelpunkt.

bei der Implementierung nach Möglichkeit zu verzichten. Der Entwickler soll sich stattdessen für einen Navigationspfad als Hauptrichtung der Assoziation entscheiden.

Um das Problem und den Aufwand der manuellen Implementierung von bidirektionalen Assoziationen zu umgehen, können als Alternative in Geschäftsanwendungen Abfragen über die Persistenzschicht (also in der Regel eine relationale Datenbank) genutzt werden, um die verknüpften Entitäten für eine gegebene Entität zu ermitteln. Hierbei findet allerdings eine Vermischung von Architekturschichten statt: das Domänenmodell sollte im Sinne einer klaren Aufgabentrennung frei von direkten Zugriffen auf die Datenbank sein, so dass als Workaround für eine saubere Architektur die Verlagerung entsprechender Datenbankabfragen und darauf aufbauender fachlicher Operationen in eine umfangreiche Serviceschicht vorgenommen wird. Das ursprünglich aus Attributen und Verhalten bestehende Domänenmodell mutiert auf diese Weise zu einem Anti-Pattern, dem so genannten „anämischen Domänenmodell"[3], welches im Kern nur noch Attribute, aber selbst keine fachlichen Operationen auf der Domäne implementiert. Die Domänenobjekte werden so auf einfache Datenhalter ohne Verhalten reduziert, was das Verständnis für das komplexe Zusammenspiel erschwert und die sprachlichen Ausdrucksmöglichkeiten der Objektorientierung ungenutzt lässt.

3.2 Code-Generatoren

Eine Möglichkeit, bidirektionale Assoziationen unterschiedlicher Kardinalitäten zu erzeugen, besteht in der automatisierten Generierung des hierfür notwendigen stereotypen Codes aus den vorhandenen Artefakten der Modellierung. So setzen [Genova et al., 2003] mit ihrem Werkzeug direkt auf Klassendiagrammen der Domänenanalyse auf und erzeugen hieraus Java-Klassen für die weitere Verwendung durch den Programmierer, die bereits mit dem Code für die Synchronisation bidirektionaler Assoziationen angereichert sind.

[Amelunxen et al., 2004] nutzen die strukturellen Möglichkeiten der „Meta Object Facility" als Modellierungssprache auf Metaebene, die als gemeinsame

[3]Das „Anemic Domain Modell" wurde als Anti-Pattern von Martin Fowler beschrieben: `http://martinfowler.com/bliki/AnemicDomainModel.html`

Beschreibung für objektorientierte Modellierungssprachen (also z.B. der UML) verwendet wird, um die Codegenerierung für Relationen auf einer höheren Ebene zu abstrahieren. Sie entwickeln auf diese Weise einen sehr generischen Ansatz für die automatisierte Codeerzeugung aus Modellierungsartefakten.

3.3 Klassenbibliotheken

[Osterbye, 2007] schlägt eine Klassenbibliothek für C# mit dem Namen „NOIAI" („No Object is an Island") vor: Relationen werden in diesem Framework als Klassen deklariert, die von einer gemeinsamen Grundklasse mit dem Namen `BaseAssociation` abgeleitet werden. Die Grundklasse stellt grundlegende Operationen zum Umgang mit der Relation zur Verfügung (`Add, Remove, Lookup`). Der Zugriff auf eine gegebene Relation ist durch die Implementierung als Singleton möglich. Darüber hinaus kann eine Relation auch in den beteiligten Domänenklassen über eine von der Relation zur Verfügung gestellte Referenz für die jeweilige Domänenklasse als Attribut benutzt werden.

[Nelson et al., 2008] entwickeln eine Bibliothek, um Relationen in Java als eigenständige Klassen typsicher definieren zu können. Sie nutzen dazu zwei Grundklassen (`Pair, PairAssoc`), die über eine komplizierte Schachtelung innerer Klassen bei der Ableitung konkreter Relationen zu dem gewünschten Ergebnis führen.

3.4 Aspekte

[Pearce and Noble, 2006] reifizieren Relationen für Java mit Hilfe von AspectJ zu eigenständigen Aspekten, um auf diese Weise die klassenübergreifenden Operationen für das Management einer Relation an einer zentralen Stelle zusammenzufassen. Sie führen hierzu eine „Relationship Aspect Library" (RAL) von statischen und dynamischen Aspekten für verschiedene Kardinalitäten ein, von denen konkrete Relationen abgeleitet werden. Die Relationsaspekte werden für den Entwickler ähnlich wie „Relationship Objects" gehandhabt, aber letztlich über den AspectJ-Compiler in den Java-Bytecode der Domänenklassen eingewoben.

[Pradel, 2008] stellt einen Ansatz für explizite Relationen auf Basis von

Scala vor, der Traits als dynamische Mixins in Klassen benutzt: Traits erlauben in Scala eine Mehrfachvererbung und stellen ein spracheigenes Konstrukt dar, das in Sprachen mit Einfachvererbung wie Java nur über den Zusatz der aspektorientierten Programmierung verfügbar ist. Dabei werden Objekte, die als Bestandteil einer Relation deklariert werden, um neue Funktionalitäten für die korrekte Synchronisierung mit der Relation angereichert.

3.5 Spracherweiterungen

[Rumbaugh, 1987] war der erste Autor, der Relationen als semantische Konstrukte in objektorientierte Sprachen einführte, so dass Relationen nicht als abgeleitete Klassen, sondern als eigenständiges und gleichberechtigtes sprachliches Mittel die Objektorientierung ergänzen sollten. Er entwickelte dazu eine Programmierumgebung mit dem Namen „Data Structure Manager" (DSM), welche in Anlehnung an die Programmiersprache C objektorientierte Konzepte mit sprachlichen Konstrukten zum Deklarieren und Verwalten von Relationen verbindet [Rumbaugh et al., 1989].

[Albano et al., 1997] erarbeiten auf Basis eines objekt-relationalen Datenmodells eine grundlegende Sprachdefinition, welche die Deklaration von Relationen als eigenständiges sprachliches Konstrukt favorisiert, so dass die Wahl bzgl. der besten Implementierung einer Relation der Datenbank überlassen werden kann.

[Osterbye, 1999] beschreibt einen „Association Compiler" auf Basis von Smalltalk, der es durch die Deklaration einer Relation erlaubt, die Relation sowohl durch ein Relationsobjekt als auch direkt über die mit der Relation verbundenen Klassen zu manipulieren. Er nutzt dabei die Möglichkeit von Smalltalk aus, die notwendigen Methoden zur Manipulation der Relationen in bestehende Klassen dynamisch einzuführen.

[Baldoni et al., 2007a, Baldoni et al., 2007b, Baldoni et al., 2009] setzen die Diskussion zur Einführung von Relationen auf „powerJava" auf, welches Java um ein Rollenkonzept erweitert. Objekte können in unterschiedlichen Rollen interagieren und entsprechend beschrieben werden, was wiederum auf die Deklaration einer Relation zurückwirkt. Eine Person, die eine Universität besucht, kann in Relation zu einer Veranstaltung sowohl in der Rolle als Student als

auch als Tutor auftreten. Eine Relation wird somit als Verknüpfung von aufeinander bezogenen Rollen und nicht mehr als direkte Beziehung zwischen Objekten erfasst.

[Bierman and Wren, 2005] diskutieren auf Basis von „RelJ" (einer Untermenge der Sprachmittel von Java), die sich sehr gut zur Formalisierung und Spracherweiterung nutzen lässt, Relationen als eigenständiges Sprachkonstrukt, das eine eigenständige Vererbungshierarchie mit an Relationen angepasster Semantik erlaubt: abgeleitete Relationen erzeugen automatisch einen Eintrag in der übergeordneten Relation. Wenn also ein Student in einer Veranstaltung ein Referat hält, wird nicht nur ein Eintrag in der veranstaltungsbezogenen Leistungsrelation erzeugt, sondern automatisch auch ein Eintrag für den Besuch der Veranstaltung.

3.6 Objektrelationale Programmierung

[Steimann and Stadtler, 2010b] begründen in ihrer Arbeit, dass aus Sicht der Praxis nur behutsame Erweiterungen des objektorientierten Datenmodells notwendig sind: die Einführung von Relationen als sprachlich eigenständiger Konstrukte sollte insbesondere vor dem Hintergrund kritisch hinterfragt werden, dass man nicht ohne Not auf die effiziente Dereferenzierung von Zeigern und damit der direkten Navigierbarkeit von Zu-1-Beziehungen verzichten sollte.

Für die Autoren liegen die eigentlichen Defizite zum Einen in der notwendigen Unterscheidung und unterschiedlichen Behandlung von Zu-1-Beziehungen und Zu-N-Beziehungen: während Zu-1-Beziehungen durch direkte Verweise sehr effizient implementiert und navigiert werden können, braucht man für Zu-N-Beziehungen einen Umweg über explizite Zwischenobjekte in Form von Kollektionen. Das dahinter liegende Problem wird sichtbar, wenn sich die Kardinalität für eine Beziehung in der Fachdomäne von einer Zu-1-Beziehung auf eine Zu-N-Beziehung ändert: mehr oder weniger umfangreiche Anpassungen im Programmcode sind die Folge. [Steimann and Stadtler, 2010a] schlagen deshalb für die „objektrelationale Programmierung" einen abstrakten Datentyp `Association` vor, der das Zugriffsprotokoll auf eine Assoziation vereinheitlicht und für eine Kardinalität von eins eine spezielle Methode anbietet, die das assoziierte Objekt direkt zurückliefert. Allerdings räumen die Autoren ein, dass

hiermit der fundamentale Unterschied zwischen einer Zu-1-Beziehung und einer Zu-N-Beziehung nicht gänzlich aufgehoben werden kann, da diese spezielle Methode beim Übergang zu einer Zu-N-Beziehung zwangsläufig eine Ausnahme erzeugen muss[4].

Das zweite fundamentale Defizit des objektorientierten Datenmodells sehen die Autoren in der mangelnden Bidirektionalität von Beziehungen, die durch paarige Attribute realisiert werden könnte:

> „Hier die Verantwortung für die Koordinierung paariger Attribute (für die Repräsentation beider Richtungen) in die Hand derjenigen zu geben, die für eine korrekte Umsetzung eines Modells zu sorgen haben, ist der Häufigkeit dieser Aufgabe und ihrer immer wieder gleichen Form nicht angemessen. Statt dessen Relationen einzuführen, denen die Bidirektionalität immanent ist, hätte aber den Preis, auf die einfache Dereferenzierung von Zeigern (s.o.) zu verzichten. Alternativ wäre vielmehr zu bedenken, ob nicht die koordinierte Pflege paariger Attribute Bestandteil des OODM werden sollte." ([Steimann and Stadtler, 2010b])

Der Vorschlag der Object Data Management Group ([Berler et al., 2000, S. 38ff u. S 242ff]) zielt in eine ähnliche Richtung, allerdings auf den Kontext der objektorientierten Persistenzstrategien beschränkt: Relationen sind bidirektionale Verweise, deren unterschiedliche Kardinalitäten mit Referenzen und Kollektionen modelliert werden und die z.B. im Falle der Java-Implementierung auf die explizite Deklaration der Relationen in Property-Dateien zurückgreifen.

```
public class Department {
 DCollection employees;
}

public class Employee {
 Department dept;
```

[4] Letztlich kann hier die Frage gestellt werden, ob es nicht gerechtfertigt wäre, bei einer grundlegenden Änderung des Modells auch Änderungen im Programmcode bzw. den Schnittstellen als notwendige Gegebenheit zu akzeptieren und hierfür auf die Einführung eines zusätzlichen Datentyps zu verzichten. Allerdings sollten die notwendigen Anpassungen im Programmcode bei der Änderung der Kardinalität einer Beziehung mehr semantischer als denn technischer Natur sein, was mit den bis dato zur Verfügung stehenden Sprachmitteln nicht möglich ist.

```
}

;Properties for class Department
class Department field employees refersTo=Employee inverse=dept

;Properties for class Employee
class Employee field dept refersTo=Department inverse=employees
```

Kapitel 4

Bidirektionale Verweise mit Annotationen und Aspekten

4.1 Grundlagen von AspectJ

Um das Verständnis für die folgende Lösung vorzubereiten, werden die für den Zusammenhang wichtigsten Grundlagen der aspektorientierten Programmierung mit AspectJ zusammengefasst: Allgemein gesagt erweitert die Aspektorientierung die Objektorientierung um die Möglichkeit, die Struktur von Programmen quer zur bestehenden Klassenstruktur zu modifizieren. Dies ist insbesondere für die Implementierung von sogenannten „Crosscutting Concerns" wie Logging, Tracing oder Profiling nützlich, kann aber auch ganz allgemein dazu verwendet werden, das Verhalten von ausgewählten Methoden, Aufrufen sowie Zugriffen nach den Wünschen des Entwicklers zu verändern[1]. AspectJ wird sehr detailliert von [Laddad, 2009] vorgestellt und ist die bekannteste Implementierung der Aspektorientierung für Java. AspectJ wird unter dem Dach der Eclipse Foundation entwickelt[2], so dass insbesondere die Integration von AspectJ in Eclipse über die „AspectJ Development Tools" (AJDT) sehr kom-

[1] In bestimmten Kontexten steht die Aspektorientierung in Konkurrenz zur Verwendung von zur Laufzeit generierten Proxy-Objekten, die aber den großen Nachteil haben, dass ein Proxy einen anderen Laufzeittyp als die durch den Proxy vertretene Klasse hat. Dies kann insbesondere im Zusammenspiel mit objektorientierten Persistenzlösungen zum Problem werden, da diese über Reflexion häufig den Laufzeittyp analysieren. Aus diesem Grunde wurde die Idee, Proxies für das Binding von öffentlichen Klassen einzusetzen, auch verworfen.

[2] `http://eclipse.org/aspectj/`

fortabel und für den Entwickler vollkommen transparent ist[3]. Neben AspectJ gibt es auch noch andere aspektorientierte Erweiterungen für Java, wie z.B. Spring-AOP oder JBoss-AOP, deren Einsatzzweck vornehmlich auf die Verwendung in Java-Applikationsservern zugeschnitten ist und deren sprachliche Ausdrucksfähigkeit z.T. etwas eingeschränkter als AspectJ ist. Nichtsdestotrotz zeigt sich daran, dass die Aspektorientierung heute ein fester Bestandteil in der Werkzeugbox vieler Java-Entwickler ist.

AspectJ ist auf den folgenden zentralen Konzepten der Aspektorientierung aufgebaut:

Joinpoints bezeichnen alle Stellen, an denen grundsätzlich in die bestehende Programmstruktur eingegriffen werden kann, wie z.B. die Ausführung von Methoden, Zugriffe auf Datenfelder, die Konstruktion bzw. Initialisierung von Objekten oder das Auslösen von Ausnahmen.

Pointcuts sind die durch den Entwickler zu bestimmenden Joinpoints, deren Verhalten modifiziert werden soll. Während Joinpoints also alle möglichen Einstiegspunkte umfassen, legen Pointcuts fest, welche Joinpoints in einem konkreten Fall genutzt werden sollen. Hierzu wird eine spezielle Syntax für die Angabe von Mustern benutzt, die einzelne oder mehrere Methoden und Felder über deren Signaturen spezifizieren. Zusätzlich können die selektierten Joinpoints durch weitere Bedingungen wie z.B. das Vorhandensein oder Fehlen von Annotationen eingeschränkt werden. Pointcuts sind darüber hinaus in der Lage, den Kontext des Joinpoints in Variablen zu speichern und diesen für die Auswertung bzw. Modifikation durch einen Advice zur Verfügung zu stellen.

Advices legen fest, was am Joinpoint eines Pointcuts passieren soll. Ein Advice enthält den zusätzlichen Java-Code, der vor, nach oder um den Pointcut herum in den vorhandenen Java-Code eingewoben werden soll.

Weaving bezeichnet das Verfahren, wie der Programmcode, der in Advices spezifiziert ist, in die vorhandenen Klassen zum Endresultat eingefügt wird. Neben dem Weaving auf der Ebene des Quelltextes unterstützt AspectJ ebenso das Einweben der Advices in bereits kompilierte Klassen

[3] `http://www.eclipse.org/ajdt/`

(„Binary Weaving") sowie das Einweben der Advices zur Laufzeit („Load-time Weaving") durch das Java VM Tools Interface (JVMTI) bzw. einen dezidierten Classloader.

4.2 Lösungsansatz

Die Grundidee für die Implementierung bidirektionaler Verweise in Domänenmodellen mit Java besteht in einer geschickten Kombination aus Annotationen[4] und aspektorientierter Programmierung auf Basis von AspectJ. Zur Deklaration der jeweils aufeinander bezogenen Verweise werden Annotationen auf Feldebene genutzt, die durch entsprechende Aspekte bei Änderungen ausgewertet und die gebundenen Felder entsprechend automatisch synchronisiert werden. Lesende Zugriffe und damit die Dereferenzierung von Verweisen bleiben auf diese Weise unberührt und effizient.

Der folgende Programmcode zeigt ein Beispiel eines einfachen Domänenmodells, welches im Idealfall so mit der zu entwickelnden Lösung umzusetzen sein soll. Im Beispiel wurden für eine bessere Übersicht nur wenige Akzessoren exemplarisch angegeben, da die Verwendung von Akzessoren identisch mit einer ungebundenen Verwendung sein soll, um den gewohnten Entwicklungsprozess und die vorhandene Werkzeugunterstützung nicht zu beeinträchtigen. Die verwendete Annotation erwartet als Parameter den mit Klassennamen vollständig qualifizierten Feldnamen[5] des bidirektional zu bindenden Feldes. Eine wichtige Anforderung an die Implementierung ist, dass diese die verwendeten Laufzeittypen öffentlicher bzw. selbst definierter Klassen nicht durch die Verwendung von Stellvertreterobjekten (Proxies) verändert, um z.B. nicht mit Persistenzframeworks[6] zu kollidieren, die für Kollektionen auf eigene Implementierungen

[4] Annotationen dienen dazu, Metadaten über den Quelltext direkt in den Quelltext einzubetten. Sie sind in Java seit der Version 5 verfügbar. Annotationen sind mittlerweile in der Java-Welt weit verbreitet und haben in vielen Projekten die Metadatenkonfiguration über XML-Dateien abgelöst.

[5] Ein Feldname ist vollständig qualifiziert, wenn dieser aus dem vollständigen Klassennamen gefolgt von einem Punkt und dann dem einfachen Namen des Feldes besteht (also z.B. `domain.Person.arbeitsplatz` wobei `domain.Person` der vollständige Klassenname und `arbeitsplatz` der Name des Feldes ist).

[6] Als Beispiel seien hier der verbreitete Object-Relational-Mapper „Hibernate" (`http://www.hibernate.org/`) sowie die Java-Objektdatenbank „db4o" (`http://www.db4o.com/`) genannt, die beide eigene Proxy-Implementierungen für persistente Kollektionen nutzen.

zurückgreifen.

```
public class Person {
 @Bind("app.domain.Arbeitsplatz.person")
 private Arbeitsplatz arbeitsplatz;
 @Bind("app.domain.Projekt.personen")
 private Set<Projekt> projekte = new HashSet<Projekt>();

 public void setArbeitsplatz(Arbeitsplatz arbeitsplatz) {
  this.arbeitsplatz = arbeitsplatz;
 }

 public boolean addProjekt(Projekt projekt) {
  return projekte.add(projekt);
 }
 ...
}

public class Arbeitsplatz {
 @Bind("app.domain.Abteilung.arbeitsplaetze")
 private Abteilung abteilung;
 @Bind("app.domain.Person.arbeitsplatz")
 private Person person;
 ...
}

public class Abteilung {
 @Bind("app.domain.Arbeitsplatz.abteilung")
 private Set<Arbeitsplatz> arbeitsplaetze = new
    HashSet<Arbeitsplatz>();
 ...
}

public class Projekt {
 @Bind("app.domain.Person.projekte")
 private Set<Person> personen = new HashSet<Person>();
 ...
}
```

Um in Java bidirektionale Verweise unterschiedlicher Kardinalitäten zu synchronisieren, müssen verschiedene Fälle unterschieden werden:

1-zu-1: Die bidirektionale Umsetzung einer 1-zu-1-Beziehung, also z.B. der

Beziehung von Mitarbeiter und Arbeitsplatz, erfolgt über zwei aufeinander bezogene Referenzen in den beiden beteiligten Klassen. Für die bidirektionale Synchronisation muss bei der Implementierung darauf geachtet werden, dass bei schreibenden Zugriffen der Rückzeiger des zuvor referenzierten Objekts auf `null` gesetzt wird. Wird also einem Arbeitsplatz ein neuer Mitarbeiter zugewiesen, muss die Referenz beim alten Mitarbeiter auf diesen Arbeitsplatz auf `null` gesetzt werden, ansonsten würden zwei Referenzen auf dasselbe Objekt existieren und damit die grundlegende Invariante einer 1-zu-1-Beziehung verletzt werden.

1-zu-N: Die Umsetzung einer 1-zu-N-Beziehung, also z.B. der Beziehung von Abteilung zu Arbeitsplätzen, erfolgt über eine Referenz auf Seiten des Objekts, was dem Einen zugeordnet wird und über eine Kollektion (also einer Liste bzw. einem Set) auf Seiten des Objekts, dem die Vielen zugeordnet werden. Ein Arbeitsplatz erhält also eine Referenz auf eine Abteilung, während eine Abteilung eine Kollektion mit allen zugeordneten Arbeitsplätzen enthält. Bei der Synchronisation muss nun beachtet werden, einen Arbeitsplatz bei der Zuweisung zu einer neuen Abteilung zuerst aus seiner vorhanden Bindung an die alte Abteilung zu lösen (d.h. aus der Kollektion der Arbeitsplätze der alten Abteilung zu entfernen), bevor er der neuen Abteilung zugewiesen und die Referenz vom Arbeitsplatz zur Abteilung aktualisiert werden kann.

N-zu-M: Die Umsetzung einer N-zu-M-Beziehung, also z.B. der Beziehung von Mitarbeitern zu Projekten, erfolgt über Kollektionen in den beiden beteiligten Klassen. Ein Mitarbeiter enthält also eine Kollektion auf seine Projekte sowie jedes Projekt eine Kollektion für die zugeordneten Mitarbeiter. Bei der Synchronisation muss nur darauf geachtet werden, einen Mitarbeiter zur Kollektion der Projekte hinzufügen oder zu entfernen bzw. umgekehrt.

Bei der automatischen Synchronisation der gebundenen Felder müssen aus Sicht von AspectJ die folgenden zwei Fälle unterschieden werden:

- Ist das annotierte Feld eine einfache Referenz auf ein anderes Domänenobjekt, können schreibende Zugriffe auf das Feld über die vorhandenen

Möglichkeiten von AspectJ als Pointcut direkt ermittelt und modifiziert werden.

- Ist das annotierte Feld eine Kollektion, ist es mit AspectJ nicht direkt möglich, alle manipulierenden Operationen der annotierten Kollektion zu ermitteln und zu modifizieren. Es werden deshalb mit AspectJ innerhalb der annotierten Domänenklassen alle Operationen, die eine Kollektion modifizieren, als Pointcut ermittelt und dann mittels Reflexion überprüft, ob die fragliche Kollektion in einem annotierten Feld gespeichert ist. Wenn dies der Fall ist, wird die automatische Synchronisation entsprechend durchgeführt, im anderen Fall werden die Aufrufe ohne Änderung durchgereicht.

Bei der Implementierung der Aspekte muss im Sinne der Vollständigkeit beachtet werden, dass in Java die Schnittstelle `Collection` von `Set, SortedSet und List` erweitert wird. Während für Sets keinerlei besondere Maßnahmen erforderlich sind, da diese in Bezug auf Änderungsoperationen nicht über die von der vererbten Schnittstelle angebotenen Methoden hinausgehen, müssen für Listen zusätzliche Methoden implementiert werden, die über Indizes die Manipulation der Listenelemente ermöglichen.

Darüber hinaus bieten Kollektionen die Erzeugung eines `Iterator` für einen Durchlauf durch die einzelnen Elemente an. Diese Iteratoren sind laut Spezifikation mit der darunter liegenden Kollektion verknüpft und bieten eine Methode zum Entfernen von Elementen aus der Kollektion an. Für diese Iteratoren muss ein entsprechender Proxy implementiert werden, der die Synchronisation übernimmt. Entsprechendes gilt bei Listen für den `ListIterator` bzw. Unterlisten sowie beim `SortedSet` für Untermengen. Eine Verwendung von Proxies ist an dieser Stelle unproblematisch, da die ungebunden gelieferten Iteratoren, Unterlisten bzw. Untermengen auf privaten Klassen beruhen, deren Implementierung sich jederzeit ändern könnte und z.B. von Persistenztechnologien deshalb auch nicht direkt persistiert werden.

4.3 Implementierung

Die folgende Implementierung lässt sich in Annotationen, Java-Klassen und Aspekte unterteilen[7]. Es werden zuerst die benötigten Annotationen `@Bind` und `@Bound` für die Auszeichnung bidirektional zu bindender Felder eingeführt, die selbst mit einer `RetentionPolicy.RUNTIME` annotiert werden, so dass sie für Auswertungen zur Laufzeit der Anwendung zur Verfügung stehen, was später für das Load Time Weaving der Aspekte benötigt wird.

Danach werden die benötigten Java-Klassen vorgestellt, wo neben technischen Hilfsklassen vor allem die Klasse `Binding` sowie die Implementierung von Proxy-Objekten für die bidirektional gebundenen Implementierungen von `Iterator, Listiterator, Set, SortedSet und List` für das das weitere Verständnis wichtig sind. Während die Klasse `Binding` die Logik zur Synchronisation von mengenwertigen Verweisen unterschiedlicher Kardinalitäten kapselt, werden die Proxies für eine vollständige Implementierung benötigt, da laut Java-Spezifikation Kollektionen Sichten in Form von Iteratoren und Teilkollektionen anbieten, welche die Manipulation der zu Grunde liegenden Kollektionen erlauben.

Abschließend werden die Aspekte eingeführt, die über entsprechende Pointcuts die manipulierenden Zugriffe auf annotierte Felder und Kollektionen um entsprechenden Code für die automatische Synchronisation anreichern. Die eigentliche Logik der Aspekte ist dabei durchgängig einfach gehalten, da diese im Wesentlichen nur den Kontext des Aufrufs ermitteln und diesen dann an die zuvor vorgestellte Klasse `Binding` zur eigentlichen Manipulation der Verweise durchreichen.

[7] Der vollständige Programmcode für die im Folgenden vorgestellte Implementierung findet sich auf der beiliegenden CD im Verzeichnis „Bind". Um eine Jar-Datei aus dem Programmcode für die Verwendung in Java-Projekten erzeugen zu können, wird eine Build-Datei auf Basis von Apache Ant genutzt. Die Build-Datei erwartet dabei die Quelltexte im Unterverzeichnis `src`. Für die erfolgreiche Kompilierung werden zwei Bibliotheken aus der AspectJ-Distribution genutzt, die sich im Unterverzeichnis `lib` befinden: während `aspectjtools.jar` für die Integration von AspectJ und Ant verantwortlich ist, wird die Datei `aspectjrt.jar` als Laufzeitumgebung im Klassenpfad für die Kompilierung benötigt. Über den Aufruf von `ant` im Projektverzeichnis wird eine Jar-Datei namens `bind.jar` erstellt, die in Java-Projekte integriert werden kann. Eine umfangreiche Sammlung von Testfällen der entwickelten Bibliothek auf Basis von JUnit (`http://www.junit.org/`) findet sich im Projekt „BindTest" auf der beiliegenden CD und kann mit dem Aufruf `ant` ausgeführt werden.

4.3.1 Annotationen

bind.annotation.Bind

Die grundlegende Annotation auf Feldebene für die Implementierung der bidirektionalen Verweise ist die Annotation `@Bind`, die als Standardargument einen vollständig qualifizierten Feldnamen erwartet, welcher das Feld spezifiziert, welches später mit dem annotierten Feld verbunden werden soll. Da es sich um ein bidirektionales Binding handelt, muss die Annotation `@Bind` grundsätzlich auf beiden Seiten der miteinander zu bindenden Felder angegeben werden, nur jeweils mit dem richtigen Verweis auf das zu bindende Feld in der anderen Klasse.

```
@Target(ElementType.FIELD)
public @interface Bind {
 String value();
}
```

bind.annotation.Bound

Eine zweite Annotation auf Klassenebene wird aus zwei Gründen eingeführt: zum Einen ist es mit AspectJ nicht möglich, Zugriffe auf zu manipulierende Operationen einer Kollektion zielgenau zu selektieren. Ohne diese zusätzliche Annotation auf Klassenebene würden sämtliche Methodenzugriffe aller Kollektionen innerhalb des Programms durch AspectJ modifiziert und geprüft werden müssen, was zu einer unerwünschten Laufzeitverschlechterung führen würde. Der zweite Grund besteht darin, später auf Klassenebene eine Markierung für das zielgenaue Weaving der Aspekte zu besitzen. Jede Klasse, welche die Felder enthält, die bidirektional synchronisiert werden sollen, muss deshalb die Annotation `@Bound` besitzen.

```
@Target(ElementType.TYPE)
public @interface Bound {
}
```

4.3.2 Java-Klassen

bind.java.ConcurrentWeakHashMap

Da sich die vorgestellte Implementierung im Kern ausgiebig der Java-Reflexion bedient (um z.B. gebundene Felder innerhalb einer Klasse zu identifizieren) und die Reflexion eine vergleichsweise langsame Operation ist, werden später mehrere Caches genutzt, die ermittelte statische Beziehungen von Klassen und Feldern zwischenspeichern. Da der Zugriff auf die Caches durch mehrere Threads gleichzeitig erfolgen kann, muss eine thread-sichere Implementierung gewählt werden. Um darüber hinaus Speicherlecks zu vermeiden, ist die Verwendung einer `WeakHashMap` erforderlich, damit Objekte, die grundsätzlich von der Garbage Collection freigegeben werden könnten, weil keine aktiven Referenzen auf diese Objekte mehr bestehen, nicht durch die Speicherung in einem Cache von der Freigabe ausgenommen werden.

Die von Java angebotene Lösung für eine thread-sichere `WeakHashMap` liefert über `Collections.synchronizedMap(new WeakHashMap())` nur eine vollständig synchronisierte `WeakHashMap`, die sämtliche Zugriffe serialisiert. Um den Durchsatz für konkurrierende Zugriffe an dieser Stelle zu erhöhen, bedarf es einer Implementierung ähnlich der in Java 1.5 eingeführten „Concurrency Utilities" (siehe [Goetz et al., 2006]), welche z.B. eine `ConcurrentHashMap` anbieten. Da eine `ConcurrentWeakHashMap` leider nicht in die Standardbibliotheken von Java aufgenommen wurde, wird auf eine Implementierung zurückgegriffen, die im Rahmen der Entwicklung des Java Specification Requests (JSR) unter der Nummer 166 für die „Concurrency Utilities" von Doug Lea entwickelt wurde[8].

bind.java.Key

Um später einen Cache zwischen Objekten und gebundenen Feldern auf Grundlage einer `WeakHashMap` implementieren zu können, wird eine Hilfsklasse zur Bildung eines zusammengesetzten Schlüssels aus zwei Elementen benötigt. Die Klasse speichert die beiden Schlüsselelemente in zwei Feldern und implementiert `Object#equals(Object)` auf Basis eines Identitätsvergleichs der beiden

[8] `http://anonsvn.jboss.org/repos/jbosscache/experimental/jsr166/src/jsr166y/ConcurrentWeakHashMap.java`

Elemente. Für eine effiziente Implementierung von `Object#hashCode()` wird nur der Hashwert des ersten Elements geliefert, der in der Regel durch nativ implementierte Funktionen der Java Virtual Machine direkt aus der Speicheradresse des Objekts ermittelt wird. Hierdurch wird eine zusätzliche und vergleichsweise langsame Berechnung eines neuen Hashwerts vermieden. Unter der Annahme, dass für das erste Element nur wenige zweite Elemente existieren (z.B. gebundene Felder für ein gegebenes Objekt), ist der ohnehin notwendige Identitätsvergleich bei der Verwendung des Schlüssels in einer `WeakHashMap` effizienter als eine nicht-native Berechnung des Hashwerts.

```
public class Key {

 private final Object major;
 private final Object minor;

 public Key(Object major, Object minor) {
  this.major = major;
  this.minor = minor;
 }

 public boolean equals(Object obj) {
  if (this == obj) {
   return true;
  }
  if (obj == null || getClass() != obj.getClass()) {
   return false;
  }
  return major == ((Key) obj).major && minor == ((Key)
     obj).minor;
 }

 public int hashCode() {
  return major.hashCode();
 }
}
```

bind.java.Fields

Für die Implementierung wird eine statische Hilfsklasse mit Methoden für Feldzugriffe über Reflexion eingeführt, die zum Einen die Lesbarkeit des weiteren

Codes erhöht, weil sie die ansonsten bei Feldzugriffen zu fangenden Ausnahmen wie die `NoSuchFieldException` bzw. die `IllegalAccessException` jeweils in eine nicht zu fangende `RuntimeException` übersetzt. Zudem rufen diese Methoden automatisch `Field#setAccessible(true)` auf, so dass die Felder per Reflexion zugreifbar sind. Wenn der Zugriff auf Feldwerte über Reflexion durch eine Sicherheitspolitik global unterbunden sein sollte, würde das implementierte Binding nicht mehr funktionieren. Durch die Zusammenführung sämtlicher Methoden für Feldzugriffe in einer Klasse könnte aber eine Sicherheitspolitik zielgenau diese Klasse als Ausnahme zulassen.

Darüber hinaus implementiert die Klasse eine statische Methode, die ein Feld über einen vollständig qualifizierten Feldnamen ermittelt. Da das Ermitteln des Feldes über Reflexion vergleichsweise langsam ist, wird ein threadsicherer Cache auf Basis einer `ConcurrentWeakHashMap` genutzt, um bei wiederholten Anfragen desselben Feldnamens das resultierende Feld schneller liefern zu können. Da in dem Cache nur Elemente der statischen Programmstruktur gespeichert werden (Feldname, Feld), kann der Cache ohne explizite Verdrängung von veralteten bzw. ungenutzten Elementen realisiert werden.

bind.java.Binding

Die Klasse `Binding` kapselt alle Informationen, die für eine bidirektionale Bindung von einem Gastgeberobjekt (Host) zu einem Gastobjekt (Guest) notwendig sind. Der Gastgeber bezeichnet in diesem Zusammenhang das Objekt, welches die Annotation auf das zu bindende Feld enthält, während der Gast das Objekt ist, welches das zu bindende Feld selbst enthält. Ein Binding kann durch die Angabe eines Gastgeberobjekts und eines vollständig qualifizierten Feldnamens über eine statische Factory-Methode geliefert werden. Dabei wird über den vollständig qualifizierten Feldnamen das Feld in der Gastklasse ermittelt, das vom Gastgeberobjekt gebunden wird. Innerhalb des Bindings werden neben dem Gastgeberobjekt die beiden gebundenen Felder gespeichert.

```
public class Binding {

 private final Object host;
 private final Field hostField; // Field on host pointing to
    guest(s)
 private final boolean hostFieldIsCollection;
```

```
 private final Field guestField; // Field on guest pointing to
    host(s)
 private final boolean guestFieldIsCollection;

 Binding(Object host, String fieldname) {
  this.host = host;
  this.guestField = Fields.getField(fieldname);
  this.guestFieldIsCollection = Collection.class
    .isAssignableFrom(guestField.getType());
  this.hostField =
      Fields.getField(guestField.getAnnotation(Bind.class)
    .value());
  this.hostFieldIsCollection = Collection.class
    .isAssignableFrom(hostField.getType());
 }
}
```

Alternativ kann ein Binding durch die Angabe eines Gastgeberobjekts und einer Kollektion im Gastgeberobjekt über eine zweite statische Factory-Methode geliefert werden. Diese zweite Variante ist notwendig, da AspectJ später beim Abfangen der Methodenzugriffe auf eine gebundene Kollektion nur die konkrete Implementierung der Kollektion zur Laufzeit, nicht aber das Feld in der Klasse liefern kann, worin die Kollektion in der Klasse gespeichert wurde. Das Binding sucht deshalb beim erstmaligen Erzeugen alle Felder in der Gastgeberklasse sowie bei Bedarf deren Oberklassen nach der Annotation `@Bind` ab und prüft, ob in diesem Feld die gegebene Kollektion gespeichert ist, um so das Binding wie oben zu erzeugen.

```
for (Class clazz = host.getClass(); clazz != null; clazz =
    clazz.getSuperclass()) {
 for (Field field : clazz.getDeclaredFields()) {
  Bind bind = field.getAnnotation(Bind.class);
  if (bind != null &&
      Collection.class.isAssignableFrom(field.getType())
    && Fields.getValue(host, field) == collection) {
   return new Binding(host, bind.value());
  }
 }
}
```

Um die aufwändige Erzeugung eines Bindings über Reflexion bei erneuten Zugriffen nicht immer wiederholen zu müssen, wird ein thread-sicherer Cache auf Basis der `ConcurrentWeakHashMap` genutzt, der einen zusammengesetzten Schlüssel aus Gastgeberobjekt und voll qualifiziertem Feldnamen bzw. der gesuchten Kollektion nutzt und dafür das entsprechende Binding liefert, sofern es bereits im Cache vorhanden ist. Die Nutzung einer `ConcurrentWeakHashMap` erfolgt an dieser Stelle, um Objekte, die ansonsten vom Garbage Collector bereinigt werden könnten, nicht durch Referenzen in einem Cache für die Laufzeit der Anwendung im Speicher zu fixieren und so ungewollt ein Speicherleck zu erzeugen.

Das Binding selbst bietet Methoden zum Hinzufügen und Entfernen von Gastobjekten auf dem Gastgeberobjekt an. Beim Hinzufügen wird zunächst geprüft, ob das Gastfeld, also das Feld, das auf dem Gastobjekt zurück zum Gastgeberobjekt verweist, eine Kollektion ist oder nicht. Wenn es sich um eine Kollektion handelt, wird zu dieser das Gastgeberobjekt hinzugefügt. Ist es keine Kollektion, wird der Wert das Gastfeldes ausgelesen, was der vormaligen Referenz auf den Gastgeber bzw. einer Kollektion von Gastgeberobjekten entspricht. Im Falle einer Kollektion wird der aktuelle Gast entfernt, ansonsten wird die Referenz des alten Gastgebers auf das Gastobjekt auf `null` gesetzt. Anschließend wird das Gastfeld mit dem aktuellen Gastgeber gesetzt.

```
public void add(Object guest) {
 if (guest != null) {
  if (guestFieldIsCollection) {
   ((Collection) Fields.getValue(guest, guestField)).add(host);
  } else {
   Object oldHost = Fields.getValue(guest, guestField);
   if (oldHost != null) {
    if (hostFieldIsCollection) {
     ((Collection) Fields.getValue(oldHost, hostField))
       .remove(guest);
    } else {
     Fields.setValue(oldHost, hostField, null);
    }
   }
   Fields.setValue(guest, guestField, host);
  }
 }
```

```
}
```

Der Algorithmus zum Entfernen eines Gastobjekts ist einfacher: wenn das Gastfeld eine Kollektion ist, wird der aktuelle Gastgeber aus dieser Kollektion entfernt, ansonsten das Feld auf `null` gesetzt.

```
public void remove(Object guest) {
 if (guest != null) {
  if (guestFieldIsCollection) {
   ((Collection) Fields.getValue(guest,
       guestField)).remove(host);
  } else {
   Fields.setValue(guest, guestField, null);
  }
 }
}
```

bind.java.BindingIterator

Da Kollektionen die Rückgabe von einem `Iterator` vorsehen, der auf der gegebenen Kollektion operiert und auf dieser Elemente entfernen kann, benötigt man ein Stellvertreterobjekt, das beim Entfernen die entsprechenden Aktualisierungen für das gebundene Feld berücksichtigt. Hierzu wird ein spezieller `BindingIterator` implementiert, der als Parameter ein `Binding` und den ursprünglichen `Iterator` der Liste erwartet. Da es von der API eines Iterators her keine Möglichkeit gibt, das aktuelle Objekt des Iterators geliefert zu bekommen, ohne den Zustand des Iterators zu verändern, wird zusätzlich eine Referenz auf das aktuelle Objekt im Proxy gespeichert.

```
public class BindingIterator implements Iterator {

 private Binding binding;
 private Iterator iterator;
 private Object current;

 public BindingIterator(Binding binding, Iterator iterator) {
  this.iterator = iterator;
  this.binding = binding;
 }

 public boolean hasNext() {
```

```
  return iterator.hasNext();
 }

 public Object next() {
  current = iterator.next();
  return current;
 }

 public void remove() {
  binding.remove(current);
  iterator.remove();
  current = null;
 }

}
```

bind.java.BindingListIterator

Der `BindingListIterator` verhält sich im Grundsatz ähnlich wie der oben beschrieben `BindingIterator`: auch er erwartet ein `Binding` und den originalen `ListIterator` und sorgt dafür, dass gebundene Objekte bei Manipulationen der zu Grunde liegenden Liste entsprechend synchronisiert werden.

bind.java.BindingList

Da die Schnittstelle `List` die Möglichkeit vorsieht, eine Teilliste einer gegebenen Liste zu liefern, benötigt man eine Implementierung für eine Liste, die bei Änderungen das Binding entsprechend berücksichtigt. Zur Konstruktion einer `BindingList` wird ein `Binding` sowie die originale `List` erwartet. Aufrufe zur Manipulation der Liste werden über das Binding entsprechend mit den gebundenen Objekten synchronisiert, bei Aufrufen, die einen Iterator, ListIterator oder wiederum eine Teilliste anfordern, werden entsprechend `BindingIterator`, `BindingListIterator` sowie eine `BindingList` geliefert.

```
public class BindingList implements List {

 private Binding binding;
 private List list;
```

```
 public BindingList(Binding binding, List list) {
  this.binding = binding;
  this.list = list;
 }

 ...

 public Object set(int index, Object element) {
  binding.add(element);
  Object removedElement = list.set(index, element);
  binding.remove(removedElement);
  return removedElement;
 }

 public ListIterator listIterator() {
  return new BindingListIterator(binding, list.listIterator());
 }

 public ListIterator listIterator(int index) {
  return new BindingListIterator(binding,
      list.listIterator(index));
 }

 public List subList(int fromIndex, int toIndex) {
  return new BindingList(binding, list.subList(fromIndex,
      toIndex));
 }

}
```

bind.java.BindingSortedSet

Da das Java Collection Framework neben dem `Set` auch ein `SortedSet` als Schnittstelle anbietet und ein `SortedSet` verschiedene Subsets liefern kann, die auf dem darunter liegenden `Set` operieren, muss ein `BindingSortedSet` implementiert werden, was ein `Binding` sowie das originale `SortedSet` erwartet.

```
public class BindingSortedSet implements SortedSet {

 ...
```

```
 public SortedSet subSet(Object fromElement, Object toElement)
    {
  return new BindingSortedSet(binding,
      sortedSet.subSet(fromElement,
    toElement));
 }

 public SortedSet headSet(Object toElement) {
  return new BindingSortedSet(binding,
      sortedSet.headSet(toElement));
 }

 public SortedSet tailSet(Object fromElement) {
  return new BindingSortedSet(binding,
      sortedSet.tailSet(fromElement));
 }
    ...
}
```

4.3.3 AspectJ

bind.aspectj.BindObjectAspect

Für das Binding einer 1-zu-1-Beziehung wird ein `BindObjectAspect` eingeführt. Der Aspekt spezifiziert als Joinpoint alle Feldzugriffe mit der Annotation `@Bind`, die keine Kollektionen oder hiervon abgeleitete Klassen sind und sich innerhalb einer Klasse befinden, die mit der Annotation `@Bound` annotiert wurde[9]. Als Kontext werden das Gastgeberobjekt, die Annotation `@Bind` sowie der zu setzende Wert geliefert. Der Code erzeugt ein `Binding` und delegiert den Aufruf zur Aktualisierung an diesen, bevor die Abarbeitung des ursprünglichen Codes von AspectJ fortgesetzt wird.

```
public aspect BindObjectAspect {

 before(Object host, Bind bind, Object guest)
  : set(@Bind !java.util.Collection+ *.*)
```

[9]Die zusätzliche Selektion über die Annotation `@Bound` ist streng betrachtet für diesen Aspekt nicht unbedingt nötig, wurde aber für eine durchgängig einheitliche Formulierung aller Pointcuts aufgenommen.

```
   && this(host) && args(guest) && @annotation(bind) &&
       @within(Bound)
 {
  Binding binding = Binding.getInstance(host, bind.value());
  if (binding != null) {
   binding.clear();
   binding.add(guest);
  }
 }
}
```

Betrachtet man nun die Übersetzung des oben aufgeführten Codes auf Ebene des Bytecodes, sieht man, dass AspectJ die Ermittlung des annotierten Feldes mit Hilfe der Java-Reflexion durchführt, ohne dabei einen Cache zu nutzen. Für eine effizientere Implementierung wurde deshalb der Zugriff auf das annotierte Feld mit einem Cache für den Wert der Annotation explizit ausprogrammiert, was zu einer deutlichen Geschwindigkeitssteigerung führt. Da AspectJ mit neueren Versionen regelmäßig neue Optimierungen enthält, kann möglicherweise in einer zukünftigen Version auf die Variante mit einem eigenem Cache verzichtet werden, da AspectJ bereits jetzt selbst an vielen Stellen für Zugriffe auf Felder über Reflexion auf ein eigenes Caching zurückgreift.

```
public aspect BindObjectAspect {

 private static final Map<Field, String> bindValueByField =
    new ConcurrentWeakHashMap<Field, String>();

 before(Object host, Object guest)
  : set(@Bind !java.util.Collection+ *.*)
   && this(host) && args(guest) && @within(Bound)
 {
  Field field = ((FieldSignature)
     thisJoinPointStaticPart.getSignature())
    .getField();
  String bindValue = bindValueByField.get(field);
  if (bindValue == null) {
   bindValue = field.getAnnotation(Bind.class).value();
   bindValueByField.put(field, bindValue);
  }
  Binding binding = Binding.getInstance(host, bindValue);
  if (binding != null) {
```

```
    binding.clear();
    binding.add(guest);
   }
  }

}
```

bind.aspectj.BindCollectionAspect

Für das Binding von Kollektionen oder hiervon abgeleiteten Klassen wird der `BindCollectionAspect` eingeführt. Es wird für jede Methode, die die Schnittstelle `Collection` zur Manipulation anbietet, ein eigener Pointcut erstellt. Als Kontext wird das aktuelle Gastgeberobjekt, die jeweilige Kollektion und ein oder mehrere Elemente, die zur Kollektion hinzugefügt bzw. entfernt werden sollen, in den Advice weiter gereicht. Jeder Pointcut wird durch die Angabe der Annotation `@Bound` eingegrenzt, da ansonsten alle Operationen auf Kollektionen durch AspectJ angereichert würden.

Die Logik für die einzelnen Methoden ist in der Regel immer ähnlich: ein `Binding` wird aus Gastgeberobjekt und Kollektion erzeugt und das jeweilige Element hinzugefügt bzw. entfernt, bevor an die ursprüngliche Methodenausführung delegiert wird. Bei der Rückgabe eines Iterators wird ein `BindingIterator` mit dem originalen Iterator instantiiert. Da der Aspekt wegen der Einschränkung von AspectJ, Methodenaufrufe auf annotierten Kollektionen nicht direkt identifizieren zu können, auch ungebundene Kollektionen in Klassen mit der Annotation `@Bound` modifiziert, muss stets mit einem Nullcheck überprüft werden, ob überhaupt ein `Binding` vorhanden ist.

```
public aspect BindCollectionAspect {

 before(Object host, Collection collection, Object guest)
  : call(boolean java.util.Collection+.add(Object))
   && this(host) && target(collection) && args(guest) &&
      @within(Bound)
 {
  Binding binding = Binding.getInstance(host, collection);
  if (binding != null) {
   binding.add(guest);
  }
```

```
 }

    ...

 Iterator around(Object host, Collection collection)
  : call(Iterator java.util.Collection+.iterator())
   && this(host) && target(collection) && @within(Bound)
 {
  Binding binding = Binding.getInstance(host, collection);
  if (binding != null) {
   return new BindingIterator(binding, proceed(host,
       collection));
  } else {
   return proceed(host, collection);
  }
 }

}
```

bind.aspectj.BindListAspect

Für das Binding von Listen wird der `BindListAspect` erzeugt. Dieser arbeitet ähnlich wie der `BindCollectionAspect`, nur widmet er sich allen Operationen, die Veränderungen an Listen durchführen und über die bereits im `BindCollectionAspect` behandelten Methoden hinausgehen. Exemplarisch sei hier die Methode `List#set(int, Object)` angeführt.

```
public aspect BindListAspect {

 Object around(Object host, List list, Object guest, int index)
  : call(Object java.util.List+.set(int, Object))
   &&  this(host) && target(list) && args(index, guest) &&
       @within(Bound)
 {
  Binding binding = Binding.getInstance(host, list);
  Object removedGuest = proceed(host, list, guest, index);
  if (binding != null) {
   binding.remove(removedGuest);
   binding.add(guest);
  }
  return removedGuest;
```

```
 }
    ...
}
```

bind.aspectj.BindSortedSetAspect

Das Binding von SortedSets wird über den `BindSortedSetAspect` realisiert. Dieser liefert für alle Aufrufe, die eine Untermenge bzw. einen Ausschnitt eines SortedSet zurückgeben, ein `BindingSortedSet`, das mit dem originalen SortedSet instantiiert wurde.

```
public aspect BindSortedSetAspect {

 SortedSet around(Object host, SortedSet sortedSet)
  : call(SortedSet java.util.SortedSet+.subSet(Object, Object))
   && this(host) && target(sortedSet) && @within(Bound)
 {
  Binding binding = Binding.getInstance(host, sortedSet);
  if (binding != null) {
   return new BindingSortedSet(binding, proceed(host,
       sortedSet));
  } else {
   return proceed(host, sortedSet);
  }
 }
 ...
}
```

META-INF/aop.xml

Um aus den oben genannten Aspekten am Ende eine in sich geschlossene Jar-Datei für die Verwendung in Java-Projekten erzeugen zu können, werden noch Metadaten für das spätere Weaving durch AspectJ benötigt. Es muss deshalb eine Datei namens `aop.xml` im Verzeichnis `META-INF` des Projekts erstellt werden. Diese Datei deklariert zum Einen die Aspekte, zum Anderen spezifiziert sie alle Klassen für das Weaving, die durch die Annotation `@Bound` gekennzeichnet wurden.

```
<aspectj>
 <aspects>
```

```
  <aspect name="bind.aspectj.BindObjectAspect" />
  <aspect name="bind.aspectj.BindCollectionAspect" />
  <aspect name="bind.aspectj.BindListAspect" />
  <aspect name="bind.aspectj.BindSortedSetAspect" />
 </aspects>
 <weaver options="-verbose␣-nowarn">
  <include within="@bind.annotation.Bound␣*"/>
 </weaver>
</aspectj>
```

Kapitel 5

Ergebnisse

Um die im vorigen Kapitel erstellte Bibliothek in einem regulären Java-Projekt zur Entwicklung eines Domänenmodells mit bidirektionalen Verweisen nutzen zu können, muss die Datei `bind.jar` in den Klassenpfad der zu erstellenden Java-Anwendung aufgenommen werden. Zusätzlich muss beim Start der Anwendung das Load Time Weaving von AspectJ über das Java VM Tools Interface aktiviert werden, damit die annotierten Domänenklassen durch AspectJ beim erstmaligen Laden durch den Classloader instrumentiert werden können. Dies geschieht über den Aufruf `java -javaagent:lib/aspectjweaver.jar`, wobei eine spezielle Jar-Datei zum Weaving der Aspekte samt zugehöriger Laufzeitumgebung für AspectJ beim Aufruf des Java-Interpreters eingebunden wird. Die hierfür benötigte Bibliothek namens `aspectjweaver.jar` befindet sich in der AspectJ-Distribution. Weitere Abhängigkeiten bestehen bei der Verwendung der erstellten Bibliothek nicht.

5.1 Implementierungsaufwand

Zur Evaluation der Bibliothek soll ein Vergleich zu einer manuellen Implementierung bidirektionaler Beziehungen an einem vereinfachten Domänenmodell durchgeführt werden[1]. Für das Domänenmodell werden zuerst Schnittstellen definiert, die anschließend einmal manuell und einmal mit Unterstützung von

[1] Das vollständige und ausführbare Projekt mit sämtlichen Quellcodes findet sich auf der beiliegenden CD im Verzeichnis „BindEvaluation". Ein Evaluationslauf kann durch Aufruf von `ant` im Projektverzeichnis gestartet werden.

Annotationen implementiert werden. Das Domänenmodell besteht aus Personen, Arbeitsplätzen, Abteilungen und Projekten. Zwischen Personen und Arbeitsplätzen besteht eine 1-zu-1-Beziehung, zwischen Abteilungen und Arbeitsplätzen eine 1-zu-N-Beziehung und zwischen Personen und Projekten eine N-zu-M-Beziehung.

```
public interface Abteilung {
 boolean addArbeitsplatz(Arbeitsplatz arbeitsplatz);
 boolean removeArbeitsplatz(Arbeitsplatz arbeitsplatz);
 boolean containsArbeitsplatz(Arbeitsplatz arbeitsplatz);
}

public interface Arbeitsplatz {
 Person getPerson();
 void setPerson(Person person);
 Abteilung getAbteilung();
 void setAbteilung(Abteilung abteilung);
}

public interface Person {
 Arbeitsplatz getArbeitsplatz();
 void setArbeitsplatz(Arbeitsplatz arbeitsplatz);
 boolean addProjekt(Projekt projekt);
 boolean removeProjekt(Projekt projekt);
 boolean containsProjekt(Projekt projekt);
}

public interface Projekt {
 boolean addPerson(Person person);
 boolean removePerson(Person person);
 boolean containsPerson(Person person);
}
```

Bei der manuellen Implementierung müssen je nach Kardinalität der zu implementierenden Beziehung unterschiedliche Vorgehensweisen genutzt werden, je nachdem, ob es sich um eine Zu-1- bzw. Zu-N-Beziehung handelt. Beispielhaft für eine 1-zu-1-Beziehung sei die Implementierung der Beziehung zwischen Personen und Arbeitsplätzen auf Seiten der Person vorgestellt. Zur Vereinfachung der manuellen Implementierung wurden Felder mit Sichtbarkeit innerhalb desselben Pakets benutzt, um so direkt die Felder manipulieren zu können. Bei einer gekapselten Verwendung über Akzessoren müssten für den schreibenden

Zugriff jeweils zwei Methoden angeboten werden: eine, die ausschließlich das Feld beschreibt und eine, die zusätzlich die Synchronisation mit dem zu bindenden Feld durchführt. Der Aufwand für die Implementierung wäre in diesem Fall also noch einmal höher als bei der vorgestellten Variante.

```
public class PersonManuell implements Person {

 Arbeitsplatz arbeitsplatz;

 public void setArbeitsplatz(Arbeitsplatz arbeitsplatz) {
  if (arbeitsplatz != null
    && ((ArbeitsplatzManuell) arbeitsplatz).person != null) {
   ((PersonManuell) ((ArbeitsplatzManuell)
      arbeitsplatz).person).arbeitsplatz = null;
  }
  if (this.arbeitsplatz != null) {
   ((ArbeitsplatzManuell) this.arbeitsplatz).person = null;
  }
  this.arbeitsplatz = arbeitsplatz;
  if (this.arbeitsplatz != null) {
   ((ArbeitsplatzManuell) this.arbeitsplatz).person = this;
  }
 }
}
```

Bei der analogen Implementierung mit Hilfe der Annotationen `@Bound` und `@Bind` muss der Entwickler in der Personenklasse keine Annahme über die Kardinalität der Beziehung vorwegnehmen und diese codieren. Die Kardinalität ergibt sich automatisch durch das gebundene Feld: ist das gebundene Feld eine Kollektion, wird zur Laufzeit eine Zu-N-Beziehung, im anderen Fall eine Zu-1-Beziehung etabliert. Bei der Implementierung ist anzumerken, dass die Akzessoren für die Feldzugriffe bei der Implementierung mit Hilfe der Annotationen keinerlei Besonderheiten aufweisen und so direkt durch die Werkzeugunterstützung einer Entwicklungsumgebung erzeugt werden können.

```
@Bound
public class PersonAnnotiert implements Person {

 @Bind("domain.ArbeitsplatzAnnotiert.person")
 private Arbeitsplatz arbeitsplatz;
```

```
 public void setArbeitsplatz(Arbeitsplatz arbeitsplatz) {
  this.arbeitsplatz = arbeitsplatz;
 }
}
```

Als Beispiel für eine 1-zu-N-Beziehung soll die Beziehung zwischen Arbeitsplatz und Abteilung dienen. Man vergleiche insbesondere die Implementierung der Zugriffsfunktion in der Arbeitsplatzklasse mit der oben angeführten Implementierung für die Zugriffsfunktion in der Personenklasse. In Abhängigkeit von der unterschiedlichen Kardinalität fallen diese trotz analoger Methodensignatur unterschiedlich aus.

```
public class ArbeitsplatzManuell implements Arbeitsplatz {

 Abteilung abteilung;

 public void setAbteilung(Abteilung abteilung) {
  if (this.abteilung != null) {
   ((AbteilungManuell)
       this.abteilung).arbeitsplaetze.remove(this);
  }
  this.abteilung = abteilung;
  if (this.abteilung != null) {
   ((AbteilungManuell)
       this.abteilung).arbeitsplaetze.add(this);
  }
 }
}

public class AbteilungManuell implements Abteilung {

 Set<Arbeitsplatz> arbeitsplaetze = new HashSet();

 public boolean addArbeitsplatz(Arbeitsplatz arbeitsplatz) {
  if (arbeitsplatz == null) {
   return false;
  }
  if (((ArbeitsplatzManuell) arbeitsplatz).abteilung != null) {
   ((AbteilungManuell) ((ArbeitsplatzManuell)
       arbeitsplatz).abteilung).arbeitsplaetze
      .remove(arbeitsplatz);
  }
```

```
  boolean result = this.arbeitsplaetze.add(arbeitsplatz);
  ((ArbeitsplatzManuell) arbeitsplatz).abteilung = this;
  return result;
 }
}
```

Die Implementierung mit Hilfe der Annotationen enthält wiederum keine nennenswerten Besonderheiten. Der Aufruf zur Veränderung der verwendeten Kollektion wird einfach an diese delegiert.

```
@Bound
public class ArbeitsplatzAnnotiert implements Arbeitsplatz {

 @Bind("domain.AbteilungAnnotiert.arbeitsplaetze")
 private Abteilung abteilung;

 public void setAbteilung(Abteilung abteilung) {
  this.abteilung = abteilung;
 }
}

@Bound
public class AbteilungAnnotiert implements Abteilung {

 @Bind("domain.ArbeitsplatzAnnotiert.abteilung")
 private Set<Arbeitsplatz> arbeitsplaetze = new HashSet();

 public boolean addArbeitsplatz(Arbeitsplatz arbeitsplatz) {
  return arbeitsplaetze.add(arbeitsplatz);
 }
}
```

Abschließend soll noch der Fall der N-zu-M-Beziehung am Beispiel von Personen und Projekten illustriert werden. In diesem Fall ist auch die manuelle Implementierung sehr übersichtlich:

```
public class PersonManuell implements Person {

 Set<Projekt> projekte = new HashSet();

 public boolean addProjekt(Projekt projekt) {
  ((ProjektManuell) projekt).personen.add(this);
  return projekte.add(projekt);
```

```
 }
}
```

Die über die Annotationen automatisch gebundene Implementierung sieht im Vergleich ähnlich aus:

```
@Bound
public class PersonAnnotiert implements Person {

 @Bind("domain.ProjektAnnotiert.personen")
 private Set<Projekt> projekte = new HashSet();

 public boolean addProjekt(Projekt projekt) {
  return projekte.add(projekt);
 }
}
```

Der Implementierungsaufwand ist bei der manuellen Implementierung in Abhängigkeit von der realisierten Kardinalität durchgängig größer als bei der Benutzung der Annotationen. Wenn man die Programmzeilen für die modifizierenden Methoden `#set()`, `#add()` und `#remove()` der vorliegenden Implementierungen für das Domänenmodell auszählt, so ergeben sich neben der absoluten Anzahl an Programmzeilen folgende Faktoren, die angeben, um wie viele Zeilen Programmcode eine manuelle Implementierung pro zusätzlich realisierter Beziehung im Vergleich zu einer Implementierung mit Hilfe der Annotationen anwächst.

	Zeilen (Manuell)	Zeilen (Annotiert)	Faktor
1-1	26	10	2.6
1-N	30	10	3.0
N-M	18	16	1.1

5.2 Laufzeitkosten

Neben dem Implementierungsaufwand sind die Laufzeitkosten zu berücksichtigen. Da die Implementierung mit den Annotationen im Hintergrund auf Reflexion zum Zugriff auf die bidirektional zu bindenden Felder zurückgreifen muss, ist zu erwarten, dass das Laufzeitverhalten bei Schreiboperationen im Vergleich zur manuellen Implementierung weniger performant ist. Um hierfür

eine Größenordnung zu bestimmen, wurden verschiedene Testfälle erstellt, die typische Schreiboperationen auf dem Domänenmodell mit den beiden Varianten der Domänenklassen ausführen und deren jeweilige Laufzeit gemessen wird. Im Folgenden ist ein Auszug aus den verschiedenen Testfällen aufgeführt. Für die 1-zu-N-Beziehung und die N-zu-1-Beziehung gibt es zwei unterschiedliche Testfälle, da unterschiedliche Codeverläufe aufgerufen werden, je nachdem über welche Seite der Beziehung eine Änderung erfolgt.

```
person.setArbeitsplatz(arbeitsplatz);
assert person.getArbeitsplatz() == arbeitsplatz;
assert arbeitsplatz.getPerson() == person;
...
abteilung.addArbeitsplatz(arbeitsplatz);
assert abteilung.containsArbeitsplatz(arbeitsplatz);
assert arbeitsplatz.getAbteilung() == abteilung;
...
arbeitsplatz.setAbteilung(abteilung);
assert abteilung.containsArbeitsplatz(arbeitsplatz);
assert arbeitsplatz.getAbteilung() == abteilung;
...
projekt.addPerson(person);
assert person.containsProjekt(projekt);
assert projekt.containsPerson(person);
```

Zur Durchführung der Auswertung werden die Testfälle jeweils mit den beiden Varianten der Domänenklassen instantiiert und dann die aufgeführten Operationen jeweils 10.000.000 mal wiederholt und die dafür notwendige Zeit gemessen. Dieser gesamte Vorgang wird wiederum in 5 Aufwärmläufen und einem abschließenden Evaluationslauf wiederholt, so dass sämtliche Optimierungen der Java HotSpot-Engine abgeschlossen sind und die gelieferten Werte der einzelnen Durchläufe nur noch geringe Abweichungen voneinander aufweisen[2]. Die Testläufe wurden jeweils mit dem OpenJDK 1.6 unter Linux auf verschiedenen Rechnern durchgeführt.

[2]Bei den vorliegenden Messungen geht es um die Ermittlung von Größenordnungen, so dass kein Anspruch auf exakte und allgemeingültige Werte insbesondere in den Nachkommastellen angestrebt wird.

	Taktung
Intel Atom	1.60 GHz
Intel Core Duo	1.80 GHz
Intel Xeon	2.27 GHz
Intel Core i5	2.53 GHz

Als Implementierung für die HotSpot-Engine wurde die Server-Variante gewählt, die eine bessere Optimierung bei der Übersetzung des Java-Bytecodes in Maschinencode erzeugt, dafür aber länger braucht, bis mögliche Optimierungen erkannt, analysiert und durchgeführt sind. Im Bereich serverseitiger, lang laufender Geschäftsanwendungen ist dies aber der Regelfall. Als Ergebnis zeigen sich die folgenden Faktoren, die jeweils angeben, wie viel schneller die manuelle Implementierung im Vergleich zur Lösung mit Hilfe der Annotationen bei der Aktualisierung verschiedener Kardinalitäten ist.

1-1	Manuell	Annotiert	Faktor
Intel Atom	1393ms	21517ms	15.4
Intel Core Duo	451ms	6533ms	14.5
Intel Xeon	354ms	5036ms	14.2
Intel Core i5	288ms	4045ms	14.0

1-N	Manuell	Annotiert	Faktor
Intel Atom	7392ms	19393ms	2.6
Intel Core Duo	4276ms	8077ms	1.9
Intel Xeon	2207ms	4532ms	2.1
Intel Core i5	1956ms	4028ms	2.1

N-1	Manuell	Annotiert	Faktor
Intel Atom	7320ms	27899ms	3.8
Intel Core Duo	4156ms	10432ms	2.5
Intel Xeon	2016ms	6381ms	3.2
Intel Core i5	1788ms	5496ms	3.1

N-M	Manuell	Annotiert	Faktor
Intel Atom	12502ms	26737ms	2.1
Intel Core Duo	6774ms	11154ms	1.6
Intel Xeon	3372ms	7146ms	2.1
Intel Core i5	2779ms	5367ms	1.9

5.3 Bewertung

Der Vergleich der beiden Implementierungen zeigt bei der Entwicklung von Domänenmodellen mit Java den großen Vorteil der Annotationen: der Entwickler braucht zum Erzeugen bidirektionaler Verweise nur ein Paar von Annotationen auf die jeweils miteinander zu verbindenden Felder zu setzen. Er kann dabei durchgängig die gebräuchlichen Idiome der Java-Entwicklung in Form von Referenzen und Kollektionen für die Erzeugung von Verweisen nutzen, so dass auch kein zusätzlicher Lern- bzw. Einarbeitungsaufwand für eine neue Syntax wie bei einer Spracherweiterung bzw. die korrekte Nutzung einer weiteren Bibliothek anfällt. Der Entwickler ist zudem frei darin, Verweise im Domänenmodell ungebunden sowie gebunden in ein- und derselben Klasse zu verwenden und kann so je nach Anforderung des zu entwickelnden Domänenmodells entscheiden, ob eine bidirektionale Bindung genutzt werden soll oder nicht. Die Lösung mit den Annotationen kann sogar in den Fällen genutzt werden, wenn kein Zugriff auf den Quelltext einer Klasse bestehen sollte und somit keine Annotationen direkt angebracht werden können: mit Hilfe von AspectJ können die entsprechenden Annotationen auf das gewünschte Feld injiziert werden[3].

Die Nutzung der Annotationen erleichtert auch den Wechsel der Kardinalität: wenn z.B. die Beziehung zwischen Person und Arbeitsplatz nicht mehr eine 1-zu-1-Beziehung sein soll, sondern ein Arbeitsplatz im Sinne des Job-Sharings auch von mehreren Personen besetzt werden kann (also die Beziehung zwischen Arbeitsplatz und Person eine 1-zu-N-Beziehung wird), müsste nur die Arbeitsplatzklasse sowie deren Schnittstelle entsprechend angepasst werden, während die Personenklasse unverändert weiter genutzt werden kann. Bei der manuellen Implementierung müssten neben den Schnittstellen die Personen- sowie die Arbeitsplatzklassen umfangreich angepasst werden. Vor der Änderung der

[3] siehe hierzu [Laddad, 2009, S. 130ff]

Kardinalität sieht die Arbeitsplatzklasse mit Hilfe der Annotationen so aus:

```
public class Arbeitsplatz {
 @Bind("domain.Person.arbeitsplatz")
 private Person person;

 public void setPerson(Person person) {
  this.person = person;
 }
}
```

Wird nun die Kardinalität von einer 1-zu-1-Beziehung auf eine 1-zu-N-Beziehung geändert, muss die Referenz in eine entsprechende Kollektion samt zugehöriger Akzessoren abgeändert werden.

```
public class Arbeitsplatz {
 @Bind("domain.Person.arbeitsplatz")
 private Set<Person> personen = new HashSet();

 public void addPerson(Person person) {
  this.personen.add(person);
 }
}
```

Ein weiterer Vorteil der vorgestellten Implementierung mit den Annotationen ist die Tatsache, dass durch die Verwendung von AspectJ keine Verwendung von Proxy-Objekten zur Erzeugung von Referenzen und gebundener Kollektionen nötig ist. Mit Hilfe von AspectJ können die vorhandenen Implementierungen über deren Schnittstellen direkt modifiziert werden. Dies ist insbesondere bei der Verwendung mit Persistenzframeworks oder Objektdatenbanken ein großer Gewinn, da diese in der Regel entweder spezielle Annahmen über den verwendeten Laufzeittyp der Kollektionen machen bzw. selbst Proxy-Klassen zur Realisierung eines Lazy-Loadings implementieren. Zudem muss vorhandener Java-Code, der den Laufzeittyp von Objekten bzw. Kollektionen über `instanceof` vergleicht, nicht modifiziert werden.

Den Vorteilen stehen auf der anderen Seite aber auch Nachteile gegenüber: die Annotation `@Bind` nutzt einen `String` zur Identifizierung des zu bindenden Feldes und ist somit nicht sicher refaktorisierbar. Bei einer Umbenennung eines Feldes kann eine Entwicklungsumgebung nicht automatisch erkennen, dass es sich in der Annotation um das geänderte Feld handelt. Das Problem, dass nun

die Annotation auf ein nicht mehr vorhandenes Feld verweist, wird erst zur Laufzeit entdeckt, wenn dieser Fall nicht durch Unit-Tests abgedeckt würde. Grundsätzliche Abhilfe für dieses Problem könnte durch ein entsprechend zu entwickelndes Refaktorisierungsplugin für die verwendeten Entwicklungsumgebungen geschaffen werden.

Desweiteren ist die Performanz bei der Verwendung der Annotationen nicht so gut wie bei einer rein compilerbasierten Lösung, die durch die manuelle Implementierung, die Verwendung einer Klassenbibliothek oder einer Spracherweiterung möglich wäre: durch die Benutzung der Java Reflexion geht eine Performanzeinbuße bei der Änderung von Verweisen in Höhe einer Größenordnung im Vergleich zur manuellen (und damit effizientesten) Implementierung einher. Während die Implementierung mit Annotationen bei der Verwendung von Kollektionen nur um den Faktor 2-3 langsamer ist, wird der Unterschied bei einer 1-zu-1-Beziehung deutlicher, da hier eine Einbuße um den Faktor 15 einher geht. Allerdings sollte diese Einbuße bei der Implementierung von Domänenmodellen in realen Projekten richtig eingeschätzt werden: da die Domänenmodelle in der Regel in einer Datenbank persistiert werden, sind die Zugriffszeiten bei Änderungen für die Kommunikation mit der Datenbank (Zugriff auf Netzwerk und Festplatten) um Größenordnungen höher. Darüber hinaus gibt es in Geschäftsanwendungen vielfach mehr lesende als schreibende Operationen. Lesende Operationen sind mit der Bibliothek genauso effizient wie ohne, so dass die schreibenden Operationen möglicherweise nicht nennenswert ins Gewicht fallen. Falls diese Einbußen für einzelne Beziehungen bei schreibenden Zugriffen jedoch nicht tolerierbar sein sollten, könnte nach dem Prototyping der Anwendung mit Hilfe der Annotationen eine manuelle Implementierung für kritische Abschnitte erfolgen.

Der größte Nachteil der vorgestellten Implementierung ist das notwendige Weaving der Aspekte. Während für ein Build-Time-Weaving die Entwicklungsumgebung oder der Build-Prozess um AspectJ erweitert werden müsste, muss für das Load Time Weaving der Start des Java-Interpreters über die zusätzliche Angabe eine Java-Agents erfolgen. Dies ist nicht immer möglich bzw. gewünscht.

Kapitel 6

Schlussbetrachtung

6.1 Zusammenfassung

Die mangelnde Existenz von bidirektionalen Verweisen in der objektorientierten Programmierung ist ein großes Manko im Vergleich zur Welt der relationalen Datenbanken. Verweise von einem Objekt auf ein anderes Objekt sind stets gerichtet, während eine Relation immer in beide Richtungen abgefragt werden kann. Um diesem Defizit zu begegnen, gibt es eine anhaltende wissenschaftliche Diskussion, die verschiedene Lösungsansätze ergeben hat:

- Patterns für die manuelle Implementierung von Relationen;
- Codegeneratoren zur automatisierten Erstellung von Relationen aus Modellen;
- Bibliotheken, die Relationen als dezidierte Klassen oder Aspekte implementieren;
- verschiedene Formen der Spracherweiterung.

Gemeinsam ist allen Ansätzen, dass sie vom Entwickler die Benutzung neuer Werkzeuge oder das Erlernen neuer programmiersprachlicher Ausdrucksformen verlangen. Die gewohnte und effiziente Verwendung von Referenzen und Kollektionen zur Realisierung von Verweisen wird nicht unterstützt.

[Steimann and Stadtler, 2010a, Steimann and Stadtler, 2010b] weisen deutlich auf das pragmatische Defizit der wissenschaftlichen Diskussion hin und

entwickeln das Konzept der „objektrelationalen Programmierung" mit dem Anspruch einer „sanften Relationalisierung" der Objektorientierung.

Die in dieser Arbeit vorgestellte Lösung versucht dem Anspruch der objektrelationalen Programmierung für bidirektionale, mengenwertige Verweise dadurch gerecht zu werden, dass auf Basis einfacher Annotationen beliebige Felder auf Klassenebene bidirektional miteinander verbunden werden können. Der Entwickler muss dazu nur ein Paar von Feldern mit dem jeweils gegenüber liegenden voll qualifiziertem Feldnamen annotieren, um auf Basis von gewöhnlichen Referenzen und Kollektionen bidirektionale Verweise zu erhalten. Als Werkzeug für die Realisierung wurde die aspektorientierte Programmierung in Form von AspectJ benutzt. Das Ergebnis ist eine Bibliothek, die auf einfache Weise in Java-Projekte eingebunden werden kann und für den Start einer Anwendung nur eine zusätzliche Konfiguration für das Load Time Weaving der Aspekte erfordert.

Während der Aufwand für die Implementierung bidirektionaler Verweise mit Hilfe der Annotationen deutlich sinkt, verschlechtert sich die Performanz bei schreibenden Zugriffen im Vergleich zu einer manuell implementierten Lösung: Beziehungen, die auf einer Seite mindestens eine Kollektion enthalten, sind um den Faktor 2-3 langsamer, 1-zu-1-Beziehungen um den Faktor 15. Ursache hierfür ist hauptsächlich die Verwendung der Java Reflexion für die Ermittlung der Felder innerhalb der Klassen und die lesenden und schreibenden Zugriffe auf Felder, die durch die Java Virtual Machine nicht so effizient optimiert werden kann wie die direkte Manipulation von Referenzen. Diese Performanzeinbuße muss aber in der Praxis kein Nachteil sein, weil die erreichte Performanz immer noch sehr hoch und z.B. im Vergleich zu Datenbankoperationen um Größenordnungen besser ist. Zudem gibt es bei lesenden Zugriffen keinerlei Einbußen.

6.2 Ausblick

Die Arbeit an der vorgestellten Bibliothek kann in verschiedene Richtungen fortgeführt werden:

- Die vorgestellte Implementierung erwartet als Feldtypen für gebundene Kollektionen ausschließlich die Schnittstellen des Java Collection Frame-

works, also `Collection`, `List`, `Set` und `SortedSet`. Um die Flexibilität der Implementierung noch weiter zu erhöhen und auch konkrete Implementierungen von Kollektionen als Feldtypen verwenden zu können, müssten die konkreten Implementierungen von Java Kollektionen wie z.B. `ArrayList`, `LinkedList`, `Stack` oder `TreeSet` durch entsprechende Aspekte für ein Binding angereichert werden. Dazu müssen alle Methoden durch Aspekte modifiziert werden, die über die bereits auf Ebene der Schnittstelle deklarierten Methoden eine Veränderung der Kollektion zur Folge haben.

- Desweiteren wäre das Zusammenspiel der vorgestellten Bibliotheken mit gängigen Persistenztechnologien wie Object-Relational-Mappern sowie Objektdatenbanken zu testen und zu optimieren.

- Darüber hinaus wäre zu untersuchen, wie die Bibliothek für eine reibungslose Verwendung in Applikationsservern angepasst werden müsste, da Applikationsserver verschiedene Anwendungen über eine komplexe Hierarchie von Classloadern separieren und die Bibliothek bereits zu einem sehr frühen Zeitpunkt über das Java VM Tools Interface geladen wird. Als problematisch könnte sich hierbei das Caching von Feldnamen und Feldern herausstellen.

- Um die Effizienz der Implementierung noch weiter zu erhöhen, könnte eine Erweiterung von AspectJ vorgenommen werden: zur Zeit ist es mit AspectJ nicht möglich, die Methodenaufrufe einer auf Feldebene annotierten Kollektion innerhalb einer Klasse als Pointcut zu spezifizieren. Es wird deshalb in der Implementierung der Umweg benutzt, die jeweilige Kollektion mit den vorhandenen Feldwerten innerhalb der Klasse zu vergleichen, um zu entscheiden, ob das Feld gebunden auftritt oder nicht. Hier könnte eine Erweiterung von AspectJ um einen neuen Joinpoint eine deutliche Verbesserung darstellen.

- Eine weitere Erweiterung von AspectJ könnte darin liegen, Klassen zu identifizieren, die auf Feldebene annotiert sind. Damit würde die Verwendung der Annotation `@Bound` wegfallen können.

- Da die Annotation `@Bind` nicht typsicher refaktorisierbar ist, wäre eine

Erweiterung der Refaktorisierungswerkzeuge für verschiedene Entwicklungsumgebungen eine Aufgabe, die die Nutzung der Annotation insbesondere in agilen Projekten deutlich vereinfachen könnte.

- Auf Basis von Annotationen und AspectJ könnte überlegt werden, wie die Spezifizierung von Invarianten für Relationen aussehen und implementiert werden könnte. Hierzu könnten z.B. die Annotationen um weitere Parameter ergänzt werden bzw. existierende Annotationen für Validierungen genutzt werden[1].

6.3 Fazit

Die entwickelte Bibliothek zur Umsetzung von bidirektionalen, mengenwertigen Verweisen in Domänenmodellen mit Java vereinfacht die Entwicklung deutlich und man möchte nach kurzer Zeit nicht mehr auf diese Möglichkeit verzichten. Allerdings kommt die Bibliothek auch zu einem Preis: auf der einen Seite ist das Laufzeitverhalten der Bibliothek im Vergleich zu einer manuellen Implementierung bzw. compilerbasierten Lösung durch die genutzte Reflexion bei schreibenden Zugriffen um den Faktor 2-15 langsamer. Dies ist allerdings bei der Entwicklung von Geschäftsanwendungen, die ihre Modelle in Datenbanken speichern, in der Regel kein Problem, da die Zugriffe auf die Persistenzschichten um mehrere Größenordnungen langsamer sind. Bei den in Geschäftsanwendungen weitaus häufiger genutzten lesenden Zugriffen gibt es keinen Unterschied zu einer compilerbasierten Lösung.

Der zweite Preis, der gezahlt werden muss, ist das notwendige Load Time Weaving der Aspekte. Dies ist über das Java VM Tools Interface zwar problemlos zu erreichen, aber es ist natürlich nicht in allen Projekten möglich. In vielen Projekten ist es sicher nicht ohne weiteres akzeptabel, die Startskripte für einen Applikationsserver zur Vereinfachung der Softwareentwicklung zu modifizieren. Von daher bleibt auch mit der vorgestellten Bibliothek ein Restproblem bestehen, das nur dadurch gelöst werden könnte, dass ein Sprachelement wie die vorgestellte Annotation zum Standard der Java-Entwicklung werden würde.

[1] siehe hierzu den Vorschlag und existierende Implementierungen für eine Bean Validation: `http://jcp.org/en/jsr/detail?id=303`

Literaturverzeichnis

[Albano et al., 1997] Albano, A., Ghelli, G., and Orsini, R. (1997). A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language. *Proceedings of the 17th International Conference on Very Large Data Bases.*

[Amelunxen et al., 2004] Amelunxen, C., Bichler, L., and Schürr, A. (2004). Codegenerierung für Assoziationen in MOF 2.0. *Proceedings Modellierung.*

[Baldoni et al., 2007a] Baldoni, M., Boella, G., and van der Torre, L. (2007a). Relationships meet their Roles in Object Oriented Programming. *Proceedings of the 2007 International Conference on Fundamentals of Software Engineering.*

[Baldoni et al., 2007b] Baldoni, M., Boella, G., and van der Torre, L. (2007b). The Interplay between Relationships, Roles and Objects. *Proceedings of Workshop From Objects to Agents (WOA).*

[Baldoni et al., 2009] Baldoni, M., Boella, G., and van der Torre, L. (2009). The Interplay between Relationships, Roles and Objects. *Proceedings of the 2009 International Conference on Fundamentals of Software Engineering.*

[Balzer et al., 2007] Balzer, S., Gross, T., and Eugster, P. (2007). A Relational Model of Object Collaborations and its Use in Reasoning about Relationships. *Proceedings of European Conference on Object-Oriented Programming (ECOOP).*

[Berler et al., 2000] Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., and Velez, F. (2000). *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann Publishers Inc.

[Bierman and Wren, 2005] Bierman, G. and Wren, A. (2005). First-class Relationships in an Object-Oriented Language. *Proceedings of European Conference on Object-Oriented Programming (ECOOP).*

[Evans, 2003] Evans, E. (2003). *Domain-Driven Design: Tackling Complexity In the Heart of Software.* Addison-Wesley Longman Publishing Co., Inc.

[Genova et al., 2003] Genova, G., Castillo, C., and Llorens, J. (2003). Mapping UML Associations into Java Code. *Journal of Object Technology.*

[Goetz et al., 2006] Goetz, B., Bloch, J., Bowbeer, J., Lea, D., Holmes, D., and Peierls, T. (2006). *Java Concurrency in Practice.* Addison-Wesley Longman, Amsterdam.

[Laddad, 2009] Laddad, R. (2009). *AspectJ in Action: Enterprise AOP with Spring Applications.* Manning Publications Co., 2nd edition.

[Nelson et al., 2008] Nelson, S., Pearce, D., and Noble, J. (2008). Implementing First-Class Relationships in Java. *Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages.*

[Noble, 1997] Noble, J. (1997). Basic Relationship Patterns. *Proceedings of European Conference on Object-Oriented Programming (ECOOP).*

[Noble and Grundy, 1995] Noble, J. and Grundy, J. (1995). Explicit Relationships in Object-Oriented Development. *Proceedings of the Conference on Technology of Object-Oriented Systems and Languages (TOOLS).*

[Osterbye, 1999] Osterbye, K. (1999). Associations as a language construct. *Proceedings of the Conference on Technology of Object-Oriented Systems and Languages (TOOLS).*

[Osterbye, 2007] Osterbye, K. (2007). Design of a Class Library for Association Relationships. *Proceedings of the Workshop on Library-Centric Software.*

[Pearce and Noble, 2006] Pearce, D. and Noble, J. (2006). Relationship Aspects. *Proceedings of the 5th International Conference on Aspect-Oriented Software Development.*

[Pradel, 2008] Pradel, M. (2008). Explicit Relations with Roles - A Library Approach. *Workshop on Relationships and Associations in Object-Oriented Languages (RAOOL) at OOPSLA.*

[Rumbaugh et al., 1989] Rumbaugh, J., Shah, A., Hamel, J., and Borsari, R. (1989). DSM: an Object-Relationship Modeling Language. *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA).*

[Rumbaugh, 1987] Rumbaugh, R. (1987). Relations as Semantic Constructs in an Object-Oriented Language. *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA).*

[Steimann and Stadtler, 2010a] Steimann, F. and Stadtler, D. (2010a). Objektrelationale Programmierung. *Tagungsband zur Software Engineering.*

[Steimann and Stadtler, 2010b] Steimann, F. and Stadtler, D. (2010b). Wie die Objektorientierung relationaler werden sollte: Eine Analyse aus Sicht der Datenmodellierung. *Proceedings of Modellierung.*