Entwicklung einer Persistenzlösung für Object Teams auf Basis der Java Persistence API

Diplomarbeit

Berlin, den 17. Juni 2009

Olaf Otto

Matr. Nr. 210341

Erstgutacher: Prof. Dr.-Ing. Stefan Jähnichen

Betreuer und Zweitgutachter: Dr.-Ing. Stephan Herrmann

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Softwaretechnik
Prof. Dr. -Ing. Stefan Jähnichen

Eidesstattliche Erklärung

Die selbständige und eigenhändige Anfertigung versichere ich an Eides Statt.

Berlin, den 17. Juni 2009

- -

Olaf Otto

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Entwicklung einer auf der Java Persistence API (JPA) [13][1] aufbauenden Lösung für die Kombination objekt-relationaler Persistenz mit dem Programmiermodell Object Teams [22]. Neben einer eingehenden Betrachtung der Entwicklung und des Status Quo relationaler Persistenz, objekt-relationalen Mappings (ORM) und Persistenzlösungen für Java ist es eines der Hauptziele dieser Arbeit, die Kapselung persistenzspezifischer Aspekte von Syntax und Semantik des Domänenmodells durch die JPA aufrecht zu erhalten und zu erproben, wie stark eine gelungene *Separation of Concerns*[2] zu einer besseren Kombinierbarkeit separierter Aspekte beiträgt.

Im Rahmen dieser Arbeit wird eine konkrete Persistenzlösung auf Basis von Eclipselink [32] entworfen und implementiert, welche nicht nur die Kombination der Persistenzaspekte mit den Object Teams-spezifischen Modellierungsaspekten demonstriert, sondern die Grundlage einer praktisch einsetzbaren Persistenzlösung für Object Teams bildet. Mit der Anwendung geeigneter Techniken zur testgetriebenen Entwicklung und der Integration in das Spring Framework [38] für Container-Manager Persistence werden weitere, wiederverwendbare Aspekte der Softwareentwicklung und -gestaltung mit Object Teams betrachtet.

[1] Zitate erfolgen im Weiteren im Format [Quellennummer], ggf. gefolgt von (Kapitel / Paragraph in der zitierten Quelle).

[2] Kursive Schrift kennzeichnet im Weiteren initial eingeführte technische Begriffe sowie Quellcode.

Inhaltsverzeichnis

1 Einleitung **5**
1.1 Komplexitätsreduktion durch Modularisierung 5
1.2 Dominante Aspekte 6
1.2.1 Ausprägung des Logging-Aspektes 7
1.2.2 Ausprägung des Persistenz-Aspektes 7
1.3 Object Teams 8

2 Relationale Datenspeicherung und Objektorientierung **10**
2.1 Datenbankmanagementsysteme 10
2.2 Relationale Datenbankmanagementsysteme 11
2.3 Der object-relational impedance mismatch 13
2.3.1 Class-Table-Mismatch 15
2.3.2 Transaktions-Mismatch 17
2.3.3 CRUD-Mismatch 18
2.3.4 Memory-Mismatch 18
2.3.5 Mismatch auf prozesslicher Ebene 19
2.4 Objektorientierte Datenbankmanagementsysteme 21
2.5 Objekt-Relationales Mapping 21

3 Persistenzlösungen für Java **23**
3.1 JDBC 23
3.2 Enterprise Java Beans (EJB) 25
3.3 Hibernate und JDO 26
3.4 Die Java Persistence API (JPA) 27
3.4.1 Aufbau und Funktionsweise der JPA 29
3.4.2 Auszeichnung von Klassen als Entities 33
3.4.3 Container Managed und Bean Managed Persistence . 34

3.4.4 Das Spring Framework als CMP Provider 35
3.4.5 OSGI und die Java Persistence API 36
3.5 Zusammenfassung 37

4 Entwurf einer Persistenzlösung für Object Teams 39
4.1 Grundlegende Entwicklungsstrategie 40
4.2 Semantik des Object Teams Programmiermodells 40
4.2.1 Teams, Rollen und Basis 41
4.2.2 Adaption von Basis-Verhalten 41
4.2.3 Kapselung von Rollen 43
4.2.4 Teamaktivierung 45
4.3 Syntax der Implementierung des Object Teams Programmiermodells 45
4.3.1 Team Infrastruktur 46
4.3.2 Rollen und Basis Infrastruktur 50
4.4 Die Object Teams Applikation 51
4.5 Die Java Persistence API als Persistenzlösung für die Object Teams Applikation 52
4.6 Allgemeine Anforderungen an eine JPA-Implementierung für Object Teams 54
4.7 Analyse der Anforderungen von Object Teams an eine JPA Implementierung 55
4.7.1 Unterstützung der Object Teams-Strukturen 55
4.7.2 Unterstützung der Object Teams-Semantik 57
4.8 Analyse der Anforderungen der JPA an die Object Teams Applikation 59
4.8.1 Modellierung von Rollen-Klassen 59
4.9 Adaption von Eclipselink als Persistenzimplementierung für Object Teams 60
4.9.1 Adaption von Entity-Metadaten in Eclipselink 60
4.9.2 Adaption zur Instanziierung von Rollen 63
4.9.3 Erweiterung von Eclipselink durch Object Teams . . . 64
4.10 Lazy-Loading Unterstützung 68
4.11 Zusammenfassung 70

5 Implementierung der Persistenzlösung auf Basis von Eclipselink 71
5.1 Entwicklungsprozess und -technologien 71
5.1.1 Testgetriebene Entwicklung (TDD) 71
5.1.2 Beschreibung durch ein standardisiertes Projektmodell 73
5.1.3 Wiederverwendbarkeit der Object Teams-spezifischen Konfiguration 77
5.1.4 Continuous Integration und Maven 77
5.1.5 Issue Management 78
5.2 Implementierung der semantischen Adaption 78
5.2.1 OTInstantiationPolicy 78
5.2.2 OTClassDescriptor 79
5.2.3 OTRoleBaseMapping und OTRoleTeamMapping . . . 80
5.3 Implementierung der strukturellen Adaption 81
5.3.1 OTFieldAdapter 82
5.3.2 OTMetadataField 82
5.3.3 OTPersistenceUnitProcessor 82
5.3.4 OTMappingAccessor 83
5.3.5 OTRoleDatabaseField 83
5.4 Implementierung des Lazy-Loadings 83
5.5 Container Manager Persistence und Object Teams 84
5.5.1 Integration von Object Teams in das Spring Framework 85
5.5.2 Spring Kontext Konfiguration 87
5.6 Einschränkungen und Anforderungen der Implementierung . 90
5.6.1 Keine Unterstützung von Property-based access in Team oder Rolle 90
5.6.2 Keine Unterstützung von Rollenvererbung zwischen Teams 91
5.6.3 Ausführung als priviligierte Applikation 91
5.7 Beispiel für die Verwendung der JPA mit Object Teams . . . 92
5.8 Zusammenfassung 94

6 Fazit und Ausblick 95
6.1 Mächtige Kombination ausgereifter Techniken 95
6.2 Ausblick 96

6.3 Anlagen 97

Kapitel 1

Einleitung

1.1 Komplexitätsreduktion durch Modularisierung

Kaum ein Gestaltungsmittel ist mit vergleichbar großen Hoffnungen verbunden wie das Prinzip der Modularisierung, der Bewältigung der *essentiellen Komplexität* [18] von Software durch Aufteilung in beherrschbar komplexe Bestandteile. Die Verinnerlichung dieses Prinzips in *objektorientierten Programmiersprachen* ließ bereits am Ende der achtziger Jahre die Hoffnung aufkommen, mit Objektorientierung die *silver bullet* gefunden zu haben, mit welcher sich die Komplexität anspruchsvoller Softwaresysteme beherrschen ließe [18].

Dennoch finden wir heute, mehr als zwanzig Jahre später, eine Situation vor, in welcher trotz der erheblichen Fortschritte im Bereich von Objektorientierung und Modularisierung die Komplexität von Software keine beherrschte oder leicht kalkulierbare Größe ist. Die Gründe für diesen Zustand sind vielschichtig. Zum einen haben sich die Anforderungen an die Leistungsfähigkeit von Software deutlich erhöht und ihre Komplexität hat weiter zugenommen. Zum anderen zeigte sich, dass Objektorientierung allein keine hinreichende *Separation Of Concerns* [15] ermöglichte. Dies führte zu einer Erweiterung bestehender Programmiermodelle um Modularisierungsmechanismen für *querschnittliche Funktionalitäten*, welche unter dem Begriff der *aspektorientierten Programmierung* (AOP) [26] zusammengefasst werden und erst in jüngerer Zeit weite Verbreitung fanden[3].

[3]Es wird davon ausgegangen, dass die grundlegende Problematik der *crosscutting concerns* und die Lösungsansätze der Aspektorientierung bekannt sind. Diese können ggf. in

Wie Hermann [23] anführt, erfordert eine effiziente Handhabung von Komplexität zudem geeignete Prozesse zur Entwicklung modularer Software und eine planvolle *Komposition* von Systemen aus den zuvor separierten Concerns[4]. Durch den ständigen – durch AOP weiter beschleunigten – Zuwachs an Gestaltungsmöglichkeiten sind die entsprechenden Prozesse und Techniken jedoch vielfach Forschungsgegenstand. Tatsächlich wurde der bekannteste Vertreter und Vorreiter der AOP-Spracherweiterungen, AspectJ [25], mit der Zielstellung entwickelt, durch einen breiten Praxiseinsatz von AOP und dem Aufbau einer Community Anwendungsmuster zu identifizieren und geeignete Prozesse zur *aspektorientierten Softwareentwicklung* zu finden – ein Vorhaben, dass bis heute fortgeführt wird.

1.2 Dominante Aspekte

Eine weiteres Problem besteht darin, dass die Aufteilung eines Systems in Aspekte bisher vorwiegend mit dem Ziel der Trennung auf Code-Ebene betrieben wurden. Ein Aspekt besitzt jedoch weitere, wichtige Dimensionen innerhalb derer ebenfalls eine Vermischung von Concerns auftreten kann. Für einen Aspekt lassen sich mindestens drei Dimensionen festlegen:

Die *syntaktische Dimension* umfasst den Quellcode der Aspekt-Implementierung. Die zweite, deutlich schlechter abzugrenzende Ebene ist die *semantische Dimension*, welche die Konzepte und Gestaltungsmöglichkeiten beinhaltet, die durch einen Aspekt in ein Softwaresystem eingeführt oder aus ihm ausgeschlossen werden. Des Weiteren besitzt ein Aspekt meist auch eine *systematische Dimension*, welche aus den aspektspezifischen entwicklungsprozesslichen und betrieblichen Anforderungen besteht.

Mit AOP lassen sich Aspekte auf syntaktischer Ebene separieren. Ob dies auf semantischer und systematischer Ebene gelingt hängt hingegen von der Dominanz des Aspektes auf diesen Ebenen ab. Zur Illustration wird im Folgenden die Dominanz zweier Aspekte, *Logging* und *Persistenz*, verglichen.

[26] und [25] nachgelesen werden.

[4] Im weiteren werden die Begriffe *Concern* und *Aspekt* gleichwertig verwendet.

1.2.1 Ausprägung des Logging-Aspektes

Logging ist einer der bekanntesten crosscutting Concerns und dient in vielen Beispielen der Illustration aspektorientierter Programmiertechniken. Logging ist jedoch ein verhältnismäßig trivialer Aspekt. Er stellt weder besondere Anforderungen an die Systematik eines Entwicklungsprozesses, noch führt die Idee des Loggings neue Konzepte oder Gestaltungsmöglichkeiten in ein Softwaresystem ein, welche die Semantik andere Module beeinflussen. Auf syntaktischer Ebene besteht Logging meist aus einem einfach Aufruf einer geeignet parametrisierten *log(...)*-Methode ohne Berücksichtigung von Seiteneffekten oder Rückgabewerten. Durch diese in Abbildung 1.1 dargestellte Beschränkung auf die syntaktische Dimension ist Logging daher ein trivial zu modularisierender crosscutting Concern, was ihn als prominentes Beispiel für die Auswirkungen von AOP wenig interessant macht.

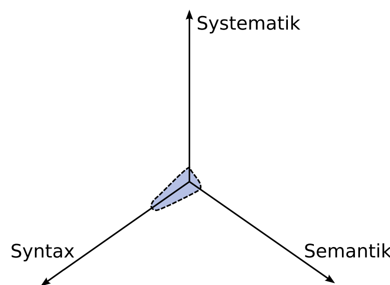

Abbildung 1.1: Ausprägung des Logging-Aspektes

1.2.2 Ausprägung des Persistenz-Aspektes

Persistenz gehört mit zu den bekanntesten crosscutting concerns – nicht zuletzt, da Persistenz eine gewichtige Menge von Konzepten, beispielsweise *Transaktionen* und *Relationale Algebra* – in die Domäne eines Softwaresystems einführt. Ob man Persistenz nun als einen einzelnen oder ein Konglomerat mehrerer Aspekte betrachtet sei dahingestellt – sie hat erhebliche Auswirkungen auf den Entwicklungsprozess einer Software. Bereits in der Analysephase müssen die möglichen Anforderungen an eine Persistenzlösung präzise erfasst werden, da kritische Designentscheidungen abhängig sein können. Des Weiteren eröffnet die Verwendung mächtiger Sprachen und Datenbanksysteme völlig neue Gestaltungsmöglichkeiten und Semantiken

in den Modulen eines Softwaresystems. Der Umgang mit persistenzspezifischen Sprachen, die Handhabung von Transaktionen und Datenbankkommunikation ziehen zudem eine nicht unerhebliche Menge neuer Syntax und querschnittlichen Codes nach sich.

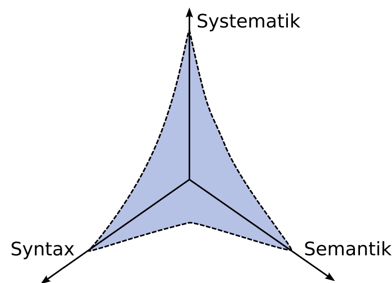

Abbildung 1.2: Ausprägung des Persistenz-Aspektes

Die in Abbildung 1.2 gezeigte starke Ausprägung von Persistenz in allen Dimensionen ist aus zwei Gründen problematisch: Zum einen ist es durchaus möglich, dass in einem Software-Projekt mehrere Aspekte mit einer dominanten systematischen oder semantischen Dimension existieren, was im Falle von Unterschieden zu Widersprüchen bzw. Konflikten führen kann. Zum anderen koppeln sich vor allem semantisch dominante Aspekte implizit mit anderen Aspekten, da sie elementare Designentscheidungen bestimmen und dazu neigen das Gesamtsystem von ihrer Semantik abhängig zu machen. So mögen die entsprechenden Aspekte zwar auf Code-Ebene getrennt sein, sie können jedoch nicht ohne weiteres separat entwickelt oder ausgetauscht werden.

Persistenz ist daher ein wesentlich interessanterer Fall für aspektorientiertes Programmieren, da sie hohe Anforderungen an die Separierung und die Komposition von Concerns stellt.

1.3 Object Teams

Die bisherige AOP-Spracherweiterungen decken meist nur Teilbereiche der Möglichkeiten von AOP ab und haben individuelle Einschränkungen, was nicht zuletzt der Tatsache geschuldet ist, dass AOP eine verhältnismäßig junge Entwicklung ist. So sind Software, Methoden und Prozesse für AOP

weiterhin Gegenstand aktiver Forschung. Allein das 2004 ins Leben gerufene Projekt *European Network of Excellence on Aspect-Oriented Software Development* (AOSD-Europe) [1] weist zahlreiche grundlegende Arbeiten im Bereich von Analyse und Design, Sprachen, formalen Methoden, Programmverifikation und Anwendungen für AOP aus.

An der Technischen Universität Berlin entstand 2002 das Programmiermodell *Object Teams* [22], welches Aspektorientierung mit weiteren Modularisierungskonzepten zusammenführt und zahlreiche Forschungstätigkeiten im Bereich von Modularisierung und Aspektorientierung motiviert hat.

Diese Arbeit beschäftigt sich im Weiteren mit der Konzeption und Implementierung einer Persistenzlösung für dieses Programmiermodell.

Kapitel 2

Relationale Datenspeicherung und Objektorientierung

2.1 Datenbankmanagementsysteme

Persistenz bezeichnet in der Informationstechnik die Speicherung von Daten über einen langen, in der Regel den Lebenszyklus einer Programmausführung überdauernden Zeitraum hinaus. Mit Anbeginn des Zeitalters der elektronischen Datenverarbeitung bestand die Notwendigkeit große, bis dato analog vorgehaltene Informationsmengen strukturiert zu persistieren und für die Datenverarbeitung nutzbar zu machen. Die zu diesem Zeitpunkt sehr kostenintensiven Systeme kamen dabei meist in einem Umfeld zum Einsatz, in welchem sie sich zum einen durch Personaleinsparungen im Bereich der Verarbeitung großer Mengen von Daten, zum anderen durch die programmatische Analyse der Datenmengen – bspw. zur strategischen Unterstützung von Managemententscheidungen oder zu wissenschaftlichen Zwecken – amortisieren mussten. Den datenspeichernden Systemen war dabei gemein, dass sie die Daten zentral vorrätig hielten und diese einer beliebig großen Menge von Benutzern zugängig machten, welche den Datenbestand sowohl lesen als auch verändern konnten.

Aus diesem Umstand resultierten die besonderen Charakteristiken von *Datenbankmanagementsystemen* (DBMS), welche bis heute bestehen.

Multiuser-Fähigkeit Der Datenbankbestand steht einer beliebig großen Menge von Benutzern zur Verfügung, welche gleichzeitig lesenden und schreibenden Zugriff auf die Daten haben können, ohne das dies zu Inkonsistenzen führt.

Skalierbarkeit Große Daten- und Benutzermengen machen das Verteilen von Datenbanksystemen über mehrere Rechnersysteme notwendig, bspw. in Form von *Clustering* und *load-Balancing*.

Transaktionsunterstützung Das Scheitern von Manipulationsoperationen darf den Datenbestand nicht beschädigen. Datenbanksysteme können daher Datenmanipulationen in *Transaktionen* kapseln, deren Änderungen rückgängig gemacht (*rolled back*) werden können. Transaktionen werden ebenfalls benötigt um Datenmanipulationen auf physisch verteilten Rechnersystemen, wie sie beim Clustering bzw. load-Balancing von Datenbanken verwendet werden, zu ermöglichen.

Speicherverwaltung Von der großen Menge eingesetzten Massenspeichers – bspw. in Form mehrerer Festplatten – wird abstrahiert. DBMS enthalten eine Abstraktionsschicht zur physikalischen Speicherung, welche die zu speichernden Daten über eine Administrationsschnittstelle wartbar macht und plattformunabhängig *Datentypen* zur Speicherung festlegt.

2.2 Relationale Datenbankmanagementsysteme

Die individuelle Anpassung an bestimmte Einsatzfelder führte zur Entwicklung spezialisierter DBMS, welche Datenspeicherung und Datenverarbeitung miteinander vereinten. Die zunehmende Verfügbarkeit günstiger Rechnersysteme erhöhte jedoch ebenfalls den Verbreitungsgrad von Datenbanksystemen und machte es notwendig, die Kosten des Einsatzes von Datenbanken durch Standardisierung zu reduzieren. Ein weiteres Problem bestand darin, dass die vorherrschende Ablage von Daten in Graph – bzw. Baumstrukturen den gespeicherten Daten ihre Struktur aufzwang, was die Nutzbarkeit der Daten deutlich einschränkte.

Vor diesem Hintergrund entwarf Codd [11] 1970 das Modell der *relationalen Datenspeicherung*, welches die Daten unabhängig von den datenspei-

chernden Strukturen organisiert und die Abfrage und Modifikation von Daten durch eine standardisierte, prädikatenlogische Sprache ermöglicht. Dieses Modell ist bis heute die Grundlage *relationaler Datenbankmanagementsysteme (RDBMS)*, welche heute etwa 98% [27] der im Einsatz befindlichen Datenbanksysteme ausmachen.

Vor Einführung des relationalen Modells bildete oftmals eine hierarchische Datenorganisation die Grundlage von DBMS, bspw. in dem Mitte der sechziger Jahre durch IBM entwickelten System *IMS* [36]. Wie in Abbildung 2.1 dargestellt, kennt das *hierarchische Modell* keine Trennung zwischen der Speicherstruktur und der Beziehungssemantik von Daten. Die Hierarchie der Speicherung kann jedoch immer auf mindestens zwei Arten modelliert werden. In Abbildung 2.1 können Produkte einer Firma untergeordnet sein – oder umgekehrt.

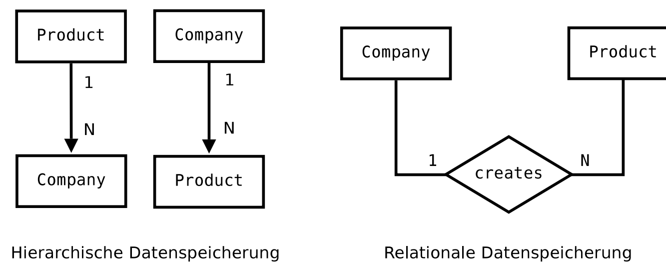

Abbildung 2.1: Hierarchische und relationale Datenspeicherung

In Abhängigkeit von der Speicherstrategie verändert sich jedoch ebenfalls die Beziehungssemantik der Daten. Im relationalen Modell hingegen werden Daten in *Relationen* gespeichert, welche keiner (sichtbaren) hierarchischen Speicherstruktur untergeordnet sind. Wie in Abbildung 2.2 dargestellt ist die Beziehungssemantik im relationalen Modell explizit modelliert, indem die beteiligten Relationen einander über eindeutige *Schlüssel* referenzieren.

Die mächtige Semantik des relationalen Modells, seine Unabhängigkeit von Datenstrukturen sowie die Verbreitung von *SQL* [10] als Datenbankkommunikationssprache hatten zur Folge, dass RDBMS eine eigene Klasse von Systemen bildeten, welche sich getrennt von auf anderen Semantiken aufbauenden Systemen und Programmiersprachen weiterentwickelten.

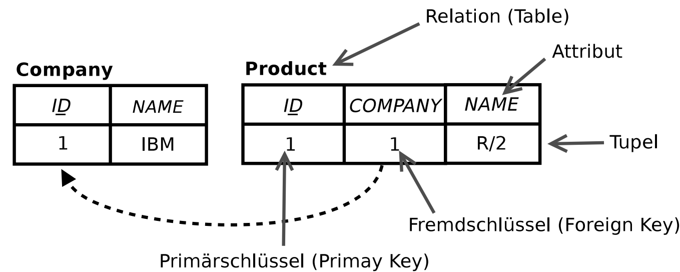

Abbildung 2.2: Explizite Beziehungssemantik im relationalen Modell

Während das relationale Modell die Welt der DBMS revolutionierte, gewann im Bereich der Programmiersprachen *Objektorientierung* zunehmend an Bedeutung. Beide Modelle basieren auf grundlegend verschiedenen Anforderungen an die Modellierung von Daten, so dass die – aufgrund des Verbreitungsgrades und der Mächtigkeit beider Systeme notwendige – Verwendung relationaler DBMS durch objektorientierte Programmiersprachen einen hohen Konversionsaufwand zwischen diesen Modellen bedingt. Diese Unterschiede in der Modellierung sind als *object-relational impedance mismatch* bekannt und haben weitreichende Auswirkungen auf Softwaregestaltung und -prozess.

2.3 Der object-relational impedance mismatch

Als object-relational impedance mismatch bezeichnet man die auf fundamentalen konzeptuellen Unterschieden zwischen relationalem und objektorientierten Modell basierenden Konflikte bei der Abbildung dieser Modelle aufeinander.

Das objektorientierte Modell beschreibt eine Problemdomäne durch miteinander interagierende Akteure, den *Objekten*. Objekte besitzen einen eigenen, durch *Kapselung* geschützten *Zustand* sowie eine eindeutige, von diesem Zustand unabhängige *Identität*. Ihr (für andere Akteure sichtbares) *Verhalten* wird durch Methoden definiert. Um Objekte mit ähnlichen bzw. gleichen Eigenschaften zu klassifizieren verwenden *typsichere* Programmiersprachen, bspw. Java, *Klassen* zur *Typisierung* und zur abstrakten Beschreibung des Verhaltens und der Datenstruktur von Objekten. Zwischen diesen Typen können Vererbungsbeziehungen bestehen (Generalisierung-Spezialisierung),

welche die Weitergabe von Methoden (Verhalten) und Attributen (Datenstrukturen) von dem beerbten an den erbenden Typ ermöglichen. Aus diesen grundlegenden Konzepten folgen eine Reihe weiterer wichtiger Eigenschaften, bspw. Polymorphie durch die Austauschbarkeit von Objekten durch Objekte eines kompatiblen Typs, welcher das gleiche Verhalten bietet.

Das von Codd definierte relationale Modell [11], auf welchem alle heute existierenden relationalen Datenbanken basieren, verfolgt ein grundlegend anderes Ziel als das objektorientierte Modell. Während letzteres Akteure modelliert, welche beliebig große Mengen von Daten und anderen Akteuren zur Umsetzung eines bestimmten Verhaltens verwenden, bildet das relationale Modell Daten als Relationen – im mathematische Sinne – ab und definiert für diese eine *Normalform*. Das resultierende relationale Datenmodell ist mittels *Prädikatenlogik* erster Ordnung beschreibbar [11]. Die in den Relationen abgelegten Daten bilden somit eine strukturierte Menge an Fakten, aus welcher jede mögliche Untermenge mittels einer an Prädikatenlogik orientierten Sprache *Ad Hoc* abgefragt werden kann. Die Stärke dieses Modells liegt also nicht in der Fähigkeit des Modellierens und Berechnens eines bestimmten Verhaltens, sondern in der Beschreibung von Daten und ihren Relationen.

Aus diesen grundlegenden Unterschieden resultieren eine Reihe praktischer Probleme bei der Abbildung dieser Modelle aufeinander. Die schwerwiegendsten und am häufigsten auftretenden Probleme werden im Folgenden näher beschrieben.

Objekt-Identitäts Mismatch

Identität ist eine zentrale Eigenschaft objektorientierter Systeme. Jedes Objekt besitzt eine eindeutige Identität, welche vom Zustand des Objektes unabhängig ist und eine Referenzierung des Objektes erlaubt. Daraus resultiert die Unterscheidung zwischen gleichen und ähnlichen Objekten, wobei ersteres die gleiche Objektidentität und die zweite Eigenschaft eine implementierbare Vergleichssemantik von Objekten bezeichnet, welche die Ähnlichkeit durch einen Vergleich der Zustände der Objekte berechnet.

Dieses Identitätskonzept ist im relationalen Modell nicht enthalten – ein Informationstupel gleicht einem anderen genau dann, wenn die Werte des Tupels identisch sind. Zur eindeutigen Identifikation von Tupeln definiert Codd [11] daher das Konzept des *Primärschlüssels*, einer Menge von Attributen, deren Werte in dieser Kombination eindeutig sind, also nicht zweimal innerhalb eines Tupels der selben Relation auftreten.

Da sich diese Konzepte nicht aufeinander abbilden lassen werden Objekte, welche in relationalen Datenbanken persistiert werden sollen, meist mit einem speziellen Attribut ausgestattet, welches der Semantik eines Primärschlüssels genügt und als solcher persistiert wird. Dies hat den Vorteil, dass Objekte anhand eines atomaren Attributes – meist eines Integer-Wertes – über den Kontext einer Programmausführung hinaus eindeutig referenzierbar sind.

2.3.1 Class-Table-Mismatch

Das relationale Modell kennt weder Vererbung noch erlaubt es nicht-atomare Attribute in Relationen, beispielsweise *Collections*. Daher müssen sowohl die Vererbungshierarchie als auch die komplexen Referenzbeziehungen zwischen Klassen aufwändig nachgebildet werden.

Für die Nachbildung von Vererbungsbeziehungen gibt es drei bekannte Lösungsstrategien.

Single table : Alle Klassen einer Vererbungshierarchie werden auf eine gemeinsame Relation abgebildet. Zur Unterscheidung der Tupel in der Relation dient ein diskriminierendes Attribut, welches in der Regel den eindeutigen Namen der jeweiligen Klasse enthält. Dies führt zu einer verhältnismäßig hohen Redundanz der Einträge in der Relation, da viele Attribute stets null-Werte enthalten, da sie zu einer anderen Vererbungsebene als der persistierten gehören. Dementsprechend ist es auch kaum möglich, die Konsistenz des relationalen Modells mittels geeigneter *constraints* sicherzustellen.

Table per concrete class : Jede konkrete, also nicht-abstrakte Klasse enthält eine eigene Tabelle. Während dies die Redundanz der Daten des Single table-Ansatzes vermeidet, wird eine schematische Redundanz

eingeführt, da alle geerbten Attribute eines Objektes stets in der Relation der erbenden Klasse erneut definiert werden.

Joined : Diese Strategie ist die präziseste Nachbildung der Vererbungsbeziehungen zwischen Objekten in das relationale Modell, da nur die Attribute einer Subklasse auf eine eigene Relation abgebildet werden, welche spezifisch für diese sind. Die Objektdaten werden also über eine der Klassenhierarchie äquivalente Menge von Relationen verteilt persistiert. Während dieses Modell optimal Redundanzen vermeidet erhöht sich die Komplexität der Abfrage von Objektdaten erheblich, was deutliche Performanceeinbußen bei der Abfrage der Daten zur Folge haben kann.

Ein weiterer Konflikt besteht bei der Abbildung mehrstelliger Relationen zwischen Klassen. Da die Attribute einer Relation atomar sind, muss die Referenzlogik im Schema umgekehrt (1:N Relationen) bzw. unter Zuhilfenahme von Mapping-Relationen (M:N, aber auch 1:N) modelliert werden.

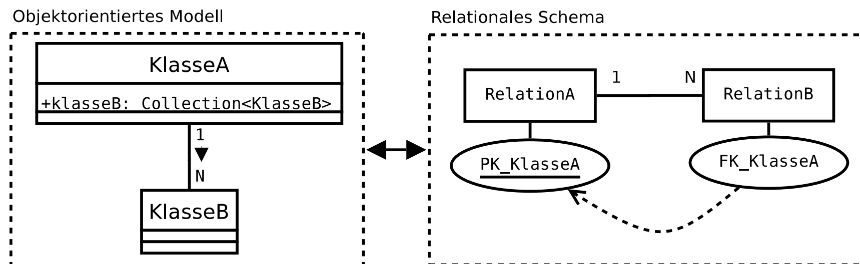

Abbildung 2.3: Objektorientierte und relationale Modellierung von 1:N Relationen

Wie in Abbildung 2.3 dargestellt wird eine 1:N Beziehung zwischen Klassen mittels eines *Fremdschlüssels (FK)* in der referenzierten Relation modelliert und somit umgekehrt.

Abbildung 2.4 zeigt die Verwendung einer Mapping-Relation zur Modellierung einer M:N Beziehung zwischen Klassen. Da keine der Relationen nicht-atomare Attribute besitzen darf, wird die Beziehung selbst – ähnlich einer Assoziationsklasse im objektorientierten Modell – als Relation modelliert.

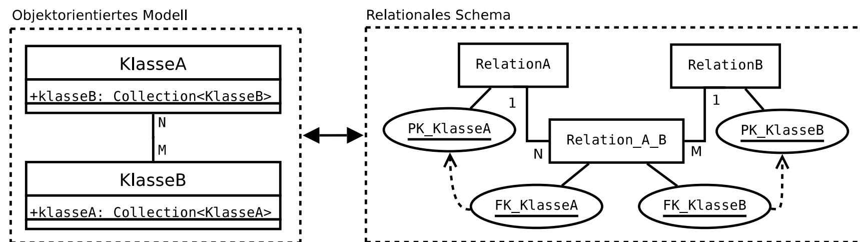

Abbildung 2.4: Objektorientierte und relationale Modellierung von M:N Relationen

2.3.2 Transaktions-Mismatch

Das Konzept der transaktionalen Durchführung von Manipulationen ist das entscheidende Konzept von Datenbanksystemen zur Gewährleistung konsistenter Operationen bei gleichzeitigem Zugriff mehrerer Benutzer. Es unterscheidet sich deutlich von der unter anderem in Java verwendeten *Synchronisation* zur Sicherung eines *kritischen Abschnittes*. Sichert letztere die Konsistenz von Programmzuständen mittels eines pessimistischen Lockings, welches den Eintritt in den Abschnitt auf meist nur einen Prozess beschränkt und somit gleichzeitigen Zugriff vermeidet, erlaubt eine *Transaktion* immer die Durchführung kritischer Operationen, führt jedoch keine Änderungen am Datenbestand durch, wenn Aktionen innerhalb einer Transaktion fehlschlagen. Diese Semantik basiert auf der *Isolation* der Arbeitsbereiche von Benutzern der Datenbank und ist geeignet die Konsistenz von Operationen auf physisch verteilten Datenbanksystemen zu sichern.

Um Transaktionen nutzbar zu machen ist es daher meist unerlässlich, transaktionale Abschnitte im objektorientierten Programm explizit zu modellieren und bereits in der objektorientierten Analyse zu berücksichtigen. Dabei gibt es zwei unterschiedliche Möglichkeiten: Entweder werden Transaktionen in einen kritischen Abschnitt integriert und somit *serialisiert*, womit ein Fehlschlagen der Transaktion durch gleichzeitigen Zugriff ausgeschlossen wird, oder Transaktionen werden wie eine *Invariante* behandelt und in einen Abschnitt mit invariantenerhaltender Fehlerbehandlung – bspw. *try ... catch ... finally ...* in Java – integriert, um die Konsistenz des Programmzustandes bei Fehlschlag der Transaktion zu gewährleisten.

2.3.3 CRUD-Mismatch

Manipulationen in der Datenbank werden im Wesentlichen durch die Operationen *Create*, *Read*, *Update* und *Delete* (CRUD) definiert, welche auf beliebig großen Mengen von Relationen und Tupeln arbeiten. Für keine dieser Operationen existiert eine natürliche Entsprechung im objektorientierten Modell, so dass Entsprechungen für alle CRUD-Operationen explizit in ein objektorientiertes Programm eingeführt werden müssen.

2.3.4 Memory-Mismatch

Lazy-Loading

Die Lebenszeit von Objekten im objektorientierten Programmen ist meist auf die Lebenszeit eines bestimmten Berechnungsprozesses – in Java höchstens auf die Lebenszeit der *Virtual Machine* – beschränkt. Die zur Berechnung verwendeten Objekte werden während der Laufzeit sukzessive erzeugt und befinden sich im Hauptspeicher.

Datenbankorientierte Anwendungen speichern jedoch große Datenmengen – bis hin zu Milliarden von Einträgen – in einem relationalen Schema. Es ist daher weder wünschenswert noch möglich alle in der Datenbank persistierten Objekte in den Hauptspeicher zu laden, um Berechnungen durchzuführen. Idealerweise werden daher Objekte aus der Datenbank geladen, wenn sie zur Ausführung einer Operation gelesen werden – und zwar auch dann, wenn sie einander referenzieren.

Ein solches Verhalten wird als *Lazy-Loading* bezeichnet und muss im objektorientierten Modell implementiert werden. Betroffen sind vor allem mehrstellige Relationen zwischen Objekten – bspw. persistente Collections – welche meist als *Futures* implementiert werden, so dass erst beim Zugriff auf Elemente der Collection die referenzierten Objekte sukzessive aus der Datenbank geladen werden.

Memory management und Relationale Modellierung

In objektorientierten Laufzeitumgebungen werden Objekte meist so lange im Speicher gehalten, wie Referenzen auf diese bestehen. In Java befreit

ein *Garbage Collector* den Hauptspeicher automatisch von nicht mehr referenzierten Objekten. Hingegen existieren Objekte im relationalen Modell so lange, bis sie durch eine delete-Operation explizit entfernt werden. Eine mit der Zählung von Referenzen vergleichbare Strategie zur Entfernung nicht referenzierter Tupel existiert im relationalen Modell nicht. Eine automatische Entfernung nicht referenzierter Objekte aus der Datenbank wäre ohnehin nicht sinvoll, da diese genau mit der Intention persistiert werden sie für zukünftige Berechnungsprozesse wieder zu verwenden.

Aufgrund dieser unterschiedlichen Semantiken ist es notwendig, den Zustand der Datenbank durch Löschen resp. Speichern von Objekten explizit mit dem objektorientiert modellierten Zustand zu synchronisieren und ggf. gewünschte Referenzierungssemantik zusätzlich im relationalen Modell zu formulieren. Dazu steht neben der Modellierung notwendiger Beziehungen zwischen Tupeln durch (*referential constraints*) mit *Kaskadierung* ein Mechanismus zur Verfügung mit welchem Tupel, die von einem zu löschenden Tupel referenziert werden, ebenfalls entfernt werden können. Die entsprechende Referenzbeziehung muss dazu mit *on delete cascade* attributiert sein.

2.3.5 Mismatch auf prozesslicher Ebene

Der object-relational impedance mismatch ist jedoch nicht auf die Modellebene beschränkt, sondern betrifft ebenso Entwicklungsprozess und Betrieb von Softwaresystemen.

Bereits während der Planungsphase müssen die Anforderungen an verwendete Datenbanksysteme und die Struktur der Daten im Datenbanksystem mit den Ergebnissen aus der objektorientierten Analyse in Einklang gebracht werden. Dabei gilt es ebenfalls die Interessen unterschiedlicher Stakeholder des Datenbestandes miteinander zu vereinen. Zu diesen zählen unter Anderem:

Drittapplikationen : Jede Art von Anwendung, welche ebenfalls auf den Datenbestand zugreift.

Datenbankadministratoren : Zuständig für die Pflege des Datenbankschemas und die Administration der gesamten Datenbank-Architektur.

Management : Definiert grundlegende Anforderungen an zu speichernde Inhalte und Auswertungsmöglichkeiten sowie ggf. technische Rahmenbedingungen.

Analysten : Definieren Anforderungen an Ad-Hoc Query Möglichkeiten für den Datenbestand.

Software-Ingenieure : Benötigen die Datenbank als Persistenz-Service für Anwendungen.

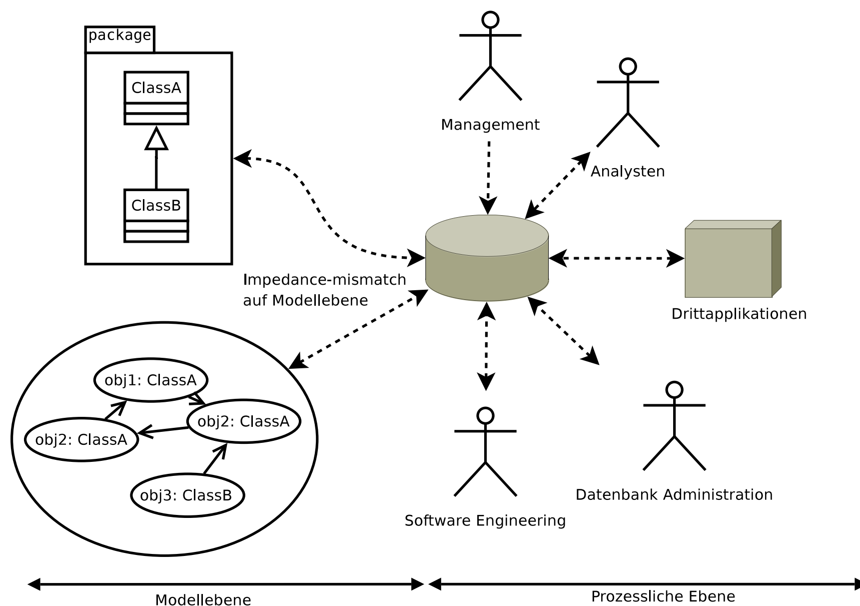

Abbildung 2.5: Aspekte des Object-relational impedance mismatch.

Während die Konflikte um die Verantwortlichkeiten für die Struktur und Semantik des Datenbestandes prinzipieller Natur und daher technisch kaum lösbar sind, existieren für den impedance mismatch auf Modellebene im Wesentlichen zwei Lösungsstrategien: Die Angleichung des Datenbanksystems an die Konzepte objektorientierter Programmiersprachen und die Verwendung einer Abstraktionsschicht zwischen Datenbanksystem und objektorientiertem Softwaresystem, welche eine transparente Konversion zwischen beiden Modellen zur Verfügung stellt. Eine solche Abstraktionsschicht wird auch als *object relational mapping (ORM)* bezeichnet.

2.4 Objektorientierte Datenbankmanagementsysteme

Ein objektorientiertes DBMS zeichnet sich durch zwei fundamentale Eigenschaften aus: Es ist ein DBMS – unterstützt also Persistenz, Concurrency, Recovery, Ad-hoc Queries und eine administrative Verwaltung des Massenspeichers – und es ist ein objektorientiertes System, unterstützt also Typen bzw. Klassen, Vererbung, Kapselung, Overriding, Identität und Verhalten [7]. Der Vorteil dieses Ansatzes liegt offenkundig darin, den Abbildungsaufwand zwischen den Objekten und ihrer persistenten Repräsentation zu minimieren. Auf diese Weise kann die Integration von Persistenz in objektorientierte Programmiersprachen erheblich vereinfacht und die Performanz von Persistenzoperationen deutlich gesteigert werden.

Die Nachteile der objektorientierten Speicherung bestehen vor allem in der Tatsache, dass die Auswertung der Datenbasis nicht mittels einer relationalen Algebra wie SQL erfolgen kann, sondern eine Query-Sprache benötigt wird, welche die Charakteristika objektorientiert strukturierter Informationen berücksichtigt und somit deutlich komplexer als SQL ist. Des Weiteren sind objektorientierte Datenbanken meist fest an die verwendete Programmiersprache und die durch diese verwendeten Datentypen gebunden, was die persistierten Daten für Dritte schwer nutzbar macht. Diese Nachteile sowie der durch die Bindung an die verwendeten Programmiersprachen bedingte Mangel an Standardisierung sind mitverantwortlich für den Umstand, dass der Einsatz von objektorientierten DBMS bis heute vorwiegend auf Spezialanwendungen begrenzt ist, welche besonders stark von der objektorientierten Informationsmodellierung profitieren bzw. spezielle Anforderungen an die Performanz einer Persistenzlösungen stellen [27]. Der Marktanteil objektorientierter Datenbanken betrug im Jahr 2000 lediglich etwa 2% – mit abnehmender Tendenz [27].

2.5 Objekt-Relationales Mapping

Die Zentrale Idee des ORM ist die Erschaffung einer Persistenzlösung, welche die Stärken relationaler DBMS und die Mächtigkeit objektorientierter Programmierung durch eine transparente Vermittlungsschicht – dem Map-

ping – vereint. Dabei übernimmt ein ORM in der Regel zwei Aufgaben: Die Ableitung eines relationalen *Schemas* aus den statischen Informationen – den Klassen –des objektorientierten Modells und die Konversion von Objekten in *Tupelmengen* und umgekehrt. Im Gegensatz zu objektorientierten Datenbanken entsteht dabei keine feste Bindung zwischen der verwendeten objektorientierten Ausführungsebene und der Datenbank. Stattdessen kommunizieren diese Schichten *transparent* über eine standardisierte und weit verbreitete Vermittlungssprache – SQL. Damit sind RDBMS und Applikationsschicht gleichfalls *datenunabhängig*. Diese Datenunabhängigkeit begründet ebenfalls die *Austauschbarkeit* des verwendeten RDBMS, welche ein wichtiger Vorteil von ORM ist.

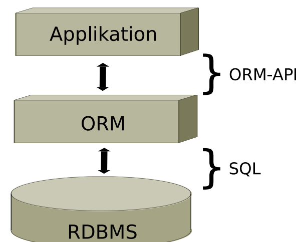

Abbildung 2.6: Schematische Darstellung eines ORM Layers

Die Aufgaben des ORM sind jedoch nicht auf die Konversion von Daten beschränkt. Weitere wichtige Aufgaben bestehen in der Abstraktion datenbankspezifischer Funktionen – bspw. Transaktionen, create, read, update und delete (CRUD) Operationen – und dem Bereitstellen einer API und Querysprache zum Selektieren und Manipulieren persistierter Objekte.

Aufgrund des hohen Verbreitungsgrades von RDBMS ist object-relational mapping die Persistenzlösung mit dem höchsten Verbreitungsgrad für objektorientierte Systeme [27]. Es existieren ausgereifte Implementierungen für zahlreiche objektorientierte Programmiersprachen, allen voran Java, welche seit 2006 über eine standardisierte ORM-API verfügt.

Kapitel 3

Persistenzlösungen für Java

3.1 JDBC

Bereits in der 1996 veröffentlichten Version 1.1 enthielt Java mit *JDBC* [19] als eine der ersten Programmiersprachen eine standardisierte low-level Abstraktionsschicht für die Kommunikation mit relationalen Datenbanken. JDBC löste die zuvor existierende herstellerspezifische Integration relationaler Datenbanken ab und ermöglichte plattformunabhängige persistierende Java-Programme.

Da sich JDBC jedoch auf die Abstraktion von Datenbankverbindungen, SQL-Datentypen, Queries und Ergebnismengen beschränkte war es weiterhin notwendig persistenzspezifische Applikationslogik in Form von Schemaerzeugung, Verbindungs- und Transaktionsmanagement, CRUD-Operationen, Abbildung von Relationen zwischen Objekten und Fehlerbehandlung innerhalb des jeweiligen Java-Programmes zu implementieren. Dies führte zu einer erheblichen Vermischung der Concerns des Modell-Codes mit den umfangreichen Concerns der Persistenzlogik, was eine sehr schlechte Wartbarkeit und deutliche erhöhte Komplexität des Codes zur Folge hatte. Listing 3.1 zeigt ein bereits stark vereinfachtes Beispiel für den Umfang der zur Persistierung einfacher Daten notwendigen Logik bei direkter Verwendung von JDBC.

```
1  public class Example {
2    int primeAttribute;
3    Example reference;
4
5    public void persist(String jdbcConnectionUrl)
6                throws SqlException, ClassNotFoundException {
7      // Herstellerspezifische Implementierung instanziieren
8      Driver d = Driver.class.forName("com.factory.JDBCDriverName")
9                          .newInstance();
10     DriverManager.registerDriver(d);
11     Connection conn = d.getConnection(jdbcConnectionUrl);
12     try {
13       // Transformation von
14       // Example in das relationale Modell
15       PreparedStatement st = conn.prepareStatement(
16                           "insert into exampletable
17                           (primeAttribute, reference)
18                           values (?, ?)"
19                           );
20       st.setInt(1, this.primeAttribute);
21       st.setInt(2, this.reference.primeAttribute);
22       st.executeUpdate();
23       } finally {
24         if (conn != null) {
25           conn.close();
26         }
27     }
28   }
29 }
```

Listing 3.1: Vermischung von Geschäftslogik und Persistenzlogik durch JDBC

Softwareingenieure waren zudem genötigt gleichzeitig objektorientiert zu entwickeln und ihr Modell mittels SQL in relationaler Algebra auszudrücken. Da sich ihre Expertise meist auf erstere Domäne beschränkte führte dies oftmals zu einer suboptimalen und ineffizienten Repräsentation der Modelle im relationalen Modell und zu schwach normalisierten, schlecht wartbaren und inkonsistenten Datenbankschemata.

3.2 Enterprise Java Beans (EJB)

Um Persistenzaspekte und Modellcode zu trennen entwickelte Sun das Modell der *Container Managed Persistence (CMP)*, einer standardisierten Form von Persistenz, welche als *Service* von einem *Container* – einem J2EE-Server – angeboten wurde und eine Abstraktionsebene zwischen Persistenzlogik und Modell bildete [30](9.10). Trotz zahlreicher Erweiterungen in EJB 2.0 [14](10) überwogen jedoch die Nachteile dieses Modells – insbesondere, da es sich um eine Erweiterung des veralteten und bürokratischen EJB Programmiermodells handelte.

Wie in Abbildung 3.1 anhand der Implementierung einer persistenten Klasse "Entity" dargestellt, vermischte das EJB Modell auf unvorteilhafte Weise die Concerns des domänenspezifischen Modells mit Aspekten der Persistenz und J2EE Laufzeitumgebung. Entwickler mussten stets ein Konglomerat von Klassen und Interfaces zur Verfügung stellen, deren Gestaltung massiven Einschränkungen unterworfen war. So erzwang das EJB-Modell die Implementierung von Schnittstellen der J2EE-Umgebung, die Bindung an die API des J2EE-Frameworks und das Durchführen diverser Konfigurationsschritte für Ausführung und Deployment persistenter Komponenten [30](9.7).

Die zunehmende Komplexität und Bürokratisierung der Entwicklungsprozesse und Implementierungsstrategien von Enterprise Java Beans führten in der Java-Community zu einer Gegenbewegung, welche einen Paradigmenwechsel forderte. Statt komplexer Programmiermodelle sollte sich die Entwicklung wieder auf den Kern objektorientierter Softwareentwicklung fokussieren: Einfache, nicht von anderen Concerns und Systemkomponenten abhängige Klassen. Am Rande einer Konferenz tauften R. Parsons, M. Fowler und J. MacKenzie dieses Prinzip *POJO* – Plain Old Java Object. Insbesondere durch den Erfolg von Aspektorientierter Programmierung mit AspjectJ [25] wurde POJO zum Begriff eines Paradigmenwechsels und einer Erfolgsgeschichte, welche die bis dato existierenden Persistenzlösungen grundlegend veränderte.

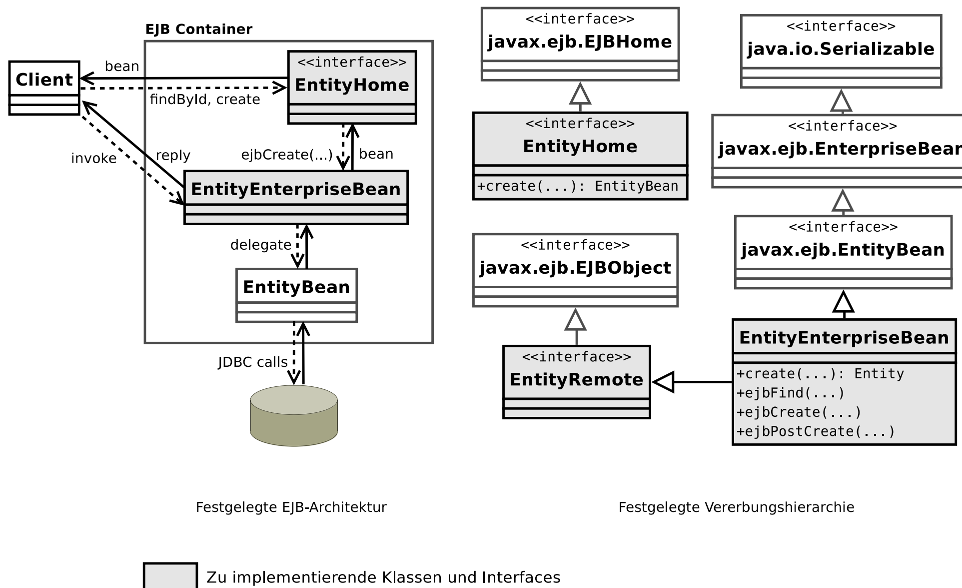

Abbildung 3.1: Zwangsweise Implementierung von Interfaces und Bereitstellung von Architekturaspekten in EJB 1 und 2

3.3 Hibernate und JDO

Bereits 1999 Begann Sun Microsystems mit den Arbeiten an JCR-12: Java Data Objects (JDO) [35]. Als Konkurrenzlösung zur Persistenz mit *Entity Beans* in EJB sollte die JDO-Architektur eine Java-zentrierte, transparente Handhabung von Persistenz bieten. Die Integration in *Enterprise Information Systems (EIS)*[5] sollte durch die Implementierung der JDO API durch *Vendors*[6] – also Experten in der jeweiligen Domäne – erfolgen.

Ein Jahr vor Veröffentlichung der JDO Spezifikation im Jahr 2002 begann G.King mit der Entwicklung von *Hibernate*, einer open-source Alternative zu CMP Entity Beans [9](10). Im Gegensatz zu diesen stellte Hibernate transparente Persistenzmechanismen zur Verfügung, indem die Persistenzeigenschaften von Klassen durch XML-basierte Metadaten beschrieben wurden. Gleichzeitig bot Hibernate mit der *Hibernate Query Language (HQL)* eine aus SQL abgeleiteten Sprache zur *Ad-Hoc*-Abfrage persistierter Objek-

[5]Der generische Begriff *EIS* bezeichnet allgemein eine Klasse von Systemen welche mit großen Mengen an Daten geschäftskritische Aufgaben bewältigt. Zu diesen gehören insbesondere DBMS.

[6]Die Begriffe Implementierung und Vendor werden im Folgenden gleichwertig verwendet.

te. Mit Hibernate konnten Entwickler somit einfache Java-Klassen – POJO's – mit geringem Aufwand um Persistenzeigenschaften erweitern. Die Übernahme durch JBoss im Jahr 2003 und die damit verbundene professionelle Unterstützung und zügige Weiterentwicklung sowie die zunehmende Unterstützung durch die open-source Community machten Hibernate so populär, dass die Veröffentlichung des JDO Standards im selben Jahr verhältnismäßig wenig Beachtung fand.

3.4 Die Java Persistence API (JPA)

Die schwindende Akzeptanz für die Komplexität und Vermischung von Concerns in EJB, die großen Erfolge von Hibernate und die Erkenntnisse aus dem Entwurf von JDO führten zu einer grundlegenden Änderung des Enterprise Java Beans Standards. Mit der 2006 veröffentlichten Version 3.0 von EJB [13] verwarf SUN erhebliche Teile vorher verwendeter Konzepte und Implementierungen und führte die Architektur von JDO mit den best-practice Ansätzen von Hibernate und Teilen der EJB-Semantik in der Java Persistence API zusammen.

Durch die Beteiligung von JBoss und Oracle, sowie durch die große Ähnlichkeit mit JDO (die JPA Spezifikation wird auch als Teilmenge der JDO Spezifikation betrachtet) verfügte die API von Beginn an über namhafte Implementierungen mit hohem Verbreitungsgrad. Neben Hibernate stellten Oracle mit *Toplink* (seit 2007 unter dem Namen *Eclipselink*) die open-source Referenzimplementierung der JPA [32] und die Apache Software Foundation (ASF) mit *OpenJPA* [3] (hervorgegangen aus BEA's kommerziellem *KODO JDO* Projekt) ausgereifte Implementierungen zur Verfügung. Diese breite Unterstützung und die klare Ausrichtung auf ORM führten zu einer großen Akzeptanz der JPA innerhalb der Java Community.

Der wesentliche Beitrag der JPA besteht in der genauen Spezifikation von Auszeichnung und *Defaults* für das Persistenzverhalten von Klassen sowie einer umfangreichen Spezifikation für den Lebenszyklus und die Abfrage persistenter Objekte über eine standardisierte Querysprache, der *Java persistence query language (JPQL)*. Die genaue Definition von Anforderungen an persistente Klassen, welche im Gegensatz zu EJB keine Einschränkungen von

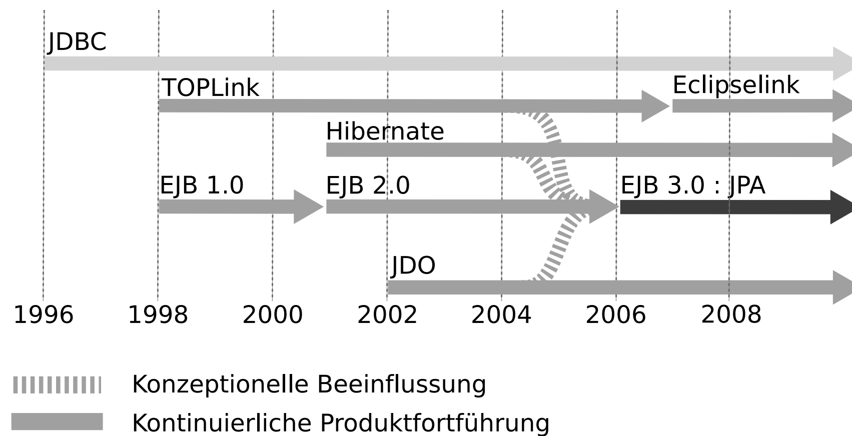

Abbildung 3.2: Chronologische Entwicklung von Persistenztechniken für Java

Vererbung, Polymorphie oder Kapselung erzwingt, ermöglicht Anwendern dabei *programming by exception*, also die Verwendung sicherer Standards (*defaults*) für das Persistenzverhalten von Entities[7] und die Beschränkung des Erstellens zusätzlicher Persistenzlogik auf Ausnahmefälle (*Exceptions*).

Die durch die JPA angestrebte *Standardisierung* objekt-relationaler Persistenz und die Leichtgewichtigkeit der API tragen wesentlich dazu bei, die Dominanz von Persistenz auf syntaktischer und systematischer Ebene zu reduzieren. Insbesondere die Entkoppelung der JPA Annotationen von der Vererbungshierarchie der Entities ist ein wesentlicher Fortschritt im Vergleich zu den vorhergehenden EJB Standards, da dies dem querschnittlichen Charakter der Persistenzeigenschaft durch eine querschnittliche Auszeichnungsebene Rechnung trägt. Gleichzeitig bilden die verwendeten *semantischen Annotationen* bzw. XML-Auszeichnungen Schnittpunkte für das Einweben zusätzlicher persistenzspezifischer Aspekte, bspw. zur Handhabung von Lazy-Loading oder Transaktionen.

[7]Der Begriff *Entity* resp. Entität wird im Kontext der JPA für persistierbare bzw. mit Persistenzinformationen annotierte Klassen verwendet, ist jedoch nicht gleichbedeutend mit dem für *Entity-relationship*-Diagramme verwendeten Entity-Begriff.

3.4.1 Aufbau und Funktionsweise der JPA

Die Codebasis der JPA besteht aus wenigen, leichtgewichtigen Interfaces und Annotationen. Wie in Abbildung 3.3 dargestellt setzt sich die Schnittstelle für eine JPA-Implementierung im Wesentlichen aus drei Interfaces zusammen, während aufgrund der definierten Defaults wenige Annotationen zur Auszeichnung einer Klasse als Entity ausreichen. Alternativ kann das Persistenzverhalten der Applikationsklassen in XML beschrieben werden. Die JPA-Implementierung liest die Metadaten somit aus den Klassen selbst und (optional) aus der XML-Beschreibung. Die Beschreibung in XML dient insbesondere für den Fall, dass der Quellcode persistenter Klassen nicht annotiert werden kann.

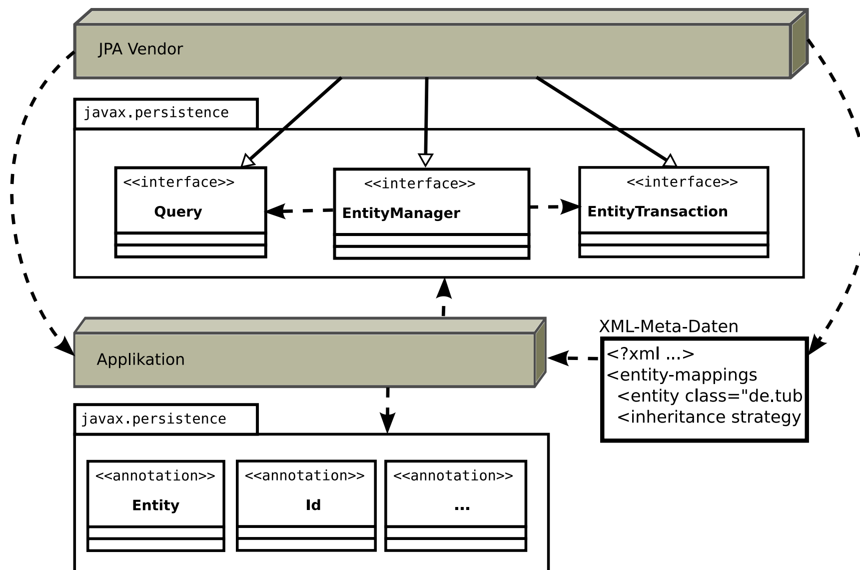

Abbildung 3.3: Schematische Darstellung der JPA Architektur

Das *EntityManager* Interface bildet die zentrale Schnittstelle zur Applikation und wird für den gesamten Lebenszyklus der persistenten Repräsentation von Entities, insbesondere für die Durchführung von CRUD--Operationen, verwendet. Jeder EntityManager isoliert seine Operationen durch Transaktionen, welche durch Implementierungen von *EntityTransaction* vorliegen. Abfragen der Datenbasis unter Verwendung der JPQL, welche ebenfalls über den EntityManager aufgerufen werden, sind durch das *Query*-Interface abstrahiert.

Wie JDO sieht die JPA die Austauschbarkeit ihrer Implementierung mittels *Konfiguration* vor. Darüber hinaus erlaubt die Spezifikation die gleichzeitige Verwendung unterschiedlicher Implementierungen innerhalb einer Applikation. Somit sind auf der JPA basierende Anwendungen nicht nur unabhängig von der Implementierung und damit portabel und plattformunabhängig, sondern auch in der Lage mit spezialisierten JPA-Implementierungen einheitlich eine große Bandbreite unterschiedlicher EIS zu unterstützen.

Lebenszyklus von JPA-Implementierungen

Wie in Abbildung 3.4 dargestellt durchläuft jede JPA-Implementierung im Wesentlichen zwei Phasen zur Bereitstellung objektrelationalen Mappings: *Bootstrapping* und *Persistence context management*.

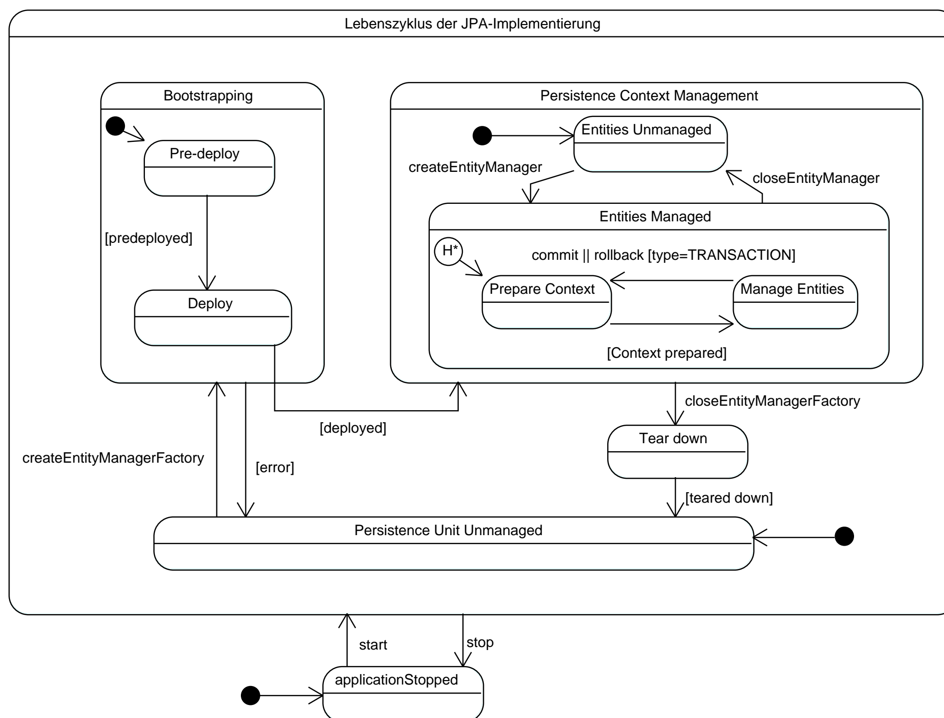

Abbildung 3.4: Lebenszyklus von JPA-Implementierungen

Bootstrapping. In der beim Erzeugen einer EntityManagerFactory einmalig ausgelösten bootstrapping-Phase werden die Metadaten zu verwaltender Klassen eingelesen und verifiziert. Die JPA-Implementierung erstellt eine

interne Repräsentation des aus der Klassenstruktur abgeleiteten relationalen Schemas (Pre-Deployment). Existiert kein entsprechendes Schema in der Datenbank, kann die JPA-Implementierung dieses anlegen[8] (Deployment). Zum Einlesen der Metadaten sucht die Implementierung dazu an den in [13](6.2) spezifizierten Orten nach der standardisierten *persistence.xml*, welche eine Menge von deployment-Einheiten, die *Persistence Units*, beschreibt. Eine EntityManagerFactory wird jeweils für eine konkrete Persistence Unit erstellt. Das Deployment wird ausschließlich für die Inhalte dieser Persistence Unit durchgeführt. Jede Persistence Unit kann separat von einer JPA-Implementierung verwaltet werden und besteht im Wesentlichen aus einem eindeutigen Namen und einer Menge von Klassen. Zusätzlich kann jede Persistence Unit eine bestimmte Transaktionsstrategie, eine Datenquelle, eine konkrete JPA-Implementierung sowie eine Reihe von Konfigurationen für diese enthalten [13](6.2.1).

Das Bootstrapping kann durch den Container (CMP) oder durch die Applikation (BMP) ausgelöst werden, indem die Factory-Methode für *javax.persistence.EntityManagerFactory*-Instanzen auf *javax.persistence.Persistence* aufgerufen wird, welche die passende Implementierung ggf. mit Hilfe standardisierter Metadaten findet [13](7.2). Am Ende der Phase existiert das relationale Schema in der Datenbank, die Konsistenz des Mappings ist validiert und es existiert eine Instanz der EntityManagerFactory, mit welcher *javax.persistence.EntityManager*-Instanzen zur Verwaltung des persistenten Zustandes der Entities erzeugt werden können, welche in den mit der Implementierung assoziierten Persistence Units enthalten sind.

Persistence context management. In dieser Phase verwaltet die Applikation den Persistenzzustand von Entities mit Hilfe der durch den EntityManager bereitgestellten CRUD-Operationen. Jeder EntityManager ist dabei mit einem *Persistence Context* assoziiert. Dieser Context ist im Standardfall mit einer Transaktion[9] assoziiert und kapselt somit eine konsistente, auch auf Datenbankebene isolierte Sicht auf den Persistenzzustand der in ihm verwalteten Entities. Dementsprechend haben alle persistenten Entities

[8]Dies ist nicht Teil der JPA Spezifikation, dennoch eine Grundvoraussetzung zum Erreichen der Context Management Phase für Persistenz in relationalen Datenbanken und wird durch alle JPA-Implementierungen unterstützt.

[9]Der Transaktionsbegriff umfasst hierbei auch abstrakte Transaktionsmodelle, wie sie bspw. durch die *Java Transaction API* (JTA) zur Verfügung gestellt werden. Ebenso sind verschachtelte Transaktionen zulässig.

der mit dem EntityManager assoziierten Persistence Unit innerhalb des Persistence Contexts maximal eine eindeutige Entsprechung (Instanz). Ebenso werden Metainformationen über den Persistenzzustand aller instanziierten Entities im Kontext verwaltet. Wie in Abbildung 3.5 gezeigt weist eine Entity dabei einen von vier Zuständen auf [13](3.2). Sie kann neu erzeugt sein und muss persistiert werden (*new*), sie kann persistiert und mit dem aktuellen Kontext assoziiert sein (*managed*) oder nicht (*detached*) oder sie wurde entfernt (*removed*). Die Zustandsänderungen erfolgen entweder durch den direkten Aufruf der jeweiligen CRUD-Operation des EntityManagers oder implizit durch Kaskadierung einer solchen über eine Referenzbeziehung einer anderen Entity.

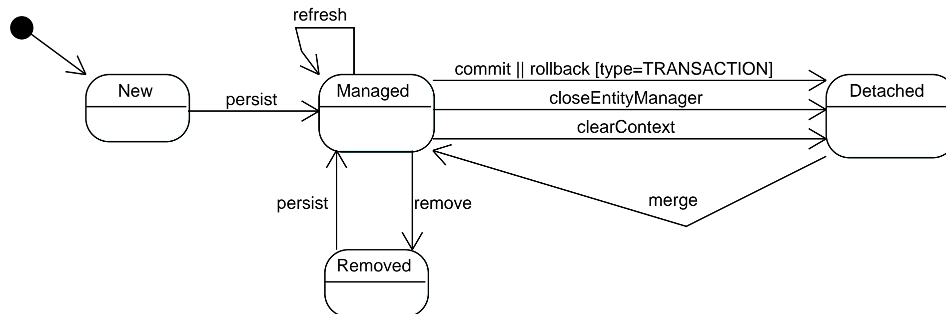

Abbildung 3.5: Lebenszyklus einer Entity [13](3.2)

Mit dem Ende der mit dem Persistence Context assoziierten Transaktion (durch commit oder rollback) wird ebenfalls die Verwaltung der Entities innerhalb des Contexts beendet (commit) oder aber die Entities wieder in ihren Ausgangszustand versetzt werden (rollback). Es besteht jedoch auch die Möglichkeit einen *extended persistence context* zu verwenden, welcher an den Lifecycle der EntityManager-Instanz gebunden ist und somit über mehrere Transaktionen hinweg existiert [13](3.3). Die Verwendung von BMP bedingt die Verwendung eines extended Persistence Context, da die Applikation in diesem Fall selbst für die Verwaltung des Persistence Context, insbesondere für das Öffnen und Schließen von Transaktionen zuständig ist. Die Verwaltung der Entities im Persistence Context endet jedoch spätestens mit dem Schließen des EntityManagers (*closeEntityManager*) oder dem expliziten Leeren des Persistence Context (*clearContext*). Die Persistence Context Management-Phase endet durch das Schließen der EntityManagerFactory und dem damit verbundenen Ende der Lebenszyklen aller EntityManager,

Persistence Context-Instanzen und abhängiger Ressourcen.

3.4.2 Auszeichnung von Klassen als Entities

Für die Umsetzung des Programming by Exception definiert die JPA eine Reihe von Anforderungen an die Gestaltung persistenter Klassen. Im Folgenden werden die wichtigsten Anforderungen beschrieben. Die vollständige Spezifikation ist ggf. unter [13](2.1) nachzulesen.

Auszeichnung: Entities müssen mittels der *javax.persistence.Entity*-Annotation ausgezeichnet sein oder in der XML-Mapping Information als Entity deklariert sein. Nur top-level Klassen dürfen als Entity deklariert werden. Die Deklaration von Interfaces oder *enum*-Typen als Entity ist unzulässig. Die Deklaration abstrakter Klassen als Entity ist zulässig.

Konstruktoren: Entities müssen einen default-Konstruktor ohne Argumente besitzen, dessen Sichtbarkeit *protected* oder *public* ist.

***final*-Deklarationen:** Entities dürfen weder *final* sein, noch dürfen sie persistente *final*-Felder besitzen.

Persistenz von Entity-Membern: Per default ist der gesamte Zustand einer als Entity ausgezeichneten Klasse persistent. Ausnahmen sind explizit zu annotieren. Besitzt eine Entity Attribute mit Referenztyp, welche nicht als Entity deklariert sind, müssen diese *java.io.Serializable* implementieren.

Zugriff auf den Entity-Zustand: JPA-Implementierungen greifen entweder über *getter* und *setter*-Methoden im *Java Beans*-Stil (*property-based access*) oder per direktem Feldzugriff (*field-based access*) auf den Zustand einer Entity zu. Die verwendete Zugriffsart hängt davon ab, ob Felder oder getter-Methoden der Entity annotiert wurden. Die default-Strategie ist der Implementierung überlassen. Eine Vermischung beider Strategien ist unzulässig. Getter und setter-Methoden sollten keine Geschäftslogik beinhalten, da der Zeitpunkt ihres Aufrufens durch die Implementierung undefiniert ist.

Identität: Jede Entity besitzt einen Primärschlüssel (primary key). Eine Entity muss genau eine primary key Auszeichnung besitzen.

Listing 3.2 zeigt eine als Entity annotierte Klasse. Die Semantik des Primärschlüssel-Members (@id) wird durch die JPA-Implementierung bereitgestellt (@GeneratedValue). Da per default alle Member von *Example* persistent sind, müssen Ausnahmen deklariert werden (@Transient). Die Semantik von Beziehungen (one to one, one to many, many to many) wird explizit deklariert (@OneToMany). Mit diesen Annotationen enthält das Beispiel bereits alle wesentlichen Bestandteile einer Auszeichnung als Entity mit der JPA.

```
1  @Entity
2  public class Example {
3    @Id
4    @GeneratedValue
5    private int id;
6
7    @Transient
8    private NotPersistentType notPersistent;
9
10   @OneToMany
11   private List<AnotherEntity> others;
12 }
```

Listing 3.2: Mit JPA Annotationen ausgezeichnete Entity mit field-based access

Alle Auszeichnungen einer Entity werden – bis auf die *@Entity*-Deklaration – vererbt. Eine von der in Listing 3.2 annotierten Entity ableitende Klasse muss somit lediglich erneut als Entity markiert werden. Alle weiteren Deklarationen werden wiederverwendet.

3.4.3 Container Managed und Bean Managed Persistence

Wie zuvor erwähnt existieren zwei grundlegende Strategien zur Verwaltung des Persistence Context: Bean Managed Persistence (BMP) und Container Managed Persistence (CMP). Im Falle von CMP bietet ein Container – bspw. ein J2EE-Server oder -Framework – eine standardisierte Integration der JPA mit dem Ziel, die JPA Konfiguration und das Management des Persistenzzustandes vom Modellcode der Anwendung zu trennen. Zu diesem Zweck stellt der Container einen konfigurierbaren *Persistence Context Lifecycle* zur

Verfügung, welcher im Wesentlichen beschreibt, zu welchem Zeitpunkt ein Persistence Context resp. eine Transaktion geöffnet und wann diese wieder beendet wird. Der Container stellt der Applikation einen mit diesem Kontext assoziierten Entity Manager zur Verfügung (bspw. durch *Dependency Injection* [17]), so dass diese CRUD-Operationen konsistent durchführen kann. Die Applikation muss sich daher weder mit dem Lifecycle der JPA-Implementierung noch mit dem Management von Entity Manager-Referenzen befassen [13](5.1).

Entsprechend muss im Falle von BMP die Applikation die Konfiguration der JPA-Implementierung und die Verwaltung des Persistence Context übernehmen, sowie einen mit dem jeweils aktiven Kontext assoziierten Entity Manager an alle Komponenten verteilen, welche Persistenzoperationen durchführen.

Durch die deutliche Vereinfachung der Applikationslogik und die meist zahlreichen zusätzlichen durch einen Container zur Verfügung gestellten Services, bspw. Dependency injection, ist Container Managed Persistence mit hoher Wahrscheinlichkeit die am weitesten verbreitet Strategie – nicht zuletzt, da mit dem *Spring Framework* [38] ein weit verbreiteter open-source Container existiert, welcher für POJOs durch Verwendung von XML, Annotationen und Defaults einen vollständigen Stack von JEE-Services sowie zahlreiche zusätzliche Dienste bietet.

3.4.4 Das Spring Framework als CMP Provider

Das Spring Framework hat sich seit seiner Entstehung im Jahr 2003 zum de-facto Standard für Applikationen entwickelt, welche die von Fowler 2004 beschriebene Dependency Injection [17] in Kombination mit zahlreichen Services der Enterprise Java-Welt nutzen möchten. Spring folgt dabei einer Reihe einfacher Grundprinzipien:

Minimale Beeinflussung der Applikationslogik: Alle Dienste von Spring können genutzt werden, ohne dass dies zu Abhängigkeiten resp. zusätzlicher Logik in der Applikation führt.

Modularität: Neben dem primär auf Dependency Injection und die damit einhergehende Konfiguration beschränkten core sind alle weiteren

Komponenten optionale Module.

Leichtgewichtige Container: Bereits mit minimaler Konfiguration kann jede Anwendung, unabhängig davon ob sie in einem JEE-Container Server oder als standalone-Anwendung läuft und unabhängig von der Laufzeitumgebung, bspw. JavaME / SE oder EE, verwendet werden. Sowohl die Speichernutzung als auch die Zahl abhängiger Bibliotheken sind gering.

Verbreitung von best practice-patterns: Spring stellt einfach zu verwendende Services für Factories, MVC, Dependency Injection und zahlreiche weitere bewährte Entwicklungsmuster zur Verfügung.

Für die Verwendung objekt-relationalen Mappings verfügt Spring im Wesentlichen über drei Module [28](2.2.1). *Spring ORM* ermöglicht die einheitliche Konfiguration konkreter JPA-Vendor, bietet ORM-spezifisches Ressourcenmanagement, bspw. in Form von Connection pools und stellt umfangreiches Transaktionsmanagement zur Verfügung. *Spring DAO* ermöglicht die transparente Konfiguration von *Data Access Objects* zum typsicheren Management von Entities über EntityManager und mit *Spring AOP* steht ein mächtiger Mechanismus zur annotationsbasierten Steuerung des Transaktionsverhaltens des Persistence Context zur Verfügung [28](2.3.4.1.1).

Da die Beschreibung der umfangreichen Konfigurationsmöglichkeiten und Features der Komponenten den Rahmen dieser Arbeit deutlich überschreiten würden werden die relevanten Teile des Spring Frameworks im weiteren Verlauf der Arbeit sukzessive vorgestellt wenn sie verwendet werden. Details sind ggf. in [38] und [28](2.1) nachlesbar.

3.4.5 OSGI und die Java Persistence API

Mit den umfangreichen Möglichkeiten zur Isolation von Applikationsmodulen durch Classloader, dem Verwalten und Auflösen von Modulabhängigkeiten und -Versionen sowie der Bereitstellung eines Lifecycle für Applikationskomponenten gewinnt OSGI [33] zunehmend an Bedeutung. Mit Spring dynamic modules [34] existiert eine umfangreiche Integration von OSGI in das Spring Framework. Zwischen der Version 1 der JPA Spezifikation [13] und der Core-Spezifikation der aktuellen OSGI-Version 4.1 [33] bestehen jedoch

Widersprüche in der Spezifikation des classloading-Verhaltens. OSGI verwendet Classloading, um Module (Bundles) zu isolieren ([33](3.2, 3.4)). Jedes Bundle besitzt einen eigenen isolierten *Bundle Class loader*. Die Bundle Class Loader besitzen lediglich den System class loader als gemeinsame Wurzel. Die JPA-Spezifikation hingegen definiert:

> "All persistence classes defined at the level of the Java EE EAR must be accessible to all other Java EE components in the application—i.e. loaded by the application classloader—such that if the same entity class is referenced by two different Java EE components (which may be using different persistence units), the referenced class is the same identical class."
> ([13], 6.2)

Daraus folgt, dass sowohl die Entity-Klassen als auch alle persistenzrelevanten Metadaten ggf. über die Classloader-Grenzen eines OSGI-Bundles hinaus erreichbar sein müssen. Obgleich dies kein fundamentaler technischer Widerspruch zu der Classloader-Isolation in OSGI ist, macht diese Spezifikation ein Aufbrechen der Bundle-Isolation oder die Beschränkung der Persistenzeigenschaft auf ein Bundle, in welchem alle Abhängigkeiten enthalten sind, notwendig. Diese Einschränkungen bedeuten jedoch de facto die Missachtung des fundamentalen Gestaltungsprinzips von OSGI und berauben OSGI-basierte Applikationen ihrer zentralen Eigenschaft – der Isolation und Austauschbarkeit ihrer Komponenten. Dementsprechend verwenden Applikationen, welche OSGI und JPA-Implementierungen einsetzen meist dynamic imports [33](3.8.2) in Zusammenhang mit vollständigen bundle-exports und durchbrechen somit die Kapselung ihrer Bundles. Eine die Semantik von OSGI erhaltende Integration beider Techniken existiert daher derzeit nicht.

3.5 Zusammenfassung

Für die Überwindung des Object-Relational Impedance mismatch existieren in Java ausgereifte Lösungen. Die Java Persistence API (JPA) ist derzeit das Modell mit dem höchsten Reife und -Verbreitungsgrad, sowohl zur Beschreibung von Persistenzeigenschaften als auch zur Durchführung von Persistenzoperationen. Für die Integration der Java Persistence API bieten

sich Eclipselink als JPA-Referenzimplementierung und das Spring Framework als ausgereifter, leichtgewichtiger Container an. Die diesen Systemen gemeine strikte Trennung des Applikationscodes von den querschnittlichen Persistenzeigenschaften und die Verwendung klar spezifizierter Defaults reduzieren die Auswirkungen von Persistenzeigenschaften auf syntaktischer Ebene erheblich. Durch die Standardisierung des persistenz-Lebenszyklus von Applikationen vereinfacht sich auch der Entwicklungsprozess persistierender Applikationen, da die Integration der Persistenzeigenschaften nicht mehr domänenspezifisch ist und Vorgehensweisen somit in weiten Teilen wiederverwendbar sind. Die semantische Dimension von Persistenz, bspw. die Einführung einer eindeutigen Objektidentität und das Berücksichtigen transaktionalen Verhaltens, bleibt jedoch in ihrer Mächtigkeit ungebrochen und muss weiterhin explizit im objektorientierten Modell beschrieben werden.

Kapitel 4

Entwurf einer Persistenzlösung für Object Teams

Das Programmiermodell Object Teams [22] erweitert Java um eine Menge neuer, zusammengesetzter Typen und Semantiken, welche unter Zuhilfenahme aspektorientierter Programmiertechniken – insbesondere Load Time-Weaving – implementiert sind. Diese neuen Typkonstruktionen und Semantiken erzeugen umfangreiche Anforderungen an eine Persistenzlösung. Während primitive Persistenzansätze wie Serialisierung oder die Verwendung von JDBC die Verantwortung für die Implementierung der Persistenzlogik in die Hände des Anwenders der Programmiersprache legen und somit – mit allen im vorhergehenden Kapitel beschriebenen Nachteilen – prinzipiell in Object Teams verwendet werden können, basiert die Verwendung eines fortschrittlichen Persistenzmodells wie der Java Persistence API [13] auf einer Menge konkreter Annahmen über die Gestaltung und das Verhalten von Entities. Dies machte es erforderlich, das in der entsprechenden Spezifikation beschriebene Verhalten einer Persistenzlösung so zu erweitern, dass auch die neuen Typkonstruktionen und Semantiken des Object Teams Programmiermodells unterstützt werden.

4.1 Grundlegende Entwicklungsstrategie

Die Entwicklung einer eigenen, Object Teams-spezifischen Persistenzlösung, bspw. auf JDBC-Basis oder durch eine eigene JPA-Implementierung, wurde aufgrund des proprietären Charakters einer solchen Lösung und des zu erwartenden extremen Entwicklungsaufwandes als nicht erstrebenswert erachtet.

Eine wesentlich erfolgversprechendere Strategie bestand hingegen in der Adaption einer bestehenden JPA-Implementierung. Es wurde dabei davon ausgegangen, dass die JPA aufgrund ihrer auf syntaktischer und semantischer Ebene auf ein essentielles Mindestmaß reduzierten Ausprägung in weiten Teilen mit Object Teams kompatibel ist, so dass sich die notwendigen Anpassungspunkte einer JPA-Implementierung auf eine Menge klar definierte Adaptionspunkte reduzieren lassen.

Des Weiteren wurde angenommen, dass die Verwendung einer ausgereiften und etablierten Implementierung zu einer hohen Akzeptanz auf Seiten potentieller Object Teams Anwender führen würde. Zudem verfügen alle am Markt befindlichen JPA-Implementierungen über eine Reihe zusätzlicher Eigenschaften, bspw. Caching, Clustering und Load Balancing, welche nicht Teil der JPA Spezifikation sind, jedoch eine zentrale Rolle bei der Implementierung komplexer, persistierender Anwendungen spielen. Die aus einer Adaption resultierende Verfügbarkeit dieser Eigenschaften für Object Teams wurde als weiterer, wichtiger Beitrag dieser Arbeit betrachtet. Im Folgenden werden die spezifischen semantischen und syntaktischen Eigenschaften von Object Teams analysiert. Diese werden mit den Anforderungen der JPA verglichen und notwendige Adaptionspunkte abgeleitet.

4.2 Semantik des Object Teams Programmiermodells

Object Teams führt in Java die Typen *Team* und *Rolle* ein. Gemeinsam bilden diese eine neue, querschnittliche Modularisierungsebene zur Beschreibung und Kapselung kontextspezifischen Verhaltens.

4.2.1 Teams, Rollen und Basis

Ein Team kapselt eine Menge von Rollen. Diese Rollen sind über einen neuen Beziehungstyp – der *playedBy*-Relation – an eine konkrete Klasse gebunden, welche in diesem Zusammenhang als *Basis* bezeichnet wird.

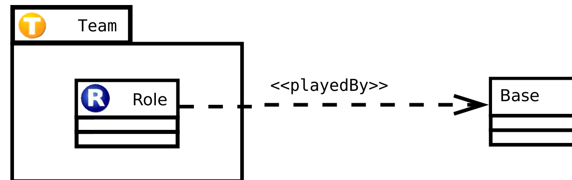

Abbildung 4.1: Team, Rolle und Basis in Object Teams

Das Team bildet dabei den Kontext der Rollen-Instanzen. Nur wenn das Team aktiviert ist, wird das Verhalten der Basistypen durch die Rollen adaptiert. Die Aktivierung von Teams findet zur Laufzeit statt und kann für einzelne Threads oder global erfolgen. Teams erfüllen dabei zwei spezifische Funktionen: Sie sind zum einem der Namensraum für die in ihm definierten Rollentypen und somit eine spezielle Art von *package*, zum anderen sind sie auch ein geschlossener Container zur Kapselung der Rolleninstanzen zur Laufzeit. Sowohl Teams als auch Rollen sind dabei ebenfalls herkömmliche Java-Klassen mit den entsprechenden bekannten Eigenschaften, wobei die Semantik von Vererbungsbeziehungen zwischen Teams so erweitert wurde, dass entlang einer *extends*-Relation zwischen Teams ebenfalls die Rollen des super-Teams geerbt werden. Wie in Abbildung 4.2 dargestellt können Rollen dabei mittels Namens-Konventionen implizit voneinander erben [24](§ 1.3.1 c). Mit diesem durch *copy inheritance* realisierten Mechanismus stellt Object Teams Mehrfachvererbung für Rollen zur Verfügung.

4.2.2 Adaption von Basis-Verhalten

Existiert ein Rollentyp mit playedBy-Relation zu einer Basis, so kann eine Instanz dieser Rolle erzeugt werden, welche das Verhalten einer konkreten Basisinstanz adaptiert. Dieser Vorgang wird als *lifting* [24](§ 4) bezeichnet. Eine Rolleninstanz ist dabei mit der adaptierten Basis-Instanz assoziiert. Es existiert höchstens eine Instanz pro Rollentyp für eine Basisinstanz innerhalb eines Teams. Die Lebenszeit von Rolleninstanzen ist auf die Lebenszeit ihrer Basisinstanz beschränkt [24](§ 2.3).

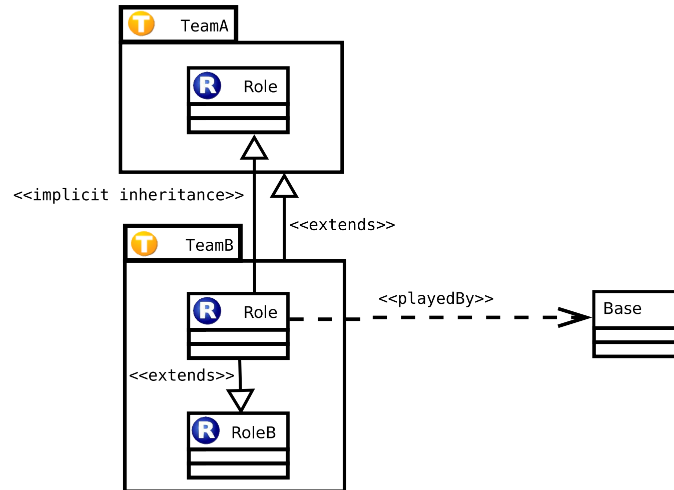

Abbildung 4.2: Vererbung und implizite Vererbung in Object Teams

Eine Rolle kann das Verhalten ihrer Basis ändern, indem sie Methodenaufrufe der Basis abfängt und an ihrer Stelle oder vor bzw. nach dem Methodenaufruf eigene Geschäftslogik ausführt (replace callin binding / before und after callin binding [24](§ 4.2 b, c)).

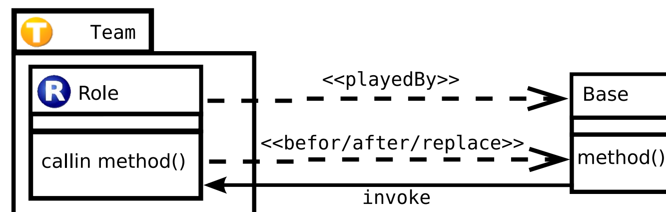

Abbildung 4.3: Callin bindings in Object Teams

Auf diese Weise kombinieren Rollen die bewährte Semantik von AspectJ [25] mit typsicherer Programmierung durch die explizite playedBy-Relation.

Smart Lifting

Smart Lifting ist ein Spezialfall von Lifting und bezeichnet die dynamische Auflösung des speziellsten möglichen Rollentyps für eine Basis. Smart Lifting wird verwendet, um den geeigneten Rollentyp für eine Basis zu finden, zu welcher voneinander erbende Rollen playedBy-Beziehungen haben. Bei einem Lifting für Basis BaseB in Abbildung 4.4 würde RoleC als Rollen-

typ ausgewählt, da diese die spezialisierteste kompatible Rolle ist. Hingegen käme für BaseA nur RoleA in Frage, da die playedBy-Beziehungen von RoleB und RoleC zu spezielleren Basisklassen bestehen.

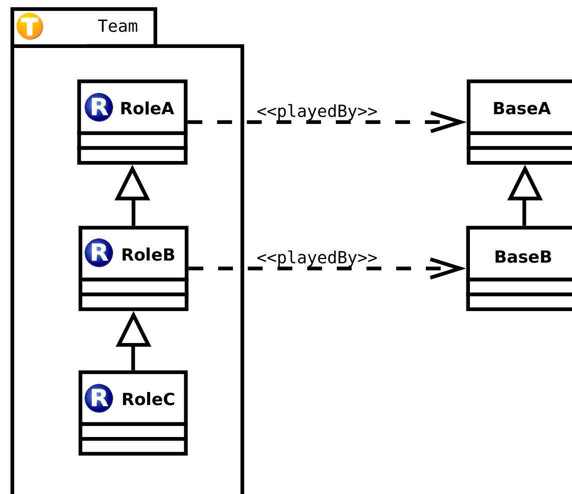

Abbildung 4.4: Spezialisierung von playedBy-Relationen zu Elementen einer Basis-Vererbungshierarchie

4.2.3 Kapselung von Rollen

Die Instanzen der in einem Team definierten Rollentypen werden in diesem eingeschlossen, indem die Sichtbarkeit des Rollentyps auf das Team beschränkt wird. Des Weiteren ist es möglich auch die Referenzierbarkeit von Rolleninstanzen auf Elemente innerhalb des Teams zu beschränken. Object Teams sieht dabei unterschiedliche Sichtbarkeiten von Rolleninstanzen vor [24](§ 7):

Public roles: Rollen mit dem Modifier *public* können auch außerhalb des Teams referenziert werden, wenngleich diese Referenzierung über einen speziellen Parameter direkt an die Teaminstanz gebunden ist, welche die Rolleninstanz beinhaltet.

```
1  public team class Team {
2      public class Role playedBy Base {
3      }
4  }
5  ...
6  final Team teamInstance = new Team();
7  Role<@teamInstance> roleInstance = new Role<@teamInstance>(new Base());
```

Listing 4.1: Externe Referenzierbarkeit von public roles

Eine Referenz der Form Role<@teaminstance> kann dabei auch Rückgabewert des Teams sein. Durch die Angabe <@teamInstance> ist dabei gewährleistet, dass eine Rolleninstanz weiterhin unveränderbar mit ihrem Kontext – der Teaminstanz – assoziiert ist. Im obigen Beispiel wird die Rolleninstanz weiterhin innerhalb des Teams verwaltet und ihr Adaptionsverhalten durch Aktivierung und Deaktivierung des Teams ein- bzw. ausgeschaltet.

Protected roles: Rollen mit dem Modifier *protected* können zwar außerhalb des Teams referenziert werden, jedoch ist ihr Typ – und damit ihre Schnittstelle – nicht sichtbar. Es besteht jedoch prinzipiell die Möglichkeit die Rollenschnittstelle mittels *reflection* zu verwenden und diese Kapselung somit zu durchbrechen, insofern dies durch die verwendete *java security policy* [21](6) zugelassen wird.

```
1  public team class Team {
2      protected class Role playedBy Base {
3      }
4
5      public Object getRoleInstance() {
6          ...
7      }
8  }
9  ...
10 Team teamInstance = new Team();
11 ...
12 Object roleInTeam = teamInstance.getRoleInstance();
```

Listing 4.2: Externe Referenzierbarkeit von protected roles

Confined roles: Für Rollen, welche die Klasse *Team.Confined* beerben garantiert der Compiler, dass zu keinem Zeitpunkt Referenzen auf Rolleninstanzen dieses Typs außerhalb des Teams existieren.

```
1 public team class Team {
2    protected class Role extends Confined playedBy Base {
3    }
4 }
```

Listing 4.3: Kapselung von confined roles

Eine Variante von confined roles sind *opaque roles*, welche ausschließlich die Exposition einer Rolleninstanz als Interface IConfined und von diesem abgeleitete Typen zulässt um die externe Sichtbarkeit der Rollenschnittstelle explizit auf die im entsprechenden Interface festgelegte Schnittstelle zu beschränken [24](§ 7.1).

4.2.4 Teamaktivierung

Die Aktivierung eines Teams hat eine Aktivierung aller callin-Bindings der Rollen des Teams zur Folge [24](§ 5). Ein Team kann für einen bestimmten oder alle Threads (global) aktiviert sein. Die Aktivierung erfolgt entweder explizit oder implizit durch Aufruf von Team-Methoden [24](§ 5.3 / 5.3)

4.3 Syntax der Implementierung des Object Teams Programmiermodells

Teams, Rollen und Basisobjekte sind herkömmliche Java-Klassen, welche ihre Eigenschaften durch die *Object Teams Infrastruktur* erhalten. Diese Infrastruktur besteht aus dem für Object Teams-Code zur compile-Zeit durch den *Object Teams Compiler* und für adaptierte Klassen zur Laufzeit durch den Object Teams Loadtime Weaver generierten Code sowie aus der Object Teams Laufzeit-Bibliothek. Object Teams ist somit eine standard Java-Applikation, deren spezielle Implementierung jedoch vor dem Anwender verborgen bleibt. Im Folgenden werden die durch Compiler und Load-Time Weaver eingeführten Infrastrukturelemente erläutert.

4.3.1 Team Infrastruktur

Eine Teamklasse besteht aus ihrer Klassendefinition sowie den in ihr deklarierten Rollenklassen. Jede Teamklasse erhält Infrastrukturcode zur Durchführung von Rollenverwaltung (Referenzierung von Rollen), Lifting (Rollen Erzeugung) und Team Aktivierung.

Rollenverwaltung

Eine Rolleninstanz wird durch genau ein Team verwaltet. Existiert eine Rolleninstanz, so existieren ebenfalls Instanzen ihres Teams und ihrer Basis. Das Team muss jederzeit in der Lage sein die adaptierende Rolleninstanz einer Basisinstanz zu finden, falls diese existiert. Zur Implementierung dieses Verhaltens wird für die Wurzel der Vererbungshierarchie jedes Rollentyps eines Teams eine *DoublyWeakHashMap* als *Rollen-Cache* generiert[10].

In diese werden Basis und Rolle eingetragen. Dadurch, dass sowohl Schlüssel (Basis) und Wert (Rolle) *weak references* sind beeinflusst dieser Cache nicht das Verhalten des Garbage Collectors. Im Gegenzug ist das Ergebnis von Operationen der Map nichtdeterministisch, da der Zustand der Map vom Verhalten des Garbage Collectors abhängt. Zur Implementierung der in Abbildung 4.2 dargestellten Mehrfachvererbung und zur Kapselung der Rollenschnittstellen wird jede Rollenklasse automatisch in ein Rolleninterface und eine Rollenimplementierung umgewandelt. Das Rolleninterface übernimmt dabei den Namen der Rollenklasse, während letztere einen Object Teams-spezifischen Präfix erhält. Auf diese Weise kann eine Rolle das Interface einer anderen Rolle implementieren und ihre Eigenschaften durch copy inheritance erben, während sie gleichzeitig eine extends-Relation zu einer weiteren Klasse besitzt (vgl. Seite 41, 4.2.1).

Lifting

Eine lifting-Operation liefert die Rolleninstanz einer durch den entsprechenden Rollentyp adaptierten Basis. Existiert noch keine Rolleninstanz, so wird diese erzeugt. Die lifting-Operation garantiert die gleiche Rolleninstanz als Ergebnis aufeinander folgender Aufrufe der lifting-Operation für

[10] Konkret wird der Rollen-Cache für den Typ der am wenigsten spezifischen Rolle der Vererbungshierarchie, welche eine playedBy-Beziehung besitzt, angelegt

die gleiche Basisinstanz [24](§ 2.3 a). Lifting wird in Form automatisch erzeugter *liftTo*-Methoden in das Team eingeführt. Für jeden Rollentyp des Teams existiert dabei eine liftTo-Methode, welche als Argument eine Instanz des durch die playedBy-Relation des Rollentyps referenzierten Basistyps und als Rückgabewert eine Instanz der Rolle liefert. Diese internen lifting-Methoden werden aufgerufen, sobald eine Rolleninstanz benötigt wird, bspw. bei callin-bindings. Die liftTo-Methode überprüft den Rollen-Cache für den gewünschten Rollentyp. Befindet sich dort für die übergebene Basis-Instanz keine Rolleninstanz, so wird die *create*-Methode für den gewünschten Rollentyp aufgerufen. Für smart lifting existieren *lift_dynamic*-Methoden, welche den zu erzeugenden Rollen-Typ bestimmen (vgl. Seite 42, 4.2.2), falls sich keine entsprechende Rollen-Instanz im Rollen-Cache befindet.

Die Rolle trägt sich bei ihrer Erzeugung in den Rollen-Cache Ihres Teams und in das *roleSet* ihrer Basisinstanz ein. Der Eintrag in das roleSet der Basis stellt sicher, dass die Rolleninstanz erst dann durch den Garbage Collector erfasst wird, wenn auch das Basisobjekt nicht mehr referenziert wird.

Auffinden von Rolleninstanzen

Über die öffentlichen Methoden *hasRole* und *getRole* können Anwender den Adaptionsstatus einer Basisklasse einsehen resp. eine Referenz auf eine adaptierende Rolle erhalten. Beide Methoden greifen auf den Rollen-Cache zu, erzeugen jedoch keine Rollen-Instanzen. Da eine Basis mehrere Rollen innerhalb eines Teams besitzen kann, existieren diese Methoden jeweils als Variante mit einem Typparameter, über welchen der gewünschte Rollentyp spezifiziert wird.

Team Aktivierung

Die Aktivierung eines Teams führt zum Eintrag des Teams in das _OT$activeTeams-Array der Basisklassen, zu denen eine playedBy-Relation von Rollentypen des Teams besteht. Das Team registriert sich im Falle von globaler Aktivierung zusätzlich beim *TeamThreadManager*, im Falle von threadspezifischer Aktivierung in einer eigenen private *WeakHashMap*, welche den threadspezifischen Aktivierungsstatus beinhaltet.

```
1  public class Team {
2      private DoublyWeakHashMap<Base, Role> _OT$cache_OT$Role;
3      private WeakHashMap<Thread, Boolean> _OT$activatedThreads;
4
5      protected interface Role {
6          ...
7      }
8
9      protected class __OT__Role implements Role /* playedBy Base */ {
10         // generated infrastructure code
11     }
12
13     public Role _OT$create$Role(Base base) {
14         return new $__OT__Role(this, base);
15     }
16
17     public Object getRole(Object base) {
18             return _OT$cache_OT$Role.get(base)
19     }
20
21     public Object getRole(Object base, Class roleClass) {
22         Map<Base, Role> typeMap = lookupCacheForType(roleClass);
23          ...
24         return typeMap.get(base);
25     }
26
27     public boolean hasRole(Object base) {
28         return _OT$cache_OT$Role.containsKey(base);
29     }
30
31     public boolean hasRole(Object base, Class roleClass) {
32         Map<Base, Role> typeMap = lookupCacheForType(roleClass);
33         ...
34         return typeMap.containsKey(base);
35     }
36
37     public Role _OT$liftTo$Role(Base instance) {
38         Role role = getRole(instance);
39         if (role == null) {
40             role = _OT$create$Role(instance);
41         }
42         return role;
43     }
44
```

```
45     public void activate() {
46       this.activate(Thread.currentThread());
47     }
48
49     public void activate(Thread thread) {
50         // register at thread manager
51         // set globally active
52     }
53 }
54
55 public class Base {
56     private static Team[] _OT$activeTeams
57     private HashSet _OT$roleSet
58
59     public void _OT$removeRole(Object role) {
60         _OT$roleSet.remove(role);
61     }
62
63     public void _OT$addRole(Object role) {
64         _OT$roleSet.add(role);
65     }
66
67     public static synchronized void _OT$addTeam(Team team) {
68         // Add team to _OT$activeTeams.
69     }
70
71     public static synchronized void _OT$removeTeam(Team team) {
72         // Remove team from _OT$activeTeams.
73     }
74 }
```

Listing 4.4: Durch Code-Generierung erzeugte Team Infrastruktur (Implementierung in Pseudocode)

Die in Listing 4.4 dargestellten Infrastrukturelemente der Basis sowie die für die Registrierung an den Basisklassen notwendige Logik der Teamklasse werden dabei durch den Load-Time Weaver erzeugt, um jedwede Klassen, insbesonders bereits kompilierte Klassen aus beliebigen Bibliotheken, adaptieren zu können.

4.3.2 Rollen und Basis Infrastruktur

Die Infrastruktur einer Rolle besteht aus den oben erwähnten generierten Feldern zur Speicherung von Rolleninstanzen und Teaminstanzen. Die Adaption durch eine Rolle mittels playedBy-Relation führt des Weiteren zur Generierung von Dispatch-Code, welcher Methodenaufrufe an die Rolleninstanz weiterleitet. In folgendem Beispiel adaptiert *Role* eine Methode *getName* von *Base*. Der Load-Time Weaver ermöglicht den Dispatch, indem die original getName()-Methode umbenannt und an ihrer Stelle eine Methode mit identischer Signatur erzeugt wird, welche ggf. an eine existierende Rolle weiterleitet.

```
1  public class Team {
2     protected interface Role {
3        ...
4     }
5     protected class $__OT__Role implements Role /* playedBy Base */ {
6        private final this$0;
7        private final _OT$base;
8
9        public Role(Team outerInstance, Base base) {
10           this$0 = outerInstance;
11           _OT$base = base;
12       }
13
14       public String getName() {
15          ...
16          // Proceed with call chain in base
17          return "Adapted" + _OT$base....();
18       }
19    }
20 }
21
22 public class Base {
23    ...
24    private String name;
25
26    public String getName() {
27       // Get team with adapting role
28       // invoke dispatch to team chain
29    }
30
```

```
31     public String _OT$getName$orig() {
32         return name;
33     }
34 }
```

Listing 4.5: Durch Code-Generierung erzeugte Rollen-Infrastruktur (Implementierung in Pseudocode)

4.4 Die Object Teams Applikation

Die in Listing 4.5 und Diagramm 4.5 dargestellte, aus benutzerspezifischen Klassen, durch Code-Generierung in Compiler und Load-Time Weaver erzeugter Infrastruktur und einer Object Teams Laufzeit Bibliothek bestehende Applikation wird im Weiteren als *Object Teams Applikation* bezeichnet. Die Geschäftslogik dieser Applikation implementiert die in der Object Teams Sprachspezifikation [24] beschriebenen Eigenschaften. Zur ordnungsgemäßen Funktion der Object Teams Applikation müssen folgende Invarianten erfüllt sein:

INV1: Jede Rolleninstanz referenziert eine Instanz ihrer deklarierenden Klasse (der Team-Klasse) über das Feld *this$0*.

INV2: Jede Rolleninstanz referenziert die von ihr adaptierte Instanz der Basisklasse über das Feld *_OT$base*.

INV3: Die mit einer Rolleninstanz assoziierte Teaminstanz besitzt den Eintrag $\{Basisinstanz \rightarrow Rolleninstanz\}$ im typspezifischen Rollen-Cache *_OT$cache_OT$RollenTyp* , wobei die Basisinstanz die von der Rolle referenzierte Basisinstanz ist.

INV4: Ist eine Rolleninstanz mit einer Basisinstanz assoziiert, so wird diese Rolleninstanz im Set des Feldes *_OT$roleSet* der Basisinstanz referenziert.

INV5: Wird eine Basisinstanz durch den Garbage Collector bereinigt, so wird ihre Rolle ebenfalls bereinigt.

INV6: Eine Rolleninstanz ist immer eine Instanz einer in ihrer Teamklasse deklarierten Rollenklasse.

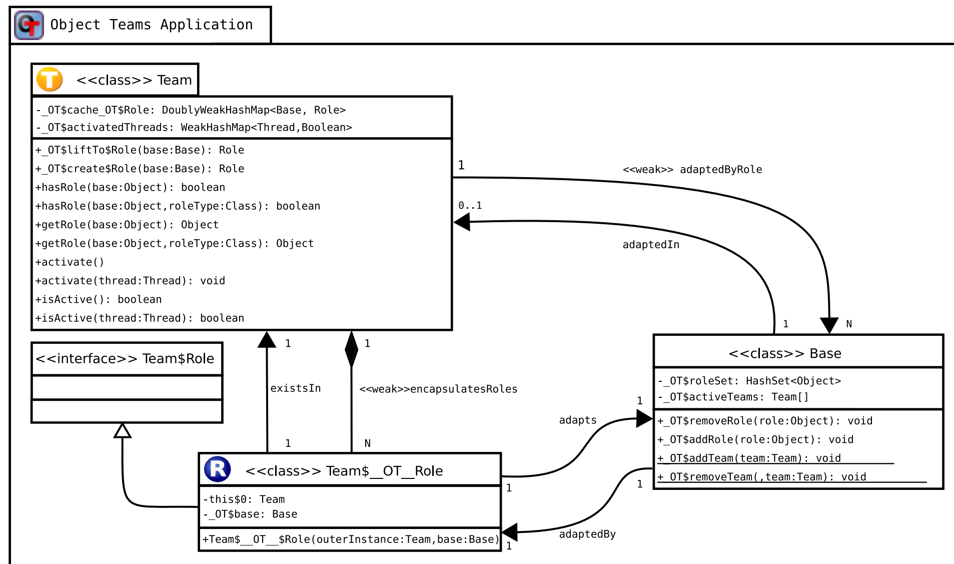

Abbildung 4.5: UML-Diagramm einer Object Teams Applikation mit konkreten Team, Rolle und Basis Implementierungen

Eine Persistenzlösung für die Object Teams Applikation muss diese Invarianten berücksichtigen, da es sich vorwiegend um Anforderungen an die *referentielle Integrität* des Modells handelt, welche durch das relationale Modell explizit beschreibbar sind (UML Diagramm 4.5). Gleichzeitig stellt der Einsatz einer standardisierten Persistenzlösung Anforderungen an Struktur und Semantik der Klassen der Object Teams Applikation. Je weniger diese Anforderungen die Gestaltungsfreiheit des Modells einschränken und je mehr Möglichkeiten die Persistenzlösung zum Anpassen ihres Verhaltens bietet, desto besser ist diese zur Adaption für Object Teams geeignet.

4.5 Die Java Persistence API als Persistenzlösung für die Object Teams Applikation

Wie in Kapitel 3 gezeigt existieren ausgereifte Standards und Implementierungen für Persistenz in Java. Unter Berücksichtigung des hohen Verbreitungsgrades relationaler Datenbanksysteme [27] bilden Systeme für Objekt-Relationales Mapping (ORM) die attraktivste Gruppe von Persistenzlösungen. Mit der Java Persistence API (JPA) besitzen diese ORM einen gemeinsamen Standard, welcher sich durch eine mächtige, dennoch einfache Syntax

sowie einen hohen Reifegrad auszeichnet und eine hohe Akzeptanz in der Java-Community besitzt.

Wie in Abbildung 4.6 dargestellt reduziert die JPA die Dominanz des Persistenzaspektes auf syntaktischer Ebene stark. Dieser Effekt basiert vorwiegend auf der Verwendung von Defaults und der Auszeichnung durch semantische Annotationen (vgl. Seite 29, 3.4.1). Da letztere problemlos mit den Object Teams spezifischen Typen verwendet werden können wurde angenommen, dass es auf syntaktischer Ebene nicht zu Konflikten zwischen dem Object Teams Programmiermodell und JPA-spezifischem Code kommen würde.

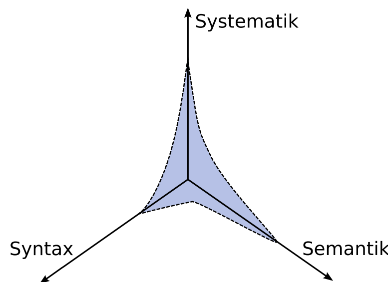

Abbildung 4.6: Ausprägung des Persistenz-Aspekts bei Verwendung der JPA

Hingegen ist die Dominanz der Persistenzeigenschaft auf systematischer und semantischer Ebene weiterhin stark. Die Auflösung der Konflikte zwischen dem relationalen Mapping einerseits und den semantischen Anforderungen der Object Teams Applikation andererseits wurden somit als integraler Bestandteil der Implementierung vorgesehen. Durch die exakte Spezifikation der Semantik ihrer Implementierungen bietet die JPA jedoch den Vorteil, mögliche Konflikte bereits im Vorfeld abschätzbar zu machen.

Die systematische Dimension hingegen ist vor allem bei der Umsetzung größerer persistierender Anwendungen relevant und spielte im Kontext der Implementierung der Persistenzlösung keine bedeutende Rolle.

Die Verwendung der JPA hat des Weiteren den entscheidenden Vorteil, dass mehrere JPA-Implementierungen als Kandidaten zur Adaption für Object Teams zur Verfügung stehen (vgl. Seite 27, 3.4). In dieser Arbeit wurde

die Referenzimplementierung der JPA, Eclipselink, verwendet. Diese Implementierung zeichnet sich durch ein hohes Maß an Standardkonformität und einen hohen Reifegrad aus – nicht zuletzt, da sie seit 1996 kontinuierlich weiterentwickelt wird. Zudem steht Eclipselink – wie auch Object Teams – unter der *Eclipse Public License (EPL)*.

4.6 Allgemeine Anforderungen an eine JPA-Implementierung für Object Teams

Die im Folgenden definierten Anforderungen sollten gewährleisten, dass die besonderen Stärken der JPA – Leichtgewichtigkeit, programming by exception, geringe Einschränkungen der Modellierungsfreiheit, semantische Auszeichnungen – auch für die Verwendung mit Object Teams bestehen blieben. Im weiteren Verlauf dieser Arbeit erhielt jede definierte Anforderung einen eindeutigen Schlüssel im Format A-<Nummer>, auf welchen später Bezug genommen werden konnte.

A1 Transparenz. Anwender der JPA für Object Teams sollen in die Lage versetzt werden, Teams, Rollen und Basisklassen so auszuzeichnen, als würde es sich um einfache Java-Klassen (POJOs) handeln. Die automatisch generierte, sprachinterne Infrastruktur muss dabei vor dem Anwender verborgen bleiben.

A2 Schutz der Invarianten. Die Wahrung der Invarianten der Object Teams Applikation, insbesondere die Herstellung und Wahrung der Referentiellen Integrität zwischen den Object Teams-spezifischen Typen ist Aufgabe der Persistenzlösung.

A3 Konformität. Die Verwendung der JPA sollte keinen Beschränkungen unterliegen. Ebenso soll die Einführung neuer, nicht in der Spezifikation definierter Auszeichnungsformen vermieden werden. Nur auf diese Weise kann gewährleistet werden, dass die entstehende Lösung von der Verbreitung des JPA-Standards und dessen Akzeptanz in der Java-Community profitieren kann.

A4 Datenunabhängigkeit. Während die Kopplung an eine konkrete JPA-Implementierung unvermeidbar ist, muss die Unabhängigkeit von der

verwendeten relationalen Datenbank und dem eingesetzten SQL-Dialekt erhalten werden.

A5 Explizite Persistenz. Die Persistenzeigenschaften aller Typen und ihrer Attribute müssen entsprechend der JPA Spezifikation sichtbar und kontrollierbar sein.

4.7 Analyse der Anforderungen von Object Teams an eine JPA Implementierung

Wie in 4.3 (Seite 45) gezeigt besteht eine Object Teams Applikation aus benutzerspezifischen und generierten Code-Bestandteilen, welche neue, zusammengesetzte Typen modellieren. Ebenso gelten zur Laufzeit die auf Seite 51 (4.4) definierten Invarianten. Eine JPA-Implementierung muss daher die Struktur der Applikation in Form der neuen Typen Team, Rolle und Basis, als auch ihre Semantik mit den beschriebenen Invarianten unterstützen.

4.7.1 Unterstützung der Object Teams-Strukturen

Team

Wird eine Klasse als Entity ausgezeichnet, so sind alle ihre Felder persistent, es sei denn sie sind als Transient ausgezeichnet [13](2.1.1). Die in Listing 4.4 dargestellte Team Infrastruktur enthält jedoch nicht persistierbare Felder:

Die DoublyWeakHashMap der Rollen-Caches *_OT$cache_OT$RollenTyp* liefern zum einen nichtdeterministische Ergebnisse beim Zugriff auf ihre Elemente, zum anderen ist die Anwesenheit einer Rolleninstanz und ihrer Basis im Cache kein hinreichender Hinweis darauf, dass die Speicherung dieser Objekte erwünscht ist. Des Weiteren besteht für einen Anwender nicht die Möglichkeit, den generierten Code mit weiteren Annotationen – beispielsweise zur Steuerung des Kaskadierungsverhaltens[11] – auszuzeichnen.

[11]Kaskadierung bezeichnet das Verhalten von Datenbanksystemen, eine Operation auf einem Tupel ebenfalls auf von diesem Tupel referenzierten Tupeln auszuführen (vgl. Seite 18, 2.3.4). Im Kontext der JPA bedeutet Kaskadierung die Übertragung von CRUD-Operationen des EntityManagers auf abhängige Entities ([13] 3.2).

Der Aktivierungszustand-Cache *_OT$activatedThreads* enthält einfache Thread-Instanzen, welche per se nicht persistierbar sind, da sie Bestandteil des Ausführungskontextes der JVM und nicht Teil der programmspezifischen Modellierungselemente sind. Da außer diesen keine weiteren generierten Felder persistenzrelevant sind gilt:

A6 Die generierten Felder eines Teams sind transient.

Rolle

Rollenklassen enthalten zwei nicht-statische, generierte Felder: Das durch den Java-Compiler erzeugte Feld zur Referenzierung der Instanz der deklarierenden Klasse, *this$0* und das generierte Feld *_OT$base*, welches die adaptierte Basisinstanz referenziert. Aufgrund der Invarianten INV1 und INV2 (vgl. Seite 51) gilt:

A7 Die Felder *this$0* und _OT$base sind persistent und repräsentieren eine 1:1 Beziehung zu Team bzw. Basis.

A8 Ist eine Rollenklasse eine Entity, so sind auch ihre Teamklasse und die durch sie adaptierte Basisklasse Entities.

Basis

Das generierte Array aktivierter Teams einer Basis ist statisch und als solches nicht persistierbar. Das Array darf zudem nicht persistent sein, da es den thread-spezifischen Aktivierungszustand der Teams widerspiegelt, welcher durch seine Abhängigkeit von den Threads der JVM nicht persistierbar ist.

Das _OT$roleSet-Feld ist ebenfalls nicht persistierbar, da es eine undefinierte Anzahl unterschiedlicher Rollentypen referenziert, bei welchen es sich nicht um Entities handeln muss. Persistente Felder müssen jedoch einen eindeutigen Entity-Typ referenzieren [13](2.1.1). Zudem ist die Anwesenheit einer Rolle im roleSet kein hinreichender Hinweis darauf, dass diese persistiert werden soll. Tatsächlich dient dieser Mechanismus ausschließlich dem Erfüllen der Invariante INV5 (vgl. Seite 51). Eine direkte Relation zwischen Basis und Rolle würde zudem die Isolation adaptierter Modellteile von den

adaptieren Rollen durchbrechen. Da keine weiteren generierten Felder existieren gilt:

A9 Die generierten Felder einer Basisklasse sind transient.

4.7.2 Unterstützung der Object Teams-Semantik

Jede persistierte Entity kann bspw. mittels eines JPQL-Queries separat aus der Persistenzschicht geladen werden [13](4). Bestehen besondere Anforderungen an die referentielle Integrität geladener Entities, so müssen diese explizit formuliert werden. Da die auf Seite 51 (4.4) definierten Invarianten zum Großteil die referentielle Integrität betreffen benötigen die Typen der Object Teams Applikation eine entsprechende Auszeichnung, welche Aufgrund der Anforderung A1 (vgl. Seite 54) durch die JPA-Implementierung bereitgestellt werden muss. Gemäß den Invarianten (vgl. Seite 51) gelten weitere, typspezifische Anforderungen.

Team-spezifische Semantik

Ein Team kapselt eine Menge von Rollen. Zwischen diesen Rollen und dem Team besteht eine 1:N Beziehung (vgl. Seite 51, INV1 und INV3). Diese Beziehung ist eine *Komposition* (Abbildung 4.5). Eine Rolle existiert daher nur, solange ihr Team existiert. Während die referentielle Integrität in der Object Teams Applikation durch die Referenzbeziehungen zwischen Rollen und Teams sichergestellt ist, muss diese im relationalen Modell explizit modelliert werden (vgl. Seite 18, 2.3.4). Daher gilt für persistierte Team-Entities:

A10 Wird eine Team-Entity gelöscht, so werden alle Rollen-Entities des Teams gelöscht.

Rollen-spezifische Semantik

Gemäß INV1 und INV2 (vgl. Seite 51) und A8 referenziert jede Rolle ein Team und eine Basis (Abbildung 4.5). Diese Basisinstanz und Teaminstanz können jedoch länger als die Rolleninstanz existieren. Daher gilt:

A11 Das Löschen einer Rolle kaskadiert nicht zu Team oder Basis.

A12 Das persistieren einer Rolle kaskadiert automatisch zu Team und Basis (A8).

A13 Wird eine Rolle geladen, so werden ihr Team und ihre Basis geladen.

A14 Wird eine Rolle geladen, so wird diese in das *_OT$roleSet*-Feld ihrer Basis und in den Rollen-Cache eingetragen (vgl. Seite 51, INV3 und INV4) bevor sie im Persistence Context sichtbar ist.

Teams, Rollen und Lazy-Loading

Zur Überwindung des Memory-Mismatch (vgl. Seite 18, 2.3.4) sieht die JPA die Annotation persistierbarer Relationen mit einem *FetchType* vor, welcher den Zeitpunkt des Nachladens der referenzierten Entities steuert. Ist der FetchType *lazy*, so werden die referenzierten Entities genau dann aus der Datenbank geladen, wenn auf diese zugegriffen wird (Lazy-Loading).

Die Implementierung des Lazy-Loading erfolgt dabei abhängig von der Art der annotierten Referenz. Bei mehrstelligen Relationen (bspw. Collections) wird die entsprechende Collection so erweitert, dass die durch sie referenzierten Objekte erst beim Zugriff auf Elemente der Relation sukzessive geladen werden. Bei einstelligen Relationen können typkompatible stellvertreter-Objekte (Proxies) verwendet werden, welche das eigentliche Zielobjekt beim Zugriff auf ihre Eigenschaften aus der Datenbank laden und alle Zugriff an dieses delegieren. Alternativ werden Lazy-Loading Aspekte um die Zugriffsmethoden auf die persistenten Felder gewebt, welche das Objekt beim Aufruf der Methode nachladen.

Während diese Strategien für normale Referenzbeziehungen keine Auswirkungen auf die Semantik haben, bildet die Relation einer Rolle zu ihrem Team und ihrer Basis eine explizite Ausnahme.

Eine Rolle wird weder von ihrem Team noch von ihrer Basis explizit referenziert[12]. Stattdessen versucht das Team bei Aufruf der lifting-Methode die geforderte Rolle aus dem Rollen-Cache zu laden. Existiert die gewünschte

[12]Der Eintrag der Rollenreferenz in der Basis dient der Zusicherung von INV5, auf die eingetragene Referenz wird jedoch nicht zugegriffen.

Rolle nicht, so wird eine neue Rollen-Instanz erzeugt und in den Cache eingetragen. Da weder das roleSet noch der Rollen-Cache persistent sein können erfolgt somit kein Zugriff auf eine möglicherweise existierende persistente Referenz auf eine Rollen-Entity. Da der indirekte Rollenzugriff die bestehenden lazy-loading-Strategien umgeht benötigt eine JPA-Implementierung für die Object Teams Applikation eine eigene Strategie für das Lazy-Loading von Rollen-Entities, welche diese zum Zeitpunkt des Liftings nachladen kann.

A15 Persistierte Rollen-Entities werden zum Zeitpunkt des lifting-Aufrufes für ihre Basis-Entity geladen.

Basis-spezifische Semantik

Aus der Begrenzung der Lebenszeit einer Rolle auf die Lebenszeit ihrer Basis (vgl. Seite 51, INV5) und der Notwendigkeit der expliziten Modellierung der Referenzsemantik im relationalen Modell (vgl. Seite 18, 18) folgt:

A16 Das Löschen einer Basis-Entity kaskadiert zu allen Rollen-Entities, welche die Basis-Entity durch das Feld *_OT$base* referenzieren.

4.8 Analyse der Anforderungen der JPA an die Object Teams Applikation

Die Anforderungen der JPA an Entities sind klar definiert [13](2.1). Die Object Teams Applikation ist bis auf die Modellierung von Rollen-Klassen zu diesen Anforderungen konform.

4.8.1 Modellierung von Rollen-Klassen

Eine Rolle ist eine innere, nicht statische Klasse. Als solche besitzt sie keinen default-Konstruktor, sondern kann nur erzeugt werden, wenn eine Instanz der deklarierenden Klasse existiert. Wie in Listing 4.5 gezeigt, wird die Referenz auf diese äußere Instanz in dem *final*-deklarierten Feld *this$0* abgelegt. Object Teams führt neben dieser Referenz ein weiteres Konstruktor-Argument in Form der Referenz auf die Basis-Instanz der Rolle ein, welche in dem ebenfalls *final* deklarierten Feld *_Ot$base* abgelegt wird.

Diese Struktur verstößt gegen zwei Bedingungen der JPA [13](2.1):

- Jede Entity besitzt einen default-Konstruktor
- Persistente Felder einer Entity sind nicht *final*

Rollenklassen benötigen daher eine adaptierte Instanziierungs- und Initialisierungsstragie der JPA Implementierung.

A17 Rollen-Instanzen werden durch die JPA-Implementierung auch ohne default-Konstruktor erzeugt.

A18 Die JPA-Implementierung ignoriert die *final*-Deklaration der Rollen-Felder *this$0* und *_OT$base*.

Die automatische Konversion einer Rollenklasse in Rollen-Interface und Rollen-Implementierung hat zur Folge, dass alle Referenzen auf Rollen das Rollen-Interface referenzieren. Die JPA erlaubt jedoch nur Top-level Klassen als Referenztypen und Entities [13](2.1). Die JPA-Implementierung muss daher alle Konfigurationen, welche sich auf ein Rollen-Interface beziehen als Konfiguration der Rollen-Implementierung auffassen und alle Referenzen auf Rollen-Interfaces als Referenzen auf die Rollen-Implementierung verstehen.

A19 Konfigurative Bezüge auf Rollen-Interfaces werden als Konfiguration der Rollen-Implementierung aufgefasst.

A20 Referenzen auf Rollen-Interfaces werden als Referenzen auf Rollen-Implementierungen aufgefasst.

4.9 Adaption von Eclipselink als Persistenzimplementierung für Object Teams

4.9.1 Adaption von Entity-Metadaten in Eclipselink

Die Erfüllung der in 4.6 bis 4.8 (Seiten 54, 59) erfassten Anforderungen bedingen ein Reihe von Anpassungen in Eclipselink. Diese Anpassungen lassen sich grob in zwei Teilbereiche unterteilen: Anpassung der Struktur und Anpassung des Verhaltens der Entity-Repräsentationen.

Das Auffinden geeigneter Adaptionspunkte und die darauf aufbauende Ableitung der Struktur einer adaptierten Persistenzlösung setzt eine genaue Kenntnis des internen Aufbaus der Eclipselink Implementierung voraus. Die verfügbare Dokumentation des internen Aufbaus beschränkte sich jedoch auf einige konzeptionelle Informationen, während die Code-Basis mehr als 2000 Klassen mit einem Gesamtvolumen von mehr als zwölf Megabytes Sourcecode umfasst. Die vorliegenden Analyseergebnisse wurden daher sowohl aus der verfügbaren Dokumentation als auch aus der Laufzeitanalyse der Eclipselink Implementierung während des Ausführens von Testcode gewonnen. Gleichfalls wurden die im Folgenden beschriebenen Klassen aufgrund ihrer Größe und Komplexität notwendigerweise auf die Darstellung ihrer zentralen Eigenschaften reduziert.

Gemäß der in 4.7 und 4.8 (Seiten 55, 59) definierten Anforderungen müssen die Instanziierung von Rollen, die Persistierbarkeit interner Felder sowie das Kaskadierungsverhalten von Basis, Team und Rolle adaptiert werden. Für jede dieser Anforderungen existieren entsprechende Adaptionspunkte in der Eclipselink Implementierung.

Adaption für Entity-Verhalten und Rollen-Interface

Wie in Abbildung 4.12 dargestellt, wird jeder einer Persistence Unit zugeordnete Entity-Typ durch einen *ClassDescriptor* repräsentiert, welcher sowohl strukturelle als auch semantischen Eigenschaften der Entity beschreibt.

Für die Adaption der Persistenzsemantik eines Entity-Typs besitzt jeder ClassDescriptor einen *DescriptorEventManager*, welcher die Registrierung eines *DescriptorEventListener* erlaubt (Abbildung 4.7). Dieser Listener wird synchron über alle Ereignisse des Lebenszyklus einer persistenten Instanz des Entity-Typs benachrichtigt. Damit bildet er eine ideale, stabile Schnittstelle zur Anpassung des Verhaltens ausgewählter Entity-Typen für Object Teams. Insbesondere die Kaskadierung von delete-Operationen kann durch einen geeigneten Listener implementiert werden.

Gemäß Anforderung A19 (vgl. Seite 59, 4.8) darf ein ClassDescriptor keine Rollen-Interfaces beschreiben. ClassDescriptor-Instanzen werden in Eclipselink zu Beginn der bootstrapping-Phase (vgl. Seite 30) durch den *PersistenceUnitProcessor* erzeugt. Dieser liest die in der persistence.xml einge-

Abbildung 4.7: EventListener-Architektur des ClassDescriptors in Eclipselink

tragenen Klassennamen ein und lädt die entsprechenden mit einem geeigneten Classloader. Die geladene Klasse wird während des pre-deployments zur Erzeugung von ClassDescriptor-Instanzen durch die EntityManagerFactory-Implementierungen verwendet.

Die durch den Object Teams Anwender in die persistence.xml eingetragenen Klassennamen zeigen für persistente Rollen jedoch auf das Rollen-Interface, da die ursprüngliche Klasse zur Compile-Zeit durch dieses ersetzt wurde. Aufgrund der Anforderung A1 (vgl. Seite 54) darf der Anwender nicht dazu gezwungen sein die Object Teams-interne Umbenennung der Rollenimplemtierung manuell in die persistence.xml zu übernehmen. Stattdessen bietet die in Abbildung 4.8 gezeigte ClassLoading-Abstraktion *loadClass(...)* des PersistenceUnitProcessor den geeigneten Adaptionspunkt zur Konversion von Rollen-Interfaces in ihre Rollen-Implementierungen, da diese für die Auflösung der in der persistence.xml enthaltenen Klassennamen verantwortlich ist.

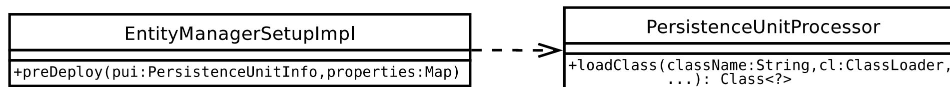

Abbildung 4.8: Classloading-Abstraktion durch den PersistenceUnitProcessor in Eclipselink

Adaption zur Behandlung generierter Felder

Die Felder einer Entity werden in Eclipselink durch die Klasse *MetadataField* repräsentiert. Diese Felder einer Entity-Klasse sind Bestandteil des *ClassAccessors*, welcher den Zugriff auf die persistenten Eigenschaften von Entity-Typen abstrahiert. ClassAccessor und ClassDescriptor werden in der Klasse *MetadataClass* als gemeinsame Metadaten zusammengefasst.

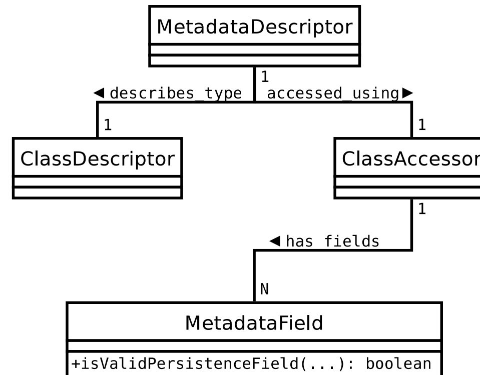

Abbildung 4.9: MetadataField, ClassAccessor und ClassDescriptor in Eclipselink

Felder, deren MetadataField für die Methode isValidPersistenceField() *false* liefert, werden von Eclipselink als transient behandelt. MetadataField bildet somit eine geeignete Schnittstelle zur Ausblendung generierter Felder.

Felder, welche eine einzelne oder eine Menge von Rollen-Interfaces referenzieren, müssen gemäß A20 (vgl. Seite 59) die entsprechenden Rollen-Implementierungen referenzieren. Der von einem Feld referenzierte Typ wird über die Methode *getRawType()* von MetadataField aufgelöst, welche Geschäftslogik zur Auflösung des generischen Typs (component type) mehrstelliger Relationen (Collections) enthält. Diese Methode bildet somit den geeigneten Adaptionspunkt für A20.

Die Handhabung von Referenzen auf Rollen-Interfaces benötigt jedoch noch weitere Anpassungen. Neben den Metadaten für Felder existieren ebenfalls abstrakte Repräsentationen der Mapping-Typen (bspw. OneToOne, OneToMany). Diese Metadaten exponieren ebenfalls den konkreten Typ ihrer Relation durch die Methode *getReferenceClass()* der Zugriffsklasse *MappingAccessor*, welche ebenfalls einen geigneten Adaptionspunkt bildet (Abbildung 4.10).

4.9.2 Adaption zur Instanziierung von Rollen

Anforderung A17 und A18 (vgl. Seite 59, 4.8) fordern eine angepasste Instanziierungslogik für Rollen-Entities. Mit der jedem ClassDescriptor zuge-

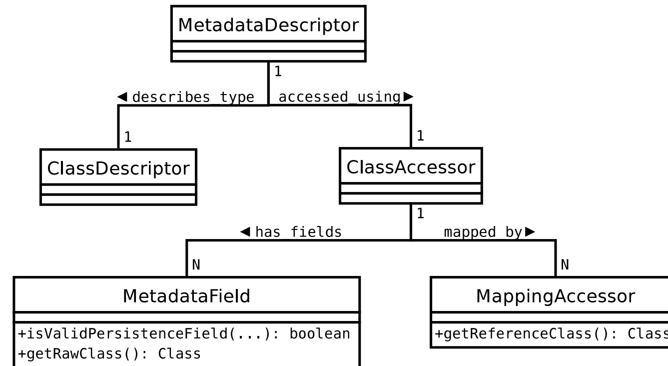

Abbildung 4.10: MappingAccessor, MetadataField, ClassAccessor und ClassDescriptor in Eclipselink

ordneten *InstantiationPolicy* existiert ein geeigneter Adaptionspunkt. Eine rollenspezifische Strategie kann die in Abbildung 4.11 gezeigte typspezifische Methode *buildNewInstance()* adaptieren um Rolleninstanzen auch ohne Verwendung eines default-Konstruktors zu erzeugen.

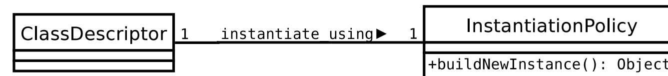

Abbildung 4.11: InstantiationPolicy und ClassDescriptor in Eclipselink

4.9.3 Erweiterung von Eclipselink durch Object Teams

Die vorangegangene Betrachtung der Adaptionspunkte zeigt die starke Querschnittlichkeit der notwendigen Anpassungen in der Eclipselink JPA Implementierung. Da es weder sinnvoll noch handhabbar wäre, ein so umfangreiches Projekt wie Eclipselink in das Sourcecodemanagement von Object Teams aufzunehmen um das gewünschte Verhalten im Quellcode der Implementierung anzupassen, ist eine modulare, aspektorientierte Lösung der Adaption bereits kompilierter Klassen der Eclipselink Bibliothek wesentlich attraktiver.

Eine solche Adaption kann typsicher und mit vergleichsweise wenig Aufwand durch Verwendung von Object Teams selbst erfolgen. Dies hat gegenüber herkömmlichen AOP-Lösungen nicht nur den Vorteil der typsiche-

Abbildung 4.12: Kern der Entity-Metadaten Repräsentation in Eclipselink. (Quelle: Eclipselink Dokumentation [31])

ren Programmierung gegen die Adaptionspunkte und der gekapselten Modellierung des Verhaltens adaptierter Klassen durch Rollen, sondern macht auch die Verwendung weiterer AOP-Programmiermodelle überflüssig. Zur Herstellung des geforderten Verhaltens werden somit Teams verwendet, welche Eclipselink durch die Adaption der entsprechenden Klassen als Rollen erweitern.

Im Laufe der Analyse wurden mit struktureller und semantischer Adaption zwei inhaltliche Schwerpunkte identifiziert. Diese werden im Weiteren durch zwei Teamklassen, *SemanticsAdapter* und *StructureAdapter*, repräsentiert. Dabei enthält der SemanticsAdapter die Adaption des Entity-Verhaltens gemäß der Anforderungen A10-A12, A14 und A16-A18, während

der StructureAdapter die verbliebenen, an die Struktur der Entities gerichteten Anforderungen erfüllt. Wie in Abbildung 4.13 gezeigt, werden dafür in den Teams jeweils Rollen für die zuvor als Adaptionspunkte identifizierten Klassen erstellt, welche die relevanten Methoden entweder durch eigene Geschäftslogik ersetzen (replace callin binding) oder zusätzliche Geschäftslogik ausführen (before/after callin binding).

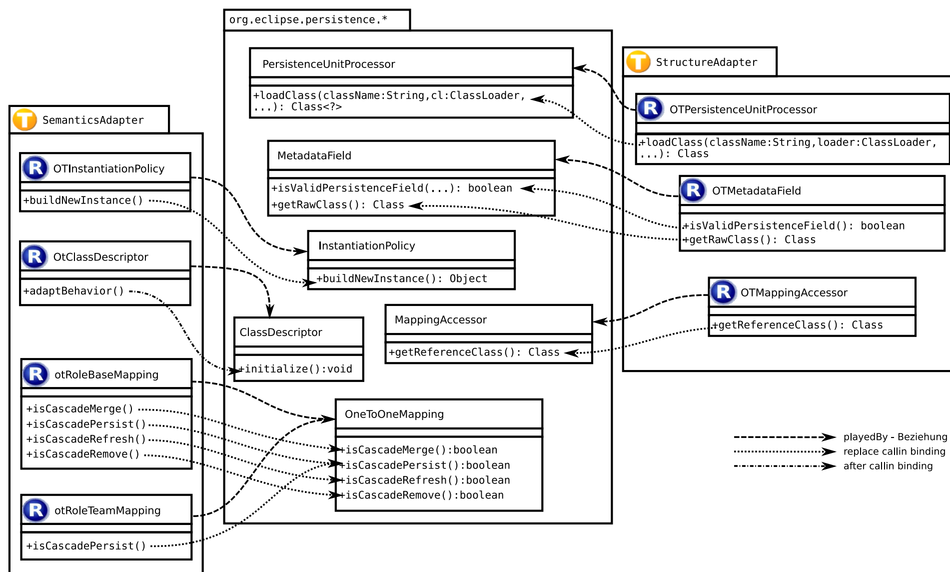

Abbildung 4.13: Schematische Darstellung der Adaption von Eclipselink durch Object Teams

Semantische Adaption

OTInstantiationPolicy Diese von der Eclipselink InstantiationPolicy gespielte Rolle hat zur Aufgabe, Instanzen von Rollen-Typen zu erzeugen. Diese Instanzen müssen auch dann erzeugt werden können, wenn weder ein default-Konstruktor noch Team und Basis als Referenz zur Verfügung stehen (A17, Seite 59). Dies ist notwendig, da eine JPA Implementierung persistierte Entities lädt, indem diese zunächst instanziiert und dann mit den persistierten Daten befüllt werden. Die Felder der durch die OTInstantiationPolicy erzeugte Rollen-Instanz dürfen dabei nicht initialisiert werden, da es sich – zumindest bei den Referenzen zu Team und Basis (this$0 und _OT$base) – um *final* Felder handelt, welche nur einmal initialisiert werden

können (A18, Seite 59).

OTClassDescriptor Der OTClassDescriptor stellt die Kaskadierung des Löschens von Teams an ihre Rollen (A10, Seite 57), das Kaskadieren des Löschens von Basen an ihre Rollen (A16, Seite 59) sowie die Registrierung einer neu geladenen Rollen in Team und Basis (A14, Seite 57) zur Verfügung. Zu diesem Zweck verwendet der OTClassDescriptor den in 4.9.1 (Seite 61) beschriebene EventListener-Mechanismus, indem automatisch geeignete EventListener-Implementierungen registriert werden, welche beim Löschen resp. Laden der entsprechenden Entities die Kaskadierung resp. Registrierung ausführen. Die Registrierung der EventListener erfolgt dabei nach Ausführung der *initialize*-Methode des ClassDescriptors, da erst zu diesem Zeitpunkt sichergestellt ist, dass die EventListener-Architektur vollständig zur Verfügung steht.

OTRoleBaseMapping Durch die Adaption der Metadaten, welche die Relation zwischen einer Rolle und ihrer Basis repräsentieren, kann die Kaskadierung des Löschens einer Rolle an ihre Basis explizit unterbunden werden. Dazu liefert die Methode *isCascadeRemove false* zurück (A11, Seite 57). Zusätzlich ist es sinnvoll (wenn auch nicht notwendig), die Operationen *merge*, *persist* und *refresh* [13](3.2 ff.) von einer Rolle an ihre Basis kaskadieren zu lassen, da eine Rolle nur dann persistiert werden kann, wenn ihre Basis persistiert wurde. Zudem greift die Rolle transparent auf den Zustand ihrer Basis zu, was eine Synchronisation der Zustände von Rolle und Basis wünschenswert macht.

OTRoleTeamMapping Wie das OTRoleBaseMapping darf das Löschen einer Rolle nicht zu dem mit der Rolle verbundenen Team kaskadieren (A11, Seite 57). Eine Kaskadierung der Operationen refresh, merge oder persist ist jedoch aufgrund des zu erwartenden Performance-Verlustes durch die potentiell große Menge transitiver Abhängigkeiten eines Teams nicht wünschenswert. Zudem sind die möglichen Abhängigkeiten einer Rolle vom Zustand ihres Teams weniger transparent als die durch callout-bindings oder base-Aufrufe gegebenen Abhängigkeiten einer Rolle vom Zustand ihrer Basis.

Strukturelle Adaption

OTPersistenceUnitProcessor Zur Erfüllung von A19 und A20 (vgl. Seite 59, 4.8.1) werden alle Typnamen der Persistence Unit, welche auf ein Rollen-Interface zeigen zu der entsprechenden Rollen-Implementierung aufgelöst. Dazu ersetzt der OTPersistenceUnitProcessor die *loadClass(...)*-Methode der Basis. Ist ein von der Basis geladener Typ ein Rollen-Interface, so wird die Implementierung des Interfaces geladen, welche in der deklarierenden Klasse des Rollen-Interfaces existiert (vgl. Seite 51, INV6). Diese Implementierung wird anstelle des Interfaces von der Methode zurückgeliefert. So ist sichergestellt, dass nicht das automatisch erzeugte Interface, sondern die durch den Benutzer angegebene JPA-konforme Implementierung durch Eclipselink verwendet wird.

OTMetadataField Die MetadataField-Klasse wird so adaptiert, dass alle nicht-persistenten Object Teams-internen Felder durch Eclipselink als transient angesehen werden (A6, A9 Seiten 55, 56). Dazu wird die *isValidPersistenceField*-Methode von OTMetadataField ersetzt. Repräsentiert MetadataField ein Object Teams-internes Feld, so darf isValidPersistenceField nur dann *true* liefern, wenn es sich um das basis-Feld oder team-Feld einer Rolle handelt (A7, A8 Seite 56, 4.7.1). Gemäß A19 (vgl. Seite 59) darf ein persistentes Feld nicht auf ein Rollen-Interface zeigen. Dazu ersetzt OTMetadataField die *getRawClass* Methode so, dass anstelle eines Rollen-Interfaces die passende Rollen-Implementierung zurückgeliefert wird.

OTMappingAccessor Wie MetadataField liefert jeder MappingAccessor ebenfalls den Typ des Feldes, welches die Relation repräsentiert. Da es sich dabei um ein Rollen-Interface handeln kann, wird die Methode *getReferenceClass* analog zu der Methode getRawClass von OTMetadataField adaptiert.

4.10 Lazy-Loading Unterstützung

Im Gegensatz zu allen anderen Anforderungen definiert A15 (vgl. Seite 58) eine Änderung der Semantik eines Object Teams Typs. Wird eine lifting-Operation für eine Basis aufgerufen und führt der Aufruf dieses Liftings zu einem Aufruf einer *create*-Methode für einen Rollentyp, so muss genau dann, wenn es sich bei diesem Typ um eine Entity handelt überprüft werden, ob

bereits eine persistente Instanz der entsprechenden Rolle für die gegebene Basis-Instanz existiert. Dazu löst die create-Methode eine geeignete Abfrage gegen den Persistence Context aus, welche die Rolleninstanz der Basisinstanz lädt. Wie in Kommunikationsdiagramm 4.14 dargestellt benötigt das Team dazu sowohl die Informationen der Persistence Unit – um herauszufinden ob eine Rollenklasse persistent ist – als auch einen EntityManager und somit einen Persistence Context um eine ggf. persistierte Instanz der Rolle für die gegebene Basis-Instanz zu finden.

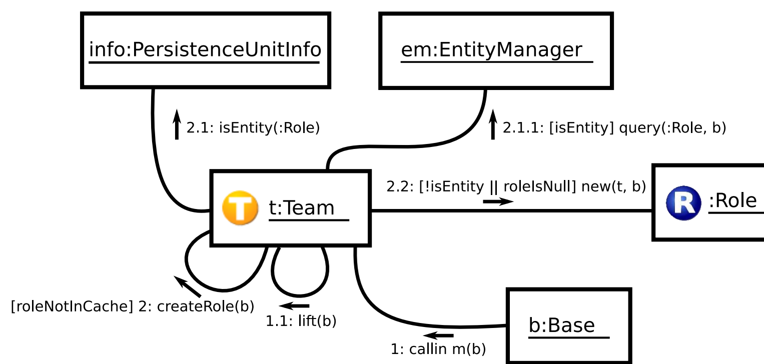

Abbildung 4.14: Kommunikationsdiagramm der Lazy-Loading Realisierung durch Delegation der Rollenerzeugung an einen Persistenzkontext

Die create-Methode für eine Rolleninstanz wird durch das Lifting aufgerufen, wenn sich die gewünschte Rolle nicht im Rollen-Cache (vgl. Seite 46, 4.3.1) befindet (*roleNotInCache*). Die create-Methode überprüft zunächst, ob es sich bei dem gewünschten Rollentyp um eine Entity handelt (*isEntity*). Ist dies der Fall, so wird der Persistence Context nach einer persistierten Instanz des Rollentyps für die gegebene Basis befragt (*query*). Ist der Rollentyp keine Entity oder liefert die Abfrage des Persistence Context kein Ergebnis, so wird eine neue Rolle erzeugt (*new*).

Da die am Lifting beteiligten Methoden Teil der automatisch generierten Infrastruktur und somit Kernbestandteil der Object Teams Applikation sind können sie nicht durch Object Teams selbst adaptiert werden. Zur Adaption des create-Verhaltens von Teams wurde daher eine Anpassung der Object Teams Infrastruktur oder eine externe aspektorientierte Lösung

benötigt[13]. Um die Persistenzunterstützung nicht mit den Concerns des Object Teams Kerns zu vermischen, wurde letztere Lösung favorisiert. Das beschriebene Verhalten sollte daher durch einen eigenen, mit einer geeigneten Webe-Technik in die Teaminfrastruktur einzubringen LazyLoading-Aspekt realisiert werden (Abbildung 4.15).

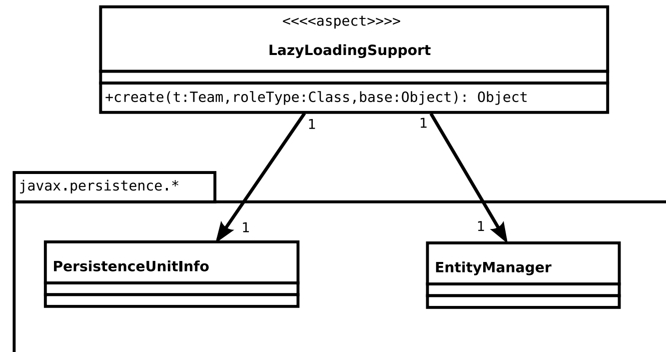

Abbildung 4.15: UML-Klassendiagramm des LazyLoadingSupport-Aspektes

4.11 Zusammenfassung

Die Analyse der Anforderungen des Object Teams Programmiermodells an eine JPA-basierte Persistenzlösung und die Ableitung geeigneter Adaptionspunkte in Eclipselink zeigten, dass die Annahmen der grundlegenden Entwicklungsstrategie zutreffend sind. Die notwendigen Änderungen an Eclipselink beschränken sich auf eine überschaubare Anzahl klar definierter semantischer und struktureller Erweiterungen, deren querschnittlicher Natur mit einer Umsetzung in Object Teams Rechnung getragen werden kann. Eine erfolgreiche Implementierung der Adaption führt somit zu einer uneingeschränkten Unterstützung von Object Teams durch die JPA-Implementierung.

[13]Die Einführung einer Eventlistener-Architektur, welche eine standardisierte Schnittstelle zur Adaption des Team-Lebenszyklus bietet, ist für zukünftige Object Teams Versionen in Planung, stand im Rahmen dieser Arbeit jedoch nicht zur Verfügung.

Kapitel 5

Implementierung der Persistenzlösung auf Basis von Eclipselink

Entsprechend dem Entwurf besteht der Kern der Implementierung im Wesentlichen aus zwei Teams, welche die semantischen und strukturellen Erweiterungen durch Adaption geeigneter Klassen von Eclipselink realisieren, sowie einer Aspektklasse zur Implementierung des Lazy-Loadings.

5.1 Entwicklungsprozess und -technologien

Neben der Umsetzung dieser primären Implementierungsziele verfolgte diese Arbeit ebenfalls das Ziel, die Umsetzung im Rahmen eines leichtgewichtigen Entwicklungsprozesses unter Verwendung aktueller Entwicklungstechniken durchzuführen. Wie in Abbildung 5.1 dargestellt wurde ein auf *testgetriebener Entwicklung* und *Continuous Integration* aufbauender *agiler Entwicklungsprozess* eingesetzt, während die den Implementierungen zugrundeliegenden Spezifikationen detailliert in einem Issue-Management System erfasst wurden. Die im Rahmen der Entwicklung verwendeten Technologien wurden falls nötig an die Bedürfnisse von Object Teams angepasst.

5.1.1 Testgetriebene Entwicklung (TDD)

Testgetriebene Entwicklung bezeichnet eine Entwicklungsstrategie, welche die Implementierung des Testfalles für eine Funktionalität vor der Imple-

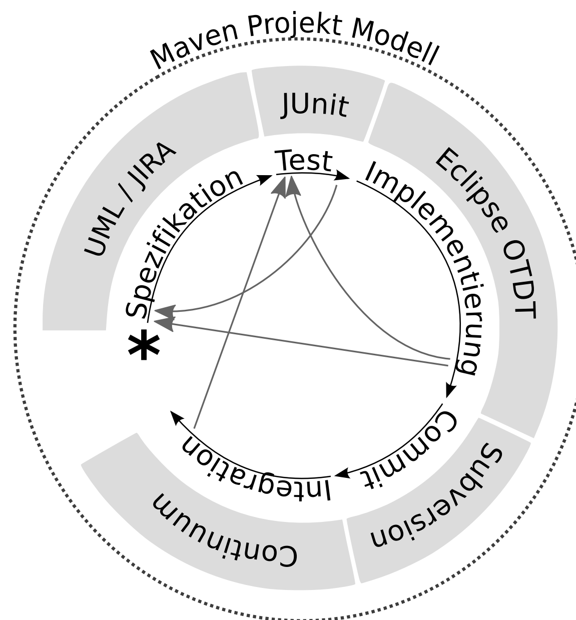

Abbildung 5.1: Entwicklungsprozess und verwendete Technologien

mentierung der Funktionalität selbst vorsieht. Der Testfall ist somit eine überprüfbare Beschreibung der Erwartungen an die Funktionalität einer Komponente und gleichzeitig ein Strukturierungswerkzeug für Entwickler, welches die Konzeption der konkreten Schnittstellen einer Komponente systematisch von ihrer Implementierung trennt. TDD bewirkt dabei eine Verschiebung der Perspektive des Entwicklers von einer auf das Innere einer Komponente fokussierten Sicht hin zu einer Client-Perspektive mit definierten Erwartungen an die Semantik der Schnittstellen einer Komponente. Dies verbessert die Qualität resultierender Schnittstellen und steigert somit die Wiederverwendbarkeit der Komponenten.

Die mit TDD entwickelten Komponenten zeichnen sich darüber hinaus durch eine reduzierte Komplexität aus, da sie der Natur ihrer Entwicklung gemäß testbar und somit für einen Entwickler verständlich bleiben müssen. Diese Tatsache wiederum bedingt eine bessere Aufteilung einer Domäne in kleinere, auf konkrete Aufgaben spezialisierte Komponenten, was die Gestaltung der gesamten Anwendung positiv beeinflusst. Des Weiteren sind die aus TDD resultierenden Testfälle Grundlage einer erheblich gesteigerten *Evolutionsfestigkeit* und Flexibilität der Komponenten, da sie sicherstellen,

dass die im Rahmen agiler Entwicklung häufig durchgeführten *Refactorings* die Semantik der Anwendung nicht beschädigen.

Bei der Erstellung von Testfällen wird zwischen *unit-Tests* und *Integrationstests* unterschieden. Überprüfen erstere ausschließlich einzelne Funktionen einer einzelnen Komponente ohne deren Abhängigkeiten (atomarer Test), so testen letztere das Zusammenspiel mehrerer Komponenten. Die in TDD eingesetzten Tests sind in erster Linie unit-Tests. Nur mit unit-Tests ist es dem Entwickler möglich, sich gedanklich auf die in einer Komponente entwickelte Funktionalität zu konzentrieren und diese zu spezifizieren, ohne die Verträge aller beteiligten Komponenten – und deren transitiver Hülle – kennen und erfüllen zu müssen oder komplexe Konfigurationslogik in die Testfälle aufzunehmen.

Gleichzeitig sind unit-Tests erheblich robuster als Integrationstests, da sie in der Regel nur durch Änderungen in einer einzigen Komponente fehlschlagen können. Die in diesem Fall oftmals eindeutige Verantwortung für den Fehlschlag einer Testausführung ist insbesondere bei der Entwicklung von Software durch ein Team ein wichtiger Vorteil. Integrationstests werden meist in einer späteren Projektphase eingesetzt um komplexere Szenarien zu überprüfen, in welchen mehrere Komponenten kollaborieren.

Voraussetzung für die testgetriebene Entwicklung ist die Verwendung einer geeigneten Bibliothek zur Formulierung und Ausführung von Testfällen sowie die Möglichkeit, die von einer Komponente geforderten Abhängigkeiten durch funktionslose Stellvertreter, sog. *Mocks* zu ersetzen um eine getestete Komponente für einen unit-Test zu isolieren. In dieser Arbeit wurden für diese Zwecke Junit [20] und Mockito [37] eingesetzt.

5.1.2 Beschreibung durch ein standardisiertes Projektmodell

Zu den zahlreichen mit der Entwicklung eines Projektes verbundenen Aufgaben gehören die Verwaltung der Projektartefakte in einer geeigneten Verzeichnisstruktur, die Definition und Verwaltung der Abhängigkeiten, die Konfiguration von Projekten in IDEs, das Übersetzen von Quellcode, die Aufteilung in Test und Implementierungscode, das Ausführen von Tests, die

Konfiguration des Classpaths für Übersetzung und Ausführung, das Generieren von Dokumentation und weiteren Projektreports, die Erstellung von Bibliotheksarchiven sowie das Deployment erzeugter Projektartefakte, Versionsmanagement und zahlreiche weitere Entwicklungsaspekte.

Mit Maven [2] steht ein Projektmodell zur Verfügung, welches die Beschreibung eines Softwareprojektes durch ein standardisiertes, XML-Basiertes POM (Projekt Objekt Modell) ermöglicht. Durch Verwendung bewährter Standards kann die notwendige projektspezifische Beschreibung dabei auf ein Minimum reduziert werden. Durch Vererbungsbeziehungen zwischen Projektmodellen können projektspezifische Eigenschaften im Sinne einer generalisierungs-spezialisierungs Relation weitergegeben werden[14]. Für Maven stehen zahlreiche Plugins zur Verfügung, welche auf dem Projektmodell aufbauend alle oben genannten Aufgaben erfüllen. In dieser Arbeit wurden die Maven-Plugins zum Kompilieren von Java-Quellcode sowie zur Ausführung von unit-Tests angepasst, so dass Maven für die Verwaltung von Object Teams Projekten verwendet werden konnte. Durch die Wiederverwendbarkeit der dabei erzeugten Konfigurationen und Plugins steht diese Möglichkeit nunmehr allen Object Teams Anwendern offen.

Integration des Object Teams-Compilers

Das maven-compiler Plugin [5] verwendet das Plexus Komponentenmodell [12] zur Trennung des Compilers von der Plugin-Implementierung. Für die Integration des Object Teams-Compilers wurde in dieser Arbeit mit dem objectteams-plexus-compiler-eclipse eine eigene Plexus-Komponente entwickelt, welche den AbstractCompiler der plexus-compiler-api erweitert und den Compile-Vorgang an den für Object Teams erweiterten Eclipse-JDT Compiler delegiert (UML-Diagramm 5.2).

Der so erzeugte Compiler wurde wie in Listing 5.1 gezeigt im Projektmodell eines Object Teams Projekts als Abhängigkeit des maven-compiler-plugins definiert und konfiguriert.

[14]Eine umfassendere Beschreibung der zahlreichen Fähigkeiten des Maven Projekt Modells kann ggf. unter [2] und [29] nachgeschlagen werden.

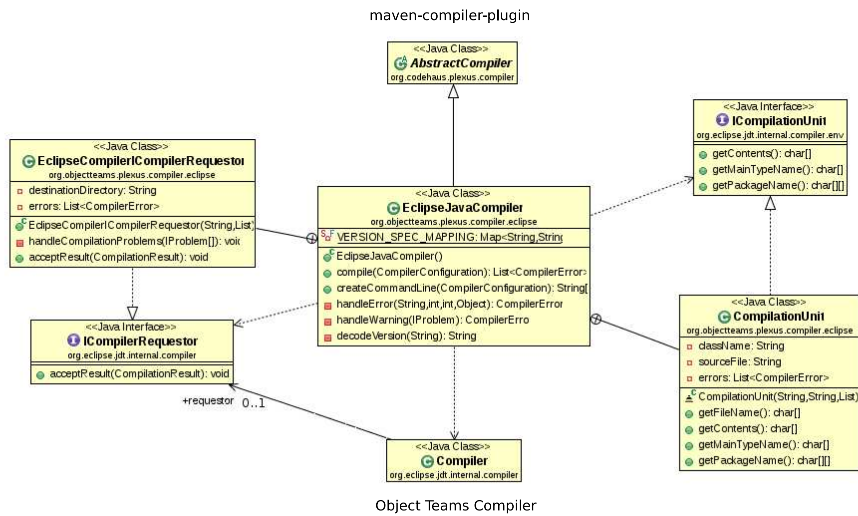

Abbildung 5.2: Integration des Object Teams Eclipse Compilers in das maven-compiler Plugin durch Erweiterung des Plexus-Compilers

```
1  /**
2   * @plexus.component role="org.codehaus.plexus.compiler.Compiler"
3   * role-hint="eclipse"
4   */
5  public class EclipseJavaCompiler extends AbstractCompiler {
6    ...
7  }
8  ...
9  <plugin>
10   <artifactId>maven-compiler-plugin</artifactId>
11   <configuration>
12     <compilerId>eclipse</compilerId>
13   </configuration>
14   <dependencies>
15     <dependency>
16       <groupId>org.objectteams</groupId>
17       <artifactId>objectteams-plexus-compiler-eclipse</artifactId>
18       <version>...</version>
19     </dependency>
20     <dependency>
21       <groupId>org.objectteams</groupId>
22       <artifactId>objectteams-jdt-compiler-core</artifactId>
23       <version>...</version>
24     </dependency>
```

```
25   </dependencies>
26 </plugin>
27 ...
```

Listing 5.1: Auszeichnung der Compiler-Klasse als Plexus Komponenten und Konfiguration des Object Teams Compilers im Maven Projekt Objekt Modell

Konfiguration der Test-Ausführung

Unit-Test Unterstützung für Maven-Projekte wird durch das surefire-Plugin [6] bereitgestellt, welches Junit-Tests konfiguriert und ausführt. Dazu verwendet Surefire die durch das maven-compiler-plugin kompilierten Klassen, welche durch den Object Teams Compiler behandelt wurden. Für die Ausführung von Object Teams Programmen müssen diese zusätzlich durch den Object Teams Load-Time Weaver behandelt werden, welcher die Rolle-Basis Infrastruktur generiert. Dieser Load-Time Weaver ist als Java-agent implementiert, welcher als Kommandozeilenargument an die JVM übergeben wird. Um von der dafür notwendigen plattformspezifischen Angabe des Pfades zu der java-agent Bibliothek zu abstrahieren, wurden die im Maven Projekt Modell verfügbaren Pfadangaben zum lokalen Code-Repository der Maven-Installation verwendet und die entsprechenden Bibliotheken als Dependencies des Projektes definiert (Listing 5.2).

```
1   <properties>
2     <otj.agent.location>
3               ${settings.localRepository}/.../objectteams-agent.jar
4     </otj.agent.location>
5     <otj.agent.arg>-javaagent:${otj.agent.location}</otj.agent.arg>
6     <surefire.argline>${otj.agent.arg}</surefire.argline>
7   </properties>
8 ...
9   <plugin>
10    <groupId>org.apache.maven.plugins</groupId>
11    <artifactId>maven-surefire-plugin</artifactId>
12    <configuration>
13      <argLine>${surefire.argline}</argLine>
14    </configuration>
15  </plugin>
16 ...
```

```
17 <dependencies>
18   ...
19   <dependency>
20     <groupId>org.objectteams</groupId>
21     <artifactId>objectteams−agent</artifactId>
22     ...
23   </dependency>
24 </dependencies>
```

Listing 5.2: Konfiguration des Maven Surefire Plugins

5.1.3 Wiederverwendbarkeit der Object Teams-spezifischen Konfiguration

Mit diesen Erweiterungen steht Object Teams das Maven Projekt Modell vollständig zur Verfügung. Für diese Arbeit wurde ein abstraktes Projekt Modell, dass *Object Teams Parent Project Object Model*, entworfen. Dieses Modell enthält die o.g. Konfiguration sowie alle wichtigen Object Teams Abhängigkeiten und weitere Plugin-Konfigurationen, unter anderem für die Erzeugung von Dokumentationen, Testberichten und Codeanalysen. Durch Beerben dieses Projektmodells lassen sich mit minimalem Aufwand neue Object Teams-Projekte anlegen.

5.1.4 Continuous Integration und Maven

Continuous Integration bezeichnet das häufige Ausführen von Projekt-Builds mit den zugehörigen Tests zur Überprüfung des Projektfortschrittes. Da jeder Entwickler eines Projektteams stets eine eigene, meist lokal modifizierte Kopie des Versionsstandes eines Projekts besitzt, übernimmt ein CI-Server die Sicht auf den aktuellen Zustand des Source-Code Repositories des Projektes und führt bei Änderung den vollständigen Build-Prozess aus. Dies stellt sicher, dass Konflikte zwischen den Änderungen unterschiedlicher Entwickler frühzeitig erkannt und behoben werden können. Gleichzeitig kann ein CI Server dazu eingesetzt werden, die Ausführung ressourcenintensiver Build-Prozesse zu übernehmen, deren Ausführung auf den Rechnersystemen einzelner Entwickler zu erheblichen Unterbrechungen im Arbeitsfluß führen würde.

Das Maven Projektmodell ermöglicht durch seine Plattformunabhängigkeit eine problemlose Übernahme von Projekt Builds durch Continuous Integration Server, ohne dass aufwändige projektspezifische Konfigurationen notwendig wären. In dieser Arbeit wurde continuum [4] erfolgreich als CI Server eingesetzt.

5.1.5 Issue Management

Alle im Rahmen dieser Arbeit durchgeführten Entwicklungsarbeiten und die diesen zugrundeliegenden Anforderungen wurden vollständig in dem auf Softwareentwicklung spezialisierten Issue-management System JIRA [8] erfasst, um den Entwicklungsprozess zu systematisieren und den Stand der resultierenden Software für spätere Entwickler nachvollziehbar zu gestalten.

5.2 Implementierung der semantischen Adaption

Die in 4.9.3 (Seite 66) beschriebene Adaption wurde durch die Team-Klasse *ObjectTeamsSemanticsAdapter* realisiert. Für die Implementierung war es dabei notwendig, beliebige von Eclipselink verwaltete Typen daraufhin zu überprüfen ob sie Teams, Rollen oder adaptierte Basisklassen repräsentieren. Da diese Informationen primär innerhalb des Object Teams Agenten verwaltet werden und somit nicht zugänglich waren, wurde ein *DefaultObjectTeamsTypeResolver* implementiert, welcher die komplexe Geschäftslogik zur Erkennung von Object-Teams Typen wiederverwendbar kapselt. Um die Möglichkeit offen zu halten, alternative Erkennungsstrategien einzuführen wurde der DefaultObjectTeamsTypeResolver durch das Interface *ObjectTeamsTypeResolver* abstrahiert. Die verwendete Resolver-Implementierung wird dem Semantics Adapter zugewiesen und von dessen Rollen verwendet.

5.2.1 OTInstantiationPolicy

Die durch die JPA geforderte unabhängige Instanziierbarkeit über einen default-Konstruktor (vgl. Seite 66, 4.9.3 und A18, Seite 59) machte ein Verfahren zur Instanziierung der nicht-statischen inneren Rollen-Klassen notwendig. Eine solche Instanziierung ist durch die Java Sprachspezifikation nicht vorgesehen. Dennoch existiert in allen bekannten JVM-Implementierung eine solche Instanziierungslogik, da die de-Serialisierung serialisierter Java-

Objekte ebenfalls Instanzen der Klassen serialisierter Objekte benötigt, welche mit den serialisierten Daten befüllt werden können.

Eine Recherche in der Open-Source Community ergab, dass die Instanziierung beliebiger Klassen unter Auslassung spezifischer Konstruktoren oder statischer Initialisierungslogik eine verbreitete Anforderung diverser Softwareprojekte ist. Insbesondere Testwerkzeuge, allen voran verbreitete Mocking-Tools wie Mockito [37] benötigen Verfahren zur zuverlässigen Erstellung von Mock-Instanzen beliebiger Java-Klassen. Das Gros dieser Werkzeuge verwendet zu diesem Zweck die Bibliothek objenesis [39], welche von den JVM-Vendor spezifischen Instanziierungsmechanismen abstrahiert und über eine geeignete *InstantiatorStrategy* die Instanziierung beliebiger Typen erlaubt. Da die Instanziierungsstrategie zuverlässig die statische Initialisierung und die Initialisierung von *final*-Feldern übergeht, lassen sich die Felder einer so erzeugten Instanz zu einem späteren Zeitpunkt initialisieren.

Die OTInstantiationPolicy verwendet zur Instanziierung von Rollenklassen einen auf der objenesis-Strategie aufbauenden *ObjectTeamsInstantiator* zur Erzeugung von Rollen-Instanzen, welche anschließend durch Eclipselink initialisiert werden. Auf diese Weise erzeugte Rollen-Instanzen weisen keine Unterschiede zu Rollen auf, welche über ihren Konstruktor erzeugt wurden.

5.2.2 OTClassDescriptor

Wie in 4.9.3 (Seite 66) definiert adaptiert der OTClassDescriptor die Kaskadierung des Löschens von Basis und Team-Entities an ihre Rollen. Dieses Verhalten machte es erforderlich, innerhalb des Semantics Adapters alle Rollen Entity-Typen zu erfassen, um untersuchen zu können, ob diese eine zu löschende Basis adaptieren oder Teil eines zu löschen Teams sind. Zu diesem Zweck wurde mit *RoleEntity* eine Rolle geschaffen, welche ClassDescriptor-Instanzen adaptiert, die einen Rollen-Typ repräsentieren. Die RoleEntity-Instanzen registrieren sich in einer Liste von Rollen-Entities, auf welche der OTClassDescriptor zugreift. Wird eine Basis oder ein Team gelöscht, so untersucht der OTClassDescriptor diese Menge von Rollen auf die entsprechende Relation zu Basis oder Team und erzeugt anschließend eine JPQL-Query [13](4) gegen den Persistence Context der zu löschenden Entity, welche die durch A10 und A16 (vgl. Seite 59, 4.7.2) definierten persistenten

Rollen löscht. Um Inkonsistenzen im Fehlerfall zu vermeiden, werden diese Rollen gelöscht, bevor die auslösende Entity (Team oder Basis) gelöscht wird. Da die JPQL-Query gegen den Persistence Context ausgeführt wird, kaskadiert dieses Verhalten, falls eine transitiv gelöschte Rolle selbst eine Basis oder ein Team ist.

Die mögliche Kapselung von Rollen in ihren Teams (vgl. Seite 43, 4.2.3) hat dabei keine Auswirkungen auf die für die Kaskadierung notwendige Implementierung. Alias-Kontrolle, bspw. in Form von Confined Roles, reduziert die mögliche Anzahl an Basis-Rolle Beziehungen nicht. Auch die mögliche Anzahl von Rollenreferenzen innerhalb eines Teams ist unbegrenzt. Die Sichtbarkeit ausserhalb der Grenzen des Teams ist irrelevant, da die Löschung von Entities explizit und unabhängig vom Referenzierungszustand der Applikation erfolgt (vgl. Seite 18, 2.3.4). Die Verwaltung aller im Hauptspeicher befindlichen, persistenten Entities im Persistence Context garantiert zudem, dass auch nach einer Löschung existierende Referenzen auf eine beliebige Entity konsistenz sind, da die entfernten Entities den entsprechenden Persistenzzustand (removed / detached, vgl. Seite 30, 3.4.1) besitzten, jedoch nicht automatisch aus dem Hauptspeicher entfernt werden. Die Kaskadierung wird ausschließlich für die Wahrung der referentiellen Integrität innerhalb des relationalen Modells benötigt.

Die in in A14 (Seite 57) definierte Registrierung einer geladenen Rolle wurde als Post-Load Verhalten von ClassDescriptor-Instanzen implementiert, welche Rollen-Entities beschreiben. Wird eine Rolle geladen, so werden Team und Basis der Rolle über die persistenten Felder this$0 und _OT$base ermittelt und die Rollen-Referenz gemäß A14 eingetragen. Auf diese Weise ist das in A14 geforderte Verhalten, unabhängig von der Art und Weise wie eine Rollen-Entity in den Persistence Context geladen wird, garantiert. Die Implementierung berücksichtigt ebenfalls die dynamische Auflösung der Wurzel von Rollenhierarchien zur Bestimmung des zu verwendenden Rollen-Caches (vgl. Seite 46, 4.3.1).

5.2.3 OTRoleBaseMapping und OTRoleTeamMapping

Zur Implementierung des Kaskadierungsverhaltens (A12, Seite 57) wurde eine gemeinsame Basisrolle für Team- und Basis-Entities, *OTRoleTeamOr-*

BaseMapping, erstellt. Diese Rolle adaptiert die *OneToOneMapping*-Klasse, welche die entsprechende 1:1 Relation zwische Rolle und Team bzw. Rolle und Basis repräsentiert. Für die Kaskadierung der Rolle-Basis Relation wurde diese Rolle erweitert und auf jene Felder beschränkt, welche eine 1:1 Relation zwischen Rolle und Basis beschreiben. Beide Rollen ersetzen jeweils die entsprechenden *isCascade...*-Methoden des OneToOneMappings, so dass diese die gewünschte Kaskadierungsart aktivieren resp. deaktivieren.

Abbildung 5.3: UML-Diagramm der Implementierung des Semantics Adapter und abhängiger Klassen

5.3 Implementierung der strukturellen Adaption

Die durch 4.9.3 (Seite 68) definierte strukturelle Adaption wurde durch das Team *ObjectTeamsStructureAdapter* umgesetzt (UML-Diagramm 5.4). Die Rollen des Teams verwenden den ebenfalls für die semantische Adaption benötigten *ObjectTeamsTypeResolver* zur Erkennung der Object Teams Typen.

5.3.1 OTFieldAdapter

Die *OTFieldAdapter*-Rolle adaptiert alle MetadataField-Instanzen, welche ein Object Teams-internes Feld bezeichnen (*isObjectTeamsInternalFieldName(String)*). Object Teams-interne Felder können eindeutig anhand ihres Namens identifiziert werden, da sie duch einen Präfix (*_OT$*) eingeleitet werden, dessen Verwendung durch Object Teams Anwender durch den Object Teams Compiler abgelehnt wird. Gemäß Entwurfsspezifikation (vgl. Seite 68, 4.9.3, OTMetadataField) sorgt der OTFieldAdapter dafür, dass alle Object Teams-internen Felder bis auf das Basis und Team-Feld persistenter Rollen als transient behandelt werden, indem isValidPersistenceField() für transiente Felder *false* zurückliefert. Die im Entwurf spezifizierte Rolle *OTMetadataField* wurde in OTFieldAdapter und in OTMetadataField aufgeteilt, da erstere auf interne Felder beschränkt ist, letztere Rolle hingegen alle MetadataField-Instanzen adaptiert.

5.3.2 OTMetadataField

Wie in 4.9.3 (Seite 68) beschrieben muss der Referenztyp eines Feldes, falls er ein Rollen-Interface bezeichnet, auf die entsprechende Rollen-Implementierung aufgelöst werden. Dazu adaptiert OTMetadataField die *getRawType()*-Methode von MetadataField. Liefert diese ein Rollen-Interface, so wird die passende Implementierung in der definierenden Team-Klasse des Rollen-Interfaces zurückgeliefert. Die dazu notwendige Geschäftslogik wurde zur einfachen Widerverwendung in der Klasse *ObjectTeamsClassUtils* umgesetzt (*ObjectTeamsClassUtils.getRoleClassWithInterface(Class<?> interface)*).

5.3.3 OTPersistenceUnitProcessor

Wie OTMetadataField löst auch der *OTPersistenceUnitProcessor* Rollen-Interfaces zu Implementierungen auf. Zur Implementierung des in 4.9.3 (Seite 68) beschriebenen Verhaltens wurde die loadClass-Methode des PersistenceUnitProcessors so adaptiert, dass Rollen-Interfaces über den übergebenen ClassLoader zu ihren Implementierungen aufgelöst werden. Damit ist die automatische Konversion der Rollenklassen in Interface und Implementierung für Object Teams-Anwender transparent (A1, Seite 54).

5.3.4 OTMappingAccessor

Analog zu OTMetadataField adaptiert die *OTMappingAccessor* -Rolle die *getReferenceClass()*-Methode ihrer Basis *MappingAccessor*, so dass diese für alle Rollen-Interfaces die durch *ObjectTeamsClassUtils.getRoleClassWithInterface* aufgelöste Rollen-Implementierung zurückliefert (vgl. Seite 68, 4.9.3, OTMappingAccessor).

5.3.5 OTRoleDatabaseField

Diese im Entwurf nicht vorgesehen Rolle dient dazu, die Lesbarkeit des durch Eclipselink in der pping-Phase (vgl. Seite 30, 3.4.1) erzeugten Schemas für Dritte zu erhöhen. Dazu wird die Schema-Repräsentation der persistierbaren, Object Teams-internen Felder der Rolle (_OT$base und this$0) in Form der entsprechenden DatabaseField-Instanzen so adaptiert, dass ihre Feldnamen *base* bzw. *team* statt _OT$base und this$0 lauten. Auf diese Weise ist die Benutzbarkeit und Lesbarkeit des Schemas deutlich verbessert.

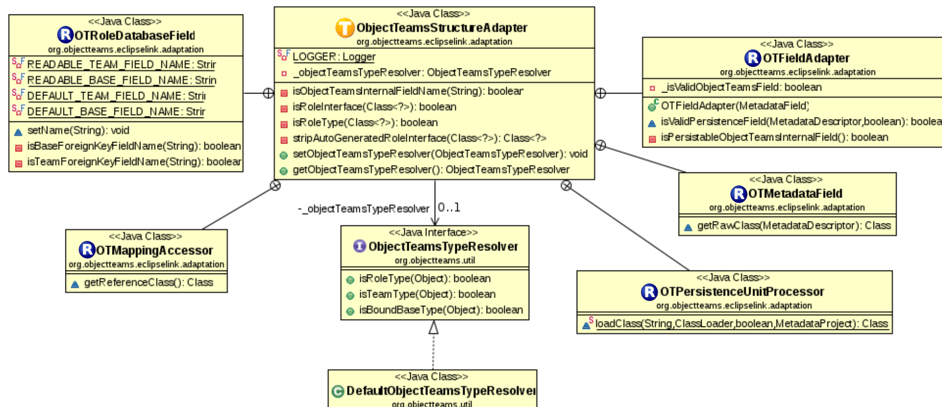

Abbildung 5.4: UML-Diagramm der Implementierung des Structure Adapters und abhängiger Klassen

5.4 Implementierung des Lazy-Loadings

Wie in 4.10 (Seite 68) definiert wurde Lazy-Loading in Form eines eigenen Aspektes in Team-Klassen integriert. Zur Implementierung und Integration des Aspektes wurde AspectJ [25] verwendet. Die resultierende Aspektklasse

DelegateRoleCreationAspect wird an alle create-Methoden jedes Teams gewebt und delegiert create-Anforderungen für Rollen an eine konfigurierbare Implementierung von *RoleCreationInterceptor*. Das in Kommunikationsdiagramm 4.14 (Seite 69) spezifizierte Lazy-Loading Verhalten wurde durch die *RoleQueryingLiftingInterceptor*-Klasse implementiert, welche für persistente Rollen ein JPQL-Query erzeugt, um eine eventuell bereits existierende persistierte Entity in den Persistence Context zu laden. Der zu ladende Rollentyp wird dabei durch den Rückgabetyp der create-Methode des Teams definiert. Ist der Rollentyp keine Entity oder existiert keine persistierte Rolle, so ruft der DelegateRoleCreationAspect die ursprüngliche create-Methode des Teams auf.

Wie in UML-Diagramm 5.5 dargestellt wird der zu verwendende Delegations-Mechanismus vollständig von der Aspektklasse getrennt. Dies dient der Isolation des Classloadings der durch den Aspekt betroffenen Teamklassen von dem Classloading der Lazy-loading Implementierungen. Auf diese Weise wird die verwendete *RoleCreationInterceptorHolder*-Implementierung erst geladen, wenn der Aspekt das erste Mal ausgelöst wird. Um zu vermeiden, dass *RoleCreationInterceptor*-Instanzen durch die statische Referenzierung zwangsweise application-scoped Singletons werden, ist der Zugriff auf diese Implementierungen durch das *RoleCreationInterceptorFactory*-Interface abstrahiert, welches durch den RoleCreationInterceptorHolder verwendet wird.

Der Aspekt wird mit Hilfe des AspectJ Load-Time Weavers in alle Teamklassen integriert, während die Konfiguration der durch den Aspekt verwendeten RoleCreationInterceptor-Implementierung über den RoleCreationInterceptorHolder zur Laufzeit vorgenommen wird.

5.5 Container Manager Persistence und Object Teams

Die beschriebene adaptierte Eclipseliunk JPA Implementierung wurde neben einfachen Testszenarien mit Bean-managed persistence ebenfalls Tests mit container-managed persistence (CMP bzw. BMP, vgl. Seite 34, 3.4.3) unterzogen. Zu diesem Zweck wurde Object Teams in das Spring Framework [38] (vgl. Seite 35, 3.4.4) integriert.

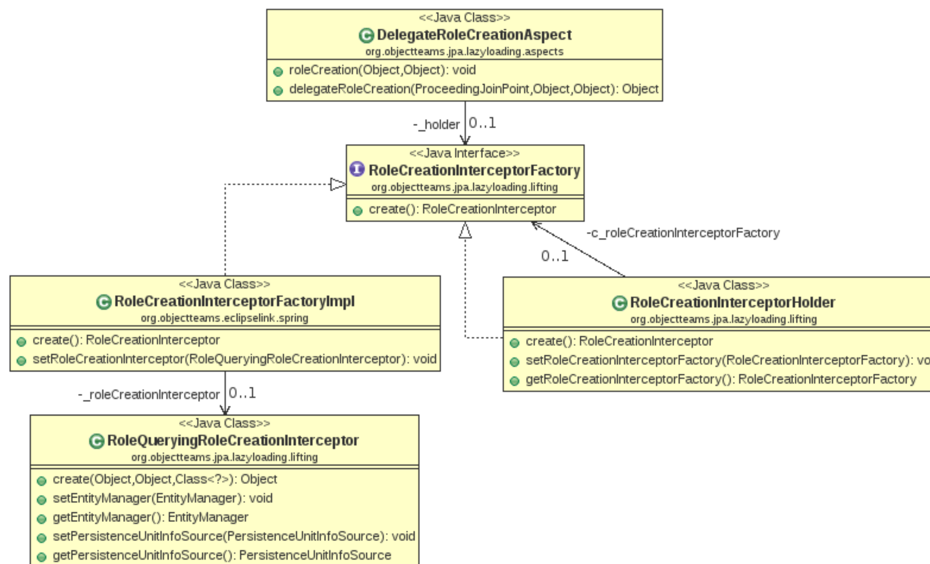

Abbildung 5.5: UML-Diagramm der Implementierung des Aspektes für Lazy-Loading Unterstützung

5.5.1 Integration von Object Teams in das Spring Framework

Die Integration von Object Teams und Spring ermöglicht nicht nur die Verwendung von CMP, sondern darüber hinaus die Nutzung der Services des Spring Frameworks – bspw. Dependency injection – für Object Teams Applikationen. Erste Integrationstests zeigten jedoch, dass der Object Teams Load-Time Weaver mit der Classloading-Strategie des Spring Frameworks inkompatibel war, da letzteres unter anderem während des Aufbaus des Spring Kontexts sowie zur Isolation der bootstrapping-Phase von JPA-Implementierungen temporäre Classloader einsetzt. Da der Load-Time Weaver jedoch den Webe-Status klassenspezifisch, jedoch nicht Classloader-spezifisch verwaltete[15], führte diese Isolation zu unvollständigen Webe-Ergebnissen.

Für die Verwendung von Object Teams mit Spring und ähnlichen Frameworks wurde der Load-Time Weaver daher so angepasst, dass der klassenspezifische Webestatus Thread-safe Classloaderspezifisch verwaltet wird. Dazu

[15]Dies ist für den Object-Teams Load-Time Weaver für die Ausführung von standalone-Applikationen der Fall, für den Einsatz in einem OSGI-Container werden andere Weaving-Mechanismen verwendet.

werden Instanzen von Klassen mit Classloader-spezifischem Zustand (*AttributeReadingGuard*, *TeamIdDispenser*, *ObjectTeamsTransformer*) durch die *ClassloaderScope*-Klasse verwaltet. Diese legt die Instanzen in einer mit einem Thread-spezifischen Classloader assoziierten Map ab oder erzeugt mittels einer generischen Factory neue, Classloader-spezifische Instanzen (UML-Diagramm 5.6).

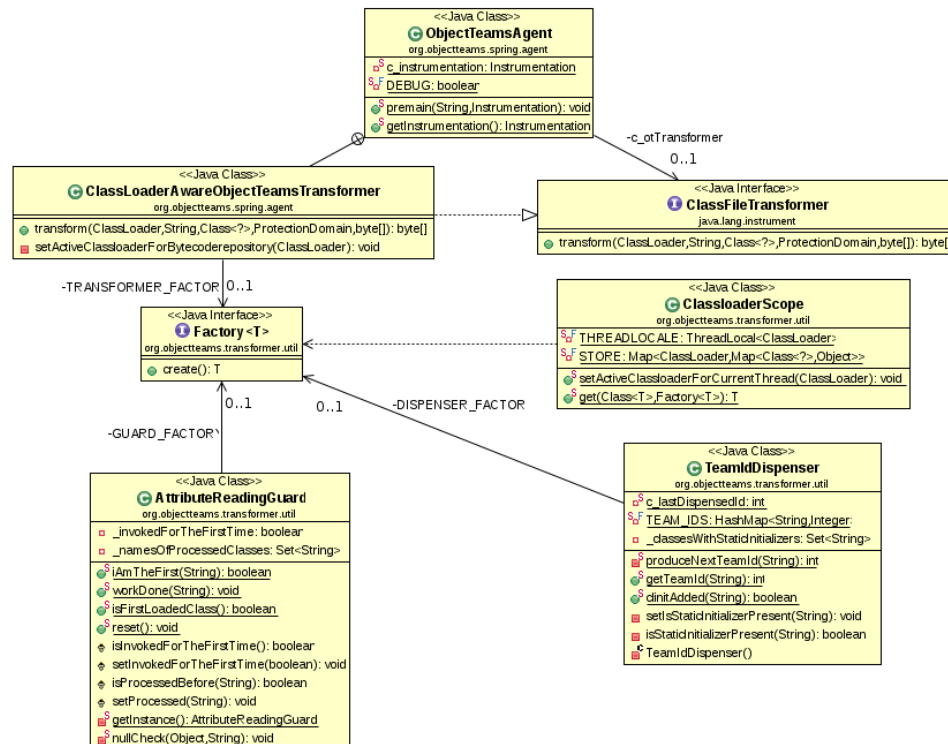

Abbildung 5.6: UML-Diagramm des multiclassloader-agent

Durch den Einsatz dieses Agenten gelang eine vollständige Integration von Object-Teams in Spring. Für die Zusammenarbeit von Eclipselink und Object Teams in Spring war es weiterhin erforderlich, Eclipselink Zugriff auf die durch Code-Generierung erzeugte Infrastruktur der Object Teams Applikation (vgl. Seite 51, 4.4) zu gewähren. Gemäß der JPA-Spezifikation muss ein Container einen temporären Classloader zur Verfügung stellen, der von den Konfigurationen anderer Classloader unabhängig ist [13](7.1.4). Ergebnisse des Classloadings in diesem Classloader sollen nach der bootstrapping-Phase verworfen werden. Dies hat unter anderem zum Ziel, das Einweben

von Attributen und Methoden in Entities zu ermöglichen, ohne dass eine JPA-Implementierung diese in die Berechnung des Persistenzverhaltens einbezieht. Im Falle von Object Teams ist genau dieses Verhalten jedoch erwünscht.

Zur Aufhebung der Isolation des bootstrapping-Classloaders wurde ein *InstrumentationPreservingLoadTimeWeaver* implementiert, der den durch den Object Teams Agenten instrumentierten Classloader der Applikation über die Methode *getThrowawayClassLoader()* an den bootstrapping-Prozess der JPA-Implementierung weiterreicht (UML-Diagramm 5.7). Somit ist die Object Teams Applikation für Eclipselink vollständig sichtbar.

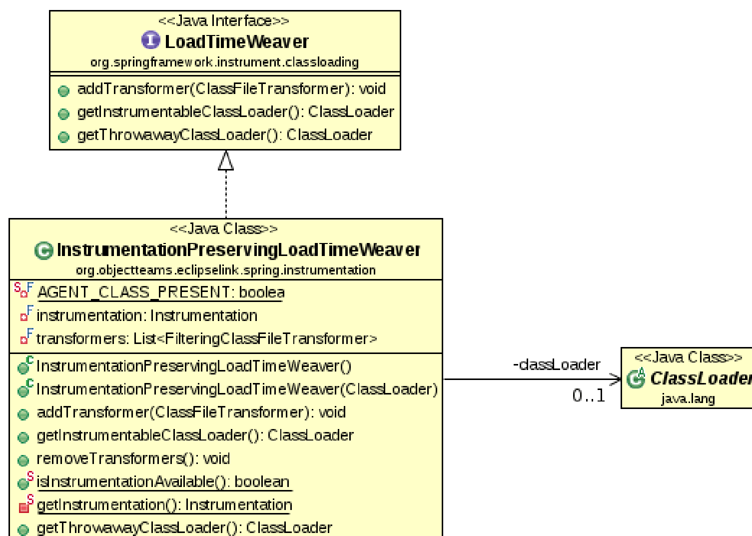

Abbildung 5.7: UML-Diagramm der InstrumentationPreservingLoadTimeWeaver-Klasse für Spring

5.5.2 Spring Kontext Konfiguration

Für die Adaption durch Eclipselink müssen die adaptierenden Teams durch den Classloader geladen und aktiviert sein, bevor die durch Rollen adaptierten Klassen geladen und instanziert werden. Wie in Abbildung 5.8 dargestellt bedeutet dies, dass die adaptierenden Teams für die semantische (vgl. Seite 78, 5.2) und strukturellen (vgl. Seite 81, 5.3) Adaptionen vor Beginn der bootstrapping-Phase (vgl. Seite 30, 3.4.1) der JPA-Implementierung ak-

tiviert und erst nach Ende des persistenz-Lebenszyklus deaktiviert werden müssen. Die Aktivierung der adaptierenden Teams muss dabei global, d.h. für alle Threads erfolgen.

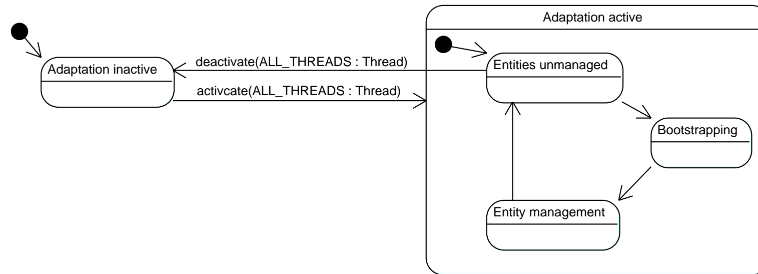

Abbildung 5.8: Abhängigkeit der Lebenszyklen von Object Teams Eclipselink Adapter und Eclipselink JPA Implementierung

Zur Integration dieses Verhaltens wurde eine Spring-spezifische Adapter-Klasse implementiert, welche die im Spring Context konfigurierten adaptierenden Teams zum Zeitpunkt des Starts des Spring-Kontexts aktiviert und beim Herunterfahren des Kontext deaktiviert. Dazu verwendet der Adapter die entsprechenden lifecycle-callbacks des Spring Frameworks (UML-Diagramm 5.9).

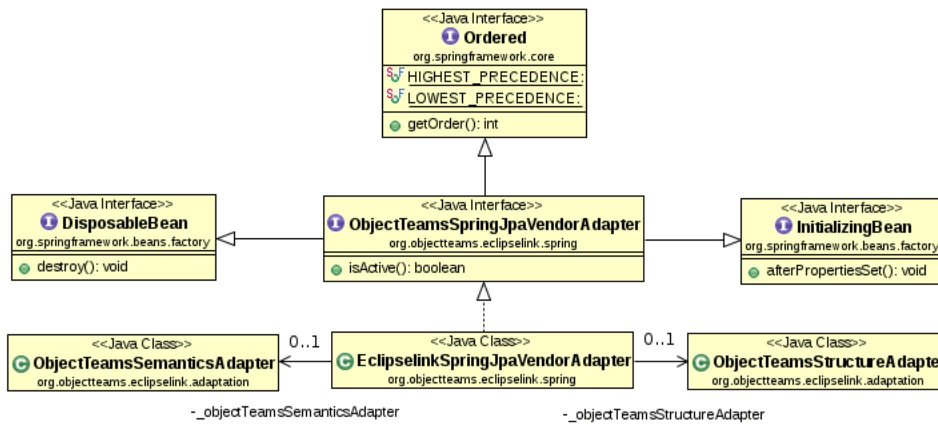

Abbildung 5.9: UML-Diagramm des Adapters zur Integration der Object Teams Eclipselink Adapter in den Spring-Kontext

Durch Verwendung dieses Adapters lassen sich die Eclipselink-Adapter und die Integration in Spring vollständig trennen, so dass keine Abhängig-

keiten der adaptierenden Teams zu Klassen des Frameworks bestehen und diese weiterhin für einfache BMP verwendet werden können.

Die vollständige Integration in das Spring-Framework besteht somit aus der in Listing 5.3 gezeigten Konfiguration des *InstrumentationPreservingLoadTimeWeaver*, des *EclipselinkSpringJpaVendorAdapter* sowie der adaptierenden Teams und ihrer Abhängigkeiten. Die gezeigte Konfiguration ließe sich durch die Verwendung von Defaults weiter vereinfachen, ist jedoch als *proof of concept* hinreichend, um die Integrationsmöglichkeit zu demonstrieren.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans ...>
3    <!--
4    This weaver is required to let the
5    JPA implementation see instrumented classes
6    -->
7    <context:load-time-weaver
8      weaver-class="[package].InstrumentationPreservingLoadTimeWeaver" />
9
10   <!--Introduces special OT semantics in eclipselink -->
11   <bean id="otSemanticsAdapter"
12     class="[package].ObjectTeamsSemanticsAdapter">
13     <property name="objectTeamsTypeResolver" ref="objectTeamsTypeResolver" />
14     <property name="instantiator" ref="instantiator" />
15   </bean>
16
17   <!--
18    Handles the ot specific structure
19    and role naming strategies
20   -->
21   <bean id="otStructureAdapter"
22     class="[package].ObjectTeamsStructureAdapter">
23     <property name="objectTeamsTypeResolver" ref="objectTeamsTypeResolver" />
24   </bean>
25
26   <!--
27   Used to determine whether an object
28   is a team, a role or a base at runtime
29   -->
30   <bean id="objectTeamsTypeResolver"
31     class="org.objectteams.util.DefaultObjectTeamsTypeResolver" />
```

```
32
33   <!-- Used to create instances of roles -->
34   <bean id="instantiator"
35     class="[package].ObjectTeamsInstantiator">
36     <property name="instantiatorStrategy" ref="instantiatorStrategy" />
37   </bean>
38
39   <!--
40   Used to create instances of
41   roles (concrete stragety)
42   -->
43   <bean id="instantiatorStrategy"
44      class="[package].ObjectTeamsInstantiationStrategy">
45      <property name="typeResolver" ref="objectTeamsTypeResolver" />
46   </bean>
47
48   <!--
49   Bean with lifecycle callbacks,
50   activates the teams configured above
51   -->
52   <bean id="otJPAVendorAdapter"
53     class="[package].EclipselinkSpringJpaVendorAdapter" />
54 </beans>
```

Listing 5.3: Konfiguration des Spring-Kontexts zur Integration von Object Teams und Eclipselink für CMP

5.6 Einschränkungen und Anforderungen der Implementierung

5.6.1 Keine Unterstützung von Property-based access in Team oder Rolle

Für die Object Teams-Typen Team und Rolle ist die Verwendung von property-based access [13](2.1.1) nicht zugelassen. Dies ist der Tatsache geschuldet, dass für die Object Teams Applikation derzeit keine getter und setter-Methoden gemäß den Java Beans-Regeln erzeugt werden. Diese Einschränkung hat jedoch keine weiteren Nachteile, sondern beschränkt lediglich den Ort, an welchem Auszeichnungen vorgenommen werden dürfen. Da die in property-based access verwendeten Methoden ohnehin keine Geschäftslogik enthalten

sollten und sowohl in ihrer Namensgebung als auch ihrer Sichtbarkeit deutlichen Einschränkungen unterliegen [13](2.1.1), bedingt property-based Acces ohnehin meist eine unnötige Häufung redundanter get und set-Methoden, welche ausschließlich Infrastruktur für die Persistenz der entsprechenden Klassen sind.

Des Weiteren hat die Verwendung von property-based access innerhalb einer Object Teams Anwendung einen spezifischen Nachteil: Bei Anwendung auf durch callin-Bindings adaptierte Methoden betrifft der Aufruf ggf. nicht die persistente Basis, sondern die adaptierende Rolle, was das Ergebnis von Persistenzoperationen ungewollt beeinflussen kann. Wenngleich dies ein steuerbares Verhalten ist, so macht es Fehler möglich, welche durch die konsequente Verwendung von field-based access vermieden werden können.

5.6.2 Keine Unterstützung von Rollenvererbung zwischen Teams

Die Rollenvererbung zwischen Teams erfolgt mittels copy inheritance (vgl. Seite 41, 4.2.1 und 46, 4.3.1). In der durch diese Arbeit verwendeten Version 1.2.8 von Object Teams wurden durch diese noch keine Annotationen vererbt. Dies ist gemäß der JPA Spezifikation jedoch erforderlich, zumal es nicht möglich ist, per copy inheritance geerbte Felder zu redefinieren, um diese zu annotieren. Die entsprechende Änderung an der copy inheritance ist für zukünftige Object Teams Versionen vorgesehen.

5.6.3 Ausführung als priviligierte Applikation

Die adaptierenden Teams verwenden – ebenso wie die eigentliche Eclipselink-Implementierung und die JPA – *Reflection* zur Analyse der Entity-Klassen. Der Zugriff auf die Eigenschaften aller Entity-Klassen und ihrer Abhängigkeiten macht es somit erforderlich, der Applikation in Umgebungen, welche strikte java security policies [21](6) einsetzen, umfangreiche Rechte einzuräumen. Dieser Umstand ist dem Konflikt zwischen der JPA Spezifikation und dem OSGI-Standard (vgl. Seite 36, 3.4.5) nicht unähnlich und ist primär dem Frameworkcharakter der JPA und ihrer Implementierungen geschuldet.

5.7 Beispiel für die Verwendung der JPA mit Object Teams

Die Anforderungen an die durch die Adaption von Eclipselink erzeugte JPA-Implementierung wurden durch eine Reihe von Integrationstests überprüft. Wie in 4.11 (Seite 70) erwartet zeigte sich, dass die Auszeichnungsmöglichkeiten der JPA vollständig durch die Implementierung unterstützt wurden und keine Kenntnis der Interna der Object Teams Applikation notwendig waren. In Listing 5.4 wird ein einfaches Beispiel der Verwendung von JPA-Annotationen in einem Object Teams Programm gezeigt, welches einem der Integrationstests entstammt. Die Art und Weise der Auszeichnung folgt dabei den in der JPA Spezifikation [13] festgelegten Standards. Die verwendeten Angaben zum Kaskadierungsverhalten (*cascade=CascadeType.PERSIST*) sind ebenso wie die *@OneToOne*-Annotation optional, da die *_role*-Referenz gemäß der JPA-Spezifikation per default persistent ist.

```
1  @Entity
2  public team class SimpleTeamEntity {
3    @Id
4    @GeneratedValue
5    private int _id;
6
7    @OneToOne(cascade = CascadeType.PERSIST)
8    private RoleEntity _role;
9
10   @Entity
11   public class RoleEntity playedBy BaseEntity {
12     @Id
13     @GeneratedValue
14     private int _id;
15
16     @SuppressWarnings("basecall")
17     callin boolean isAdapted() {
18       return true;
19     }
20
21     isAdapted <- replace isAdapted;
22   }
23
24   ...
```

```
25 }
26
27 @Entity
28 public class BaseEntity {
29     @Id
30     @GeneratedValue
31     private int _id;
32
33     public boolean isAdapted() {
34         return false;
35     }
36
37     ...
38 }
```

Listing 5.4: Beispiel für die Verwendung von JPA-Annotation in einer Object Teams Anwendung

Sind die so annotierten Typen in der persistence.xml als Teil einer Persistence Unit konfiguriert (Listing 5.5), lassen sich ihre Instanzen wie gewöhnliche Entities über einen EntityManager persistieren.

```
1  <?xml ...>
2  <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3    ...
4    version="1.0">
5  ...
6    <!-- A JPA persistence unit with for eclipselink -->
7    <persistence-unit name="my-unit"
8      transaction-type="...">
9      <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
10     <class>[package].SimpleTeamEntity</class>
11     <class>[package].SimpleTeamEntity$RoleEntity</class>
12     <class>[package].BaseEntity</class>
13   </persistence-unit>
14 ...
15 </persistence>
```

Listing 5.5: Beispiel für die Konfiguration von Object Teams Typen als Entities in einer Persistence Unit

Für das Beispiel erzeugt Eclipselink dabei das in Abbildung 5.10 gezeigte relationale Schema.

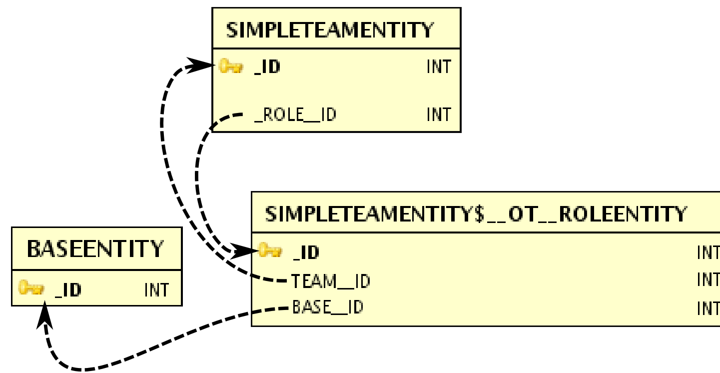

Abbildung 5.10: Von Eclipselink erzeugtes relationales Datenbankschema für Object Teams-Entities

5.8 Zusammenfassung

Die Implementierung der Eclipselink-Adaption durch Object Teams ließ sich wie geplant umsetzen und wurde im Rahmen von Intergationstests erfolgreich validiert. Darüber hinaus zeigte die Implementierung die Mächtigkeit der *decapsulation* [24](§ 2.1.2 c) von Object Teams, welche die Adaption beliebiger Teile der Eclipselink Implementierung ermöglichte. Die Integration von Object Teams in das Maven Build Modell und das Spring Framework war mit geringen Anpassungen möglich und eröffnet Anwendern zahlreiche Möglichkeiten in der Entwicklung und Gestaltung Object Teams-basierter Applikationen.

Kapitel 6

Fazit und Ausblick

Im Rahmen dieser Arbeit wurde das aspektorientierte Programmiermodell Object Teams mit der querschnittlichen Auszeichnungsebene und Semantik der Java Persistence API kombiniert. Dabei zeigte sich, dass sich auf syntaktischer und semantischer Ebene auf ihre essentielle Komplexität reduzierte Aspekte gut miteinander kombinieren lassen, da sich auftretende Konflikte primär auf die semantische Dimension der Aspekte beschränken und sich somit vorhersehen und planvoll lösen lassen.

Die für Planung und Implementierung verwendeten Programmiermodelle, Bibliotheken und Techniken erwiesen sich als qualitativ hochwerig und ausgereift. Die Eclipselink JPA Implementierung zeigte sich als strikteste am Markt befindliche Implementierung des JPA Standards und enthielt trotz ihres enormen Umfanges keine innerhalb dieses Projektes auffälligen Fehler. Im Rahmen dieser Arbeit wurde jedoch eine Schwachstelle in der JPA-Spezifikation entdeckt, welche zu einer Spezifikationsanfrage führte ([16]).

6.1 Mächtige Kombination ausgereifter Techniken

Die in dieser Arbeit entwickelte Adaption der Eclipselink JPA Implementierung bietet mit einer Reihe leichtgewichtiger, klar definierter Erweiterungen eine vollständige Unterstützung des JPA-Persistenzmodells für das Object Teams Programmiermodell. Neben den umfangreichen Möglichkeiten der Kombination dieser Modelle lassen sich nun weitere, wichtige Features der Eclipselink JPA Implementierung – unter anderem caching und clustering –

für Object Teams Nutzen. Durch die Integration von Object Teams in das Spring Framework können Anwender neben der weit verbreiteten container-managed persistence die mächtigen Services dieses populären Frameworks verwenden.

Auf entwicklungsprozesslicher Ebene steht mit der Integration in das Maven Projekt Modell ein standardisierter Mechanismus für zahlreiche Aspekte der Softwareentwicklung für Object Teams zur Verfügung. Die im Rahmen dieses Arbeit erzeugten Bibliotheken und Artefakte sind für zukünftige Anwendungsfälle als Dependencies in einem standardisierten Repository erreichbar, so dass diese über das *dependency management* ([29], 1.2.4) beliebiger maven-Projekte leicht integrierbar sind.

6.2 Ausblick

Die Möglichkeiten der Persistenzmechanismen der JPA und des Einsatzes des Spring Frameworks sind in ihrer Komplexität weitestgehend unbeschränkt. Die im Rahmen dieser Arbeit erzeugten Implementierungen sind über die Absicherung durch zahlreiche Testfälle dazu ausgelegt, in zukünftigen Projekten erweitert und kontinuierlich verbessert zu werden. Ein solcher Praxiseinsatz wäre ein entscheidender Beitrag dazu, die erreichte Entwicklung mit der stetigen Weiterentwicklung von Object Teams und den kommenden Neuerungen im Bereich standardisierter Persistenz in Einklang zu halten. Insbesondere eine Übertragung der Ergebnisse dieser Arbeit auf die derzeit in Entwicklung befindliche JPA Version 2.0 wäre in diesem Zusammenhang interessant, nicht zuletzt, da die Version 2.0 möglicherweise eine verbesserte Kompatibilität mit dem OSGI Standard bietet.

Eine konkrete Verbesserungsmöglichket an der vorliegenden Implementierung würde sich durch die Integration einer internen Schnittstelle zur Anpassung der Rollenerzeugung in den Object Teams-Kern ergeben. Der derzeit verwendete Mechanismus zur Unterstützung von Lazy-Loading verwendet als Übergangslösung eine Adaption aller Teamklassen durch AspectJ (vgl. Seite 83, 5.4), so dass neben Object Teams eine wenig wünschenswerte Abhängigkeit zu einem weiteren aspektorientierten Programmiermodell besteht. Des Weiteren könnte die Verwendung einer internen Schnittstelle

die vorliegende Implementierung im Bereich der Unterstützung von Team-interner Rollenvererbung und Lazy-Loading vereinfachen und weniger von reflection-Mechanismen abhängig machen.

Eine wichtige unbeantwortete Frage ist die der Performanz der vorliegenden Lösung im Vergleich zu herkömmlichen, auf der JPA und Eclipselink basierenden Anwendungen. Inbesondere die automatische Kaskadierung von Persistenzoperationen und die Object Teams-spezifische Lazy Loading-Implementierung bilden kritische Punkte, welche im Rahmen eines größeren Projektes analysiert werden könnten. In diesem Zusammenhang spielt ebenfalls eine Rolle, dass bisher keine Techniken zur planvollen Kombination von Aspekten unterschiedlicher Domänen durch entsprechende Aspektweber existieren. Aufgrund dieser Tatsache ist es derzeit nicht möglich, das Weaving von Object Teams mit dem für Eclipselink verwendbaren Load Time-Weaving oder anderen Weaving-Techniken konfliktfrei zu kombinieren, weshalb die vorliegende Implementierung das durch den Eclipselink Load Time-Weaver generierbare Lazy-Loading von 1:1-Relationen zwischen Entities nicht unterstützt. Die Integration einer zu erforschenden geeigneten Technik zur Kombination von Aspekten unterschiedlicher Domänen wäre ein interessanter Ansatz zur Lösung dieser Konflikte.

6.3 Anlagen

- CD-ROM mit ersteller Software und digitaler Version dieser Arbeit
- Beschreibung der CD-ROM Inhalte

Abbildungsverzeichnis

1.1 Ausprägung des Logging-Aspektes 7
1.2 Ausprägung des Persistenz-Aspektes 8

2.1 Hierarchische und relationale Datenspeicherung 12
2.2 Explizite Beziehungssemantik im relationalen Modell 13
2.3 Objektorientierte und relationale Modellierung von 1:N Relationen 16
2.4 Objektorientierte und relationale Modellierung von M:N Relationen 17
2.5 Aspekte des Object-relational impedance mismatch. 20
2.6 Schematische Darstellung eines ORM Layers 22

3.1 Zwangsweise Implementierung von Interfaces und Bereitstellung von Architekturaspekten in EJB 1 und 2 26
3.2 Chronologische Entwicklung von Persistenztechniken für Java 28
3.3 Schematische Darstellung der JPA Architektur 29
3.4 Lebenszyklus von JPA-Implementierungen 30
3.5 Lebenszyklus einer Entity [13](3.2) 32

4.1 Team, Rolle und Basis in Object Teams 41
4.2 Vererbung und implizite Vererbung in Object Teams 42
4.3 Callin bindings in Object Teams 42
4.4 Spezialisierung von playedBy-Relationen zu Elementen einer Basis-Vererbungshierarchie 43
4.5 UML-Diagramm einer Object Teams Applikation mit konkreten Team, Rolle und Basis Implementierungen 52
4.6 Ausprägung des Persistenz-Aspekts bei Verwendung der JPA 53
4.7 EventListener-Architektur des ClassDescriptors in Eclipselink 62

4.8 Classloading-Abstraktion durch den PersistenceUnitProcessor in Eclipselink . . . 62
4.9 MetadataField, ClassAccessor und ClassDescriptor in Eclipselink . . . 63
4.10 MappingAccessor, MetadataField, ClassAccessor und ClassDescriptor in Eclipselink . . . 64
4.11 InstantiationPolicy und ClassDescriptor in Eclipselink . . . 64
4.12 Kern der Entity-Metadaten Repräsentation in Eclipselink. (Quelle: Eclipselink Dokumentation [31]) . . . 65
4.13 Schematische Darstellung der Adaption von Eclipselink durch Object Teams . . . 66
4.14 Kommunikationsdiagramm der Lazy-Loading Realisierung durch Delegation der Rollenerzeugung an einen Persistenzkontext . 69
4.15 UML-Klassendiagramm des LazyLoadingSupport-Aspektes . 70

5.1 Entwicklungsprozess und verwendete Technologien . . . 72
5.2 Integration des Object Teams Eclipse Compilers in das maven-compiler Plugin durch Erweiterung des Plexus-Compilers . . 75
5.3 UML-Diagramm der Implementierung des Semantics Adapter und abhängiger Klassen . . . 81
5.4 UML-Diagramm der Implementierung des Structure Adapters und abhängiger Klassen . . . 83
5.5 UML-Diagramm der Implementierung des Aspektes für Lazy-Loading Unterstützung . . . 85
5.6 UML-Diagramm des multiclassloader-agent . . . 86
5.7 UML-Diagramm der InstrumentationPreservingLoadTimeWeaver-Klasse für Spring . . . 87
5.8 Abhängigkeit der Lebenszyklen von Object Teams Eclipselink Adapter und Eclipselink JPA Implementierung . . . 88
5.9 UML-Diagramm des Adapters zur Integration der Object Teams Eclipselink Adapter in den Spring-Kontext . . . 88
5.10 Von Eclipselink erzeugtes relationales Datenbankschema für Object Teams-Entities . . . 94

Listings

3.1 Vermischung von Geschäftslogik und Persistenzlogik durch JDBC . 24
3.2 Mit JPA Annotationen ausgezeichnete Entity mit field-based access 34
4.1 Externe Referenzierbarkeit von public roles 44
4.2 Externe Referenzierbarkeit von protected roles 44
4.3 Kapselung von confined roles 45
4.4 Durch Code-Generierung erzeugte Team Infrastruktur (Implementierung in Pseudocode) 48
4.5 Durch Code-Generierung erzeugte Rollen-Infrastruktur (Implementierung in Pseudocode) 50
5.1 Auszeichnung der Compiler-Klasse als Plexus Komponenten und Konfiguration des Object Teams Compilers im Maven Projekt Objekt Modell 75
5.2 Konfiguration des Maven Surefire Plugins 76
5.3 Konfiguration des Spring-Kontexts zur Integration von Object Teams und Eclipselink für CMP 89
5.4 Beispiel für die Verwendung von JPA-Annotation in einer Object Teams Anwendung 92
5.5 Beispiel für die Konfiguration von Object Teams Typen als Entities in einer Persistence Unit 93

Literaturverzeichnis

[1] AOSD-europe: *European Network of Excellence on Aspect-Oriented Software Development.* `http://www.aosd-europe.org`. [Online; Zugriff am 22.01.2009].

[2] Apache software foundation: *The Apache Maven Project.* `http://maven.apache.org/`. [Online; Zugriff am 28.04.2009].

[3] Apache software foundation: *The Apache openJPA project.* `http://openjpa.apache.org/`. [Online; Zugriff am 11.05.2009].

[4] Apache software foundation: *The continuum continous integration server.* `http://continuum.apache.org/`. [Online; Zugriff am 28.04.2009].

[5] Apache software foundation: *The maven compiler plugin.* `http://maven.apache.org/plugins/maven-compiler-plugin/`. [Online; Zugriff am 28.04.2009].

[6] Apache software foundation: *The maven surefire plugin.* `http://maven.apache.org/plugins/maven-surefire-plugin/`. [Online; Zugriff am 28.04.2009].

[7] Atkinson, M., F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier und S. Zdonik: *The object-oriented database system manifesto.* In: *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, Band 57, 1989.

[8] Atlassin: *The JIRA issue management System.* `http://www.atlassian.com/software/jira/`. [Online; Zugriff am 28.04.2009].

[9] Bauer, C. und G. King: *Java Persistence with Hibernate*. Manning Publications Co. Greenwich, CT, USA, 2006. ISBN-13: 978-1932394887.

[10] Chamberlin, D.D. und R.F. Boyce: *SEQUEL: A structured English query language*. In: *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, Seiten 249–264. ACM New York, NY, USA, 1974.

[11] Codd, E.F.: *A relational model of data for large shared data banks*. Communications of the ACM, 13(6):377–387, Juni 1970.

[12] Codehaus.org: *The plexus component model*. `http://plexus.codehaus.org/`. [Online; Zugriff am 28.04.2009].

[13] DeMichiel, L. und M. Keith: *JSR 220: Enterprise JavaBeans 3.0, Final Release*. Technischer Bericht, Sun Microsystems, Mai 2006.

[14] DeMichiel, L., L. Ümit Yalcinalp und S. Krishnan: *Enterprise JavaBeans (TM) Specification, Version 2.0*. Technischer Bericht, Sun Microsystems, August 2001.

[15] Dijkstra, E. W.: *EWD 447: On the role of scientific thought*. Selected Writings on Computing: A Personal Perspective, Seiten 60–66, 1982.

[16] eclipse.org: *Eclipselink Bug 275367 (Clarification of strictness)*. `https://bugs.eclipse.org/bugs/show_bug.cgi?id=275367`. [Online; Zugriff am 10.06.2009].

[17] Fowler, M.: *Inversion of control containers and the dependency injection pattern*. Actualizado el, 23, 2004.

[18] Frederick P. Brooks, Jr.: *No Silver Bullet: Essence and Accidents of Software Engineering*. IEEE Computer, 20(4):10–19, 1987.

[19] Gamilton, G. und R. Catell: *JDBC: a Java SQL API, Specification 1.2*. Technischer Bericht, JavaSoft - a Sun Microsystems, Inc. Business, 1997.

[20] Gamma, E. und K. Beck: *JUnit*. `http://www.junit.org`. [Online; Zugriff am 28.04.2009].

[21] Gong, L., G. Ellison und M. Dageforde: *Inside Java 2 platform security: architecture, API design, and implementation.* Addison-Wesley Professional, 2003. ISBN-13: 978-0201787917.

[22] Herrmann, S.: *Object Teams: Improving modularity for crosscutting collaborations.* In: *In Procs. of Net.ObjectDays*, Seiten 248–264. Springer, 2002.

[23] Herrmann, S.: *Views and Concerns and Interrelationships - Lessons Learned from Developing the Multi–View Software Engineering Environment PIROL.* Doktorarbeit, Technische Universität Berlin, 2002.

[24] Herrmann, S., C Hundt und M. Mosconi: *ObjectTeams/Java Language Definition, Version 1.2.* Technischer Bericht, Object Teams, November 2008.

[25] Kiczales, Gregor, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm und William G. Griswold: *An overview of AspectJ.* Lecture Notes in Computer Science, Seiten 327–353, 2001.

[26] Kiczales, Gregor, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier und John Irwin: *Aspect-Oriented Programming.* In: *ECOOP*, Seiten 220–242, 1997.

[27] Leavitt, N.: *Whatever happened to object-oriented databases?* IEEE Computer, 33(8):16–19, 2000.

[28] Liebhart, D.: *Architecture Blueprints: Ein Leitfaden zur Konstruktion von Softwaresystemen mit Java Spring,. net, ADF, Forms und SOA.* Hanser Fachbuchverlag, 2007. ISBN-13: 978-3446412019.

[29] Massol, V., J. Van Zyl, B. Porter, J. Casey und C. Sanchez: *Better Builds with Maven.* Exist Global, 2008.

[30] Matena, V. und M. Hapner: *Enterprise JavaBeans (TM), Version 1.0.* Technischer Bericht, Sun Microsystems, März 1998.

[31] Obrien, M: *The Eclipselink architecture documentation.* `http://wiki.eclipse.org/index.php?oldid=131958`. [Online; Zugriff am 23.03.2009].

[32] ORACLE und THE ECLIPSELINK COMMUNITY: *Eclipse persistence: Eclipselink*. http://www.eclipse.org/eclipselink/. [Online; Zugriff am 26.02.2009].

[33] OSGI ALLIANCE: *OSGi Service Platform. Core Specification, Version 4.1*. Technischer Bericht, OSGi Alliance, April 2007.

[34] RUBIO, D.: *Pro Spring Dynamic Modules for OSGi Service Platforms*. Springer-Verlag, 2009. ISBN: 978-1-4302-1612-4.

[35] RUSSELL, C.: *JSR-12: Java (TM) Data Objects (JDO) Specification*. Technischer Bericht, Sun Microsystems, 2002.

[36] STRICKLAND, J.P., P.P. UHROWCZIK und V.L. WATTS: *IMS/VS: An evolving system*. IBM Systems Journal, 21(3):490–510, 1982.

[37] SZCZEPAN, F. ET AL.: *Mockito*. http://mockito.org. [Online; Zugriff am 28.04.2009].

[38] WALLS, C. und R. BREIDENBACH: *Spring in action*. Manning Publications Co. Greenwich, CT, USA, 2007. ISBN: 1-933988-13-4.

[39] WALNES, J.: *The objenesis library*. http://code.google.com/p/objenesis/. [Online; Zugriff am 29.5.2009].