



FOM Hochschule für Oekonomie & Management

Hochschulzentrum München

Bachelor-Thesis

im Studiengang Wirtschaftsinformatik

zur Erlangung des Grades eines

Bachelor of Science (B.Sc.)

über das Thema

Verbesserung der Wartbarkeit von automatisierten Tests für ORM-Anwendungen durch Nutzung von Builder-Pattern und bidirektionalem Mapping

von

Adrian Uffmann

Erstgutachter	Dr. rer. nat. Manfred Stadel
Matrikelnummer	367454
Abgabedatum	2018-11-10

Inhaltsverzeichnis

1	Einleitung.....	1
1.1	Hinführung zum Thema.....	2
1.2	Forschungsfrage.....	3
1.3	Herangehensweise.....	3
1.4	Stand der Forschung.....	4
1.5	Aufbau der Arbeit.....	6
1.6	Aufbau eines Tests.....	6
2	Bisheriger Einsatz des Builder-Patterns.....	7
2.1	Vermeidung von zentralen Testdaten.....	7
2.2	Robustere Testdatenerzeugung durch Vermeidung von Konstruktoren.....	8
2.3	Vermeidung des manuellen Erstellens der Builder-Klassen durch Nutzung von Testdatenfabriken.....	12
2.4	Einsatz einer DSL zur Verkürzung des Testcodes.....	14
2.5	Vermeidung von identischen Instanzen mit dem Nested-Builder-Pattern.....	16
3	Bidirektionales Mapping.....	18
3.1	Implementierung von bidirektionalem Mapping ohne Veröffentlichung von Hilfsmethoden.....	22
3.2	Equals und Hashcode in Entity-Klassen.....	26
3.3	Vermeidung von Performanceproblemen mit bidirektionalem Mapping.....	30
3.3.1	Performantes Aufbauen von bidirektionalen Beziehungen.....	30
3.3.2	Performantes Laden von Entitäten mit bidirektionalen Beziehungen.....	33
4	Umfrage über die Auswirkungen von bidirektionalem Mapping.....	38
4.1	Gestaltung der Umfrage.....	38
4.2	Durchführung der Umfrage.....	40
4.3	Auswertung der Umfrage.....	40
5	Fazit.....	46
6	Ausblick.....	47
	Anhang 1: Experteninterview mit Michael Karneim.....	48
	Anhang 2: Bidirektionale 1-zu-1-Beziehung.....	64
	Anhang 3: Bidirektionale m-zu-n-Beziehung.....	65
	Anhang 4: Fragebogen der Umfrage.....	66
	Literaturverzeichnis.....	71

III

Codebeispiele

Codebeispiel 1: Verwechselbare Konstruktorparameter.....	9
Codebeispiel 2: Test mit Object-Mother.....	9
Codebeispiel 3: Object-Mother Implementierung.....	10
Codebeispiel 4: ArtikelBuilder.....	11
Codebeispiel 5: Test mit Builder.....	12
Codebeispiel 6: ArtikelBuilder Fabrikmethode.....	13
Codebeispiel 7: Builder-Interface.....	14
Codebeispiel 8: TestDsl-Klasse.....	14
Codebeispiel 9: Nutzung der Test-DSL.....	15
Codebeispiel 10: listOf-Methode.....	15
Codebeispiel 11: Nested-Builder-Pattern.....	16
Codebeispiel 12: Zustandsbehaftete Testdatenfabrik.....	17
Codebeispiel 13: Test für hatPrioritaet mit unidirektionalen n-zu-1-Beziehungen...	19
Codebeispiel 14: Test für hatPrioritaet mit unidirektionalen 1-zu-n-Beziehungen...	20
Codebeispiel 15: Test für hatBtmBezogen mit unidirektionalen 1-zu-n-Beziehungen	21
Codebeispiel 16: Test für hatBtmBezogen mit unidirektionalen n-zu-1-Beziehungen	21
Codebeispiel 17: Bidirektionale Implementierung mit öffentlicher Hilfsmethode....	23
Codebeispiel 18: Bidirektionale 1-zu-n-Beziehung mit List#contains(Object).....	25
Codebeispiel 19: Bidirektionale n-zu-1-Beziehung.....	25
Codebeispiel 20: Implementierung von Object#equals(Object).....	27
Codebeispiel 21: IdentityHashSetCustomizer.....	29
Codebeispiel 22: Bidirektionale 1-zu-n-Beziehung mit runNonRecursive.....	32
Codebeispiel 23: Methode runNonRecursive.....	33
Codebeispiel 24: Batch-Fetch-Abfrage für Aufträge.....	35
Codebeispiel 25: Batch-Fetch-Abfrage für Positionen.....	35
Codebeispiel 26: Setzen eines Load-Graphs.....	37

Abkürzungsverzeichnis

API	Abstract Programming Interface
BTM	Betäubungsmittel
CI	Continuous Integration
DSL	Domain Specific Language
ID	Identifizier (Der Primärschlüssel einer Entity)
JPA	Java Persistence API
JPQL	Java Persistence Query Language
ORM	Objekt Relationales Mapping
SQL	Structured Query Language
TDD	Test driven development / Testgetriebene Entwicklung
XML	eXtensible Markup Language

1 Einleitung

Software-Tests sind ein wesentlicher Bestandteil der Softwareentwicklung. So schreibt Wiklund: "30%-80% of the development costs are reported to be related to testing" (Wiklund, 2015, S. 25) und Myers: "it [is] a well-known rule of thumb that in a typical programming project approximately 50 percent of the elapsed time and more than 50 percent of the total cost were expended in testing the program or system being developed" (Myers, 2004, S. xii).

Automatisierte Software-Tests können dabei helfen diesen Aufwand zu verringern. Darüber hinaus bilden automatisierte Tests die Grundlage für testgetriebene Entwicklung. Entsprechend ist eine Verbesserung von automatisierten Tests ein sehr relevantes Thema.

ORM steht für Objekt-Relationales-Mapping und ist eine weit verbreitete Methode objektorientierte Daten in relationalen Datenbanken zu speichern. In der Programmiersprache Java wurde ORM in der Spezifikation "Java Persistence API" (JPA) standardisiert (vgl. Müller-Hofman u. a., 2015, S. 304). Die Grundidee von ORM ist es, eine Abbildung von sogenannten Entity-Klassen auf die Tabellen einer relationalen Datenbank zu schaffen. "Die Namen der Attribute einer Entity-Klasse repräsentieren in der Regel die Namen der Spalten einer Datenbanktabelle. Jedes konkrete Objekt steht dabei für eine konkrete Zeile der Tabelle." (Müller-Hofman u. a., 2015, S. 305).

ORM-Anwendungen arbeiten mit fachlichen Daten. Dementsprechend müssen automatisierte Tests diese Daten erzeugen, bevor die zu testenden Komponente aufgerufen werden kann. Wenn die Abhängigkeiten innerhalb des fachlichen Modells solch einer Anwendung sehr komplex und weitreichend sind, ist die Testdatenerzeugung allerdings kein triviales Problem.

Diese Arbeit soll sich daher mit einem vielversprechenden Konzept der Testdatenerzeugung für ORM-Anwendungen auseinandersetzen, dem Builder-Pattern, wie es von Nat Pryce (siehe Pryce, 2007) beschrieben wurde, und dessen Zusammenspiel mit bidirektionalem OR-Mapping.

1.1 Hinführung zum Thema

Diese Arbeit entstand in der Entwicklungsabteilung eines deutschen Pharmagroßhandelsunternehmens. Bei den hier entwickelten Anwendungen handelt es sich fast ausschließlich um Java-Datenbankanwendungen, von denen die neueren ORM verwenden. Als JPA-Implementierung werden sowohl EclipseLink, als auch Hibernate eingesetzt. Von 2004 bis 2011 wurde hier versucht automatisierte Tests zur Verbesserung der Codequalität und damit der Wartbarkeit der Software einzusetzen (vgl. Anhang 1). Die automatisierten Tests hielten in diesen Projekten jedoch nicht besonders lange und wurden nach und nach wieder ausgebaut. Hierfür gab es mehrere Gründe.

Ein Grund war die fehlende Robustheit der Tests, so musste man bei Änderungen der produktiven API oft viele Tests anpassen, da diese zum Beispiel durch neue Pflichtparameter nicht mehr kompilierbar waren.

Ein weiterer Grund war, dass die Testdaten zum Teil nicht im Test selbst, sondern in zentralen Datenbanken abgelegt waren und diese Datenkonstellationen dann aus den verschiedenen Tests referenziert wurden. Musste ein Entwickler diesen zentralen Datenbestand anpassen, zum Beispiel um einen neuen Test zu erstellen, konnte es leicht passieren, dass Tests für ein komplett unabhängiges Feature unbemerkt fehlschlügen. Da die für den nun defekten Test erforderliche Datenkonstellation nirgendwo dokumentiert war, war es im Anschluss oft schwer diesen zu reparieren.

Darüber hinaus wurden die automatisierten Tests nicht kontinuierlich, sondern meist erst kurz vor einem neuen Release ausgeführt. So wurden kaputte Tests erst spät bemerkt, wodurch der Reparaturaufwand deutlich höher war und Tests eher auskommentiert wurden.

2011 kam dann das erste Projekt, bei dem ab 2012 absehbar war, dass die automatisierten Tests hier langfristig erhalten bleiben und die gewünschten Vorteile erbringen würden. Dieses Projekt hat auch heute noch (2018) eine Testabdeckung von 57 % der Codezeilen (gemessen mit Jacoco). Dieses Projekt nutzte ein CI-System, um eine kontinuierliche Ausführung der Tests sicherzustellen und das von Nat Pryce beschriebene Builder-Pattern (siehe Pryce, 2007) sowie eine eigene

Test-DSL um die oben genannten Probleme zu überkommen.

In einem späteren Projekt wurde versucht bidirektionales Mapping einzusetzen, um die Wartbarkeit der automatisierten Tests weiter zu verbessern. Das bidirektionale Mapping erhöht jedoch auch die Komplexität des Produktivcodes. Daher stellt sich nun die Frage, ob der Einsatz von bidirektionalem Mapping die Wartbarkeit der automatisierten Tests tatsächlich verbessert und welche Auswirkungen das bidirektionale Mapping im Produktivcode hat.

1.2 Forschungsfrage

Die Forschungsfrage dieser Arbeit lautet:

Kann die Wartbarkeit von Testcode mit dem Builder-Pattern durch den Einsatz von bidirektionalem Mapping verbessert werden?

1.3 Herangehensweise

Obwohl die Forschungsfrage und Zielsetzung dieser Arbeit sehr spezifisch sind, gibt es einige verschiedene Gebiete der Informatik, die für diese Arbeit relevant sind. Als besonders wichtig wurden die Themen Testen, Object Relational Mapping, Datenbanken und Software Qualität identifiziert. Darüber hinaus wurde testgetriebene Entwicklung als relevantes Thema eingeschlossen, in der Hoffnung mehr Literatur über die Entwicklung von Testcode zu finden. Aus diesen Themen ergaben sich dann die folgenden Suchbegriffe: Automatisierte Software Tests, Bidirektionales Mapping, Builder Pattern, Datenbanktest, Object Relational Mapping, Objektorientierte Testdaten, Objektorientierte Testdatenerzeugung, Software Qualität, Software Test Qualität, Software Testautomatisierung, Testdaten Builder Pattern, Testgetriebene Software Entwicklung und Test Software Qualität. Darüber hinaus wurde auch nach den englischen Übersetzungen automated software tests, bidirectional mapping, database testing, object oriented test data, software quality, software test automation, software test quality, test data builder pattern, test driven development und test software quality gesucht. Durchsucht wurden dabei die Datenbanken EBSCOHost und Google Scholar. Es wurden für jeden Suchbegriff die ersten 10 Treffer anhand von Titel und Abstract auf Relevanz

untersucht. Einige Suchbegriffe wurden anhand der Suchergebnisse angepasst, um bessere Treffer zu erzielen. Die oben genannten Suchbegriffe sind die aus diesem Prozess resultierenden Suchbegriffe.

Durch Rückwärtssuche und anschließende Vorwärtssuche nach (nach Webster und Watson, 2002, S. xvi) ausgehend von den wichtigsten Werken wurde noch weitere Literatur gefunden. Jedoch wurden keine Werke über Testdatenerzeugung mit dem Builder-Pattern oder dem Object-Mother-Pattern gefunden, daher wurden noch Suchen über Google durchgeführt, die zu einer Vielzahl an Blog-Posts führten. Eine erneute Rückwärtssuche ausgehend von diesen Blog-Posts lieferte schließlich noch ein Werk über das Object-Mother-Pattern und das c2-Wiki (<http://wiki.c2.com>), welches das Builder-Pattern dokumentiert.

Leider ließen sich die gefundenen Metriken zur Messung von Wartbarkeit nur schlecht auf Testcode anwenden, da der Aufbau von und die Anforderungen an Testcode sich massiv von Produktivcode unterscheiden. Daher wurde als Messung der Wartbarkeit eine Umfrage unter den Entwicklern der betrachteten Firma durchgeführt. Als Vorbereitung für die Umfrage wurde ein Experteninterview mit dem Softwarearchitekten der Firma durchgeführt (siehe Anhang 1) um ein genaueres Verständnis dafür zu erlangen, welche Aspekte des Themas besonders wichtig sind und sich durch eine Umfrage unter den Mitarbeitern gut messen lassen. Bei diesem Experteninterview wurde auch klar, dass obwohl bidirektionales Mapping vor allem für die Verbesserung des Testcodes eingeführt wurde, auch die Auswirkungen auf den Produktivcode unerwartet gut waren. Auch wenn die Wartbarkeit und Performance des Produktivcodes nicht im zentralen Fokus dieser Arbeit stehen, spielen sie trotzdem eine Rolle. Selbst wenn sich die Wartbarkeit des Testcodes durch bidirektionales Mapping deutlich verbessert, bringt dieses keinen Vorteil, wenn dafür der Produktivcode langsam oder unwartbar wird. Daher wurde in der Umfrage der Fokus auch auf die Auswirkungen im Produktivcode gesetzt.

1.4 Stand der Forschung

Eine Vielzahl von Werken behandelt die automatisierte Generierung von Testdaten durch verschiedene Verfahren. Diese lassen sich einteilen in: "three types of test

data generators: pathwise test data generators [6], [8], [10], [16], [30], data specification generators [3], [19], [24], [25], and random test data generators [7]." (Korel, 1990, S. 870). Diese Verfahren haben allerdings nichts mit dem Thema dieser Arbeit zu tun. Diese Arbeit beschäftigt sich mit der programmatischen Erzeugung von Testdaten in den Testfällen selbst, damit soll testgetriebene Entwicklung unterstützt werden. Die automatischen Testdatengeneratoren hingegen scheinen eher nützlich für Regressionstests. Wissenschaftliche Werke über programmatische Testdatenerzeugung für konkrete Testfälle wurden leider nicht gefunden. Zwar werden Tests häufig kategorisiert (zum Beispiel in Modultest, Systemtest, usw.) und diese Kategorien werden allgemein beschrieben, aber auf konkrete programmiertechnische Probleme, die sich beim Warten von Tests ergeben, wird nicht eingegangen. Auch über bidirektionales Mapping konnten keine wissenschaftlichen Quellen gefunden werden. Es scheint, als würde in diesen Bereichen keine oder nur sehr wenig Forschung stattfinden, obwohl diese Themen in der Wirtschaft eine hohe Relevanz haben, da ORM-Frameworks weit verbreitet und automatisierte Tests ein wichtiges Mittel der Qualitätssicherung sind.

Die Werke, die zu Softwarequalität gefunden wurden beziehen sich auf die Qualität des Produktivcodes. Diese Arbeit untersucht jedoch die Wartbarkeit und damit die Qualität des Testcodes. Dieses Thema scheint jedoch bisher kaum wissenschaftlich untersucht zu sein. Die Metriken für die Qualität von Produktivcode lassen sich nicht einfach auf Testcode anwenden, da für automatisierte Tests andere Anforderungen gelten. So ist die Performance bei Tests deutlich weniger wichtig, während die Unabhängigkeit von Testmethoden, eine deklarative Testdefinition und die Vermeidung jeglicher Kontrollstrukturen Qualitäten sind, die für Produktivcode nicht in der Form relevant sind. Anders als bei Produktivcode kommt es bei Tests nicht darauf an, die konkrete Funktionsweise von Algorithmen oder einen fachlichen Ablauf verständlich zu präsentieren, sondern die Intention des Testschreibers und die fachlichen Anfangs- und Endbedingungen klar zu definieren. So ist bei der Erzeugung von Testdaten weniger wichtig, wie die Testdaten erzeugt wurden, sondern dass erkennbar ist, welche Testdaten erzeugt wurden.

1.5 Aufbau der Arbeit

Diese Arbeit gliedert sich in drei Hauptteile (Kapitel 2-4). In Kapitel 2 wird beschrieben, wie in der betrachteten Firma das Builder-Pattern im Testcode eingesetzt wurde, bevor bidirektionales Mapping eingeführt wurde. Kapitel 3 erklärt den Einsatz von bidirektionalem Mapping und welche Verbesserungen sich dadurch im Zusammenspiel mit dem Builder-Pattern ergeben. Dabei wird auch darauf eingegangen, wie große Performanceverluste verhindert werden können. In Kapitel 4 wird schließlich die Umfrage unter den Mitarbeitern ausgewertet.

1.6 Aufbau eines Tests

Bevor die Nutzung des Builder-Patterns in der betrachteten Firma erläutert wird, soll in diesem Kapitel auf den allgemeinen Aufbau eines automatisierten Tests eingegangen werden. Ein automatisierter Test für eine fachliche Funktion hat nach Meszaros vier Bestandteile: Setup, Exercise, Verify und Teardown (vgl. Meszaros, 2007, S. 358). Ähnliche Aufteilungen findet man auch in verschiedenen Test-Frameworks, wie z. B. Gherkin (vgl. Cucumber, 2018) und Spock (vgl. Niederwieser, 2018), hier werden dieselben Bestandteile als Given, When, Then und Cleanup bezeichnet.

Der Given-Teil bereitet alles Notwendige vor um die Voraussetzungen für die Ausführung des zu testenden Codes zu erfüllen. Der When-Teil führt die Aktion aus, die unter Test steht. Der Then-Teil überprüft, ob das erwartete Ergebnis erzielt wurde und der Cleanup-Teil räumt gegebenenfalls noch auf, z. B. indem offene Ressourcen geschlossen werden oder persistente Testdaten wieder entfernt werden. Eine explizite Markierung und Trennung dieser Bestandteile im Testcode kann die Lesbarkeit deutlich erhöhen (vgl. Meszaros, 2007, S. 359). Daher werden in der betrachteten Firma Kommentare verwendet, um Tests in diese Bestandteile aufzuteilen (siehe Codebeispiel 2). Diese Arbeit setzt sich mit dem Builder-Pattern und dessen Einsatz mit bidirektionalem Mapping auseinander, daher liegt der Fokus dieser Arbeit auf dem Given-Teil, da dort die Voraussetzungen für den Testfall erfüllt werden, was die Testdatenerzeugung mit einschließt.

2 Bisheriger Einsatz des Builder-Patterns

Der Begriff Builder-Pattern ist nicht eindeutig definiert, so gibt es beispielsweise das Builder-Pattern der Gang of Four. In dieser Arbeit geht es jedoch um das Builder-Pattern, wie es von Nat Pryce (siehe Pryce, 2007) beschrieben wurde. In diesem Teil der Arbeit soll die Funktionsweise des Builder-Patterns, sowie dessen bisheriger Einsatz in der betrachteten Firma beschrieben werden. Dabei soll auch erläutert werden, welche Probleme mithilfe des Builder-Patterns und den damit verbundenen Konzepten gelöst wurden.

2.1 Vermeidung von zentralen Testdaten

Wie sich auch in dem Interview mit Michael Karneim (siehe Anhang 1) zeigte, war eines der ursprünglichen Hauptprobleme, dass zentrale Datenkonstellationen Test übergreifend verwendet wurden. Dies ergab vor allem zwei Probleme.

Erstens konnte eine Änderung des zentralen Datenbestandes unbeabsichtigt viele Tests beeinflussen. Der Datenbestand musste jedoch häufig angepasst werden, da neue Tests oft andere Daten benötigen, als vorangegangene Tests, da diese andere Szenarien durchspielten. Außerdem konnte man sich bei einem fehlschlagenden Test nicht sicher sein, ob der Fehler durch eine Änderung der zentralen Testdaten verursacht wurde, oder ob es sich um einen Bug im Produktivcode handelte.

Zweitens war im Code des Tests nicht direkt ersichtlich, welche Eigenschaften der Testdaten relevant für diesen Test waren. Einerseits führte dies dazu, dass es schwer war bei einer vorausgegangenen Änderung der zentralen Testdaten, diese wieder zu korrigieren, andererseits machte es den Test unverständlich, weil ein Leser zunächst in die Datenbank sehen musste, um die Voraussetzungen des Szenarios zu verstehen. Auch das Schreiben von neuen Tests wurde negativ durch die zentrale Testdatenhaltung beeinflusst, da ein Entwickler zwischen Java für die Erstellung des Testfalls, und SQL, für die Erzeugung der Testdaten, hin und her wechseln musste.

Diese Probleme von Testdaten in einer zentralen Datenbank wurden auch von

anderen Autoren festgestellt: "First, because the loaded data is available for any test to use, there is no guarantee that one test will not corrupt another test's data. Second, because the data is being maintained in a system separate from the application code, keeping the test data in sync with the code is a neverending struggle" (Schuh und Punke, 2001, S. 2). Aus diesen Gründen wurde in der betrachteten Firma später auf eine zentrale Konfiguration der Testdaten verzichtet und stattdessen auf das Konzept einer fresh Fixture (siehe Meszaros, 2007, S. 311) gesetzt. Bei einer fresh Fixture werden sämtliche Testdaten, die ein Test benötigt, von diesem Test selbstständig erzeugt. Das bedeutet, wenn ein Test keine Testdaten erzeugt, ist die Datenbank, bis auf notwendige Daten, wie etwa der *User* der Anwendung, leer.

2.2 Robustere Testdatenerzeugung durch Vermeidung von Konstruktoren

Der naheliegendste Ansatz für die Erzeugung von objektorientierten Testdaten im Test ist wohl die Nutzung von Konstruktoren. Damit werden auch die Probleme der zentralen Testdatenhaltung vermieden. Konstruktoren verursachen allerdings ein eigenes Problem, welches auch in dem Interview mit Michael Karneim (siehe Anhang 1) erwähnt wurde. Konstruktoren können sich oft ändern und Tests, die Konstruktoren verwenden sind nicht sehr robust. Änderungen an den Konstruktoren können hunderte Tests kaputt machen und diese wieder zu reparieren kann sehr aufwendig sein. Darüber hinaus führen Konstruktoren bei großen Objektnetzen zu sehr komplexem und unübersichtlichem Code. Dies hat mehrere Gründe:

1. Alle Pflichtfelder müssen vom Test explizit befüllt werden, selbst wenn sie für den Test irrelevant sind. Beispielsweise müsste ein Test für die Berechnung des Verkaufspreises eines Artikels auch einen Lieferanten und eine Warengruppe für den Artikel erzeugen, obwohl diese nicht in die Berechnung mit einfließen. Einen Lieferanten zu erzeugen kann jedoch selbst wieder beliebig kompliziert sein, da ein Lieferant möglicherweise nicht ohne Abrechnungskonto oder Anschrift erzeugt werden kann. Dies führt auch zu vielfach wiederholtem Code, da jeder Test, der einen Artikel benötigt wieder dieselben Parameter erzeugen muss.

2. Bei Konstruktorparametern ist es nicht ersichtlich, ob mehrere Parameter desselben Typs vertauscht wurden. So ist in Codebeispiel 1 nicht erkennbar, ob die *Baker Street* nicht fälschlicherweise als Nachname übergeben wurde.

Codebeispiel 1: Verwechselbare Konstruktorparameter

```
new Address(  
    "Sherlock Holmes",  
    "221b Baker Street",  
    "London",  
    "NW1");
```

Quelle: In Anlehnung an Pryce, 2007

Eine Lösung für das erste dieser beiden Probleme bietet das Object-Mother-Pattern (siehe Schuh und Punke, 2001). Hierbei werden Konstruktoraufrufe vermieden, indem stattdessen Fabrikmethoden einer sogenannten Object-Mother aufgerufen werden. Diese Fabrikmethoden können dann komplexe Erzeugungslogik kapseln und irrelevante Pflichtparameter mit Standardwerten befüllen. Außerdem können sich die Fabrikmethoden gegenseitig aufrufen, wodurch eine hohe Wiederverwendbarkeit erreicht wird. Das Object-Mother-Pattern ist sehr simpel und daher leicht zu verstehen. Ein Test für das oben genannte Beispiel einer Verkaufspreisberechnung mit dem Object-Mother-Pattern in Java ist im Codebeispiel 2 zu sehen.

Codebeispiel 2: Test mit Object-Mother

```
@Test  
public void test_berechneVerkaufspreis() {  
    // given:  
    int einkaufsPreis = 2;  
    int marge = 3;  
    Artikel underTest = ObjectMother.createArtikel(einkaufsPreis, marge);  
  
    // when:  
    int actual = underTest.berechneVerkaufspreis();  
  
    // then:  
    assertThat(actual).isEqualTo(5);  
}
```

Der Test ist sehr einfach, da die Logik für die Erzeugung des Artikels in die Fabrikmethode ausgelagert wurde (siehe Codebeispiel 3).

Codebeispiel 3: Object-Mother Implementierung

```
public class ObjectMother {
    public static Artikel createArtikel(int einkaufsPreis, int gewinnMarge) {
        Artikel artikel = createArtikel();
        artikel.setEinkaufsPreis(einkaufsPreis);
        artikel.setGewinnMarge(gewinnMarge);
        return artikel;
    }

    public static Artikel createArtikel() {
        Warengruppe warengruppe = createWarengruppe();
        Lieferant lieferant = createLieferant();
        return new Artikel(warengruppe, lieferant);
    }

    public static Warengruppe createWarengruppe() { /* ... */ }

    public static Lieferant createLieferant() {
        Anschrift anschrift = createAnschrift();
        Konto abrechnungskonto = createKonto();
        return new Lieferant(anschrift, abrechnungskonto);
    }

    public static Anschrift createAnschrift() { /* ... */ }

    public static Konto createKonto() { /* ... */ }
}
```

Auch wenn das Object-Mother-Pattern einen großen Vorteil gegenüber reinen Konstruktoraufrufen bietet, indem es Codeduplikate in den Tests reduziert, bringt es selbst neue Probleme mit sich. So schreibt Nat Pryce: "Every time programmers need some slightly different test data they add another factory method to the Object Mother. [...] Over time, the Object Mother becomes bloated, messy and hard to maintain. Either programmers add new factory methods without refactoring, in which case the Object Mother becomes full of duplicated code, or programmers refactor diligently, in which case the Object Mother becomes full of many, many fine-grained methods that each contain little more than a single new statement" (Pryce, 2007).

Außerdem hat das zweite der oben genannten Probleme von Konstruktoren

weiterhin Bestand. Auch beim Aufruf einer Fabrikmethode können Parameter desselben Typs leicht verwechselt werden.

Nat Pryce schlägt als Alternative das Builder-Pattern vor. Hierbei wird für jede Testdatenklasse eine Builder-Klasse erstellt, die über with-Methoden konfiguriert werden und Objekte des entsprechenden Typs erzeugen kann. Eine Builder-Klasse für einen Artikel ist in Codebeispiel 4 zu sehen.

Codebeispiel 4: ArtikelBuilder

```
public class ArtikelBuilder {
    private Warengruppe warengruppe = new WarengruppeBuilder().build();
    private Lieferant lieferant = new LieferantBuilder().build();
    private int einkaufsPreis;
    private int gewinnMarge;

    public ArtikelBuilder withWarengruppe(Warengruppe warengruppe) {
        this.warengruppe = warengruppe;
        return this;
    }

    public ArtikelBuilder withLieferant(Lieferant lieferant) {
        this.lieferant = lieferant;
        return this;
    }

    public ArtikelBuilder withEinkaufsPreis(int einkaufsPreis) {
        this.einkaufsPreis = einkaufsPreis;
        return this;
    }

    public ArtikelBuilder withGewinnMarge(int gewinnMarge) {
        this.gewinnMarge = gewinnMarge;
        return this;
    }

    public Artikel build() {
        Artikel artikel = new Artikel(warengruppe, lieferant);
        artikel.setEinkaufsPreis(einkaufsPreis);
        artikel.setGewinnMarge(gewinnMarge);
        return artikel;
    }
}
```

Wie auch bei dem Object-Mother-Pattern wird die Erzeugungslogik wiederverwendet, da sich Builder-Klassen gegenseitig kennen und aufrufen können. Ein Test muss auch weiterhin nur die Parameter angeben, die für ihn

relevant sind, alle Anderen werden mit den Standardwerten des Builders befüllt.

Der große Vorteil des Builder-Patterns gegenüber dem Object-Mother-Pattern ist, dass jeder Test seine Testdaten sehr leicht modifizieren kann, ohne dass eine Klasse des Testframeworks angepasst werden muss. Außerdem können Parameter desselben Typs nicht mehr miteinander vertauscht werden, da bei der Konfiguration des Builders immer eine with-Methode mit dem Namen des entsprechenden Parameters aufgerufen wird. Ein Test für die Verkaufspreisberechnung mit dem Builder-Pattern ist in Codebeispiel 5 zu sehen.

Codebeispiel 5: Test mit Builder

```
@Test
public void test_berechneVerkaufspreis() {
    // given:
    Artikel underTest = new ArtikelBuilder()
        .withEinkaufspreis(2)
        .withGewinnMarge(3)
        .build();

    // when:
    int actual = underTest.berechneVerkaufspreis();

    // then:
    assertThat(actual).isEqualTo(5);
}
```

Der Test ist etwas länger als mit dem Object-Mother-Pattern, aber dafür können der Einkaufspreis und die Gewinnmarge nicht mehr verwechselt werden und die ObjectMother-Klasse, welche für jede Parameterkombination eine passende Methode anbieten würde, ist nicht mehr notwendig. Statt für jede Parameterkombination enthält die Builder-Klasse nur für jeden Parameter eine Methode. Für n Parameter werden also statt bis zu 2^n nur noch n Methoden benötigt.

2.3 Vermeidung des manuellen Erstellens der Builder-Klassen durch Nutzung von Testdatenfabriken

Das Builder-Pattern hilft, die Tests lesbarer zu machen und leichter erstellen zu können, doch dies wird durch repetitiven Wartungsaufwand erkaufte. Wann immer

eine neue für Tests relevante Datenklasse entsteht, muss auch eine entsprechende Builder-Klasse angelegt werden. Immer wenn ein neues Attribut in solch einer Datenklasse hinzugefügt oder umbenannt wird, muss auch die entsprechende with-Methode hinzugefügt oder umbenannt werden. Diese Wartungstätigkeit ist trivial und daher sehr gut für Automatisierung geeignet. Das einzige Problem ist, dass die Builder-Klassen sinnvolle Standardwerte für alle Felder enthalten müssen, damit diese nicht in jedem Test definiert werden müssen. Aus diesem Grund können die Builder-Klassen, wie sie bisher beschrieben wurden, nicht ohne weiteres generiert werden. Das ist jedoch leicht zu ändern, indem die Standardwerte nicht mehr in der Builder-Klasse selbst, sondern in einer Fabrikmethode zu definieren (siehe Codebeispiel 6).

Codebeispiel 6: ArtikelBuilder Fabrikmethode

```
public static ArtikelBuilder newArtikelBuilder() {
    return new ArtikelBuilder()
        .withEinkaufspreis(0)
        .withGewinnMarge(0)
        .withLieferant(newLieferantBuilder().build())
        .withWarengruppe(newWarengruppeBuilder().build())
    ;
}
```

Dabei empfiehlt es sich, alle with-Methoden des Builders aufzurufen, selbst wenn sich dadurch die Konfiguration nicht ändert (z. B. durch Setzen von 0, **false** oder **null**), um zu zeigen, dass der Wert so gewünscht ist und die Konfiguration dieses Attributs nicht lediglich vergessen wurde. Statt den Konstruktor der Builder-Klasse, müssen die Tests nun diese Fabrikmethode aufrufen. Da die Builder-Klasse nun trivial ist, kann sie generiert werden, dafür eignet sich in Java ein sogenannter Annotation-Prozessor sehr gut. Mit einem Annotation-Prozessor muss lediglich der Konstruktor der Datenklasse annotiert werden, damit beim Kompilieren die Builder-Klasse generiert wird. Hierfür gibt es zum Beispiel das Open-Source-Projekt *Pojobuilder* (<https://github.com/mkarneim/pojobuilder>).

Die Fabrikmethode im letzten Beispiel ist ähnlich zu den Fabrikmethoden einer Object-Mother. Wie auch im Object-Mother-Pattern macht es daher Sinn, diese Methoden in einer Klasse zu sammeln. In der betrachteten Firma hat sich für diese

Klasse der Name `TestDataFactory` (im folgenden Testdatenfabrik) durchgesetzt. Um die Methoden der Testdatenfabrik einfach im Test nutzen zu können, kann ein statischer Import auf alle Methoden der Klasse genutzt werden. Alternativ kann die Testklasse die Testdatenfabrik-Klasse erweitern und alle Methoden erben.

2.4 Einsatz einer DSL zur Verkürzung des Testcodes

Durch den Einsatz von Namenskonventionen und ein paar Hilfsmethoden kann die Lesbarkeit der Tests weiter verbessert werden. Die Tests können so deklarativer und kürzer formuliert werden. Diese Hilfsmethoden ergeben dann eine Art Domain-Specific-Language (DSL) für die Tests. Dafür ist es zunächst notwendig, dass die Builder-Klassen ein gemeinsames generisches Interface, wie in Codebeispiel 7 implementieren.

Codebeispiel 7: Builder-Interface

```
public interface Builder<T> {
    T build();
}
```

Statt in den Tests die Methode `build()` direkt aufzurufen kann dieser Aufruf dann zum Beispiel in einer `a(Builder)`, `an(Builder)` oder `some(Builder)` Methode versteckt werden (siehe Codebeispiel 8).

Codebeispiel 8: TestDsl-Klasse

```
public class TestDsl {
    public static <T> T a(Builder<T> builder) {
        return builder.build();
    }

    public static <T> T an(Builder<T> builder) {
        return builder.build();
    }

    public static <T> T some(Builder<T> builder) {
        return builder.build();
    }
}
```

Wenn man zusätzlich die Fabrikmethoden der Testdatenfabrik in z. B. `$Artikel()`

umbenannt ermöglicht das eine sehr deklarative Erzeugung von Testdaten (siehe Codebeispiel 9).

Codebeispiel 9: Nutzung der Test-DSL

```
Artikel underTest = an($Artikel()
    .withEinkaufspreis(2)
    .withGewinnMarge(3)
);
```

Das Zeichen \$ wird dabei genutzt, um Methoden zu markieren, die einen Builder zurückgeben. Diese Namenskonvention hilft besonders dann, wenn als Teil der DSL mehrere Methoden angeboten werden, von denen manche Builder liefern und manche nicht. Ein Beispiel für solch eine Methode ist `listOf(int, Builder)`. Diese Methode erleichtert es, Listen beliebiger Größe mit einem Builder zu erstellen (siehe Codebeispiel 10). Die Methode `$listOf(int, Builder)` würde hingegen einen Builder von Listen liefern, um mehrere Listen herzustellen.

Codebeispiel 10: listOf-Methode

```
public static <T> List<T> listOf(int count, Builder<T> $element) {
    List<T> result = new ArrayList<>();
    for (int i = 0; i < count; i++) {
        result.add(an($element));
    }
    return result;
}
```

Wie auch bei der Testdatenfabrik können die Methoden der Klasse `TestDsl` dem Test durch einen statischen Import oder durch Vererbung zur Verfügung gestellt werden. Da in Java keine Mehrfachvererbung von Klassen möglich ist, muss für die Vererbungsvariante entweder die Testdatenfabrik-Klasse von der `TestDsl`-Klasse ableiten oder umgekehrt, damit die Testklasse alle Methoden erbt. Alternativ kann `TestDsl` als Interface deklariert werden. In diesem Fall können jedoch keine statischen Methoden verwendet werden, da in Java 8 statische Methoden aus Interfaces nicht vererbt werden. Stattdessen müssen die Hilfsmethoden dann den Modifier `default` nutzen.

2.5 Vermeidung von identischen Instanzen mit dem Nested-Builder-Pattern

Wenn alle Defaultwerte, wie bisher gezeigt als Konstanten in den Fabrikmethoden der Testdatenfabrik konfiguriert werden, ergibt sich das Problem, dass alle durch einen Builder erzeugten Instanzen identisch sind. Zum einen kann dies dazu führen, dass die Instanzen nicht durch `Object#equals(Object)` unterschieden werden können und somit immer nur eine der Instanzen in einem `java.util.Set` verwendet werden kann, zum anderen können hierdurch Unique-Constraints beim Speichern in eine Datenbank verletzt werden. Um dies zu vermeiden, benötigt der Builder einen Zustand, der sich bei jedem Aufruf von `build()` ändert.

Hierfür eignet sich das Nested-Builder-Pattern. Dabei wird an eine `with`-Methode kein konkreter Wert übergeben, sondern ein weiterer Builder. Wenn beispielsweise einem `ArtikelBuilder` kein `Lieferant`, sondern ein `LieferantBuilder` übergeben wird, kann der `ArtikelBuilder` die Referenz auf den `LieferantBuilder` halten und für jeden neuen Artikel auch einen neuen Lieferanten erzeugen. Dasselbe Prinzip kann auch bei primitiven Datentypen verwendet werden. Statt einer Konstante, wie zuvor gezeigt, kann ein `IntBuilder` in der Fabrikmethode verwendet werden. Dieser `IntBuilder` kann dann in seiner `build()` Methode einen Zähler inkrementieren oder seinen Wert aus einem deterministischen Zufallszahlengenerator beziehen.

Codebeispiel 11: Nested-Builder-Pattern

```
public static ArtikelBuilder $Artikel() {  
    return new ArtikelBuilder()  
        .withEinkaufsPreis($int())  
        .withGewinnMarge($int())  
        .withLieferant($Lieferant())  
        .withWarengruppe($Warengruppe())  
    ;  
}
```

Wenn nun mit einem `ArtikelBuilder` mehrere Artikel erzeugt werden, haben diese verschiedene Einkaufspreise, außer ein Test konfiguriert einen konstanten Einkaufspreis. Werden allerdings zwei `ArtikelBuilder` erzeugt, indem die `$Artikel()` zweimal aufgerufen wird, werden weiterhin zwei identische Artikel erzeugt. Um dies zu vermeiden, muss die Testdatenfabrik einen Zustand halten,

nicht nur jeder Builder für sich. Dabei gilt es jedoch zu beachten, dass dieser Zustand vor jedem Test zurückgesetzt werden muss, da sonst die Ausführungsreihenfolge das Testergebnis beeinflussen könnte, was zu undeterministischen Tests, sogenannten erratic Tests (vgl. Meszaros, 2007, S. 228) führen kann.

In *JUnit 4* ist eine einfache Variante, den Zustand in der Instanz der Testklasse zu halten, da diese für jeden Test neu erzeugt wird. Diese Strategie passt auch gut dazu, die Testklasse von der Testdatenfabrik abzuleiten, um statische Imports zu vermeiden. In diesem Fall wird der Zustand direkt in der Testdatenfabrik gehalten und die Fabrikmethoden sind nicht statisch (vgl. Codebeispiel 12).

Codebeispiel 12: Zustandsbehaftete Testdatenfabrik

```
public class TestDataFactory {  
    private final IntBuilder $int = new IntBuilder();  
  
    private IntBuilder $int() {  
        return $int;  
    }  
  
    public ArtikelBuilder $Artikel() {  
        return new ArtikelBuilder()  
            .withEinkaufspreis($int())  
            .withGewinnMarge($int())  
            .withLieferant($Lieferant())  
            .withWarengruppe($Warengruppe())  
        ;  
    }  
}
```

3 Bidirektionales Mapping

Bidirektionales Mapping bezeichnet bei ORM ein Mapping, bei dem sich die beiden Seiten einer Beziehung gegenseitig kennen. "Ein Beispiel hierfür ist die Beziehung zwischen einem Bankangestellten und seinen Kunden. Ein Bankangestellter betreut viele Kunden. Die Kunden kennen dabei den für sie zuständigen Bankangestellten. Diese Beziehung ist also bidirektional." (Müller-Hofman u. a., 2015, S. 332).

Bidirektionales Mapping wurde in der betrachteten Firma eingeführt, da sich eine Verbesserung der Wartbarkeit des Testcodes erhofft wurde. Insbesondere erhoffte man sich eine Vereinfachung der Testdatenerzeugung durch die Vermeidung des Codes für den Aufbau der Beziehungen. Ohne das bidirektionale Mapping mussten Tests meist mehrere Variablen benutzen, um zunächst verschiedene Entitäten zu erzeugen und anschließend miteinander zu verbinden. Die Testdatenerzeugung war kaum deklarativ, sondern sehr imperativ implementiert. Dies lag unter anderem auch daran, dass die von der Testdatenfabrik gelieferten Builder oftmals keine Entitäten erzeugen konnten, die alle Foreign-Key-Constraints der Datenbank erfüllten. Aus diesem Grund mussten in den Tests selbst Entitäten erzeugt werden, um die entsprechenden Foreign-Key-Constraints zu erfüllen, selbst wenn diese Entitäten für den Test irrelevant waren. Langfristig führten diese Probleme zu schwer verständlichen Tests, auch obscure Tests genannt (siehe Meszaros, 2007, S. 186). Der Grund weshalb die Builder oftmals keine datenbanktauglichen Entitäten erzeugen konnten, lässt sich am besten an einem Beispiel erklären. Betrachtet werden die zwei Entity-Klassen Auftrag und Position. Ein Auftrag kann mehrere Positionen beinhalten, diese werden in einer Liste im Auftrag referenziert. Da im Produktivcode immer über den Auftrag auf die Positionen zugegriffen wird, gibt es keine Rückreferenz von einer Position zu einem Auftrag, die Beziehung ist somit unidirektional. Wird nun eine Position durch einen Builder erzeugt, gibt es bisher keine Möglichkeit, beim Erzeugen der Position einen Auftrag mitzuerzeugen und dem Test zugänglich zu machen, da weder die Position, noch der PositionBuilder, noch sonst eine beteiligte Instanz eine Referenz auf den Auftrag hat. Der Test ist somit gezwungen, explizit einen Auftrag zu erzeugen, selbst wenn

der Auftrag für den Test irrelevant ist. Umso mehr unidirektionale Beziehungen verwendet werden, umso höher ist die Wahrscheinlichkeit, dass Tests irrelevante Entitäten anlegen müssen, nur um Datenbank-Constraints zu erfüllen. Selbst wenn eine Beziehung optional ist, kann es passieren, dass ein Test diese aufbauen muss, um relevante Informationen in den erzeugten Testdaten zu hinterlegen. Beispielsweise könnte ein Test erfordern, dass die Position Teil eines Auftrags von einem Gold-Kunden ist, um sicherzustellen, dass Positionen von Gold-Kunden priorisiert werden. In diesem Fall muss neben der Position auf jeden Fall auch ein Auftrag und ein Kunde angelegt werden. Wenn die beiden Beziehungen als unidirektionale n-zu-1-Beziehungen abgebildet sind, lässt sich der Test mit dem Builder-Pattern sehr deklarativ und lesbar schreiben (siehe Codebeispiel 13).

Codebeispiel 13: Test für hatPrioritaet mit unidirektionalen n-zu-1-Beziehungen

```
@Test
public void test_hatPrioritaet() {
    // given:
    Position underTest = some($Position()
        .withAuftrag($Auftrag()
            .withKunde($Kunde()
                .withStatus(Status.GOLD))));

    // when:
    boolean actual = underTest.hatPrioritaet();

    // then:
    assertThat(actual).isTrue();
}
```

Wurden hingegen zwei unidirektionale 1-zu-n-Beziehung verwendet, müssen mehrere Variablen verwendet werden, um die Testdaten zu erzeugen. Außerdem ist in diesem Fall keine objektorientierte Implementierung möglich, da eine Position keinen Zugriff auf ihren Auftrag oder gar Kunden hat. Stattdessen ist eine serviceorientierte Implementierung der zu testenden Methode mit Zugriff auf eine Datenbank erforderlich. Neben der etwas komplexeren Testdatenerzeugung führt dies auch dazu, dass ein einfacher Unit-Test nicht mehr ausreicht, da nun eine Datenbank und möglicherweise auch ein Injectionframework im Test genutzt

werden muss, um eine korrekt initialisierte Instanz des unter Test stehenden Services zu erhalten (siehe Codebeispiel 14).

Codebeispiel 14: Test für hatPrioritaet mit unidirektionalen 1-zu-n-Beziehungen

```
@Inject
private PositionService underTest;

@Test
public void test_hatPrioritaet() {
    // given:
    Position position = some($Position());
    db().persist(some($Kunde()
        .withStatus(Status.GOLD)
        .withAuftraege($ListOf(1, $Auftrag()
            .withPositionen($ListOf(position))))));

    // when:
    boolean actual = underTest.hatPrioritaet(position);

    // then:
    assertThat(actual).isTrue();
}
```

Alternativ zu der serviceorientierten Implementierung ist noch eine statische Methode mit 3 Parametern (Kunde, Auftrag und Position) denkbar. In diesem Fall ist wieder ein einfacher Unit-Test möglich, jedoch führt diese Variante zu vielen Methoden mit sehr langen Parameterlisten, wobei die übergebenen Parameter inkonsistent miteinander sein können (z. B. wenn die übergebene Position nicht Teil des übergebenen Auftrags ist).

Eine einfache Lösung für die genannten Probleme scheint der Verzicht auf unidirektionale 1-zu-n-Beziehungen zu sein. Dies würde auch die zuvor erläuterten Probleme beim Einhalten der Foreign-Key-Constraints bei unidirektionalen 1-zu-n-Beziehungen vermeiden. Jedoch können unidirektionale n-zu-1-Beziehungen für Tests ebenso unhandlich sein. Ein Beispiel aus dem Pharmaumfeld ist die Feststellung, ob ein Kunde Betäubungsmittel (BTM) bezogen hat. Dies kann wichtig sein, da BTM besonderen gesetzlichen Auflagen unterliegen. Codebeispiel 15 zeigt, dass sich ein Test für diese Prüfung sehr einfach formulieren lässt, wenn unidirektionale 1-zu-n-Beziehungen verwendet werden, während Codebeispiel 16

zeigt, dass bei unidirektionalen n-zu-1-Beziehungen die Testdatenerzeugung wieder deutlich komplexer ausfällt.

Codebeispiel 15: Test für hatBtmBezogen mit unidirektionalen 1-zu-n-Beziehungen

```
@Test
public void test_hatBtmBezogen() {
    // given:
    Kunde underTest = some($Kunde()
        .withAuftraege($ListOf(few(), $Auftrag()
            .withPositionen($ListOf(several(), $Position()
                .withArtikel($Artikel()
                    .withBtm(false)))))));

    // when:
    boolean actual = underTest.hatBtmBezogen();

    // then:
    assertThat(actual).isFalse();
}
```

Codebeispiel 16: Test für hatBtmBezogen mit unidirektionalen n-zu-1-Beziehungen

```
@Inject
private KundeService underTest;

@Test
public void test_hatBtmBezogen() {
    // given:
    Kunde kunde = some($Kunde());
    List<Auftrag> auftraege = listOf(few(), $Auftrag().withKunde(kunde));
    db().persist(
        listOf(few() * several(), $Position()
            .withAuftrag($oneOf(auftraege))
            .withArtikel($Artikel().withBtm(false))
        )
    );

    // when:
    boolean actual = underTest.hatBtmBezogen(kunde);

    // then:
    assertThat(actual).isFalse();
}
```

Die Methoden *few()* und *several()* in Codebeispiel 15 und 16 geben kleine Zahlen

zurück und werden statt festen Werten verwendet, um zu zeigen, dass die exakten Werte für den Test nicht relevant sind. Sie sind vergleichbar mit dem Ausdruck `some($int())`. Die Methode `$oneOf(Collection)` liefert einen Builder, der bei jedem Aufruf von `build()` das nächste Element der übergebenen Liste liefert.

Wie gezeigt wurde, ist weder eine unidirektionale 1-zu-n-Beziehung noch eine unidirektionale n-zu-1-Beziehung für Tests mit dem Builder-Pattern optimal. Es kommt immer darauf an, welche Entität im Fokus des Tests steht. Gleiches gilt auch für unidirektionale 1-zu-1- und m-zu-n-Beziehungen. Bidirektionales Mapping ermöglicht es die Beziehung von beiden Seiten aufzubauen, daher können alle Tests die für sie optimale Seite verwenden, wodurch sich die Wartbarkeit der Tests erhöht (vgl. Anhang 1).

Anders als beispielsweise das Object-Mother oder Builder-Pattern beeinflusst bidirektionales Mapping unvermeidlich auch den Produktivcode, daher ist eine reine Untersuchung der Auswirkungen auf den Testcode für eine nützliche Aussage nicht ausreichend. Erstens müssen die Auswirkungen auf die Qualität des Produktivcodes betrachtet werden und zweitens darf die Performance durch das bidirektionale Mapping nicht entscheidend verschlechtert werden.

3.1 Implementierung von bidirektionalem Mapping ohne Veröffentlichung von Hilfsmethoden

Bei bidirektionalem Mapping haben beide Seiten einer Beziehung eine Referenz auf die jeweils andere Seite. Das bedeutet, die Information über die Beziehung wird redundant gehalten und wie bei jeder redundanten Datenhaltung muss besondere Aufmerksamkeit auf die Konsistenz der Daten gelegt werden. Dabei muss beachtet werden, dass beide Seiten der bidirektionalen Beziehung Methoden für die Modifikation der Beziehung anbieten müssen, um die zuvor beschriebenen Vorteile im Testcode zu erreichen.

JPA-Implementierungen kümmern sich standardmäßig nicht um eine Konsistenzhaltung der beiden Seiten von bidirektionalen Beziehungen, daher muss die Konsistenz durch den Produktivcode der ORM-Anwendung sichergestellt werden. Wird dies nicht getan, verhalten sich die beteiligten Entitäten inkonsistent,

da zwar beide Seiten einer bidirektionalen Beziehung beim Laden aus der Datenbank befüllt werden, aber Änderungen einer Seite nicht auf der anderen Seite sichtbar sind und nur eine der beiden Seiten beim Speichern in die Datenbank berücksichtigt wird (vgl. Elliott u. a., 2008, S. 81).

Eine einfache Implementierung einer bidirektionalen Beziehung kann durch die Nutzung einer öffentlichen Hilfsmethode erreicht werden (siehe Codebeispiel 17). Diese Implementierung hat jedoch den Nachteil, dass die Hilfsmethode Teil der öffentlichen Schnittstelle der Entität wird, obwohl diese Methode aus keinem anderen Kontext aufgerufen werden soll.

Codebeispiel 17: Bidirektionale Implementierung mit öffentlicher Hilfsmethode

```
public class Auftrag {
    @OneToMany(mappedBy = "auftrag")
    private List<Position> positionen = new ArrayList<>();

    public void addPosition(Position position) {
        position.setAuftragInternal(this);
        positionen.add(position);
    }

    public void removePosition(Position position) {
        position.setAuftragInternal(null);
        positionen.remove(position);
    }
}

public class Position {
    @ManyToOne
    private @Nullable Auftrag auftrag;

    public void setAuftragInternal(@Nullable Auftrag auftrag) {
        this.auftrag = auftrag;
    }

    public void setAuftrag(@Nullable Auftrag auftrag) {
        if (this.auftrag != null) this.auftrag.removePosition(this);
        if (auftrag != null) auftrag.addPosition(this);
    }
}
```

Wenn öffentliche Hilfsmethoden vermieden werden sollen, ist die Implementierung gezwungen mit Rekursion zu arbeiten, da die Methode `setAuftrag(Auftrag)` die

Methoden `addPosition(Position)` und `removePosition(Position)` aufrufen muss und umgekehrt. Dabei muss zumindest auf der 1-zu-n-Seite der Beziehung eine Abbruchbedingung geprüft werden, um zu vermeiden, dass ein Element doppelt in die Liste hinzugefügt wird. Dies würde sonst passieren, wenn die Methode `addPosition(Position)` von außen aufgerufen wird:

1. `addPosition(Position)` fügt das neue Element hinzu und ruft die `setAuftrag(Auftrag)` auf.
2. `setAuftrag(Auftrag)` prüft die Abbruchbedingung, diese schlägt fehl, da die Rückreferenz noch fehlt, daher wird die Rückreferenz gesetzt und `addPosition(Position)` aufgerufen.
3. `addPosition(Position)` fügt das neue Element erneut hinzu, da sie fälschlicherweise keine Abbruchbedingung prüft und ruft erneut `setAuftrag(Auftrag)` auf.
4. `setAuftrag(Auftrag)` prüft die Abbruchbedingung und stellt fest, dass die Beziehung fertig aufgebaut ist.

Dies bedeutet auch, dass im Falle einer m-zu-n-Beziehung auf beiden Seiten eine Abbruchbedingung geprüft werden, da beide Seiten eine `java.util.Collection` verwenden und Elemente dort nicht doppelt hinzugefügt werden sollen. Auf der Seite, die keine `Collection` verwendet, ist keine Abbruchbedingung erforderlich, da es kein Problem darstellt, wenn das Feld doppelt beschrieben wird. Bei 1-zu-1-Beziehungen muss mindestens eine Seite natürlich trotzdem die Abbruchbedingung prüfen, um eine endlose Rekursion zu vermeiden. Bei der Nutzung von `java.util.Set` in 1-zu-n- und m-zu-n-Beziehungen wird auf der zu-n-Seite nicht zwingend eine Abbruchbedingung benötigt, da in diesem Fall kein doppeltes Hinzufügen möglich ist. Bei 1-zu-n-Beziehungen ist es trotzdem besser die Abbruchbedingung auf der 1-zu-n-Seite zu prüfen, da dann immer dieselbe Implementierung für n-zu-1-Seiten verwendet werden kann, unabhängig davon, ob auf der 1-zu-n-Seite ein `Set` oder eine Liste verwendet wird. Bei m-zu-n-Beziehungen kann allerdings auf einer der beiden Seiten durch die Nutzung eines `Set` auf eine Prüfung einer Abbruchbedingung verzichtet werden (siehe Anhang 3). Eine einfache Implementierung einer bidirektionalen 1-zu-n-Beziehung mit

Verwendung einer Liste, kann als Abbruchbedingung die Methode `List#contains(Object)`.

Codebeispiel 18: Bidirektionale 1-zu-n-Beziehung mit `List#contains(Object)`

```
public class Auftrag {
    private List<Position> positionen = new ArrayList<>();

    public void addPosition(Position position) {
        if (!positionen.contains(position)) {
            positionen.add(position);
            position.setAuftrag(this);
        }
    }

    public void removePosition(Position position) {
        if (positionen.contains(position)) {
            positionen.remove(position);
            position.setAuftrag(null);
        }
    }
}
```

Codebeispiel 19 zeigt die Gegenseite zu Codebeispiel 18, die n-zu-1-Seite der bidirektionalen Beziehung. Die Implementierung in Codebeispiel 19 ist auch mit allen weiteren 1-zu-n-Seiten kompatibel und wird daher im weiteren Verlauf der Arbeit als Bestandteil aller 1-zu-n-Beziehungen angenommen. Die Reihenfolge der Befehle in Codebeispiel 19 ist zwar für die Implementierung in Codebeispiel 18 irrelevant, wird jedoch in Kapitel 3.2.1 für Codebeispiel 21 wichtig.

Codebeispiel 19: Bidirektionale n-zu-1-Beziehung

```
public class Position {
    private @Nullable Auftrag auftrag;

    public void setAuftrag(@Nullable Auftrag auftrag) {
        if (Objects.equals(this.auftrag, auftrag)) return;
        if (this.auftrag != null) this.auftrag.removePosition(this);
        if (auftrag != null) auftrag.addPosition(this);
        this.auftrag = auftrag;
    }
}
```

Wie zuvor beschrieben ist die Abbruchbedingung nur auf der 1-zu-n-Seite

notwendig, daher kann auf die erste Zeile der Methode `setAuftrag(Auftrag)` in Codebeispiel 19 verzichtet werden. In diesem Fall würde ein Setzen desselben Auftrags allerdings zwei Aufrufe von `contains(Object)` verursachen, wodurch die Dauer des Aufrufs linear von der Anzahl der Positionen abhängt. In der Praxis kommt dieser Fall vermutlich selten vor und es spielt auch nur bei einer großen Anzahl von Positionen eine Rolle. Handelt es sich allerdings um einen Auftrag, der gerade aus der Datenbank geladen wurde, kann die Liste die Positionen verzögert aus der Datenbank laden (Lazy-Loading). In diesem Fall kann die zusätzliche Abbruchbedingung einen Datenbankzugriff vermeiden. Auf Lazy-Loading wird näher in Kapitel 3.2.1 eingegangen.

3.2 Equals und Hashcode in Entity-Klassen

Ein weiterer Punkt, der zu beachten ist, ist, dass sowohl die Abbruchbedingung in `setAuftrag(Auftrag)`, als auch die `Collection`-Methoden auf `Object#equals(Object)` und nicht auf dem Referenzgleichheitsoperator `==` basieren. Das ist wichtig, da in JPA mehrere Objekte dieselbe Datenbankzeile repräsentieren können. So nutzt zum Beispiel Hibernate Proxy-Klassen, um Lazy-Loading zu ermöglichen. Für die korrekte Funktionsweise der bidirektionalen Beziehung sollte `equals(Object)` Proxys wie normale Instanzen behandeln und entsprechend auch `true` liefern, wenn eine Entity mit ihrem Proxy verglichen wird. Somit muss die Methode `equals(Object)` in Entity-Klassen überschrieben werden. Dabei gibt es prinzipiell zwei Möglichkeiten, es kann ein fachlicher oder ein synthetischer Schlüssel verwendet werden, um Gleichheit festzustellen. Ein fachlicher Schlüssel ist eine Kombination von fachlichen Daten, die die Entity eindeutig machen. Das könnte zum Beispiel die Kombination aus Land, Ort, Straße und Hausnummer für ein Gebäude sein oder eine Telefonnummer für einen Ansprechpartner. Das Problem bei der Verwendung von fachlichen Schlüsseln ist, dass nicht immer eine eindeutige Kombination von Attributen existiert. Daher werden auch in Datenbanken häufig synthetische Primärschlüssel verwendet. Bei der Verwendung von synthetischen Schlüsseln gibt es wiederum das Problem, dass diese Schlüssel (im Folgenden ID) meist erst beim Persistieren von der Datenbank zum Beispiel durch Sequences erzeugt werden. Die Beziehungen zwischen den Java-Objekten

müssen jedoch schon vor dem Speichern aufgebaut werden. Da Proxys nur für Entitäten verwendet werden, die aus der Datenbank geladen werden und damit bereits eine ID haben, ist eine Entity ohne ID nur zu sich selbst äquivalent. Daher hat sich in der betrachteten Firma die Implementierung in Codebeispiel 20 durchgesetzt. Die Implementierung von `Object#hashCode()` erzeugt dazu passend einen auf dem Rückgabewert von `getId()` basierenden Hashcode.

Codebeispiel 20: Implementierung von `Object#equals(Object)`

```
public abstract class AbstractEntity<ID extends Serializable> {
    public abstract ID getId();

    @Override
    public final boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (!(obj instanceof AbstractEntity)) return false;
        AbstractEntity<?> other = (AbstractEntity<?>) obj;
        if (unproxy().getClass() != other.unproxy().getClass()) return false;

        Object id = getId();
        if (id == null) return false;
        if (!id.equals(other.getId())) return false;
        return true;
    }

    protected final AbstractEntity<ID> unproxy() {
        return unproxyInternal().get();
    }

    protected Optional<AbstractEntity<ID>> unproxyInternal() {
        return Optional.of(this);
    }
}
```

Da die Methode `Object#getClass()` nicht überschrieben werden kann und sich die Klasse im Falle eines Proxys von der Entity-Klasse unterscheidet, muss vor einem Aufruf von `getClass()` sichergestellt werden, dass es sich nicht um einen Proxy handelt. Ansonsten wären Entitäten nicht äquivalent zu ihren Proxys. Um sicherzustellen, dass keine Proxy-Klassen verglichen werden, wird die Methode `unproxy()` verwendet. Um ihre Funktionsweise zu verstehen, muss man wissen, dass Proxy-Klassen in Hibernate die Entity-Klassen erweitern und alle Methoden überschreiben, die überschrieben werden können. Wird solch eine überschriebene

Methode auf einem Proxy aufgerufen, wird die eigentliche Entity aus der Datenbank geladen und der Aufruf wird an die eigentliche Entity weitergeleitet. Durch diesen Mechanismus erhält die Methode `unproxyInternal()` eine Referenz auf die eigentliche Entity. Diese Referenz kann jedoch nicht einfach zurückgegeben werden, da in Hibernate die Rückgabewerte von Proxy-Methoden überprüft werden. Gibt eine Methode die Referenz auf die eigentliche Entity zurück, wird stattdessen der Proxy zurückgegeben (siehe Hibernate, JavassistLazyInitializer, 2018, Zeilen 91-93). Aus diesem Grund wird die Entity in ein `Optional` eingewickelt und in der Methode `unproxy()` wieder ausgewickelt. Da die Methode `unproxy()` **final** ist, kann sie in der Proxy-Klasse nicht überschrieben und der Rückgabewert kann nicht geändert werden. Der Vorteil von dieser Vorgehensweise gegenüber der Verwendung von Methoden wie `Hibernate#getClass(Object)` oder `Hibernate#unproxy(Object)` ist, dass die Implementierung in Codebeispiel 20 unabhängig von Hibernate ist und auch mit anderen JPA-Implementierungen genutzt werden kann.

Dadurch dass die ID einer Entity erst beim Speichern festgelegt wird, ergibt sich noch ein weiteres Problem: der Hashcode, welcher von der ID abhängig ist, ändert sich, wenn eine Entity das erste Mal gespeichert wird. Dies hat zur Folge, dass Hash basierte Implementierungen, wie `java.util.HashSet` und `java.util.HashMap` für neu erzeugte Entity-Instanzen nicht korrekt funktionieren. Ändert sich der Hashcode von einem Element in einem `HashSet` oder von einem Schlüssel in einer `HashMap`, so kann dieses Element / dieser Schlüssel meist nicht mehr gefunden werden. Dies ist besonders kritisch für 1-zu-n- und m-zu-n-Beziehungen, die ein Set verwenden, da es hier kaum vermeidbar ist, dass neue Elemente persistiert werden, während sie im Set enthalten sind. Eine Lösung für dieses Problem ist die Nutzung eines `IdentityHashSet`. Ein `IdentityHashSet` nutzt statt `equals(Object)` den Referenzgleichheitsoperator `==` und statt `Object#hashCode()` die Methode `System#identityHashCode(Object)`, welche immer den originalen Hashcode liefert, unabhängig davon, ob die Methode `Object#hashCode()` in dem übergebenen Objekt überschrieben ist. In Java existiert zwar keine `IdentityHashSet` Implementierung, es gibt aber die Klasse `java.util.IdentityHashMap`.

Für die Nutzung in Hibernate kann diese mit der Methode

`Collections#newSetFromMap(Map<E, Boolean>)` in ein Set umgewandelt werden. Es ist noch wichtig zu beachten, dass es nicht ausreicht eine `IdentityHashSet` Instanz in der Entity-Klasse zu verwenden. Es muss auch über die Hibernate-Annotation `@CollectionType` sichergestellt werden, dass auch bei Lazy-Loading ein `IdentityHashSet` genutzt wird.

Für die Nutzung in EclipseLink kann die Methode `newSetFromMap(Map<E, Boolean>)` nicht verwendet werden, da EclipseLink erfordert, dass die Implementierung `Object#clone()` unterstützt (siehe Eclipse, `IndirectSet`, 2018, Zeilen 285-296). Daher muss eine eigene `IdentityHashSet`-Klasse erstellt werden, welche analog zu `java.util.HashSet` intern eine Map nutzt, in diesem Fall eine `IdentityHashMap`. Damit diese `IdentityHashSet`-Klasse auch bei Lazy-Loading verwendet wird, kann die Methode `CollectionMapping#useCollectionClass(Class<?>)` (siehe Eclipse, `useCollectionClass`, 2017) mit einer eigenen `IndirectIdentityHashSet`-Klasse aufgerufen werden. `IndirectIdentityHashSet` muss lediglich `IndirectSet` erweitern und eine Instanz von `org.eclipse.persistence.indirection.ValueHolder` mit einem `IndirectIdentityHashSet` als Wert bereitstellen. Der Aufruf von `useCollectionClass(Class<?>)` kann zum Beispiel in einem `DescriptorCustomizer` erfolgen (siehe Codebeispiel 21).

Codebeispiel 21: IdentityHashSetCustomizer

```
@Customizer(IdentityHashSetCustomizer.class)
public class Auftrag {
    @OneToMany(mappedBy = "auftrag")
    private Set<Position> positionen = new IdentityHashSet<>();
}

public class IdentityHashSetCustomizer implements DescriptorCustomizer {
    @Override
    public void customize(ClassDescriptor descriptor) {
        CollectionMapping mapping = (CollectionMapping)
descriptor.getMappingForAttributeName("positionen");
        mapping.useTransparentCollection();
        mapping.useCollectionClass(IndirectIdentityHashSet.class);
    }
}
```

3.3 Vermeidung von Performanceproblemen mit bidirektionalem Mapping

Neben der korrekten Implementierung zur Konsistenzhaltung der beiden Seiten einer bidirektionalen Beziehung, spielt auch die Performance eine große Rolle für den Einsatz im Produktivcode. Eine der größten Gefahren für die Performance besteht darin, dass an verschiedenen Stellen automatisch zusätzliche Datenbankabfragen ausgeführt werden, wenn dies nicht bewusst verhindert wird. Dies kann beispielsweise beim Aufbau einer bidirektionalen Beziehung passieren, wenn die Abbruchbedingung einer rekursiven Implementierung einen Datenbankzugriff verursacht, oder beim Laden der Entitäten, wenn die Beziehungen nicht korrekt annotiert sind. Allgemein kann man feststellen: da bei bidirektionalem Mapping mehr Code objektorientiert implementiert werden kann, anstatt explizit Datenbankabfragen auszuführen, werden indirekt mehr Abfragen automatisch vom Framework generiert. Deshalb ist es bei der Nutzung von bidirektionalem Mapping wichtiger, die richtigen Einstellungen des Frameworks zu nutzen. Zunächst soll es jedoch um die erste der beiden Gefahren gehen: wie vermeidet man unnötige Datenbankabfragen beim Auf- und Abbau von bidirektionalen Beziehungen?

3.3.1 Performantes Aufbauen von bidirektionalen Beziehungen

Betrachtet man die Implementierung in Codebeispiel 18 vor dem Hintergrund von Performance in ORM-Anwendungen, stellt man schnell fest, dass die Verwendung von `contains(Object)` als Abbruchbedingung suboptimal ist. Das liegt daran, dass es in ORM das Konzept von Lazy-Loading gibt. Bei Lazy-Loading wird das Laden von bestimmten Daten aus der Datenbank verzögert und erst dann durchgeführt, wenn diese Daten wirklich benötigt werden. Dies kann enorme Performancevorteile bringen, da in vielen Fällen nur ein kleiner Teil der Daten benötigt wird und das Laden von unbenötigten Daten so komplett vermieden werden kann. In JPA sind 1-zu-n- sowie m-zu-n-Beziehungen defaultmäßig lazy. Das bedeutet, dass beim Laden eines Auftrags aus der Datenbank die Liste der Positionen noch nicht initialisiert sein muss. Wird jedoch eine Position hinzugefügt oder entfernt, führt die Nutzung von `contains(Object)` unweigerlich zu einem Datenbankzugriff. Dieser Datenbankzugriff ist jedoch vermeidbar, da Aufrufe von `add(E)` und `remove(Object)`

keine lesenden Datenbankzugriffe erforderlich machen. Statt die `Collection` zu initialisieren kann die Aktion des Hinzufügens oder Entfernens gespeichert und entweder bei einem späteren Initialisieren der `Collection` angewendet oder, falls dies nicht passiert, zum Ende der Transaktion direkt gegen die Datenbank ausgeführt werden.

Die JPA-Referenzimplementierung EclipseLink unterstützt diese Performance-optimierung, bei Listen ist sie sogar standardmäßig aktiv (vgl. Eclipse, `setUseLazyInstantiationForIndirectCollection`, 2017). Dies hat den Seiteneffekt, dass der Rückgabewert der Methode `remove(Object)` nicht der Spezifikation in `java.util.Collection` entspricht (siehe Eclipse, `IndirectList`, 2018, Zeilen 683-693). Die Spezifikation besagt, dass `true` dann und nur dann zurückgegeben wird, wenn das zu entfernende Objekt auch in der `Collection` vorhanden war (vgl. Oracle, `Collection`, 2018). Ohne einen Datenbankzugriff ist das allerdings unmöglich festzustellen. Dies ist vermutlich der Grund, weshalb Hibernate Lazy-Loading nur bei `List#add(E)`, aber nicht bei `List#remove(Object)` oder bei `java.util.Set` unterstützt (siehe Hibernate, `PersistentList`, 2018, Zeilen 171-206). Durch Nutzung der Annotation `@CollectionType` kann aber auch in Hibernate sehr einfach eine eigene `Collection`-Implementierung verwendet werden, die verzögertes Laden auch beim Aufruf von `remove(Object)` unterstützt.

Für die performante Implementierung einer bidirektionalen 1-zu-n-Beziehung ergeben sich also zwei zusätzliche Einschränkungen. Die Implementierung darf keine Methoden aufrufen, die die `Collection` initialisieren (z. B. `contains(Object)` oder `iterator()`) und die Implementierung darf sich nicht auf den Rückgabewert von `remove(Object)` verlassen. Dies ist auch nicht notwendig, da bei der bidirektionalen Beziehung die Information, ob zwei Objekte in Beziehung stehen, redundant gehalten wird. Stattdessen kann diese Information der n-zu-1-Seite entnommen werden (siehe Codebeispiel 22), bzw. bei m-zu-n-Beziehungen muss nur eine der beiden `Collections` initialisiert werden (siehe Anhang 3).

Codebeispiel 22: Bidirektionale 1-zu-n-Beziehung mit runNonRecursive

```

public class Auftrag extends RecursionPrevention {
    @OneToMany(mappedBy = "auftrag")
    private List<Position> positionen = new ArrayList<>();

    public void addPosition(Position position) {
        if (Objects.equals(this, position.getAuftrag())) return;
        runNonRecursive(() -> {
            position.setAuftrag(this);
            positionen.add(position);
        });
    }

    public void removePosition(Position position) {
        if (!Objects.equals(this, position.getAuftrag())) return;
        runNonRecursive(() -> {
            position.setAuftrag(null);
            positionen.remove(position);
        });
    }
}

```

Die Abbruchbedingung wird demnach zu einer Prüfung, ob die Position bereits den korrekten Auftrag referenziert. Da nun die Abbruchbedingung auf die andere Seite zugreift, wird der Ablauf der Rekursion komplizierter. Die Methode `setAuftrag(Auftrag)` (siehe Codebeispiel 19) muss nun `addPosition(Position)` und `removePosition(Position)` aufrufen, bevor dem Feld `auftrag` der neue Wert zugewiesen wird, da sonst bei einem initialen Aufruf von `setAuftrag(Auftrag)` die Abbruchbedingung von `addPosition(Position)` ein Hinzufügen in die Liste von Positionen verhindern würde. Dies führt jedoch zu einer endlosen Rekursion, da `setAuftrag(Auftrag)` immer `addPosition(Position)` aufrufen muss, bevor die Abbruchbedingung ausgelöst werden darf, aber nur `setAuftrag(Auftrag)` Zugriff auf das Feld `auftrag` hat, welches die Abbruchbedingung auslösen kann. Es wird demnach noch eine andere Abbruchbedingung benötigt.

In der betrachteten Firma hat sich hierfür die Methode `runNonRecursive(Runnable)` durchgesetzt (siehe Codebeispiel 23). Diese Methode erhält ein `Runnable`, welches nur ausgeführt wird, wenn die Methode `runNonRecursive(Runnable)` nicht bereits läuft. So kann eine endlose Rekursion leicht vermieden werden.

Codebeispiel 23: Methode runNonRecursive

```

public class RecursionPrevention {
    private final transient AtomicBoolean pending = new AtomicBoolean();

    protected void runNonRecursive(Runnable runnable) {
        if (pending.compareAndSet(false, true)) {
            try {
                runnable.run();
            } finally {
                pending.set(false);
            }
        }
    }
}

```

Wichtig zu beachten ist, dass `runNonRecursive(Runnable)` nicht statisch sein kann, da `runNonRecursive(Runnable)` sowohl in `addPosition(Position)`, als auch in `removePosition(Position)` benötigt wird. In dem Fall, dass eine Position zu einem neuen Auftrag per `addPosition(Position)` hinzugefügt werden soll, muss sie von ihrem bisherigen Auftrag intern per `removePosition(Position)` entfernt werden. Wäre `runNonRecursive(Runnable)` statisch würde die Position nicht entfernt werden, da `runNonRecursive(Runnable)` bereits durch den äußeren Aufruf von `addPosition(Position)` in Verwendung ist. Damit `runNonRecursive(Runnable)` nicht in jeder Entity-Klasse erneut implementiert werden muss, empfiehlt es sich die Methode in einer gemeinsamen Superklasse zu deklarieren. Auch mit Verwendung von `runNonRecursive(Runnable)` ist die Prüfung, ob die Position bereits den richtigen Auftrag hat, notwendig, da sonst ein doppeltes Hinzufügen möglich wäre.

3.3.2 Performantes Laden von Entitäten mit bidirektionalen Beziehungen

Da Entity-Klassen bei bidirektionalem Mapping im Durchschnitt doppelt so viele Beziehungen haben, ist es umso wichtiger, Zugriffe auf diese Beziehungen performant zu machen. Außerdem bedeuten mehr Beziehungen auch, dass potenziell mehr Daten geladen werden als tatsächlich benötigt. Das liegt vor allem daran, dass in JPA der Default für 1-zu-1-Beziehungen (vgl. Jungmann & DeMichiel, 2017, S. 478) und n-zu-1-Beziehungen (vgl. Jungmann & DeMichiel, 2017, S. 463) Eager-Loading ist. Im Gegensatz zu Lazy-Loading erfordert Eager-Loading, dass die Beziehung initialisiert werden muss, wenn die Entity aus der

Datenbank geladen wird. Bei Eager-Loading können in manchen Fällen mehrere Datenbankabfragen durch Verwendung von Joins zusammengefasst werden. Wird zum Beispiel über eine Beziehung auf eine Position zugegriffen, muss diese Position geladen werden. Ist die Beziehung von Position zu Auftrag eager, kann der passende Auftrag durch einen Outer-Join in derselben Abfrage mitgeladen werden. Ist die Beziehung jedoch lazy, wird der Auftrag erst bei Zugriff geladen, wodurch zwei Abfragen notwendig sind, eine um die Position zu laden und später eine um den Auftrag zu laden. Die Performancevorteile durch die Vermeidung des Ladens überflüssiger Daten überwiegen jedoch meist die Performancevorteile, die durch Eager-Loading erreicht werden können. Außerdem können die beiden Datenbankabfragen auch zusammengefasst werden, indem explizit eine JPQL-Abfrage mit dem entsprechenden Outer-Join ausgeführt wird, bevor auf die Position zugegriffen wird. Durch die explizite Abfrage sind dann sowohl der Auftrag, als auch die Position im JPA-Cache und es wird keine Datenbankabfrage mehr beim Zugriff auf die Beziehung ausgeführt. Hierzu wird im offiziellen Hibernate-User-Guide folgendes geschrieben:

"EAGER fetching is almost always a bad choice. ... The EAGER fetching strategy cannot be overwritten on a per query basis, so the association is always going to be retrieved even if you don't need it. More, if you forget to JOIN FETCH an EAGER association in a JPQL query, Hibernate will initialize it with a secondary statement, which in turn can lead to N+1 query issues. So, EAGER fetching is to be avoided. For this reason, it's better if all associations are marked as LAZY by default." (Mihalcea u. a., 2018).

Hier wird noch eines der verheerendsten Performanceprobleme angesprochen, welches in JPA auftreten kann, das n+1 Problem. Durch diesem Problem wird beim Ausführen einer Datenbankabfrage für jede geladene Entity eine weitere Abfrage ausgeführt. Lädt eine Anwendung zum Beispiel n Positionen, werden zunächst die Positionen in einer Datenbankabfrage geladen. Wenn die Beziehung von Position zu Auftrag eager ist, muss Hibernate nun auch die Aufträge laden. Dafür wird standardmäßig eine Abfrage pro Auftrag ausgeführt. Wenn jede der n Positionen einen anderen Auftrag hat, werden also n weitere Abfragen ausgeführt, zusammen mit der initialen Abfrage sind das n+1 Abfragen. Der Grund für dieses Verhalten ist, dass Hibernate explizite Abfragen nicht um Joins für Eager-Loading erweitert, da das bei komplexen Abfragen unbeabsichtigte Seiteneffekte haben könnte. Es gibt

zwei Möglichkeiten, um dieses Problem zu vermeiden:

1. Auf Eager-Loading verzichten, wie im Hibernate-User-Guide vorgeschlagen. Dies hat den zusätzlichen Vorteil, dass weniger Daten geladen werden. Wenn die Anwendung allerdings auf die Beziehung von Position zu Auftrag zugreift, werden insgesamt weiterhin $n+1$ Abfragen ausgeführt, lediglich verzögert. Daher sollte zusätzlich die zweite Möglichkeit genutzt werden.
2. Batch-Fetching einsetzen.

Batch-Fetching ist eine Performanceoptimierung, die zwar nicht in JPA spezifiziert ist, aber sowohl von EclipseLink, als auch Hibernate unterstützt wird. Die Idee bei Batch-Fetching ist, dass mehrere Entitäten desselben Typs von der Anwendung ähnlich behandelt werden. Das bedeutet, dass wenn die Anwendung auf den Auftrag von einer Position zugreift, es sehr wahrscheinlich ist, dass die Anwendung gleich auch auf die Aufträge der anderen Positionen zugreifen möchte. Daher ist es in der Regel performanter, wenn beim Zugriff auf den Auftrag der ersten Position, implizit auch die Aufträge der andern Positionen geladen werden. Statt $n+1$ Abfragen führt dieses Vorgehen zu einer Abfrage pro Entity-Klasse, im oberen Beispiel eine Abfrage für die Positionen und eine weitere Abfrage für die Aufträge. Die Abfrage für das Laden der Aufträge nutzt eine **in**-Bedingung (siehe Codebeispiel 24) um alle Aufträge über ihre ID zu laden. Die IDs sind alle bekannt, da die Tabelle der Positionen für jede Position die ID des zugehörigen Auftrags enthält. Würde von den Aufträgen auf die Positionen zugegriffen werden, würde ebenfalls eine **in**-Bedingung mit den IDs der Aufträge genutzt werden (siehe Codebeispiel 25).

Codebeispiel 24: Batch-Fetch-Abfrage für Aufträge

```
select * from auftrag a where a.id in (?, ?, ?);
```

Codebeispiel 25: Batch-Fetch-Abfrage für Positionen

```
select * from position p where p.auftrag_id in (?, ?, ?);
```

In EclipseLink lässt sich Batch-Fetching mit der Annotation `@BatchFetch(value = BatchFetchType.IN)` nutzen (siehe Eclipse, BatchFetch, 2017), in Hibernate kann die Annotation `@BatchSize` (siehe Mihalcea u. a., 2018, Kapitel 11.8. Batch fetching) oder die Property `hibernate.default_batch_fetch_size` in der Persistence-XML-Datei verwendet werden (siehe Mihalcea u. a., 2018, Kapitel 23.9.1. Fetching properties). Um so größer die Batch-Fetch-Size eingestellt wird, umso mehr IDs können in der **in**-Bedingung verwendet werden und umso weniger Abfragen müssen ausgeführt werden. Es gibt jedoch in manchen Datenbanken eine maximale Größe für **in**-Bedingungen, so sind in Oracle-SQL höchstens 1000 Werte erlaubt (vgl. Oracle, 2018, Kapitel 6, S. 35). In diesem Fall darf die Batch-Fetch-Size höchstens auf 1000 gestellt werden.

Wenn die verwendete Datenbank ihre Ausführungspläne speichert, ist es schlecht für die Performance, wenn eine Anwendung viele verschiedene Abfragen nutzt, da der Cache der Ausführungspläne eine begrenzte Größe hat. Werden viele verschiedene Abfragen genutzt kann der Cache nicht für jede Abfrage den Ausführungsplan speichern, was bedeutet, dass häufiger Abfragen neu geparkt und analysiert werden müssen. **in**-Bedingungen können besonders leicht für einen Überlauf des Ausführungsplancaches sorgen, da sich die Anzahl der Parameter einer **in**-Bedingung häufig ändert und sich somit eine andere SQL-Abfrage ergibt. In Hibernate gibt es aus diesem Grund die Persistence-XML-Property `hibernate.query.in_clause_parameter_padding`. Wird diese Property auf **true** gestellt, wird als Anzahl der Abfrageparameter immer eine zweier Potenz verwendet, indem einer der Parameter mehrmals eingesetzt wird (vgl. Hibernate, IN_CLAUSE_PARAMETER_PADDING, 2018). Es ist zwar nicht notwendig, dieses Feature nur für Batch-Fetch-Abfragen zu nutzen, da diese bereits eine ähnliche Performanceoptimierung verwenden, doch wenn dieses Feature für **in**-Bedingungen in expliziten Datenbankabfragen der Anwendung genutzt wird, sollte die Property `hibernate.batch_fetch_style` auf `dynamic` gestellt werden, damit die beiden Performanceoptimierungen sich nicht in die Quere kommen (siehe Hibernate, BatchFetchStyle, 2018).

Durch Batch-Fetching wird das n+1 Problem sowohl für eager, als auch lazy Beziehungen behoben. In Hibernate gibt es allerdings einen Bug, durch den bei

bidirektionalen 1-zu-1-Beziehungen nur von der Besitzerseite (die Seite mit der Fremdschlüsselspalte), nicht aber von der Gegenseite Batch-Fetching genutzt wird (siehe Uffmann, HHH-12711, 2018). Das führt dazu, dass bei 1-zu-1-Beziehungen weiterhin das n+1 Problem auftreten kann, wenn Entitäten mit der Gegenseite einer 1-zu-1-Beziehung geladen werden. Erschwerend kommt noch hinzu, dass durch einen weiteren Bug die Gegenseite einer 1-zu-1-Beziehung nicht lazy sein kann (siehe Uffmann, HHH-12709, 2018). Hier scheint bis zur Behebung des oberen Bugs die einzige Möglichkeit, um das n+1 Problem zu vermeiden, zu sein, dass sichergestellt ist, dass die Entitäten der besitzenden Seite in den JPA-Cache geladen werden, bevor die Entitäten der Gegenseite geladen werden. Statt alle Abfragen anzupassen, die im Kontext von 1-zu-1-Beziehungen verwendet werden, kann auch ein Load-Graph genutzt werden (siehe Codebeispiel 26). Hierbei handelt es sich um ein JPA-Feature, mit dem für einzelne Datenbankabfragen geändert werden kann, welche Beziehungen von der Abfrage zusätzlich geladen werden sollen (vgl. Jungmann & DeMichiel, 2017, S. 120 ff.).

Codebeispiel 26: Setzen eines Load-Graphs

```
public List<Auftrag> loadAuftraege() {
    Query query = entityManager.createQuery("from Auftrag");
    EntityGraph<?> entityGraph = entityManager.getEntityGraph("Auftrag");
    query.setHint("javax.persistence.loadgraph", entityGraph);
    return query.getResultList();
}

@Entity
@NamedEntityGraph(attributeNodes = @NamedAttributeNode("auftragInfo"))
public class Auftrag {
    @OneToOne(mappedBy = "auftrag")
    private AuftragInfo auftragInfo;
}

@Entity
public class AuftragInfo {
    @OneToOne(fetch = FetchType.LAZY)
    private Auftrag auftrag;
}
```

Die Implementierung der bidirektionalen 1-zu-1-Beziehung ist in Anhang 2 zu finden.

4 Umfrage über die Auswirkungen von bidirektionalem Mapping

Um eine belastbarere Schlussfolgerung aus dem Vergleich von Testdatenerzeugung mit Nutzung des Builder-Patterns und Testdatenerzeugung mit Nutzung des Builder-Patterns und bidirektionalem Mapping ziehen zu können, wurde eine Umfrage unter den Mitarbeitern durchgeführt. Dabei wurden alle Mitarbeiter befragt, die in der betrachteten Firma bereits mit dem Builder-Pattern und bidirektionalem Mapping, gearbeitet hatten. Befragt wurden neun Mitarbeiter, von denen acht antworteten. Damit ist die Umfrage zumindest repräsentativ für die betrachtete Firma. Um eine allgemein repräsentative Umfrage durchführen zu können, müsste zunächst das Builder-Pattern in Kombination mit bidirektionalem Mapping in weiteren Firmen eingesetzt werden.

4.1 Gestaltung der Umfrage

Um besser abschätzen zu können, welche Fragen in einer Umfrage von den Teilnehmern sinnvoll beantwortet werden können und hilfreich für die Bewertung des Einsatzes von bidirektionalem Mapping sind, wurde ein Experteninterview mit Michael Karneim, dem Softwarearchitekt der betrachteten Firma, durchgeführt (siehe Anhang 1). Die vier Hauptaspekte, zu denen Fragen gestellt wurden, sind: Beurteilung der Auswirkungen auf Testcode, Beurteilung der Auswirkungen auf Produktivcode und allgemeine Beurteilung. Zusätzlich wurden am Anfang der Umfrage noch ein paar Fragen darüber gestellt, ob der Mitarbeiter auch ohne bidirektionales Mapping gearbeitet hat und ob er an einer Umstellung eines Projektes beteiligt war. Der komplette Fragebogen ist in Anhang 4 zu finden.

Für die Beurteilung der Auswirkungen auf den Testcode wurde mit einer Frage begonnen, bei der die Teilnehmer zwei konkrete Testfälle vergleichen und deren Verständlichkeit beurteilen sollten. Dafür wurden leicht modifizierte Varianten von Codebeispiel 15 und Codebeispiel 16 miteinander verglichen. Diese Beispiele wurden ausgewählt, da sie sehr eindeutig die verschiedenen Vorteile, die sich durch den Einsatz von bidirektionalem Mapping im Testcode ergeben, zeigen. Hierbei sollte noch angemerkt werden, dass Codebeispiel 15 zwar, wie in Kapitel 3 beschrieben, mit unidirektionalen 1-zu-n-Beziehungen realisiert ist, eine

bidirektionale Implementierung jedoch syntaktisch zu demselben Testfall führen würde. Durch die Nutzung eines Codebeispiels in der ersten inhaltliche Frage, sollte auch nochmals klar gestellt werden, welche zwei Techniken miteinander verglichen werden, und dem Teilnehmer die Möglichkeit gegeben werden, sich anhand eines konkreten Codebeispiels an die Thematik zu erinnern.

Anschließend wurde eine Frage über die Entwicklungsgeschwindigkeit von Testfällen gestellt. Hier hatte sich im Experteninterview ergeben, dass eine Frage nach konkreten Zahlen über die durchschnittliche Entwicklungsdauer oder Änderung der Entwicklungsdauer kaum hilfreiche Antworten ergeben würde, da die Einschätzung der Entwicklungszeit ohne echte Messungen für Teilnehmer schwer zu beurteilen sei. Aus diesem Grund wurde eine Frage mit vorgegebenen Antworten gewählt, bei der die Befragten angeben sollten, ob sie nach ihrer eigenen Einschätzung durch bidirektionales Mapping Tests im Durchschnitt deutlich schneller, etwas schneller, gleich schnell, etwas langsamer oder deutlich langsamer schreiben können. Die letzte Frage über die Auswirkungen auf den Testcode war eine offene Frage, die eine allgemeine Beurteilung ermöglichte.

Für die Beurteilung der Auswirkungen auf den Produktivcode wurden vier Teilaspekte identifiziert:

1. Komplexität in den Entity-Klassen, durch den Code für den Auf- und Abbau der bidirektionalen Beziehungen (siehe Kapitel 3.1 und 3.3.1)
2. Wartbarkeit von Produktivcode, der die bidirektionalen Beziehungen nutzt (vgl. Anhang 1).
3. Tendenz zu Objektorientierung statt Serviceorientierung durch bidirektionales Mapping
4. Performance von Produktivcode mit bidirektionalem Mapping

Zu jedem dieser Teilaspekte wurde eine offene Frage gestellt, sodass die Befragten ihre Beurteilung frei abgeben konnten. So konnte ein möglichst umfangreicher Eindruck von den Meinungen der Teilnehmer gewonnen werden.

Abschließend wurden noch Fragen über die allgemeine Bewertung von bidirektionalem Mapping gestellt. Dafür wurde noch je eine offene Frage nach

weiteren Vorteilen und weiteren Nachteilen gestellt, um mögliche Erfahrungen einzufangen, die nicht berücksichtigt wurden und die Befragten sollten in einer weiteren offenen Frage den Aufwand für den Einsatz von bidirektionalem Mapping mit dem Nutzen vergleichen und bewerten. Abschließend wurden noch zwei Ja-Nein-Fragen darüber gestellt, ob der Befragte bidirektionales Mapping in einem neuen, bzw. bestehenden Projekt einsetzen, bzw. einführen würde.

4.2 Durchführung der Umfrage

Die Mitarbeiter wurden ausgewählt, indem mit allen Java-Mitarbeitern ein kurzes persönliches Gespräch geführt wurde, um festzustellen, ob sie bereits mit bidirektionalem Mapping in der Firma gearbeitet hatten. Das Builder-Pattern wird in allen neueren Java-Anwendungen verwendet, daher war jeder Mitarbeiter, der bidirektionales Mapping verwendet hatte, auch mit dem Builder-Pattern vertraut. Nachdem die Mitarbeiter ihre Bereitschaft zur Teilnahme an der Umfrage ausgedrückt hatten, wurde die Umfrage per E-Mail als docx-Dokument zugestellt, mit dem Hinweis das ausgefüllte Dokument bis eine Woche später zurückzusenden. Von den neun ausgewählten Mitarbeitern beantworteten acht die Umfrage.

4.3 Auswertung der Umfrage

Die Namen der Teilnehmer wurden pseudonymisiert. An der Umfrage nahmen 5 Entwickler (E1-E5), ein Praktikant (P), ein Mitarbeiter der Qualitätssicherung (Q) und der Softwarearchitekt Michael Karneim (A) teil.

Vergleich der Codebeispiele

Alle acht Teilnehmer gaben an, dass der Test mit bidirektionalem Mapping (Codebeispiel 15) leichter verständlich sei, als der Test ohne bidirektionales Mapping (Codebeispiel 16). Als Begründung wurden unter anderem die folgenden Punkte angeführt:

- Kleinerer Test-Kontext (A, E2, E3, E5, P, Q)
- Kompakterer Code (E1, E3, Q)

- Bessere (baumartige/deklarative) Struktur (A, E2, E3)
- Flüssiger lesbar (E3, E4)

Auswertung der Frage: "Können Sie nach ihrer eigenen Einschätzung durch bidirektionales Mapping Tests durchschnittlich schneller oder langsamer schreiben?"

- Deutlich schneller (A, E1, P, Q)
- Etwas schneller (E2, E3, E4)
- Gleich schnell (E5)
- Etwas langsamer
- Deutlich langsamer

50 % der Teilnehmer wählten die Option "Deutlich schneller", 37,5 % der Teilnehmer wählten "Etwas schneller" und 12,5 % der Teilnehmer wählten "Gleich schnell". Keiner der Teilnehmer wählte die Optionen "Etwas langsamer" oder "Deutlich langsamer". Insgesamt fühlten sich also 87,5 % der Teilnehmer durch bidirektionales Mapping schneller beim Schreiben von Tests.

Auswertung der Frage: "Wie beurteilen Sie die Nutzung von bidirektionalem Mapping im Testcode (gegenüber rein unidirektionalem Mapping)?"

Positive Äußerungen:

- Leichteres Schreiben von Tests (A, E1, E2, E4)
- Schnelleres Schreiben von Tests (E1, E2, E3)
- Lesbarer / übersichtlicher / wartbarer Code (E1, E3)

Negative Äußerungen:

- Nur übersichtlicher, solange die Menge an Testdaten nicht zu groß ist (E3)
- Nutzung von vielen Klammern ist unübersichtlich (P)

50 % der Teilnehmer (A, E1, E2, Q) äußerten sich ausschließlich positiv, 37,5 % der

Teilnehmer (E3, E4, P) äußerten sich überwiegend positiv, und 12,5 % der Teilnehmer äußerten sich neutral.

Der Teilnehmer E4 führte Schwierigkeiten beim Speichern durch Foreign-Key-Constraints an, diese konnten allerdings nach Rücksprache nicht reproduziert werden. Entweder handelte es sich hierbei um einen selten auftretenden Bug des ORM-Systems oder um einen Fehler im Code dieses Entwicklers. Ohne den Fehler reproduzieren zu können ist das schwer zu beurteilen. Da der Fehler nicht weiter auftrat, wird er in der Auswertung nicht weiter berücksichtigt.

Auswertung der Frage "Wie beurteilen Sie die erhöhte Komplexität der JPA-Beans durch bidirektionales Mapping?"

Bei dieser Frage gingen die Antworten weit auseinander. Die eine Hälfte der Teilnehmer (A, E1, E4, P) beurteilte die Komplexität als negativ, während die andere Hälfte (E2, E3, E5, Q) die Komplexität eher neutral sah. Es wurden sogar zwei positive Argumente genannt:

- Deckt Schwachstellen im Datenmodell auf (E2)
- Erleichtert Übersicht über die Beziehungen zwischen Entity-Klassen (E4, P)

Als negative Argumente wurden unter anderem die folgenden Punkte genannt:

- Kann schwer auffindbare Bugs verursachen (E1)
- Erhöhter Wartungsaufwand (E4, P)

Zwei der Teilnehmern (A, E1) führten außerdem an, dass die negativen Auswirkungen der erhöhten Komplexität durch Kopiervorlagen, bzw. generische Tests zum Teil ausgeglichen werden können.

Auswertung der Frage: "Wie beurteilen Sie die Wartbarkeit von Produktivcode, der bidirektionales Mapping aufruft (gegenüber Produktivcode, der ausschließlich unidirektionales Mapping aufruft und gegebenenfalls zusätzliche Queries absetzt)?"

Teilnehmer E2 konnte diese Frage nicht beantworten, da bidirektionales Mapping

im entsprechenden Projekt bisher nicht im Produktivcode verwendet wurde. Von den restlichen sieben Teilnehmern äußerten sich 57,1 % (A, E1, E4, Q) eher positiv, 18,6 % (E5, P) eher neutral und 14,3 % (E3) eher negativ.

Dabei führte Teilnehmer E3 an, dass die stärkere Kopplung zwischen den Entity-Klassen durch bidirektionales Mapping einen negativen Effekt auf die Wartbarkeit habe. Auch Teilnehmer E5 äußerte Bedenken bezüglich der Wartbarkeit, allerdings bezog sich E5 auf die Möglichkeit von Fehlern im bidirektionalen Mapping. Der Großteil der Teilnehmer war sich allerdings einig, dass sich die Wartbarkeit des Produktivcodes durch bidirektionales Mapping verbessert.

Auswertung der Frage: "Wie beurteilen Sie die Änderungen der Software Architektur durch bidirektionales Mapping (insbesondere den zunehmenden Fokus auf Objektorientierung gegenüber Serviceorientierung)?"

Wie auch bei der vorangegangenen Frage konnte Teilnehmer E2 diese Frage nicht beantworten. Von den restlichen sieben Teilnehmern äußerten sich 85,7 % (A, E1, E4, E5, P, Q) positiv und 14,3 % (E3) neutral. Als positiv wurde unter anderem angeführt, dass der Code verständlicher (P) und intuitiver (Q) ist und die Architektur übersichtlicher wird (E1). Darüber hinaus führte Teilnehmer A an, dass sich die Lokalität verbessert, da die Distanzen zwischen Daten und Funktionen verkürzt werden.

Teilnehmer E3 führte als negativen Punkt an, dass die Kohäsion der Entity-Klassen sinkt, da diese zu Sammelklassen für mehr oder weniger Entity bezogenem Code werden. Außerdem bezeichnete er die Kopplung der Entity-Klassen als Architekturdefizit. Trotzdem schreibt Teilnehmer E3, dass die Verbesserung der Testbarkeit, besonders bei kleinen Projekten, die Nachteile ausgleichen könne.

Auswertung der Frage: "Wie beurteilen Sie die Performance von Produktivcode, der bidirektionales Mapping nutzt?"

50 % der Teilnehmer (E1, E2, E3, E4) konnten diese Frage nicht beantworten. Teilnehmer E5 nannte Erfahrungen mit schlechter Performance, räumte aber auch ein, dass diese nicht unbedingt durch bidirektionales Mapping verursacht wurden.

Teilnehmer A äußerte sich weder positiv noch negativ und führte stattdessen einige Punkte an, die zu beachten sind, um gute Performance zu erreichen. Zwei der Teilnehmer (P, Q) äußerten sich positiv, wobei Teilnehmer P selbst angab, wenig Erfahrung diesbezüglich zu besitzen.

Insgesamt scheint die Performance mit bidirektionalem Mapping also kein Problem zu sein, da nur ein Teilnehmer (E5) sich leicht negativ und ein Teilnehmer (Q) sich positiv äußerte. Aus den Antworten der Teilnehmer A und E5 geht allerdings hervor, dass die Nutzung von bidirektionalem Mapping mehr Expertise erforderlich macht.

Auswertung der Frage: "Was sind aus Ihrer Sicht weitere Vorteile, die sich durch den Einsatz von bidirektionalem Mapping gegenüber dem Einsatz von unidirektionalem Mapping ergeben?"

Einige der Teilnehmer führten in dieser Frage zusammenfassend einige der bereits besprochenen Punkte an, es gab allerdings auch ein paar Vorteile, die bisher noch nicht besprochen wurden. So schrieb Teilnehmer P, dass m-zu-n-Beziehungen mit zusätzlichen Attributen leichter umsetzbar seien und Teilnehmer E4 meinte, dass zu bidirektionalem Mapping mehr Beispiele im Internet zu finden seien. Da die Aussage von Teilnehmer E4 allerdings als Vor- und als Nachteil aufgeführt wurde, wird diese Antwort nicht weiter betrachtet.

Teilnehmer E5 nannte außerdem den Vorteil: "Man muss weniger Objekte hin- und herreichen. Konsistenz kann immer lokal sichergestellt werden". Damit ist vermutlich gemeint, dass Methoden zum Teil weniger Parameter benötigen, da mehr Kontextinformationen über die Entity-Klassen verfügbar sind. Statt zum Beispiel einer Methode einen Auftrag und eine Position zu übergeben, die möglicherweise nicht zusammen passen, muss bei bidirektionalem Mapping nur eine Position übergeben werden und die Methode hat trotzdem Zugriff auf die Auftragsinformationen.

Auswertung der Frage: "Was sind aus Ihrer Sicht weitere Nachteile, die sich durch den Einsatz von bidirektionalem Mapping gegenüber dem Einsatz von unidirektionalem Mapping ergeben?"

25 % der Teilnehmer (A, E3) nannten bei dieser Frage keine Nachteile. Die anderen 75 % der Teilnehmer (E1, E2, E4, E5, P, Q) führten hier noch einmal die Komplexität des bidirektionalen Mappings und den damit verbundenen Aufwand auf. Lediglich Teilnehmer Q nannte zusätzlich noch einen weiteren Punkt, und zwar die Performanceprobleme von 1-zu-1-Beziehungen, die im Ende von Kapitel 3.3.2 behandelt werden. Daher scheint die Komplexität der Mapping-Implementierung der Hauptnachteil von bidirektionalem Mapping zu sein.

Auswertung der Frage "Wie beurteilen Sie den Aufwand für den Einsatz von bidirektionalem Mapping im Verhältnis zum Nutzen und begründen Sie?"

Teilnehmer E4 machte bei dieser Frage keine Angabe. Von den restlichen sieben Teilnehmern antworteten 100 % (A, E1, E2, E3, E5, P, Q), dass der Aufwand den Nutzen überwiegt. Davon gaben 42,9 % (E1, E2, E3) an, dass es einen erhöhten initialen Entwicklungs- bzw. Einarbeitungsaufwand gab.

Auswertung der Frage: "Würden Sie in einem neuen Projekt bidirektionales Mapping einsetzen?"

87,5 % der Teilnehmer (A, E1, E3, E4, E5, P, Q) antwortete mit "Ja", die restlichen 12,5 % (E2) machten keine Angabe.

Auswertung der Frage "Würden Sie in einem bestehenden Projekt mit unidirektionalem Mapping langfristig auf bidirektionales Mapping umstellen?"

75 % der Teilnehmer (A, E1, E2, E4, E5, Q) antwortete mit "Ja", die restlichen 25 % (E3, P) wählten als Antwort "Nein".

5 Fazit

Die Forschungsfrage kann positiv beantwortet werden. Die Wartbarkeit von Testcode mit dem Builder-Pattern kann durch den Einsatz von bidirektionalem Mapping verbessert werden. So äußerten sich 87,5 % der befragten Mitarbeiter positiv über die Nutzung im Testcode und gaben an, dass sie durch bidirektionales Mapping nach eigener Einschätzung etwas oder sogar deutlich schneller Tests schreiben konnten. Diese Vorteile werden vor allem durch eine erhöhte Komplexität der Entity-Klassen erkauft, welche von 75 % der befragten Mitarbeiter, als Nachteil aufgeführt wurde. Diese Komplexität lässt sich jedoch durch Kopiervorlagen und generische automatisierte Tests zum Teil ausgleichen.

Darüber hinaus hatte der Einsatz von bidirektionalem Mapping unerwartet positive Auswirkungen auf den Produktivcode. So gaben von den Befragten mit Erfahrung mit bidirektionalem Mapping im Produktivcode 57,1 % an, dass das bidirektionale Mapping einen positiven Einfluss auf die Wartbarkeit des Produktivcodes hat und sogar 85,7 % beurteilten damit einhergehende Softwarearchitekturänderungen als positiv. Das liegt vor allem daran, dass bidirektionales Mapping objektorientierte Programmierung unterstützt, was von vier der acht Befragten explizit angegeben wurde.

Bidirektionales Mapping hatte bei korrektem Einsatz in den meisten Fällen keine negativen Auswirkungen auf die Performance. Bedingt durch Bugs in Hibernate sind allerdings bei 1-zu-1-Beziehungen zusätzliche Maßnahmen erforderlich um Probleme mit $n+1$ Abfragen zu verhindern (siehe Kapitel 3.3.2). Daher kann es bei 1-zu-1-Beziehungen in manchen Fällen sinnvoll sein, auf bidirektionale Beziehungen zu verzichten, bis die entsprechenden Bugs behoben sind.

Auch wenn die Umfrage mit nur acht Teilnehmern aus einer Firma nicht repräsentativ genug ist um eine allgemeingültige Antwort zu treffen, ist doch eine eindeutige Tendenz festzustellen, insbesondere da 100 % der Teilnehmer angaben, dass der Nutzen des bidirektionalen Mappings den Aufwand überwiegt.

6 Ausblick

Bidirektionales Mapping wurde in Kombination mit dem Builder-Pattern in der betrachteten Firma erfolgreich eingesetzt und hatte die erwünschten Vorteile ohne überwiegende Nachteile. Die Umfrage, auf der dieses Ergebnis beruht, wurde allerdings mit einer kleinen Anzahl an Mitarbeitern und in nur einer Firma durchgeführt. Daher sollte dieses Ergebnis mit einer größeren Auswertung verifiziert werden. Dies ist allerdings erst möglich, wenn weitere ORM-Anwendungen auch außerhalb der betrachteten Firma das Builder-Pattern und bidirektionales Mapping einsetzen. Darüber hinaus ergeben sich aus dem Erfolg des bidirektionalen Mappings noch weitere Forschungsthemen:

- Der Vergleich von Builder-Pattern und bidirektionalem Mapping mit anderen Techniken der Testdatenerzeugung.
- Da in dieser Arbeit nur die beiden JPA-Implementierungen EclipseLink und Hibernate untersucht wurden, wäre eine weitere mögliche Forschungsfrage, ob auch andere ORM-Frameworks, möglicherweise auch in anderen Programmiersprachen, die notwendigen Features unterstützen, um bidirektionales Mapping nutzbringend einzusetzen.
- Untersuchung der Auswirkungen von bidirektionalem Mapping auf den Produktivcode durch Metriken.
- Gibt es mögliche Einsatzfelder von den Implementierungen der bidirektionalen Beziehungen außerhalb von ORM?

Unabhängig von dem Builder-Pattern oder bidirektionalem Mapping wurde ein Mangel an Forschung zu Testcodequalität festgestellt. Bei Testcode lassen sich Metriken für Produktivcode nur eingeschränkt nutzen, daher ist die Evaluierung der Wartbarkeit schwieriger. Dies ist vermutlich darauf zurückzuführen, dass Testcode im Bewusstsein vieler Entwickler immer noch einen geringeren Stellenwert hat als Produktivcode. Metriken, die explizit für Testcode erstellt wurden, könnten auch in anderen Projekten helfen, mehr Aufmerksamkeit auf die Tests zu lenken und deren Qualität so zu steigern.

Anhang 1: Experteninterview mit Michael Karneim

Uffmann: Das ist jetzt ein Interview für meine Bachelorarbeit, das Thema ist: "Verbesserung der Wartbarkeit von automatisierten Tests durch Nutzung von Builder-Pattern und bidirektionalem Mapping". Kannst du dich einmal kurz vorstellen?

Karneim: Mein Name ist Michael Karneim, ich bin Softwarearchitekt bei der Firma Sanacorp und nutze automatisiertes Testen sinnvollerweise ungefähr seit 2011.

Uffmann: Die testgetriebene Entwicklung war ja hier ausschlaggebend dafür, dass wir das Builder-Pattern einsetzen oder dass wir überhaupt den Fokus so sehr auf die automatisierten Tests legen. Da wollte ich kurz zum Hintergrund fragen: warum sollte testgetriebene Entwicklung überhaupt in der Sanacorp eingeführt werden?

Karneim: Also der primäre Grund ist natürlich, dass wir unsere Codequalität verbessern, insgesamt die Wartbarkeit der Software, dass wir weniger Fehler haben in Produktion. Der sekundäre Grund, also die eigentliche auslösende Ursache, warum wir es dann eingeführt haben war, weil wir ein großes Projekt durchgeführt haben, bei dem wir einen Webservice anzubieten haben der Industriestandard ist und wir wollten eben sicherstellen, dass wir da drin keine Fehler machen, weil dort wird unser Auftragseingang sozusagen stattfinden und wenn da Fehler drin sind, dann kann das eben sehr teuer werden.

Uffmann: Und wie lange wurde vorher schon versucht testgetriebene Entwicklung einzusetzen?

Karneim: Also wir haben 2004 glaube ich zum ersten Mal versucht automatisierte Tests zu schreiben, also wir haben auch welche geschrieben, aber meistens haben die dann nicht bis sagen wir mal 2010 überlebt. Die meisten Projekte, die irgendwie Tests hatten, haben die dann sukzessive verloren aus verschiedensten Gründen. Das heißt also, der Wille war da, aber leider wussten wir nicht genau, wie man es

richtig macht. Es wurde immer schwieriger die Tests mit dem Code gleichzeitig zu evolvieren.

Uffmann: Und 2011 hat es dann das erste Mal richtig funktioniert?

Karneim: Genau, da haben wir damit angefangen und da wussten wir noch nicht, ob es diesmal funktioniert, aber 2012 etwa konnte man dann schon sehen: das funktioniert jetzt.

Uffmann: Und was waren aus deiner Sicht die größten programmiertechnischen Schwierigkeiten beim Einführen von testgetriebener Entwicklung?

Karneim: Also bei den Altsystemen war die Problematik, dass jedes Mal, wenn man irgendetwas an der API geändert hat, an der Produktiv-API, die man testet, musste man durch alle Tests durchgehen und überprüfen, also die konnten nicht mehr kompilieren, man musste quasi jeden einzelnen anschauen und sich überlegen, was muss ich jetzt hier tun. Ja jetzt kommt hier ein zusätzlicher Parameter, ist der für diesen Test relevant oder nicht. Und wenn der Entwickler, der den neuen Parameter eingefügt hat, in den Code aber die Tests nicht kannte, dann konnte er nicht sagen, was hier sinnvoll ist und dann hat er das auskommentiert und gesagt: "das kann ja jemand später mal fixen" und das hat aber dann nie jemand getan. Das ist so eine Problematik. Die andere Problematik ist, dass wir die Testdaten, die wir benutzt haben normalerweise gar nicht im Test stehen hatten, sondern wir hatten Datenkonstellationen in der Datenbank. Die wurden vorbereitet in der Testdatenbank und da kannte man die IDs, die Primärschlüssel von diesen Testdaten und hat die dann einfach eingeladen und benutzt und die Assertions haben dann geprüft: ist das Ergebnis auch genau so wie man es erwartet? Das funktioniert auch so, aber diese Testdaten waren nicht sehr stabil in der Datenbank, irgendwann hat jemand irgendetwas geändert, vielleicht weil er einen anderen Test schreiben wollte und dann ist der erste Test fehlgeschlagen und das hat man vielleicht nicht sofort bemerkt, weil wir die Tests auch gar nicht immer sofort ausgeführt haben, wenn wir was geändert haben, sondern irgendwann später kurz

bevor man halt ein Release gemacht hat, hat man mal die Tests laufen lassen und das war dann auch beliebig lange, vielleicht 8 Wochen nach so einer Änderung und dann wusste man nicht mehr, was ist jetzt eigentlich hier schiefgelaufen und derjenige der das vielleicht gemacht hat, also die Daten verändert hat, wusste nicht, welche Daten der ursprüngliche Test gebraucht hat. Weil die waren ja auch nirgendwo dokumentiert und die waren ja auch mittlerweile weg. Ja und dann hat man auch diesen Test auskommentiert.

Uffmann: Und letztendlich hat es sich dann gelohnt, also kann man das quasi auch irgendwie belegen, jetzt zum Beispiel die Rate der Produktionsprobleme, ist die durch den Einsatz von automatisierten Tests gesunken?

Karneim: Also ich habe jetzt keine konkreten Zahlen, sondern eher so eine Art Bauchgefühl. Normalerweise ist es so, wenn wir also vor 2011 mit einer Software in Produktion gegangen sind, mit der 1.0 oder mit der 1.0.0, dann hat das vielleicht einen Tag gedauert und man hat die ersten Bugreports gehabt und musste dann innerhalb von einer Woche schon die nächsten Bugfixes nachschieben. Und bei dem Projekt wo wir das eben zum ersten Mal erfolgreich eingesetzt haben ist es so, dass mehrere Monate lang die Version 1.0.0 in Produktion war, bevor wir überhaupt ein neues Release nachschieben mussten, allein daran erkennt man es.

Uffmann: Genau das waren ja jetzt nur ein paar Fragen zum Hintergrund. Jetzt geht es ja bei mir auch konkret darum: wir haben ja das System noch seit 2011 weiterentwickelt mit dem Builder-Pattern und dann jetzt eben vor kurzem auch mit bidirektionalem Mapping. Jetzt geht es ja bei meiner Arbeit auch vor allem darum zu vergleichen, wie das ohne und mit bidirektionalem Mapping ist. Da wollte ich ja noch eine Umfrage unter den Mitarbeitern durchführen und da wollte ich dir noch ein paar Fragen dazu stellen und dann im Anschluss auch, was du von den Fragen hältst, ob man diese Fragen, oder ähnliche Fragen auch in einer Umfrage so, ob du die für sinnvoll hältst, beziehungsweise ob dir dann noch da weitere einfallen, wo irgendwelche Aspekte, die noch nicht berücksichtigt sind. Und zwar - erstmal wollte ich vielleicht so einen Beispielttest nehmen, wo man quasi sieht, wie ist das mit und

ohne bidirektionalem Mapping geschrieben und dann kann man fragen, was davon ist einfacher zu verstehen. Dann wäre das sowas wie: "Was würden sie schätzen, wie viel Prozent schneller Tests mit bidirektionalem Mapping sich schreiben lassen?". Was würdest du dazu sagen? Also, wenn man nur so schätzt.

Karneim: Oh das ist schwierig zu sagen. Das hängt ja nicht nur davon ab, dass bidirektionales Mapping drin ist, sondern das hängt auch davon ab, dass die Testdatafactories richtig aufgesetzt sind.

Uffmann: Ja, wenn man jetzt davon ausgeht, dass die Testdatafactories richtig aufgesetzt sind.

Karneim: In beiden Fällen.

Uffmann: Genau in beiden Fällen und der einzige Unterschied ist quasi, ob man jetzt noch bidirektionales Mapping verwendet.

Karneim: OK, also ich kann vielleicht etwas dazu sagen, was tatsächlich der Entwickler anders empfindet. Wenn ich das bidirektionale Mapping habe, dann kann ich quasi die Entität, die ich gerade testen will, also das Hauptobjekt, an dem ich die Änderungen durchgeführt habe, dann kann ich hinschreiben: erzeuge mir mal so ein Ding. Und mehr mache ich nicht. Ich muss mir nicht darüber Gedanken machen: was muss ich denn alles noch erzeugen, damit das in der Datenbank landet. Während wenn ich ohne bidirektionales Mapping arbeite, dann geht das nicht so leicht. Ich muss möglicherweise von oben nach unten erzeugen. Also ich muss erst andere Objekte anlegen, bevor ich das Objekt, das ich anlegen will anlegen kann. Dazu muss ich mehr Kontextinformationen haben.

Uffmann: Muss ich das wirklich? Weil wenn ich kein... OK klar, wenn ich sowas habe, wie OneToOne-Relationen und die wären unidirektional, dann ist es schwieriger. Bei OneToMany-Relationen zum Beispiel würde mir üblicherweise ein unidirektionales Mapping reichen, um alle Kontextinformationen zu erzeugen.

Karneim: Also das kommt darauf an. Wenn ich einen Auftrag habe und ich habe eine Auftragsposition, jetzt ist die Frage: wie rum habe ich das Mapping? Habe ich es vom Auftrag zur Position oder andersherum? Wenn mein Test die Position braucht und ich kann aber von der Position nicht zum Auftrag navigieren, muss ich erst den Auftrag anlegen und dann die Position hinzufügen und das heißt das muss ich wissen. Wenn die Position auch auf den Auftrag zeigt, dann hilft mir natürlich die Testdatafactory, dann habe ich in beiden Fällen keinen Unterschied. Aber das ist genau der Punkt, von welcher Seite komme ich her und wenn ich bidirektional mappe, dann kann ich von beiden Seiten einen Test aufbauen und es sieht für mich komplett gleich aus, als Entwickler, und wenn ich das nicht habe, dann muss ich von der richtigen Seite anfangen und die muss ich kennen. Und wenn ich gleich mehrere Relationen über mehrere Schritte habe, so ein Projekt hatten wir hier mit dem Lagerspiegel Projekt, da muss man irgendwie 10 Relationen in eine Richtung aufbauen, von oben nach unten, bis man das eigentliche Objekt erst hat und die muss man alle kennen. Und das alleine, dieser Aufwand das zu kennen, ist schon größer und du musst natürlich auch die 10 Zeilen hinschreiben.

Uffmann: Da ist noch die Frage: könnte man da irgendwie eine sinnvolle Frage zu stellen, wo halt die Entwickler ihre Meinung zu äußern können? Also weil quantifizieren ist natürlich ein bisschen schwer, das verstehe ich schon gut, wenn man nur so Schätzwerte benutzt, aber hast du da eine Idee, was man da als sinnvolle Frage in einer Umfrage verwenden könnte, wenn man auf den Punkt hinaus will.

Karneim: Ich möchte vielleicht die Frage noch kurz vorher beantworten, ja also wie lange brauche ich? Ich würde sagen, wenn ich bidirektional mappe brauche ich egal, ob ich es von links oder von rechts aufziehe, gleich lang, während wenn ich unidirektional mappe brauche ich für die eine Seite immer noch kurz und für die Andere beliebig viel länger. Das ist der Punkt und wie lange ich brauche, ist davon abhängig, wie tief ich in dem Projekt drin sitze. Und wenn ich jetzt hauptberuflich nur dieses eine Projekt mache, dann ist es wahrscheinlich nicht so dramatisch. Wenn ich aber länger nicht mehr an dem Projekt gearbeitet habe oder so, dann

kriegen wir so eine Art "hab schon lange nicht mehr gesehen" Faktor dazu. Ich glaube, so ungefähr wird es sich wohl dann äußern. Genau das gleiche ist, wenn ich den Test lese. Ich muss halt in dem Fall 10 Zeilen lesen und ich muss erstmal überprüfen: sind die 10 Zeilen richtig? Da könnte ja ein Fehler drin sein. Es könnte eine fehlen und ich merke das ja nicht, wenn es nur 9 sind oder 10, woher soll ich das wissen. All diese Sachen. Also wenn ich jetzt irgendeine Zahl nennen soll, dann ist es nicht nur doppelt so schnell. Ja das ist locker im Faktor 10 Bereich, das ist eine ganze Größenordnung schneller. Also vor allem über die Wartungszeit hinweg, je seltener man sich im Code bewegt, desto extremer wird eigentlich dieser Faktor und genau das ist es ja: wir wollen ja viel Software schreiben, mit wenig Leuten und das heißt alle unsere Entwickler sind in mehreren Gebieten tätig und kommen seltener am gleichen Code vorbei und das kann ich nicht machen, wenn ich mich immer sehr gut auskennen muss, auch beim Testdaten Erzeugen und das führt dann schlussendlich auch dazu, dass man gar keinen Test schreibt. Möglicherweise ist dann die Zeit ihn zu erstellen unendlich, weil er gar nicht entsteht. Das dazu. So und welche Frage könnten wir jetzt...? Also wenn man Entwicklern die Frage stellt, wie lange er dafür braucht, ich glaube, der hat dann ein ähnliches Problem wie ich, dass er sagt: "Na ja, wenn ich den Code gut kenne, dann brauche ich nicht lang und wenn ich ihn nicht gut kenne, dann brauche ich länger".

Uffmann: Das heißt, dann müsste man eher auf sowas gehen, wie überhaupt mal die Frage zu stellen, für die Entwickler, die die Erfahrung gemacht haben, mit und ohne bidirektionalem Mapping, ob es sich so anfühlt, als ob sie mit bidirektionalem Mapping die Tests besser verstehen würden.

Karneim: Ja genau.

Uffmann: Das ist ja im Prinzip nur eine Ja-Nein-Frage. Und ich meine um wie viel besser ist dann ja auch nicht so wichtig. Wenn quasi jeder sagt, es ist besser, ist es ja auch eindeutig als Ergebnis.

Karneim: Ja vor allem was Zeit angeht: Entwickler lieben diese Frage nach Zeit ja eh nicht.

Uffmann: Ja das stimmt.

Karneim: Lines of Code kann man eher Fragen, wo brauche ich mehr Zeilen.

Uffmann: Eine andere Frage, die ich mir noch so mal überlegt hatte, war sowas wie: "Wie schätzen sie die erhöhte Komplexität der JPA-Beans durch das bidirektionale Mapping ein?". Was würdest du dazu sagen?

Karneim: Also die ist auf jeden Fall höher, weil wenn ich nicht bidirektional mappe, dann habe ich ein unidirektionales Mapping, was relativ leicht zu verstehen ist, weil das wird vor allem bei uns schon lange gemacht, also da hat kein Entwickler damit Probleme. Wenn ich bidirektional mappe muss ich mehrere Sachen beachten: ich muss erstmal beide Seiten richtig annotieren, richtig hinschreiben im Code, muss wissen, wie das geht. Das muss ich unter Umständen nachschauen, wenn ich ganz gut im Sattel sitze diesbezüglich. Dann muss ich die Relation ja von beiden Seiten pflegen, auch der Code kann Fehler enthalten. Das kann ich allerdings vereinfachen, also das haben wir ja auch vereinfacht, dein Code ist ja ein Teil davon, dass du quasi einen Prototypen geschrieben hast, für alle Varianten von bidirektionalem Mapping und auch Tests dazu, die einem dann auch beweisen, oder zumindest mal Hinweise darauf geben, dass der Code, wenn er kompiliert und läuft, dann ob er richtig ist oder falsch und das kann ich mir einfach kopieren. Da gibt es auch Codesnippets dazu, wo ich dann nur ein Tastaturkürzel eingebe und dann habe ich den richtigen Code dort stehen. Trotzdem ist es so, dass der entstandene Code ja dann da steht und interpretiert werden muss und da könnte sich ein Fehler einschleichen, also den man nicht erkennt, vom reinen Draufschauen. Das heißt, er ist natürlich komplexer. Aber die Komplexität lohnt sich, weil die habe ich ja nur einmal, an einer Stelle, nämlich genau da wo ich das einfüge und dann habe ich sie nicht mehr überall in meinen Tests. Und Tests habe ich ja erheblich mehr als Relationen und deswegen lohnt es sich.

Uffmann: Ja und dann kommt ja noch dazu, ich weiß nicht, ob du das schonmal gesehen hast, aber die Java 8 Variante von den bidirektionalen Mappings, dadurch, dass ich Lamdas verwenden kann, kann ich das ganze Auslagern und das ist ja nochmal viel kürzer. Das kommt auch noch dazu. Als Frage für die Entwickler in der Umfrage, wäre das eine gute Frage, wie man die Komplexität einschätzt?

Karneim: Ja. Ich glaube, da würden sie auch alle so antworten. Man kann ja einen Test machen, man kann ihnen...

Uffmann: Ich meine der Sinn von der Umfrage ist ja zu gucken, ob sie auch alle so antworten.

Karneim: Also man kann natürlich auch gemein sein und eine Implementierung hinlegen, die richtig ist und eine daneben, die falsch ist und sie sollen sagen, welche von den beiden bidirektionalen Mapping Implementierungen ist die Richtige.

Uffmann: Ja gut, aber dafür gibt es ja dann die Tests, die automatisch über alles drüber laufen. Insofern sagen die mir ja dann, welche Implementierung falsch ist und dann muss ich nur noch gucken: die ist falsch, die ist richtig und wo ist der Unterschied.

Karneim: Ja.

Uffmann: Dann hatte ich hier noch eine Frage: Was sind aus ihrer Sicht die größten Vor- und Nachteile die durch den Einsatz von bidirektionalem Mapping sich ergeben?

Karneim: Also für die Tests ist der große Vorteil, dass ich meine Testdaten entitätszentrisch aufbauen kann, also ich fange mit der Entität an, die mich am meisten interessiert und schreibe die als Erstes hin. Der Rest passiert dann under the hood und ich muss dann quasi nur noch Objekte hinzufügen, die ich halt speziell für meinen Test brauche und ich kann das so hinschreiben, wie ich im

Endeffekt rede. Damit ist der Test schnell hinzuschreiben. Ich kann auch einen bestehenden Test, der ähnlich ist, einfach kopieren und dann abändern, das sind dann auch nur 2-3 Zeilen, die ich anpassen muss und das funktioniert auch gleich direkt wieder. Also das heißt, ich kann schnell weiter Tests hinzufügen, wenn ich mal einen habe, ich kann sie eben leicht anpassen. Dann kann ich bestehende Tests, die jemand anders geschrieben hat, die so entstanden sind, auch viel leichter lesen und verstehen, weil ich nur das sehe, was relevant ist. Ich habe den ganzen anderen Kram, der nur deswegen im Test drin ist, damit man die Datensätze überhaupt in die Datenbank bringt, die sehe ich im Test nicht, das muss ich auch gar nicht sehen. Das heißt leichter verständlich. Dann ist es so, wenn ein Fehler auftritt an irgendeiner Stelle, ein Test raucht ab, kann ich reinschauen und sehe: "OK, was sind denn seine Testdaten" und sehe auch mehr oder weniger mit einem Blick: "ah ja das wird da gemacht", also ich verstehe auch gleich: berührt dieser Test überhaupt meine Fachlichkeit, mit der ich gerade gearbeitet habe oder nicht. Ich kann leichter entscheiden, ist dieser Test jetzt ursächlich mit beteiligt ist, also ist der Test nahe an der Ursache dran oder ist der einfach nur ein Kollateralopfer von meiner Routine, die ich irgendwo verändert habe und die falsch läuft.

Uffmann: Also durch das bidirektionale Mapping wird das Testdaten Setup deklarativer und deswegen sehe ich es quasi auf einen Blick, weil ich keine for-Schleifen oder sowas habe.

Karneim: Genau und ich kann mich dann, wenn viele Tests gleichzeitig abrauchen, kann ich schneller erkennen, habe ich den Test der nahe am Fehler dran ist oder nicht. Weil den würde ich ja dann als Referenz benutzen zum Reproduzieren des Fehlers und gegen den würde ich entwickeln beim Bugfixen.

Uffmann: Ansonsten Vorteile oder Nachteile?

Karneim: Ja dann gibt es ja Vorteile in Produktion, die wir am Anfang auch gar nicht vermutet hätten, oder zumindest gar nicht im Blick hatten, so ist es, dass man bidirektionales Mapping ja auch im Productivcode nutzen kann. Das führt dazu,

also wenn man sich denn traut, haben wir uns entschieden, dass wir sagen: "wir wollen unsere Software mehr objektorientiert schreiben", das heißt die Funktionen sollen zu den Daten. Im Fall von Hibernate und OR-Mapping heißt das also, dass die Preisberechnung möglicherweise direkt im Artikel drinnen liegt, also die Menge mal den Preis und damit habe ich im Endeffekt den Preis der Position oder sowas, oder auch wenn Rabatte dran hängen kann ich über den Auftrag nach oben navigieren, ich kann mir den Kunden suchen, ich kann überprüfen, ob der Kunde irgendwie Platinkunde ist und einen bestimmten Rabatt bekommt und kann das auch noch gleich rein multiplizieren und das kann ich direkt von der Position aus machen. Das war vorher überhaupt nicht möglich, weil die Position eben gar nicht an den Auftrag ran kam. Das ist das Eine, das heißt wir schreiben den Code dahin, wo er hingehört, wir können ihn auch dort testen und ein weiterer Vorteil, der noch entsteht ist: sollten wir merken, dass man den Code der da ist, objektorientiert, in Produktion nicht nutzen können, weil der einfach zu inperformant läuft, weil man eben durch eine Datenbank durch navigieren muss, kann man eine Performantere Variante implementieren, über einen Service, kann aber dann die Implementierung, die in den Objekten drinnen ist wiederverwenden, um dagegen zu testen und damit hat man quasi so eine Art doppelbödigen, Doppelnetztest, ich weiß nicht wie man das nennen soll. Man testet den einen Code gegen den Anderen und das ist auch recht praktisch.

Uffmann: Wobei, wenn man ja nur den Service hätte, dann müsste man ja auch nur den Service testen.

Karneim: Dann müsste man nur den Service testen, richtig, aber man hat dann eben noch diese Wiederverwendung, möglicherweise kann man es halt leichter hinschreiben, objektorientiert, und kann dann die performanceoptimierte Variante, die wahrscheinlich schwerer zu lesen ist, weil sie tricksen muss, die kann ich gegen die leichte Implementierung dann testen und das ist schon ein Vorteil.

Uffmann: Und ansonsten noch weiter Vor- oder auch Nachteile von bidirektionalem Mapping, die man so festgestellt hat.

Karneim: Also ein Nachteil ist, dass man jetzt nicht einfach hergehen kann und sagen ich baue jetzt bidirektionales Mapping in mein Datenmodell ein und benutze dafür Hibernate oder EclipseLink oder was auch immer und es geht out of the box, sondern das erfordert eine Feinjustierung der Implementierung. Es gibt auch in den Systemen, die wir da einsetzen Bugs oder gab Bugs, die man kennen muss und die muss man auch erstmal finden und um die muss man rum programmieren. Es gibt auch Performanceprobleme, um die man sich dann kümmern muss, beispielsweise wenn man mit einer Oracle Datenbank spricht und man irgendwie relationale Daten liest, wo mehr als 1000 Elemente drin sitzen möglicherweise, also wenn man eine ID-Query macht, dann geht das gar nicht, dann muss man sich da was überlegen, wenn man das tut. Und da haben wir dann, hast du dann, zusammen mit einem weiteren Mitarbeiter, so eine performanceoptimierte Lösung programmiert und die ist natürlich nicht mehr ganz trivial. Also das muss man alles machen, man muss viel im Backend schonmal vorbereiten, damit man diese Features dann auch so problemlos nutzen kann. Es ist halt so ein bisschen, ja man muss Pioniersarbeit leisten, aber wenn man das mal hat, also das ist jetzt ein Nachteil, den man hat als Investitionsmaßnahme, aber wenn man die Investition mal geleistet hat, man muss sie ja nur einmal bezahlen und danach kann man es ja immer wieder verwenden und dann wird es immer billiger logischerweise. Ein weiterer Nachteil: also möglicherweise sieht das Datenmodell in der Datenbank auch anders aus, also wir haben ja eh den Architekturansatz, dass wir sagen, wir wollen, dass unsere Anwendungen definieren, wie die Struktur der Daten in der Datenbank aussieht. Weil die Anwendung muss ja damit leben und möchte das natürlich optimal so designt haben, damit sie optimal damit arbeiten kann und dieser Schritt oder dieses Pattern das wir da verwenden oder diese Vorgabe, die hatten wir früher nicht. Früher war das andersherum, da haben wir gesagt, die Datenbank ist gegeben, die wird von irgendjemandem bereitgestellt und wir müssen uns mit der Anwendung oben draufsetzen und das funktioniert bei unidirektionalem Mapping noch einigermaßen OK, aber spätestens dann, wenn man bidirektional arbeitet, dann ist es wichtig, dass die Datenbank auf eine bestimmte Weise aussieht, so wie Hibernate das braucht, so wie wir das brauchen, damit das performant ist und spätestens dann muss man das Pattern umdrehen,

also sozusagen: "wir definieren wie unsere Daten aussehen" und das ist so ein Paradigmenwechsel und wenn man jetzt in einer Firma ist, wo man das nicht darf, dann ist das halt ein Hindernis, also da muss man dann halt Arbeit leisten und Überzeugungsarbeit leisten.

Uffmann: Wobei ist das wirklich durch bidirektionales Mapping oder ist das nicht eher dadurch, also wenn ich jetzt kein bidirektionales Mapping machen würde, würden wir ja trotzdem wollen, dass unsere Anwendungen das Datenmodell bestimmen und wenn ich jetzt das Datenmodell vorgegeben habe, bedeutet das bereits, dass bidirektionales Mapping nicht funktioniert, eigentlich doch nicht oder?

Karneim: Ne also wir hätten eigentlich schon viel früher sagen müssen: "unsere Anwendung definiert, wie das Datenmodell aussieht". Nur bis zu dem Zeitpunkt, also als wir noch unidirektionales Mapping hatten, hatten wir nicht diese üble Notwendigkeit. Ich sage es mal so, also als wir noch unidirektionales Mapping gemacht haben, haben wir auch so böse Sachen gemacht, wie Tabellen mehrfach zu mappen oder Views nochmal zusätzlich zu mappen, also Hibernate-Beans auf dieselben Daten zu setzen, die über Views gemappt sind, weil die dann performanter waren um die Daten abzufragen und das war allgemein akzeptiert und deswegen konnte die Datenbank aussehen, wie sie will, weil im schlimmsten Fall benutzen wir halt ein paar Views, um die Abfrage zu machen. So und wenn man jetzt bidirektional mappt geht das nicht mehr, also erstens geht es nicht mehr so richtig und zweitens will ich auch nicht dann mehrmals mappen, weil ich will ja dann die Vorteile haben, dass mein Datensatz nur einmal irgendwo beschrieben wird, weil das ist ja auch ein Vorteil. Den könnte man eigentlich noch Vorne nennen. Ich kann jetzt bei meinem Testaufbau, weil ich ja jede Information nur über ein einziges Bean abbilde, gibt es auch nur eine Stelle, wo ich die Daten eintragen muss. Und vorher war das eben so, dass wir die Informationen an mehreren Stellen hatten und das heißt, man hat mehrere Datensätze anpassen müssen für den gleichen Test und das ist eben weggefallen. Ich habe nur eine Schnittstelle gegen die Datenbank und durch den Einsatz von bidirektionalem Mappen ist dieser Prozess in Gang gestoßen worden, dass wir das so gemacht haben. Ich mag jetzt

nicht behaupten, dass wenn man das bidirektionale Mapping nicht hat, dass man dann das immer anders hätte oder falsch. Es hätte auch zufällig sein können, dass wir das auch schon hätten richtig gemacht, aber spätestens da fällt es halt auf, meines Erachtens.

Uffmann: Fallen dir sonst noch irgendwelche Vor- oder Nachteile ein?

Karneim: Wahrscheinlich fallen mir noch ein paar ein, wenn das Gespräch zu Ende ist. Lass mal überlegen. Ich meine es gibt natürlich die ästhetischen Vorteile, die im Code sind, es sieht halt einfach viel besser aus, auch beim Test schreiben schon.

Uffmann: Gut das geht ja eher mit dem Thema "Serviceorientierung versus Objektorientierung" einher, denke ich mal.

Karneim: Es ist wartbarer, es ist lesbarer, es ist modularer. Es ist auch so, wenn man das Problem einmal gelöst hat mit dem bidirektionalen Mapping, also auch die Performanceprobleme mit den In-Clause und so weiter und dieses Paket einfach wiederverwendet in einem neuen Projekt, dann kann man auch ein Entwickler sein, der das ursprüngliche Problem gar nicht kennt und der wird diesem Problem auch nie wieder über den Weg laufen, weil es ist eben grundsätzlich schon gelöst. Während wenn man das nicht einsetzt und man irgendwie mit handgeschriebenen Queries an die Datenbank rangehen muss, um an die Daten ran zu kommen, weil man eben über Code nicht navigieren kann, muss man jedes Mal wieder sich überlegen, wie schreibe ich den Query so, dass er performant ist und das brauche ich hier an der Stelle nicht. Ich kann vor allem simple Design first mäßig erst mal über diese Bidirektionalität navigieren, durch meinen Code und dann schaue ich, ist der Code denn schnell genug und erst dann, wenn ich merke, er ist nicht schnell genug, dann kann ich mir überlegen, ob ich noch eine Alternative dazu programmiere. Aber ich kann es erstmal so nutzen, wenn die Relation, wenn die Hibernate-Beans so gebaut sind, dann kann ich sie fachlich schon in meinem Code nutzen. Und das war vorher nicht so, vorher habe ich zwar die Hibernate-Beans gehabt, aber wenn ich da irgendwie ran wollte, musste ich immer noch Queries

schreiben. Ja und das fällt halt komplett weg. Ich glaube, das ist es so im Wesentlichen.

Uffmann: Und wenn man so eine Frage, also die Antwort war ja jetzt relativ komplex, aber natürlich will auch von den Entwicklern eigentlich den Eindruck haben, so was sie als Vor- oder Nachteile sehen. Würdest du dann die Frage so, also ich wiederhole die Frage nochmal kurz, das war: "Was sind aus ihrer Sicht die größten Vor- und Nachteile die sich durch den Einsatz von bidirektionalem Mapping ergeben?". Denkst du, das ist eine gute Frage, oder?

Karneim: Das finde ich, ist eine super Frage. Dann werden sicherlich alle erstmal so die Sachen beantworten, die ihnen halt am stärksten aufgefallen sind. Wenn man Entwickler hat, die nie die andere Variante gesehen haben, das haben wir ja auch, also wir machen es ja seit 2011, wir haben jetzt 2018, also da gibt es durchaus Entwickler, die später eingestiegen sind, die haben diese schlimme Phase nicht mitgemacht.

Uffmann: Ja wobei das bidirektionale Mapping war ja nicht 2011 schon.

Karneim: Ah ja stimmt, das ist später entstanden, aber auch da gibt es wahrscheinlich jemanden, der die andere Variante nicht gesehen hat. Also die ganz frisch Eingestiegenen. Da ist interessant, was die dann sagen, wenn sie die Alternative nicht kennen.

Uffmann: Ja stimmt. Welche Fragen würdest du ansonsten als sinnvoll erachten, also übergreifend. Das mit den Vor- und Nachteilen ist ja anscheinend sehr gut. Das "Wie schätzen sie die Komplexität der JPA-Beans durch bidirektionales Mapping ein?", was hattest du da dazu gesagt, ob das eine gute Frage ist?

Karneim: Das ist eine gute Frage, ja. Ich würde jetzt aber überrascht sein, wenn sie sagen, die ist gleich groß, aber ja.

Uffmann: Naja gut, "wie schätzen sie ein", dabei dachte ich halt, dass man auch darauf eingehen könnte, es ist komplexer und deswegen finde ich es irgendwie doof oder gut oder?

Karneim: Ja, wie bewerten sie das.

Uffmann: Ja genau, wie bewerten sie das. Genau dann hatte ich noch "Was würden sie schätzen, wie viel Prozent schneller", das ist natürlich dann ein bisschen schlecht, hattest du da eine andere Idee?

Karneim: Ja, also das ist, glaube ich, also wenn überhaupt kann man das ja nur beantworten, bezüglich wie viel schneller ist man selbst.

Uffmann: Genau da hatten wir ja gesagt, es ist wahrscheinlich am sinnvollsten nur zu sagen, ob man sich dadurch schneller fühlt. Genau und dann natürlich noch der Vergleich von zwei Tests. Natürlich welche, wo es auch einen Unterschied macht, ob ich bidirektionales Mapping verwende. Und dann, fallen dir sonst noch irgendwelche Aspekte ein bei bidirektionalem Mapping, die ich jetzt hier noch nicht berücksichtigt habe, oder die man vielleicht noch in einer Umfrage angehen sollte?

Karneim: Also man könnte vielleicht noch hinterfragen, ob es Tests gibt oder Sachen gibt, die man mit bidirektionalem Mapping jetzt nicht mehr machen kann, also ob es irgendwelche Nachteile gibt, also ganz massive, keine Ahnung, ich kann bestimmte Relationen nicht mehr machen, also ich weiß nicht vielleicht ist ja jemand auf sowas gestoßen.

Uffmann: Da würde ich dann sowas, also im Prinzip habe ich ja schon, welche Vor- und Nachteile sich durch den Einsatz von bidirektionalem Mapping ergeben, aber ich könnte natürlich extra noch eine Zusatzfrage stellen, ob es irgendwelche Einschränkungen gibt, die sich speziell durch bidirektionales Mapping ergeben.

Karneim: Yes. So was könnte man noch fragen? Es ist ja im Endeffekt, das

Hauptaugenmerk ist ja auf den Tests.

Uffmann: Ja ansonsten passt ja dann auch, dann habe ich ja vier, fünf Fragen.
Dann würde ich alle Leute, die ja schonmal mit bidirektionalem Mapping gearbeitet haben befragen. OK, das wäre es dann von meiner Seite her. Vielen Dank für das Interview.

Karneim: Gerne. Viel Erfolg mit der Arbeit.

Uffmann: Ja Dankeschön. Dann beende ich hiermit das Protokoll.

Anhang 2: Bidirektionale 1-zu-1-Beziehung

```

@Entity
@NamedEntityGraph(attributeNodes = @NamedAttributeNode("auftragInfo"))
public class Auftrag {
    @OneToOne(mappedBy = "auftrag")
    private @Nullable AuftragInfo auftragInfo;

    public @Nullable AuftragInfo getAuftragInfo() {
        return auftragInfo;
    }

    public void setAuftragInfo(@Nullable AuftragInfo auftragInfo) {
        if (Objects.equals(this.auftragInfo, auftragInfo)) return;
        AuftragInfo oldAuftragInfo = this.auftragInfo;
        this.auftragInfo = auftragInfo;
        if (oldAuftragInfo != null) oldAuftragInfo.setAuftrag(null);
        if (auftragInfo != null) auftragInfo.setAuftrag(this);
    }
}

@Entity
public class AuftragInfo {
    @OneToOne(fetch = FetchType.LAZY)
    private @Nullable Auftrag auftrag;

    public @Nullable Auftrag getAuftrag() {
        return auftrag;
    }

    public void setAuftrag(@Nullable Auftrag auftrag) {
        if (Objects.equals(this.auftrag, auftrag)) return;
        Auftrag oldAuftrag = this.auftrag;
        this.auftrag = auftrag;
        if (oldAuftrag != null) oldAuftrag.setAuftragInfo(null);
        if (auftrag != null) auftrag.setAuftragInfo(this);
    }
}

```

Anhang 3: Bidirektionale m-zu-n-Beziehung

```

@Entity
public class Auftrag {
    @ManyToMany
    private Set<Artikel> artikel = Collections.newSetFromMap(new
IdentityHashMap<>());

    public Collection<Artikel> getArtikel() {
        return Collections.unmodifiableSet(artikel);
    }

    public void addArtikel(Artikel artikel) {
        requireNonNull(artikel, "artikel == null!");
        artikel.addAuftrag(this);
        this.artikel.add(artikel);
    }

    public void removeArtikel(Artikel artikel) {
        requireNonNull(artikel, "artikel == null!");
        artikel.removeAuftrag(this);
        this.artikel.remove(artikel);
    }
}

@Entity
public class Artikel extends RecursionPrevention {
    @ManyToMany(mappedBy = "artikel")
    private List<Auftrag> auftraege = new ArrayList<>();

    public Collection<Auftrag> getAuftraege() {
        return Collections.unmodifiableList(auftraege);
    }

    public void addAuftrag(Auftrag auftrag) {
        requireNonNull(auftrag, "auftrag == null!");
        if (auftrag.getArtikel().contains(this)) return;
        runNonRecursive(() -> {
            auftrag.addArtikel(this);
            auftraege.add(auftrag);
        });
    }

    public void removeAuftrag(Auftrag auftrag) {
        requireNonNull(auftrag, "auftrag == null!");
        if (!auftrag.getArtikel().contains(this)) return;
        runNonRecursive(() -> {
            auftrag.removeArtikel(this);
            auftraege.remove(auftrag);
        });
    }
}

```

Anhang 4: Fragebogen der Umfrage

Umfrage für meine Bachelor Arbeit

mit dem Thema:

*Verbesserung der Wartbarkeit von automatisierten Tests für ORM-Anwendungen
durch Nutzung von Builder-Pattern und bidirektionalem Mapping*

Hinweis: Sie werden in der Bachelor Arbeit nicht namentlich benannt.

Haben Sie in der Sanacorp bereits in einem Projekt mit automatisierten Tests, aber ohne bidirektionales Mapping gearbeitet?

☐ Ja

☐ Nein

Haben Sie in der Sanacorp bereits in einem Projekt mit automatisierten Tests und mit bidirektionalem Mapping gearbeitet?

☐ Ja

☐ Nein

Waren Sie bereits an einem Projekt beteiligt, bei dem bidirektionales Mapping eingeführt wurde?

☐ Ja

☐ Nein

Vergleichen Sie die folgenden beiden Tests.

Mit bidirektionalem Mapping:

```
@Test
public void test_hatBtmBezogen() {
    // Given:
    Kunde underTest = some($Kunde())//
        .withAuftraege($ListOf(few(), $Auftrag())//
            .withPositionen($ListOf(several(), $Position())//
                .withArtikel($Artikel())//
                    .withBtm(false))))));

    // When:
    boolean actual = underTest.hatBtmBezogen();

    // Then:
    assertThat(actual).isFalse();
}
```

Ohne bidirektionales Mapping:

```
@Inject
private KundeService underTest;

@Test
public void test_hatBtmBezogen() {
    // Given:
    Kunde kunde = some($Kunde());
    List<Auftrag> auftraege = ListOf(few(), $Auftrag().withKunde(kunde));
    db().persist(//
        ListOf(few() * several(), $Position())//
            .withAuftrag($oneOf(auftraege))//
            .withArtikel($Artikel().withBtm(false))//
        )//
    );

    // When:
    boolean actual = underTest.hatBtmBezogen(kunde.getId());

    // Then:
    assertThat(actual).isFalse();
}
```

Welcher der beiden gezeigten Tests ist ihrer Meinung nach leichter zu verstehen?

☐ Der obere Test ☐ Der untere Test

Begründen Sie ihre Entscheidung:

Können Sie nach ihrer eigenen Einschätzung durch bidirektionales Mapping Tests durchschnittlich schneller oder langsamer schreiben?

- ☐ Deutlich schneller
- ☐ Etwas schneller
- ☐ Gleich schnell
- ☐ Etwas langsamer
- ☐ Deutlich langsamer

Wie beurteilen Sie die Nutzung von bidirektionalem Mapping im Testcode (gegenüber rein unidirektionalem Mapping)?

Wie beurteilen Sie die erhöhte Komplexität der JPA-Beans durch bidirektionales Mapping?

Wie beurteilen Sie die Wartbarkeit von Produktivcode, der bidirektionales Mapping aufruft (gegenüber Produktivcode, der ausschließlich unidirektionales Mapping aufruft und gegebenenfalls zusätzliche Queries absetzt)?

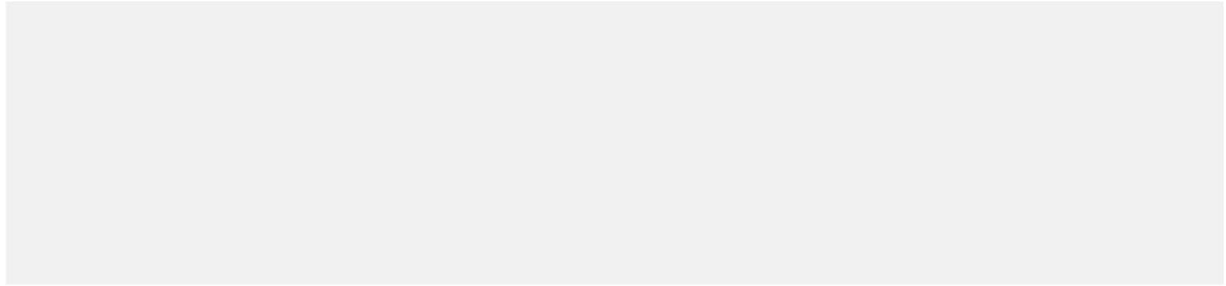
Wie beurteilen Sie die Änderungen der Software Architektur durch bidirektionales Mapping (insbesondere den zunehmenden Fokus auf Objektorientierung gegenüber Serviceorientierung)?

Wie beurteilen Sie die Performance von Produktivcode, der bidirektionales Mapping nutzt?

Was sind aus Ihrer Sicht weitere **Vorteile**, die sich durch den Einsatz von bidirektionalem Mapping gegenüber dem Einsatz von unidirektionalem Mapping ergeben?

Was sind aus Ihrer Sicht weitere **Nachteile**, die sich durch den Einsatz von bidirektionalem Mapping gegenüber dem Einsatz von unidirektionalem Mapping ergeben?

Wie beurteilen Sie den Aufwand für den Einsatz von bidirektionalem Mapping im Verhältnis zum Nutzen und begründen Sie?



Würden Sie in einem neuen Projekt bidirektionales Mapping einsetzen?

- ☐ Ja ☐ Nein ☐ Keine Angabe

Würden Sie in einem bestehenden Projekt mit unidirektionalem Mapping langfristig auf bidirektionales Mapping umstellen?

- ☐ Ja ☐ Nein ☐ Keine Angabe

Literaturverzeichnis

- *Cucumber* (2018): Gherkin Reference, <<https://docs.cucumber.io/gherkin/reference/#keywords>> (2018-09-09) [Zugriff 2018-11-02].
- *Eclipse* (BatchFetch, 2017): Java Persistence API (JPA) Extensions Reference for EclipseLink, Release 2.7, <http://www.eclipse.org/eclipselink/documentation/2.7/jpa/extensions/annotations_ref.htm#CHDCCIDA> (2017) [Zugriff 2018-11-02].
- *Eclipse* (setUseLazyInstantiationForIndirectCollection, 2017): CollectionMapping (EclipseLink 2.7.0, API Reference), <<https://www.eclipse.org/eclipselink/api/2.7/org/eclipse/persistence/mappings/CollectionMapping.html#setUseLazyInstantiationForIndirectCollection-java.lang.Boolean->>> (2017) [Zugriff 2018-10-22].
- *Eclipse* (useCollectionClass, 2017): CollectionMapping (EclipseLink 2.7.0, API Reference), <<https://www.eclipse.org/eclipselink/api/2.7/org/eclipse/persistence/mappings/CollectionMapping.html#useCollectionClass-java.lang.Class->>> (2017) [Zugriff 2018-10-22].
- *Eclipse* (IndirectList, 2018): IndirectList, <<https://github.com/eclipse-ee4j/eclipselink/blob/2.7.3/foundation/org.eclipse.persistence.core/src/org/eclipse/persistence/indirection/IndirectList.java#L683-L693>> (2018-07-03) [Zugriff 2018-11-01].
- *Eclipse* (IndirectSet, 2018): IndirectList, <<https://github.com/eclipse-ee4j/eclipselink/blob/2.7.3/foundation/org.eclipse.persistence.core/src/org/eclipse/persistence/indirection/IndirectSet.java#L285-L296>> (2018-07-03) [Zugriff 2018-11-01].
- *Elliott, James, O'Brien, Timothy M., Fowler, Ryan* (2008): Harnessing Hibernate: Step-by-step Guide to Java Persistence, Sebastopol: O'Reilly Media.
- *Hibernate* (BatchFetchStyle, 2018): BatchFetchStyle (Hibernate JavaDocs), <<https://docs.jboss.org/hibernate/orm/5.3/javadocs/org/hibernate/loader/BatchFetchStyle.html>> (2018-10-17) [Zugriff 2018-11-03].
- *Hibernate* (IN_CLAUSE_PARAMETER_PADDING, 2018): AvailableSettings (Hibernate JavaDocs), <https://docs.jboss.org/hibernate/orm/5.3/javadocs/org/hibernate/cfg/AvailableSettings.html#IN_CLAUSE_PARAMETER_PADDING> (2018-10-17) [Zugriff 2018-11-03].

- *Hibernate* (JavassistLazyInitializer, 2018): JavassistLazyInitializer, <<https://github.com/hibernate/hibernate-orm/blob/5.3.7/hibernate-core/src/main/java/org/hibernate/proxy/pojo/javassist/JavassistLazyInitializer.java#L91-L93>> (2018-07-09) [Zugriff 2018-10-30].
- *Hibernate* (PersistentList, 2018): PersistentList, <<https://github.com/hibernate/hibernate-orm/blob/5.3.7/hibernate-core/src/main/java/org/hibernate/collection/internal/PersistentList.java#L171-L206>> (2018-08-01) [Zugriff 2018-11-01].
- *Jungmann, Lukas, DeMichiel, Linda* (2017): Oracle JSR 338: Java Persistence API, Version 2.2 <https://download.oracle.com/otn-pub/jcp/persistence-2_2-mrel-spec/JavaPersistence.pdf> (2017-07-17) [Zugriff 2018-10-29].
- *Korel, Bogdan* (1990): Automated Software Test Data Generation, in: IEEE Transactions on Software Engineering, Vol. 16, No. 8, S. 870-879.
- *Meszaros, Gerard* (2007): xUnit Test Patterns. Refactoring Test Code, Boston: Pearson Education, Inc.
- *Müller-Hofmann, Frank, Hiller, Martin & Wanner, Gerhard* (2015): Programmierung von verteilten Systemen und Webanwendungen mit Java EE. Erste Schritte in der Java Enterprise Edition, Wiesbaden: Springer Vieweg, verfügbar unter: <https://link.springer.com/content/pdf/10.1007%2F978-3-658-10512-9.pdf>.
- *Myers, Glenford J.* (2004): The Art of Aoftware Testing, 2. Aufl., Hoboken, New Jersey: Jhon Wiley & Sons, Inc.
- *Niederwieser, Peter* (2018): Spock Primer, <http://spockframework.org/spock/docs/1.2/spock_primer.html#_blocks> (2018-09-23) [Zugriff 2018-11-02].
- *Oracle* (2018): Oracle Database SQL Language Reference, 18c Version 18.1, <<https://docs.oracle.com/en/database/oracle/oracle-database/18/sqlrf/sql-language-reference.pdf>> (2018-07) [Zugriff 2018-11-03].
- *Oracle* (Collection, 2018): Collection (Java SE 10 & JDK 10), <[https://docs.oracle.com/javase/10/docs/api/java/util/Collection.html#remove\(java.lang.Object\)](https://docs.oracle.com/javase/10/docs/api/java/util/Collection.html#remove(java.lang.Object))> [Zugriff 2018-10-22].
- *Pryce, Nat* (2007): Test Data Builders: an alternative to the Object Mother pattern, <<http://www.natpryce.com/articles/000714.html>> (2007-08-27) [Zugriff 2018-11-01].

- *Schuh, Peter, Punke, Stephanie* (2001): ObjectMother. Easing Test Object Creating in XP, XP Universe Conference, verfügbar unter: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.18.4710&rep=rep1&type=pdf>.
- *Mihalcea, Vlad, Ebersole, Steve, Boriero, Andrea, Morling, Gunnar, Badner, Gail, Cranford, Chris, Bernard, Emmanuel, Grinovero, Sanne, Meyer, Brett, Ferentschik, Hardy, King, Gavin, Bauer, Christian, Andersen, Max Rydahl, Maesen, Karel, Vansa, Radim, Jacomet, Louis* (2018): Hibernate ORM 5.3.7.Final User Guide, <https://docs.jboss.org/hibernate/orm/5.3/userguide/html_single/Hibernate_User_Guide.html> (2018-10-16) [Zugriff 2018-10-29].
- *Uffmann, Adrian* (HHH-12709, 2018): Non-optional, non-owning side of a bidirectional OneToOne Relation ignores FetchType.LAZY, <<https://hibernate.atlassian.net/browse/HHH-12709>> (2018-06-24) [Zugriff 2018-11-04].
- *Uffmann, Adrian* (HHH-12711, 2018): @BatchSize and hibernate.default_batch_fetch_size should affect eager loading of non-owning one-to-one relations, <<https://hibernate.atlassian.net/browse/HHH-12711>> (2018-06-24) [Zugriff 2018-11-04].
- *Webster, Jane & Watson, Richard* (2002): Analyzing the Past to Prepare For the Future: Writing a Literature Review, in: MIS Quarterly, Jg. 2002, Nr. 26 (2), S. Xiii-xxiii.
- *Wiklund, Kristian* (2015): Impediments for Automated Software Test Execution, Västerås: Mälardalen University, verfügbar unter: <http://mdh.diva-portal.org/smash/get/diva2:808786/FULLTEXT01.pdf>.

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig und ohne unerlaubte Hilfe angefertigt worden ist, insbesondere dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen sind, durch Zitate als solche gekennzeichnet habe. Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Weiterhin erkläre ich, dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde / Prüfungsstelle vorgelegen hat. Ich erkläre mich damit einverstanden, dass die Arbeit der Öffentlichkeit zugänglich gemacht wird. Ich erkläre mich damit einverstanden, dass die Digitalversion dieser Arbeit zwecks Plagiatsprüfung auf die Server externer Anbieter hoch geladen werden darf. Die Plagiatsprüfung stellt keine Zurverfügungstellung für die Öffentlichkeit dar.

München, den 10.11.2018: _____