Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik

Bachelorarbeit

Konzept zur effizienten Datenanalyse auf Basis feingranularer Änderungserkennung am Beispiel von MinD.banker

Autor:

Steffen Schulze

April, 2014

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake
M.Sc. David Broneske

Institut für Technische und Betriebliche Informationssysteme

Schulze, Steffen:
Konzept zur effizienten Datenanalyse auf Basis feingranularer Änderungserkennung am Beispiel von MinD.banker
Bachelorarbeit, Otto-von-Guericke-Universität Magdeburg, 2014.

Inhaltsangabe

Vor dem Hintergrund, ständig wachsender Datenmengen sind die Unternehmen gefordert, technische Fortschritte im Bereich der Datenanalyse zu erzielen. Mittels einer Datenanalyse soll aus den vorliegenden Daten verwertbare Informationen gewonnen werden um aus diesen Informationen die relevanten Entscheidungen abzuleiten. Aktuell wird dies durch Data-Warehouse-Technologien abgedeckt. Ein Problem, dass sich daraus ergibt, ist, dass in aktuellen Implementierungen die Datenanalyse ineffizient ist, da feingranulare fachliche Abhängigkeiten zwischen operativen Daten und Analysedaten nicht betrachtet werden.
In dieser Arbeit wird ein Konzept erstellt, um Datenänderungen von operativen Daten automatisch feingranular zu erkennen. Ausgehend von den geänderten Daten soll der Aktualisierungsbedarf der Analysedaten auf Basis formalisierter Abhängigkeiten zwischen Operativ- und Analysedaten ermittelt werden. Dieses Konzept wird im Rahmen einer prototypischen Implementierung am Beispiel von MinD.banker umgesetzt.

Inhaltsverzeichnis

Abbildungsverzeichnis **vii**

Tabellenverzeichnis **ix**

Quelltextverzeichnis **xi**

1 Datenanalyse als Schlüssel für effiziente Datenverarbeitung **1**

2 Grundlagen **5**
- 2.1 Methoden zur Erkennung von Datenänderungen 5
  - 2.1.1 Datenbank-Trigger 5
  - 2.1.2 Eventlistener in Hibernate 8
  - 2.1.3 Aspektorientierte Programmierung 11
  - 2.1.4 Vorteile und Nachteile der Ansätze 14
- 2.2 Modellgetriebene Softwareentwicklung 14
- 2.3 Zusammenfassung 17

3 Anforderungsanalyse **19**
- 3.1 Problemstellung 19
- 3.2 Anforderungen an eine effiziente Datenanalyse 22
- 3.3 Zusammenfassung 24

4 Konzept zur feingranularen Änderungserkennung **25**
- 4.1 Modellierung des Konzeptentwurfs 25
- 4.2 Abhängigkeiten zwischen den operativen Daten und den Analysedaten 27
- 4.3 Komponenten des Modells 29
  - 4.3.1 Konfigurationsdatei 31
  - 4.3.2 Abhängigkeitsdatei der Analysedaten 32
  - 4.3.3 Template 33
  - 4.3.4 Transformatoren 34
- 4.4 Zusammenfassung 36

5 Prototypische Implementierung am Beispiel von MinD.banker **37**
- 5.1 MinD.banker 37
- 5.2 Umsetzung 38
  - 5.2.1 Konfigurationsdatei 39
  - 5.2.2 Abhängigkeitsdatei 41
  - 5.2.3 Template 42

5.2.4 Transformatoren 44
5.2.5 Build-Prozess 49
5.3 Zusammenfassung 49

6 Auswertung der erfüllten Anforderungen 51
6.1 Auswertung der Anforderungen 51
6.1.1 Funktionale Anforderungen 51
6.1.2 Nicht-funktionale Anforderungen 55
6.2 Zusammenfassung 57

7 Zusammenfassung und Ausblick 59
7.1 Zusammenfassung 59
7.2 Ausblick 60

Literaturverzeichnis 63

Abbildungsverzeichnis

2.1 Rolle von Hibernate in einer Java-Anwendung [ML07] 9

2.2 Advice-Typen [Gol11] 13

4.1 Ablauf zur effizienten Datenanalyse 26

4.2 Abhängigkeitsmodell 28

4.3 Modell zur effizienten Datenanalyse 30

4.4 Vorgang zur Ermittelung der abhängigen Segmente 35

4.5 Prozess der Template-Generierung 36

Tabellenverzeichnis

2.1 Vor- und Nachteile der Methoden zur Erkennung von Datenänderungen 14

3.1 Anforderungen an effiziente Datenanalyse 24

6.1 Umsetzung der funktionalen Anforderungen 55
6.2 Testszenario mit und ohne Ermittlung des Aktualisierungsbedarfs . . 56
6.3 Umsetzung der nicht-funktionalen Anforderungen 57

Quelltextverzeichnis

2.1 Syntax SQL:2003 Trigger [SSH10] 6
2.2 Beispiel für einen LoadEventListener [KBA+10] 10
2.3 Eventlistener-Registrierung in XML-Konfigurationsdatei [KBA+10] . 10
2.4 Programmatische EventListener-Registrierung [KBA+10] 11
2.5 AspectJ [ABKS13] 13
5.1 Konfigurationsdatei für ein Account-Objekt 40
5.2 Abhängigkeitsdatei 41
5.3 DirtyConfiguration 43
5.4 Template-Datei 43
5.5 generierte Konfigurationsdatei mit allen abhängigen Analysesegmenten 45
5.6 Implementierungsklasse 46
5.7 Aspekt für ein Account-Objekt 48

1. Datenanalyse als Schlüssel für effiziente Datenverarbeitung

Vor dem Hintergrund der heutigen Datenmengen, der Wachstumsprognosen und der zunehmenden Wichtigkeit von Daten, stellt die Datenanalyse für die Unternehmen in den nächsten Jahren eine große Herausforderung dar. Dabei ist es unumgänglich, aus den vorhandenen Daten die wichtigsten Informationen für relevante Entscheidungen zu extrahieren, denn erst mit diesen verdichteten Informationen lässt sich Geld verdienen. Dies ist der Grund, weshalb diese Informationen so wertvoll für die Unternehmen sind.
Auch im Bankwesen ist es notwendig, große operative Datenmengen aus einer Vielzahl von Datenquellen zu sammeln und aufzubereiten, um sich mit seinen Produkten klar gegenüber Wettbewerbern zu positionieren. Je genauer die zur Verfügung stehenden Informationen sind, desto besser können sie zur Unterstützung von Entscheidungen verwendet werden und tragen somit zum Erfolg des Unternehmens bei. Mit MinD.banker[1] (= Managementinstrumente & Dialog) hat die GAD[2], ein genossenschaftlicher Verbund von Volks- und Raiffeisenbanken, eine bislang einzigartige Software im Portfolio, die den Banken eine strukturierte Analyse und ganzheitliche Beratung ermöglicht [GAD08].

Aktuell wird die Datenanalyse in heutigen Anwendungen durch Data-Warehouse-Technologien abgedeckt. Ein Data Warehouse wird als Sammlung von Technologien zur Unterstützung von Entscheidungsprozessen beschrieben [KSS12]. Dabei werden die Daten entsprechend des ETL-Prozesses (Extraktion-Transformation-Laden) aus den verschiedenen Datenquellen entnommen, mit Hilfe der Transformation bereinigt und in ein einheitliches Format umgewandelt, um danach in das Data Warehouse geladen zu werden. Diese Daten werden vor allem für die Analyse, Planung und Informationsbereitstellung verwendet. Aufbauend auf dieser Datenbasis können mit Hilfe von verschiedenen Technologien aus dem Bereich Business Intelligence Analysen und Reporte generiert werden, welche den Entscheidungsprozess unterstützen

[1] http://min-d.de/index.php?cid=216
[2] http://www.gad.de

können [KSS12]. Ein Problem, das sich bei der Extraktion ergibt, ist, dass diese Vorgänge zeitintensiv sind und daher meist zu festgelegten Zeitpunkten und für fest definierte Datenbestände (statische Extraktion) durchführt werden [GRC09]. Dadurch wird eine Analyse auf aktuellen Daten erschwert, da die Aktualität der Daten nicht gewährleistet werden kann. Die Verwendung von inkrementeller Extraktion von operativen Daten zur Aktualisierung von Analysedaten erscheint hierfür geeignet. Die inkrementelle Extraktion sammelt nur die Datenänderungen ein, die zwischen dem aktuellen und letztem Extraktionsprozess durchgeführt wurden. Für den Einsatz der inkrementellen Extraktion wäre es daher wünschenswert, dass die fachlichen Abhängigkeiten zwischen den operativen Daten und den Analysedaten definiert werden. Nur wenn diese Abhängigkeiten bekannt sind, können die Analysedaten effizient aktualisiert werden.

Die feingranularen fachlichen Abhängigkeiten zwischen operativen Daten und Analysedaten sind in der Bankensoftware MinD.banker nicht vorhanden und weshalb die Datenanalyse ineffizient ist. So werden in der aktuellen Implementierung die Analysedaten immer aktualisiert, obwohl kein Aktualisierungsbedarf vorhanden ist, da zwischen den geänderten operativen Daten und den Analysedaten keine Abhängigkeit modelliert sind. Zudem ist die Erkennung von Änderungen, auf Basis der Abhängigkeiten zwischen den operativen und Analysedaten, abhängig vom individuellen Wissen der Entwickler. Dies bedeutet, dass der Entwickler ganz genau wissen muss, welche Analysedaten nach dem Ändern der operativen Daten aktualisiert werden sollen.
Die fachlichen Abhängigkeiten sollten feingranular modelliert werden, um einen möglichst detaillierten Überblick über Zusammenhänge zwischen Operativ- und Analysedaten zu erhalten. Anhand dieses Abhängigkeitsmodells kann ermittelt werden, ob ein Bedarf zur Aktualisierung der Analysedaten existiert und in diesem Fall eine Aktualisierung der Analysedaten durchzuführen ist. Weitere Vorteile des Abhängigkeitsmodells sind eine einfachere Erweiter- und Wartbarkeit sowie automatische Testbarkeit. Des Weiteren wird die Transparenz gegenüber den Entwicklern erhöht, um auch anderen einen Einblick in das Zusammenspiel zwischen operativen und Analysedaten zu bieten.

Zielstellung der Arbeit

Mit der vorliegenden Bachelorarbeit wird vorrangig das Ziel verfolgt, ein Konzept zu erstellen, um Datenänderungen von operativen Daten automatisch feingranular zu erkennen. Unter Verwendung der geänderten Daten soll der Aktualisierungsbedarf der Analysedaten auf Basis formalisierter Abhängigkeiten zwischen Operativ- und Analysedaten ermittelt werden. Die Machbarkeit des Konzepts wird anhand eines Prototypen nachgewiesen.

Die Forschungsfrage der vorliegenden Arbeit lautet demnach:

Wie können Änderungen an den operativen Daten unabhängig vom individuellen Wissen automatisiert erkannt werden, um den Aktualisierungsbedarf an aufbereiteten abhängigen Analysedaten festzustellen, sodass eine effiziente Datenanalyse gewährleistet ist?

Um das Konzept herleiten zu können, wird die Untersuchung der Forschungsfrage in mehrere Teilprobleme aufgeteilt. Die Untersuchungsfragen werden im Laufe der Arbeit betrachtet und näher erläutert.

Dabei sind folgende Untersuchungsfragen zu beantworten:

1. Welche Methoden existieren zur automatischen und transparenten Erkennung von feingranularen Datenänderungen?
2. Wie lassen sich Abhängigkeiten zwischen operativen Daten und Analysedaten formal beschreiben, um den Aktualisierungsbedarf der Analysedaten zu bestimmen?
3. Welche Anforderungen bezüglich Effizienz der Ermittlung von Aktualisierungsbedarf bestehen?
4. Wie können die Erkenntnisse zur Erkennung von Datenänderungen, ein Abhängigkeitsmodell und die Anforderungen an effiziente Datenanalyse umgesetzt werden?
5. Wie lassen sich die Inhalte des Konzepts in MinD.banker umsetzen?

Gliederung der Arbeit

Im Kapitel 2 werden die benötigten Grundlagen für diese Arbeit beschrieben. Dazu wird im ersten Schritt der Stand der Technik analysiert. Dies soll einen Überblick über die Methoden der softwaretechnischen als auch datenbankbezogenen Erkennung von Datenänderungen geben. Darüber hinaus wird ein Einblick in die modellgetriebene Softwareentwicklung gegeben. Anhand der gegebenen Untersuchungsfragen werden im Kapitel Kapitel 3 die Anforderungen an die effiziente Datenanalyse spezifiziert.
Um den Aktualisierungsbedarf an den Analysedaten zu bestimmen, wird formal ein Abhängigkeitsmodell erstellt. Die Erkenntnisse münden in einem Konzept. Im Kapitel 4 wird der Konzeptentwurf in Form eines Modells vorgestellt. Die Komponenten des Modells werden in einer prototypischen Implementierung im Kapitel 5 am Beispiel von MinD.banker umgesetzt. Der hervorgebrachte Prototyp wird anhand der aufgestellten Anforderungen im Kapitel 6 evaluiert. Dabei wird erläutert, inwieweit die einzelnen Anforderungen umgesetzt wurden und welche noch offen sind. Zum Abschluss werden die Ergebnisse in dem Kapitel 7 zusammengefasst und wir schließen mit einem Ausblick.

2. Grundlagen

In diesem Kapitel werden die Grundlagen, die für das Verständnis des später zu entwickelnden Konzeptes wichtig sind, erläutert. Der Abschnitt 2.1 veranschaulicht die vorhandenen Methoden zur Erkennung von Datenänderungen, um den Aktualisierungsbedarf der Analysedaten automatisch bestimmen zu können. Im Abschnitt 2.2 wird gezeigt, wie sich diese Methoden mit Hilfe modellgetriebener Softwareentwicklung in Programmcode umsetzen lassen.

2.1 Methoden zur Erkennung von Datenänderungen

In diesem Unterkapital werden die verschiedenen Methoden zur Erkennung von Datenänderungen vorgestellt. Dabei wird nicht nur auf Methoden aus der Softwaretechnik, aspektorientierte Programmierung und Eventlistener in Hibernate, sondern auch auf Trigger aus dem Bereich der Datenbanken eingegangen.

2.1.1 Datenbank-Trigger

Der Begriff *"Trigger"* (zu Deutsch *Auslöser*) wird in vielen Bereichen wie z.B. der Medizin, der Psychologie oder auch der Informatik verwendet. Die Bedeutung ist gleichwohl, in welchen Gebieten der Begriff verwendet wird, immer dieselbe. In der Informatik spielen Trigger im Datenbankbereich eine große Bedeutung. Unter einem Datenbank-Trigger, im Weiteren von Triggern gesprochen, wird eine Anweisung bzw. Prozedur, die bei Eintreten eines bestimmten Ereignisses automatisch vom Datenbankmanagementsystem ausgeführt wird, verstanden [SSH10]. Dabei nimmt das Datenbankmanagementsystem, eine Software zur Verwaltung von Datenbanken, die Schnittstelle zwischen den Anwendungen und der Datenbank ein. Das Datenbankmanagementsystem kontrolliert alle lesenden und schreibenden Zugriffe auf die Datenbank und ist u.a. für die Datensicherung, Transaktionen und Integritätssicherung zuständig.
Datenbank-Trigger sind weit verbreitet und in den meisten kommerziellen Datenbanksystemen wie Oracle, DB2 oder Microsoft SQL Server verfügbar. Überdies

spielen sie in Open-Source-Datenbanken wie MySQL oder Firebird ebenfalls eine wichtige Rolle. Im Rahmen von SQL:1999 [ISO99] wurden sie erstmals in den SQL-Standard aufgenommen. Datenbank-Trigger bieten zahlreiche Einsatzmöglichkeiten u.a. zur Protokollierung von Datenbankänderungen (Auditing), der Validierung von geänderten Daten oder der Überwachung von Integritätsbedingungen.
Eine Datenbank-Trigger-Anweisung folgt immer den ECA-Regeln (Event, Condition, Action). Wenn ein auslösendes Ereignis (Event) eintritt und eine Bedingung (Condition) dafür erfüllt ist, wird eine Aktion (Action) ausgeführt. Im Allgemeinen ist das auslösende Ereignis ein Einfügen, Löschen oder Ändern von Tupeln einer Tabelle. Die Bedingungen können individuell festgelegt werden. Die Aktionen bestehen grundsätzlich aus SQL-Anweisung. So können z.B. SQL-Anweisungen, welche zu Fehlern geführt haben, abgebrochen werden und alle vorgenommenen Änderungen rückgängig gemacht werden oder entsprechend Fehlermeldungen ausgeben werden. Da Datenbank-Trigger selbst Datenänderungen durchführen können, kann diese Ausführung weitere Datenbank-Trigger auslösen.

Definition eines Trigger

Die Definition eines Trigger wird nachfolgend anhand der Syntax für eine `create trigger`-Anweisung im SQL:2003-Standard dargestellt. Die Hersteller von kommerziellen Datenbanksystemen passen die Syntax individuell ihrem System an, sodass einige Statements vom Standard abweichen können.

Listing 2.1: Syntax SQL:2003 Trigger [SSH10]

```
create trigger <Name>
after | before <Ereignis>
on <Relation>
[referencing Transitionsvariablen/-tabellen]
[for each row | for each statement]
[when <Bedingung>]
begin atomic < SQL-Anweisungen > end
```

Jede Triggerdefinition beginnt mit den Worten `create trigger` gefolgt von einem eindeutigen Namen des Triggers innerhalb des Datenbankschemas. Unter diesem Namen wird der Trigger in der Datenbank gespeichert, um den Trigger wieder löschen zu können. Ein Trigger wird eindeutig einer Tabelle zugeordnet. Weiterhin ist zu beachten, dass nur ein Datenbank-Trigger pro Aktivierungszeitpunkt, Ereignis und Tabelle zuzulassen ist.
Der Aktivierungszeitpunkt wird mit `before` und `after` festgelegt. Dieser entscheidet, ob vor oder nach der Aktivierung des Ereignisses die Aktion des Datenbank-Triggers ausgelöst wird. Folglich hat der Aktivierungszeitpunkt Auswirkungen darauf, in welchem Kontext der Trigger in Anspruch genommen werden sollte. Mit den `before`-Triggern kann geprüft werden, ob die gewünschte Änderung zugelassen ist. Wenn die Überprüfung negativ ausfällt, kann eine Fehlermeldung angezeigt werden und die Änderung verworfen werden. `after`-Trigger werden bevorzugt zur Aktualisierung der Werte des neu anlegten, aktualisierten oder gelöschten Tupels verwendet. Um auf den alten oder neuen Zustand des veränderten Tupels bzw. Tabelle zugreifen zu können, steht die optionale Referenz-Klausel (`referencing`) mit den Transitionsvariablen und -tabellen zur Verfügung. Die Transitionstabellen (`old table`

`as [Variablenname]`, `new table as [Variablenname]`) ermöglichen während der Ausführung den Zugriff auf den vorherigen und den neuen Zustand einer Tabelle. Um auf den neuen oder alten Wert eines Attributs des veränderten Tupels zugreifen zu können, wird das Schlüsselwort `new as [Variablenname]` oder `old as [Variablenname]` verwendet [Cha99]. Mit diesen Transitionsvariablen und -tabellen können die Veränderungen der Werte eines geänderten Tupels nachvollzogen werden.

Als auslösendes Triggerereignis stehen die `insert`-, `update`- oder `delete`-Anweisung auf den Attributen einer Tabelle zur Verfügung. Das Ereignis gilt nur für die in der `on`-Klausel spezifizierten Tabelle. Die SQL-Anweisung, die das Trigger-Ereignis ausgelöst hat, kann eine oder mehrere Tupel hinzufügen, ändern oder löschen. Die Granularität des Triggers, ob der Trigger für alle betroffenen Tupel ausgeführt oder nur einmal für die gesamte Anweisung aktiviert wird, kann mit den Anweisungen `for each row` (Zeilentrigger) und `for each statement` (Anweisungstrigger) festgelegt werden. Falls keine Tupel durch das auslösende Ereignis verändert werden, wird der Zeilentrigger nicht ausgeführt. Aus diesem Grund sollte der Zeilentrigger nur eingesetzt werden, wenn sichergestellt werden kann, dass die Änderung mindestens ein Tupel betrifft, was bei einer `update`-Anweisung der Fall sein kann. Im Gegensatz dazu wird der Anweisungstrigger einmal pro Anweisung ausgeführt, unabhängig davon, wieviele Tupel betroffen sind. Bei Anweisungstriggern stehen nur die Transitionstabellen zur Verfügung, bei den Zeilentriggern dagegen die Transitionstabellen, die -variablen sowie eine `when`-Klausel. Diese Klausel ermöglicht eine optionale Bedingung anzugeben, die sobald sie angegeben wird, erfüllt werden muss, damit die Triggeraktion ausgeführt werden kann. Innerhalb der `when`-Klausel sind alle Bedingungen erlaubt, die auch in `where`-Klauseln von `select`-Anfragen zugelassen sind. Sobald die Bedingungen des Triggers erfüllt sind, wird der Aktionsblock ausgeführt. Der Aktionsblock beinhaltet einzelne oder mehrere SQL-Anweisungen, die von einen `begin atomic` und `end` umschlossen werden. Innerhalb eines Aktionsblocks sind keine Transaktionsanweisungen erlaubt. Falls bei Ausführung des Triggers, sei es im Bedingungs- oder Aktionsblock, Fehler auftreten, werden alle Datenänderungen rückgängig gemacht und eine Fehlermeldung ausgegeben [Kle11].

Probleme bei der Verwendung von Triggern

Wenn in einem DBMS sehr viele Trigger eingesetzt werden, kann sich dies negativ auf die Bearbeitungszeit auswirken, da z.B. ein Trigger mit `for each row` für jedes Tupel abgearbeitet werden muss. Ein weiteres Problem, das sich dadurch ergibt, ist, dass durch eine Triggeraktion ein neuer Trigger aktiviert werden kann. Daraus können sich Ketten von Triggeraufrufen bilden, die schrittweise abgearbeitet werden müssen, was letztendlich zu Endlosschleifen führen kann. Zur Vermeidung dieses Problems werden die Tabellen, auf denen gerade ein Trigger läuft, gesperrt. So wird bei erneuten Triggerausführung die gesamte Aktion mit einer Fehlermeldung abgebrochen [Kle11].
Des Weiteren kann der Einsatz von fehlerhaften Triggern zur einer Beschädigung oder Zerstörung von Daten führen. Aus diesem Grund sollte bei Verwendung eines Triggers genau geprüft werden, ob nur die von der Datenänderung betroffenen Daten geändert werden. Im Fehlerfall ist die Suche nach dem Auslöser des Fehlers schwierig, da das Debuggen eines Triggers nicht unterstützt wird. Die Vorgehensweise in

diesem Fall ist Trail-And-Error, bei dem solange versucht wird bis der Fehler gefunden ist, dies kann aber viel Zeit in Anspruch nehmen.

Zusammengefasst sind die wichtigsten Klauseln, die zur Definition eines Triggers benötigt werden, folgende:

- nach dem Aktivierungszeitpunkt (`before` oder `after`)
- nach dem auslösenden Ereignis (`insert`, `update`, `delete`)
- nach Granularität (`row` oder `statement`)

Abschließend lässt sich feststellen, dass Trigger ein mächtiges Werkzeug im Kontext von Datenbanken sind. Die Aufgabe eines Triggers besteht hauptsächlich darin, beim Eintreten bestimmter Ereignisse (Insert, Update, Delete) selbstdefinierte Aktionen automatisch auszuführen. Dazu gehören neben der automatischen Prüfung von Werten und anschließender Nachbearbeitung dieser Werte, auch die Überprüfung der Einhaltung verschiedener Konsistenzregeln oder die Plausibilitätsprüfungen. Dadurch können die Trigger dem Entwickler sehr viel Arbeit abnehmen.

2.1.2 Eventlistener in Hibernate

Zusätzlich zu Datenbank-Triggern bietet das objektrelationale Mapping Methoden und Technologien an, um Objekte von objektorientierten Anwendungen in relationalen Datenbanken zu speichern. Die Hauptaufgabe des objektrelationalen Mappings ist die Zuordnung von Java-Objekten (im `Java-Bereich Plain Old Java Object` genannt) zu relationalen Entities in einer Datenbank [O'N08]. Somit wird eine objektorientierte Sicht auf Tabellen und Beziehungen der Datenbanken ermöglicht. Die lesenden und schreibenden Zugriffe auf die Datenbank werden vom objektrelationalen Mapping in Abhängigkeit des SQL-Dialekt der verwendeten Datenbanken generiert, sodass die Entwickler anstatt SQL-Anweisungen nur noch mit Objekten agieren. Eines der bekanntesten und weitverbreiteten objektrelationalen Mapping-Framework für Java ist Hibernate[1], ein Open-Source-Projekt für Java. Im Jahre 2002 wurden die ersten Ansätze des objektrelationalen Mapping in Hibernate realisiert. Der Einsatz von Hibernate ist in nahezu allen aktuellen relationalen Datenbanksystemen realisierbar. Zusätzlich stellt Hibernate Caching, Session- und Transaktion-Management bereit [ML07, O'N08, Leo13].
Um Hibernate den Zugriff auf die Datenbank zu erlauben, werden alle benötigten Informationen, wie z. B. Datenbank-Dialekt, in einer Konfigurationsdatei zusammengefasst, die bei der Instanziierung der Klasse SessionFactory von Hibernate geladen wird. Für jeden Zugriff auf eine Datenbank wird eine eigene Hibernate-Session verwendet. In Hibernate stehen mit Hibernate Query Language, ähnliche Syntax wie SQL, SQL-Statements oder der Hibernate Criteria-API verschiedene Methoden zur Datenbankabfrage zur Verfügung. Diese Abfragen werden mit Hilfe von Java Data Base Connectivity in den SQL-Dialekt der verwendeten Datenbank übersetzt [ML07, Leo13].

[1] http://www.hibernate.org/

Die Informationen, welche Objekte welchen Tabellen zugeordnet sind, wird in Mapping-Dateien abgelegt. Für jedes Plain Old Java Object, das einer Datenbanktabelle zugeordnet werden soll, wird eine selbige Mapping-Datei erstellt. Eine weitere Möglichkeit, wäre das Mapping unter Verwendung von Java-Annotationen zu realisieren. Die Abbildung 2.1 zeigt die Beziehung von Hibernate zwischen dem Clientcode und der Datenbank.

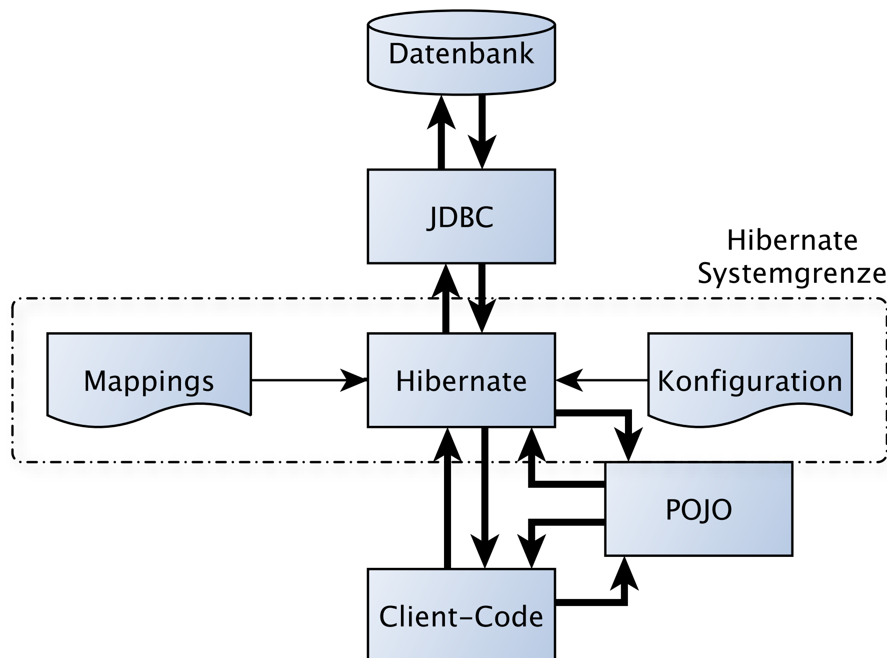

Abbildung 2.1: Rolle von Hibernate in einer Java-Anwendung [ML07]

Konzept der Eventlistener

Das Konzept der Eventlistener beruht auf dem Observer Pattern [SI10], bei dem die Veränderung eines Objektes (Ereignis) an das abhängige Objekt weitergeleitet wird oder dieses lediglich benachrichtigt wird. Mit den Eventlistener können die Ereignisse abgefangen werden, um danach bestimme Aktionen auszuführen. Dafür ist es notwendig, dass die Eventlistener für das Ereignis registriert werden.
Beim Umstieg auf Hibernate 3 wurde der Hibernate-Kern auf einem Modell von Ereignissen und Listenern umgestellt [KBA+10]. Jede Aktion, u.a. Laden, Speichern oder Löschen von Objekten, die innerhalb einer Hibernation-Session ausgeführt wird, erzeugt ein entsprechendes Ereignis. Diese Ereignisse können durch eigene Eventlistener behandelt werden. Eine Session stellt die Schnittstelle zwischen einer Java Applikation und Hibernate dar. Der Lebenszyklus einer Session wird durch den Beginn und das Ende einer Transaktion begrenzt [KBA+10].
In Hibernate existiert für jedes eintretende Ereignis sowohl ein Interface als auch eine Standard-Implementierung, den Defaultlistener. Zur Verwendung eines eigenen Eventlistener ist es notwendig, dass dieser entweder das Interface implementiert oder von einem Defaultlistener erbt. Es ist auch denkbar, einen Defaultlistener zu

ersetzen. Dabei muss beachtet werden, dass der neue Defaultlistener die entsprechende Funktionalität abdeckt, die sonst von Hibernate automatisch übernommen wird [Sip09]. Eine Übersicht aller verfügbaren Interfaces und Defaultlistener ist im Package `org.hibernate.event` zu finden [KBA+10].

Beispielcode eines LoadEventListener

Im folgenden Codebeispiel wird ein LoadEventListener erzeugt, der überprüft, ob die betroffene Entität geladen werden darf. Um auf LoadEvents zu reagieren, implementiert der `MyLoadListener` das Interface `LoadEventListener`. Mit der `isAuthorized()`-Methode wird kontrolliert, ob die benötigten Rechte für das Laden dieser Entität vorhanden sind. Wenn die Authorisierung erfolgreich bestätigt wurde, wird die Entität geladen, ansonsten wird eine `MySecurityException` geworfen.

```
public class MyLoadListener implements LoadEventListener
{
    // single method defined by the LoadEventListener interface
    public void onLoad(LoadEvent event,
        LoadEventListener.LoadType loadType) throws HibernateException
    {
        if (!MySecurity.isAuthorized(event.getEntityClassName(),
                                     event.getEntityId()))
        {
            throw MySecurityException("Unauthorized access");
        }
    }
}
```

Listing 2.2: Beispiel für einen LoadEventListener [KBA+10]

Die erforderliche Registrierung des `MyLoadListener` kann entweder programmatisch am `Configuration`-Objekt oder deklarativ in der Hibernate-XML-Konfigurationsdatei erfolgen. Im Listing 2.3 wird gezeigt, wie der Konfigurationseintrag in der Hibernate XML-Konfigurationsdatei verfasst werden muss, um den gezeigten `MyLoadListener` zu registrieren. Dazu muss der vollständige Klassenpfad zu den Eventlistener-Klassen angeben werden.

```
<hibernate-configuration>
  <session-factory>
    . . .
    <event type="load">
        <listener class="com.eg.MyLoadListener"/>
    </event>
    . . .
  </session-factory>
</hibernate-configuration>
```

Listing 2.3: Eventlistener-Registrierung in XML-Konfigurationsdatei [KBA+10]

```
Configuration cfg = new Configuration();
LoadEventListener[] stack = {new MyLoadListener()};
cfg.EventListeners().setLoadEventListeners(stack);
```

Listing 2.4: Programmatische EventListener-Registrierung [KBA+10]

Listing 2.4 zeigt die programmatische Variante, in der `MyLoadListener` an dem `Configuration`-Objekt registriert wird.

Das Hibernate Eventsystem bietet mit Hilfe von Eventlistener ein breites Spektrum an Aktionen an, um eintretende Ereignisse zu behandeln. Dabei werden die Ereignisse nur an die Eventlistener weitergeleitet, die sich auch für dieses Event registriert haben. Da die Eventlistener an einer zentralen Stelle registriert werden müssen, erlaubt es der Eventlistener leider nicht, nur auf Ereignisse von bestimmten Objekten zu reagieren. Um dies zu realisieren, muss der Eventlistener manuell erweitert werden. Dazu ist es notwendig vorab eine Überprüfung auf die bestimmten Objekte in der Event-Methode des Eventlistener zu veranlassen. Sobald sich herausstellt, dass es sich um eine Instanz des gewünschten Objektes handelt, wird der Code der Event-Methode weiter ausgeführt. Andernfalls wird das Event für das Objekt nicht weiter betrachtet.

2.1.3 Aspektorientierte Programmierung

Die aspektorientierte Programmierung ist aus der Idee entstanden, **Crosscutting Concerns** (querschneidene Belange) modularisieren zu können. In der Literatur wird ein querschneidener Belang als eine Funktionalität, die sich über verschiedene Module erstreckt, beschrieben. Diese Funktionalität lässt sich nicht eindeutig einem Modul zuordnen - der einzufügender Code würde sich über die gesamte Anwendung verteilen [Mer07]. Somit verhindern sie eine saubere Modularisierung und erschweren die Wartbarkeit, Lesbarkeit und Wiederverwendbarkeit [BK04]. Typische Anwendungsbeispiele für querschneidene Belange sind Logging, Tracing und Transaktionsverwaltung. Um diese querschneidene Belange modularisieren zu können, wird der objektorientierte Ansatz um Aspekte erweitert. Mit Unterstützung von Aspekten wird versucht, querschneidene Belange zentral an einer Stelle zu definieren und mit Hilfe des sogenannten Weavers an die benötigten Codestellen des Moduls einzuweben. Dies geschieht entweder im Compiler (Compile-time weaving) oder wird zur Laufzeit durch den Classloader (Load-time weaving) durchgeführt. Die aspektorientierte Programmierung wurde 1997 vom Team um Gregor Kiczales in den PARC-Labors von Xerox entwickelt [KLM+97]. Als am weitesten verbreitete aspektorientierte Programmierungssprache gilt heute AspectJ[2], das ebenfalls von Xerox PARC entwickelt wurde und gegenwärtig ein Teil der Eclipse Foundation ist. Mittlerweile gibt es jedoch auch für andere Programmiersprachen wie Python, C++ oder PHP entsprechende aspektorientierte Implementierungen.
AspectJ ist eine AOP-Erweiterung für Java, bei dem die Aspekte als modulare Einheit ähnlich wie Klassen implementiert werden. Es kommen zusätzlich eine Reihe neuer Sprachkonstrukte hinzu [KHH+01, EFB01]. Der Basiscode wird weiterhin in Java implementiert.

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

In AspectJ kann ein Aspekt (Schlüsselwort `aspect`) [ABKS13]

- Klassenhierarchien manipulieren
- Methoden und Felder zu einer Klasse hinzufügen
- Methoden mit zusätzlichem Code erweitern
- Ereignisse wie Methodenaufrufe oder Feldzugriffe abfangen und zusätzlichen oder alternativen Code ausführen

Komponenten eines Aspektes

Die wichtigsten Bestandteile eines Aspektes sind Pointcuts mit Join Points, Advices, Introductions und Compile-Time Declarations [Gol11]. Auf die Konstrukte Pointcuts, Join Points und Advices wird im Folgenden genauer eingegangen.

Alle Stellen im Code, an denen der Aspekt eingewebt werden könnte, werden Join Points genannt. Ein Join Point ist nichts anderes als ein eindeutig identifizierbarer Punkt im Programmfluss, wie z.B. der Aufruf der `print()`-Methode aus der Klasse `Edge` aus dem Beipielcode 2.5. Im Join-Point-Modell [CJR06] sind alle verfügbaren Join-Point-Typen aufgelistet. Zu den Wichtigsten zählen Aufruf (`call`) oder Ausführung (`execution`) von Methoden oder Konstruktoren, die Initialisierung von Klassen und Objekten und die Behandlung von Exceptions, in denen der `catch`-Block ausgeführt wird. Die Join Points sind schon im zu verfeinernden Code vorhanden, so dass die Entwickler für diese Join Points treffende Pointcuts definieren können. Pointcuts gehören zu den Programmierkonstrukten und können daher vom Entwickler definiert werden. Sie beschreiben, unter welchen Bedingungen – z. B. Parameter eines Methodenaufrufs – Join Points zu einem Pointcut gehören [Gol11]. Ein Pointcut setzt sich aus einer Menge von Join Points zusammen. Join Points können innerhalb eines Pointcuts beliebig mit den logischen Operatoren &&, || und ! kombiniert werden. Je genauer die Join-Point-Menge beschränkt wird, desto konkreter kann der Kontext, in dem ein solcher Join Point ausgeführt wird, bestimmt werden. Dabei ist der Zugriff auf Kontextinformationen, wie z.B. Quell- bzw. Zielobjekt oder Methodenparametern, notwendig für die Umsetzung des Aspekts [Mer07]. Wenn mehrere Aspekte zu einem Pointcut passen, wird durch Regeln bestimmt, in welcher Reihenfolge sie ausgewertet werden [Gol11]. AspectJ stellt zwei Arten von Pointcuts bereit: *benannt*, um in einem oder mehreren Advices angesprochen zu werden, und *anonym*, somit nur am Ort der Verwendung bekannt, bereit.
Ein Advice definiert die durchzuführenden Aktionen bei Erreichen eines Join Points, der durch einen angegebenen Pointcut ausgewählt wurde. In der Abbildung 2.2 werden drei mögliche Statements, die den Advice-Code unterschiedlich einflechten, dargestellt.

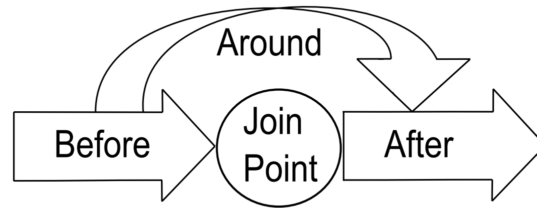

Abbildung 2.2: Advice-Typen [Gol11]

Beim `before`-Advice wird der Code vor dem jeweiligen Join Point ausgeführt. Falls im Advice-Body eine Exception geworfen wird, so wird der Join Point nicht mehr ausgeführt. Der `after`-Advice wird im Gegensatz zum `before`-Advice erst nach dem Join Point abgearbeitet. Bei Verwendung des `around`-Advices wird der Code des Advice-Body anstelle des Join Points ausgeführt. Dieser Advice-Typ wird verwendet, um den Join Point zu überspringen oder ihn mit anderen Parametern aufzurufen. Mit Hilfe des `proceed(..)`-Befehls innerhalb des `around`-Advice ist es möglich den ursprünglichen Join Point auszuführen [Gol11].

Beispielcode einer aspektorientierten Programmierung

Der folgende Beispielcode zeigt einen Aspekt, innerhalb dessen ein benannter Pointcut und ein `after`-Advice definiert wird. Der zusammengesetzte Pointcut *printExecution* selektiert alle Ausführungen der `print`-Methode der Klasse `Edge`. In Kombination mit dem `this(edge)`-Aufruf wird der Kontext des Join Point bestimmt. In diesem Fall, muss das `edge`-Objekt, eine Instanz der Klasse `Edge` sein. Der `after`-Advice gibt nach allen Aufrufen dieser Methode das Gewicht des auslösenden `edge`-Objekts auf der Systemkonsole aus. Wie das Beispiel zeigt, kann in einem Advice auch auf die Parameter des Pointcut zugegriffen werden.

```
aspect Weighted                                  \\ Aspektdeklaration
{
    ...
    pointcut printExecution(Edge edge) :         \\ Pointcutdeklaration
        execution(void Edge.print()) && this(edge);

    after(Edge edge) : printExecution(edge)      \\ Advice
    {
        System.out.print("weight " + edge.weight);
    }
}
```

Listing 2.5: AspectJ [ABKS13]

Die aspektorientierte Programmierung schafft nicht nur die Möglichkeit, ein sauberes Software-Design zu erreichen, sondern bietet auch Methoden an, um ein existierendes Programm zu erweitern, ohne dabei den eigentlichen Code anzupassen [Gol11]. Dadurch kann sich das Debugging des Programms oder einzelner Programmteile erschweren, da der gewobene Aspektcode nicht im Quellcode zu lokalisieren ist.

2.1.4 Vorteile und Nachteile der Ansätze

In der nachfolgenden Tabelle werden die Vor- und Nachteile der Methoden zur Erkennung von Datenänderungen beschrieben. Dies soll eine Übersicht über die beschriebenen Verfahren geben und uns im Verlauf der Arbeit helfen, ihre Anwendbarkeit für unsere Problemstellung zu begründen.

Methode	**Vorteile**	**Nachteile**
Trigger	- Entlastung des Client-Serververkehrs - Ausführung direkt im Datenbanksystem - Sicherstellung von referentieller Integrität	- Belastung des Datenbanksystems, da Trigger wiederum Trigger auslösen können - Beschädigung oder Zerstörung von Daten durch fehlerhaften Einsatz von Triggern - Erschweren des Debuggen und der Fehlersuche - Für Update, Insert und Delete seperate Trigger notwendig - Keine Kontrolle über die Reihenfolge der Ausführung bei mehr als einem Trigger pro Tabelle
Eventlistener	- Standard-Implementationen in Hibernate vorhanden - Zuweisen von mehreren Listenern zu einem Ereignis	- Keine Möglichkeit auf Ereignisse von bestimmten Objekten zu reagieren - Nur Erkennung von Änderungen auf Plain Old Java Objects möglich
AOP	- Erhöhte Wiederverwendbarkeit - Bessere Wartbarkeit - Klare Trennung der Verantwortungen	- Schwer nachvollziehbarer Ablauf des Programms - Erschweren des Debuggen und der Fehlersuche - Auftreten von unerwünschten Wechselwirkungen zwischen verschiedenen Aspekten

Tabelle 2.1: Vor- und Nachteile der Methoden zur Erkennung von Datenänderungen

2.2 Modellgetriebene Softwareentwicklung

Modellgetriebene Softwareentwicklung, Model Driven Software Development, ist eine Technik aus dem Software-Engineering, bei der mit Hilfe von formalen Modellen und Transformationen automatisch lauffähige Software erzeugt wird. Das Ziel dieses modellgetriebenen Ansatzes ist die Verbesserung der Software-Qualität, die Wiederverwendbarkeit sowie die Steigerung der Effizienz des Software-Entwicklungsprozesses [SV05, LÖ6, Her13]. In der Fachliteratur werden für modellgetriebene Softwareentwicklung synonyme Begriffe wie Model Driven Engineering oder Model Driven Development verwendet.

Im Allgemeinen besteht der modellgetriebene Softwareentwicklungsansatz aus folgende Komponenten:

- einem Modell
- einer domänenspezifischen Modellierungssprache, das das Modell formal beschreibt
- einer Menge von Transformatoren, die das Modell überführen

Diese Komponenten werden im Folgenden kurz erläutert.

Modell

Die Ideen der modellgetriebenen Entwicklung sind nicht neu, Modelle spielten in der Software-Entwicklung schon immer eine wichtige Rolle. Früher wurden sie in den ersten Phasen des Softwareentwicklungsprozesses benutzt, um Aspekte der Anforderungen, des Entwurfs oder der Implementierung zu beschreiben [Bal09]. Bei der modellgetriebenen Vorgehensweise werden Modelle nicht nur für Dokumentationen der Systemarchitektur verwendet, sondern sind Bestandteil des Softwareentwicklungsprozesses [SV05]. Die Modelle sind demnach abstrakt und formal zugleich, was nichts anderes bedeutet, als dass sie ein vollständiges Abbild des Programmcodes darstellen. Sie bilden somit im Zusammenhang mit den Transformationen die Grundlage für die automatische Generierung von Programmcode [SV05, LÖ6, Her13].

Domänenspezifische Modellierungssprache

Um formale Modelle beschreiben zu können, gibt es spezielle Sprachen. Bei der modellgetriebenen Softwareentwicklung wird diese Sprache als domänenspezifische Modellierungssprache, Domain Specific Language, bezeichnet [SV05, LÖ6, Bal09, Her13]. Im Gegensatz zu einer allgemeinen Programmiersprache, wie Java oder C++ ist sie auf einen bestimmten Problemraum (Anwendungsbereich) der jeweiligen Domäne zugeschnitten. Mit anderen Wort ist die domänenspezifische Modellierungssprache ein Werkzeug, um ein Modell einer Domäne zu erzeugen. Für domänenspezifische Modellierungssprachen existiert kein definierter Standard, so können die Modelle in beliebigen Modellierungssprachen beschrieben werden. Bei der modellgetriebenen Softwareentwicklung steht das Erreichen durch effiziente Modellierungen mit einer domänenspezifischen Modellierungssprache, die Wiederverwendbarkeit und hohe Codequalität im Vordergrund und vor der Verwendung von Standards [WS07]. Heutzutage wird meist eine UML-basierte Modellierungssprache eingesetzt, weil UML eine Vielzahl von Werkzeugen für Modellierungssprachen zur Verfügung stellt. Vor der Verwendung einer domänenspezifischen Modellierungssprache muss die Struktur dieser Modellierungssprache festgelegt werden. In der modellgetriebenen Softwareentwicklung übernimmt diese Aufgabe das Meta-Modell, welches die Konstrukte einer Modellierungssprache (abstrakte Syntax), ihre Beziehungen untereinander, sowie Einschränkungen bzw. Modellierungsregeln beschreibt [SV05]. Somit bestimmt das Meta-Modell die Grammatik der domänenspezifischen Modellierungssprache.

Transformatoren

Anhand von Transformationsregeln kann das mit Hilfe der domänenspezifischen Modellierungssprache beschriebene Modell transformiert werden. Bei der Modelltransformation wird zwischen den Varianten Modell-zu-Modell-Transformation, Abbildung von einem Quellmodell auf ein anderes Modell, und der Modell-zu-Text-Transformation unterschieden. Die Modell-zu-Text-Transformation wird hauptsächlich eingesetzt um aus einem Modell Programmcode zu generieren. Mit dieser Transformation lassen sich auch andere textuelle Systembeschreibungen erzeugen, wie Konfigurationsdateien oder Dokumentationen [LÖ6]. Dabei verwendet die Modell-zu-Text-Transformation Templates, Textvorlagen mit Platzhaltern, die während des Generierungsprozess mit den Daten aus dem Modell ersetzt werden. Bei der Erstellung eines Templates ist zu beachten, dass der generierte Programmcode von den Entwicklern z.B. beim Debuggen verstanden werden muss. Deshalb sollte versucht werden, einen möglichst "guten" Code zu generieren. Die Vorteile einer automatischen Code-Generierung sind die Synchronität zwischen dem Modell und dem Programmcode, die beliebig oft wiederholbare Generierung und dass Fehler leichter als bei manueller Umsetzung behoben werden können [Bal09]. Das manuelle Verändern von generiertem Programmcode sollte nicht erlaubt werden, weil sich dadurch viele Probleme bei Konsistenz, Build-Management oder Versionierung ergeben. Manuelle Veränderungen sollten klar gekennzeichnet werden, damit sie bei der Generierung von Code nicht überschrieben werden. Aus diesem Grund ist es ratsam, eine Trennung zwischen generiertem und nicht-generiertem Programmcode zu vollziehen [SV05]. Ein bekannter Vertreter der Modell-zu-Text-Transformation ist das template-basierte Werkzeug Java Emitter Templates (JET)[3], Bestandteil des Eclipse Modeling Frameworks. Dieses Werkzeug wird im Konzept zur feingranularen Änderungserkennung (Kapitel 4) genauer betrachtet.

Zusammenfassung der modellgetriebenen Softwareentwicklung

Durch den Einsatz von modellgetriebener Softwareentwicklung wird der Ansatz unternommen, ein Problem eines bestimmten Geltungsbereichs (Domäne) zu abstrahieren, um sich auf das Wesentliche zu konzentrieren. Die Abstraktion der Domäne soll sich in einem formalen Modell widerspiegeln. Mit Hilfe von Transformatoren oder Generatoren wird aus einem domänenspezifischen Modell automatisch Programmcode erzeugt, der auf den vorhandenen Plattformen ausgeführt werden kann [SV05]. Durch die Sicherstellung der Korrektheit und Fehlerfreiheit von Transformationen kann die Korrektheit und Fehlerfreiheit angenommen werden [Sch06].
Modellgetriebene Softwareentwicklung wird die Programmiersprachen und damit die normale Programmierung nicht komplett ablösen können, da es immer nicht generalisierbaren Programmcode geben wird, der nicht sinnvoll abstrahierbar und damit nicht effizient modellierbar ist.
Ein prominentes Beispiel für die Umsetzung von modellgetriebener Softwareentwicklung ist die Model Driven Architecture der Object Management Group[4]. Im Gegensatz zur modellgetriebenen Softwareentwicklung hat Model Driven Architecture das Ziel, die Interoperabilität zwischen den Werkzeugen und damit die Standardisierung von Modellen für populäre Anwendungsbereiche zu verwirklichen [SV05].

[3] http://www.eclipse.org/modeling/m2t/?project=jet
[4] http://www.omg.org/

2.3 Zusammenfassung

Ziel dieses Kapitels war es, einen Einblick in die Vorgehensweise bei der Erkennung von Datenänderungen zu geben und die Möglichkeit der Umsetzung in einem modellgetriebenen Softwareentwicklungsansatz zu beschreiben. Im Rahmen der Erkennung von Datenänderungen wurde sich auf die Methoden der Datenbank-Trigger, der Eventlistener in Hibernate und der aspektorientierten Programmierung konzentriert. Die Vor- und Nachteile der verschiedenen Methoden sind in der Tabelle 2.1 dargestellt. Nach Einführung der Methoden wurde der Ansatz modellgetriebener Softwareentwicklung vorgestellt. Dabei wurde das Zusammenspiel der Komponenten von modellgetriebener Softwareentwicklung zur Erzeugung von Programmcode durch Transformationen eines Modells veranschaulicht.

3. Anforderungsanalyse

In diesem Kapitel werden anhand der gegebenen Untersuchungsfragen (siehe Kapitel 1) die Anforderungen an die effiziente Datenanalyse spezifiziert. Zuvor werden in Abschnitt 3.1 die existierenden Probleme kategorisiert und innerhalb der Problemkategorien ausführlich beschrieben. Resultierend aus den aufgezeigten Problemen werden dann in Abschnitt 3.2 die Anforderungen an die umzusetzenden Lösungen aufgezeigt.

3.1 Problemstellung

Im Bankwesen werden viele Daten von Kunden gesammelt, um daraus eine gezielte, optimale und individuelle Betreuung des Kunden zu gewährleisten. Zu diesen Daten gehören neben den privaten Informationen über den Kunden auch die Wünsche und Ziele der potentiellen Neukunden beziehungsweise der Bestandskunden. Aus den so gewonnenen Informationen erfolgt mit Hilfe der Datenanalyse die systematische Aufbereitung der erhobenen Daten. Im nächsten Schritt werden diese Analysedaten verwendet, um den Kunden in ein bestimmtes Kundensegment einzuordnen. Die Einordnung der Kunden in ein bestimmtes Segment wird benötigt, um daraufhin geeignete Strategien zu entwickeln. Die Strategien beinhalten Vorschläge, bei welchen Produkten für den Kunden ein Bedarf besteht oder welche für sie interessant sein könnten. Diese Strategien präsentiert dann der Bankberater dem Kunden.
Die Banken im GAD-Verbund setzen die Software MinD.banker zur Steuerung der Führungs- und Vertriebsprozesse im Firmenkundengeschäft von Volksbanken und Raiffeisenbanken dar. Zahlreiche Funktionen in MinD.banker dienen der innovativen Analyse und Steuerung des Bestands- und Neukundengeschäfts und helfen, den Vertriebserfolg der Firmenkundenbetreuer zu optimieren[1].

Im folgenden werden die Probleme beschrieben, die sich aus einer Datenanalyse ergeben. Dazu wurden die Probleme in verschiedenen Kategorien eingeteilt.

[1]http://www.eudemonia-solutions.de/referenzen

Generelle Probleme

In der ersten Problemkategorie werden die generellen Probleme betrachtet, die in allen Anwendungen zur Datenanalyse vorhanden sind.
Die wichtigste Grundlage für die Datenanalyse bildet die Datenerfassung. Wenn die erhobenen Daten fehlerhaft oder nicht vollständig sind, kann die beste Datenanalyse keine nutzbaren Ergebnisse liefern. Aus diesem Grund ist oftmals mühsame und zeitintensive Datenerfassung unverzichtbar, wenn die Banken auch zukünftig mit ihren Produkten erfolgreich sein wollen. Bei gewissenhafter Erfassung, Analyse und anschließender Auswertung kann erkannt werden, welche Produkte auf einem aufsteigenden Trend sind und welche zunehmend weniger gefragt sind.
Bedeutsam in diesem Zusammenhang ist das Problem der Aktualität der Daten. Hinsichtlich der schnellen Änderungen in der heutigen Zeit sollte die Anpassung der Daten zeitnah vorgenommen werden. Diese sollten mittels regelmäßiger Gespräche zwischen dem Kunden und dem Bankberater durchgeführt werden, damit die Strategien an eventuelle Veränderungen des Kunden anpasst werden. Diese veränderten Daten fließen wieder in das System ein, um im Anschluss daran die Aktualisierung der Analysedaten und darauf aufbauend eine eventuelle Anpassung des Kunden in ein anderes Segment durchzuführen. Dieses führt letztendlich zur Aktualisierung der Strategien. Von der Auswahl der richtigen Strategie basierend auf den vorhandenen Daten hängt unter anderem die Entscheidung ab, ob der Kunde die vorgeschlagenen Produkte kauft. Die Datenanalyse muss optimal auf die Interessen des Kunden abgestimmt und immer aktuell sein.

Probleme von State-of-the-Art-Lösungen (ETL)

In diesem Abschnitt werden die Probleme, die sich durch die Datenanalyse in Interaktion mit den ETL-Prozessen aus dem Bereich Data Warehouse ergeben, beschrieben.
In heutigen Anwendungen wird die Datenanalyse aktuell durch Data-Warehouse-Technologien abgedeckt. Dabei werden die operativen Daten mit Hilfe des ETL-Prozesses (Extraktion-Transformation-Laden) aus den verschiedenen Datenquellen entnommen, mit Hilfe der Transformation bereinigt und in ein einheitliches Format umgewandelt, um danach in das Data Warehouse geladen zu werden. Ein Problem, das sich bei der Extraktion ergibt, ist, dass diese Vorgänge zeitintensiv sind und daher meist zu festgelegten Zeitpunkten und für fest definierte Datenbestände (statische Extraktion) durchgeführt werden [GRC09]. Aus diesem Grund besteht die Möglichkeit, dass die Analysedaten auf einem alten Stand sind. Dadurch kann die Aktualität der Analysedaten nicht gewährleistet werden und somit kann diese Extraktionstechnik nur bedingt eingesetzt werden. An dieser Stelle eignet sich die Anwendung von inkrementeller Extraktion von operativen Daten zur Aktualisierung der Analysedaten. Dabei werden nur die operativen Daten, die zwischen dem aktuellen und letztem Extraktionsprozess verändert wurden, bei der Aktualisierung der Analysedaten betrachtet.
Für den Einsatz der inkrementellen Extraktion wäre es daher wünschenswert, dass die fachlichen Abhängigkeiten zwischen den operativen Daten und den Analysedaten definiert werden. Nur wenn diese Abhängigkeiten bekannt sind, können die Analysedaten effizient aktualisiert werden. Es ist empfehlenswert, die fachlichen Abhängigkeiten so feingranular wie möglich zu beschreiben. Je detaillierter die Abhängigkeiten

zwischen den operativen Daten und den Analysedaten skizziert sind, desto kleiner ist die Notwendigkeit der Aktualisierung der Analysedaten. Dies hat den Vorteil, dass bei einer Änderung der operativen Daten nur die Analysedaten, die fachlich mit dem geänderten operativen Daten verbunden sind, aktualisiert werden. Damit wird der Aufwand der Aktualisierung der Analysedaten deutlich reduziert, was zu einer effizienteren Datenanalyse führt.

Probleme von MinD.banker

In der letzten Kategorie wird auf die derzeitigen Problem in MinD.banker eingegangen.
In MinD.banker müssen durch Änderungen an den operativen Daten (z.B. Änderung des Kapitalsaldos eines Kontos) die abhängigen Analysedaten aktualisiert werden. Diese Datenanalyse ist in MinD.banker ineffizient, da die fachlichen Abhängigkeiten zwischen operativen Daten und Analysedaten nicht betrachtet werden. Dies führt dazu, dass die Analysedaten immer aktualisiert werden, auch wenn keine Abhängigkeit zwischen den geänderten operativen Daten und den Analysedaten besteht. Ein anderer Punkt in dem Zusammenhang mit der Ermittlung des Aktualisierungsbedarf ergibt sich, wenn die aktuelle Implementation in MinD.banker untersucht wird. Gegenwärtig wird die Aktualisierung der Analysedaten immer durchgeführt, auch wenn bei den operativen Daten keine "wirkliche" Datenänderung stattgefunden hat. Ein weiteres Problem im Zuge der Datenanalyse in MinD.banker ist, dass die Aktualisierung der Analysedaten nach Erkennung von Datenänderungen manuell ausgelöst werden muss. Dieses wird derzeit durch direkte Aufrufe an den entsprechenden Codestellen sichergestellt. Damit muss der Entwickler ganz genau wissen, welche Analysedaten nach dem Ändern der operativen Daten aktualisiert werden sollen. Erschwerend kommt hinzu, dass die direkten Aufrufe über die gesamte Anwendung verteilt sind. Das stellt eine große Fehlerquelle dar, da die Entwickler diese Aufrufe explizit setzen müssen und sie so vergessen könnten. Das führt dazu, dass die Erkennung von Änderungen auf Basis der Abhängigkeiten zwischen den operativen und Analysedaten abhängig vom individuellen Wissen der Entwickler ist. Des Weiteren lässt sich die Erkennung von Datenänderungen und darauffolgende Aktualisierung der Analysedaten unter Berücksichtigung der Abhängigkeiten schwer testen, da die Aufrufe über den kompletten Programmcode verstreut sind.

Zusammenfassung

Zusammenfassend legen wir im Folgenden alle genannten Probleme in einer Übersicht darlegt.

- Gewährleisten von aktuellen, fehlerfreien und vollständigen operativen Daten
- Aktualisierung der Analysedaten zu festgelegten Zeitpunkten und für fest definierte Datenbestände
- Nichtberücksichtigung der fachlichen Abhängigkeiten zwischen den operativen Daten und Analysedaten
- ineffiziente Aktualisierung der Analysedaten

- manuelles Auslösen der Aktualisierung durch den Entwickler ⇒ sehr fehleranfällig
- Testbarkeit schwierig, da die Aufrufe zur Aktualisierung über das gesamte Projekt verteilt sind

3.2 Anforderungen an eine effiziente Datenanalyse

Die Anforderungen für die effiziente Datenanalyse ergaben sich aus den aktuellen Problemen der MinD.banker Anwendung, die im Abschnitt 3.1 beschrieben sind und einem offenen Interview mit dem Architekturboard der Eudemonia Solutions AG. Eine Anforderung stellt dabei eine Bedingung oder eine Eigenschaft dar, die von einem System oder einer Person zur Lösung eines Problems oder zur Erreichung eines Ziels benötigt wird [Poh08]. Für den Benutzer bedeutet dies also, dass sowohl seine Wünsche als auch seine Ziele in dem Begriff Anforderung zusammengefasst werden. Dazu können gesetzliche Normen und Standards oder auch organisatorische Notwendigkeiten gehören.
Die Anforderungen können in funktionale und nicht-funktionale Abhängigkeiten unterteilt werden [Poh08]. Die funktionalen Anforderungen beantworten die Frage „Was das System machen soll?“. In den funktionalen Anforderungen werden Funktionen aus Sicht des Benutzers oder der Systemumgebung, die das System oder eine Systemkomponente ausführen soll, beschrieben. Häufig werden auch Systemzustände und das Verhalten des Systems und seiner Umgebung in funktionalen Anforderungen veranschaulicht. Dabei werden detailliert die Eingaben (Daten, Ereignisse) und deren Einschränkungen, Ausgaben (Daten, Fehlermeldungen) sowie bekannte Ausnahmen beschrieben.
Die nicht-funktionalen Anforderungen hingegen definieren gewünschte Qualitätsmerkmale des geplanten Systems, darunter zählen unter anderem die Benutzbarkeit, die Performanz des Systems oder die Zuverlässigkeit. Nicht-funktionale Anforderungen können unterteilt werden in

- Qualitätsattribute der gewünschten Funktionen
- Anforderungen an das implementierte System als Ganzes
- Vorgaben für die Durchführung der Systemerstellung
- Anforderung an Prüfung, Einführung, Betreuung und Betrieb

Nicht-funktionale Anforderungen beeinflussen sich gegenseitig und hängen voneinander ab. Die verschiedenen Arten von nicht-funktionalen Anforderungen werden in vielen Projekten sichtbar, weshalb sie im Rahmen der DIN 66272 des ISO/IEC 9126 klassifiziert sind.
Die funktionalen und nicht-funktionalen Anforderungen sollten immer gleich behandelt werden, da ein System, dass sämtliche funktionale Anforderungen perfekt abdeckt und erfüllt, die nicht-funktionalen Anforderungen aber nicht korrekt berücksichtigt, nicht vom Benutzer bzw. Kunden akzeptiert wird.

Funktionale Anforderungen

Die erste funktionale Anforderung an die effiziente Datenanalyse ist die automatische Erkennung von Datenänderung. Sobald eine Datenänderung an den operativen Daten stattgefunden hat, soll das System automatisch benachrichtigt werden, dass Änderungen durchgeführt wurden, ungeachtet davon, an welchen Stellen im System die Datenänderungen erfolgt sind. Dadurch müssen die Änderungen am System nicht mehr manuell durch den Entwickler durchgeführt werden, was zur Zeit in der Anwendung MinD.banker der Fall ist. Zu den Datenänderungen gehören das Hinzufügen, Aktualisieren und Löschen von Daten.
Eine weitere Anforderung ist, dass die automatische Erkennung von Datenänderungen feingranular erfolgen muss. Die feingranulare Erkennung ist der erste Schritt, um die Effizienz der Datenanalyse zu erhöhen, um gezielt nur die Analysedaten, die von den Datenänderungen betroffen sind, zu aktualisieren. Um zu bestimmen, ob sich die geänderten operativen Daten von den Bestandsdaten, operativen Daten vor der Änderungen unterscheiden, wird eine Funktion benötigt, die den Aktualisierungsbedarf an den geänderten operativen Daten ermittelt. Diese Funktion vergleicht den Inhalt bzw. Wert des Attributs des geänderten Datenobjekts mit dem Datenobjekt vor der Änderung. Falls keine Änderung stattgefunden hat müssen die Analysedaten nicht aktualisiert werden.
Eine weitere wichtige Anforderung, um die Datenanalyse möglichst effizient zu gestalten, ist die Berücksichtigung der fachlichen Abhängigkeiten zwischen den operativen Daten und den Analysedaten, um gezielt betroffene Analysedaten zu aktualisieren. Die Übersicht der fachlichen Abhängigkeiten zwischen den operativen Daten und den Analysedaten sollen modelliert werden. Dabei soll dokumentiert werden, welche operativen Daten welche Analysedaten beeinflussen und wie sich Beziehungen zueinander verhalten. Außerdem soll deutlich gemacht werden, ob Abhängigkeiten zwischen den Analysedaten bestehen und, wenn ja, welche diese sind. Sobald ein Segment aus Analysedaten aktualisiert wird, müssen ebenso alle abhängigen Segmente der Analysedaten aufgefrischt werden.

Nicht-funktionale Anforderungen

Die bedeutendste nicht-funktionale Anforderung ist die Effizienz der Ermittlung des Aktualisierungsbedarfs der Analysedaten. Hierdurch kann garantiert werden, dass nur die erforderlichen Analysedaten, die in einer Abhängigkeit zu den geänderten operativen Daten stehen, aktualisiert werden. Die Modellierung der Abhängigkeiten zwischen den operativen Daten und den Analysedaten soll einen durchschaubaren und nachvollziehbaren Überblick über die Zusammenhänge zwischen operativen Daten und Analysedaten ermöglichen. Die Entwickler müssen sich nicht mehr selbst um die Abhängigkeiten kümmern, sondern diese Aufgabe übernimmt in diesem Punkt das Modell. Dadurch wird die Transparenz gegenüber dem Entwickler erhöht, welche die erste nicht-funktionale Anforderung ist.
Eine weitere nicht-funktionale Anforderungen, die an das System gestellt wird, ist die Testbarkeit der kompletten Anwendung. Dies soll mit verschiedenen Tools aus dem Bereich der Unit-Tests sichergestellt werden, um frühzeitig Fehler im Ablauf der Datenanalyse zu erkennen und zu beseitigen. Zudem soll die Anwendung die Voraussetzung für eine erhöhte Erweiterbarkeit und Wartbarkeit schaffen.

Eine Übersicht der ermittelten Anforderungen und deren Kategorisierung ist in Tabelle 3.1 zusammengefasst dargestellt. Hierbei steht *f* für eine funktionale, *nf* für eine nicht-funktionale Anforderung.

Anforderungen	**Erläuterung**	**Art**
automatische Erkennung	Der Prototyp soll Datenänderungen an den operativen Daten automatisch erkennen können.	f
Feingranularität	Mit einer feinen Granularität von operativen Daten soll die Effizienz der Datenanalyse sichergestellt werden.	f
Modellierung der Abhängigkeit	Die analysierten fachlichen Abhängigkeiten zwischen operativen Daten und Analysedaten sollen mit Hilfe eines Modell visualisiert werden. Dazu wird dokumentiert, welche operativen Daten welche Analysedaten beeinflussen und wie sich Beziehungen zueinander gestalten.	f
Ermittlung des Aktualisierungsbedarfs	Nach Erkennung von Datenänderungen erfolgt die Ermittlung, ob eine Abhängigkeit zwischen den geänderten operativen Daten und den Analysedaten vorliegt. Falls keine Beziehung existiert, wird keine Aktualisierung durchgeführt.	f
Effizienz	Die Aktualisierung der Analysedaten wird nur auf Grundlage eines berechtigten Aktualisierungsbedarfs durchgeführt.	nf
Transparenz gegenüber den Entwicklern	Es muss für die Entwickler klar erkennbar sein, welche Abhängigkeiten zwischen den operativen Daten und den Analysedaten bestehen.	nf
Testbarkeit	Die Anwendung soll mit Hilfe von automatisierten Unit-Tests testbar sein, um Fehler frühzeitig zu erkennen.	nf
Erweiterbarkeit und Wartbarkeit	Der Prototyp soll zu einer Erhöhung der Erweiterbarkeit und Wartbarkeit führen.	nf

Tabelle 3.1: Anforderungen an effiziente Datenanalyse

3.3 Zusammenfassung

In diesem Kapitel wurden die Probleme bei der Aktualisierung von Analysedaten geschildert und die daraus resultierenden Anforderungen an eine effiziente Datenanalyse vermittelt. Im Rahmen der Bestimmung der Anforderungen wurde eine Unterteilung in funktionale und nicht-funktionale Anforderungen vorgenommen. Die Übersicht der ermittelten Anforderungen und deren Kategorisierung ist in Tabelle 3.1 veranschaulicht.

4. Konzept zur feingranularen Änderungserkennung

Der Fokus dieser Arbeit liegt darauf, die Effizienz der bestehenden Datenanalyse zu steigern, indem die fachlichen Abhängigkeiten zwischen den operativen Daten und den Analysedaten feingranular berücksichtigt werden und anhand dessen der Aktualisierungsbedarf ermittelt wird.
In Abschnitt 4.1 erfolgt zunächst die Vorstellung der Ideen des Konzepts. Bevor der Konzeptentwurf in Form eines Modells veranschaulicht wird, werden im Abschnitt 4.2, die für das Konzept relevanten Abhängigkeiten zwischen operativen Daten und Analysedaten vermittelt. Im Anschluss daran wird im Abschnitt 4.3 das Modell des Konzept skizziert und auf die Komponenten dieses Modells eingegangen.

4.1 Modellierung des Konzeptentwurfs

In Kapitel Abschnitt 2.2 wurde die Ideen eines modellgetriebenen Softwareentwicklungsansatzes erläutert. Dieser Ansatz scheint für das Konzept zur feingranularen Änderungserkennung in MinD.banker geeignet zu sein, da durch den Einsatz von automatisierter Transformation und formal definierter Modellierungssprachen die Softwarequalität gesteigert werden kann. Des Weiteren können über die komplette Anwendung verteilte Implementierungsaspekte zentral an einer Stelle verändert werden, was die Fehlerbeseitigung im generiertem Code erleichtert [SV05]. Aus diesem Grund wird versucht, den modellgetriebenen Ansatz auf die Implementierung in MinD.banker zu übertragen. Bevor auf das Modell näher eingegangen wird, wird der Ablauf bei der Erkennung von Datenänderungen anhand eines Aktivitätsdiagrammes gezeigt. In Abbildung 4.1 ist die Grundidee, um automatisch Datenänderungen von operativen Daten feingranular zu erkennen und darauf aufbauend den Aktualisierungsbedarf der Analysedaten zu ermitteln, in Form eines Aktivitätsdiagramms beschrieben. Im ersten Schritt nach Erkennung einer Datenänderung wird überprüft, ob für diese Datenänderung ein Aktualisierungsbedarf besteht. Nachdem bei der Analyse festgestellt wurde, dass eine Aktualisierung der Analysedaten unerlässlich ist, können als nächstes die abhängigen Analysedaten ermittelt werden. Danach

sind alle erforderlichen Daten vorhanden, um die Aktualisierung der Analysedaten durchzuführen. Wie diese Aktualisierung der Analysedaten realisiert wird, ist nicht mehr Bestandteil dieses Konzeptes. Sobald sich in einem der ersten beiden Schritte herausstellt, dass keine Änderungen der Analysedaten notwendig ist, werden die nachfolgenden Schritte gar nicht ausgeführt.

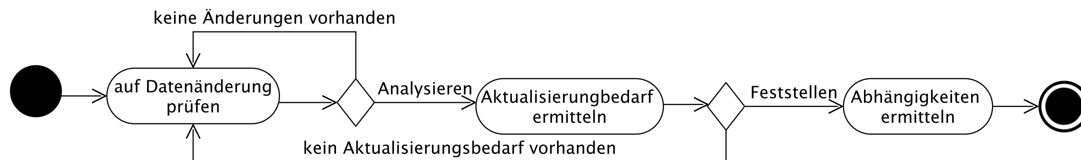

Abbildung 4.1: Ablauf zur effizienten Datenanalyse

Mit Hilfe dieses Konzeptes ist es möglich, die verschiedenen Methoden zur Erkennung von Datenänderungen (siehe Abschnitt 2.1) umzusetzen. Im aktuell vorliegenden Konzept können die verschiedenen Methoden frei gewählt werden; es wird keine Methode vorgeschrieben. Dadurch ist das Konzept relativ flexibel. So ist es jederzeit möglich, den derzeit verwendeten Ansatz durch einen anderen Ansatz auszutauschen. Durch den Austausch der Methoden ändert sich nichts am Ablauf der Erkennung von Datenänderungen und anschließender Ermittlung des Aktualisierungsbedarfs. Alle Ansätze sollten das gleiche Ergebnis liefern. Lediglich die Durchführung der Umsetzung ist in den Methoden grundverschieden.
Es existiert allerdings die Einschränkung, dass nur ein Ansatz gleichzeitig unterstützt ist. Es ist durchaus denkbar durch Anpassung beziehungsweise Erweiterung des Konzepts, diese Beschränkung auf eine Methode aufzuheben. Dies kann zukünftig interessant werden, wenn eine Methode zur Erkennung von Datenänderungen aufgrund einer Beschränkung in ihrer Funktionalität nicht alle Anforderungen erfüllen kann. In diesem Fall bestünde die Möglichkeit, durch die Kombination mit einem anderen Ansatz diese Lücke in der Abdeckung zu schließen. Ein weiterer Vorteil dieses Konzeptes ist, dass die Änderungen zentral an einer fest definierten Stelle im Code erfolgen können und somit nicht mehr über die gesamte Anwendung verteilt vorzunehmen sind.
Um das komplette Modell automatisch erzeugen und ausführen zu lassen, bietet es sich an, dies in einem Build-Prozess umzusetzen. In diesen Build-Prozess werden Regeln definiert, wie das Modell abgearbeitet werden soll, um so den durch das Modell erzeugten Programmcode in die bestehende Anwendung zu integrieren. Auf diese Weise bietet sich der Build-Prozess gleichwohl zum Testen des Modells an. Des Weiteren benötigen die meisten Entwickler kein Wissen über das Modell und den Build-Prozess, da sie nur an dem Ergebnis, dem Programmcode, interessiert sind.

4.2 Abhängigkeiten zwischen den operativen Daten und den Analysedaten

Bevor das Modell mit den dazugehörigen Komponenten beschrieben wird, werden die Abhängigkeiten zwischen den operativen Daten und den Analysedaten herausgestellt.
Um die fachlichen Abhängigkeiten formal zu beschreiben, nutzen wir ein Modell. Dabei stellt ein Modell ein beschränktes Abbild der Wirklichkeit dar, das eine vereinfachte, reduzierte Sicht auf bestimmte Aspekte der Wirklichkeit veranschaulichen soll. Hierfür werden zunächst die Abhängigkeitsbeziehungen zwischen den operativen Daten und den Analysedaten mit Hilfe von Modellen beschrieben und im nächsten Schritt ausgewertet. In der Literatur herrscht Einigkeit darüber, dass für das Beschreiben und Auswerten von Abhängigkeitsbeziehungen ein Abhängigkeitsmodell benötigt wird. Dieses Abhängigkeitsmodell definiert die Struktur der Abhängigkeitsbeziehungen, anhand dessen die Abhängigkeiten zwischen den Operativ- und Analysedaten relativ einfach identifiziert werden können. Dabei können folgende Situationen analysiert werden und darauf aufbauend folgende Fragen mit Hilfe des Abhängigkeitsmodells beantwortet werden:

- Welche Voraussetzung müssen grundsätzlich gelten, damit eine Aktualisierung der Analysedaten durchgeführt werden kann?
- Welche fachlichen Abhängigkeiten bestehen zwischen Operativ- und Analysedaten?
- Welche Beziehungen bestehen zwischen Analysedaten untereinander?
- Wie wird mit zirkulären Abhängigkeiten umgegangen?

Abhängigkeitsmodell

Die Grundlage für die gewählte Struktur des Abhängigkeitsmodells sind die bisherigen Ansätze von Abhängigkeitsmodellen [RJ01, WP10]. Im Folgenden wird angelehnt an die Modellierungssprache UML die grundsätzliche Struktur des Abhängigkeitsmodells und ihre Elemente beschrieben. Abbildung 4.2 zeigt die Modellelemente des Abhängigkeitsmodells. Zu den Elementen gehören `Entity`, `EntityType`, `AnalysisSegment` und `Dependency`.

Um eine der Anforderungen, die Feingranulariät, gewährleisten zu können, werden die operativen Daten in mehrere kleine, physisch selbständige Dateneinheiten aufgeteilt. Die so entstehenden einzelnen Dateneinheiten können zielgerichtet bestimmten Datensegmenten der Analysedaten zugeordnet werden. Mit diesen kleineren Dateneinheiten lässt sich die Datenanalyse wesentlich effizienter und einfacher durchführen. Da in den meisten Fällen die Änderungen an den Dateneinheiten nur ein beziehungsweise wenige Datensegmente betreffen, können so gezielt nur die relevanten Datensegmente aktualisiert werden. Dabei spiegelt das Modellelement `Entity`, Entität, eine Dateneinheit aus der Menge der operativen Daten wider. Eine Entität ist immer durch einen Namen und ein Attribut bestimmt. Dabei dient das Attribut zur Beschreibung der Eigenschaft der Entität. Die Art der Entität wird durch

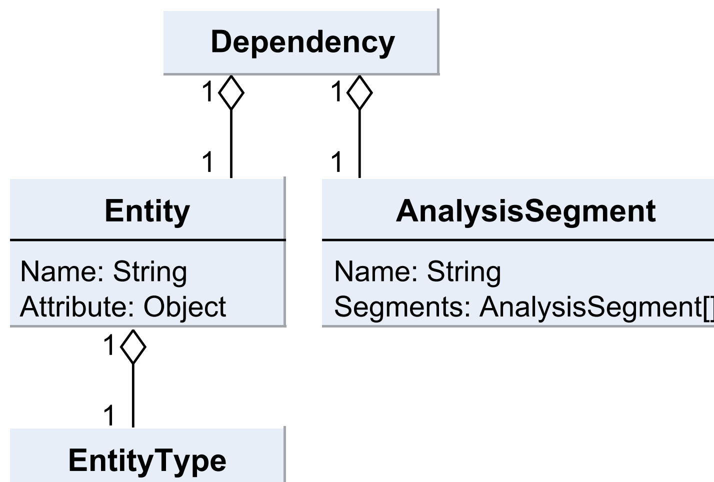

Abbildung 4.2: Abhängigkeitsmodell

deren Typ festgelegt. Dadurch können Entitäten gleichen Entitätstyps voneinander unterschieden werden. Das Modellelement `AnalysisSegment` repräsentiert ein Datensegment innerhalb der Analysedaten. Hierbei ist ein Datensegment im Sinne von MinD.banker ein Funktionsbereich innerhalb der Anwendung. Da die Datensegmente durch andere Datensegmente beeinflusst werden können, müssen diese Abhängigkeiten festgehalten werden. Um diese Abhängigkeiten zwischen den Datensegmenten abzubilden, hat das Modellelement `AnalysisSegment` zusätzlich zum Namen eine Menge von abhängigen Datensegmenten. Diese Menge kann leer sein, ein Element oder mehrere Elemente beinhalten.
Das Modellelement `Dependency` bildet genau eine Abhängigkeit zwischen einer `Entity` und einem `AnalysisSegment` ab. Da innerhalb eines Datensegments eine Menge von abhängigen Datensegmenten definiert sind, können so indirekt die Beziehungen zwischen einer Entität und allen abhängigen Datensegmenten abgebildet werden. Dabei ist zu beachten, dass zirkuläre Effekte durch das Abhängigkeitsmodell erkannt, aber nicht behoben werden können. Diese Effekte müssen bei der Implementierung des Modells manuell bereinigt werden.

Als nächstes werden die möglichen Abhängigkeiten aufgelistet und eine Nutzung dieser Abhängigkeiten im Abhängigkeitsmodell bestimmt. Es gibt drei Arten von möglichen Abhängigkeiten:

1. operativen Daten <-> operativen Daten
2. operativen Daten <-> Analysedaten
3. Analysedaten <-> Analysedaten

Für MinD.banker benötigen wir nur die Abhängigkeiten zwei und drei, weil die erste Abhängigkeit in MinD.banker nicht wichtig ist. Diese beiden Abhängigkeiten können in unserem Modell genutzt werden. Die zweite Abhängigkeit bildet unter anderem

die Grundlage für die in dem Konzept vorgestellten Konfigurationsdateien. Die dritte Abhängigkeit wird in einer eigenen Abhängigkeitsdatei ausgelagert.

Beispiel für eine fachliche Abhängigkeit in MinD.banker

Um das Abhängigkeitsmodell anschaulicher zu gestalten, wird es mit einem Beispiel aus MinD.banker illustriert. Dieses Beispiel wird in den fortlaufenden Kapiteln der Arbeit teilweise wiederverwendet. In MinD.banker sind jedem Kunden bestimmte Konten zugeordnet. Zu einem Konto gehört unter anderem immer ein Saldo, Zinssatz und der dazugehöriger Kunde. Ein Attribut eines Kontos stellt eine Entität des Modells dar. Der Enitätstyp ist das Konto selbst. Wenn wir jetzt nur mal den Saldo eines Kontos betrachten, stellt sich eine Abhängigkeit zu dem Analysesegment *Vermögens- und Schuldenübersicht* heraus. Dieses Analysesegment hat wiederum Beziehungen zu den Analysesegmenten *Produktvorschläge* und *Datensegmentierung*. Sobald Änderungen an dem Betrag des Saldo durch einen Import oder durch manuelle Eingabe vorgenommen wird, müssen die drei abhängigen Analysesegmente aktualisiert werden. Bei allen anderen vorhandenen Analysesegmenten besteht keine Notwendigkeit zur Aktualisierung.

4.3 Komponenten des Modells

In diesem Kapitel wird das vollständige Modell zur effizienten Datenanalyse vorgestellt, um darauffolgend die Komponenten des Modells detaillierter zu betrachten und einen Einblick in das Zusammenspiel der Komponenten zu geben.

In der nachfolgenden Abbildung 4.3 wird das Modell und die Beziehungen zwischen den Komponenten dargestellt.

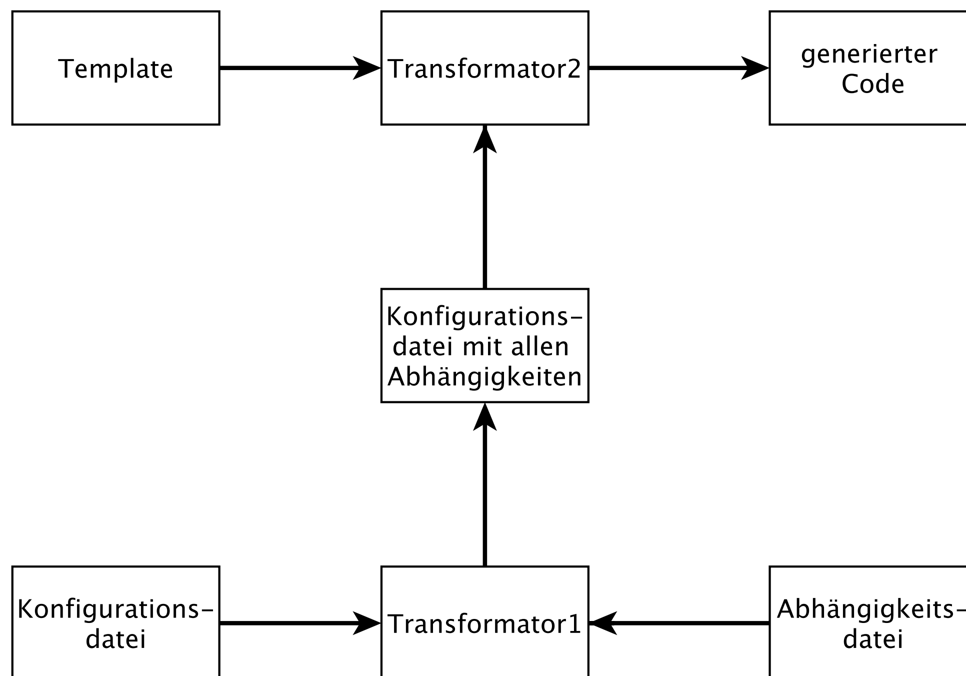

Abbildung 4.3: Modell zur effizienten Datenanalyse

Das oben dargestellte Modell besteht aus folgenden Komponenten:

- einer Konfigurationsdatei, die alle benötigten Einstellungen beinhaltet
- einer Abhängigkeitsdatei der Analysedaten, die die Abhängigkeiten der Analysedaten untereinander beinhaltet
- einem Transformator, der aus der Konfigurationsdatei und dem Abhängigkeitsmodell eine Konfigurationsdatei mit allen Abhängigkeiten erzeugt
- einer Template-Datei, eine Vorlage für den zu generierenden Code ist
- einem Transformator, der aus dem Template und der erzeugten Konfigurationsdatei den generierten Code erzeugt

Wie sich aus der Abbildung Abbildung 4.3 des gezeigten Modells erkennen lässt, kann man das Modell in zwei aufeinander aufbauende Prozesse unterteilen. Die Basis für die automatische Erkennung von Datenänderungen und der daraufhin folgenden Ermittlung des Aktualisierungsbedarfes bilden die Konfigurationsdateien. Der erste Prozess beschäftigt sich mit der Erzeugung von Konfigurationsdateien mit allen erforderlichen Abhängigkeiten. Dazu erzeugt der Transformator1 auf der Grundlage einer Konfigurationsdatei und der Datei des Abhängigkeitsmodells eine generierte Konfigurationsdatei, die alle Abhängigkeiten zwischen den operativen Daten und den Analysedaten, als auch der Abhängigkeiten der Analysedaten untereinander enthält. Die erzeugte Konfigurationsdatei beinhaltet somit alle Einstellungen, die benötigt werden, um eine effiziente Datenanalyse durchzuführen. Das Ergebnis des

ersten Prozesses, die generierte Konfigurationsdatei, fließt in den zweiten Prozess ein und bildet somit das Fundament für diesen Prozess. Im zweiten Prozess wird aus dem vom Entwickler formulierten Template und der erzeugten Konfigurationsdatei durch Transformation der generierte Code erzeugt. Dabei ersetzt der Transformator die Platzhalter des Templates mit den Einstellungen der Konfigurationsdatei.
Der in der Abbildung 4.1 gezeigte Ablauf zur feingranularen Änderungserkennung spiegelt sich in dem mit Hilfe des Modell generierten Code wider. Dabei ist es unerheblich, ob der Output des Templates einen Java-Code oder einen SQL-Code erzeugt. Sobald das Modell einmal korrekt beschrieben wurde und die Transformatoren fehlerfrei arbeiten, müssen die Änderungen nur noch an zwei Elementen des Modells vorgenommen werden. Zum einen an den Konfigurationsdateien, sobald eine Strukturanpassung an den operativen Daten stattgefunden hat und zum anderen an der Abhängigkeitsdatei. Die weiteren Komponenten des Modells benötigen keine Anpassung.

In dem nächsten Kapitel wird auf die einzelnen Komponenten des Modells detaillierter eingegangen.

4.3.1 Konfigurationsdatei

Eine der wichtigsten Komponenten dieses Modells bildet die Konfigurationsdatei. Aus Gründen der Anpassbarkeit, Wartbarkeit sowie der Übersichtlichkeit wurde eine Aufteilung der Konfigurationsdatei in mehrere kleinere Konfigurationsdateien vorgenommen. Die Aufteilung der Konfigurationsdatei wurde so vorgenommen, dass jede Konfigurationsdatei immer auf einen bestimmten Bereich, eine Domäne, zugeschnitten ist. Im Datenbankumfeld könnte man sich vorstellen, dass eine Domäne eine Tabelle repräsentiert. Auf dem Gebiet der Objektorientierung könnte eine Klasse eine Domäne abbilden. Für weitere Gebiete der Softwaretechnik können analog bestimmte Objekte für eine Domäne gefunden werden.

Es gibt zwei Möglichkeiten, wie die Konfigurationsdateien in einer Anwendung umgesetzt werden können. Auf der einen Seite können für jede Domäne innerhalb der Anwendung Konfigurationsdateien erzeugt werden und auf der anderen Seite besteht die Möglichkeit, nur die benötigten Konfigurationsdateien für Domänen zu erstellen, die zu einer Aktualisierung der Analysedaten beitragen. Der Vorteil bei der ersten Variante ist, dass es konsequent zu jeder Domäne eine Konfigurationsdatei gibt, was zu einem positiven Effekt beim Anpassen oder Ändern der Konfigurationsdatei führen kann. Im Gegensatz zur ersten Möglichkeit ist der Implementierungsaufwand bei der zweiten Variante wesentlich geringer. Gerade in Anwendungen mit teilweise sehr vielen Domänen sollte der Aufwand nicht unterschätzt werden. Der Nutzen, der sich durch die beiden Ansätze ergibt ist, dass die einmal erstellten Konfigurationsdateien nur angepasst oder erweitert werden müssen, wenn sich die Eigenschaften der Domäne ändern oder weitere Eigenschaften hinzugefügt werden. Spätestens bei der Implementierung muss entschieden werden, welche der sinnvollste Ansatz für eine Umsetzung ist, beziehungsweise welcher Ansatz den meisten Einsatz verspricht ist. Bei dem vorliegenden Konzept wurde sich für den zweiten Ansatz entschieden. Die Analyse der MinD.banker Anwendung hat gezeigt, dass der Aufwand für die Erstellung aller Konfigurationsdateien zu aufwendig wäre, da nur ein relativ kleiner

Teil der Konfigurationsdateien aktiv verwendet werden würde.

Alle Konfigurationsdateien leiten sich aus dem Abhängigkeitsmodell ab. Dabei stellen Konfigurationsdateien nur die vorhandenen Abhängigkeiten zwischen operativen Daten von Domänen und den Analysesegmenten der Analysedaten dar. Die abhängigen Analysesegmente werden in den Konfigurationsdateien nicht betrachtet. In jeder Konfigurationsdatei wird konkret beschrieben, welche Abhängigkeiten in der jeweiligen Domäne bestehen. Dabei werden für jede Abhängigkeit der Domäne folgende Eigenschaften festgelegt:

- Auslöser einer Datenänderung: definiert ein konkretes Ereignis, z.B. das Ändern des Kapitalsaldos eines Kunden
- abhängige Analysesegmente: Segmente die nach einer Datenänderung aktualisiert werden müssen
- Art der Aktualisierung: legt fest, wie die Aktualisierung durchgeführt werden soll, z.B. kundenbezogen oder kundenübergreifend

Im ersten Schritt enthalten die Konfigurationsdateien nur die direkt abhängigen Analysesegmente. Nachdem die Konfigurationsdatei mit der Abhängigkeitsdatei der Analysedaten transformiert wird, beinhaltet die generierte Konfigurationsdatei alle vom auslösenden Ereignis abhängigen Segmente der Analysedaten. Aus diesen Konfigurationsdateien werden alle doppelten abhängigen Segmente und zirkulären Abhängigkeiten zwischen den Segmenten entfernt.

Werden an einer Domäne Anpassungen und Erweiterung an Eigenschaften einer Domäne vorgenommen, müssen sich diese Änderungen gegebenenfalls in den entsprechenden Konfigurationsdateien niederschlagen. Nach einer Änderungen an den Konfigurationsdateien muss der gesamte Prozess von der Erzeugung der Konfigurationsdateien mit allen Abhängigkeiten bis zur Code-Generierung wiederholt werden.

4.3.2 Abhängigkeitsdatei der Analysedaten

In Abschnitt 4.2 wurde das vollständige Abhängigkeitsmodell vorgestellt und herausgestellt, dass sowohl die Abhängigkeiten zwischen den operativen Daten und den Analysedaten als auch zwischen unterschiedlichen Analysedaten zu erfassen sind. Nachdem wir schon die Abhängigkeiten zwischen den operativen Daten und Analysedaten mit Hilfe der Konfigurationsdateien umgesetzt haben, wollen wir uns in diesem Abschnitt mit den Abhängigkeiten zwischen den Analysedaten beschäftigen. Diese Abhängigkeiten werden in eine Abhängigkeitsdatei ausgelagert. Die Gründe für die Trennung des Abhängigkeitsmodells in zwei Teilaspekte bestehen darin einen klaren strukturierten Überblick über die verschiedenen Abhängigkeiten zu erhalten und die Erweiterbarkeit und Wartbarkeit zu erhöhen. Durch den Überblick wird für den Entwickler deutlich erkennbar, welche Abhängigkeiten zwischen den Daten bestehen. Aufgrund der Trennung in Konfigurationsdateien und Abhängigkeitsdatei muss bei einer Änderung nur eine Datei, entweder Konfigurationsdatei oder Abhängigkeitsdatei, angepasst werden. Wenn alle abhängigen Analysesegmente in den

Konfigurationsdateien enthalten wären, müsste bei einer Anpassung eines Analysesegments alle davon betroffenen Konfigurationsdateien angepasst werden. Bei einer Vielzahl von Konfigurationsdateien können die Änderungen schnell unübersichtlich werden. Zudem sollte auch der Aufwand der Änderungen nicht unterschätzt werden. Des Weiteren beinhaltet dieses Vorgehen eine große Fehlerquelle, da Konfigurationsdateien vergessen werden könnten.

Die Abhängigkeitsdatei der Analysedaten enthält für jedes Analysesegment die direkten Abhängigkeiten zu anderen Analysesegmenten. Um für ein Analysesegment alle abhängigen Analysesegmente zu erhalten, müssen von den direkten Abhängigkeiten wiederum die abhängigen Analysesegmente in die Menge der abhängigen Analysesegmente übernommen werden. Diese vollständige Menge der abhängigen Analysesegmente wird in dem Abhängigkeitsmodell nicht abgebildet, sondern mit Hilfe des Transformators1 erzeugt.

4.3.3 Template

Um aus einem Modell mittels einer Transformation einen Programmcode zu erzeugen, wird ein Template benötigt, aus dessen Basis der generierte Code beruht. Im Rahmen dieses Konzepts wurde sich für das Java Emitter Template (JET) entschieden. Dieses Java Emitter Template ist Bestandteil des Eclipse Modelling Frameworks, welches ein quelloffenes Java-Framework zur automatisierten Erzeugung von Code anhand von strukturierten Modellen ist. Das Framework bietet eine Vielzahl an Techniken zur automatischen Generierung von Code an. Eine dieser Techniken ist das Java Emitter Template, welches die Möglichkeit für das Erzeugen von beliebigem Output wie Java, XML oder SQL bereitstellt. Zur Implementierung wird die Version 1.x des Java Emitter Templates verwendet. Von Java Emitter Template existiert schon die Version 2.x, die Vorteile im Handling mit Template-Datei bietet. Der Grund, warum die Implementierung in der älteren Version umgesetzt wurde, liegt darin, dass für den Einsatz des Java Emitter Template in der neueren Version die Entwicklungsumgebung Eclipse mit einer kompatiblen Version des Eclipse Modelling Frameworks notwendig ist. Somit können die Methoden des Java Emitter Templates nur innerhalb der Entwicklungsumgebung Eclipse verwendet werden. Damit wird die Verwendung von Java Emitter Templates außerhalb von Eclipse ausgeschlossen. In der 1.x Version besteht die Möglichkeit die Generierung von Code aus einem Template ohne die Verwendung von Eclipse durchzuführen.
Aufgrund der automatischen Generierung von Teilen des Quellcodes wird das Schreiben von redundantem Code reduziert und gleichzeitig die fehleranfällig minimiert. Das Java Emitter Template basiert auf den Konzepten von Java Server Pages und ist an die Syntax dieser Sprache angelehnt, was es einfach macht, aus den Templates den gewünschten Code zu erzeugen.

Komponenten des Templates

Das Template besteht aus drei Komponenten: [SS05]

- Direktiven (Umgebungsparameter)

- Scriptlets (Java-Code-Fragmente)
- Expressions (Java-Ausdrücke)

Die Direktiven können in zwei Arten unterschieden werden, die JET-Direktive und die Include-Direktive. In der Implementierung werden nur die JET-Direktiven verwendet. Aus diesem Grund wird die Include-Direktive betrachtet. In der JET-Direktive werden die Umgebungsparameter, die für die Übersetzung eines Templates benötigt werden definiert. Darunter zählt unter anderem das Ort (`package`) an dem die übersetzte Implementierungsklasse abgelegt wird, der Klassenname (`class`) und die benötigten Klassen (`imports`). Darüber hinaus können weitere Einstellungen für die Übersetzung angegeben werden. Die JET-Direktive wird innerhalb von `<%@jet` und `%>` definiert.
Die Expressions sind Java-Ausdrücke, die Ausgaben von Werten einer Variablen oder Methode bereitstellen. Sie stellen somit die Platzhalter innerhalb des Templates dar. Die Auswertung dieser Expressions erfolgt nach dem Aufruf der `generate()`-Methode einer kompilierten Implementierungsklasse. Eine Expression wird innerhalb `<%=` und `%>` beschrieben.
Scriplets stellen Code-Fragmente, die jeden beliebigen gültigen Java-Code enthalten, dar. Jeder Syntaxfehler innerhalb des Java-Codes führt zu einem Fehler in der Implementierungsklasse. In den Scriptlets wird der Großteil der Logik definiert. Die Ausführung der Scriplets wird wie bei den Expression nach dem Aufruf der `generate()`-Methode durchgeführt. Ein Scriptlet wird mit `<%` begonnen und mit `%>` beendet [SS05].

Ablauf des Parsen eines Templates

Der Ablauf des Parsen eines Templates wird in folgenden Schritten gegliedert.

1. JET-Direktiven analysieren
2. Scriplets analysieren und übersetzen
3. Expressions analysieren und übernehmen

Nachdem das Parsen des Templates abgeschlossen ist, kann die Übersetzung des Templates in eine Implementierungsklasse erfolgen. Aus dieser Implementierungsklasse wird in der Instanziierung der generierte Code erzeugt.

4.3.4 Transformatoren

Das Modell besteht aus zwei Transformatoren. Diese beiden Transformatoren müssen nacheinander ausgeführt werden, da der zweite Transformator das Ergebnis der ersten Transformation als Eingabe erhält.

Transformator1

Die Aufgabe der ersten Transformation des Modells ist die Umwandlung der Konfigurationsdatei unter Berücksichtigung der Abhängigkeitsdatei der Analysedaten in ein Zwischenmodell. Dieses Zwischenmodell dient als Grundlage für die Code-Generierung des zweiten Transformators. Diese Funktion führt der Transformator für alle vorhandenen Konfigurationsdateien durch. Aus diesem Grund benötigt der Transformator einen Pfad, unter welchem die Konfigurationsdateien zu finden sind. Deshalb sollte, wenn möglich, versucht werden, alle Konfigurationsdateien zentral an einem Ort innerhalb der Anwendung abzulegen. Unter der Verwendung von Transformationsregeln wird jede Konfigurationsdatei mit allen erforderlichen abhängigen Segmenten der Analysedaten erweitert. Die Reihenfolge der Anwendung der Transformationsregeln ist in diesem Fall determiniert. Damit ist gemeint, dass die Reihenfolge der Ausführung der Transformationsregeln völlig irrelevant ist. Wichtig hingegen ist, dass der bei der Transformation immer dasselbe Ergebnis herauskommt. Auf unsere Situation bezogen bedeutet dies, dass es unbedeutend ist, in welcher Abfolge die Transformationsregeln ausgeführt werden, es jedoch immer die selben abhängigen Segmente für ein Datensegment sein müssen.

Der Vorgang der Transformation kann so beschrieben werden, dass im ersten Schritt die Menge der abhängigen Segmente nur aus einem Element besteht. Dieses Element ist das Segment aus der Konfigurationsdatei, das in Abhängigkeit zum auslösenden Ereignis steht. Anhand der Abhängigkeitsdatei werden alle zu diesem Segment abhängigen Segmente ermittelt und zur Menge der abhängigen Segmente hinzugefügt. Im nächsten Schritt wird das nächste Element aus der Menge betrachtet und erneut werden alle abhängigen Segmente ermittelt. Dieser Schritt wird solange wiederholt, bis keine abhängigen Segmente mehr ermittelt werden konnten, da alle benötigten Elemente schon in der Menge enthalten sind. Um zirkuläre Effekte zu verhindern und zu erreichen, dass der Transformator immer terminiert, werden die abhängigen Segmente nur einmal in die Menge aufgenommen. Wenn abhängige Datensegmente ermittelt werden, die schon in der Menge enthalten sind, werden diese Segmente ignoriert. Dieser beschriebene Vorgang zur Ermittelung der abhängigen Segmente ist in der Abbildung 4.4 verdeutlicht.

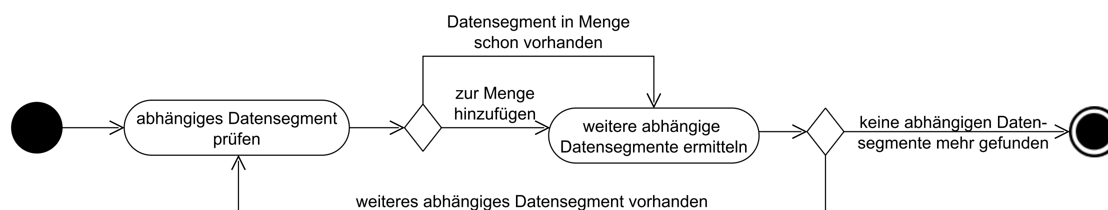

Abbildung 4.4: Vorgang zur Ermittelung der abhängigen Segmente

Transformator2

Der Transformator2 generiert aus dem Template und einer Konfigurationsdatei mit allen abhängigen Datensegmenten den entsprechenden Code. Als Template wird

das in dem Abschnitt 4.3.3 geschriebene Java Emitter Template verwendet. Dieses Template ist nicht nur ein reines Textdokument mit Platzhaltern, sondern beinhaltet zusätzlich einen Builder, der die Generierung des Codes durchführen kann. Der Prozess der Code-Generierung durch den Builder lässt sich in zwei Abschnitte, der Übersetzung und Generierung, zerlegen. Der Ablauf der Template-Generierung wird in Abbildung 4.5 dargestellt.

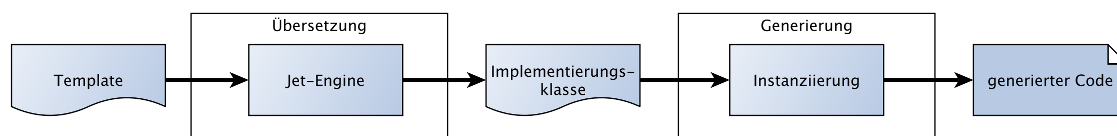

Abbildung 4.5: Prozess der Template-Generierung

Zunächst wird vom Entwickler ein Template entworfen, das den Code in dem gewünschten Output-Format enthält. Die Java Emitter Templates stellen keine Vorlagen für die Templates zur Verfügung, da sich die Templates immer individuell an der geforderten Zielsetzung orientieren. Deshalb muss, wenn durch die Generierung verschiedene Output-Formate erzeugt werden sollen, für jeden gewünschten Output ein Template erstellt werden.
Im ersten Teilschritt wird dieses Template von der JET-Engine interpretiert und in eine Implementierungsklasse übersetzt. Dabei verwendet die JET-Engine ein Skeleton als Vorlage für diese Implementierungsklasse. An dieser Stelle erfolgt das Parsen des Templates, das aus der Beschreibung des Templates im oberen Abschnitt entnommen werden kann. Das Resultat des Parsens ist eine Java-Datei, die Implementierungsklasse. Falls diese Implementierungsklasse schon durch einen vorherigen Generierungsprozess vorhanden ist, wird sie überschrieben. Die Implementierungsklasse enthält den Code des Templates in aufbereiteter Form für den nächsten Schritt. Um aus der erzeugten Implementierungsklasse den gewünschten Code zu generieren, muss die Implementierungsklasse instanziiert werden. Auf dieser Instanz wird daraufhin die `generate`-Methode, die `java.lang.Object` als Parameter erwartet, aufgerufen. Über den Parameter können zusätzlich Einstellungen oder Objekte, die zur Generierung verwendet werden sollen, festgelegt werden. Das Ergebnis des Generators ist der erzeugte Code, der als String zurückgeliefert wird. Dieser Code kann beliebig weiter verarbeitet werden. In diesem Fall wird der generierte Code in einer `java`-Datei gespeichert. Nachdem die Generierung des Codes abgeschlossen ist, wird die Implementierungsklasse nicht mehr benötigt und kann manuell gelöscht werden.

4.4 Zusammenfassung

Dieses Kapitel stellt das Konzept zur feingranularen Änderungserkennung vor. Dazu wurde der Konzeptentwurf mit Hilfe eines Modells dargestellt. Um die Komponenten des Modells ausführlich beschreiben zu können, wurde vorher das benötigte Abhängigkeitsmodell skizziert. Im Anschluss daran werden Komponenten und deren Beziehung detailliert erläutert. Dieses Konzept bildet die Grundlage für den im nächsten Kapitel beschriebenen Prototyp.

5. Prototypische Implementierung am Beispiel von MinD.banker

Das Konzept (Kapitel 4) soll in diesem Kapitel im Rahmen einer prototypischen Implementierung am Beispiel von MinD.banker verfeinert werden. Dafür wird zunächst ein kleiner Einblick in die zur Umsetzung der lauffähigen Prototyps verwendeten Software MinD.banker gegeben. Im Anschluss wird der Prototyp detailliert vorgestellt. Die Umsetzung des Prototyps erfolgt mit der Programmiersprache Java, da MinD.banker in dieser Programmiersprache geschrieben ist.

5.1 MinD.banker

Auf Grundlage des Konzeptentwurfs wurde im Rahmen des Praktikums die prototypische Implementierung am Beispiel von MinD.banker realisiert. Das Praktikum wurde in der Eudemonia Solutions AG, verantwortlich für die Entwicklung der Produkte der MinD-Familie, durchgeführt. Um den Prototyp umsetzen zu können, ist es notwendig, diesen in eine reale IT-Systemlandschaft einzubetten. Dieser Rahmen bietet sich bei der Portierung von MinD.banker Classic auf MinD.banker WAVE an. Mit MinD.banker WAVE, einer Softwarelösung für das Kundengeschäft in den Banken, vollzieht sich der zukunftsweisende Wechsel von einer Desktopanwendung, die auf jedem Arbeitsplatz installiert ist, hin zu einer Webanwendung, die auf einem Web-Server im Serverzentrum oder in nächster Zukunft sogar in der "Cloud" gehostet wird. Durch die Umstellung auf die Webfähigkeit besteht für die Bankberater die Möglichkeit mit jedem internetfähigen Endgerät (neben dem PC auch Smartphones und Tablets) auf die Software zuzugreifen und damit zu arbeiten.
Durch die Ablösung von MinD.banker Classic durch MinD.banker WAVE soll in der Zukunft dank der erweiterten Möglichkeiten erreicht werden, dass Datenänderungen jederzeit durchgeführt werden können und dadurch immer mehr Datenänderungen auftreten können. Auch müssen kleine Änderungen möglichst zeitnah auf die Analysedaten angewandt werden, da man mobil drauf zugreifen will.
Für den Prototyp wird die Programmiersprache Java eingesetzt, da MinD.banker WAVE ebenso fast komplett in Java implementiert ist. In MinD.banker werden zwei

Methoden schon eingesetzt, und zwar das objektrelationale Mapping und die aspektorientierte Programmierung. Das objektrelationale Mapping wird zum Laden und Speichern von Objekten in relationalen Datenbanktabellen und die aspektorientierte Programmierung wird zum Steuern und Kontrollieren von Transaktionen und zum Sperren von Kunden verwendet. Keine dieser Techniken wird jedoch zur Überwachung der Datenänderungen genutzt, weshalb wir uns bei der Umsetzung an keinem existierenden Teilsystem orientieren müssen. Aus diesem Grund besteht die Möglichkeit alle vorgestellten Methoden zu verwenden.

5.2 Umsetzung

Wir haben uns entschieden, die Umsetzung des vorliegenden Konzepts nur in MinD. banker WAVE zu realisieren, weil in den Banken in nächster Zeit sukzessiv MinD. banker Classic durch MinD.banker WAVE ersetzt wird. Um eine effiziente Datenanalyse in MinD.banker WAVE zu gewährleisten, wird eine Analyse der fachlichen Abhängigkeiten zwischen den operativen Daten und den Analysedaten durchgeführt. Diese Analyse wurde in zwei Schritten absolviert. Im ersten Schritt erfolgte eine grobe Untersuchung auf allen Objekten des Classic-Codes. Dabei wurden alle Objekte ermittelt, die nach einer Datenänderung eine Aktualisierung der Analysedaten auslösen müssen. Dies bildete die Grundlage für den zweiten Schritt, der feingranularen Analyse auf Objektebene. In diesem Prozess wurde für jedes ermittelte Objekt detailliert beschrieben, welche Attribute beziehungsweise Eigenschaften zur Aktualisierung der Analysedaten führen. Die Ergebnisse dieser Analyse wurden umfassend dokumentiert und im Anschluss daran ausgewertet, um zu entscheiden, welche Methode zur Erkennung von Datenänderungen in MinD.banker WAVE den größtmöglichen Nutzen bietet.

Auswahl des Ansatzes zur Erkennung von Datenänderungen

Bei der Umsetzung des Prototyps wurde sich die Frage gestellt, welcher Ansatz zur Erkennung von Datenänderungen eingesetzt werden sollte. Die Wahl fiel letztendlich auf den aspektorientierten Ansatz, weil er aus unserer Sicht die größtmögliche Flexibilität liefert. Mit dem aspektorientierten Ansatz kann auf alle Stellen innerhalb des Codes zugegriffen werden und zusätzlich der Bereich, aus dem die Datenänderungen durchgeführt wurden, bestimmt werden. Dies ist deshalb relevant, weil die Analyse des Ist-Zustandes ergeben hat, dass die Datenänderungen aus verschiedenen Bereichen zu unterschiedlichen Typen der Aktualisierung führen können. In MinD.banker kann zwischen drei Typen der Aktualisierung der Analysesegmente unterschieden werden. Der Typ beschreibt, in welchem Kontext die Aktualisierung durchzuführen ist. Dabei legt der Kontext fest, dass die Aktualisierung nur für den zugehörigen Kunden, den zugehörigen Kunden und seiner Engagements oder für alle Kunden vorzunehmen ist. Für eine effiziente Aktualisierung der Analysedaten ist es deshalb essentiell wichtig, den Bereich zu bestimmen, aus dem die Datenänderung durchgeführt wird, um nicht unnötige Analysedaten von Kunden aktualisieren zu müssen.
Bei den Eventlistener in Hibernate existiert leider das Problem, dass man in der Event-Methode des Eventlistener nur auf `Plain Old Java Objects`, Objekte, die auf eine Tabelle in einer relationalen Datenbank abbilden, beschränkt ist. Damit

lassen sich keine Datenänderungen an transienten Objekten kennen. Ein weiteres Problem ist, dass die Datenänderungen zwischen dem alten Objekt und dem neuen Objekt auf Attributebene zwar erkannt werden, die Änderungen jedoch keine Auskunft darüber liefern in welchem Bereich beziehungsweise von wem diese Datenänderungen durchgeführt wurde.
Gegen die Umsetzung mit einen Datenbanktrigger spricht, wie bei den Eventlistener der Punkt, der Bestimmung des Änderungsbereiches. Des Weiteren kann durch den Einsatz eines Triggers die Performance der Datenbank erheblich beeinträchtigt werden.
Zusammenfassend lässt sich festhalten, dass wir uns aufgrund der hier genannten Nachteile von Eventlistener in Hibernate und Datenbanktriggern für die Nutzung der aspektorientierten Programmierung entschieden haben.

In den nachfolgenden Kapiteln wird die Umsetzung der Komponenten des Modells einzeln beschrieben.

5.2.1 Konfigurationsdatei

Eine Konfigurationsdatei stellt alle Abhängigkeitseinstellungen, die in der jeweiligen Domäne bestehen, zur Verfügung. Für die Konfigurationsdateien und die Abhängigkeitsdatei wird die domänenspezifische Modellierungssprache XML verwendet. XML zeichnet sich dadurch aus, dass die Struktur und der Aufbau der Daten mit Hilfe von Elementen detailliert beschrieben werden kann. Diese Elemente (Tags) unterliegen keinem Standard und können demzufolge frei definiert werden. Ein weiterer Vorteil von XML ist, dass die Darstellung in einer strukturierten, hierarchischen Form erfolgt und so vergleichsweise leicht von Menschen und Maschinen gelesen und weiterverarbeitet werden kann. Die Semantik sowie die Grammatik einer XML-Datei wird erst über eine Dokumenttypdefinition oder ein XML-Schema vorgegeben. Mit dem Einsatz eines XML-Schemas, das die Regeln enthält, welche Elemente und Attribute erforderlich oder zulässig sind und in welcher Reihenfolge sie stehen müssen beziehungsweise dürfen, kann die Gültigkeit und Vollständigkeit der XML-Datei gewährleistet werden. In der oben aufgelisteten XML-Datei wurde auf solch ein XML-Schema verzichtet, weil es sich um einen Prototypen handelt und sich bis zum Endprodukt durchaus Änderungen an der Struktur und dem Aufbau ergeben könnten.

Elemente der Konfigurationsdatei

Die wichtigsten Elemente der Konfigurationsdatei sind:

- `class`, zugehörige Klasse der DomainObjects
- `objectId`, Provider für die Ermittlung des zum DomainObject gehörigen Kunden
- `method`, auslösende Methode zur Erkennung von Datenänderungen
- `dataSegments`, beinhaltet alle Analysesegmente, die durch die auslösende Methode aktualisiert werden sollen

- `dirtySettings`, beschreibt der Art die Aktualisierung
- `name`, beschreibt den Wert des Attribut-Tags

Nachfolgend wird auf diese Elemente im Rahmen des Beispielcode 5.1 genauer eingegangen. Dabei präsentiert der Beispielcode eine Konfigurationsdatei für das DomainObject `Account`.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
  <class name="de.esag.mind.banker.wave.server.domain.Account">
    <objectId name="de.esag.mind.banker.wave.server.domain.trans.
                CustomerIdProvider"/>

    <method name="setBalance">
      <dataSegments>
        <dataSegment name="DataSegment.DS_ASSETLIABILITY"/>
      </dataSegments>
      <dirtySettings>
        <dirtySetting name="CustomerDirtyType.
                CUSTOMER_AND_PARENT_ENGAGEMENTS_DIRTY"
         includePackages="de.esag.mind.banker.wave.server.imports.."/>
        <dirtySetting name="CustomerDirtyType.CUSTOMER_DIRTY"
         excludePackages="de.esag.mind.banker.wave.server.imports.."/>
      </dirtySettings>
    </method>
    ...
  </class>
</configuration>
```

Listing 5.1: Konfigurationsdatei für ein Account-Objekt

Das Wurzelelement `configuration` kann eine Menge von `class`-Objekten enthalten. In der aktuellen Implementationsstufe ist die Konfigurationsdatei immer auf eine Klasse eines bestimmtes DomainObject beschränkt. Es ist durchaus denkbar, eine Konfigurationsdatei zu erstellen, die alle abgeleiteten DomainObject-Klassen beinhaltet, anstatt für jede Klasse eine eigene Konfigurationsdatei zu erzeugen. Innerhalb dieses `class`-Tags müssen zwei weitere Tags definiert werden. Zum einem der Provider und zum anderen die Menge von Auslösern. Bei dem Provider handelt es sich um eine Java-Klasse, die die Aufgabe hat, den vom DomainObject zugehörigen Kunden zu ermitteln. Auf diesen Provider wird im Zuge der Verarbeitung des durch den Transformator2 generierten Codes detaillierter eingegangen. Die Einstellungen, die zur Erkennung von Datenänderungen und anschließender Aktualisierung der Analysedaten benötigt werden, sind innerhalb des `method`-Block definiert. Im Rahmen des Prototyps wurde entschieden, die Erkennung von Datenänderungen anhand von Methodenaufrufen zu organisieren. Dies bedeutet, für jede Methode eines Objektes, die in Abhängigkeit zu einem Analysesegment steht, muss ein `method`-Block erzeugt werden. Jeder `method`-Block ist eindeutig durch einen Namen gekennzeichnet. Innerhalb dieses Blockes wird festgestellt, welche Analysesegmente durch die Änderung betroffen sind (`dataSegments`) und welche Art der Aktualisierung durchgeführt werden soll (`dirtySettings`). In diesem Beispielcode soll bei

Änderungen der Kredithöhe beziehungsweise des Kapitalsaldo an einem Account-Objekt durch Aufruf der `setBalance()`-Methode eine Aktualisierung des Analysesegments (*DS_ASSETLIABILITY*) ausgelöst werden. Zusätzlich ist angegeben, welche Art der Aktualisierung für welchen Bereich durchgeführt werden soll.
Die unterschiedlichen Aktualisierungsstrategien bei Änderungen an den DomainObjects in verschiedenen Bereichen der Anwendung werden in einem `dirtySetting`-Block angegeben. Dafür werden die beiden Element `includePackages` und `excludePackages` verwendet. Mit `includePackages` wird angegeben, dass die Aktualisierung der Analysesegmente nur aus diesem Bereich mit diesem Aktualisierungstyp durchgeführt werden darf. Das Gegenteil dazu bietet `excludePackages`, mit dem man bestimmte Bereich ausschließen kann. Bezogen auf das Listing 5.1 bedeutet dies, dass, wenn die Datenänderungen im Bereich des Imports durchgeführt werden, immer die Analysedaten für den Kunden und seiner Engagements aktualisiert werden. Für alle anderen Bereiche gilt, dass die Analysedaten nur für den Kunden aktualisiert werden.

5.2.2 Abhängigkeitsdatei

Die Abhängigkeitsdatei enthält für jedes Analysesegment, die direkten Abhängigkeiten zu anderen Analysesegmenten. Ebenso wie die Konfigurationsdateien wird diese Abhängigkeitsdatei, manuell vom Entwickler angelegt. Dafür wird für jedes Analysesegment ein Eintrag, Tag-Element `datasegment`, in der Abhängigkeitsdatei angelegt. Innerhalb dieses Tags wird die Menge der abhängigen Analysesegmente, Tag-Element `affectedDataSegments` festgelegt. Für die Analysesegmente werden Konstanten verwendet, die zu einer Enumeration zusammengefügt werden.

In dem nachfolgenden Listing 5.2 wird ein Ausschnitt der Abhängigkeitsdatei dargestellt. In diesem Ausschnitt wird das im Listing 5.1 von der `setBalance`-Methode abhängige Analysesegment *DS_ASSETLIABILITY* betrachtet. Dieses Analysesegment stellt eine Übersicht über das Vermögen und die Schulden der Kunden bereit. Dazu gehören unter anderem die Saldo-Werte verschiedener Konten, die Kreditbeträge oder die aktuellen Verkehrswerte vorhandener Grundstücke. Die abhängigen Analysesegmente der Vermögens- und Schuldensübersicht sind die betriebliche und private Kapitaldienstfähigkeit (*DS_BKDFVALUES*, *DS_PKDFVALUES*), die Segmentierung (*DS_SEGMENTATION*) und die Produktvorschläge (*DS_PRODUCTSUGGESTIONS*). Die Änderung eines Wertes in der Vermögens- und Schuldensübersicht hat zur Folge, dass diese Analysesegmente ebenfalls aktualisiert werden müssen. Werden diese Segmente nicht aktualisiert, besteht die Möglichkeit, dass die Kunde im nachfolgenden Verlauf falsch beraten werden, weil die Berater auf veralteten Analysedaten arbeiten.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <dataSegments>
4    <dataSegment name="DataSegment.DS_ASSETLIABILITY">
5      <affectedDataSegments>
6        <dataSegment name="DataSegment.DS_BKDFVALUES"/>
7        <dataSegment name="DataSegment.DS_PKDFVALUES"/>
8        <dataSegment name="DataSegment.DS_SEGMENTATION"/>
9        <dataSegment name="DataSegment.DS_PRODUCTSUGGESTIONS"/>
```

```
10        </affectedDataSegments>
11      </dataSegment>
12      ...
13    <dataSegments>
```

Listing 5.2: Abhängigkeitsdatei

5.2.3 Template

Mit dem Java Emitter Template (JET) lassen sich Teile des Quellcodes automatisch generieren. Dabei ist JET in der Lage, jede Art von Output - SQL, Java, XML oder Text zu generieren [SS05]. In der vorliegenden Implementierung wird als Output Java-Code erzeugt. Dieser Java-Code repräsentiert einen aspektorientierten Aspekt mit einer Vielzahl von Pointcuts, Join Points und Advices.
Bei dem Template handelt es sich um eine mit Platzhaltern versehene Textvorlage. Diese Platzhalter werden während der Generierung des Codes durch die Einstellungen aus den Konfigurationsdateien ersetzt.
In dem Abschnitt 4.3.3 wurden die Komponenten (Direktiven, Scriptlets, Expressions) des Templates vorgestellt. Zum Verständnis der Template-Datei werden weitere Grundlagen benötigt. Diese Grundlagen werden in zwei Abschnitten kurz erläutert.

Skeleton-Datei

Bei der Übersetzung des Templates in eine Implementierungsklasse wird immer eine Skeleton-Datei benötigt. Diese Skeleton-Datei bildet die Vorlage für die Implementierungsklasse und definiert dazu die `generate()`-Methode. Die Skeleton-Datei besteht aus Java-Sprachkonstrukten und kann somit als Java-Klasse angesehen werden, mit Ausnahme des Klassennamens. Der Klassenname wird bei der Übersetzung durch den Wert das `class`-Attribut aus den JET-Direktive ersetzt. Im Java Emitter Template existiert für diese Skeleton-Datei eine Standard-Implementierung. In dieser Standard-Skeleton-Datei ist die `generate()`-Methode, die ein `java.lang.Object` als Parameter erwartet, implementiert. Neben der Standard-Skeleton-Datei besteht die Möglichkeit eine eigene Skeleton-Datei zu erzeugen. In dieser Skeleton-Datei muss eine `generate()`-Methode mit einem Parameter deiner Wahl definiert werden. Durch eine eigene Skeleton-Datei kann die Implementierungsklasse individuell angepasst werden, zum Beispiel durch Implementation eines Interface oder Erweiterung der Klasse um beliebige Methoden, die nur zur Generierung des Codes benötigt werden.
Um eine eigene Skeleton-Datei zu verwenden, muss sie in der JET-Direktive des Templates über das optionale Attribut *skeleton* unter dem Namen der Skeleton-Datei angegeben werden. Für den Fall, dass innerhalb der JET-Direktive keine Skeleton definiert ist, wird die Standard-Skeleton-Datei verwendet.

DirtyConfiguration

Eine Möglichkeit, um die Einstellungen (Auslöser der Datenänderung, abhängige Analysesegmente und Art der Aktualisierung) aus einer Konfigurationsdatei im Template verwenden zu können, bietet die Erstellung einer Konfigurationsklasse (`DirtyConfiguration`). Diese Konfigurationsklasse bildet alle Einstellungen einer

Konfigurationsdatei eins zu eins ab. Dieser Zwischenschritt ist notwendig, weil in der verwendeten Version des Java Emitter Template, die Transformation mit XML-Dateien noch nicht unterstützt wird. Aus diesem Grund können in den Expressions oder Scriptlets nicht direkt auf die Werte der XML-Dateien zugriffen werden.
Diese Konfigurationsklasse ermöglicht eine einfachere und fehlerfreie Bearbeitung innerhalb des Templates. In dem Listing 5.3 wird die `DirtyConfiguration`-Klasse, die korrespondierenden Attribute und zusätzlich zum dargestellten auch noch je Attribut eine `get`- und `set`-Methode enthält, dargestellt.

```
public class DirtyConfiguration
{
    private String className;
    private String packageName;
    private String customerIdProvider;
    private Map<String, Map> methods = new HashMap<String, Map>();
    ...
}
```

Listing 5.3: DirtyConfiguration

Um diese `DirtyConfiguration`-Klasse im Template verwenden zu können, muss entweder eine eigene Skeleton-Datei verwendet werden oder im Template der Parameter des Objektes `java.lang.Object` in ein `DirtyConfiguration`-Objekt gecastet werden. Bei der Umsetzung wurde sich entschieden, eine eigene Skeleton-Datei (*DirtyInterceptor.skeleton*) zu verwenden. Zusätzlich zu den Methoden für die Geschäftslogik, die zur Bearbeitung des Templates notwendig sind, wurde in der Skeleton-Datei eine `generate()`-Methode mit einem `DirtyConfiguration`-Objekt als Parameter definiert.

Template-Datei

Nachdem die erforderlichen Grundlagen für die Template-Datei ausgeführt wurde, wird im nachfolgenden Beispielcode die Template-Datei dargestellt.

```
1  <%@ jet imports="java.util.* de.esag.wavefmk.transformator.* ..."
2   skeleton="DirtyInterceptor.skeleton"
3  %>
4  ...
5  @Aspect
6  public class <%=configuration.getClassName()%>
7  {
8    ...
9  <%
10 for (Map.Entry<String, Map> method : configuration.getMethods().
11                                                     entrySet())
12 {
13   Map<String, Object> values = method.getValue();
14   String adviceName = getAdviceName(method.getKey());
15   String methodName = getMethodName(className+"."+method.getKey(),
16          values);
17   String packageJoinPointExecution = getJoinPointExecution(values);
18 %>
19   @Around("cflow(execution(* <%=packageJoinPointExecution%>(..)))
20     && call(* <%=methodName%>(..))")
```

```
21   public Object <%=adviceName%>(ProceedingJoinPoint pjp)
22         throws Throwable
23   {
24     DataSegment[] dataSegments =
25         new DataSegment[]{<%=getDataSegmentsAsString(values)%>};
26     return setDirty(pjp, getDomainObject(pjp.getTarget()),
27         pjp.getSignature().toLongString(), dataSegments,
28         CustomerDirtyType.<%=setting.getDirtyType()%>);
29   }
30   ...
31 <%}%>
32 }
```

Listing 5.4: Template-Datei

Das Listing 5.4 zeigt auszugsweise ein Template, das die Grundlage für die generierten Dateien legt. Im Template wird für alle Methoden des übergebenen Parameters `configuration` analog zu den Einstellungen Textvorlagen für Pointcuts und Advices definiert. Die erste Zeile eines Templates muss zwingend eine JET-Direktive sein. Die JET-Direktive wird innerhalb von `<%@jet` und `%>` definiert. Diese JET-Direktive enthält die für die Implementationsklasse, benötigten Umgebungsparameter, wie den Klassennamen, die Importe und die Skeleton-Datei. Der tatsächliche Code, der sich in den generierten Dateien widerspiegelt, ist ab der Zeile fünf definiert. Innerhalb des Templates wechseln sich Textelemente, Scriptlets und Expressions gegenseitig ab. Im generierten Code sind nur die Textelemente und die durch den Platzhalter ersetzten Werte enthalten. Die Expressions können als Platzhalter, die Ausgaben von Werten einer Variablen oder Methode bereitstellen, angesehen werden. Im ersten Textelement wird die zu erzeugende Klasse als Aspekt (Annotation `@Aspect`) definiert. Ab der Zeile 19 sind die Textelemente für einen Pointcut und einen Around-Advice beschrieben. Diese Textelemente sind von einem Scriptlet umschlossen. Dieses Scriptlet iteriert über alle Methoden des `configuration`-Objekts. Dadurch werden im generierten Code für jede Methode diese Textelemente mit den hinterlegten Einstellungen erzeugt. Der Pointcut beinhaltet die Ausführung aus einem bestimmten Package der MinD.banker-Anwendung. Bei der Umsetzung wird sich für den Around-Advice entschieden, weil der Advice selbst den Aufruf des Join Points steuern kann. Der große Vorteil dieses Advices ist, dass man sehr flexibel bestimmen kann, wann der Join Point aufgerufen wird. Dadurch besteht die Möglichkeit vor und nach Ausführung des Join Points bestimmte Aktionen durchzuführen. Auf die Pointcuts und Around-Advices wird bei dem Transformator2 (Abschnitt 5.2.4) nochmal etwas genauer eingegangen.

Wie sich aus diesem kleinen Code-Beispiel erkennen lässt, geht die Übersichtlichkeit durch Abwechseln von Textelementen, Scriptlets und Expressions recht schnell verloren. Dadurch ist es relativ fehleranfällig. Diese Fehler im Template werden erst bei der Übersetzung in die Implementationsklasse oder im generierten Code sichtbar.

5.2.4 Transformatoren

Die Implementation besteht aus zwei Transformatoren. Diese beiden Transformatoren müssen nacheinander ausgeführt werden, da der zweite Transformator das Ergebnis der ersten Transformation als Eingabe erhält.

Transformator1

Dieser Transformator ist für die Generierung von vollständigen Konfigurationsdateien mit allen Abhängigkeiten verantwortlich. Dafür wird für jede manuell vom Entwickler erstellte Konfigurationsdatei aus dem Package (`de.esag.mind.banker.wave.server.dirtyconfig`) die abhängigen Analysesegmente bestimmt und diese einer neuen Konfigurationsdatei mit den vorhandenen Einstellungen zusammengefügt. Dazu werden die Konfigurationsdateien nacheinander mit Hilfe eines XML-Reader aus dem XML-Framework dom4j[1] eingelesen. dom4j ist eine Open-Source-Java-Bibliothek zur Verarbeitung von XML-Dokumenten.
Ein XML-Handler übernimmt das Handling von XML-Dateien. Dazu gehören unter anderem das Laden und Speichern der Konfigurationsdateien und der Abhängigkeitsdatei. Im ersten Schritt wird eine Konfigurationsdatei eingelesen und durch den SAX-Parser in ein `Document`-Objekt umgewandelt. Das `Document`-Objekt ist als Baumstruktur aufgebaut, das in jedem Zweig die zugehörigen XML-Elemente und ihre Werte enthält. In einem `Document`-Objekt kann mit Hilfe von XPath-Ausdrücken navigiert und nach bestimmten Elementen oder Werten gesucht werden. Die `method`-Elemente können mit folgendem XPath-Ausdruck gefunden werden: *`//configuration/class/method`*.
Nach der Navigation wird für jedes `method`-Element im ersten Schritt das abhängige Analysesegment bestimmt und zu einer Menge von abhängigen Analysesegmenten hinzugefügt. Im nächsten Schritt wird die Abhängigkeitsdatei eingelesen und in ein anderes `Document`-Objekt umgewandelt. Danach wird über das `Document`-Objekt iteriert und für alle in der Menge vorhandenen Segmente, die abhängigen Analysesegmente ermittelt und zur Menge hinzugefügt. Dieser Algorithmus wird solange ausgeführt, bis sich die Anzahl der Elemente vom vorherigen zu dem gegenwärtigen Durchlauf nicht mehr ändert. Sobald ein ermitteltes Analysesegment in der Menge schon vorhanden ist, wird es kein weiteres Mal hinzugefügt. Dadurch wird verhindert, dass der Algorithmus nicht terminiert. Nachdem die Ermittlung der abhängigen Analysesegmente abgeschlossen ist, wird im vorletzten Schritt für jeden Eintrag aus der Menge der abhängigen Analysesegmente ein neues Element dem `Document`-Objekt der Konfigurationsdatei hinzugefügt. Im letzten Schritt wird das geänderte `Document`-Objekt unter Verwendung des XML-Writers in eine Konfigurationsdatei geschrieben.
Aus dem Listing 5.1 und 5.2 erzeugt der Transformator die folgende Konfigurationsdatei.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <configuration>
4    <class name="de.esag.mind.banker.wave.server.domain.Account">
5      ...
6      <method name="setBalance">
7        <dataSegments>
8          <dataSegment name="DataSegment.DS_ASSETLIABILITY"/>
9          <dataSegment name="DataSegment.DS_OBLIGO"/>
10         <dataSegment name="DataSegment.DS_INTOBLIGO"/>
11         <dataSegment name="DataSegment.DS_PPAVERAGES"/>
12         <dataSegment name="DataSegment.DS_PKDFVALUES"/>
```

[1]http://dom4j.sourceforge.net/

```
13         <dataSegment name="DataSegment.DS_PRODUCTSUGGESTIONS"/>
14         <dataSegment name="DataSegment.DS_COMPAREGROUPAVERAGES"/>
15         <dataSegment name="DataSegment.DS_PRODUCTPOTENTIALVALUES"/>
16         <dataSegment name="DataSegment.DS_BKDFVALUES"/>
17         <dataSegment name="DataSegment.DS_COMPAREGROUPS"/>
18         <dataSegment name="DataSegment.DS_PRODUCTMARGINCONNECTIONS"/>
19         <dataSegment name="DataSegment.DS_SEGMENTATION"/>
20       </dataSegments>
21       ...
22     </method>
23     ...
24   </class>
25 </configuration>
```

Listing 5.5: generierte Konfigurationsdatei mit allen abhängigen Analysesegmenten

Transformator2

Dieser Transformator wird für die automatische Generierung von Code verwendet. Dabei wird aus einem Java Emitter Template (JET) und einer Konfigurationsdatei ein AOP-Aspekt erzeugt. Wie im Konzept des Transformators beschrieben, wird die Transformation in zwei Schritten durchgeführt. Bevor der erste Schritt der Übersetzung des Templates in eine Implementierungsklasse erfolgt, wird aus einer Konfigurationsdatei ein Konfigurationsobjekt (`DirtyConfiguration`) erstellt. Dafür werden die Einstellungen der Konfigurationsdatei mit Hilfe des beim Transformator1 ebenfalls verwendeten XML-Handler ausgelesen und auf ein `DirtyConfiguration`-Objekt übertragen. Der Aufbau des Konfigurationsobjekts wird Abschnitt 5.2.3 beschrieben. Dieses Konfigurationsobjekt wird für die Generierung des Codes benötigt. Zur Übersetzung des Template wird der Ant-Task[2] von Knut Wannheden verwendet. Dieser Task kompiliert das JET-Template in eine Implementierungsklasse unter Verwendung des Eclipse Modelling Frameworks. Dabei müssen in diesem Task die benötigten Jar-Plugin-Dateien angegeben werden. Eine Anleitung zur Verwendung des Tasks ist unter dem JET-Tutorial[3] zu finden. Der JET-Compiler parst das Template auf die Textelemente und extrahiert diese Textelemente in Klassenvariablen mit fortlaufenden Variablennamen `TEXT_x`. Diese Klassenvariablen werden dem `StringBuffer` an den entsprechenden Stellen innerhalb der `generate()`-Methode angehängt. Die Scriptlets und Expressions werden direkt an den dazugehörigen Stellen im Code übernommen. An dem Ende der `generate()`-Methode wird der Inhalt des StringBuffers als String zurückgegeben [SS05].

In dem Listing 5.6 wird die Java-Implementierungsklasse dargestellt.

```
public class DirtyInterceptor
{
  public final String NL = System.getProperties()
    .getProperty("line.separator");
  ...
  protected final String TEXT_22 = NL + "\t@Around(\"cflow(+ "
    "execution(* ";
```

[2] http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.html#jetc
[3] http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.html

```
  protected final String TEXT_23 = "(..))) && call(* ";
  protected final String TEXT_24 = "(..))\")" + NL + "\tpublic Object";
  protected final String TEXT_25 = "(ProceedingJoinPoint pjp, " +
    "Method method, Object obj) throws Throwable  " + NL + "\t{" +
    NL + "\t\tDataSegment[] dataSegments = new DataSegment[]{";
  protected final String TEXT_26 = "};" + NL + "\t\treturn " +
   "setDirty(pjp, getDomainObject(obj), getFullMethodName(method)," +
   "dataSegments, CustomerDirtyType.";
  protected final String TEXT_27 = ");" + NL + "\t}";
  ...
  public String generate(DirtyConfiguration configuration)
  {
    final StringBuffer stringBuffer = new StringBuffer();
    ...
    for (Map.Entry<String, Map> method : configuration.getMethods().
                                 entrySet())
    {
      ...
      stringBuffer.append(TEXT_22);
      stringBuffer.append(pointCutName);
      stringBuffer.append(TEXT_23);
      stringBuffer.append(methodName);
      stringBuffer.append(TEXT_24);
      stringBuffer.append(adviceName);
      stringBuffer.append(TEXT_25);
      stringBuffer.append(datasegements);
      stringBuffer.append(TEXT_26);
      stringBuffer.append(dirtyType);
      stringBuffer.append(TEXT_27);
      ...
    }
    ...
    return stringBuffer.toString();
  }
}
```

Listing 5.6: Implementierungsklasse

Nachdem die Implementierungsklasse erzeugt wurde, kann im nächsten Schritt die Generierung eines AOP-Aspekts erfolgen. Vor Verwendung der Implementierungsklasse wird sie vom Java-Compiler in eine class-Datei übersetzt. Im Anschluss wird die übersetzte Implementierungsklasse per Java-Reflection dynamisch zur Laufzeit instanziiert. Auf eine Instanz dieses Objekts wird ebenfalls per Java-Reflection die `generate()`-Methode aufgerufen. Der Methode wird das zu verarbeitende `DirtyConfiguration`-Objekt als Parameter übergeben. Als Rückgabewert liefert die Methode einen String, der den generierten Code enthält. Mit dem XML-Handler wird der generierte Java-Code unter einer Java-Klasse (Interceptor) gespeichert. Eine Interceptor-Klasse repräsentiert einen Aspekt mit den verschiedenen Pointcuts und Advices, die aus den Einstellungen der Konfigurationsdatei hervorgehen. Mit diesem Aspekt wird die Erkennung von Datenänderungen und der darauffolgenden Ermittlungen des Aktualisierungsbedarfs verwirklicht.

In dem Beispielcode 5.7 wird die `AccountDirtyInterceptor`-Klasse dargestellt, die einen Aspekt für die Account-Klasse mit entsprechender Pointcut- und Advice-

Annotation bereitstellt. In dieser Klasse wird zuerst ein `AccountStateObserver`-Objekt deklariert, der die Ermittlung des Aktualisierungsbedarfs übernimmt. Innerhalb des Around-Advices (adviceSetBalance1) wird der kontrollflussbasierte Pointcut (`cflow`) definiert. Dieser Pointcut wählt alle Join Points, die im angegeben Pointcut liegen aus. In Verbindung mit dem `call`-Pointcut bedeutet dies, dass die Join Points des `call`-Pointcuts, die im Kontrollfluss des Pointcuts liegen, betrachten werden. Auf den Beispielcode bezogen bedeutet es, dass der Advice nur ausgeführt wird, wenn der Aufruf der `setBalance()`-Methode des `Account`-Objekts aus dem Import-Package ausgeführt wurde. Dadurch lässt sich erreichen, dass die Ausführung des Advices nur aus dem definierten Bereichen der Anwendung durchgeführt werden kann. Demgegenüber steht, dass für jeden Bereich, in dem eine mögliche Datenänderungen eintreten kann, ein Advice definiert werden muss.
Sofort nach dem Aufrufen der `setBalance()`-Methode aus dem Import, noch vor Änderung des Saldo-Wertes wird der Around-Advice-Code ausgeführt. Dabei enthält das `org.aspectj.lang.ProceedingJoinPoint`-Objekt alle abhängigen Objekte, wie zum Beispiel das zu ändernde Objekt (`pjp.getTarget()`). Die erste Aktion, die im Advice-Code ausgeführt wird, ist das Zwischenspeichern des zu ändernden Objekts mit Hilfe des `AccountStateObserver`. Daran anschließend wird der ursprüngliche Join Point ausgeführt, in dem die `proceed()`-Methode auf das `ProceedingJoinPoint`-Objekt aufgerufen wird. Vorausgesetzt der Wert des Saldos wird durch Ausführung des Join Points geändert, werden in der `setCustomerAndParentEngagementsDirty()`-Methode alle abhängigen Analysesegmente *dirty* gesetzt. Dies bedeutet, dass diese Analysesegmente “dreckig” sind und sobald wie möglich aktualisiert werden müssen. Die Umsetzung der Aktualisierung der Analysesegmente ist nicht Bestandteil dieser Implementierung.

```
@Aspect
public class AccountDirtyInterceptor
{
  private final AccountStateObserver stateObserver =
    new AccountStateObserver();
  ...
  @Around("cflow(execution(* de.esag.mind.banker.wave.
    server.imports..*(..))) && call(* de.esag.mind.banker.
    wave.server.domain.Account.setBalance(..))")
  public Object adviceSetBalance1(ProceedingJoinPoint pjp)
    throws Throwable
  {
    DataSegment[] dataSegments = new DataSegment[]{
      DataSegment.DS_ASSETLIABILITY,..,DataSegment.DS_SEGMENTATION};
    saveObjectState(obj);
    Object result = pjp.proceed();

    if (stateObserver.hasChanged())
    {
      setCustomerAndParentEngagementsDirty(obj, dataSegments);
    }
    return result;
  }
  ...
}
```

Listing 5.7: Aspekt für ein Account-Objekt

5.2.5 Build-Prozess

Die Umsetzung des Modells wird durch einen Build-Prozess durchgeführt. Dabei wird das Build-Management-Tool Ant[4] eingesetzt, um den Build-Prozess zu unterstützen. Ant ist ein in Java implementiertes Open-Source-Projekt, welches ein Teilprojekt der Apache Software Foundation ist. Voraussetzung für die Verwendung von Ant ist die Java-Laufzeitumgebung, dadurch kann dieses Build-Werkzeug auf verschiedenen Plattformen verwendet werden.
Ant benötigt zur Ausführung eine XML-Datei, die so genannte Build-Datei. Diese Build-Datei beschreibt den Ablauf des Build-Prozesses und damit, welche Schritte von Ant auszuführen sind. Zur Beschreibung stehen Targets zur Verfügung, in denen ein bestimmter Teil des Build-Prozesses definiert wird. Jedes Target beinhaltet eine Menge von den Tasks. In den Tasks werden die einzelnen Aufgaben, wie zum Beispiel die Kompilierung einer Java-Klasse, ausgeführt. Neben dem Java-Compiler stellt Ant eine große Zahl zusätzlicher Tasks zur Verfügung. Eine komplette Liste aller vorhandenen Tasks ist unter der Ant-Dokumentation[5] zu finden. Es besteht neben den vordefinierten Tasks die Möglichkeit, eigene Tasks zu definieren und diese in den Targets zu verwenden.

In MinD.banker wird der Ablauf des Build-Prozesses durch folgende Tasks definiert:

1. Erzeugung von Konfigurationsdateien mit allen Abhängigkeiten
2. Übersetzung der Implementationsklasse aus dem Template
3. Instanziierung der Implementationsklasse
4. Generierung der Aspekte
5. Hinzufügen der Aspekte zur AOP-Konfigurationsdatei
6. Löschen aller generierten Konfigurationsdateien und der Implementationsklasse

5.3 Zusammenfassung

In diesem Kapitel wurde ausgehend vom Konzept (Kapitel 4) die Umsetzung im Rahmen einer prototypische Implementation am Beispiel von MinD.banker beschrieben. Dazu wurde im Abschnitt 5.1 einen Einblick in für den Prototypen verwendete Bankensoftware MinD.banker gegeben. In den nachfolgenden Kapiteln wurde detailliert beschrieben, wie die Komponenten des Modells in Java umgesetzt wurden. Zudem wurde erläutert, wie die umgesetzten Komponenten in einen Build-Prozess integriert werden.

[4] http://ant.apache.org/
[5] http://ant.apache.org/manual/index.html

6. Auswertung der erfüllten Anforderungen

In den vorherigen Kapiteln wurde ausgehend von den Anforderungen ein Konzept erstellt. Aufbauend auf diesem Konzept wurde im Rahmen der Implementierung ein Prototyp entwickelt. Dieser Prototyp dient in diesem Kapitel als Grundlage für die Auswertung der umgesetzten Anforderungen. Dabei wird darauf eingegangen, welche Anforderungen vollständig, teilweise oder nicht umgesetzt werden konnten. Die Probleme die während der Evaluierungsphase abgetreten sind und wie der Prototyp angepasst. In dem nächsten Kapital können darauf aufbauend ausstehende Anforderungen oder Anregungen zur Verbesserung des Prototyps definiert werden.

6.1 Auswertung der Anforderungen

Im folgenden Abschnitt wird die Umsetzung der definierten funktionalen und nicht-funktionalen Anforderungen beschrieben. Dabei wird für die einzelnen Anforderungen diskutiert, wie und ob sie umgesetzt wurden. Am Ende eines jeden Abschnitts werden zusammenfassend nochmal die umgesetzten Anforderungen in Form einer Tabelle dargestellt.

6.1.1 Funktionale Anforderungen

Die funktionalen Anforderungen ergeben sich aus der Tabelle 3.1. Dazu gehören

automatische Erkennung

Als erste funktionale Anforderung an den Prototyps wurde die automatische Erkennung von Datenänderungen an den operativen Daten definiert. Sobald eine Datenänderungen an den operativen Daten stattgefunden hat, soll der Prototyp andere Prozesse in Kenntnis setzen, dass sich eine Änderung an den operativen Daten ereignet hat. Von elementarer Bedeutung ist dabei, dass die Erkennung automatisch erfolgen soll. Im Zuge dessen spielt es keine Rolle welche Art von Datenänderung

(Hinzufügen, Aktualisieren und Löschen) oder welche operative Daten geändert wurden.
Die Umsetzung dieser Anforderung wurde vollständig umgesetzt. Dazu wurden die verschiedenen Methoden zur automatischen Erkennung von Datenänderungen aus dem Kapitel Abschnitt 2.1 auf eine Umsetzung in dem Prototyp überprüft. Aufgrund der Vorteile der Trennung der Verantwortungen und der Bestimmung des Kontrollflusses von Datenänderungen wurde sich für die Nutzung des aspektorientierten Ansatzes entschieden. Hierdurch lässt sich der Bereich, aus dem die Datenänderungen ausgeführt wurden bestimmen. Dies ist für MinD.banker von großer Bedeutung, damit der Prototyp korrekt arbeitet und in Verbindung mit anderen Anforderungen, wie der Feingranularität, eine effiziente Datenanalyse gewährleisten kann.
Zur Umsetzung des aspektorientierten Ansatzes wird AspectJ, eine aspektorientierte Erweiterung von Java verwendet. In AspectJ werden für die operativen Daten dazugehörige Aspekte generiert. Innerhalb dieser Aspekte werden bestimmte Pointcuts definiert. Die Pointcuts beschreiben eine Menge von Join Points. Dabei stellt ein Join Point einen eindeutigen Punkt, Aufruf einer Methode oder Zuweisung eines Wertes einer Variablen, im Quellcode dar. Sobald im Programmablauf die Join Points eines Pointcuts detektiert werden, ist die automatische Erkennung abgeschlossen.

Feingranularität

Die nächste funktionale Anforderung beschreibt die Feingranularität der operativen Daten und der Analysedaten. Diese Anforderung ist eng mit der vorherigen Anforderung der automatischen Erkennung verbunden. In diesem Zusammenhang beschreibt die Granularität, wie fein die operativen Daten in mehrere kleine, physisch selbständige Dateneinheiten und die Analysedaten in Segmente untergliedert werden. Durch die feine Granularität reduziert sich der Aktualisierungsaufwand für die Analysedaten, da bei einer Erkennung einer Änderung an einer Dateneinheit gezielt einzelne betroffene Analysesegmente aktualisiert werden können. Wäre die Untergliederung nicht vorhanden, müssten immer alle Analysedaten nach einer Änderungen aktualisiert werden, obgleich nur einige Segmente von der Aktualisierung betroffen wären. Eine weitere positive Eigenschaft dieser Feingranularität ist, dass die Aktualisierungen aufgrund des hohen Aktualisierungsaufwandes nicht mehr zu festgelegten Zeiten, sondern zeitaktuell durchgeführt wurden und die Analysedaten dadurch den aktuellen Stand widerspiegeln.
Eine Dateneinheit stellt dazu eine Ausprägung oder Eigenschaft eines Objektes aus den operativen Daten dar. Diesen so entstehenden einzelnen Dateneinheiten können zielgerichtet bestimmte Datensegmente der Analysedaten zugeordnet werden. Mit diesen kleineren Dateneinheiten lässt sich die Datenanalyse wesentlich effizienter und einfacher durchführen. Da in den meisten Fällen die Änderungen an den Dateneinheiten nur ein beziehungsweise wenige Datensegmente betreffen, können so gezielt nur die relevanten Datensegmente aktualisiert werden.
In dem Prototyp erfolgt die automatische Erkennung von Datenänderungen auf der Ebene von Methoden und nicht auf Attributebene. Der Grund ist darin zu suchen, dass die Erkennung von Änderungen nicht nur auf die Attributwerte der operativen Daten beschränkt ist, sondern sie sich ebenso auf alle anderen Objekten anwenden lassen. Die Segmente der Analysedaten sind an die Analyse-Bereiche innerhalb von MinD.banker ausgerichtet. Diese granulare Unterteilung der Analysesegmente stellt

den ersten Schritt hin zur effizienten Datenanalyse dar. In der Regel müssen bei einer Aktualisierung eines Analysesegments nur bestimmte Werte des Segments aktualisiert werden. In der aktuellen Implementationsstufe werden stattdessen alle Werte für die Neuberechnung vorgemerkt. Aus diesem Grund besteht die Notwendigkeit, die Analysesegmente noch weiter aufzugliedern. Dies wurde nicht umgesetzt, weil die Analysesegmente zu verschieden aufgebaut sind und die Vereinheitlichung sehr zeitintensiv wäre.

Modellierung der Abhängigkeit

Diese Anforderung charakterisiert die Abhängigkeiten zwischen den operativen Daten und den Analysedaten. Die analysierten fachlichen Abhängigkeiten zwischen operativen Daten und Analysedaten sollen mit Hilfe eines Modells visualisiert werden. Dazu wurde dokumentiert, welche operativen Daten welche Analysedaten beeinflussen und wie sich die Beziehungen zueinander gestalten. Dieses Abhängigkeitsmodells unterscheidet auf Grundlage der aktuelle MinD.banker Version zwei Arten von Abhängigkeiten. Zum einen die Abhängigkeiten zwischen den operativen Daten und den Analysedaten und zum anderen die Abhängigkeiten zwischen den Analysedaten. Zur Umsetzung wurden die beiden Abhängigkeitsarten in verschiedene Dateien separiert. Die erste Kategorie der Abhängigkeiten wurde in die Konfigurationsdateien abgespalten. Der zweite Teil wurde in eine Abhängigkeitsdatei ausgelagert.
Die Konfigurationsdateien wurden so umgesetzt, dass jede Konfigurationsdatei immer auf einen bestimmten Bereich, eine Domäne, zugeschnitten ist. Innerhalb dieser Konfigurationsdateien werden sehr präzise Abhängigkeiten zwischen den operativen Daten und den direkt abhängigen einzelnen Segmenten der Analysedaten gespeichert, was eine effiziente Datenanalyse ermöglicht. Dies ist jedoch durch die Feingranularität mit dem Nachteil verbunden ist, das viele Abhängigkeitsdatensätze vorhanden sind. Die Abhängigkeitsdatei enthält für jedes Analysesegment, die direkten Abhängigkeiten zu anderen Analysesegmenten.
Aufgrund der Trennung in Konfigurationsdateien und einer Abhängigkeitsdatei muss bei einer Änderung nur eine Datei, entweder Konfigurationsdatei oder Abhängigkeitsdatei, angepasst werden. Wenn alle abhängigen Analysesegmente in den Konfigurationsdateien enthalten wären, müsste bei einer Anpassung eines Analysesegments alle davon betroffenen Konfigurationsdateien angepasst werden. Bei einer Vielzahl von Konfigurationsdateien könnten die Änderungen schnell unübersichtlich werden und auch sollte der Aufwand an Änderungen nicht unterschätzt werden. Des Weiteren beinhaltet dieses Vorgehen eine große Fehlerquelle, da Konfigurationsdateien vergessen werden könnten. Da die Abhängigkeiten statisch im Programm durch Dateien abgebildet werden, können sie zur Laufzeit in MinD.banker nicht geändert werden.

Ermittlung des Aktualisierungsbedarfs

Die letzte funktionale Anforderung der Ermittlung des Aktualisierungsbedarfs beschreibt eine Funktion, die aus zwei Teilfunktionen besteht. Zu einem eine Bestimmung der fachlichen Abhängigkeit zwischen den geänderten operativen Daten und einem Segment aus den Analysedaten und zum anderen eine Validierung der Datenänderung. Die Funktion wird nach einer Erkennung einer Datenänderung ausgeführt.

Dazu wird im ersten Schritt die Methode identifiziert, welche die Datenänderung veranlasst hat. Bevor die Änderung an einem Objekt ausgeführt werden kann, wird mit Hilfe eines StateObservers wird der Zustand des Objekts gespeichert. Dafür wird der Inhalt beziehungsweise Wert des Attributs der identifizierten Methode festgehalten. Nach Änderung des Objekts werden die Zustände vor und nach einer Änderung abgeglichen. Wenn sich herausstellt, dass die Zustände des Objekts identisch sind, wird die Ermittlung des Aktualisierungsbedarfs beendet. In diesem Fall werden keine weiteren Aktionen durchgeführt. Dieser Thematik ist in MinD.banker gar nicht so selten, da beispielsweise beim Bearbeiten eines Kontos nicht nur die geänderten Werte des Kontos, sondern alle Werte zurück in die Datenbank geschrieben werden. Beim Zurückschreiben in die Datenbank werden somit auch Änderungen erkannt, die keine "wirklichen" Änderungen darstellen. Andernfalls kann im nächsten Schritt mit der Bestimmung der fachlichen Abhängigkeiten unter Berücksichtigung des Abhängigkeitsmodells fortgefahren werden. Dazu wird anhand des Abhängigkeitsmodells ermittelt, ob diese Änderung an der operativen Dateneinheit, eine Abhängigkeit zu einem Segment der Analysedaten hat. Mit der Umsetzung dieser Anforderung ist es nun möglich, nach der Erkennung einer Datenänderung den Aktualisierungsbedarf zu ermitteln.
Um herausfinden, ob die Anforderung der Ermittlung des Aktualisierungsbedarfs korrekt funktioniert, bietet der Prototyp ein Logging an. Dieses umfasst das Speichern der Zustände vor und nach einer Änderung in eine Log-Datei. Ebenfalls werden die ermittelten Abhängigkeiten gespeichert. Anhand dieser Log-Datei kann nachvollzogen werden, ob die Ermittlung des Aktualisierungsbedarfs korrekt funktioniert.

Die dargestellten Tabelle gibt einen Überblick über den Stand der Umsetzung der fachlichen Anforderungen. Dabei wurden folgende Symbole verwendet:

- "✓" ⇒ Anforderung vollständig umgesetzt
- "∼" ⇒ Anforderung zumindest teilweise umgesetzt
- "✗" ⇒ Anforderung nicht umgesetzt

Anforderungen	**Umsetzung**	
automatische Erkennung	Die automatische Erkennung von Datenänderungen an den operativen Daten wurde umgesetzt. Dafür wurden Punkte, die eintretende Ereignisse während des Programmlaufs beschreiben, definiert. Wird ein Ereignis im Programmablauf ausführt, so wird ein Signal ausgelöst, dass ein bestimmter Punkt erreicht wurde.	✓
Feingranularität	Die feine Granularität von operativen Daten und Analysedaten wurde umgesetzt. Dazu wurden die operativen Daten in kleine, physisch selbständige Dateneinheiten und die Analysedaten in Segmente untergliedert. Eine weitere Aufgliederung der Analysesegmente ist denkbar.	∼

Anforderungen	Umsetzung	
Modellierung der Abhängigkeit	Wie in Abschnitt 4.2 dargestellt wurde, erfolgt die Umsetzung der fachlichen Abhängigkeiten zwischen operativen Daten und Analysedaten in einem Modell. Die Abbildung dieses Modells wurde in einzelnen Konfigurationsdateien und einer Abhängigkeitsdatei vorgenommen.	✓
Ermittlung des Aktualisierungsbedarfs	In dem Prototyp wurde die Ermittlung des Aktualisierungsbedarfs umgesetzt. Dabei wird bestimmt, ob die geänderte Dateneinheit Abhängigkeiten zu Analysesegmenten hat und wenn ja, welche Abhängigkeiten bestehen. Dadurch werden gezielt nur die dazugehörenden Analysesegmente aktualisiert.	✓

Tabelle 6.1: Umsetzung der funktionalen Anforderungen

6.1.2 Nicht-funktionale Anforderungen

Neben den funktionalen werden auch die nicht-funktionalen Anforderungen festgelegt. Dazu gehören

Effizienz

Die wichtigste nicht-funktionale Anforderung ist die Effizienz. Dazu kann die Effizienz unterschiedlich betrachtet werden, zum einen in die Effizienz des Aktualisierungsbedarfs der Analysedaten im Verhältnis zur Erkennung von Datenänderungen und zum anderen in die Effizienz des Algorithmus zur Ermittlung des Aktualisierungsbedarfs. Der Prototyp ist dahingehend effizient, dass nur diese Segmente der Analysedaten zur Aktualisierung vorgemerkt werden, die in direkter oder indirekter Abhängigkeit zu der geänderten operativen Dateneinheit stehen. Durch die effiziente Datenanalyse können unnötige Aktualisierungen der Analysedaten vermieden, was im Endeffekt zur Reduzierung der Verwendung von Ressourcen führt. Die freigewordenen Ressourcen können dazu verwendet werden, um die Aktualisierung der Analysedaten in kürzeren Intervallen durchzuführen. Dies führt zu einem weiteren positiven Nebeneffekt der Erhöhung der Aktualität der Analysedaten.
Um den Grad der Effizienz zwischen der Erkennung von Datenänderungen und der Aktualisierung der Analysedaten zu ermitteln, werden weitere Tests benötigt. Aus Mangel an Testdaten wurde kann Effizienz bisher nicht quantifiziert werden.
Um zu bestimmen, wie sich Effizienz des Algorithmus zur Ermittlung des Aktualisierungsbedarfs verhält, wurde ein kleines Testszenario aufgebaut. Innerhalb dieses Testszenarios wurden 100000 Konten mit unterschiedlichen Werte generiert. In einer Messung wurde für jedes Konto einmal alle `set`-Methoden, die zu einer Datenänderungen führen können, aufgerufen. Dieser Messung wurde fünf mal in MinD.banker jeweils mit und ohne Erkennung von Datenänderungen und Ermittlung des Aktualisierungsbedarfs durchgeführt. Die Ergebnisse des Testszenarios sind in Tabelle 6.2 aufgelistet.

Messung	**1.**	**2.**	**3.**	**4.**	**5.**	$\bar{x}$
mit Ermittlung des Aktualisierungsbedarfs [ms]	71577	72842	71647	71982	72036	72017
ohne Ermittlung des Aktualisierungsbedarfs [ms]	61149	63259	63358	61415	63150	62520

Tabelle 6.2: Testszenario mit und ohne Ermittlung des Aktualisierungsbedarfs

Aus der Tabelle 6.2 lässt sich erkennen, dass die durchschnittliche Differenz zwischen den Messung mit und ohne Ermittlung des Aktualisierungsbedarfs auf 100000 Konten rund 10 s beträgt. Dieses Ergebnis auf ein Konto bezogen bedeutet, dass die Differenz zwischen den beiden Messungen rund 0,1 ms beträgt. Aufgrund dieser Erkenntnis und dem Umstand, dass die `set`-Methoden nur vor dem Speichern in die Datenbank aufgerufen werden, lässt sich feststellen, dass die Ermittlung des Aktualisierungsbedarfs zu einer erhöhten Laufzeit führt. Diese Erhöhung hat keinen signifikanten Einfluss auf Performance und beeinflusst somit die Effizienz der Anwendung nicht.
Um jedoch eine abschließende Beurteilung durchführen zu können, muss eine umfangreichere Evaluation durchgeführt werden, die im Rahmen dieser Arbeit, jedoch nicht realisiert werden konnte. Dafür wäre einen Messungen zum Beispiel durch einen Import von Kundendaten in MinD.banker erforderlich. Diese Messungen könnten auch einen Überblick über die Effizienz zwischen der Erkennung von Datenänderungen und des Aktualisierungsbedarfs der Analysedaten geben.

Transparenz gegenüber den Entwicklern

Das Ziel dieser Anforderung war es, die Transparenz gegenüber den Entwicklern zu erhöhen, um den Mangel an Abhängigkeit vom individuellen Wissen der Entwickler bei der Erkennung von Änderungen, auf Basis der Abhängigkeiten zwischen den operativen und Analysedaten zu beseitigen. Entwickler sollten zukünftig besser nachvollziehen können, inwieweit sich Änderungen an den operativen Daten auf die Aktualisierung der Analysedaten auswirken. Dafür muss für die Entwickler klar erkennbar sein, welche Abhängigkeiten zwischen den operativen Daten und den Analysedaten bestehen. Dazu wurde ein Abhängigkeitsmodells umsetzt. Die Modellierung der Abhängigkeiten zwischen den operativen Daten und den Analysedaten soll einen durchschaubaren und nachvollziehbaren Überblick über die Zusammenhänge zwischen operativen Daten und Analysedaten ermöglicht. Die Entwickler müssen sich nicht mehr selbst um die Abhängigkeiten kümmern, sondern diese Aufgabe übernimmt in diesem Punkt das Modell.

Testbarkeit

Die Anforderung der Testbarkeit wurde in dem Prototyp nicht umgesetzt. Mit der Testbarkeit sollte die semantische und syntaktische Korrektheit des Prototyps sichergestellt werden um frühzeitig Fehler im Ablauf der Datenanalyse zu erkennen und zu beseitigen. Dies lässt sich mit verschiedenen Tools aus dem Bereich der Unit-Tests realisieren. Die Umsetzung der Testbarkeit soll nicht nur auf die einzelnen Komponenten des Prototyps beschränkt sein, sondern darüber hinaus auch auf

den Prototypen als Ganzes. So könnten zum Beispiel einzelne Konfigurationsdateien auf Syntaxfehler getestet, um fehlerhafte Einträge in der Konfigurationsdatei zu minimieren oder semantische Fehler im Algorithmus zur die Ermittlung der Abhängigkeiten aufgedeckt werden. Somit lässt sich bei einem Auftreten eines Fehlers vornherein die Generierung von fehlerhaftem Code verhindern. Diese Tests können im Rahmen eines Build-Prozesses automatisiert werden. Um manuelle Veränderung an dem durch den Transformator generiertem Code zu verhindern, können ebenfalls entsprechende Tests implementiert werden.

Erweiterbarkeit und Wartbarkeit

Durch eine Zentralisierung des Codes an eine fest definierte Stelle wurde die letzte nicht-Funktionale Anforderung umgesetzt. Infolgedessen müssen die Änderungen nicht mehr über die gesamte Anwendung verteilt vorgenommen werden. Eine sinnvolle Aufteilung in Konfigurationsdateien und einer Abhängigkeitsdatei trägt gleichwohl zum Verständnis und Übersichtlichkeit bei. Dadurch können die Änderungen ohne großen Aufwand in die betroffenen Dateien eingepflegt werden. Gleichfalls benötigen die Entwickler aufgrund der Tatsache, dass in den Dateien eine fest definierte Syntax vorgeschrieben ist, keine große Einarbeitungszeit. Anhand von Unit-Tests kann der Entwickler sofort erkennen, ob vorgenommene Erweiterungen an den Dateien korrekt umgesetzt wurden.

Die nachfolgende Tabelle stellt den Stand der Umsetzung der nicht-fachlichen Anforderungen dar. Dabei wurden die selben Symbole wie in der Tabelle 6.1 verwendet.

Anforderungen	**Umsetzung**	
Effizienz	Die Effizienz des Prototyps wird durch den Aktualisierungsbedarf bestimmt. Generell wurde die Ermittlung des Aktualisierungsbedarfs umgesetzt und führt nachweislich zu einer Erhöhung der Effizienz. Aus Mangel an Testdaten kann die Effizienz nicht genau quantifiziert werden.	✓
Transparenz gegenüber den Entwicklern	Durch die Visualisierung der Abhängigkeiten zwischen den operativen Daten und den Analysedaten wurde die Transparenz gegenüber den Entwicklern umgesetzt.	✓
Testbarkeit	Die Testbarkeit wurde in dieser Arbeit nicht umgesetzt und bleibt für weitere Arbeiten als Anforderung erhalten.	✗
Erweiterbarkeit und Wartbarkeit	Der Prototyp führt durch Zentralisierung des Codes und Aufteilung der Abhängigkeiten in einzelne Dateien zu einer Erhöhung der Erweiterbarkeit und Wartbarkeit.	✓

Tabelle 6.3: Umsetzung der nicht-funktionalen Anforderungen

6.2 Zusammenfassung

In diesem Kapitel wurden die funktionalen und nicht-funktionalen Anforderung nochmals betrachtet und deren Umsetzung detailliert beschrieben. Dabei wurden

sechs der insgesamt acht Anforderungen vollständig umgesetzt, wobei Verbesserungen der einzelnen Umsetzungen im späteren Verlauf notwendig werden können. Die nicht-funktionale Anforderung Testbarkeit wurde innerhalb der prototypischen Implementierung nicht umgesetzt und könnte in einem Folgeprojekt umgesetzt werden. Ebenfalls wurde die Feingranularität, funktionale Anforderung, nur teilweise umgesetzt, da die Untergliederung der Analysesegmente noch nicht komplett abgeschlossen ist.

7. Zusammenfassung und Ausblick

7.1 Zusammenfassung

In den nächsten Jahren wird sich zeigen, welchen Stellenwert eine effiziente Datenanalyse in Unternehmen für zukünftige relevante Entscheidungen hat. Zahlreiche Faktoren, wie Extraktion und Verdichtung der vorhandenen Daten spielen dabei eine wichtige Rolle.
Die vorliegenden Arbeit beschäftigt sich mit dem Konzept zur effizienten Datenanalyse auf Basis feingranularer Änderungserkennung. Die Inhalte dieses Konzepts wurde im Rahmen einer prototypischen Implementierung am Beispiel am MinD.banker umgesetzt werden. Dabei stellt MinD.banker eine Bankensoftware zur strukturierten Analyse und Beratung von Kunden dar.

Um das Konzept herleiten zu können, wurde die Untersuchung der Forschungsfrage in mehrere Teilprobleme aufgeteilt. Die Ergebnisse der Untersuchungsfragen werden nun näher erläutert.
Dabei wurden in der Untersuchungsfrage 1 zunächst existierende Methoden zur automatischen und transparenten Erkennung von feingranularen Datenänderungen betrachtet. Aus der ersten Untersuchungsfrage resultierte eine Übersicht über die vorhandenen Methoden der softwaretechnischen als auch datenbankbezogenen Erkennung von Datenänderungen.
Im Rahmen der Untersuchungsfrage 2 wurden anschließend, die Abhängigkeiten zwischen operativen Daten und Analysedaten formal beschrieben, um den Aktualisierungsbedarf der Analysedaten zu bestimmen. Die Beschreibung wurde in Form eines Abhängigkeitsmodells visualisiert.
Die Beantwortung der Untersuchungsfrage 3 lieferte einen Katalog der relevanten Anforderungen zur effizienten Datenanalyse. Dazu zählen unter anderem die automatische Erkennung von Datenänderungen und die Modellierung der Abhängigkeiten zwischen den operativen Daten und den Analysedaten. Weiterhin soll die Ermittlung des Aktualisierungsbedarfs ermöglicht werden, um damit effizient nur die abhängigen Analysedaten zu bestimmen und zu aktualisieren.
Die Untersuchungsfrage 4 beschreibt einen Konzeptentwurf. Dabei flossen die Er-

kenntnisse der vorherigen drei Untersuchungsfragen in das Konzept ein. Dieser Konzeptentwurf wurde mit Hilfe eines Modells, welche die Komponenten Abhängigkeitsmodell, Template und Transformatoren beinhaltet, dargestellt.
Anhand des Konzeptes resultierte aus der Untersuchungsfrage 5 die prototypische Implementierung der Erkennung von Datenänderungen und der Ermittlung des Aktualisierungsbedarfs der Analysedaten mit MinD.banker. Bei der prototypischen Umsetzung des Konzeptes wurde entschieden, den aspektorientierten Ansatz zur Erkennung von Datenänderungen zu verwenden. Dafür wird aus einer Konfigurationsdatei und einem Template unter Einsatz des Transformators automatisch Quellcode generiert, der einen AOP-Aspekt repräsentiert.
Nachdem die Implementierung vorgestellt wurde, konnte die Umsetzung der Anforderungen ausgewertet werden. Insgesamt sind sechs der acht formulierten Anforderungen umgesetzt wurden, wobei eine Anforderung nur teilweise und eine Anforderung nicht umgesetzt wurde. Diese Anforderungen bleiben offen für weitere Projekte. In der Evaluation konnte ebenfalls gezeigt werden, dass durch Ermittlung des Aktualisierungsbedarfs auf Basis feingranularer Änderungserkennung, die Effizienz der Datenanalyse gesteigert werden konnte. Auch hat der Algorithmus zur Ermittlung des Aktualisierungsbedarfs keinen negativen Einfluss auf die Effizienz der Anwendung. Um eine abschließende Beurteilung zur Effizienz der Datenanalyse vorzunehmen, muss eine umfangreichere Evaluation, zum Beispiel durch einen Import von Kundendaten in MinD.banker, durchgeführt werden.
Es sind noch einige Erweiterungen und Verbesserungen der entwickelten Prototyps möglich, die im folgenden Abschnitt kurz erläutert werden.

7.2 Ausblick

Die vorliegende Arbeit bildet die Grundlage für die Aktualisierung der Analysedaten. Diese Aktualisierung der Analysedaten können im Rahmen eines weiteren Projektes bearbeitet werden. Die erforderlichen Grundlagen für diese nachfolgende Arbeit wurde durch die feingranulare Erkennung von Datenänderungen und die Ermittlung des Aktualisierungsbedarfs gelegt.
Der Prototyp bietet einige Möglichkeiten zur Weiterentwicklung in nachfolgenden Projekten an. Da bei der Implementierung nicht alle Anforderungen vollständig umgesetzt worden, wäre eine erste Maßnahme die beiden offen gebliebenen Anforderungen zu implementieren. Dazu gehört die Anforderung der Testbarkeit des Prototyps. Mit der Testbarkeit soll die semantische und syntaktische Korrektheit des Prototyps sichergestellt werden um frühzeitig Fehler im Ablauf der Datenanalyse zu erkennen und zu beseitigen. Die Umsetzung ließe sich mit den verschiedenen Methoden aus dem Bereich der Unit-Tests durchführen. Ein weiterer Punkt in Zusammenhang mit der Testbarkeit, wäre die Gültigkeit und Vollständigkeit der Konfigurationsdateien und der Abhängigkeitsdatei durch den Einsatz eines XML-Schemas sicherzustellen.
Eine weitere unvollständig umgesetzte Anforderung stellt die Feingranularität der Analysesegmente dar. In der aktuellen Implementationsstufe werden einer operativen Dateneinheit immer komplette Analysesegmente zugeordnet. In den meisten Fällen sollte durch eine Änderung an einer operativen Dateneinheit nicht das gesamte Analysesegment, sondern nur bestimmte Werte des Analysesegments aktualisiert.

Zusätzlich zu den noch nicht umgesetzten Anforderungen des Prototyps bestünde die Möglichkeit das Abhängigkeitsmodell grafisch in einem Plugin zu visualisieren. So wäre denkbar, die Abhängigkeiten zwischen den operativen Daten und den Analysedaten in dem Plugin nicht nur einfach darzustellen, sondern das Hinzufügen, Bearbeiten und Löschen von Abhängigkeiten zu erlauben. Infolgedessen wäre es für die Entwickler unerheblich, wie und wo die Abhängigkeiten gespeichert werden.
Da sich die Abhängigkeiten in dem generierten Code widerspiegeln, muss bei einer Änderung der Abhängigkeiten dieser erzeugte Code aktualisiert werden. Dadurch ergibt sich eine weitere Verbesserung des Prototyps, die automatische Aktualisierung des generierten Codes nach einer Änderung der Abhängigkeiten. Derzeit muss diese Generierung des Codes durch ein Build-Skript manuell gestartet werden. Die Umsetzung der Verbesserung ließe durch automatisierte Ausführung des Build-Skripts zum einen sofort nach einer Änderungen oder zum anderem während des Kompiliervorgangs der MinD.banker-Anwendung realisiert werden.

Literaturverzeichnis

[ABKS13] Apel, S.; Batory, D.; Kästner, C.; Saake, G.: *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, October 2013. (zitiert auf Seite xi, 12 und 13)

[Bal09] Balzert, H.: *Lehrbuch der Software-Technik: Basiskonzepte und Requirements Engineering*. Spektrum, 3.. Auflage, 2009. (zitiert auf Seite 15 und 16)

[BK04] Bonomon-Kappeler, I.: Aspektorientierte Software-Entwicklung - Unter besonderer Berücksichtigung der begrifflichen Zusammenhänge und der Einbettung in den Entwicklungsprozess. Diplomarbeit, Universität Zürich, 2004. (zitiert auf Seite 11)

[Cha99] Chamberlin, D. D.: *DB 2 Universal Database - Der unentbehrliche Begleiter*. Addison-Wesley, 1999. (zitiert auf Seite 7)

[CJR06] Cazzola, W.; Jézéquel, J.-M.; Rashid, A.: Semantic Join Point Models: Motivations, Notions and Requirements. In *In Software Engineering Properties of Languages and Aspect Technologies (SPLAT)*, 2006. (zitiert auf Seite 12)

[EFB01] Elrad, T.; Filman, R. E.; Bader, A.: Aspect-oriented Programming: Introduction. *Communications of the ACM*, Band 44, Nr. 10, S. 29–32, Oktober 2001. (zitiert auf Seite 11)

[GAD08] GAD: forum, 2008. http://www.gad.de/content/dam/g0002-0/service/Publikationen/forum-archiv/forum_08/forum_03_08.pdf. (zitiert auf Seite 1)

[Gol11] Goll, J.: *Methoden und Architekturen der Softwaretechnik*. Vieweg+Teubner Verlag, 2011. (zitiert auf Seite vii, 12 und 13)

[GRC09] Gómez, J. M.; Rautenstrauch, C.; Cissek, P.: *Einführung in Business Intelligence mit SAP NetWeaver 7.0*. Springer, 2009. (zitiert auf Seite 2 und 20)

[Her13] Herden, S.: *Model-Driven-Configuration-Management*. Springer, 2013. (zitiert auf Seite 14 und 15)

[ISO99] *ANSI/ISO/IEC International Standard (IS) Database Language SQL – Part 1: SQL/Framework, ISO/IEC 9075-1:1999 (E)*, September 1999. (zitiert auf Seite 6)

[KBA+10] King, G.; Bauer, C.; Andersen, M. R.; Bernard, E.; Ebersole, S.: *Hibernate Reference Documentation, 3.6.0.cr2 edition*, 2010. (zitiert auf Seite xi, 9, 10 und 11)

[KHH+01] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G.: An Overview of AspectJ. In *Proceedings of the Conference on Object-Oriented Programming (ECOOP)*, S. 327–353. Springer, 2001. (zitiert auf Seite 11)

[Kle11] Kleuker, S.: *Grundkurs Datenbankentwicklung, Von der Anforderungsanalyse zur komplexen Datenbankanfrage*. Vieweg+Teubner Verlag, 2011. (zitiert auf Seite 7)

[KLM+97] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C. V.; Loingtier, J.-M.; Irwin, J.: Aspect-Oriented Programming. In *Proceedings of the Conference on Object-Oriented Programming (ECOOP)*, S. 220–242, 1997. (zitiert auf Seite 11)

[KSS12] Köppen, V.; Saake, G.; Sattler, K.-U.: *Data Warehouse Technologien*. mitp, 2012. (zitiert auf Seite 1 und 2)

[LÖ6] Löcher, S.: Modellgetriebene Konfiguration von Transaktionsdiensten in der komponentenbasierten Softwareentwicklung. Dissertation, Technische Universität Dresden, 2006. (zitiert auf Seite 14, 15 und 16)

[Leo13] Leonard, A.: *Pro Hibernate and MongoDB*. Apress, 1st. Auflage, 2013. (zitiert auf Seite 8)

[Mer07] Mertgen, A.: Modellbasierte Integration von Joinpoint Queries für die aspektorientierte Sprache ObjectTeams/Java. Diplomarbeit, Technische Universität Berlin, Oktober 2007. (zitiert auf Seite 11 und 12)

[ML07] Minter, D.; Linwood, J.: *Einführung in Hibernate*. mitp; 1. Auflage, 2007. (zitiert auf Seite vii, 8 und 9)

[O'N08] O'Neil, E. J.: Object/Relational Mapping 2008: Hibernate and the Entity Data Model (Edm). In *Proceedings of the International Conference on Management of Data (SIGMOD)*, S. 1351–1356. ACM, 2008. (zitiert auf Seite 8)

[Poh08] Pohl, K.: *Requirements Engineering: Grundlagen, Prinzipien,Techniken*. dpunkt.Verlag GmbH, 2. Auflage, 2008. (zitiert auf Seite 22)

[RJ01] Ramesh, B.; Jarke, M.: Toward Reference Models for Requirements Traceability. *IEEE Transactions on Software Engineering*, Band 27, Nr. 1, S. 58–93, Januar 2001. (zitiert auf Seite 27)

[Sch06] Schmidt, D. C.: Guest Editor's Introduction: Model-Driven Engineering. *Computer*, Band 39, Nr. 2, S. 25–31, Februar 2006. (zitiert auf Seite 16)

[SI10] Schanz, T.; Izurieta, C.: Object Oriented Design Pattern Decay: A Taxonomy. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, S. 7:1–7:8. ACM, 2010. (zitiert auf Seite 9)

[Sip09] Sippach, F.: *Auditierung mit JPA und Hibernate*, November 2009. http://www.oio.de/public/java/auditierung-jpa-historisierung-hibernate-audit-artikel.pdf. (zitiert auf Seite 10)

[SS05] Schill, P.; Schmauder, R.: Codegenerierung mit dem Eclipse Modeling Framework und JET. *ObjektSpektrum*, S. 59 – 65, 2005. (zitiert auf Seite 33, 34, 42 und 46)

[SSH10] Saake, G.; Sattler, K.-U.; Heuer, A.: *Datenbanken - Konzepte und Sprachen, 4. Auflage.* mitp, 2010. (zitiert auf Seite xi, 5 und 6)

[SV05] Stahl, T.; Völter, M.: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management.* dpunkt.Verlag GmbH, 1. Auflage, 2005. (zitiert auf Seite 14, 15, 16 und 25)

[WP10] Winkler, S.; Pilgrim, J.: A Survey of Traceability in Requirements Engineering and Model-driven Development. *Software & Systems Modeling*, Band 9, Nr. 4, S. 529–565, September 2010. (zitiert auf Seite 27)

[WS07] Wanner, G.; Siegl, S.: Modellgetriebene Softwareentwicklung auf Basis von Open-Source-Werkzeugen - reif für die Praxis? *Informatik-Spektrum*, Band Band 30, Ausgabe 5, 2007. (zitiert auf Seite 15)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den *[. . .]*