Inkrementelle Aktualisierungstechniken für Modelle und ihre Datenbankrepräsentation

Eine Bachelorarbeit von

Sebastian Hahner

an der Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation (IPD)

Erstgutachter:	Prof. Dr. Ralf H. Reussner
Zweitgutachter:	Jun.-Prof. Dr.-Ing. Anne Koziolek
Betreuer:	M. Sc. Stephan Seifermann
Zweiter Betreuer:	Dipl.-Inform. Jörg Henß

1. Juli 2016 – 31. Oktober 2016

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, 31. Oktober 2016

. . . . . .

(Sebastian Hahner)

Zusammenfassung

Modellgetriebene Software-Entwicklung (engl. *Model Driven Software Development*) steht für die Entwicklung von Software auf der Basis von Modellen. Diese stehen im Entwicklungsprozess auf einer Stufe mit Quelltext, der aus den Modellen generiert werden kann. Um die Arbeit im Team effizient zu ermöglichen, ist eine Synchronisierung der zu bearbeitenden Modelle nach einer Änderung zwischen allen Mitgliedern zwingend notwendig. Bei der Nutzung von Datenbanken zur Speicherung der Modelle ergeben sich dabei jedoch Probleme, da eine generierte Persistenzschicht häufig nur eine vollständige Ersetzung von bereits geladenen Elementen unterstützt. Eine Lösungsmöglichkeit ist die Detektion von Änderungen und die anschließende, inkrementelle Aktualisierung. Hierfür stehen bereits vorhandene Techniken zur Verfügung, welche kombiniert werden können. Im Rahmen dieser Arbeit sollten verschiedene Ansätze gefunden und anhand von Metriken bewertet werden. Eine Lösung des Problems ist die Kombination von aktiven Datenbanksystemen zur Änderungserkennung mit einer Persistenzschicht zur inkrementellen Aktualisierung einzelner Modellelemente. Bei der Untersuchung von UML-Modellen durchschnittlicher Größe (ca. 400 Elemente) ergibt sich bereits eine 25-fache Beschleunigung im Vergleich mit nicht-inkrementellen Methoden.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation und Problemstellung

Modellgetriebene Software-Entwicklung (engl. *Model Driven Software Development*) ermöglicht die Beschreibung von Software mit Hilfe von formalen Modellen. Einzelne Komponenten und Zusammenhänge können so auf einer höheren Abstraktionsebene dargestellt werden. Durch bessere Übersicht, Strukturierung und Automatisierung sollen somit Performanz, Skalierbarkeit, Wartbarkeit und Wiederverwendbarkeit verbessert werden [1, p. 13ff].

Formale Modelle sind Instanzen von Meta-Modellen. In diesen werden Struktur und Eigenschaften abhängig von der zu beschreibenden Domäne definiert. Mit Hilfe von Quelltextgeneratoren wird anschließend aus den Modellen Software erzeugt. Dieser Vorgang ist nicht zu verwechseln mit dem klassischen Entwurf und der anschließenden Implementierung von Software oder einer Automatisierung im Sinne eines IDE-Wizards [1, p. 13]. Die generierte Software ist nur ein Artefakt des Entwicklungsprozesses. In Kombination mit manuell implementierter Funktionalität ergibt sich somit lauffähige Software.

Um auf Modellen arbeiten zu können, ist neben einer konkreten Darstellung, wie etwa durch Text oder Grafik, auch eine Art der Persistenz notwendig. In der Praxis werden hierfür XML-Serialisierung oder Datenbanken verwendet.

In der (Software-)Entwicklung ist die Arbeit im Team nicht unüblich. Soll von mehreren Stellen aus verteilt auf Modelle zugegriffen werden, gelten spezielle Anforderungen. Um reibungsfreie Zusammenarbeit zu ermöglichen, muss jedes Teammitglied stets den aktuellen Stand sehen können. Hierfür müssen bei Änderungen andere Teilnehmer benachrichtigt und anschließend das Modell vollständig neu geladen oder inkrementell aktualisiert werden. Dies stellt insbesondere dann eine Herausforderung dar, wenn durch die Nutzung verschiedener Software keine einheitliche Persistenzschicht vorhanden ist. Entscheidend ist die Synchronisierung aller Zugriffspunkte auf das Modell.

Erschwerend kommt hinzu, dass eine vollständige Ersetzung eines bereits geladenen Modells oft keine Option darstellt. In einem typischen Szenario, wo in einem Editor aktiv am Modell gearbeitet wird, würden sonst bereits vorhandene Referenzen überschrieben. Ohne korrekte Zusammenführung verschiedener Zustände kann dies zu schwer lösbaren Laufzeitfehlern und Inkonsistenzen führen. An dieser Stelle fehlt ein generischer Ansatz zur Kopplung und Synchronisierung verschiedener Software-Lösungen mit ausreichender Performanz, um dem Echtzeitanspruch der Teamarbeit gerecht zu werden.

1.2 Zielsetzung

In dieser Arbeit sollen verschiedene Lösungsansätze für die soeben vorgestellte Problematik gefunden und anhand von zuvor definierten Metriken evaluiert werden. Hierfür werden vorhandene Technologien untersucht und unter Berücksichtigung der oben genannten Anforderungen kombiniert. Konkret betrachtet werden Konzepte zur Überwachung und Änderungsdetektion.

Die für diese Arbeit angewandte Methodik lässt sich grob in drei Abschnitte unterteilen. Zunächst werden verwandte Arbeiten auf Konzepte und Metriken untersucht. Auf dieser Wissensbasis wird anschließend eine Vorauswahl in Bezug auf den konkreten Anwendungsfall, der im nächsten Abschnitt beschrieben wird, getroffen und der erfolgversprechendste Ansatz prototypisch umgesetzt. Abschließend wird dieser dem herkömmlichen Ansatz, einer vollständigen Modellersetzung, gegenübergestellt.

1.3 Anwendungsfall

Der Anwendungsfall dieser Arbeit stellt das verteilte Erstellen und Editieren von UML-Modellen dar, wozu zwei unterschiedliche Software-Lösungen genutzt werden (siehe Abbildung 1.1). Die zugrundeliegende, relationale Datenbank wird von beiden Anwendungen geteilt, zwischen Anwendung und Datenbank liegen voneinander verschiedene Persistenzschichten. Zusätzlich ist eine Seite proprietär, bietet also keine Möglichkeit, den Datenbankzugriff direkt anzupassen. Außerdem ist somit das Datenbankschema vorgegeben. Änderungsverfolgung und inkrementelle Aktualisierung sind hier trotz allem möglich; beispielsweise durch Überwachung der Kommunikation oder der Datenbank.

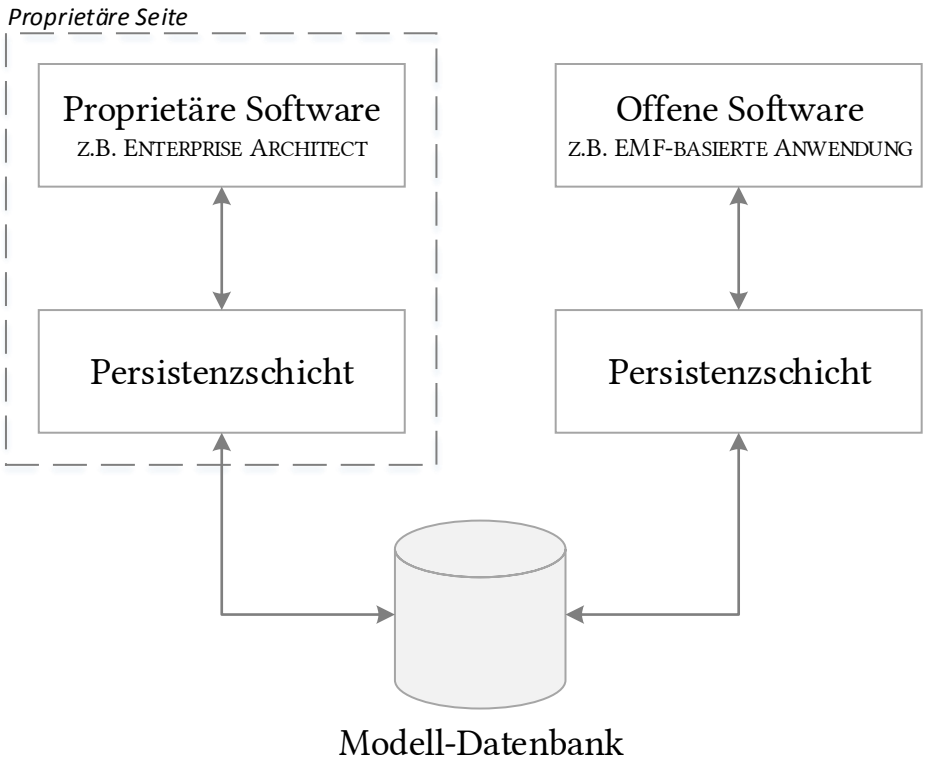


Abbildung 1.1: Szenario mit zwei Software-Lösungen und gemeinsamer Datenbank

Der Beitrag dieser Arbeit ist die Untersuchung und der Vergleich von inkrementellen Aktualisierungstechniken im Kontext von Modellen und deren Persistenz über eine gemeinsam genutzte Datenbank. Außerdem soll ein Lösungskonzept für die gegebene Problemstellung entwickelt werden, was anschließend anhand einer prototypischen Umsetzung und deren Evaluierung validiert wird. Eingeschränkt wird dieser Ansatz auf das konkrete Szenario des Anwendungsfalls, wobei gleichzeitige Änderungen nicht betrachtet werden. Anforderungen wie die relationale Datenbank und die Integration eines Legacy-Systems werden hierbei als gegeben angenommen.

1.4 Überblick

Die vorliegende Arbeit ist wie folgt strukturiert:

In **Kapitel 2** werden zunächst Grundlagen erklärt. Hierzu gehört modellgetriebene Software-Entwicklung sowie deren Umsetzung im Eclipse Modeling Framework. **Kapitel 3** behandelt verwandte Arbeiten, welche sowohl als Grundlage als auch Bausteine für diese Arbeit dienen. Die Methodik und das Vorgehen der Ausarbeitung werden in **Kapitel 4** erläutert.

Kapitel 5 liefert eine Übersicht, welche Techniken und Konzepte genauer untersucht wurden und wie sich diese im Kontext der inkrementellen Aktualisierung einsetzen lassen. Eine mögliche Kombination dieser Konzepte und deren Umsetzung wird in **Kapitel 6** vorgestellt.

Abschließend wird in **Kapitel 7** das Konzept anhand der protypischen Umsetzung validiert. **Kapitel 8** fasst die Arbeit zusammen und bietet Ansatzpunkte für zukünftige Untersuchungen.

2 Grundlagen

Im folgenden Kapitel werden für die Arbeit benötigte Grundlagen erläutert. Dazu gehören grundlegende Konzepte modellgetriebener Software-Entwicklung [1], sowie deren Umsetzung im Eclipse Modeling Framework [2] und Hibernate [3], einer Persistenzschicht des Frameworks. Als Erweiterung für die Datenbankschicht sind außerdem aktive Datenbanksysteme relevant [4].

2.1 Modellgetriebene Software-Entwicklung

Unter modellgetriebener Software-Entwicklung (engl. *Model Driven Software Development*) versteht man Techniken, um mit formalen Modellen Software zu erzeugen [1]. Das schließt insbesondere die automatisierte Generierung von Quelltext mit ein. Dieser Ansatz ist nicht mit der Darstellung eines Software-Systems als UML-Diagramm zu verwechseln. Während in diesem Fall das Modell nur zur Dokumentation dient, stehen in der modellgetriebenen Software-Entwicklung Modelle im Mittelpunkt.

Vorteil dieser Technik ist verringerte Komplexität durch die Arbeit auf einer höheren Abstraktionsebene. Hieraus kann eine höhere Entwicklungsgeschwindigkeit sowie bessere Performanz, Wiederverwendbarkeit, Wartbarkeit und Interoperabilität folgen. Notwendig hierfür ist ein wohldefiniertes Meta-Modell, also eine formalisierte Beschreibung einer Domäne.

Während im Kontext dieser Arbeit das Meta-Modell immer als fix angenommen wird, können sich Modelle als Instanzen eines Meta-Modells sehr wohl ändern. Um diese Modell-Evolution über die Zeit festzuhalten, ist eine Persistenzschicht notwendig, die ein Modell auf eine XML- oder Datenbankrepräsentation abbildet.

2.2 Eclipse Modeling Framework

Das Eclipse Modeling Framework (EMF) stellt umfangreiche Werkzeugunterstützung für modellgetriebenes Arbeiten in Eclipse bereit [2]. Hierzu gehören nicht nur grafische Editoren, Modellerzeugung aus Quelltext und Quelltextgenerierung aus Modellen, sondern auch das EMF-eigene Metameta-Modell Ecore. Im Kontext modellgetriebener Software stellen Metameta-Modelle eine weitere Abstraktionsebene dar. Konkrete Meta-Modelle sind Instanzen dieser.

Für die Persistenz von (Meta-) Modellen bringt EMF standardmäßig die Serialisierung zum XML Metadata Interchange-Format (XMI) mit. Alternativ hierzu kann auch eine datenbankfreundliche Persistenzschicht wie CDO verwendet werden [5]. Außerdem bietet EMF Laufzeitfunktionalität wie effizientere Introspektion und Modell-Änderungsbenachrichtigungen durch Adapter. Umgesetzt werden diese durch zusätzliche und für den Nutzer transparente Kapselung.

All diese Funktionalität ist durch Nutzung der API zusätzlich erweiterbar. So können beispielweise eigene Editoren und benutzerdefinierte Ansichten von Modellen durch EMF.Edit implementiert werden. Somit eignet sich EMF als Grundlage für offene modellgetriebene Software-Entwicklung.

2.3 Hibernate

Hibernate ist ein Open-Source-Persistenz-Framework für Java. Der Hauptverwendungszweck liegt in der Persistierung von Objekten in Datenbanken. Hierbei bietet Hibernate ebenfalls eine Lösung für den Paradigmenkonflikt zwischen Objektorientierung und relationalen Datenbanken durch Objekt-Relationales Mapping (ORM) [3].

Für den Zugriff auf die zugrundeliegende Datenbank kommt die Hibernate Query Language (HQL) zum Einsatz. Anfragen werden zur Laufzeit auf den konkreten SQL-Dialekt abgebildet, wodurch zusätzliche Technologieunabhängigkeit erzielt wird. Ebenfalls liefert Hibernate Funktionen zur Verbesserung der Performanz wie Caches, verzögertes Nachladen und der Verwaltung paralleler Zugriffe, inkl. dem gezielten Nachladen von Objekten.

Im Zusammenhang mit der Modellierung in EMF kommt zusätzlich Teneo zum Einsatz. Teneo ist eine Softwarelösung zur Generierung einer modellrelationalen Abbildung. In Hibernate integriert ermöglicht dies die Persistierung von EMF-Modellen.

2.4 Aktive Datenbanksysteme

Aktive Datenbanksysteme stellen eine Möglichkeit dar, auf Datenbankebene auf Änderungen direkt zu reagieren [4]. Separates Überwachen von Datenbankzugriffen oder periodische Überprüfung von Änderungen kann so vermieden oder vereinfacht werden.

Umgesetzt wird dies mit Hilfe von dreiteiligen Event-Condition-Action-Regeln. Im Event-Teil der Regeln wird spezifiziert, wo Änderungen auftreten müssen, damit das Anwenden einer Regel in Frage kommt. Hierbei sind sowohl das Ziel (z.B. die Tabelle) als auch die überwachte Operation (z.B. Einfügen oder Aktualisieren) frei einstellbar.

Im Condition-Teil können Eigenschaften des Events genauer betrachtet werden. Beispielsweise können hier kontextabhängig weitere Datenbankeinträge verglichen werden, um relevante Situationen herauszufiltern. Dieser Teil ist optional; wird er ausgelassen, wird die Regel bei jeder spezifizierten Änderung ohne Filter ausgeführt.

Im Action-Teil folgt zuletzt die Reaktion auf die soeben entdeckte Situation. Aktionen können sowohl neue Operationen und interne Änderungen sein, als auch Änderung oder Abbruch der Operation, die durch die ECA-Regeln entdeckt wurde. Auch externe Benachrichtigung ist grundsätzlich möglich.

Die effiziente Regelauswertung und interne Umsetzung geschieht für den Nutzer transparent. Eine Umsetzung von ECA-Regeln sind Trigger in MySQL [6]. Eine für die Ausarbeitung relevante Einschränkung hierbei ist, dass ein Trigger auf Änderungen genau einer Tabelle hört und diese nicht verändern kann, um Endlosschleifen zu vermeiden.

3 Verwandte Arbeiten

In diesem Kapitel wird ein Überblick über themenverwandte Arbeiten gegeben. Grundsätzlich bieten diese Arbeiten theoretische Grundlagen und Techniken, die als Bausteine für inkrementelle Aktualisierung eingesetzt werden können. Die vorliegenden Arbeiten lassen sich grob in drei Kategorien einteilen: Modellierung und Persistenz im Eclipse Modeling Framework, Änderungsverfolgung/-vergleich und Synchronisierung, sowie Konzepte in Bezug auf aktive Datenbanksysteme.

3.1 EMF Modelle und Persistenz

In **„The Design of a Robust Persistence Layer for Relational Databases"** [7] werden sowohl grundlegende Eigenschaften als auch wesentliche Design-Entscheidungen für die datenbankbasierte Persistenzschicht beschrieben. Insbesondere wird eine Übersicht über Arten der Persistenz geboten, welche einen guten Einstiegspunkt für diese Arbeit bietet. Unterschieden wird hierbei zwischen der statischen Bindung der Persistenz, einer zusätzlichen Abstraktion durch Datenklassen und einer separaten Persistenzschicht. Trotz der, durch den Anwendungsfall festgelegten, Anforderungen und Einschränkungen, ist eine allgemeine, modellunabhängige Lösung wünschenswert. Die Kapselung der Persistenzschicht ermöglicht einen generischen Zugriff auf das persistierte Modell; zusätzliches Wissen über die Abbildung des Modells ist nicht notwendig. Aus diesen Gründen wird dieser Ansatz für diese Arbeit vorgezogen.

Einen Überblick über in der Praxis verfügbare Persistenzschichten bietet **„Persistenz von EMF-Modellen"** [8]. Neben Grundlagen zur Persistenz und XML-Serialisierung werden vor allem datenbankbasierte Lösungen wie Teneo/Hibernate und Connected Data Objects (CDO) vorgestellt. Während Teneo/Hibernate für die Abbildung und Persistierung von Modellen in relationalen Datenbanken eingesetzt wird, kann CDO durch zusätzliche Abstraktion auch mit Objekt- oder NoSQL-Datenbanken eingesetzt werden. Beides sind für diese Arbeit relevante Techniken und werden in Kapitel 5.1 genauer beschrieben. Ebenfalls vorgestellt wird EMFStore, eine Technik zur Versionsverwaltung von EMF-Modellen.

Weitere Informationen zu Hibernate als Persistenzschicht können **„Object/Relational Mapping: Hibernate and the Entity Data Model (EDM)"** [9] entnommen werden. Hier werden technische Details von Hibernate wie Sessions und Transactions beschrieben und mit der Funktionsweise des Entity Data Model, einer Persistenzschicht für Microsofts .NET-Plattform, verglichen.

In **„Neo4EMF, A Scalable Persistence Layer for EMF Models"** [10] wird zuletzt eine Persistenzschicht für EMF vorgestellt, die eine Anbindung an die NoSQL-Graphen-Datenbank Neo4j ermöglicht. Neben der konkreten Umsetzung von Neo4EMF wurde die Thematik der Persistenz auch theoretisch untersucht. Ein Ergebnis ist, dass diese von großer Bedeutung ist, um Speicherung und Modell zu entkoppeln. Die anschließende

Evaluation zeigt zusätzlich die Nachteile der vollständigen Serialisierung auf: Da serialisierte XML-Dateien vollständig gelesen werden müssen und nicht kompakt im Speicher vorliegen, können bei dieser Technik erhebliche Leistungseinbußen entstehen. Im Hinblick auf Performanz und Skalierbarkeit liegt hier CDO mit bedarfgesteuertem Laden vorne. Allerdings beschränkt sich die Arbeit auf den Einsatz von NoSQL-Datenbanken und einer einheitlichen Persistenzschicht, ist für den vorliegenden Anwendungsfall mit relationaler Datenbank und verteiltem Zugriff also nicht anwendbar.

3.2 Modellvergleich und Synchronisierung

Um inkrementelle Aktualisierung zu ermöglichen, stehen verschiedene Konzepte und Techniken zur Verfügung. Hierzu gehört die Synchronisierung einzelner Modell-Elemente oder das Zusammenfügen verschiedener Zustände eines Modells mit Hilfe des Modellvergleichs. Im Kontext des Anwendungsfalls bleibt das Meta-Modell hierbei über den gesamten Zeitraum der Aktualisierung unverändert.

Werden Teile eines Modells nachgeladen, müssen diese in das bereits geladene Modell eingefügt und ggf. Referenzen angepasst werden. Hierfür müssen Modelle verglichen und zusammengeführt werden. Einige Ansätze hierfür werden in **„Different Models for Model Matching: An Analysis of Approaches to support Model Differencing“** [11] vorgestellt. Für die Nutzung mit EMF ist insbesondere **EMF Compare** [12] relevant, eine Erweiterung zum Vergleichen und Zusammenfügen von EMF Modellen.

Mit Modellvergleich im Eclipse Modeling Framework beschäftigt sich auch **„Model Differences in the Eclipse Modelling Framework“** [13]. Untersucht werden die Punkte Berechnung, Repräsentation und Visualisierung von Modell-Differenzen. Als konkrete Technik wird ebenfalls EMF Compare angesprochen. Eine detailliertere Untersuchung von EMF Compare findet sich in **„Presentation of EMF Compare Utility“** [14]. Hier werden Details zur Funktionsweise und Beispiele zur korrekten Verwendung aufgeführt.

3.3 Aktive Datenbanksysteme und Auditing

Thematiken wie das Eclipse Modeling Framework, Persistenzschichten und Modellvergleich ermöglichen Aktualisierungskonzepte auf Anwendungsebene. Diese können durch die Erweiterung der Funktionsweise der Datenbankschicht ergänzt werden. In **„Distribution and Synchronisation of Engineering Information using Active Database Technology“** [15] kommen hierfür aktive Datenbanksysteme zum Einsatz. ECA-Regeln können verwendet werden, um Änderungen am Datenbestand zu entdecken und entsprechende Maßnahmen wie die Weiterleitung von Informationen über die Änderung in die Wege zu leiten.

In **„Active Information Delivery in a CORBA-based Distributed Information System“** [16] wird diese Vorgehensweise erneut aufgegriffen. Hierbei liegt der Fokus auf dem verteilten Arbeiten auf einer Datenbank. Das Ziel ist die Benachrichtigung bei kritischen Änderungen der zu verwaltenden Daten. Für die Umsetzung wird der Einsatz von ECA-Regeln in CORBA-basierten Systemen mit Hilfe von Wrappern diskutiert. Da CORBA für diese Arbeit als Zwischenanwendung nicht in Frage kommt, ist auch diese Arbeit vor allem wegen konzeptioneller und theoretischer Inhalte relevant.

ECA-Regeln sind ebenfalls die Grundlage in **„Implementing Agent Coordination for Workflow Management Systems Using Active Database Systems“** [17]. Im Kontext des Workflow-Managements ist das Ziel der Arbeit die Koordinierung mehrerer Nutzer mit Hilfe von aktiven Datenbanksystemen. Konkret wird die Modellierung, Ausführung und Synchronisierung von Prozessen diskutiert.

Eine weitere Möglichkeit der Änderungsverfolgung bietet die Überwachung der Datenbankoperationen in einer zusätzlichen Schicht zwischen Anwendung und Datenbank. Diese Möglichkeit wird in **„A Framework for Database Auditing“** [18] diskutiert. Untersucht werden passive und aktive Ansätze, um Änderungen in Datenbanken zu überwachen und ggf. im Fall eines Angriffs zu verhindern. Der klare Vorteil einer entkoppelten, passiven Lösung ist, dass die Performanz der Datenbank in keiner Form beeinflusst wird. Als konkrete Umsetzung wird die Untersuchung von Datenverkehr zur Änderungsverfolgung vorgeschlagen. Allerdings ist davon auszugehen, dass diese Implementierung einen erheblichen Mehraufwand darstellt und deswegen im Rahmen dieser Arbeit nicht umsetzbar ist. Eine Alternative stellen aktive Ansätze wie Proxies, z.B. der MySQL-Proxy [19], dar. Ein weiterer Vorteil dieser Technik ist, dass diese auch zum Einsatz kommen kann, wenn kein direkter Zugriff auf die Datenbank durch eingeschränkte Berechtigungen möglich ist.

4 Methodik

Nachdem in den vorherigen Kapiteln bereits Grundlagen und verwandte Arbeiten vorgestellt wurden, folgt nun eine kurze Zusammenfassung der Vorgehensweise dieser Arbeit. Hierbei wurde die Erarbeitung, Umsetzung und Evaluation von Konzepten und Techniken zur inkrementellen Aktualisierung in einzelne Arbeitsabschnitte unterteilt.

Das Vorgehen lässt sich grob in zwei Teile gliedern: Zunächst werden verwandte Arbeiten auf Techniken und theoretische Grundlagen untersucht und Metriken zur Bewertung der Ansätze herausgearbeitet. Auf deren Basis erfolgt die Vorauswahl für eine prototypische Umsetzung der erfolgversprechendsten Kombinationen und deren iterative Implementierung. Abschließend werden diese jeweils evaluiert. Die einzelnen Schritte werden im Folgenden genauer erläutert. Eine Übersicht kann der Abbildung 4.1 entnommen werden.

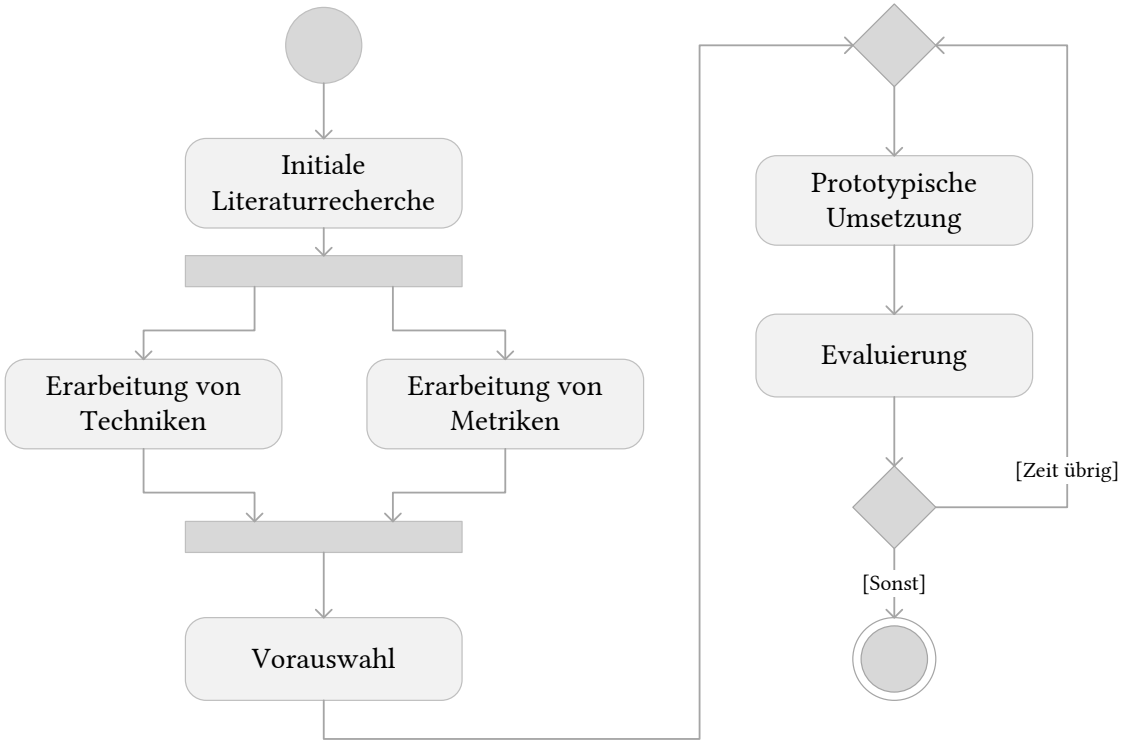


Abbildung 4.1: Ablauf der Arbeit

Die Bearbeitung beginnt mit der Literaturrecherche. Während dieser sollen ein erster Überblick über das Themengebiet gewonnen und Grundlagen erarbeitet werden, die für das Verständnis und das Erarbeiten von Techniken und Metriken notwendig sind. Außerdem soll die grundsätzliche Durchführbarkeit der Arbeit sichergestellt werden. Das Ergebnis dieses Schrittes ist Kapitel 2: Grundlagen und Kapitel 3: Verwandte Arbeiten.

Im nächsten Schritt werden aus dem Ergebnis der Literaturrecherche Techniken erarbeitet. Einige relevante Ansätze wurden bereits in Kapitel 3 vorgestellt. Obwohl im Beispielszenario (siehe Kapitel 1) einige Einschränkungen durch die proprietäre Seite und die relationale Datenbank vorgegeben sind, stehen dennoch einige Konzepte zur Verfügung, die in ähnlichen Szenarien bereits erfolgreich eingesetzt wurden. Eine Übersicht bietet Kapitel 5.

Parallel hierzu werden Metriken erarbeitet. Diese bieten als Werkzeug zur Bewertung der Ansätze die Grundlage für die Validierung der Umsetzung in Kapitel 7. Verschiedene, unabhängige Metriken sind vor allem relevant, weil es je nach Szenario eine andere optimale Kombination vorhandener Technologien geben kann. Eine für einen Anwendungsfall optimale Lösung muss also nicht zwangsläufig allgemeingültig sein. Ebenfalls kann die Relevanz mancher Metriken von Fall zu Fall variieren.

Nachdem die Recherche und sowohl die Erarbeitung von Techniken als auch Metriken weitestgehend abgeschlossen ist, erfolgt eine Vorauswahl für die anschließende prototypische Umsetzung. Hierbei werden Technologien und Ergebnisse vorausgehender Arbeiten kombiniert.

Nach der Vorauswahl folgt im zweiten Teil der Arbeit die iterative Umsetzung. Abhängig von der Dauer der Implementierung und der Vorauswahl werden mehrere Iterationen durchgeführt. Das Ergebnis einer vollständigen Iteration wurde in Kapitel 6: Umsetzung festgehalten. Ebenfalls werden hier alternative Konzepte für Umgebungen mit anderen Anforderungen/Einschränkungen vorgestellt, die sich im Rahmen einer weiteren Iteration ergeben haben. Im konkreten Szenario sollen die Modellierungssoftware Enterprise Architect und eine auf dem Eclipse Modeling Framework basierende Anwendung über eine relationale Datenbank miteinander verknüpft werden.

In der anschließenden Evaluierung in Kapitel 7 wird die Umsetzung anhand der prototypischen Implementierung und der bereits definierten Metriken validiert. Insbesondere liegt hierbei der Fokus auf dem Vergleich der Umsetzung mit herkömmlichen nicht-inkrementellen Lösungen.

5 Bausteine für Aktualisierungstechniken

Im folgenden Kapitel werden Konzepte und Techniken für Änderungsdetektion und (inkrementelle) Aktualisierung aufgezählt, welche sich aus Grundlagen und verwandten Arbeiten ergeben. Auf Basis dieser Aufzählung erfolgt auch die Vorauswahl für die genauere Untersuchung und konkrete Umsetzung. Wie schon in Kapitel 4 beschrieben, ist eine allgemeine Lösung unwahrscheinlich; stattdessen ist davon auszugehen, dass für jeden Anwendungsfall eine individuelle Kombination von vorhandenen Techniken zum besten Resultat führt.

Aufteilen lässt sich der Aufgabenbereich in Änderungsdetektion und Aktualisierung. Einen Überblick, wo diese Techniken im Anwendungsfall (siehe Kapitel 1) einzuordnen sind, liefert Abbildung 5.1. Relevante Konzepte sind u.a. Serialisierung① und Persistenzschichten②. Während erstere die Umwandlung einer Objekthierarchie in eine sequentielle Darstellung ausdrückt, ermöglicht eine Datenbankanbindung durch eine Persistenzschicht die Speicherung der Objekte in einer relationalen Datenbank. Um Änderungen zu erfassen und zu verarbeiten, eignen sich ECA-Regeln in aktiven Datenbanksystemen③ und Beobachter④. Ist ein direkter Eingriff in die Datenbank, z.B. wegen beschränkter Berechtigungen, nicht möglich, können Änderungen auch mit Hilfe von Proxies⑤ überwacht werden. Eine Verarbeitung der Änderungen ist mit Hilfe von Techniken zum Modelvergleich⑥ möglich. Zuletzt können Änderungen auch durch Aktives Warten⑦, also dem Neuladen von Teilen einer Objekthierarchie erhalten und verarbeitet werden.

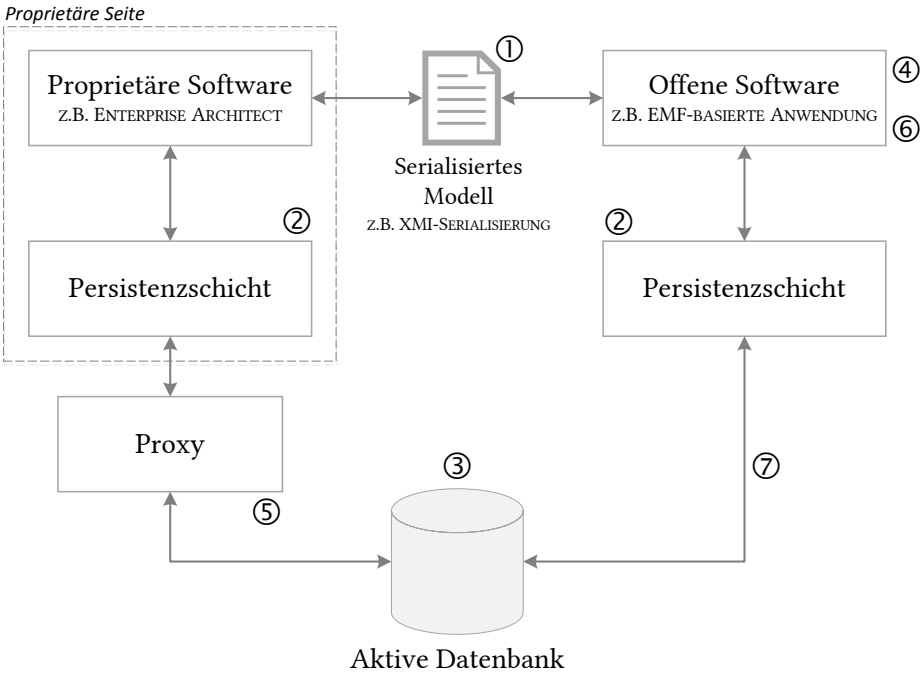


Abbildung 5.1: Bausteine für inkrementelle Aktualisierung im Anwendungsfall

5.1 Persistenzschicht

Um Objekte und Modelle zu speichern gibt es zwei Möglichkeiten: Entweder die Implementierung des für die Persistierung benötigten Quelltextes in den Klassen selbst, oder die Verwendung einer Persistenzschicht. Da eine direkte Implementierung eine lose Kopplung von Modell und Persistenz verhindert, ist der zweite Ansatz auf Grund der besseren Flexibilität und Wartbarkeit vorzuziehen [10].

Zur Speicherung stehen dateibasierte Techniken wie Serialisierung zu XML oder XMI (siehe Kapitel 5.2) oder die Anbindung an (relationale) Datenbanksysteme zur Verfügung. Neben einer für den Entwickler möglichst transparenten Verwaltung der Persistenz zeichnen sich Persistenzschichten u.a. durch folgende Funktionalität aus: Unterstützung für Multi-Objekt-Anfragen, Zusammenfassen von Anfragen in Transaktionen, Erweiterbarkeit und Anpassbarkeit, bedarfgesteuertes Laden, Unabhängigkeit von der zugrundliegenden Datenbanksoftware und Ausführbarkeit von objektorientierten Anfragen an den Datenbestand [7].

Für die Entwicklung im Eclipse Modeling Framework stehen neben der nativen Unterstützung für Serialisierung zwei Lösungsansätze unter Verwendung einer Datenbank zur Verfügung: Connected Data Objects (CDO) und Teneo/Hibernate. Die Idee von CDO ist die Verbindung des Eclipse Modeling Frameworks mit einem zentralen CDO Model-Repository. Während die eigentliche Kommunikation zwischen dem Repository und EMF stattfindet, ist das Model-Repository an ein Persistierungs-Backend wie NoSQL- oder relationale Datenbank angebunden. Diese Kapselung ermöglicht die Unabhängigkeit vom konkreten Datenbanksystemanbieter. Außerdem unterstützt CDO bereits vorgestellte Techniken wie bedarfgesteuertes Laden und Aktualisierung von Modellen aus dem Datenbestand [8].

Eine weitere Vorgehensweise ist die Verwendung von Teneo in Verbindung mit Hibernate. Da diese Technik auch für die prototypische Umsetzung in Kapitel 6 eingesetzt wurde, finden sich Grundlagen bereits in Kapitel 2. Während Teneo für die Generierung einer Modell-Abbildung für Hibernate zuständig ist, ist Hibernate ein reines Java-Persistenzframework. Hibernate erfüllt alle bereits angesprochenen Anforderungen wie Transaktionen und eine eigene Anfragesprache namens HQL (*Hibernate Query Language*). Für diese Arbeit ebenfalls elementar sind grundlegende Funktionen zum Abrufen, Speichern und inkrementellen Nachladen von Objekten aus dem relationalen Speicher [20].

5.2 Serialisierung

Serialisierung bedeutet die Umformung von strukturierten Daten (eines Modells) in eine sequentielle Repräsentation. Das Eclipse Modeling Framework unterstützt diese nativ [2] und bietet Unterstützung zur Erzeugung von XMI-Dateien (*XML Metadata Interchange*), einem standardisierten Austauschformat der Object Management Group [21]. Ein Ausschnitt aus einem serialisierten Modell ist in Quelltext 5.1 zu sehen.

```
<eaobjectmodel:Package Name="Model" PackageID="1">
  <Packages Name="CD" PackageID="2" >
    <Elements ElementID="2" Name="SimpleClass" Version="1.0" />
  </Packages>
</eaobjectmodel:Package>
```

Quelltext 5.1: Gekürzter Ausschnitt einer XMI-Datei

Obwohl es sich hierbei um ein Austauschformat zur Kommunikation zwischen verschiedenen Softwarelösungen handelt, ist der Einsatz im gegebenen Szenario nicht praktikabel. Zwar ist es möglich, einzelne Zustände des Modells auszutauschen und auf textueller oder Modell-Ebene zu vergleichen, allerdings bringt die XMI-Serialisierung zwei erhebliche Nachteile mit sich. Zum einen sind XML-Dateien für Menschen lesbar gemacht, was Einschränkungen in der Formatierung mit sich bringt, die sich negativ auf die Lese- und Schreibperformanz auswirken können. Zum anderen ist der wahlfreie Zugriff bei einer dateibasierten Speicherung nicht ohne weiteres möglich [8], was den einfachen Zugriff auf Elemente eines Modells erschwert. Betrachtet man unter diesen Voraussetzungen die Performanz des Lese/Schreib-Prozesses, liegen datenbankbasierte Persistenzschichten vorne [10].

5.3 Modellvergleich

Wird keine Persistenzschicht verwendet oder liefert diese keine Mechanismen zum inkrementellen Nachladen von veränderten Elementen, können diese auch von Hand neu geladen und anschließend mit dem lokalen Modell zusammengeführt werden. Für die zustandsbasierte Synchronisation ist zunächst ein Modellvergleich notwendig. Grundlagen zur Differenz-Berechnung und hierfür angewandter Methodik ist nicht Teil dieser Arbeit und kann [11] entnommen werden.

Die bekannteste Implementierung für das Eclipse Modeling Framework ist EMF Compare [12]. Der eigentliche Vorgang der Differenzbildung zweier Modelle ist für den Entwickler

transparent, kann aber bei Bedarf angepasst werden. Die Verwendung von EMF Compare geschieht auf einem ähnlich hohen Level wie auch die Persistierung mit Hibernate: Zunächst wird ein neu geladenes Modell (oder Teile von diesem) mit dem lokalen Modell verglichen, anschließend alle relevanten Änderungen ins lokale Modell integriert.

Der große Vorteil dieses Vorgehens ist, dass Änderungen analog zum inkrementellen Nachladen einer Persistenzschicht übernommen werden. Auf Objektebene findet, wenn nicht absolut notwendig, keine Ersetzung statt. Somit müssen keine bestehenden Referenzen angepasst werden, was zu Laufzeitfehlern führen kann.

5.4 Beobachter

Um Änderungen im geladenen Modell speichern zu können, müssen diese zunächst erkannt werden. Bietet die Persistenzschicht keine Funktionalität für die automatische Speicherung bei Änderungserkennung an, können lokale Änderungen leicht mit Hilfe von Beobachtern überwacht werden. Im Eclipse Modeling Framework heißen diese Adapter, da sie über ihre eigentliche Funktionalität hinaus oft für die Erweiterung von Objekteigenschaften eingesetzt werden [2].

Adapter wurden nach dem Beobachter-Entwurfsmuster implementiert. Wird eine Änderung an einem Objekt vorgenommen, werden alle zuvor an diesem Objekt registrierten Beobachter synchron benachrichtigt. Ebenfalls für die Überwachung vollständiger Objekthierarchien bietet EMF vorgefertigte Implementierungen. Ein beispielhafter Ablauf ist in Abbildung 5.2 zu sehen.

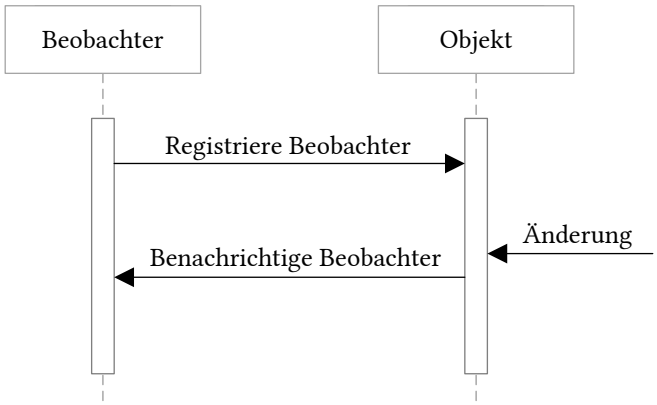


Abbildung 5.2: Funktionsweise eines Beobachters

In proprietären Umgebungen ist die Funktionalität von Beobachtern potenziell nicht gegeben. Übernimmt die proprietäre Persistenzschicht nicht automatisch die Persistierung lokaler Änderungen, kann das Verhalten von Beobachtern mit Hilfe eines lokalen Proxys nachgeahmt werden. Hierfür ist jedoch eine überwachbare Verbindung zu einer Datenbank notwendig. Details zu diesem Vorgehen sind in Kapitel 6.4 zu finden.

5.5 Datenbank-Trigger

Mit Hilfe von Beobachtern ist es möglich, Änderungen am lokalen Modell zu erkennen, die vom Nutzer der Software selbst verursacht wurden. Für die Erkennung von Änderungen anderer Nutzer lässt sich diese Technik nicht einsetzen, da hierfür erst der gesamte Datenbestand aus der Datenbank neu geladen werden müsste.

Im gegebenen Anwendungsfall erfolgen der Austausch und die Synchronisation des Modells über eine geteilte Datenbank. Somit kann auch die globale Änderungserkennung in diese ausgelagert werden, beispielweise mit Hilfe von ECA-Regeln in aktiven Datenbanksystemen. Grundlagen hierzu wurden bereits in Kapitel 2.4 erklärt.

Eine Umsetzung von ECA-Regeln sind Trigger in MySQL-Datenbanksystemen. Trigger werden mit Hilfe einer zweigeteilten Anfrage erstellt: Im Kopf-Teil werden Attribute, wie der Name des Triggers, über den dieser später referenziert werden kann, und die Tabelle, auf deren Änderung reagiert werden soll, festgelegt. Ebenfalls wird hier eingestellt, ob der Trigger auf das Einfügen, Ändern oder Löschen von Einträgen reagieren soll. Im Hauptteil des Triggers stehen optional weitere Bedingungen an die Änderung in Form von Wenn-Dann-Sonst-Blöcken und neben der Filterung die eigentliche Aktion als klassische SQL-Anweisungen [6].

Ein beispielhafter Ausdruck zur Erzeugung eines MySQL-Triggers wird in Quelltext 5.2 gezeigt. In diesem Beispiel wird ein Trigger namens *mySimpleTrigger* definiert, der nach jedem Update in der bereits vorhandenen Tabelle *myTable* das Attribut *myAttribute* auf Änderungen überprüft. Diese werden in einer separaten Tabelle dokumentiert.

```
CREATE TRIGGER `mySimpleTrigger` AFTER UPDATE ON `myTable`
FOR EACH ROW
      BEGIN
            IF NEW.myAttribute <> OLD.myAttribute
            THEN
                  REPLACE INTO myLoggingTable VALUES (OLD.ID, NOW());
            END IF;
      END
$
```

Quelltext 5.2: SQL-Ausdruck zur Erzeugung eines Datenbank-Triggers

Eine Einsatzmöglichkeit von Datenbank-Triggern im Kontext des Anwendungsfalls ist die Dokumentation aller relevanten Änderungen in separate Protokolltabellen. Die Auswahl relevanter Ergebnisse wird über die Auswahl der zu überwachenden Tabellen und dem Bedingungs-Block festgelegt. Der Vorteil dieser direkten Lösung ist, dass Änderungen ohne viel Zusatzaufwand erkannt und getrennt vom konkreten Datenbestand gesammelt werden. Das erleichtert den Zugriff und die Verarbeitung auf der Anwendungsschicht.

Eine alternative Vorgehensweise ist die Verwendung von nutzerdefinierten Funktionen (*User Defined Functions*, *UDF*) [22]. Mit Hilfe des MySQL-eigenen Befehls *sys_exec()* ist es möglich, in einem Trigger externe Programme auszuführen. Somit kann die Änderungserkennung in der Datenbank verbleiben, die Verarbeitung aber getrennt davon passieren. Ein Grund hierfür kann die bessere Skalierbarkeit durch eine dedizierte Änderungsver-

waltung sein. Ebenfalls ist ein externer Prozess nicht an die Datenbank gebunden, weswegen hier grundsätzlich mehr Möglichkeiten der Verarbeitung existieren. Nachteil eines externen Prozesses ist neben der Aufspaltung der Entwicklung die potenziell schlechtere Laufzeit, wenn in jedem Trigger-Aufruf zusätzliche blockierende Aufrufe der externen Anwendung erfolgen. Hinzu kommt die Tatsache, dass externe Prozesse nicht in Transaktionen funktionieren, also nicht mehr rückgängig gemacht werden können. Es wurde im Rahmen dieser Arbeit aus Zeitgründen nicht untersucht, wann die Nutzung von UDF wirklich einen Vorteil darstellt.

5.6 Proxies

Der Einsatz von Proxies wurde bereits in Kapitel 3.3 diskutiert. Datenbank-Proxies werden in die Verbindung zwischen Anwendung und Datenbank eingefügt und erhalten somit vollen Zugriff auf den Datenverkehr. Anfragen können abhängig von der gewünschten Funktionalität dokumentiert, ausgewertet, umgeschrieben oder abgelehnt werden.

Der große Vorteil von Proxies ergibt sich aus der Kapselung: Handelt es sich bei der Datenbank- oder Anwendungsschicht um ein geschlossenes System und können Änderungen deswegen nicht direkt erkannt und verarbeitet werden, kann dieser Schritt in einen Proxy ausgelagert werden. Neben der Einrichtung des Proxys ist in der Anwendung nur eine Änderung der Adresse der Datenbank zur Adresse des Proxys notwendig.

Die Nachteile von Proxies sind die erhöhte Komplexität durch eine zusätzliche Zwischenschicht, was sich durch die synchrone Auswertung des Datenverkehrs auch negativ auf die Performanz auswirken kann. Ebenfalls ist die Auswertung prinzipiell aufwendiger, da alle benötigten Informationen erst aus der textuellen Repräsentation der Datenbank-Anfrage extrahiert werden müssen. Aus diesem Grund wurden für die Umsetzung in Kapitel 6 Datenbank-Trigger für die Änderungserkennung vorgezogen. Der beispielhafte Einsatz von MySQL-Proxies wird in Kapitel 6.4 diskutiert.

5.7 Aktives Warten

Aktives Warten steht im Kontext dieser Arbeit für das repetitive/zyklische Laden von Informationen (z.B. Modell-Elementen) aus einer Datenbank. Abhängig von der gewählten Kombination von Techniken kommt Aktives Warten an mehreren Stellen zum Einsatz.

Kann wegen Einschränkungen (z.B. fehlender Berechtigung) die Anwendungsschicht oder Datenbank nicht erweitert werden, ist das vollständige Nachladen und Ersetzen oder Zusammenführen eines Modells die einzige Möglichkeit, Änderungen einzupflegen. Diese Vorgehensweise wurde bereits im Kontext des Modellvergleichs vorgestellt. Ist bereits eine Technik vorhanden, um Änderungen nachzuverfolgen und zu analysieren, kann das Nachladen auch inkrementell geschehen und auf Teile des Modells reduziert werden. Je nach Granularität der nachzuladenden Modell-Teile kann dies signifikante Auswirkungen auf die Performanz haben.

Ebenfalls kann Aktives Warten verwendet werden, um Informationen aus der Datenbank abzurufen, die nicht zum Modell selbst gehören. Hierzu gehören auch Ausschnitte einer Änderungs-Dokumentation, wie sie z.B. durch Datenbank-Trigger oder Proxies erstellt werden kann. Im Kontext einer klassischen Client-Server-Architektur steht dem Aktiven

Warten ereignisgesteuerte Aktion gegenüber. Dies stellt im konkreten Szenario einen erheblichen Mehraufwand der Implementierung dar, da statt generischen Serverabfragen ein System zur Benachrichtigung aller Nutzer entworfen werden muss. Daher wurde dieser Ansatz im Rahmen der Arbeit nicht weiterverfolgt.

6 Umsetzung

In diesem Kapitel wird die konkrete Kombination und Umsetzung der in Kapitel 5 vorgestellten Aktualisierungstechniken diskutiert. Bezogen auf die in Kapitel 4 dargestellte Methodik entspricht dies der prototypischen Umsetzung einer Iteration. Ebenfalls werden alternative Konzepte vorgestellt, die zum Einsatz kommen können, wenn durch die Umgebung Einschränkungen gegeben sind. Diese Konzepte haben sich im Rahmen einer weiteren Iteration ergeben.

Folgende Techniken wurden für die prototypische Umsetzung ausgewählt:

- **Datenbank-Trigger** zur Änderungsdetektion in einer MySQL-Datenbank
- Das Eclipse Modeling Framework mit Hibernate als **Persistenzschicht**, da hierfür bereits die Grundlagen für die konkrete Umsetzung vorhanden sind
- **Aktives Warten**, um aktuelle Änderungen aus der Datenbank abzurufen
- **Beobachter**, welche EMF ebenfalls in Form eines Adapter Frameworks mitbringt

Ein Überblick, wie diese Techniken im Gesamtbild einzuordnen sind, kann Abbildung 5.1 in Kapitel 5 entnommen werden.

Wird von einem anderen Nutzer eine Änderung am Modell vorgenommen und in die Datenbank geschrieben, wird dieser Vorgang zunächst von einem Datenbank-Trigger erkannt und in einer Protokolltabelle dokumentiert. In regelmäßigen Abständen wird in der Persistenzschicht diese Tabelle auf neue Einträge geprüft. Gibt es diese, werden die betroffenen Elemente inkrementell neugeladen. Die hierfür benötigte Funktionalität liefert Hibernate. Lokale Änderungen werden ebenfalls unterstützt. Diese werden mit Hilfe von Beobachtern erkannt und mit der Datenbank synchronisiert. Ein Beispiel-Szenario wird in Abbildung 6.1 dargestellt. Hierbei wird zwischen globalen Änderungen, also einer Änderung eines anderen Nutzers, die den Zustand des persistierten Modells verändert hat, und einer lokalen Änderung unterschieden.

In den folgenden Unterkapiteln werden Details zur konkreten Umsetzung erläutert. Ebenfalls werden alternative Konzepte bzw. die Variation der Umsetzung diskutiert. Hierzu gehört die Änderungserkennung durch Proxies und die Modellaktualisierung durch Modellvergleich.

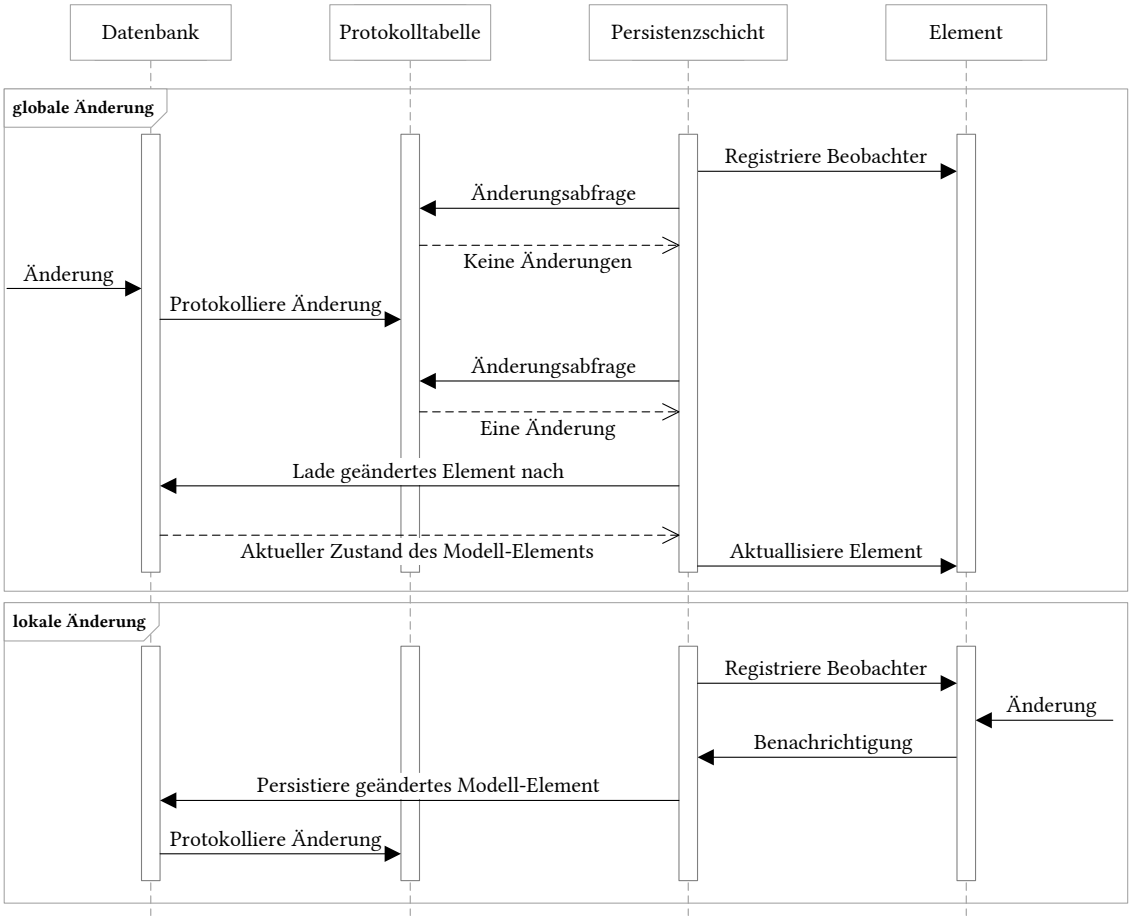


Abbildung 6.1: Beispiel-Szenario der Umsetzung

6.1 Änderungserkennung

Zur Änderungserkennung werden ECA-Regeln in Form von Triggern einer MySQL-Datenbank verwendet. Diese Technik hat sich im Rahmen der Voruntersuchung als effizienteste herausgestellt, da Änderungen so direkt wie möglich verarbeitet werden. Durch Trigger ist keine zusätzliche Schicht (wie z.B. ein Proxy) notwendig und der geringe Zusatzaufwand fällt bei jeder Änderung direkt an. Aufwendige Modellvergleiche sind somit ebenfalls überflüssig. Die Implementierung ist einfach; schon mit wenigen Zeilen SQL-Quelltext können alle Änderungen einer Tabelle verarbeitet und separat gesammelt werden (siehe Kapitel 5.5).

Um alle relevanten Änderungen durch Trigger dokumentieren zu lassen, müssen vor der Implementierung zunächst Informationen über das zu persistierende Meta-Modell gesammelt werden. Wichtig sind hierbei alle Tabellen und Attribute, in denen Modellinformationen in der Datenbank gespeichert werden. Diese Informationen können entweder von Hand gewonnen oder aus dem Mapping der Persistenzschicht extrahiert werden.

Für jede Tabelle sind anschließend, abhängig von den gegebenen Anforderungen, drei Trigger notwendig: Jeweils einer für das Einfügen, Ändern und Löschen von Zeilen einer Tabelle. Ebenfalls ist mindestens eine weitere Tabelle notwendig, in der alle Änderungen

gesammelt werden. Änderungen können zentral für alle Tabellen oder separat gesammelt werden. Auch zusätzliche Tabellen je nach Art der Änderung sind möglich. Für den Protoypen wurde eine Protokolltabelle pro Tabelle verwendet.

Optional möglich sind zusätzliche, geplante Events der Datenbank, die in einem gesetzten Intervall veraltete Einträge aller Protokolltabellen löschen. Finden häufige Änderungen statt, kann somit der zusätzliche Speicher- und Rechenaufwand minimiert werden. Das Intervall der Säuberung sollte in jedem Fall höher liegen als das Intervall der Änderungsabfrage der Anwendungsschicht; andernfalls besteht die Gefahr, dass Änderungen zu früh verworfen und nicht von den Systemen aller Nutzer verarbeitet werden. Der vollständige Quelltext für die Erstellung von Triggern, Protokolltabellen und Events für eine zu überwachende Tabelle ist im Anhang A.1 zu finden.

Diese Umsetzung ist modellunabhängig, da dokumentierte Änderungen nur zeitlich begrenzt für die Änderungsbenachrichtigung relevant sind. Wird das Modell vollständig ausgetauscht, reicht die Säuberung aller Protokolltabellen aus. Ein Problem ergibt sich allerdings bei der Evolution des Meta-Modells: Um die Performanz zu maximieren, überwacht ein Trigger nur eine Tabelle. Änderungen des Meta-Modells erfordern deshalb auch Anpassungen der Trigger und Protokolltabellen. Andernfalls kann die korrekte Funktionalität nicht gewährleistet werden.

Im Rahmen der prototypischen Umsetzung wurde ein Trigger-Generator [23] implementiert, um den Umgang mit der Meta-Modell-Evolution zu vereinfachen. Das Sammeln von Informationen über das Meta-Modell, die Interpretation dieser und die Umformung in SQL-Anweisungen zur Erstellung der Trigger wird hierbei automatisiert.

Als Eingabe erhält der Generator die von Teneo/Hibernate generierte Abbildungsdatei. Diese enthält für alle Modell-Elemente und deren Attribute Informationen über die Abbildung auf Tabellen und Spalten einer relationalen Datenbank. Ein Ausschnitt dieser Abbildung ist in Quelltext 6.1 zu sehen. Aus diesen Informationen werden relevante Tabellen abgeleitet, anschließend werden SQL-Befehle zur Erstellung von Triggern, Tabellen und Events erzeugt.

```
<hibernate-mapping>
    <class entity-name="Attribute" table="`t_attribute`">
        <id name="AttributeID" type="java.lang.Long">
            <column name="`ID`"/>
        </id>
        <property name="AllowDuplicates" type="java.lang.Boolean">
            <column name="`AllowDuplicates`"/>
        </property>
    </class>
</hibernate-mapping>
```

Quelltext 6.1: Ausschnitt der Objekt-Relationalen Abbildung von Hibernate

Eine potenzielle Beschleunigung ergibt sich aus dem zusätzlichen Zusammenfassen von Änderungen. Werden Änderungen für jede relevante Tabelle separat gesammelt, bietet sich die Erstellung einer Master-Tabelle an. In dieser wird stets das Datum der letzten Änderung festgehalten. Werden Änderungen in regelmäßigen Abständen von der Persistenzschicht abgefragt, reicht somit eine einzelne Anfrage aus, ob im letzten Aktualisierungsintervall überhaupt eine Änderung erkannt wurde. Ist dies nicht der Fall, erüb-

rigen sich weitere Anfragen an den konkreten Protokolltabellen. Nutzer einer Modellierungssoftware ändern das zu bearbeitende Modell üblicherweise nicht im Sekundentakt. Unter der Annahme, dass Änderungen in Bezug auf das gewählte Aktualisierungsintervall selten vorkommen, ergibt sich hierbei ein Gewinn an Performanz.

Unabhängig davon, ob Trigger und Protokolltabellen von Hand implementiert oder generiert wurden, bieten diese eine Lösung für die Detektion von Änderungen auf dem Datenbestand. Protokolltabellen bilden dabei den Einstiegspunkt, um den Datenbestand auf kürzliche Änderungen zu überprüfen und diese auf der Anwendungsschicht zu protokolieren oder betroffene Elemente inkrementell zu aktualisieren. Für die Protokollierung ist keine weitere Aufwendung auf einer anderen Schicht notwendig; Datenbank-Trigger bilden ein abgeschlossenes System.

6.2 Änderungsbenachrichtigung

Die Umsetzung mit Hilfe von Triggern und Protokolltabellen ermöglicht es, kürzliche Änderungen direkt aus der Datenbank abzurufen. Dieses Vorgehen minimiert den Zusatzaufwand in der Anwendungsschicht. Für die Änderungsbenachrichtigung sind dennoch eine regelmäßige Abfrage der Protokolldatenbanken und die korrekte Interpretation dieser Informationen notwendig.

Im konkreten Szenario werden Enterprise Architect und das Eclipse Modeling Framework mit Hibernate als Persistenzschicht an die Nutzung der Protokolltabellen angepasst [24]. Da das Meta-Modell von Enterprise Architect (als proprietäre Software) vorgegeben wird, ist hier bereits die Abbildung des Modells in die relationale Datenbank vorhanden. Um diese auch im Eclipse Modeling Framework nutzen zu können, muss zur Laufzeit die Abbildung des EMF-Modells auf die Datenbank ausgelesen werden. Diese wurde zuvor von Teneo generiert; das Meta-Modell entspricht dem von Enterprise Architect. Beim Start von Hibernate wird die aktuelle Instanz von Hibernate mit den Abbildungsinformationen initialisiert. Diese stimmen mit dem Input des Trigger-Generators überein. Somit ist es möglich, für jede Tabelle vier Attribute abzurufen: Den Namen der Entität und deren Identifier im EMF-Modell und die dazugehörige Tabelle und Primärschlüssel in der Datenbank.

Mit diesen Informationen kann auf die Protokolltabellen in der Datenbank zugegriffen werden. Hibernate unterstützt die inkrementelle Aktualisierung von Modellelementen. Die Information, welche Elemente bzw. Modellteile aktualisiert werden, ergibt sich aus den Protokolltabellen. Insbesondere sind die Protokolltabellen kein Teil des Modells; alle Anfragen an diese werden also nicht von Hibernate verwaltet. Werden diese in regelmäßigen Abständen wiederholt, ergibt sich hieraus eine Änderungsbenachrichtigung.

Die Umsetzung der Änderungsbenachrichtigung in Enterprise Architect funktioniert analog. Mit Hilfe der statischen Abbildung werden Anfragen an die zugrundeliegende Datenbank gestellt und kürzliche Änderungen abgefragt. Im Gegensatz zu Hibernate ist hier keine zusätzliche Synchronisation notwendig. Die Programmierschnittstelle von Enterprise Architect bietet bereits von Haus aus Funktionalität an, um die Oberfläche nach einer externen Änderung zu aktualisieren.

6.3 Synchronisierung

Die Änderungserkennung und –Benachrichtigung sind im konkreten Szenario nicht ausreichend, um Modell-Elemente im Eclipse Modeling Framework inkrementell zu aktualisieren. Während in Enterprise Architect bereits Funktionalität bereitgestellt wird, fehlt in Hibernate der Schritt von der Benachrichtigung zur Aktualisierung.

Das Nachladen und Ersetzen von Elementen stellt hierbei keine Alternative dar. Werden neu geladene Modell-Teile nicht mit dem bereits im Speicher vorhandenen Modell korrekt zusammengeführt, können Referenzen ungültig werden. Diese Problematik wurde bereits in Kapitel 5.3 besprochen. Eine Lösung ist der Einsatz von Techniken zum Modellvergleich wie EMF Compare. Glücklicherweise bietet aber auch Hibernate eine Funktion zum inkrementellen Aktualisieren von Modell-Elementen an. Beim Aufruf dieser werden das Element selbst und alle Elemente, die dieses referenzieren, rekursiv bis zum Wurzelelement des Modells aktualisiert und damit auf den Stand der in der Datenbank gespeicherten Modell-Repräsentation gebracht. Andere Elemente sind von dieser Aktualisierung nicht betroffen; der Vorgang entspricht keinem vollständigen Neuladen.

Während der lokale Zustand des Modells nach einer fremden Änderung inkrementell aktualisiert wird, ist für die Gegenrichtung der Einsatz von Beobachtern notwendig. Diese wurden bereits in Kapitel 5.4 vorgestellt. Bei der Initialisierung wird ein spezieller EMF Adapter in das Root Element eingefügt, welcher die Eigenschaft hat, sich rekursiv bei allen Kindelementen zu registrieren. Somit ist es ohne weiteres möglich, auf lokale Änderungen zu reagieren. Als Reaktion auf eine erkannte Änderung wird das betroffene Element mit Hilfe von Hibernate in der Datenbank aktualisiert. Durch die Kombination mehrerer Änderungen könnte die Anzahl der Datenbankanfragen reduziert und somit die Performanz verbessert werden. Ein zusätzlicher Puffer zum Sammeln mehrerer Änderungen wäre denkbar, wurde jedoch nicht umgesetzt.

Ist sowohl die Änderungsbenachrichtigung und die inkrementelle Aktualisierung als auch die lokale Änderungserkennung durch Adapter aktiviert, ergibt sich hieraus zur Laufzeit das Problem der Rückkopplung. Wird eine Änderung in der Datenbank erkannt und in die Protokolltabelle geschrieben, wird das Element in der nächsten Iteration der Persistenzschicht aktualisiert. Diese Aktualisierung hat einen Aufruf der Adapter zur Folge, da diese nicht zwischen externer und lokaler Änderung unterscheiden können. Nachdem die vermeintliche Änderung in die Datenbank geschrieben wurde, wiederholt sich der Zyklus (siehe Abbildung 6.2).

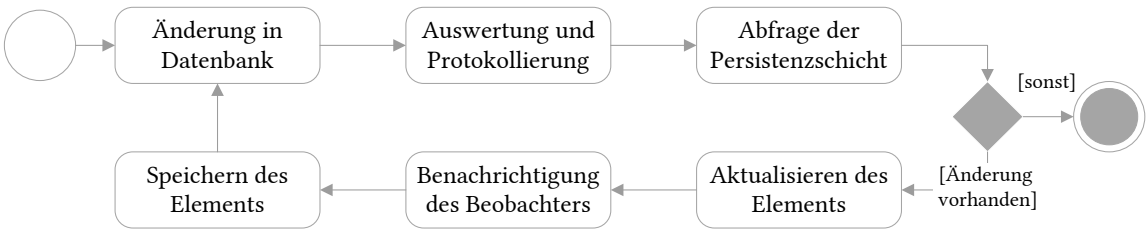


Abbildung 6.2: Endlosschleife während der Synchronisierung

Um Rückkopplungen zu verhindern, wurde die Funktionalität der Trigger und des Aktualisierungsalgorithmus erweitert: Sowohl Trigger als auch Adapter prüfen zunächst, ob sich bei der Aktualisierung Werte geändert haben. Ist dies nicht der Fall, sind keine wei-

teren Schritte notwendig. Zusätzlich werden Adapter zur Laufzeit während des Aktualisierens von Elementen deaktiviert. Da der Adapter-Aufruf synchron und direkt nach einer Änderung geschieht, ist dies ohne weiteres möglich. Durch diese Maßnahmen wird die Rückkopplung im Adapter an zwei Stellen unterbrochen. Diese Vorgehensweise wird in Abbildung 6.3 dargestellt.

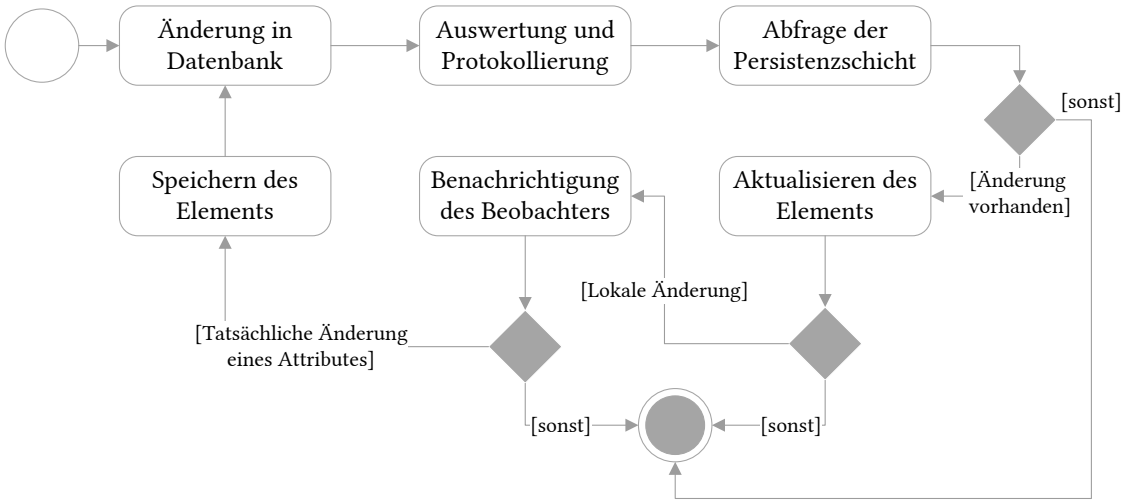


Abbildung 6.3: Verhinderung der Endlosschleife während der Synchronisierung

6.4 Alternative Änderungserkennung

Im bisherigen Kapitel wurden Datenbank-Trigger zur Änderungserkennung eingesetzt. Um diese verwenden zu können, muss jedoch der Zugriff auf die Datenbank gewährleistet sein. Steht dieser nicht zur Verfügung oder unterstützt die verwendete Datenbanksoftware keine Trigger (bzw. ähnliche Konzepte), lässt sich dieser Ansatz nicht umsetzen. Eine potenzielle Lösung für diese Problematik sind Datenbank-Proxies.

Proxies werden in eine Verbindung zwischen Datenbank und Anwendung eingefügt. Diese erhalten somit die Fähigkeit, sämtliche Anfragen mitzuschreiben, zu verändern oder abzulehnen [19]. Insbesondere lässt sich mit Proxies der Datenverkehr geschlossener Systeme (sowohl auf Datenbank, als auch Anwendungsebene) überwachen.

Im Rahmen der Voruntersuchung wurden Datenbank-Trigger und Beobachter der Nutzung von Proxies vorgezogen. Während Proxies zwar universal einsetzbar sind, profitieren sie nicht von der vereinfachten Umsetzung aus einem System heraus. So können in Triggern als auch in Beobachtern Änderungen effizient ohne aufwendige Implementierungsarbeiten untersucht werden. Proxies erhalten nur die rohen Anfragen; jegliche Verarbeitung muss selbst implementiert werden. Weitere Nachteile wurden bereits in Kapitel 5.6 diskutiert.

Werden Proxies zur Dokumentation von Änderungen eingesetzt, müssen zunächst relevante Anfragen herausgefiltert werden. Im Kontext von SQL-Queries entspricht dies beispielsweise *INSERT*- oder *UPDATE*-Anfragen. Anschließend müssen diese geparsed und aufbereitet werden. Die Nutzung eigener Tabellen, einer eigenen Datenbank oder einer anderen Möglichkeit der Speicherung ist naheliegend. Im Idealfall ergibt sich eine einfache Schnittstelle für die Persistenzschicht, analog zur Umsetzung mit Triggern in Kapitel 6.1.

Im Rahmen der Umsetzung wurden MySQL Proxies genauer untersucht. Die Einrichtung ist denkbar einfach: Der Proxy wird lokal gestartet, notwendige Verbindungsinformationen zur Datenbank werden beim Start angegeben. In der Anwendungsschicht wird die Adresse der Datenbank durch die Adresse des Proxys ersetzt. Ab diesem Punkt werden alle Anfragen durch den Proxy an die Datenbank weitergeleitet. Eine zusätzliche Verarbeitung findet zu diesem Zeitpunkt noch nicht statt. Hierfür muss der Proxy mit einem Lua-Skript gestartet werden. Ein simples Protokollierungsskript ist in Quelltext 6.2 zu sehen.

```
function read_query(packet)
  if string.byte(packet) == proxy.COM_QUERY then
    print(string.sub(packet, 2))
  end
end
```

Quelltext 6.2: Lua-Skript zur Protokollierung von Anfragen mit MySQL-Proxy

Für die weitere Auswertung kann das verwendete Skript erweitert werden. Alternativ ist ebenfalls die Auswertung der Prozessausgabe denkbar. Wird diese analysiert und anschließend lokal oder in einer Datenbank gespeichert, entspricht die Funktionalität des Proxies der eines Triggers oder Beobachters. Dieser Prozess dürfte der Performanz einer internen Lösung nicht gleichkommen, da nicht notwendigerweise alle Informationen ohne zusätzliche Verarbeitungsschritte (hierzu zählt auch das Parsen der Eingabe) zur Verfügung stehen. Aus diesem Grund wurde für die Umsetzung auf die Nutzung von Proxies verzichtet.

6.5 Alternative Synchronisierungstechniken

Wird keine Persistenzschicht verwendet oder besitzt diese keine Funktionalität zur Aktualisierung einzelner Elemente, kann der in diesem Kapitel vorgestellte Ansatz ebenfalls nicht umgesetzt werden. Eine alternative Synchronisierungstechnik stellt Aktives Warten in Verbindung mit Modellvergleichen dar.

Aktives Warten wurde bereits im Rahmen der prototypischen Umsetzung eingesetzt, um von Triggern geführte Protokolltabellen in regelmäßigen Abständen auf neue Einträge zu überprüfen. Darüber hinaus kann Aktives Warten auch zum Nachladen von Modellelementen und als Grundlage zur Änderungserkennung genutzt werden. Bietet eine Persistenzschicht bereits Funktionalität zur inkrementellen Aktualisierung an (wie im Fall von Hibernate), kann dies eine bessere Performanz mit sich bringen.

Steht diese Funktionalität nicht zur Verfügung oder wird auf den Einsatz einer Persistenzschicht verzichtet, kann diese mit Hilfe von Aktivem Warten nachgebildet werden. Wichtig hierbei ist, dass Aktives Warten im Kontext des Anwendungsfalls nur das repetitive Laden von Informationen beschreibt; die inkrementelle Aktualisierung nach Änderungserkennung ist hiermit nicht abgedeckt.

Aktives Warten kann in verschiedenen Granularitätsstufen erfolgen. Am Beispiel eines Klassendiagramms wäre das Nachladen einzelner Elemente oder Pakete möglich, um diese anschließend mit dem bereits geladenen Modell zu vergleichen und ggf. Änderungen zu übernehmen. Um das Nachladen auf Modell-Teile zu beschränken, muss sichergestellt werden, dass diese im gegebenen Kontext relevant sind. Dies kann beispielsweise

mit Hilfe einer der bereits vorgestellten Techniken zur Änderungserkennung umgesetzt werden. Alternativ kann das Nachladen auch auf den aktuell vom Nutzer betrachteten Teil des Modells beschränkt werden. Nach diesem Konzept funktioniert die Aktualisierung der Oberfläche in Enterprise Architect. Kann das geöffnete Modell nicht auf relevante Teile reduziert werden, kann Aktives Warten auch eingesetzt werden, um das Modell vollständig nachzuladen, was eine tendenziell schlechtere Performanz mit sich bringt.

Ist keine Technik zur Änderungserkennung vorhanden, unterstützt die Persistenzschicht wie im Falle von Hibernate aber die Aktualisierung einzelner Modell-Elemente, kann hier auf Aktives Warten verzichtet werden. Durch die von der Persistenzschicht zur Verfügung gestellte Traversierung können alle Elemente eines Modells rekursiv aktualisiert werden. Diese Vorgehensweise verhindert nicht den Aufwand für das stückweise vollständige Nachladen des Modells. Allerdings kann auf den Vergleich und das Zusammenführen des lokalen Modells mit dem frisch aus der Datenbank geladenen Modell verzichtet werden.

7 Evaluation

In diesem Kapitel soll das bereits vorgestellte Konzept anhand der Untersuchung der Umsetzung evaluiert werden. Hierbei wird insbesondere die Performanz der inkrementellen Aktualisierung dem vollständigen Nachladen gegenübergestellt. Ist der Zusatzaufwand der Änderungserkennung klein genug, stellt die inkrementelle Vorgehensweise einen Fortschritt gegenüber der vollständigen Ersetzung dar.

Das Ziel dieser Arbeit ist die Untersuchung von Konzepten und Techniken zur inkrementellen Aktualisierung. Während in den vorherigen Kapiteln bereits auf Unterschiede und Einsatzgebiete einzelner Techniken eingegangen wurde, soll nun auch ein Vergleich auf technischer Ebene mit Hilfe von Metriken erfolgen. Hierbei wird keine allgemeine Lösung gefunden; die Untersuchung bezieht sich nur auf den Anwendungsfall, der bereits in Kapitel 1 vorgestellt wurde.

Nachfolgend werden zunächst Qualitätskriterien spezifiziert, anhand derer die Implementierung getestet werden soll. Anschließend werden diese anhand mehrerer Untersuchungen gemessen. Das Ergebnis der Untersuchung wird abschließend diskutiert.

7.1 Qualitätskriterien

Die nachfolgenden Qualitätskriterien wurden im Rahmen der Literaturrecherche erarbeitet und repräsentieren die Anforderungen an die Umsetzung.

- **Granularität**. Bereits aus dem Titel dieser Arbeit lässt sich diese Metrik ableiten. Wird inkrementell aktualisiert, stellt sich immer auch die Frage nach dem Grad der Inkrementalität. Das vollständige Nachladen eines Modells stellt hierbei den schlechtesten Wert, das Laden von Modell-Teilen eine Verbesserung und das Laden von einzelnen Modell-Elementen den besten Wert dar. Die Aktualisierung einzelner Attribute eines Elementes wurde nicht berücksichtigt, da die untersuchten Persistenzschichten diesen Detailgrad nicht unterstützen.
- **Performanz**. Die Laufzeit eines Aktualisierungsschrittes liegt im Mittelpunkt dieser Evaluation. Von dieser ist abhängig, ob eine Lösung im Kontext des Anwendungsfalls sinnvoll einsetzbar ist. Die Speicherauslastung während der Aktualisierung wird nicht weiter betrachtet, da Aktualisierungsinformationen in der Regel neben dem tatsächlichen Modell nicht ins Gewicht fallen sollten.
- **Skalierbarkeit**. Während in kleineren Modellen eine vollständige Ersetzung noch performant umsetzbar ist, wird dies mit steigender Modellgröße schwieriger. Die Laufzeit einer Lösung sollte unabhängig von der Modellgröße und bestenfalls konstant sein. Ebenfalls sollte sie nicht nennenswert nachlassen, wenn die Anzahl der Teilnehmer steigt.

Die Messung dieser Qualitätskriterien ist in drei Bereiche aufgeteilt. Hierbei werden alle Komponenten einer Performanz-Analyse umgesetzt: Messung, Simulation und analytische Modellierung [25]. Getestet werden hierbei die prototypische Umsetzung sowie nicht-iterative bzw. grobgranularere Ansätze. Im Rahmen der Messung wurde die Laufzeit einzelner Aktualisierungsschritte in realer Umgebung untersucht. Hierbei wurde ebenfalls die Skalierbarkeit mit Hilfe von Modellen verschiedener Größe analysiert. Anschließend wurde die Performanz gleichzeitiger Datenbankanfragen simuliert, um die Auswirkung des Datenbank-Caches auf die Performanz der Umsetzung zu zeigen. Zuletzt wird theoretisch analysiert wie viele Anfragen der Datenbankserver unter den verschiedenen Konfigurationen und Ansätzen beantworten muss.

7.2 Laufzeitmessung

Im Rahmen der Laufzeitmessung wird die Laufzeit eines Aktualisierungsschrittes in realer Umgebung gemessen. Hierzu wurde die prototypische Umsetzung im EMF-Umfeld entsprechend präpariert. Sowohl in Hibernate als auch in der Datenbank sind Caches zur Beschleunigung der Performanz aktiviert. Um ein aussagekräftiges Ergebnis zu erhalten, wird jede Messung 100-mal wiederholt. Der Zusatzaufwand für die Änderungserkennung bzw. Benachrichtigung wird separat gemessen. Bereits in der Messvorbereitung hat sich ergeben, dass die Anzahl der tatsächlichen Änderungen in einer Iteration keine nennenswerten Auswirkungen auf das Ergebnis hat, solange sich diese in einem realistischen (und im Kontext des Anwendungsfalls annehmbaren) Bereich wie 5-10 Änderungen pro Wiederholung bewegt.

Die Laufzeitmessung wurde in realer Umgebung ausgeführt, wie sie auch in einem typischen Szenario zum Einsatz käme. Getestet wurde auf einem Computer mit Intel® Xeon® E3-1240-Prozessor mit 3,4 GHz und 8 GB Arbeitsspeicher. Dieser ist über eine 1 GHz-Leitung an die lokal gehostete Datenbank angebunden. Der Server besitzt einen Intel® Xeon® E5320-Prozessor mit 1.86 GHz und 4 GB Arbeitsspeicher. Verwendet wurde die Datenbanksoftware MySQL 5.7.

Getestet wurden mehrere UML-Modelle aus realen Enterprise Architect-Projekten, welche einer statistischen Untersuchung von öffentlich verfügbaren UML-Modellen entnommen wurden [26]. Die Angabe der Modellgröße stammt aus dieser Arbeit, die Anzahl von Klassen, Paketen und Diagrammen wurde selbst ermittelt. In Tabelle 7.1 sind die wichtigsten Kennzahlen der Modelle aufgeführt.

Eigenschaft /Modell	**Post-Evolution**	**DataModel**	**OmeroDb**
Modellgröße [26]	396	2349	3903
Objekte	165	319	244
Attribute	301	617	1315
Methoden	172	225	156
Verbindungen	107	2603	714
Pakete	22	4	2

Tabelle 7.1: Kennzahlen der UML-Modelle

Anhand dieser Modelle wurde die Laufzeit der prototypischen Umsetzung und Alternativen dieser gemessen. Hierbei wurde auf den drei Granularitätsstufen Modell, Paket und Element jeweils eine beispielhafte Aktualisierung im Sinne der Umsetzung getestet. Diesem Ergebnis werden ein Lösungsansatz mit Aktivem Warten und anschließendem Modellvergleich, sowie ein Ansatz mit rekursiver Aktualisierung gegenübergestellt. Werden Trigger zur Änderungsdetektion eingesetzt, müssen die entsprechenden Protokolltabellen abgefragt werden, wenn nicht das vollständige Modell nachgeladen werden soll. Als Laufzeit hat sich hierbei für jeweils 11 zu untersuchende Tabellen des Enterprise Architect Meta-Modells ca. 200 ms ergeben. Diese werden auf die Ergebnisse von Paket- und Element-Aktualisierung in Tabelle 7.2 (Modellvergleich) und Tabelle 7.3 (rekursive Aktualisierung) addiert.

Einzelne Schritte der Aktualisierung werden hierbei getrennt betrachtet. Die Summe aus dem Laden des Modells, dem Auflösen aller Modellabhängigkeiten (inkl. potenziellem Nachladen), dem Modellvergleich und dem Zusatzaufwand durch die Änderungsdetektion ergibt die vollständige Laufzeit. Die Aktualisierung auf Elementebene (siehe Tabelle 7.3) entspricht der prototypischen Umsetzung aus Kapitel 6.

	Post-Evolution			**DataModel**			**OmeroDb**		
	Modell	Paket	Element	Modell	Paket	Element	Modell	Paket	Element
Laden	0,014	0,026	0,023	0,013	0,020	0,018	0,013	0,021	0,016
Auflösen	4,907	1,098	0,757	9,663	3,450	0,009	10,407	10,248	0,05
Vergleichen	0,051	0,037	0,027	0,137	1,654	0,041	0,201	0,193	0,042
Zusatzaufwand	0,000	0,200	0,200	0,000	0,200	0,200	0,000	0,200	0,200
Gesamt	**4,972**	**1,361**	**1,007**	**9,813**	**5,324**	**0,268**	**10,621**	**10,662**	**0,308**

Tabelle 7.2: Ergebnisse der Laufzeitmessung (Modellvergleich) in Sekunden

	Post-Evolution			**DataModel**			**OmeroDb**		
	Modell	Paket	Element	Modell	Paket	Element	Modell	Paket	Element
Aktualisieren	3,492	0,752	0,014	4,757	1,180	0,015	2,782	2,777	0,007
Zusatzaufwand	0,000	0,200	0,200	0,000	0,200	0,200	0,000	0,200	0,200
Gesamt	**3,492**	**0,952**	**0,214**	**4,757**	**1,38**	**0,215**	**2,782**	**2,977**	**0,207**

Tabelle 7.3: Ergebnisse der Laufzeitmessung (Rekursive Aktualisierung) in Sekunden

Sowohl bei der Nutzung von Modellvergleich als auch rekursiver Aktualisierung ergibt sich eine Beschleunigung abhängig von der gewählten Granularität. Während die Aktu-

alisierung auf Elementebene immer einen Performanzgewinn darstellt, ist gerade im Modell *OmeroDb* nur ein geringer Unterschied zwischen Modell- und Paketgranularität zu sehen. Dieser lässt sich mit der Anzahl Pakete des Modells (siehe Tabelle 7.1) und der Tatsache, dass jedes Paket viele Elemente enthält, erklären. Ebenfalls auffällig ist, dass die gemessenen Zeiten der Modelle *OmeroDb* und *DataModel* nicht im selben Verhältnis stehen, wie die Modellgröße. Neben dieser ist hierfür die Anzahl der Objekte und Verbindungen ausschlaggebend.

7.3 Query-Simulation

Zur Simulation kommt das MySQL-Diagnose-Werkzeug **mysqlslap** [27] zum Einsatz. Mit dessen Hilfe ist es möglich, die Antwortzeit eines Datenbankservers unter der Last mehrerer simultaner Verbindungen zu testen. Als Eingabe erhält die Software eine Datenbankverbindung und mehrere SQL-Anfragen. Diese wurden durch die Protokollierung der Anfragen des Prototypen ermittelt. Ebenfalls kann eine Anzahl Iterationen angegeben werden und wie viele gleichzeitige Benutzer simuliert werden sollen. Das Ergebnis ist die beste, schlechteste und durchschnittliche Antwortzeit des Servers.

Während der Messung hat sich ergeben, dass die Ergebnisse zwar untereinander vergleichbar sind, aber nicht mit denen der Laufzeit-Messung übereinstimmen, obwohl dieselben Modelle verwendet wurden. Die simulierten Laufzeiten sind ca. doppelt so groß wie reale Ergebnisse. Trotzdem lassen sich hier die Effekte des Datenbank-Caches beobachten. Getestet wurden 1, 10 und 20 parallele Nutzer, jeweils mit 50 Iterationen. Als Modell wurde Post-Evolution (siehe Kapitel 7.2) als UML-Modell durchschnittlicher Größe gewählt.

Die Ergebnisse der Messung werden nachfolgend dargestellt. Tabelle 7.4 zeigt die Laufzeiten der Anfragen des Prototypen. Hierbei wird zwischen der regelmäßig stattfindenden Abfrage der Protokolltabellen und dem Aktualisieren eines Elements nach einer Änderung unterschieden. Tabelle 7.5 zeigt die Laufzeiten alternativer, nicht-inkrementeller Ansätze. Hierfür wurde sowohl die rekursive Aktualisierung aller Modellelemente als auch das vollständige Nachladen des Modells für den Modellvergleich simuliert.

	Nutzer	**Beste Laufzeit**	**Durchschn. Laufzeit**	**Schlechteste Laufzeit**
Abfrage der Protokoll-tabellen (10 Queries)	1	0,000	0,029	1,015
	10	0,000	0,039	1,015
	20	0,015	0,077	1,218
Aktualisieren eines Elements (14 Queries)	1	0,000	0,071	1,203
	10	0,000	0,074	1,218
	20	0,031	0,194	1,624

Tabelle 7.4: Ergebnisse der Simulation (Inkrementelle Aktualisierung) in Sekunden

	Nutzer	Beste Laufzeit	Durchschn. Laufzeit	Schlechteste Laufzeit
Rekursives Aktualisieren eines Modells (2536 Queries)	1	6,526	8,477	10,257
	10	12,193	13,820	15,598
	20	25,870	26,524	27,759
Vollständiges Nachladen eines Modells (3135 Queries)	1	8,072	9,391	11,928
	10	13,942	14,803	18,282
	20	26,416	27,841	29,989

Tabelle 7.5: Ergebnisse der Simulation (Alternative Ansätze) in Sekunden

Das Ergebnis der Query-Simulation zeigt unter anderem auf, wie groß die Beschleunigung durch Nutzung des Datenbank-Caches ist. Im Kontext des Anwendungsfalls greifen mehrere Nutzer parallel lesend auf dieselbe Datenbank zu, was im Falle weniger Änderungen eine wesentliche Beschleunigung durch Caches bedeutet. Ein typisches Nutzungsszenario stellt die Kombination von Abfragen der Protokolltabellen und Aktualisieren einzelner Elemente dar. Für beide Fälle liefert die Simulation für eine Teamgröße von 10 Teilnehmern eine Laufzeit von unter 100 ms. Das vollständige Aktualisieren oder Nachladen hingegen wurde mit einer Laufzeit von ca. 15 Sekunden simuliert.

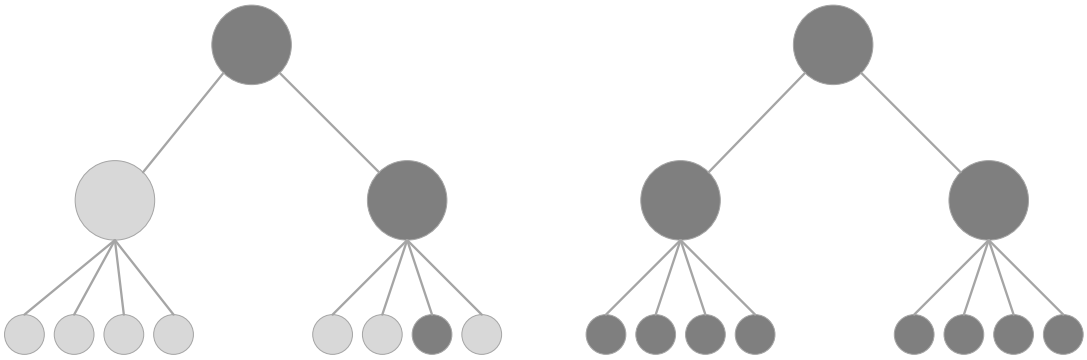
7.4 Anzahl der Anfragen

Neben konkreter Messung und Simulation, lässt sich der Unterschied verschiedener Ansätze ebenfalls auf theoretischer Basis ermitteln. Hierzu wird die Anzahl vergleichbarer Anfragen für verschiedene Konzepte ermittelt.

Unter der Annahme, dass die zugrundeliegende Datenbanksoftware eine performante Regelauswertung implementiert, kann der Aufwand einer Trigger-Auswertung wie eine zusätzliche Anfrage angesehen werden. Somit ergibt sich sowohl für Insert-, Update-, als auch Delete-Operationen auf überwachte Tabellen der doppelte Aufwand. Statt einer Anfrage muss zusätzlich die Trigger-Anfrage (z.B. der Eintrag in eine Protokolltabelle) ausgeführt werden. Dieser Aufwand ist konstant und nicht von der Größe des Modells abhängig. Ebenfalls kann der Aufwand für die Erstellung der drei Trigger und mindestens einer Protokolltabelle (inkl. Event) ignoriert werden, da dieser nur einmalig bei der Einrichtung der Datenbank bzw. bei der Umstellung des Meta-Modells notwendig ist.

Werden Proxies statt Datenbank-Trigger eingesetzt, verteilt sich dieser Aufwand. Während die Datenbank nur die Last einer einzelnen Anfrage tragen muss, liegt die Verarbeitung beim Proxy. Eine genauere Messung der Performanz wurde im Rahmen dieser Arbeit nicht vorgenommen, es ist jedoch anzunehmen, dass die Performanz schlechter als beim Einsatz von Triggern ist, da sowohl die Datenbanksoftware als auch der Proxy die Anfrage unabhängig voneinander analysieren muss. Durch die Kapselung können Informationen nicht weiterverwendet werden, wie dies im Falle von Triggern möglich wäre.

Zur Laufzeit werden in regelmäßigen Abständen alle Protokolltabellen auf neue Einträge untersucht. Dieses Vorgehen ergibt einen regelmäßigen Zusatzaufwand, welcher proportional zur Anzahl der zu verwaltenden Tabellen ist. Im Fall des Enterprise Architect Meta-Modells sind dies beispielweise 10 Tabellen, also 10 zusätzliche Anfragen in jeder Iteration. Wurde eine Änderung entdeckt, wird der betroffene Modell-Teil inkrementell aktualisiert. Für die Anzahl der notwendigen Anfragen ist hierbei sowohl die Größe des Modell-Teils als auch die verwendete Software der Persistenzschicht entscheidend. Im Fall von Hibernate wird das Modellelement selbst und alle Pakete bis zum Root-Paket aktualisiert. Dies ist im Allgemeinfall nicht mit einer vollständigen Ersetzung zu vergleichen. Wird beispielweise ein Element in einem Paket aktualisiert, welches direkt unter dem Root Paket liegt, ergibt dies eine Anfrage für das Element und zwei zusätzliche Anfragen für beide Pakete. Im Fall von Aktivem Warten (z.B. im Kontext des Modellvergleichs) müssen ebenfalls das Element und beide Pakete geladen werden; und alle anderen Elemente in diesen Paketen ebenfalls. Dieses Beispiel wird in Abbildung 7.1 verdeutlicht. Hervorgehoben sind hierbei alle Elemente, welche geladen werden.



Inkrementelle Aktualisierung Vollständiges Nachladen

Abbildung 7.1: Nachgeladene Elemente verschiedener Aktualisierungstechniken

Vergleicht man unter diesen Voraussetzungen die inkrementelle Aktualisierung direkt mit dem vollständigen Neuladen und zählt die Anzahl der Anfragen, ergibt sich folgendes Ergebnis: Durch die regelmäßigen Änderungsabfragen in den Protokolltabellen ist es durchaus möglich, dass im inkrementellen Ansatz mehr Anfragen gestellt werden, als für das einmalige Nachladen des gesamten Modells notwendig wären. Diese finden jedoch über die Zeit verteilt statt und haben keinen nennenswerten Einfluss auf die Performanz des Systems. Im Sinne der Performanz sind also über einen Zeitraum verteilte, schnell zu verarbeitende Anfragen vorzuziehen, wenn die Alternative viele Anfragen in kurzer Zeit sind, wie es beim Nachladen eines vollständigen Modells der Fall ist.

Der Zusatzaufwand durch Trigger in Verbindung mit der Abfrage der Protokolltabellen ist in Formel 7.1 dargestellt. Formel 7.2 zeigt die Anzahl der Anfragen für die vollständige Aktualisierung. Setzt man hier bereits vorgestellte Modellzahlen ein (beispielsweise 1 Iteration, 3 Änderungen, 10 Tabellen, 20 Elemente, 7 Anfragen pro Elementaktualisierung) ergeben sich hierbei 37 Anfragen bei inkrementeller Aktualisierung und 140 Anfragen bei vollständigem Nachladen. Hinzu kommt die Tatsache, dass die Gestalt der Aktualisierungs-Anfragen komplexer ist, da hierfür in der Regel viele Attribute und Fremdschlüsselbeziehungen ausgewertet werden müssen.

$$Anzahl_{Anfragen} = \left(Anzahl_{Änderungen} * 2\right) + Anzahl_{Tabellen} + Aktualisierung$$

Formel 7.1: Anzahl der Anfragen für Änderungserkennung und inkrementelle Aktualisierung

$$Anzahl_{Anfragen} = Aktualisierung * Anzahl_{Elemente}$$

Formel 7.2: Anzahl der Anfragen für das vollständige Nachladen eines Modells

7.5 Diskussion

Das Ziel dieser Arbeit ist die Untersuchung von Konzepten und Techniken zur inkrementellen Aktualisierung. Im Rahmen der Untersuchung haben sich verschiedene Konzepte ergeben, die im Rahmen der Evaluation verglichen und einer vollständigen Ersetzung gegenübergestellt wurden. Untersucht wurden die Qualitätskriterien Performanz, Granularität und Skalierbarkeit.

Die Ergebnisse der Laufzeitmessung, Simulation und theoretischen Untersuchung stimmen weitestgehend überein: Die inkrementelle Aktualisierung weist in allen Messungen eine bessere Laufzeit als die vollständige Ersetzung auf. Abhängig von der Modellgröße lässt sich eine unterschiedliche Beschleunigung feststellen: Während die Performanz der vollständigen Modellaktualisierung mit der Steigerung der Modellgröße abnimmt, bleibt die inkrementelle Aktualisierung auf Elementebene nahezu konstant. Dies zeigen sowohl die Ergebnisse der realen Laufzeitmessung als auch die der Query-Simulation. Unterstützt wird dies auch durch die theoretische Anzahl der Anfragen.

Die größte Beschleunigung der Laufzeit wurde beim Modell *OmeroDb* gemessen. Stellt man hierbei das vollständige Nachladen beim Modellvergleich (10,621 Sekunden) der inkrementellen Aktualisierung (0,207 Sekunden) gegenüber, ergibt sich eine 51-fache Beschleunigung. Hinzu kommt, dass die Aktualisierung auf Elementebene von der Modellgröße unabhängig ist, also wesentlich besser skaliert als die Aktualisierung mit Modell- oder Paketgranularität.

Durch den Zusatzaufwand der Änderungserkennung bzw. Benachrichtigung ist es unabhängig von der tatsächlichen Synchronisierung möglich, dass insgesamt mehr Anfragen gestellt werden, als beim einmaligen Nachladen des Modells. Dies ist dann der Fall, wenn die Aktualisierung über einen längeren Zeitraum auf Änderungen achtet, ohne dass diese auftreten. Die Query-Simulation hat gezeigt, dass die Anfragen zur Änderungserkennung (0,077 Sekunden bei 20 Nutzern) jedoch in keinem Verhältnis zur vollständigen nicht-inkrementellen Aktualisierung (27,841 Sekunden bei 20 Nutzern) stehen. Im konkreten Anwendungsfall ist eine Wartezeit von mehreren Sekunden pro Änderung nicht akzeptabel, da somit effektives Arbeiten verhindert wird. Inkrementelle Aktualisierung stellt eine potenzielle Lösung dieses Problems dar.

8 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurden Konzepte und Techniken für inkrementelle Aktualisierung untersucht. Insbesondere wurde inkrementelles Vorgehen herkömmlichen Lösungen wie der vollständigen Ersetzung gegenübergestellt. Das Ziel war die Erarbeitung einer Lösung, die bei hoher Benutzerfreundlichkeit eine bessere Performanz und geringere Auslastung des zugrundeliegenden Systems mit sich bringt.

Im konkreten Anwendungsfall arbeiten Nutzer verschiedener Software-Lösungen für Modellierung gemeinsam an denselben Modellen. Diese sind in einer gemeinsam genutzten Datenbank persistiert. Um die performante Teamarbeit zu ermöglichen, muss jeder Nutzer stets den aktuellen Zustand des Modells vor sich haben.

Hierfür wurden Konzepte und Techniken aus verwandten Arbeiten erarbeitet und anschließend anhand prototypischer Umsetzung verglichen. Als bester Ansatz für das konkrete Szenario haben sich im Laufe der Umsetzung Datenbank-Trigger in Verbindung mit einer Persistenzschicht wie Hibernate herausgestellt. Sind Einschränkungen auf Datenbank- oder Anwendungsebene gegeben, sind auch alternative Konzepte wie Proxies oder Aktives Warten und Modellvergleich denkbar.

Während der Evaluation wurden diese Konzepte im Hinblick auf die Qualitätskriterien Performanz, Skalierbarkeit und Granularität getestet. Hierzu wurden Messungen im realen Umfeld und mit Hilfe einer Query-Simulation durchgeführt. Ebenfalls wurden die zuvor untersuchen Konzepte theoretisch analysiert. Die Ergebnisse zeigen deutlich, wie wichtig die Wahl der richtigen Granularitätsstufe ist: Während die gemessenen Laufzeiten bei vollständiger Modellaktualisierung proportional mit der Modellgröße wachsen, bleibt die Laufzeit der inkrementellen Aktualisierung nahezu konstant. Bei der Vermessung eines UML-Klassendiagramms mit ca. 4000 Elementen ergibt sich hierbei eine 51-fache Beschleunigung. Ebenfalls bei Modellen durchschnittlicher Größe liegt die inkrementelle Aktualisierung vorne.

Diese Verbesserung ermöglicht es, im Team verteilt an Modellen zu arbeiten. Die gemessene Laufzeit liegt im Bereich von ca. 200 ms und ist somit klein genug, um Echtzeitanforderungen im Kontext der Modellierung zu genügen. Die herkömmliche Lösung der vollständigen Ersetzung benötigt im Test mehrere Sekunden, auch wenn sich die Änderung beispielsweise auf den Namen eines Attributes beschränkt.

Im Rahmen zukünftiger Arbeit könnte der Einsatz inkrementeller Methoden weiter untersucht werden. Durch die zeitlichen Beschränkungen konnten nicht alle Konzepte und Techniken untersucht werden, die im Laufe der Literaturrecherche gefunden wurden.

Im Laufe der Arbeit wurden zudem alternative Konzepte zur prototypischen Umsetzung untersucht, die zum Einsatz kommen können, wenn konkrete Einschränkungen in der

Umgebung gegeben sind. Hierzu gehören Datenbank-Proxies und der Einsatz von Aktivem Warten und Modellvergleich. Obwohl diese Techniken zwar eine Variation der prototypischen Umsetzung darstellen, wurden diese nicht implementiert. Um den Prototypen in einem größeren Umfeld einsetzbar zu machen, wäre auch die Umsetzung dieser Konzepte denkbar.

Das Ergebnis dieser Arbeit bestätigt die Vermutung, dass inkrementelle Aktualisierung einen Vorteil gegenüber vollständiger Modellersetzung für interaktive Modellierungsszenarien darstellt. Dies wurde sowohl auf konzeptueller Ebene untersucht als auch durch konkrete Evaluation gezeigt.

Literatur

[1] T. Stahl, M. Völter, S. Efftinge und A. Haase, Modellgetriebene Softwareentwicklung, Heidelberg: dpunkt.verlag GmbH, 2007.

[2] D. Steinberg, F. Budinsky, M. Paternostro und E. Merks, Eclipse Modeling Framework, Second Edition, Boston, MA: Pearson Education Inc., 2009.

[3] C. Bauer und G. King, Hibernate in Action, Greenwich: Manning Publications Co., 2005.

[4] N. W. Paton und O. Diaz, „Active database systems," *ACM Computing Surveys,* pp. 63-103, März 1999.

[5] Oracle Corporation, „eclipse.org," The Eclipse Foundation, 16 September 2015. [Online]. Available: https://wiki.eclipse.org/CDO. [Zugriff am 07 Juni 2016].

[6] Oracle Corporation, „MySQL Reference Manual - Using Triggers," Oracle Corporation, 2016. [Online]. Verfügbar: http://dev.mysql.com/doc/refman/5.7/en/triggers.html. [Zugriff am 07 Juni 2016].

[7] S. W. Ambler, „The Design of a Robust Persistence Layer for relational Databases," Ambysoft, 2005.

[8] O. Schumann, „Persistenz von EMF-Modellen," *Informatics Inside,* pp. 36-40, 2012.

[9] E. O'Neil, „Object/Relational Mapping 2008: Hibernate and the Entity Data Model (EDM)," University of Massachusetts, Boston, 2008.

[10] A. Benelallam, A. Gomez, G. Sunyé, M. Tisi und D. Launay, „Neo4EMF, A Scalable Persistence Layer for EMF Models," *Modelling Foundations and Applications,* pp. 230-241, 21 Juli 2014.

[11] D. S. Kolovos, D. Di Ruscio, A. Pierantonio und R. F. Paige, „Different models for model matching: An analysis of approaches to support model differencing," *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models,* pp. 1-6, 2009.

[12] The Eclipse Foundation, „EMF Compare - Compare and Merge Your EMF Models," The Eclipse Foundation, 2016. [Online]. Verfügbar: https://www.eclipse.org/emf/compare/index.html. [Zugriff am 12 06 2016].

[13] C. Brun und A. Pierantonio, „Model Differences in the Eclipse Modelling Framework," *UPGRADE,* pp. 29-34, April 2008.

[14] A. Toulmé, „Presentation of EMF Compare Utility," Intalio Inc., Redwood City, CA.

[15] H. Ma, H. Johansson und K. Orsborn, „Distribution and synchronisation of engineering information using active database technology," *Advances in Engineering Software,* pp. 720-728, 2005.

[16] G. von Biiltzingsloewen, A. Koschel und R. Kramer, „Active information delivery in a CORBA-based distributed information system," *Cooperative Information Systems,* pp. 218-227, 19 Juni 1996.

[17] C. Bufller und S. Jablonski, „Implementing agent coordination for workflow management systems using active database systems," *Research Issues in Data Engineering,* pp. 53-59, 14 Februar 1994.

[18] L. Liu und Q. Huang, „A Framework for Database Auditing," *Computer Sciences and Convergence Information Technology,* pp. 982-986, 24 November 2009.

[19] Oracle Corporation, „MySQL Proxy," Oracle Corporation, [Online]. Verfügbar: http://dev.mysql.com/doc/mysql-proxy/en/. [Zugriff am 28 April 2016].

[20] JBoss, „Hibernate ORM documentation," [Online]. Verfügbar: http://hibernate.org/orm/documentation/5.2/. [Zugriff am 18 10 2016].

[21] Object Management Group, „XMI Spezifikation," [Online]. Verfügbar: http://www.omg.org/spec/XMI/. [Zugriff am 15 10 2016].

[22] Oracle Cooperation, „Adding a New User-Defined Function," [Online]. Verfügbar: http://dev.mysql.com/doc/refman/5.7/en/adding-udf.html. [Zugriff am 18 10 2016].

[23] S. Hahner, „Incremental DB Sync Utils," [Online]. Verfügbar: https://github.com/Cooperate-Project/IncrementalDBSyncUtils. [Zugriff am 18 10 2016].

[24] S. Hahner, „Incremental Synchronization," [Online]. Verfügbar: https://github.com/Cooperate-Project/EnterpriseArchitectBridge/tree/IncrementalSync/incrementalsync. [Zugriff am 23 10 2016].

[25] D. J. Lilja, Measuring Computer Performance: A Practitioner's Guide, Cambridge: Cambridge University Press, 2000.

[26] P. Langer, T. Mayerhofer, M. Wimmer und G. Kappel, „On the Usage of UML: Initial Results of Analyzing Open UML Models," *Modellierung 2014,* pp. 289-304, 19 März 2014.

[27] Oracle Corporation, „mysqlslap — Load Emulation Client," [Online]. Verfügbar: http://dev.mysql.com/doc/refman/5.7/en/mysqlslap.html. [Zugriff am 24 10 2016].

Abbildungsverzeichnis

Tabellenverzeichnis

Formelverzeichnis

Quelltextverzeichnis

A. Anhang

A.1 Quelltext-Ausschnitt zur Erstellung von Triggern

Der nachfolgende Quelltext zeigt beispielhafte Anfragen zur Erstellung von Insert-, Update- und Delete-Triggern und einer Protokolltabelle. Zusätzlich wird ebenfalls ein Event erstellt, um veraltete Einträge in der Protokolltabelle automatisiert zu entfernen. Der Quelltext ist im MySQL-Dialekt geschrieben und entstammt dem Trigger-Generator der prototypischen Umsetzung für das Enterprise Architect Meta-Modell [23].

```
DROP TABLE IF EXISTS `ht_t_operationparams`;

DROP TRIGGER IF EXISTS `ht_t_operationparamsInsertTrigger`;

DROP TRIGGER IF EXISTS `ht_t_operationparamsUpdateTrigger`;

DROP TRIGGER IF EXISTS `ht_t_operationparamsDeleteTrigger`;

DROP EVENT IF EXISTS `ht_t_operationparamsCleanEvent`;

CREATE TABLE `ht_t_operationparams`
(
	`ID`		INT,
	`Timestamp` TIMESTAMP(6) NULL DEFAULT NULL,
	PRIMARY KEY (`ID`)
);

DELIMITER $
CREATE TRIGGER `ht_t_operationparamsInsertTrigger` AFTER INSERT ON `t_operationparams`
FOR EACH ROW
	BEGIN
		REPLACE INTO ht_t_operationparams VALUES (NEW.Name, NOW(6));
	END $
DELIMITER ;

DELIMITER $
CREATE TRIGGER `ht_t_operationparamsUpdateTrigger` AFTER UPDATE ON `t_operationparams`
FOR EACH ROW
	BEGIN
		IF NEW.ea_guid <> OLD.ea_guid OR
			NEW.OperationID <> OLD.OperationID OR
			NEW.Type <> OLD.Type OR
			NEW.Name <> OLD.Name
		THEN
			REPLACE INTO ht_t_operationparams VALUES (OLD.Name, NOW(6));
		END IF;
	END $
```

```
DELIMITER ;

DELIMITER $
CREATE TRIGGER `ht_t_operationparamsDeleteTrigger` AFTER DELETE ON `t_operationparams`
FOR EACH ROW
	BEGIN
		REPLACE INTO ht_t_operationparams VALUES (OLD.Name, NOW(6));
	END $
DELIMITER ;

CREATE EVENT `ht_t_operationparamsCleanEvent`
	ON SCHEDULE EVERY 1 MINUTE
	ON COMPLETION PRESERVE
ENABLE
DO
	DELETE FROM `ht_t_operationparams`
	WHERE `Timestamp` < DATE_SUB(NOW(6), INTERVAL 1 MINUTE);
```