# CoObRA: Eine Plattform zur Verteilung und Replikation komplexer Objektstrukturen mit optimistischen Sperrkonzepten

Dissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
(Dr. rer. nat.)

im Fachgebiet Software Engineering
Prof. Dr. Albert Zündorf,
Fachbereich Elektrotechnik / Informatik
der Universität Kassel

vorgelegt von
Dipl. Inform. Christian Schneider
Disputation am 30. November 2007

**Zusammenfassung**

In der vorliegenden Arbeit wird die Konzeption und Realisierung der Persistenz-, Verteilungs- und Versionierungsbibliothek CoObRA 2 vorgestellt. Es werden zunächst die Anforderungen an ein solches Rahmenwerk aufgenommen und vorhandene Technologien für dieses Anwendungsgebiet vorgestellt. Das in der neuen Bibliothek eingesetzte Verfahren setzt Änderungsprotokolle beziehungsweise -listen ein, um Persistenzdaten für Dokumente und Versionen zu definieren. Dieses Konzept wird dabei durch eine Abbildung auf Kontrukte aus der Graphentheorie gestützt, um die Semantik von Modell, Änderungen und deren Anwendung zu definieren.

Bei der Umsetzung werden insbesondere das Design der Bibliothek und die Entscheidungen, die zu der gewählten Softwarearchitektur führten, eingehend erläutert. Dies ist zentraler Aspekt der Arbeit, da die Flexibilität des Rahmenwerks eine wichtige Anforderung darstellt. Abschließend werden die Einsatzmöglichkeiten an konkreten Beispielanwendungen erläutert und bereits gemachte Erfahrungen beim Einsatz in CASE-Tools, Forschungsanwendungen und Echtzeit-Simulationsumgebungen präsentiert.

**Abstract**

This thesis presents the conception and realization of the persistency, distribution and versioning library CoObRA 2. Foremost the requirements for such a framework are analyzed and defined. Subsequently existing technologies from the application area and related work are introduced. Afterward the approach used in the library is presented. It utilizes change protocols to define persistent data and versions of documents. This concept is based on a mapping to constructs from graph theory, to define the semantics of models, changes and their application.

The desciption of the realization focuses on the design of the library and the decisions which were leading to the software architecture, afterwards. This is a central aspect of the presented work, as the flexibility of the framework was an important requirement. The thesis concludes with presenting several example applications, including the experiences gained while using the library in CASE tools, research programs and real time simulation software.

# Inhaltsverzeichnis

# 1 Einleitung

Aus heutigen Softwareentwicklungsprozessen sind Werkzeuge zur Versionierung von Dokumenten und insbesondere von Quelltexten nicht mehr wegzudenken. Sie stellen nicht nur Methoden bereit mit der Evolution eines Softwareprojektes umzugehen, sondern sind auch das zentrale Mittel zur Einrichtung einer kollaborativen Arbeitsumgebung. Der Einsatz von Versionierung ist allerdings nicht nur auf die Softwaretechnik beschränkt. Mehr und mehr setzen sich Versionierung und die Zusammenarbeit über Softwaresysteme in diversen Bereichen durch.

Weitere etablierte Vorgehensweisen in der Softwareentwicklung sind die objektorientierte und die modellzentrierte Entwicklung von Software. Dadurch entsteht die Notwendigkeit Objektstrukturen beziehungsweise Modelle in objektorientierten Endanwendungen und auch Metamodelle in CASE-Tools zu versionieren. Die für diese Aufgabe etablierten Techniken und Rahmenwerke erfüllen jedoch nicht das volle Anforderungsspektrum, sind nicht immer leicht zu nutzen und sind zum großen Teil proprietär für eine spezielle Anwendung. Insbesondere das nebenläufige Arbeiten beziehungsweise das Arbeiten ohne Netzwerkverbindung wird nur von proprietären Lösungen hinreichend unterstützt.

Mit dem Ansporn diese Lücke zu füllen wurde das CoObRA Konzept aus der Taufe gehoben. Während des Testens der ersten Implementierung und dem Einsatz in verschiedenen Anwendungen sind die Anforderungen an die zweite Fassung der CoObRA Bibliothek noch gewachsen. Daher werden

hier zunächst einige Szenarien für den gewünschten Einsatz der Bibliothek skizziert.

## CASE-Tools

CASE-Tools im Allgemeinen werden intensiv in Softwareentwicklungsprozessen eingesetzt. Wie eingangs beschrieben, heißt dies bei größeren Projekten auch sofort für die von diesen Tools produzierten Daten, dass sie unter Versionskontrolle gestellt werden. Im Speziellen wurden für diese Arbeit die Anforderungen des Fachgebiets von Prof. Zündorf an die dort mitentwickelte FUJABA Tool Suite[1] angenommen. Auch mit FUJABA wurden zunehmend größere Forschungsprojekte realisiert und dadurch mussten größere Modelle bearbeitet werden. Daher kann auch bei der Verwendung von FUJABA zur Entwicklung von Software mittlerweile auf nebenläufiges Arbeiten mit mehreren Entwicklern, Versionshistorie, Mischen und Verzweigen von Versionen nicht verzichtet werden. Auch die Integration von FUJABA-Dateien in bestehende SCM Repositories mit anderen Ressourcen und Quelltexten ist für einen reibungsfreien Entwicklungsprozess wünschenswert. Die Anforderungen orientierten sich daher stark an den Möglichkeiten, die herkömmliche Source Code Management (SCM) Systeme für die Quellcodeverwaltung mit wachsenden Entwicklerteams bieten.

## Szenario aus der Softwareentwicklung

Um einen Eindruck von der Arbeit mit einem Source Code Management (SCM) System zu vermitteln, wird nun kurz ein mögliches Szenario aus der späten Phase eines Softwareentwicklungsprozesses skizziert: Arbeiten zwei

[1] FUJABA ist ein CASE-Tool und Forschungsplattform für mehrere Universitäten. Es wurde ursprünglich von Prof. Zündorf ins Leben gerufen[4]. Siehe auch http://www.fujaba.de/

oder mehr Entwickler gemeinsam an demselben Quellcode, kommt nahezu immer eine SCM-Software zum Einsatz. Von dieser erhält jeder Entwickler eine Kopie des bearbeiteten Quelltextes. Teilweise wurden (und werden) *pessimistische Sperrkonzepte* eingesetzt, diese haben jedoch zur Folge dass ganze Dateien oder gar Verzeichnisse, nur von je einer Person bearbeitet werden können. Dadurch behindern sich die Entwickler teilweise in Ihrer Arbeit. Werden dagegen *optimistische Sperrkonzepte* eingesetzt kann jeder Entwickler in seiner Kopie Änderungen vornehmen, ohne dass dies Auswirkungen auf die Kopien seiner Kollegen hat. So können neue oder geänderte Codezeilen zunächst von dem Autor ausgiebig getestet oder verifiziert werden.

Ist eine Teilaufgabe soweit abgeschlossen, dass ein Entwickler seine Arbeit den Kollegen verfügbar machen will, sendet er die gewünschten Änderungen an den SCM-Server. Diese werden dort zunächst nur abgelegt. Die anderen Nutzer des Systems können diese Neuerungen dann - zu einem von ihnen gewählten Zeitpunkt - vom Server herunterladen. Bei diesem Vorgang werden ihre eigenen ungesendeten Änderungen mit denen vom Server gemischt und es entsteht eine neue Version ihrer eigenen Quellcodekopie. Dabei werden zeilenweise Konflikte erkannt. Das bedeutet Textzeilen, die sowohl vom aktuellen Entwickler als auch von einem Kollegen geändert wurden, werden mit speziellen Markierungen versehen. Es bleibt dann dem Entwickler überlassen die Konflikte geeignet zu beheben - eventuell auch mit Rücksprache im Kreise seiner Kollegen.

Diese Arbeitsweise hat sich für textbasierte Dokumente in der Softwareentwicklung bewährt und soll nun auch in CASE-Tools auf Basis der Modelle vermehrt zum Einsatz kommen.

## Effizienz und Arbeitsaufwand

In Softwareentwicklungsprozessen ist Teamarbeit gerade bei größeren Projekten gefragt. Mit wachsendem (Meta-)Modell erhöhen sich jedoch auch die Anforderungen an die Performanz - auch weitergehende Mechanismen werden eventuell nötig: So könnten etwa Projekte in mehrere Modelldateien aufgeteilt werden, die dann jedoch Querbeziehungen unterstützen müssen. Es könnten nur Ausschnitte aus Dateien oder gar von entfernten Rechnern geladen werden, wenn nicht das komplette Projekt zum Arbeiten benötigt wird. Ebenso ist eine hierarchische Organisation des Projekts analog zu einer etwaigen Personalstrukturierung denkbar. So könnten auch Teams von unterschiedlichen Standorten untereinander einen Server zur Zusammenarbeit verwenden, während sich dieser bei Bedarf mit dem Server eines anderen Teams synchronisieren lässt.

All diesen hohen funktionalen Erwartungen an einen Versionierungsmechanismus steht sogleich der Wunsch nach möglichst geringem Integrationsaufwand gegenüber: Ein Werkzeug wie FUJABA, mit hunderttausenden Zeilen an Quelltext, soll in möglichst kurzer Zeit für den Einsatz des Mechanismus' vorbereitet werden können. Auch bei anderen Applikationen, die bereits existieren und eine eventuell handgeschriebene Implementierung eines Metamodells einsetzen, ist der Integrationsaufwand ein Kriterium, das über den Einsatz einer Bibliothek entscheidet.

## Generierte Anwendungen

Ein weiteres wichtiges Einsatzgebiet von Persistenz- und Versionierungsmechanismen ist die Unterstützung von neu entworfenen Anwendungen. Insbesondere geht es dabei um Forschungsanwendungen aus dem Fachgebiet. So wird zum Beispiel das Metamodell der Anwendung für das Forschungsprojekt *Optimale Bordnetz Architektur* (OBA) mit FUJABA generiert. Die

Modelle in der Applikation müssen natürlich gespeichert werden. Allein diese Anforderung erzeugte bisher großen Arbeitsaufwand für den Entwickler. Mit der CoObRA Bibliothek soll der Aufwand dank Integration in den von FUJABA generierten Code nahe Null liegen. Kommen dann weitere Anforderungen an die Datenverwaltung für die Applikation hinzu, werden nach und nach die weitergehenden Funktionen der Bibliothek - wie Versionieren, Replizieren und Mischen - genutzt.

## Echtzeitnahe Anwendungen

Ein anders gelagertes Einsatzgebiet von Persistenz und Replikation sind echtzeitnahe Anwendungen wie interaktive Simulationsumgebungen und Computerspiele. Mit den bisher vorgestellten Szenarien haben sie die Notwendigkeit gemein, ihre Modelldaten speichern zu können, die Versionierung ist jedoch meist nicht nötig. Weiterhin sind die Anforderungen an die Replikation von Daten jedoch von den bisherigen Anforderungen recht stark zu unterscheiden: Bei einer verteilten interaktiven Anwendung - sei es Simulation oder Spiel - ist es notwendig die internen Zustände der verschiedenen Programminstanzen zu synchronisieren, um den Benutzern einen konsistenten Zustand präsentieren zu können. Dies kann sogar soweit gehen, dass die Synchronisation framegenau (das heißt etwa 30-60 mal pro Sekunde) korrekt stattfindet, wenn ein und derselbe Benutzer mehr als eine Programminstanz gleichzeitig betrachtet (zum Beispiel in einer CAVE[2] auf Basis eines Clusters). Eine Lösung für solche Applikationen kann es sein die Daten in einem gemeinsamen Speicher vorzuhalten und auf Replikation zu verzichten. Diese Möglichkeit ist jedoch nicht immer gegeben, da gemeinsamer Speicher (sha-

[2] „Der aus dem englischen stammende Begriff Cave Automatic Virtual Environment (abgekürzt: CAVE; wörtlich übersetzt: „Höhle mit automatisierter, virtueller Umwelt“) bezeichnet einen Raum zur Projektion einer dreidimensionalen Illusionswelt der virtuellen Realität.“ [wikipedia.org]

red memory) zum Beispiel in Clustern nicht unbedingt zur Verfügung steht. Wird auf Replikation und Synchronisation der Daten zurückgegriffen soll die in dieser Arbeit entwickelte Bibliothek zum Einsatz kommen können.

## 1.1 Forschungsgegenstand

Gegenstand dieser Arbeit ist die Erforschung eines Ansatzes zur persistenten Speicherung von Objektstrukturen, der bereits in [22], [24] und [23] vorgestellt wurde. Zu erforschen war zum Einen die Tragfähigkeit des Ansatzes in verschieden Anwendungsgebieten und -szenarien, zum Anderen war die Universalität beziehungsweise Vielseitigkeit auszuloten.

Die prinzipielle technische Eignung des Ansatzes wurde bereits in der Diplomarbeit [22] gezeigt. Die dort gezeigten Features waren Persistenz, Undo/Redo und Mehrbenutzerfähigkeiten. Um die Vielseitigkeit des Ansatzes zu zeigen, sollten diese während der Promotionszeit um hierarchische Repositories, Transaktionskonzepte, Teilgraphen-Persistenz und Loading on Demand erweitert werden.

Die für die Dissertation zu erforschenden Einsatzgebiete umfassten CASE-Tools - insbesondere das im Fachgebiet entwickelte CASE-Tool FUJABA - sowie damit generierte Anwendungen, einschließlich der Software des Projekts OBA, weiterhin MOF/JMI-kompatible Software, proprietäre objektorientierte Anwendungen und den Einsatz in echtzeitnahen Softwareanwendungen wie 3D-Spielen.

Als Basis dieser Forschungen entstand die Bibliothek CoObRA 2. Diese Bibliothek deckt das komplette soeben beschriebene Funktionsszenario ab. Bei der Entwicklung der Bibliothek wurde auch auf die produktive Einsetzbarkeit geachtet sowie die Performanz optimiert. Hierzu wurde das Design der Software neben der Modularität auch auf Laufzeitkomplexität und

Speicherverbrauch abgestimmt. Die Bibliothek wurde während der Tests auf Performanzengpässe untersucht und diese wurden beseitigt.

### 1.1.1 Ansatz

Die grundlegende Idee zur Umsetzung der Anforderungen und der in der Arbeit getestete Ansatz stützen sich auf die Protokollierung von atomaren Änderungen an der Objektstruktur. Eine Liste von Änderungsdaten kann dann rückwärts (undo) oder vorwärts (redo/laden) erneut auf ein Modell angewendet werden, um zwischen Modellzuständen zu wechseln. Auf dieser Basis werden Versionierung, Replikation und Mischen realisiert. Dies unterscheidet den Ansatz auch substantiell von anderen Lösungen, die Momentaufnahmen einer Objektstruktur machen und diese im Nachhinein miteinander vergleichen. Daher ist ein Aspekt der vorliegenden Arbeit die Eignung des Ansatzes im Vergleich mit anderen Lösungen herauszustellen und etwaige Vor- und Nachteile aufzuzeigen.

## 1.2 Aufbau des Dokuments

Im Folgenden werden nun die Anforderungen an einen Persistenzmechanismus für objektorientierte Anwendungen beschrieben. In Kapitel 2 werden diese Anforderungen zunächst festgelegt. In Kapitel 3 werden dann etablierte Techniken zur Objektpersistenz diskutiert und deren Vor- und Nachteile, insbesondere bezogen auf die zuvor definierten Anforderungen, dargestellt. Der zu erforschende Ansatz wird danach in Kapitel 4 vorgestellt und dessen Umsetzung in der CoObRA 2 Bibliothek in Kapitel 5 beschrieben. Es folgt dann Kapitel 6 über den Einsatz der entwickelten Bibliothek in verschiedenartigen Softwaresystemen. Abschließend wird noch ein kurzer Ausblick auf zukünftige Forschungs- und Entwicklungsarbeit gegeben.

# 2 Anforderungen

In diesem Kapitel werden die vor Beginn der Arbeit definierten Anforderungen an das Konzept und die daraus entstandene Bibliothek CoObRA 2 dargestellt.

## 2.1 Kernanforderungen

Die wichtigste Anforderung an das zu konzipierende System ist die Eignung zur persistenten Speicherung von anwendungsspezifischen Objektstrukturen[1]. „Anwendungsspezifisch“ bedeutet in diesem Fall insbesondere, dass das Metamodell der Objektstrukturen unter Umständen nicht oder nur geringfügig an eine Persistenzbibliothek angepasst werden kann. Die Notwendigkeit einer Instrumentierung oder anderweitigen tiefgreifenden Veränderung der modellseitigen Implementierung soll vermieden werden. Unter persistenter Speicherung wird hier die Externalisierung einer Objektstruktur in einen geeigneten Speicher verstanden. Dieser Speicher kann dabei sowohl der Arbeitspeicher als auch eine Datenbank, eine Datei auf der Festplatte oder ein anderer Datenstrom sein. Die persistente Speicherung impliziert ebenso die Möglichkeit zum anschließenden Laden des Datenstroms, um daraus wieder eine Objektstruktur herzustellen. Die wiederhergestellte Objektstruk-

[1] Anschaulich wird in dieser Arbeit unter Objektstruktur der logische Speicherinhalt eines objektorientierten Programms verstanden - also eine Menge von Objekten oder Knoten und deren Links beziehungsweise Kanten; für eine formale Definition siehe Abschnitt 4.1

tur muss dann mit der zuvor gespeicherten Objektstruktur übereinstimmen (siehe Gleichheit).

Eine Erweiterung zur persistenten Speicherung ist die Funktion der Replikation und Synchronisation von Objektstrukturen in derselben oder weiteren Programminstanzen. Während eine einfache Replikation durch das schlichte Speichern und spätere Laden einer Objektstruktur stattfinden kann, wird für die Synchronisation von bereits replizierten Datenstrukturen mindestens ein inkrementelles Einspielen von neuen Versionen der Persistenzdaten benötigt. Das heißt es werden nur die Änderungen am Modell werden vorgenommen, die zum Erreichen der neuen Version nötig sind, anstatt ein Modell komplett neu zu laden.

Ein weiteres Aufgabengebiet ist *Undo* und *Redo*. Einer Applikation soll die Möglichkeit zur Verfügung gestellt werden, Änderungen an ihrer internen Objektstruktur rückgängig zu machen beziehungsweise einmal rückgängig gemachte Änderungen zu wiederholen. Dabei soll die Anwendung nicht, wie sonst beim Command-Pattern [6] üblich, selbst die Operationen zum Rückgängig-Machen definieren und benötigte Daten vorhalten, sondern ein Bibliotheksaufruf soll die geeigneten Operationen ausführen.

Dies leitet über zu der Forderung nach einem Transaktionsbegriff. Ein solcher bietet sich zum Beispiel an, um zu definieren, welchen Umfang eine *Undo*-Operation haben soll. Auch soll eine Applikation mittels Transaktionen definieren können, dass sich die interne Objektstruktur nach einer abgeschlossenen Transaktion wieder in einem für die Applikation konsistenten Zustand befindet. Da Operationen auf einem Objektmodell gängigerweise geschachtelt werden, wird auch für die Transaktionen die Möglichkeit zur Schachtelung verlangt.

Eine Isolierung[2] von Transaktionen gegeneinander wird nur zwischen verschiedenen Replikaten verlangt, nicht jedoch innerhalb derselben Objektstruktur. Hierauf kann im Mehrbenutzerbetrieb gängigerweise problemlos verzichtet werden, da es mindestens ein Replikat pro Benutzer gibt. Letztere Forderung würde nämlich in Konflikt mit der eingangs beschriebenen Anforderung stehen, dass die Persistenzschicht direkt auf den anwendungsspezifischen Objektstrukturen arbeitet. Eine Anwendung kann jedoch Transaktionen von einander isolieren, indem sie den Replikations- und Synchronisationsmechanismus nutzt.

Die letzte Kernanforderung ist die Möglichkeit verschiedene Versionen einer Objektstruktur *mischen* zu können und der Applikation eine Schnittstelle zur Verfügung zu stellen, mit der auf das Mischen Einfluss genommen werden kann. So sollen syntaktische Konflikte von der Bibliothek erkannt werden und die Konfliktlösung - eventuell mit Benutzerinteraktion - durchgeführt werden. Die Erkennung von etwaigen semantischen Konflikten soll allerdings der Applikation überlassen werden - ebenso die Lösung derselben.

## 2.2 Ergänzende funktionale Anforderungen

Eine zusätzliches wichtiges Kriterium das von der entwickelten Bibliothek erfüllt werden sollte und auf die Replikation und Synchronisation aufsetzt, ist die Möglichkeit zur Client-Server-Kommunikation. Mit ihr sollen Applikationen auf einfache Weise - über Rechnergrenzen hinweg - Modellinformationen austauschen. So kann zum Beispiel eine kollaborative Arbeitsum-

[2] Von Isolierung oder Entkopplung von Transaktionen spricht man, wenn Operationen die in eine Transaktion gebettet werden, die Durchführung anderer Transaktionen nicht beeinflussen. In dieser Arbeit ist damit insbesondere gemeint, dass Schreibzugriffe innerhalb einer Transaktion erst dann auf Lesezugriffe außerhalb derselben Transaktion Auswirkungen haben, wenn die Transaktion abgeschlossen ist. Im Allgemeinen gibt es verschiedene Abstufungen von Transaktionsisolierungen.

gebung schnell realisiert werden.

Ein Client soll also die Möglichkeit haben seine Objektstrukturen als Replikat auf einem Server abzulegen. Ein weiterer Client kann dann wiederum ein Replikat der Serverdaten erstellen. Durch die Synchronisation können dann Änderungen an den Client-Replikaten in die Serverdaten eingespielt werden. Kann jeder Client auch ein Server sein beziehungsweise sind Client und Server gleichwertig, spricht man von Peer-to-Peer Kommunikation. Diese Kommunikation über Synchronisation von Replikaten soll auch über die verschiedenen Kommunikations-Kanäle möglich sein. Ein für die Anwendung im CASE-Tool-Bereich besonders interessanter Kommunikationskanal ist die Textdatei basierte Kommunikation über einen Source-Code-Management-Server (SCM-Server). Über diese Kommunikationsvariante können dann nämlich nicht nur Modelldaten sondern auch Quelltexte und andere Ressourcen in einem Versionierungsmechanismus gemeinsam verwaltet werden. Dies erleichtert die Arbeit der Anwender durch eine einheitliche Verwaltungssoftware und den Einsatz bereits bekannter Software.

Eine Anforderung die ebenfalls aus den bestehenden Möglichkeiten von Software-Konfigurations-Management Programmen herrührt, ist die hierarchische Organisation von Servern und deren Repositories. Das bedeutet, ein Server für einige Clients kann wiederum selbst ein Client von einem weiteren Server sein. Es soll so möglich sein, die oft hierarchische Organisation von Benutzer- oder Entwicklergruppen auch auf der Datenhaltungsschicht abbilden zu können. Abbildung 2.1 zeigt eine Skizze von mehreren Teamservern, die ihrerseits mit einem übergeordneten Server abgeglichen werden können.

Die folgenden Anforderungen zielen zum Teil auf die Performanz der Bibliothek. Da sie aber bereits als konkrete Funktionalitäten vorliegen, werden Sie in diesem Abschnitt aufgeführt. Um Kommunikationsaufwand bezie-

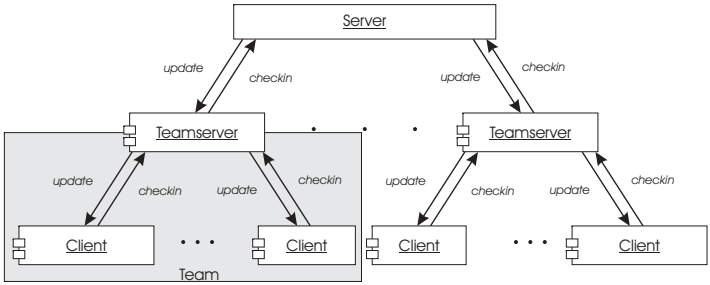


Abbildung 2.1: *Beispiel für die hierarchische Organisation von Servern*

hungsweise Zeit zu sparen, kann es sinnvoll sein, nur Teile der Persistenzdaten wieder in die Objektstrukturen zurück zu wandeln. Um dies zu ermöglichen, sollen Filter auf die zu ladenden Daten angewendet werden können. Ebenso soll es möglich sein beim Schreiben der Daten bereits Filter einzusetzen. Weiterhin sollen Teile der Objektstrukturen in andere Repositories ausgelagert werden können. Dabei entstehen üblicherweise auch Querreferenzen zwischen den unterschiedlichen Repositories. Dieser Themenkomplex wird in dieser Arbeit als Teilgraphen-Persistenz bezeichnet.

Wird beim Wiederherstellen von Persistenz-Daten ein Filter verwendet, kann es zu einem späteren Zeitpunkt sinnvoll sein, weitere Persistenz-Daten in den Objekt-Speicher zu laden. Passiert dies bei Bedarf automatisch, spricht man von Loading-on-Demand. Auch diese Funktionalität sollte die Bibliothek so weit möglich unterstützen.

Obwohl Kopieren und Einfügen keine originäre Anforderung an die Bibliothek war, soll hier erwähnt werden, dass auch diese Funktionalität mit dem Thema Persistenz verbunden ist. Sollen Daten in einer Anwendung „kopiert“ werden und in einer anderen Programminstanz wieder eingefügt werden, ist das Problem stark verwandt mit dem Speichern (beziehungswei-

se Serialisieren) und anschließendem Laden von Teilgraphen. Hinzu kommt jedoch noch die Schwierigkeit den Teilgraphen - abhängig von der Selektion des Benutzers - geeignet zu wählen. Da auch hier die vorgestellte Bibliothek große Hilfestellungen geben kann, wird „Kopieren und Einfügen“ ein der Abschnitt 6.6 gewidmet.

## 2.3 Nichtfunktionale Anforderungen

Die wichtigste nicht-funktionale Anforderung wurde bereits im ersten Abschnitt unter den Anforderungen erwähnt. Es handelt sich um die Anwendbarkeit des Ansatzes auf nahezu beliebig implementierte Meta-Modelle. Die Anforderungen, die eine Anwendung bezüglich eines Meta-Modells erfüllen muss, sollen also möglichst gering gehalten werden. Wie in Abschnitt 3 noch dargestellt, unterstützen viele existierende Persistenz-Mechanismen dies nicht. Viele Ansätze erfordern sogar, dass das Meta-Modell der Anwendung mit einem speziellen Generator erzeugt wird, den der Persistenz-Mechanismus zur Verfügung stellt.

Eine weitere nicht-funktionale Anforderung, die schwer zu quantifizieren ist, ist die zu erreichende Performanz des Frameworks. Insbesondere das Laden einer Objektstruktur soll in möglichst kurzer Zeit passieren. Diese Anforderung gilt auch für das Laden der durch SCM-Systeme verwalteten Dateien. Diese sind notwendigerweise Textdateien, was die Optimierung der Ladegeschwindigkeiten eventuell einschränkt. Auch die Kommunikation über eine Netzwerkverbindung soll möglichst echtzeitnah durchführbar sein. Für den Einsatz in echtzeitnahen Anwendungen wie Spielen, ist auch eine übermäßige Erzeugung von Garbage[3] zu verhindern. Ein Garbage-Collector

[3] Objekte die nach kurzer Zeit nicht mehr benötigt werden und vom Garbage-Collector eingesammelt werden müssen.

verursacht ansonsten eventuell nicht umgehbare Verzögerungen beim Ablauf der Anwendung.

## 2.4 Zielplattform

Die Konzepte für CoObRA 2 sollten so allgemein wie möglich gehalten werden. Die konzeptionelle Zielplattform ist daher eine beliebige Umgebung für objektorientierte (siehe Definitionen in Kapitel 4) Applikationen. Für die Implementierung der Bibliothek wurde allerdings die Sprache Java gewählt, da diese im Fachgebiet von Prof. Zündorf weitgehend eingesetzt wird. Daher sind auch das Einsatzgebiet der aktuellen Bibliotheksimplementierung und somit alle Beispielanwendungen auf Java-Anwendungen beschränkt. Doch die Sprache soll soweit möglich die einzige Beschränkung bezüglich des Einsatzes der Bibliothek darstellen. Weiterhin soll der Einsatz auf Client- sowie auf Server-Seite möglich sein - mit und ohne Kenntnis des Metamodells der Applikation. Details zur Nutzung ohne Kenntnis des Metamodells finden sich in Abschnitt 5.4.1.

# 3 Existierende Lösungsansätze

Es sollen hier einige Ansätze und Beispielprojekte vorgestellt werden, die zumindest Auszüge aus dem vorgestellten Anforderungsprofil erfüllen. Dabei wird dieser Abschnitt geteilt in bereits in der Industrie etablierte Techniken und interessante Forschungsansätze.

## 3.1 Etablierte Techniken

Zunächst werden Techniken und Frameworks, die bereits produktiv eingesetzt werden und einen merklichen Verbreitungsgrad haben, betrachtet:

### 3.1.1 Dateibasierte Persistenz

Ein sehr einfacher Persistenzmechanismus ist das Speichern von Objektdaten in einer Text- oder Binärdatei. Hier weichen die Anforderungen signifikant von den Anforderungen in dieser Arbeit ab. Häufige Anforderung an einen Applikationsentwickler ist es, auf einfache Weise die komplette Objektstruktur im Anwendungsspeicher abzuspeichern und nach einem Neustart der Applikation wieder zu laden. Manchmal werden auch Graphen über Netzwerk übermittelt. Insbesondere bei Kommunikation über das Protokoll HTTP sind dabei Textdaten erforderlich, wie zum Beispiel XML. Aspekte der Versionierung werden bei dieser Form der Speicherung weitestgehend vernachlässigt (siehe dazu auch Mischen von XML in Abschnitt 3.2.4).

### Seralisierung

Der häufig am schnellsten zum Erfolg führende Weg zum Speichern von Javaobjekten ist die in der Java Umgebung enthaltene *Serialisierung*[1]. Eine Klasse muss schlicht ein Interface names *Serializable* implementieren, nicht gespeicherte Attribute als *transient* markieren und alle referenzierten Objekte müssen ebenfalls serialisierbar sein. Daraufhin können dann Instanzen dieser Klasse als binärer Datenstrom gespeichert werden. Dabei wird keine Abstraktion vorgenommen, es werden schlicht der Klassenname für die Instanziierung und ein Wert für die Belegung jedes Java Attributs abgelegt. Die Deserialisierung passiert analog und ist mittlerweile durch gespeicherte Attributnamen (beziehungsweise IDs) auch ein wenig gegen Schemaänderungen (neue Attribute in Klassen et cetera) gesichert. Der Serialisierung wird mittlerweile die Externalisierung[2] vorgezogen, die jedoch für die Arbeit konzeptionell keinen Unterschied bringt. Die Ausgabe des Datenstroms als XML-Daten ist mittlerweile ebenso möglich. Eine Versionierung oder andere weitere Features sind aber nicht vorgesehen - weder für binäre noch für XML-Dateien.

### JAXB, XMI und andere XML-basierte Persistenz

Eine ausgeklügeltere Variante XML-Persistenz für Javaobjekte zur Verfügung zu stellen bietet JAXB [5]. Die JAXB Schnittstelle sieht im Gegensatz zur Serialisierung in XML eine Schemabeschreibung und eine Abbildung auf Java beim Speichern vor. Die resultierenden XML-Dateien sind jedoch von der Struktur her mit denen der XML-Serialisierung vergleichbar und bieten ähnlich wenig Spielraum für Versionierung und Mischen.

Als Beispiel für die metamodellgebundenen Persistenzmechanismen sei

[1] java.io.Serializable stellt dafür im Java Development Kit ein Interface zur Verfügung.
[2] java.io.Externalizable ist ebenfalls ein Interface aus dem JDK.

hier noch XMI [17] erwähnt. Vorschriften für Implementierungen von MOF Modellen (zum Beispiel JMI [26], EMF [3]) schreiben u.a. vor, dass Modelle in dem XML-Dialekt XMI abgespeicherbar sind. Dieser dient jedoch wie andere XML-Dateien eher dem Datenaustausch zwischen verschiedenen Programmen als der Versionierung und anderen erweiterten persistenzbezogenen Funktionen.

### 3.1.2 Dateibasierte Versionierung

Dateibasierte Versionierung wird in der Softwaretechnik insbesondere für die Verwaltung von Quelltextdateien verwendet (Source Code Management, SCM). Die verbreitetsten sind dabei CVS [34] (entstanden aus RCS [29]) und Subversion [36], diese sind frei verfügbar. Zu bekannten kommerziellen Versionsverwaltungen zählen Visual SourceSafe [14], Perforce [30] und Rational ClearCase [32]. Neben der Verwaltung von Quelltextdateien unterstützen alle diese Systeme natürlich auch andere Textdateien sowie beliebige Binärdateien. Während bei Textdateien teilweise noch ein Mischen von zwei nebenläufig veränderten Versionen unterstützt wird, wird bei Binärdateien maximal ein Mischen durch Zusatzmodule für einzelne Dateitypen unterstützt. Beide Verfahren lösen jedoch das eigentliche Problem - das Mischen von verschiedenen Versionen einer etwaigen Objektstruktur - nicht. Nach dem Mischen von in Textdateien gespeicherten Objektstrukturen können diese oft nicht mehr korrekt geladen werden und der Benutzer muss die Dateien zunächst manuell korrigieren. Da die Formate aber maschinell erzeugt werden und nicht handgeschrieben sind, fällt dies sehr schwer.

Trotz des fehlenden Aspekts des Mischens von Objektstrukturen, dienten die dateibasierten Versionierungswerkzeuge für diese Arbeit als Vorbild. Die zu entwickelnde Bibliothek CoObRA 2 sollte für Objektstrukturen das werden, was SCM Werkzeuge für Textdateien sind. Weiterhin wird wie in

den Anforderungen bereits dargestellt, eine Integration in etablierte SCM-Systeme angestrebt.

### 3.1.3 Datenbanklösungen

Ein sehr häufig verwendeter Ansatz, wenn es um Persistenz und Nebenläufigkeit geht, ist der Einsatz von Datenbanken (Datenbankmanagementsystemem - DBMS) beziehungsweise von sogenanntem Object Relational Mapping (ORM). Diese Techniken erfüllen jedoch andere Anforderungen als die eingangs vorgestellten. Die signifikantesten Unterschiede sind dabei die oft fallen gelassene (oder nicht existente) Forderung nach optimistischen Sperrkonzepten (beziehungsweise langen Transaktionen) mit Konfliktlösung, Mischen und dem sogenannten *Sandboxing*, das es jedem Benutzer gestattet seine Änderungen zunächst nur lokal durchzuführen und später an Kollegen weiterzugeben. Auch die Flexibilität bezüglich des Metamodells steht hier nicht im Vordergrund. Weiterhin werden meist anders gelagerte Performanzanforderungen gestellt: Es werden nur kleine Ausschnitte der Objektstrukturen aus der Datenbank geladen, performantes Laden der kompletten Struktur sowie performante Verarbeitung von kleinen Änderungen an einem gesamten Modell sind nicht gefragt. Stattdessen werden Teile der Objektstrukturen mit einer Abfragespache (oft SQL oder ähnlich) angefragt, in Java Objekte geladen, modifiziert und danach wieder in die Datenbank geschrieben. Passt das Anforderungsprofil und die Programmierung der Anwendung (zum Beispiel SQL statt Suche mit Java-Code), skalieren Datenbanklösungen sehr gut. Sind allerdings komplexe Suchoperationen oder Modifikationen auf einem großen Modell auszuführen und ist eine Implementierung in einer Abfragesprache nicht mehr praktikabel, ist der Einsatz von Datenbanken in dieser Weise nicht angezeigt.

Auch bezüglich Transaktionen werden an Datenbanklösungen oft ande-

re Anforderungen gestellt. So ist es üblich Transaktionen gegeneinander zu isolieren - auch innerhalb ein und derselben Applikation. Insbesondere innerhalb von Serverdiensten ist dies notwendig. Was dagegen nur teilweise möglich ist (je nach Datenbank) ist das Zurückrollen von bereits abgeschlossenen Transaktionen (Undo), beziehungsweise das Zurücksetzen auf eine frühere Version der Objektstrukturen.

Es gibt diverse Frameworks, die sich mit dieser Form der Persistenz befassen. Beispielhaft sollen hier die zwei populärsten näher beleuchtet werden.

### Hibernate

Hibernate [35] ist mittlerweile eine der am weitesten verbreiteten objektorientierten Datenbankanbindungen für Java. Das Konzept ist so einfach wie effektiv: Sogenannte *plain old java objects* (POJO, „normale“ Java-Objekte) werden nach Standardvorgaben oder nach speziellen Annotationen inspiziert (Java Reflection). Die so gewonnen Daten werden nach einer (vorgegebenen oder selbst in XML definierten) objektrelationalen Abbildung (ORM) in eine relationale Datenbank geschrieben. Später können sie mit einer einfachen SQL-ähnlichen Sprache wieder abgefragt und in Java-Objekte zurückgewandelt werden. Bei der Wahl der verwendeten Datenbank ist der Programmierer weitgehend frei, sofern eine Javaschnittstelle (JDBC) existiert. Die Implementierung der verwendeten Javaobjekte ist beliebig wählbar aber lässt für viele Anwender noch genug Freiheiten. Die Implementierung muss sich nach dem De-facto-Standard *Java Persistence API* [2] richten. Dieser ist mittlerweile in Form von Interfaces in die Java-Plattform von Sun integriert.

#### Cayenne

Cayenne verfolgt einen anderen Ansatz als Hibernate bezüglich der zu verwendenden Implementierung für das Objektmodell. Der Programmierer kann sie nicht selbst vorgeben, sondern das Rahmenwerk generiert die Java-Klassen aus einer Beschreibungsdatei. Dadurch können die datenhaltenden Objekte direkt, ohne Reflection, vom Rahmenwerk ausgelesen werden. Weiterhin bietet Cayenne im Gegensatz zu Hibernate eine grafische Benutzoberfläche zum Definieren des Modells und der objektrelationalen Abbildung.

### 3.1.4 Proprietäre Realisierungen

In industrieller Software gibt es vereinzelt auch Mechanismen für das Vergleichen und Mischen von Modellen. Insbesondere in den CASE-Tools der Marktführer sind diese Techniken zu finden. Ein Beispiel hierfür ist der „Model Integrator“ von IBM Rational [9]. Kein (dem Autor bekanntes) Werkzeug ist jedoch universell einsetzbar, das heißt sie sind auf die proprientären Datenstrukturen des jeweiligen CASE-Tools ausgelegt. Auch eine Schnittstelle für domänenspezifische Darstellung der Unterschiede oder des Mischergebnisses bieten sie nicht. Für die Versionierung und Replikation werden auch hier oft oben genannte Werkzeuge für dateibasierte Versionierung benutzt, teilweise auch Datenbanken.

## 3.2 Ansätze aus der Forschung

Die Ansätze für Persistenz, Versionierung und Mischen gehen in der Forschung bereits weiter als bisher etablierte Techniken. Hier sollen kurz einige interessante Ansätze präsentiert werden.

### 3.2.1 CoAct, Darmstadt

Ein 1995 vorgestelltes Versionierungs- und Mischmodell von der GMD[3] Darmstadt namens *CoAct* [21, 33, 12] liegt thematisch sehr nahe an der hier vorgestellten Arbeit. In CoAct werden Protokolle der durchgeführten Änderungsaktionen samt Parametern aufgezeichnet. Der signifikante Unterschied zu CoObRA ist die Granularität der Änderungsaktionen: Diese Aktionen können bei CoAct beliebig komplex sein.

Die Versionierung und das Mischen von Versionen geschieht auf Basis dieser Aktionen, beziehungsweise Aktionslisten. Mit Vor- und Nachteilen behaftet sind dabei die strikten Vorgaben für diese Aktionen: Jede Aktion muss ein gültigen Modellzustand in einen anderen gültigen Modellzustand überführen. Dabei bedeutet gültig syntaktisch und semantisch[4] korrekt. Weiterhin muss eine parametrisierte Aktion prüfen können, ob sie auf einen Modellzustand angewendet werden kann, oder nicht.

Diese Anforderungen an die Aktionen erlauben dem Mischalgorithmus mit der Bezeichnung *history merging* die Modellkonsistenz zu jedem Zeitpunkt zu gewährleisten. Dies hat jedoch auch zur Folge, dass sich Versionen teilweise nicht automatisch mischen lassen. Stattdessen muss ein Benutzer eventuell Aktionen per Hand wiederholen, da die aufgezeichneten Aktionen nicht mehr alle gültig sind. Das Auftreten solcher Situationen wird durch ein Sortierverfahren, das auf den Aktionen vor dem Mischen ausgeführt wird, reduziert. Der gravierendste Nachteil des Ansatzes ist jedoch, dass die Implementierung der abgesicherten Aktionen einen sehr großen Aufwand für den Applikationsentwickler bedeutet. Weiterhin erlaubt der Ansatz keine

[3] Die GMD (vor März 1995: Gesellschaft für Mathematik und Datenverarbeitung mbH; dann Forschungszentrum Informationstechnik GmbH) wurde in die FhG (Fraunhofer-Gesellschaft zur Förderung der angewandten Forschung e. V.) integriert.

[4] Dies bezieht sich auf kontextsensitive Semantik. Die domänenspezifische Semantik, wie sie der Benutzer erwartet, kann natürlich nicht garantiert werden.

temporär ungültigen Modelle, welche aber für ein intuitives Arbeiten am Modell teilweise nötig sind[5].

### 3.2.2 Ohst, Siegen

Dirk Ohst stellt in seiner Dissertation [19] an der Universität Siegen ein datenbankbasiertes Versionierungsverfahren für UML Modelle vor. Dieses Verfahren basiert auf einem Datenbanksystem und verwendet keine Replikation. Änderungen an dem Modell werden direkt auf alle Clients propagiert und sind nur für die Zeit kurzer Transaktionen isoliert. Es ist allerdings möglich auf Basis der Objektidentifizierer eine Differenz zwischen zwei Versionen zu bestimmen und diese in eine andere Version zu mischen. Realisiert wurde das Konzept teilweise in H-PCTE und die Darstellungen auch in FUJABA. Der Autor selbst gibt an, dass die meisten Werkzeuge jedoch nicht die Anforderungen erfüllen, um das Konzept einzusetzen. Der Integrationsaufwand wäre also für bestehende Werkzeuge sehr hoch. Der Aspekt der zentralen Datenhaltung kollidiert weiterhin mit der Anforderung an die hier vorliegende Arbeit, die die Trennung der Daten einzelner Benutzer fordert.

### 3.2.3 IPSEN, Aachen

Bernhard Westfechtel entwickelte eine Revisionskontrolle [31] für das IPSEN-Projekt [15] im Rahmen seiner Dissertation an der Technischen Hochschule Aachen. Diese baute auf der GRAS-Datenbank[6] auf und erweiterte sie um die Möglichkeit protokollierte Deltas an Objektstrukturen zu speichern. Die Protokollierung der Änderungen und die Nutzung von Deltas für Undo,

[5] Ausführungen hierzu finden sich etwa in „The IPSEN Aproach“ [15] in den Abschnitten 2.1.3 und 2.2.3 und in [28] in Abschnitt 8.4.

[6] Bei GRAS handelt es sich nicht um eine relationale Datenbank sondern um eine Datenbank zur Speicherung von Graphen. [11]

Redo und Versionskontrolle, sind Ähnlichkeiten der vorgestellten Revisionskontrolle zu den in der vorliegenden Arbeit vorgestellten Konzepten. Der Schwerpunkt der Arbeit aus Aachen liegt allerdings auf der Erkennung und Gewährleistung von sogennanter *externer Konsistenz*, womit die Modellkonsistenz zwischen verschiedenen Revisionen gemeint ist. Von der vorliegenden Arbeit unterscheidet sich die Dissertation von Herrn Westfechtel weiterhin dadurch dass der theoretischen Betrachtung für zum Beispiel das Mischen von Versionen ein anderes Graphmodell zu Grunde liegt. Es wird mit abstrakten Syntaxgraphen - Bezeichnern, sowie deren Bindungen und Listen von Bezeichnern - gearbeitet um Knoten und zwei Arten von Kanten zu modellieren. Dies liegt unter anderen daran, dass der Ansatz für allgemein strukturierte Dokumente - also auch solche mit Textanteilen - gültig ist und nicht wie die vorliegende Arbeit nur für Objektstrukturen. Die Basiskonzepte für Deltas und Undo/Redo sind jedoch auch auf Objektstrukturen (Graphen in GRAS) realisiert. Die formale Beschreibung der Modelle, der Konflikterkennung und des Mischens erfolgt nicht direkt auf graphentheoretischen Konzepten sondern mittels IPSEN-Schemata und -Transaktionen. Auch ist es nicht Ziel der Revisionskontrolle „offline“ Arbeiten zu ermöglichen, echtzeitnahe Anwendungen zu realisieren oder erweiterte Funktionen wie Copy & Paste zur Verfügung zu stellen. Dies ist unter Anderem auf die Realisierung in IPSEN mit der GRAS-Datenbank zurückzuführen. Alle verwalteten Daten liegen in ein und derselben Datenbank und Modelle werden immer als Ganzes zur Bearbeitung pessimistisch gesperrt.

### 3.2.4 XML-Dateien Mischen

In der Diplomarbeit [20] wurde versucht Versionen von XML-Dokumenten zu mischen, ohne deren eventuelle graphenbezogene Semantik auszunutzen. Die Erfahrung war jedoch, dass dies nicht universell möglich ist und dass sich

XML-Dateien nur sehr eingeschränkt als Artefakte in einer Versionsverwaltung eignen. Nutzt man allerdings aus, dass die Daten in den XML-Dateien einen Objektgraphen repräsentieren gelingt es dem SiDiff [10] Werkzeug brauchbare Differenzinformationen für zwei Versionen einer XML-Datei zu berechnen. Wie die Differenzen zu berechnen sind - insbesondere welche Attribute auf Ähnlichkeiten von Objekten schließen lassen - muss vom Entwickler festgelegt werden. Der Aufwand zur Nutzung von SiDiff ist daher abhängig von der Größe des Metamodells und für jede Anwendung erneut aufzuwenden. Mit den von diesem Werkzeug berechneten Differenzen sind auch Zwei- und Drei-Wege-Mischen möglich. Eine komplette Versionsverwaltung mit Mischmöglichkeiten gibt es jedoch auf Basis von SiDiff bisher nicht.

# 4 Konzept

Um das Konzept unabhängig von einer Programmiersprache beziehungsweise der Implementierung von Meta-Modellen der Anwendungen zu entwickeln, ist die Abstraktion von der konkreten objektorientierten Sprache unabdingbar. Diese Sicht auf die zu verarbeitenden Objektdaten soll nun mit Hilfe der Graphentheorie definiert werden. Die Begriffe *Objektstruktur* und *Modell* werden dabei als synonym betrachtet. Im folgenden werden zunächst *Modell* und *Gleichheit* von Modellen formal definiert. Folgen werden Definitionen zu Unterschieden zwischen Modellen und dem Begriff des *atomaren* Unterschieds, um schließlich auf die Änderungsprotokolle zu kommen.

## 4.1 Modell

### Definition Modell

Ein Modell sei ein gerichteter Graph G; Das Meta-Modell zum Modell sei das Schema S zum Graph G, so dass

| | |
|---|---|
| $G := (S, Ext)$ | Graph, mit |
| $S := (VL, EL, ef)$ | Schema, mit |
| $VL$ | endliche Menge von Knotennamen |
| $EL$ | endliche Menge von Kantennamen |
| $ef \subseteq VL \times EL \times VL$ | erlaubte Kantennamen für Knotennamenpaare |
| $PL \subseteq VL$ | Menge der „primitiven“ Knotennamen |
| $Ext := (V, E, vl)$ | Ausprägung, mit |
| $V$ | endliche Menge von Knoten |
| $vl : V \rightarrow VL$ | Abbildung von Knoten auf Knotennamen |
| $E \subseteq \{(v_1, el, v_2) \mid v_1, v_2 \in V \wedge el \in EL \wedge (vl(v_1), el, vl(v_2)) \in ef\}$ | endliche Menge von Kanten |

Diese Definition orientiert sich an üblichen Definitionen von Objektgraphen mit Schema. Es wurde allerdings eine Vereinfachung vorgenommen: Es wird nicht zwischen Attributen und Assoziationen (Kanten) unterschieden.

Ein attributierter Graph lässt sich jedoch leicht auch mit dieser Definition abbilden: Jeder Attributwert wird in einen Knoten umgewandelt und über eine Kante verknüpft. Auf Schemaseite wird schlicht die Menge der Attributnamen zu der Menge der Kantennamen hinzugefügt. Der Weg in die entgegengesetzte Richtung ist ebenso möglich, wenn über eine Untermenge der Kantennamen entschieden werden kann, ob es sich bei einem Namen um eine Assoziation oder ein Attribut handelt. Die Universalität des Ansatzes geht also durch diese Vereinfachung nicht verloren. Die Einführung der Menge $PL$ wird im nächsten Abschnitt motiviert.

Aus softwaretechnischer Sicht sind im Schema, beziehungsweise Metamodell, $VL$ die Menge von Klassen, $EL$ die Menge von Feldnamen (Rollennamen und Attributnamen) und $ef$ die Menge der Felder (Rollen und Attribute). Weiterhin entsprechen in der Ausprägung, beziehungsweise dem Modell, $V$ der Menge von Objekten, $vl$ der Zuordnung von Objekten zu Klassen und $E$ der Menge der Links (beziehungsweise Attributsbelegungen). Schließlich ist noch $PL$ die Menge der primitiven Datentypen (oder Klassen) deren Instanzen nur durch ihren Namen definiert sind und keine ausgehenden Kanten haben (wie zum Beispiel Zahlen oder Buchstaben).

### 4.1.1 Gleichheit

Um den Unterschied zwischen zwei Modellen definieren zu können, muss zunächst festgelegt werden, wann zwei Modelle $G_1$ und $G_2$ gleich sind. Dabei wird vorausgesetzt, dass beide Graphen dasselbe Schema S haben: $G_1 := (S, Ext_1), G_2 := (S, Ext_2)$. Die Handhabung unterschiedlicher Schemata wird erst im Abschnitt 4.3.3 bezüglich Schemaevolution behandelt. Zwei Modelle mit voneinander verschiedenen Schemata, die sich nicht auseinander entwickelt haben, zu vergleichen, ist nicht Teil dieser Arbeit, da es für die Anforderungen nicht relevant ist.

Der Vergleich von $Ext_1$ und $Ext_2$ genügt also, um die Gleichheit von $G_1$ und $G_2$ zu definieren. Auf Grund der einheitlichen Behandlung von Attributen und Kanten (s.o.) wurde zuvor die Menge $PL \subseteq VL$ von primitiven Klassen beziehungsweise Knotennamen eingeführt. Diese Menge enthält die Klassen (Knotennamen), deren Instanzen (Knoten mit diesem Namen) nicht explizit angelegt und zerstört werden. Ein Beispiel für eine solche Instanz ist ein Zahlenwert, somit wäre also beispielsweise Klasse „Integer“ in der Menge der Primitive. Die Existenz socher Primitive wird bei der Gleichheitsbetrachtung ignoriert.

Um die Gleichheitsdefinition kürzer zu fassen wird weiterhin die Menge $P(G) := \{v | v \in V \wedge vl(v) \in PL\}$ der primitiven Werte eingeführt. Dabei werden die Symbole ($G$, $V$, $vl$, $PL$) aus obiger Modelldefinition verwendet. Diese Menge der primitiven Werte wird verwendet, um bei Gleichheitsbetrachtung einmal verwendete Werte nicht explizit Löschen zu müssen, sondern die Existenz dieser Primitive beim Vergleich zu ignorieren.

### Definition Gleichheit

Zwei Modelle $G_1 := (S_1, (V_1, E_1, vl_1)), G_2 := (S_2, (V_2, E_2, vl_1))$ sind gleich, genau dann wenn

$$V_1 \setminus P(G_1) = V_2 \setminus P(G_2) \wedge E_1 = E_2$$

dabei wird vorrausgesetzt, dass

$$S_1 = S_2 \wedge vl := vl_1 \cup vl_2 \wedge \nexists (v, n_1) \in vl : \exists (v, n_2) \in vl : n_1 \neq n_2$$

Wie aus der Definition ersichtlich, wird also verlangt, dass die *gleichen* Modelle abgesehen von den Primitiven *dieselben* Knoten enthalten müssen. In der Praxis sind zwei Knoten (Objekte) dieselben, wenn sie den gleichen Identifizierer haben. Ohne Beschränkung der Allgemeinheit nehmen wir jedoch für die theoretische Betrachtung an, dass es sich um die selben Knoten

handelt. Die Kanten der Modelle müssen in Gegensatz zu den Knoten vollständig übereinstimmen, also auch die Kanten zu primitiven Knoten. Die Schemata müssen, wie bereits begründet gleich sein. Die Abbildung auf Knotennamen darf keine Knoten aus dem zweiten Modell auf einen anderen Namen als im ersten Modell abbilden - die selben Objekte haben immer den selben Typ.

Mit der Gleichheit wird für zwei Graphen $G_1$ und $G_2$ ein Gleichheitsoperator definiert als

$$G_1 = G_2 \Leftrightarrow G_1 \text{ und } G_2 \text{ sind gleich}$$

### 4.1.2 Unterschiede

Sind zwei Graphen nicht gleich gibt es ein nicht leeres Delta (Unterschied) zwischen den beiden Graphen:

#### Definition Delta

Für zwei Modelle $G_1 := (S, (V_1, E_1, vl))$ und $G_2 := (S, (V_2, E_2, vl))$ mit dem selben Schema $S$ und derselben Abbildung $vl$ sei das Delta $D(G_1, G_2)$ zwischen den Graphen definiert als

$$\begin{array}{lll} D(G_1, G_2) := (V_n, V_a, E_n, E_a) & \text{mit} & \\ \text{hinzugefügten Knoten} & V_n := V_2 \setminus V_1 & , \\ \text{entfernten Knoten} & V_a := V_1 \setminus V_2 & , \\ \text{hinzugefügten Kanten} & E_n := E_2 \setminus E_1 & \text{und} \\ \text{entfernten Kanten} & E_a := E_1 \setminus E_2 & . \end{array}$$

Zur Vereinfachung der folgenden Definitionen vereinbaren wir allgemein die Schreibweisen $\mathbb{G}$ :=*Menge aller Modelle* und $\mathbb{D}$ :=*Menge aller Deltas.*

Es werden nun zwei Operatoren definiert um Deltas auf Modelle anzuwenden:

$$+ : ((S,(V,E,vl),(V_n,V_a,E_n,E_a)) \in \mathbb{G} \times \mathbb{D} \mapsto \\ (S,((V \cup V_n) \setminus V_a,(E \cup E_n) \setminus E_a,vl)) \in \mathbb{G}$$

und

$$- : ((S,(V,E,vl),(V_n,V_a,E_n,E_a)) \in \mathbb{G} \times \mathbb{D} \mapsto \\ (S,((V \setminus V_n) \cup V_a,(E \setminus E_n) \cup E_a,vl)) \in \mathbb{G}$$

Wird also ein Delta auf ein Modell angewendet (+) werden die neuen Knoten und Kanten dem Modell hinzugefügt und die alten Knoten und Kanten aus dem Modell entfernt. Wird ein Delta rückwärts angewendet beziehungsweise „rückgängig gemacht“ werden die neuen Elemente entfernt und die alten hinzugefügt. Zu beachten ist, dass beide Operationen nicht kommutativ sind.

Aus der Definition ergibt sich sofort, dass für zwei beliebige Modelle mit demselben Schema und derselben Namensabbildung $G_1 := (S,(V_1,E_1,vl))$ und $G_2 := (S,(V_2,E_2,vl))$ gilt

$$G_1 + D(G_1,G_2) = G_2 \wedge G_2 - D(G_1,G_2) = G_1$$

Auch zwei Deltas können vereint werden durch

$$+_d : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$$

$$+_d : ((V_n1,V_a1,E_n1,E_a1),(V_n2,V_a2,E_n2,E_a2)) \\ \mapsto ((V_n1 \setminus V_a2) \cup V_n2,(V_a1 \setminus V_n2) \\ \cup V_a2,(E_n1 \setminus E_a2) \cup E_n2,(E_a1 \setminus E_n2) \cup E_a2)$$

Dabei ist zu beachten, dass auch dieser Operator nicht kommutativ ist, da sich die Definition aus der Annahme ergibt, dass zunächst das vorstehende Delta angewendet wird und dann das nachstehende. Da die verschiedenen Operatoren $+$ und $+_d$ durch Operandenart voneinander leicht zu unterscheiden sind, wird der Index $_d$ im Folgenden weggelassen. Aus der Definition folgt, dass für zwei Modelle $G_1$ und $G_2$ zwei Deltas $D_1$ und $D_2$ gilt

$$(G_1 + D_1) + D_2 = G_2 \Leftrightarrow G_1 + (D_1 + D_2) = G_2$$

Zwei Deltas nacheinander auf ein Modell anzuwenden hat also das gleiche Resultat wie die Deltas zunächst in der gleichen Reihenfolge zu vereinen und dann auf das Modell anzuwenden (Assoziativgesetz).

Um die Definitionen zunächst abzuschließen, wird nun noch der Begriff des *atomaren* Deltas eingeführt, das eine einzelne Änderung an einem Modell beschreibt:

#### Definition atomar

Ein Delta $D := (V_n, V_a, E_n, E_a)$ heißt *atomar* genau dann wenn

$$|V_n| + |V_a| + |E_n| + |E_a| = 1$$

In einem atomaren Delta wurde also entweder genau ein Knoten hinzugefügt, genau ein Knoten entfernt, genau eine Kante hinzugefügt oder aber genau eine Kante entfernt.

### 4.1.3 Änderungslisten

Um die zeitliche Abfolge von Änderungen repräsentieren zu können werden Listen von Deltas verwendet. Der Einfachkeit halber werden diese Deltas zunächst als atomar vorrausgesetzt. Die spätere Gruppierung von atomaren Deltas in Transaktionen reicht als Zusammenfassung von Änderungen aus

und ist flexibel einsetzbar, wie später gezeigt wird. Eine Liste von atomaren Deltas kann man auf zwei Arten erhalten:

Eine Möglichkeit ist die Berechnung des Deltas zwischen zwei gegebenen Modellen (sofern Vorraussetzungen, wie Identifizierbarkeit der Objekte, dafür erfüllt sind) und die Umwandlung in eine Liste von atomaren Änderungen. Auf Grund der einfachen Identifizierung der Knoten im Modell, ist die Deltaberechnung trivial. Das Delta kann dann, wie man leicht sieht, in atomare Delta aufgeteilt werden. Die Reihung in einer Liste kann in nahezu beliebiger Reihenfolge[1] passieren, da die chronologische Änderungsreihenfolge nicht mehr berechnet werden kann. Technische Besonderheiten der Reihenfolge werden später diskutiert, für die theoretische Betrachtung ist die Reihenfolge jedoch irrelevant.

Die zweite Möglichkeit eine Liste von atomaren Deltas zu erreichen ist, die tatsächlichen Änderungen am Modell zu protokollieren. Die daraus resultierende Liste ist - im Gegensatz zu der im Nachhinein berechneten Liste - chronologisch in der korrekten Reihenfolge, wenn die Deltas atomar erfasst werden. Modelle, die Änderungen auch nicht-atomar berichten (zum Beispiel für zwei Richtungen einer bidirektionalen Assoziation), verursachen eventuell Listen, die chronologisch nicht komplett korrekt sind. Die Konversion ist aber analog zu der ersten Möglichkeit noch immer möglich.

### Redundanz in Listen

Wie schon allein an der Reihenfolge leicht nachvollziehbar sind Änderungslisten, die ein Modell in ein anderes überführen, im Gegensatz zu Deltas nicht eindeutig. Es kann sogar redundante Information in einer Änderungsliste geben: Ein Knoten kann innerhalb derselben Liste hinzugefügt und

[1] Knoten müssen erzeugt werden bevor Kanten zu oder von ihnen gezogen werden können. Ebenso dürfen sie erst gelöscht werden, nachdem alle Kantenänderungen die sie involvieren in die Liste eingefügt wurden.

wieder entfernt werden. Vereinigt man diese beiden atomaren Deltas ist das resultierende Delta leer. Redundanz und fehlende Eindeutigkeit sind jedoch für die nachfolgend beschriebene Arbeitsweise nicht hinderlich sondern ergeben sich teilweise sogar aus ihr.

#### Definition Liste von atomaren Deltas

Eine Liste L von atomaren Deltas ist beschrieben durch

$L \in \mathbb{D}_1^n$ mit $\mathbb{D}_1 := \{D | D \in \mathbb{D} \wedge D \text{ atomar}\}$ Menge der atomaren Deltas und $n \in \mathbb{N}$ eine natürliche Zahl.

Auch eine Liste von Deltas kann auf ein Modell angewendet werden indem alle Deltas in gegebener Reihenfolge auf das Modell angewendet werden:

Für ein Modell $G$ und eine Liste von Deltas $L := (D_1, D_2, ...D_n)$ ist die Vereinigung definiert als

$$+_l : (G, L) \in \mathbb{G} \times \mathbb{L} \mapsto (((G + D_1) + D_2) + ...D_n) \in \mathbb{G}$$

mit $\mathbb{L}$ Menge der Listen von Deltas. Analog kann die Liste rückwärts angewandt beziehungsweise abgezogen werden:

$$-_l : (G, L) \in \mathbb{G} \times \mathbb{L} \mapsto (((G - D_n) - ...D_2) - D_1) \in \mathbb{G}$$

Zu beachten ist, dass die Deltas in umgekehrter Reihenfolge abgezogen werden. Auch bei den Operatoren auf Listen wird im Folgenden auf die Indizes zur Unterscheidung von den anderen Operatoren verzichtet.

## 4.2 Arbeitsweise

Mit einer Liste von Änderungen lässt sich nicht nur der Unterschied zwischen zwei Versionen eines Modells beschreiben, sondern auch ein komplettes Modell. Dies ist leicht einsichtig, wenn man als erste Version ein leeres Modell

(ohne Knoten und Kanten) annimmt. Die Änderungen, die an einem leeren Modell vorgenommen werden müssen, um ein konkretes Modell zu erhalten, beschreiben es vollständig.

Daher ist es also möglich ein Modell persistent zu speichern, wenn man eine Änderungsliste persistent speichern (beziehungsweise serialisieren) kann. Diesen Fakt macht sich die CoObRA Bibliothek für die Persistenzfunktionen zunutze. Das Laden eines Modells bedeutet also in CoObRA das Laden einer Änderungsliste und die Anwendung derselben auf ein leeres Modell. Darauf wie Änderungen und Änderungslisten serialisiert beziehungsweise gespeichert werden, wird erst in den Abschnitten 4.4 und 5 eingegangen.

#### Versionen von Dokumenten

Es soll vorweg der Begriff einer Version, wie er in dieser Arbeit verwendet wird, geklärt werden: Als Versionen eines Dokuments werden all die Modelle bezeichnet, die sich aus ein und demselben Ursprungsmodell durch Bearbeiten entwickelt haben. Es gibt mindestens ein Modell aus dem sich all diese Modelle beziehungsweise Versionen durch aufgezeichnete Änderungen erzeugen lassen. Für je zwei Modelle wird ein solches Modell als gemeinsame Ursprungsversion bezeichnet[2] Alle Versionen eines Dokuments haben dasselbe Schema (siehe Sonderfall Schemaevolution in Abschnitt 4.3.3).

### 4.2.1 Mischen und Synchronisation

Die Synchronisation zweier anfänglich gleicher Kopien eines Modells ist schnell auf Basis der Änderungen aus den obigen Definitionen beschrieben: Jede Änderung an einem der beiden Modelle wird aufgezeichnet und über einen geeigneten Kanal (zum Beispiel einen Netzwerkstrom) zum anderen

[2] Im Extremfall ist dies das leere Modell.

Modell transportiert. Jede einzelne Änderung kann auf das andere Modell dort wieder angewandt werden. Unter der Voraussetzung, dass keine nebenläufigen Änderungen stattfinden - sondern alle Änderungen an einem der Modelle auf das andere angewendet wurden, bevor das zweite selbst geändert wird - ist so die Synchronisation bereits komplett.

Als Beispielszenario soll nun ein Vertragsverwaltungssystem einer fiktiven Versicherungsgesellschaft eingeführt werden. Es verwaltet Kunden und Verträge als objektorientierte Daten. Zu Beginn nehmen wir an, die Gesellschaft hat zwei Berater A und B, die nun zum ersten Mal die Software starten.

Die Objektstruktur enthält zu Beginn keine Objekte und die beiden Betreuer initiieren den Synchronisationsmodus. Als der erste Kunde von Berater A in seinen Rechner eingetragen wird, sendet CoObRA die Änderungen an der Objektstruktur sofort an den zweiten Rechner und auch Berater B hat den neuen Kunden samt seinen Eigenschaften in seinen Daten zur Verfügung. Eine Kurzdarstellung der versandten Änderungen könnte etwa so aussehen:

```
- Knoten vom Typ „Kunde“ mit Identifizierer „1“ anlegen
- Kante „Nachname“ von Knoten „1“ zu Text-Primitiv „Schmidt“
ziehen
```

Formal gesehen sieht ein Synchronisationsschritt so aus, dass ein Modell $G_1$ verändert wird zu $G_1'$. Ein Modell $G_2$, das gleich einem $G_1$ ist ($G_1 = G_2$), existiert ebenfalls. Ein Delta $D$ wird dann empfangen und auf $G_2$ angewendet; man erhält $G_2' := G_2 + D$. Ist $D = D(G_1, G_1')$ folgt sofort $G_2'$ ist wieder gleich $G_1'$. Werden also die Änderungen vollständig erfasst und korrekt übertragen sind die Modelle nach jedem Schritt erfolgreich synchronisiert. Sie enthalten alle Änderungen, die an den einzelnen Dokumenten durchgeführt wurden.

### Mischen nach nebenläufigen Änderungen

Können oder sollen zwei Modelle nicht immer sofort synchronisiert werden, entstehen irgendwann nebenläufige Änderungen an den Modellen. Das bedeutet, beide Modelle sind bezüglich der gemeinsamen Ursprungsversion geändert. Die Modelle müssen *gemischt* werden.

In dem Beispielszenario mit dem Vertragsverwaltungssystem könnte eine solche Situation entstehen, wenn Berater A sich in den Außendienst begibt. Er trennt die Verbindung mit Berater B und reist zum Kunden. Dort korrigiert er den Namen des Kunden von „Schmidt“ auf „Schmid“ und trägt einen neuen Vertrag ein. Während dessen könnte Berater B seinerseits ein neues Kundenobjekt anlegen.

Kehrt Berater A nun in die Firma zurück, sollen die Datenbestände erneut synchronisiert werden. Beide Rechner haben jedoch eine Änderungsliste, die nun folgerdermaßen aussehen würden:

```
Berater A:
- Kante „Nachname“ von Knoten „1“ zu Text-Primitiv „Schmidt“
löschen
- Kante „Nachname“ von Knoten „1“ zu Text-Primitiv „Schmid“
ziehen
Berater B:
- Knoten vom Typ „Kunde“ mit Identifizierer „2“ anlegen
- Kante „Nachname“ von Knoten „2“ zu Text-Primitiv „Meier“
ziehen
```

Das Mischen der Versionen kann realisiert werden, indem alle Änderungen auf die gemeinsame Ursprungsversion angewendet werden. Die Reihenfolge der Änderungen ist hier allerdings nicht eindeutig. Da die Operationen zum Anwenden der Änderungen jedoch nicht kommutativ sind, spielt die Rei-

henfolge eine Rolle. Die Reihenfolge der Änderungen an einem Modell ist durch ihr zeitliches Auftreten gegeben und muss auch beibehalten werden, sofern Redundanzen vorhanden sind (s.o., Hinzufügen und Entfernen desselben Knotens beziehungsweise derselben Kante). Abgesehen von dieser zeitlichen Reihenfolge innerhalb der einzelnen Listen werden die Änderungslisten von verschiedenen Modellen zunächst im Beispiel des nächsten Abschnitts in beiden möglichen Reihenfolgen angewendet, um die Konsequenzen zu verdeutlichen.

Wendet man die oben aufgeführten Beispieländerungen nun auf die gemeinsame Ursprungsversion an, erhält man eine neue gemeinsame Version. Die sowohl den geänderten Nachnamen enthält, als auch den neuen Kunden. Hier führen die unterschiedlichen möglichen Reihenfolgen der Listen bei der Anwendung noch zu den gleichen Ergebnissen.

#### Konflikte

Ändern nun jedoch beide Berater den Namen „Meier“, gibt es einen Konflikt: Berater A ändert den Namen auf „Maier“ und Berater B ändert ihn, während er von Berater A getrennt ist, auf „Mayer“. Wird nun erneut synchronisiert gibt es mehrere möglichkeiten für neue gemeinsame Modelle. Werden zuerst die Änderungen von Berater A angewendet wird der Attributwert zuletzt auf „Mayer“ gesetzt. Anders herum wäre der letzte Attributwert „Maier“. Verhielte sich das Modell exakt nach den formalen Definitionen, enthielte es sogar beide Attributwerte im Nachnamen. Zusätzlich gibt es zwei Änderungen, die beide den Wert „Meier“ löschen, während er natürlich nur ein Mal tatsächlich gelöscht wird.

Solche syntaktischen[3] *Konflikte*, wie im Beispiel, gilt es zu erkennen und geeignet zu behandeln. Es wird daher immer zunächst eine Änderungsliste aus einem Modell auf die gemeinsame Ursprungsversion angewendet. Danach werden Konflikte zwischen den Änderungslisten erkannt und vor dem Anwenden der zweiten Liste behandelt.

Als syntaktische Konflikte werden hier die Änderungen bezeichnet, die sich auf die Erzeugung oder das Löschen desselben Knotens beziehen, oder die eine gleich benannte Kante von demselben Knoten aus ziehen oder löschen. Weiterhin liegt ein Konflikt vor, wenn ein Knoten gelöscht wurde, der in einem anderen Delta neue Kanten erhalten hat. Formal lässt sich dies wie folgt formulieren:

### Definition syntaktischer Konflikt

Für zwei Mengen von atomaren Deltas $L_1$ und $L_2$ stehen all die Deltas $D := (V_n, V_a, E_n, E_a) \in L_2$ mit einem Delta aus $L_1$ in Konflikt für die gilt

$$\begin{aligned} \exists D' = (V'_n, V'_a, E'_n, E'_a) \in L_1 : \quad & V_n \cup V_a = V'_n \cup V'_a \\ & \vee E_n \cup E_a \approx E'_n \cup E'_a \\ & \vee \exists v \in V_a \cup V'_a : \exists (v_1, el, v_2) \in E_n \cup E'_n \\ \text{mit } v_1 = v \vee v_2 = v & \end{aligned}$$

mit der Relation $\approx$ auf zwei Kantenmengen (mit maximal einem Element, da die Änderungen atomar sind)

$$\begin{aligned} (E, E') \in \approx \quad \Leftrightarrow \quad & \forall e = (v_1, el, v_2) \in E : \\ & \forall e' = (v'_1, el', v'_2) \in E' : el = el' \wedge v_1 = v'_1 \end{aligned}$$

[3] Als syntaktische werden diese Konflikte hier bezeichneit, um auszudrücken, dass sie aus der Definition von Änderungen abzuleiten sind und dass diese Änderungen ohne Konfliktbehebung eventuell zu einem syntaktisch inkorrekten Modell führen.

Mit dieser Definition werden alle potentiellen syntaktischen Konflikte erkannt. Die Relation kann bei Bedarf noch eingeschränkt werden, so dass Mehrfachänderungen an zu-n Kanten erlaubt werden. Es genügt hierzu bestimmte Kantennamen aus der $\approx$ Relation komplett auszuschließen. Dies ist so in der späteren Implementierung geschehen.

Neben den syntaktischen Konflikten gibt es noch eine weitere Konfliktart, die mit *semantische Konflikte* bezeichnet werden soll. Diese Konflikte ergeben sich nur aus der Bedeutung des Modells für die jeweilige Applikation. Sie werden nicht von der Bibliothek erkannt. Jede Applikation muss eigene Mechanismen zur Verfügung stellen, um solche Konflikte nach oder während eines Mischvorgangs zu erkennen und zu beheben.

### Konfliktbehebung

Bevor eine Menge (beziehungsweise Liste) von Änderungen in einem Mischvorgang angewandt werden kann, muss sie frei von syntaktischen Konflikten mit bereits angewandten Änderungen sein. Zunächst können dazu doppelte Änderungen entfernt werden. Wird derselbe Knoten oder dieselbe Kante in beiden Änderungsmengen gelöscht, kann eine der Änderungen verworfen werden. Dasselbe gilt für zwei Änderungen, die dieselbe Kante anlegen. Werden jedoch zwei unterschiedliche Kanten angelegt, die in Konflikt stehen, muss eine Entscheidung getroffen werden, welche der beiden Änderungen bestehen bleiben soll. Hier ist üblicherweise der Benutzer zu fragen, oder eine einfache Regel (zum Beispiel Auswählen der chronologisch ersten Änderung) anzuwenden.

Für die erfolgreiche Synchronisation ist es dabei jedoch notwendig, dieselbe Liste von Änderungen auf alle Kopien des Modells anzuwenden. Strategien um dies zu gewährleisten werden im Folgenden diskutiert.

### 4.2.2 Synchronisationsstrategien

Die einfachste Situation ist, die Synchronisation von genau zwei Kopien eines Dokuments. Diese wird hier zunächst kurz diskutiert.

Für jede Synchronisationstrategie nach obigem Konzept muss eine gemeinsame Ursprungsversion $UV$ bekannt sein. In dem Fall von zwei Kopien muss genau eine Version $UV_12$ neben den aktuellen Modellen $V_1$ und $V_2$ aufbewahrt werden. Bei jeden Synchronisationsvorgang werden dann die Änderungen $D_1$ an der einen Kopie $V_1$ zu der gemeinsamen Version $UV_12$ auf die andere Kopie $V_2$ angewendet, es entsteht $V_2'$. Es werden währenddessen Konflikte der dortigen Änderungen $D_2$ zu den einzuspielenden Änderungen $D_1$ festgestellt und behoben. Nach der Synchronisation haben beide Kopien wieder dasselbe Modell und nehmen dieses als gemeinsame Version für den nächsten Synchronisationsvorgang.

#### Peer-to-Peer

Diese Synchronisationstrategie ist ein einfaches *Peer-to-Peer* Szenario. In der Praxis ist die Anzahl an Peers jedoch nicht festgelegt und nur in seltenen Fällen genau zwei. Ist die Anzahl an Peers und damit die Anzahl an Kopien jedoch höher beziehungsweise unbekannt, reicht das Vorhalten einer einzigen gemeinsamen Ursprungsversion nicht aus. Es muss für jeden Peer die Version vorgehalten werden, die zuletzt mit ihm synchronisiert wurde (siehe auch Abbildung 4.2). Da diese Versionen auch von einem anderem Peer zu einem dritten weitergegeben werden kann, muss die Version auch nach der Synchronisation mit dem betreffenden Peer weiter gespeichert bleiben. Volle Peer-to-Peer-Flexibilität setzt also ein Vorhalten der kompletten Versionshistorie in Form von Änderungslisten voraus. Komplette Redundanzentfernung, etwa um Speicherplatz zu sparen, ist so nicht möglich. Es kann nur die Redundanz aus den Änderungslisten zwischen den zahlreichen

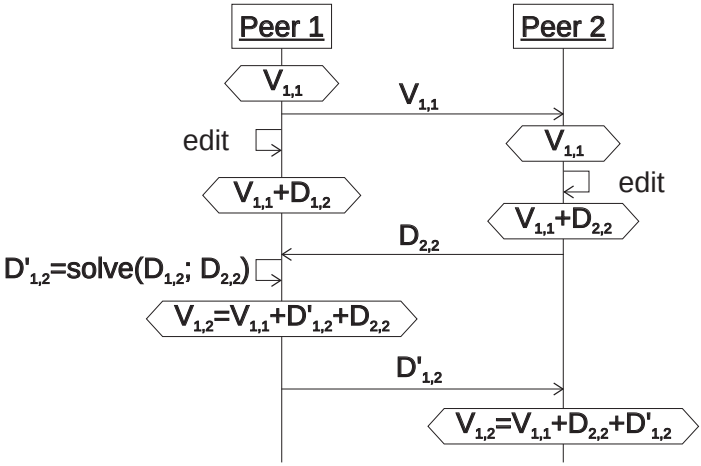


Abbildung 4.1: *Darstellung der Synchronisation zweier Peers: Die Version $V_1$ wird an Peer 2 übertragen. Beide Peers ändern ihre Version und haben nun ein Delta ($D_{1,2}$ und $D_{2,2}$) zur gemeinsamen Version $V_1$. Peer 2 übertragt daraufhin seine Änderungen $D_{2,2}$ an Peer 1. Dieser löst Konflikte mit seinen eigenen Änderungen und wendet das emfangene Delta an. Er erhält so eine neue Version $V_2$. Nun sendet Peer 1 seine konfliktbereinigten Änderungen an Peer 2, die dieser direkt anwenden kann. Beide Peers haben nun wieder eine gemeinsame Version $V_2$.*

Versionen entfernt werden.

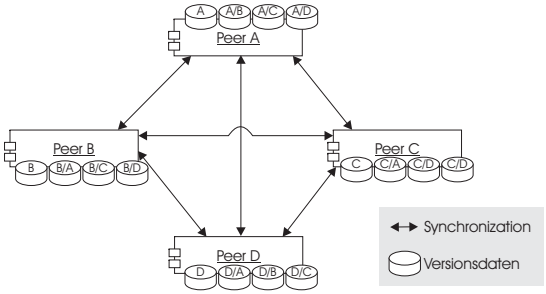


Abbildung 4.2: *Peer-zu-Peer-Synchronisation: Jeder Peer muss eine gemeinsame Ursprungsversion für jeden Kommunikationspartner speichern (entgegen der Darstellung sind die Versionsdaten jedoch nicht disjunkt).*

### Client-Server

Um diesem erhöhten Datenvolumen für jede Kopie vorzubeugen, kann eine *Client-Server-Strategie* genutzt werden. Hier hält ein zentraler Server die zur Synchronisation nötige Versionshistorie bereit. Die Clients halten alle nur eine zusätzliche mit dem Server gemeinsame Version vor. Diese Strategie bedingt allerdings auch, dass alle Änderungen, die einmal auf dem Server gespeichert wurden, nicht im Zuge der Konfliktlösung wieder verworfen werden können. Das liegt daran, dass es ja bereits andere Clients gibt (geben kann), die sich darauf verlassen eine gemeinsame Version mit dem Server zu haben. Ändert sich die Version auf dem Server jedoch im Nachhinein, ist

die nächste Synchronisierung eventuell nicht syntaktisch korrekt[4]. Die Konflikte müssen also anhand von Modifikationen an der Änderungsliste eines Clients gelöst werden.

Die Konfliktlösung auf Seiten der Clients ist nicht nur notwendig sondern auch sinnvoll bezüglich der Nutzerinteraktion. Die Anwendung kann mit eventuellen Rückfragen an den Benutzer die Konflikte lokal lösen um aktuelle Änderungen, die vom Server empfangen wurden, in die Kopie des Clients einzuarbeiten. Dieser Prozess soll analog zu dem Prozess bei üblichen Versionsverwaltungssystemen *Update* genannt werden.

Berater B im Beispielszenario der Vertragsverwaltung kann sich im Client-Server-Betrieb also, nachdem er im Außerndienst war, im Büro ein Update vom Server laden. Dabei könnten Ihm Fragen der folgenden Art vom Programm gestellt werden:

- Der Nachname des Kunden „Meier“ wurde von Berater A auf „Maier“ geändert. Sie haben den Nachnamen auf „Mayer“ geändert. Welcher Wert soll übernommen werden?
- Berater A hat den Vertrag „XY“ bearbeitet. Sie haben diesen Vertrag gelöscht. Soll der Vertrag wirklich gelöscht werden?
- Sowohl Sie als auch Berater A haben für den Kunden „Z“ einen neuen Vertrag eingepflegt. Wollen Sie beide Verträge übernehmen?

Alle Informationen, die für solche Rückfragen zur Konfliktlösung notwendig sind ergeben sich aus den Deltas mit Konflikt. Abhängig von den Antwor-

[4] Andere Clients gehen davon aus, dass sich die Serverversionen, die sie empfangen haben, nicht mehr verändern, um das übertrage Datenvolumen gering zu halten. Werden die Versionen auf dem Server trotzdem im Nachhinein auf dem Server verändert, kann es zum Beispiel sein, dass Objekte nicht mehr erzeugt werden, bei denen ein Client aber davon ausgeht, dass sie existieren, und beispielsweise Links zu ihnen erzeugt.

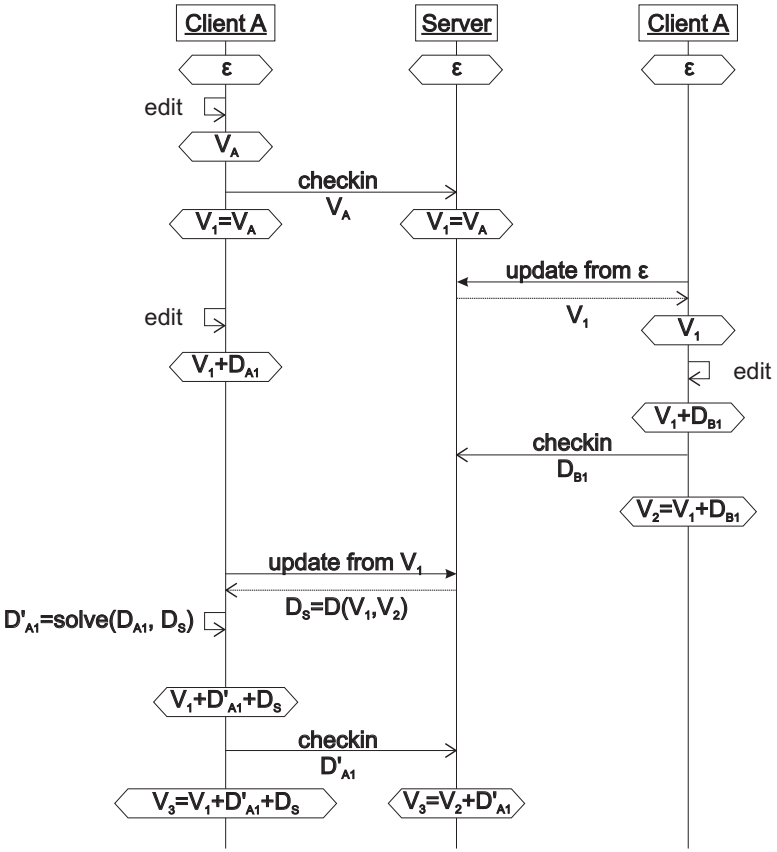


Abbildung 4.3: *Client-Server-Synchronisation: Client A, Client B und der Server starten mit einem leeren Repository. Client A legt eine erste Version an und sendet diese an den Server. Client B lädt sie vom Server herunter. Nebenläufig führen nun Client A und Client B Änderungen durch. Client B sendet seine Änderungen als erster an den Server. Client A muss daher beim Herunterladen der Änderungen Konflikte mit seinen lokalen Änderungen lösen. Danach sendet er seinerseits die nun konfliktfreien Änderungen an den Server.*

ten des Benutzers wird nun die Änderungsliste und das Modell des Clients geändert.

Nachdem ein Client die erste Hälfte der Synchronisation (Update) abgeschlossen hat, hat er immer noch ein Delta zur Version des Servers. Es gibt jedoch keine konfliktbehafteten Deltas mehr. Nun kann die zweite Phase der Synchronisation durchgeführt werden, der sogenannte *Checkin*. Hierbei werden die Änderungen des Clients, die das Delta zur Serverversion ausmachen, zum Server übertragen und dort gespeichert. Dadurch entsteht auf dem Server eine neue Version des Dokuments, die der Version des Clients entspricht.

### Hierarchisch

Im Gegensatz zum Peer-to-Peer Betrieb kann man sich bei der Client-Server-Strategie nicht entscheiden, mit welchen anderen Nutzern man sich synchronisiert. Das Arbeiten in Gruppen kann dadurch behindert werden. Abhilfe kann hier neben dem Peer-to-Peer Betrieb und dem damit verbundenen zusätzlichen Aufwand auch die hierachische Organisation der Server schaffen: Eine *hierarchische Client-Server-Strategie* lässt sich etablieren, indem jeder Server wieder Client eines übergeordneten Server ist. So kann zum Beispiel jede Arbeitsgruppe einen gemeinsamen Server haben, während die Arbeitsgruppen-Server sich wieder auf Benutzerwunsch hin über einen projektweiten Server synchronisieren (siehe Abbildung 4.4).

Da jeder Client auch bei der hierarchischen Strategie nur mit seinem zuständigen Server kommuniziert, brauchen die Clients wie gehabt nur *eine* gemeinsame zusätzliche Version des Dokuments vorhalten. Holt ein Server eine neue Version vom übergeordneten Server ist es aus Client-Sicht, als habe ein anderer Client eine neue Version eingespielt. Für die Kommuni-

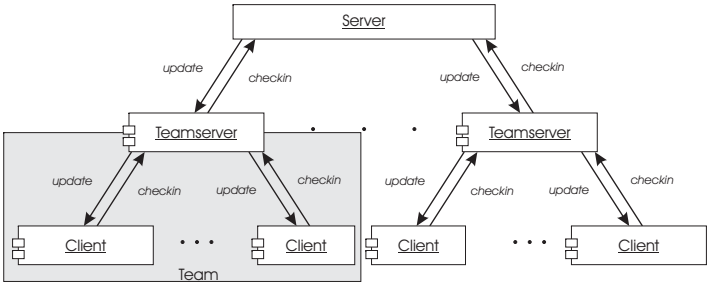


Abbildung 4.4: *Beispiel für die hierarchische Organisation von Servern*

kation zwischen Client und Server ändert sich also nichts. Jeder Server hat nun allerdings zwei Möglichkeiten eine neue Version zu erhalten: eine wie gehabt durch die Clients, die ein Checkin durchführen und die neue Möglichkeit selbst ein Update mit dem übergeordneten Server durchzuführen. Das Server-Update unterliegt hier den gleichen Anforderungen wie das Client-Update; es müssen Konflikte erkannt und behoben werden. Weiterhin können nun auch die Server ein Checkin durchführen, um dem übergeordneten Server eine neue Version des Dokuments zuzusenden.

### Echtzeitnah

Wie bereits in den Anforderungen dargestellt hat das Anwendungsgebiet der Echtzeitsimulation und der Spiele einen anderen Bedarf an Synchronisationseigenschaften. Hier treten in kurzen Zeitabständen regelmäßig Änderungen am Modell auf, daher ist teilweise keine der bisher beschriebenen Synchronisationsstrategien praktikabel. Insbesondere wenn es weiterhin Latenzen und knappe Bandbreite auf dem Kommunikationsmedium gibt, ist die sichere, konsistente Synchronisation nicht mehr - wie eingangs in den Anforderungen beschrieben - die wichtigste Funktion. Ist es dagegen wichti-

ger zu jedem Zeitpunkt die Modelle möglichst ähnlich zu halten, bietet sich eine rigorosere Synchronisationsstrategie an: Jede Änderung wird einfach sofort an alle zu synchronisierenden Kopien gesendet und dort angewandt.

Durch etwaige Latenzen gibt es auch bei dieser Art der Synchronisation nebenläufige Änderungen und damit Konflikte. Diese zu erkennen ist jedoch ungleich schwerer als bei den anderen Strategien, da die Änderungslisten, die miteinander in Konflikt stehen, nicht zur selben Zeit vorliegen. Sollen Konflikte zuverlässig erkannt werden, ist es notwendig festzustellen, ob die einzelnen atomaren Deltas entstanden sind, bevor oder nachdem ein potentiell in Konflikt stehendes Delta angewandt wurde. Dies lässt sich durch einen verteilten Zeitbegriff oder durch Sequenznummern realisieren, ist aber sehr aufwändig. Eine andere Möglichkeit ist, mit nicht erkannten Konflikten umzugehen, indem andere Methoden der Konsistenzerhaltung (üblicherweise bereits im Modell integriert) genutzt werden, um potentiell in Konflikt stehende Knoten erneut zu synchronisieren. Näheres zur Realisierung der Strategie in Abschnitt 6.5.

### 4.2.3 Garantien bei der Synchronisation

Beim Mischen von konfliktfreien oder konfliktbereinigten Versionen ist die syntaktische Korrektheit durch die wohldefinierten Operationen auf dem Modell gewährleistet. Die semantische Korrektheit kann jedoch im Allgemeinen nicht garantiert werden. Die sematische Korrektheit eines Modells kann nach beliebig komplexen Bedingungen entschieden werden und es ist im Einzelfall zu entscheiden, wie sie sichergestellt oder wieder hergestellt werden soll. Einige Eigenschaften, die für die Konsitenzerhaltung eines Modells hilfreich sein können, sollen hier jedoch diskutiert werden.

### Gemeinsame erzeugte Objekte

Werden bestimmte Gruppen von Objekten immer gemeinsam angelegt und auch gemeinsam wieder gelöscht kann man einige Eigenschaften des Modells auch nach der Synchronisation sicherstellen. Für jedes Objekt in einer Gruppe gemeinsam angelegter Objekte gilt zum Beispiel, dass es nur dann im Modell existiert, wenn auch die anderen Objekte der Gruppe existieren. Diese Invariante ist leicht nachzuvollziehen, wenn einerseits die Gruppe feststeht und andererseits alle modellverändernden Aktionen immer alle oder keine Objekte einer Gruppe anlegen oder löschen. Zu den potentiell modellverändernden Aktionen kommt nun das Synchronisieren hinzu. Es gilt daher zunächst zu zeigen, dass Objekte, die in einer der Kopien gemeinsam angelegt wurden auch in der resultierenden Version komplett vorhanden sind. Im zweiten Schritt ist zu zeigen, dass Objekte, die in einer der Kopien gemeinsam gelöscht wurden im Resultat ebenfalls komplett gelöscht sind:

Wenn alle konfliktfreien Änderungen aus der Kopie mit den neu angelegten Objekten übernommen werden - objekterzeugende Änderungen sind stets frei von syntaktischen Konflikten - sind auch alle gemeinsam erzeugten Objekte stets in einer gemischten Version vorhanden. Das gemeinsame Erzeugen der Objekte erscheint also unkritisch. Das Löschen der Objekte aus einer solchen Gruppe, kann jedoch zu Konflikten mit einer zu mischenden Kopie führen, denn dieselben Objekte könnten in der anderen Kopie bearbeitet worden sein (die Knoten könnten neue Kanten bekommen haben).

Werden individuell Änderungen während des Mischens von der Applikation gefiltert[5] oder treten Konflikte in den genannten Änderungen auf, gilt die Invariante genau dann noch, wenn die erzeugenden Änderungen alle

[5] In den ergänzenden Anforderungen in Abschnitt 2.2 wurde gefordert Änderungsdaten bei Bedarf filtern zu können. Das heißt für einzelne Änderungen kann entschieden werden diese nicht zu laden beziehungsweise zu empfangen.

nicht oder alle komplett gefiltert beziehungsweise abgelehnt werden. Dies kann applikationsseitig zum Beispiel dadurch sichergestellt werden, dass zur Konfliktlösung nur die normalen Anwendungsaktionen zur Verfügung stehen. Eine weitere Möglichkeit ist, die Applikationsaktionen in Transaktionen zu verpacken und nur komplette Transaktionen abzulehnen oder zu akzeptieren.

Als kurzes Fazit lässt sich ziehen, dass Invarianten auf Modellen durchaus auch bei der Synchronisation bewiesen werden können. Es gibt jedoch auch Modelleigenschaften, die komplexere Beweisführungen benötigen würden. So ist es zum Beispiel schwerer sicherzustellen, dass ein bestimmtes Objekt niemals zwei Nachbarn mit bestimmten Attributwerten hat. Wird nämlich in jeder von zwei Kopien jeweils ein Nachbar mit entsprechenden Werten angelegt und die Kopien danach zu einer neuen Version gemischt, entstehen keine syntaktischen Konflikte. Dennoch ist die Invariante verletzt. Die Applikation muss den semantischen Konflikt erkennen und entsprechend nachträglich beheben oder im Voraus die Änderungen filtern. Hier unterscheidet sich der präsentierte Ansatz wesentlich von dem bereits erwähnten CoAct (s. Abschnitt 3.2.1). Die dargestellte Zusicherung für das gemeinsame Erzeugen von Objekten wird in Abschnitt 4.3.2 verwendet, um die syntaktische Korrektheit von den dort eingeführten geordneten Assoziationen sicherzustellen.

### 4.2.4 Transaktionen

Der Transaktionsbegriff in dieser Arbeit wurde schon in den Anforderungen umrissen. Änderungen sollen in Transaktionen gruppiert werden können. So ist es etwa möglich die soeben besprochenen Änderungen zum Erzeugen einer Objektgruppe in eine Transaktion zu fassen und diese nur vollständig anzuwenden oder komplett zu verwerfen. Weiterhin können Transaktionen

auch wieder Transaktionen enthalten um diese schachteln zu können. Diese Transaktionen werden dann, wie von anderen Anwendungen (zum Beispiel Datenbanken) gewohnt, im Bedarfsfall zurückgenommen. Dies wird in Applikationen genutzt um zum Beispiel Benutzeraktionen in Transaktionen zu verpacken und im Fehlerfall die ganze Aktion rückgängig zu machen, um wieder einen definierten Modellzustand zu erreichen. Die Transaktionen auf unterschiedlichen Replikaten sind - wie zu erwarten - voneinander isoliert. Aber wie bereits in den Kernanforderungen dargestellt, ist die Isolation von Transaktionen innerhalb eines Modells nicht gefordert. Dies wäre auch nicht mit der Anforderung direkt auf dem Java-Modell zu arbeiten - also gemeinsamen Speicher für die Transaktionen zu verwenden - vereinbar.

In der Theorie lassen sich die Transaktionen als Listen von Änderungen und Transaktionen darstellen. Für die Operationen mit Transaktionen lassen sich diese zu einer Liste von Änderungen plätten: Jede Transaktion in einer Transaktion wird ersetzt durch die Folge ihrer Elemente. Führt man dies rekursiv durch, bleibt am Ende eine Liste von Änderungen. Diese kann nun, wie gewohnt, auf ein Modell angewendet werden.

Die praktische Relevanz der Transaktionen drückt sich hauptsächlich dadurch aus, dass sich durch den Abschluss einer Transaktion Zeitpunkte festlegen lassen, an denen ein Modell ein gewisses Konsistenzkriterium erfüllt. Aus diesem Grund wird eine Transaktion oft mit einer Applikationsaktion gestartet und nach erfolgreichem Ende der Aktion beendet, beziehungsweise bei Fehler zurückgenommen. Auch für eine „Rückgängig“-Funktion in der Applikation bietet sich ein solches Vorgehen an, da es so möglich ist immer komplette Aktionen rückgängig zu machen und so den Konsistenzgrad, den die Anwendungsaktionen garantieren, auch bei „Rückgängig“ und „Wiederholen“ zu gewährleisten.

### 4.2.5 Konfliktlösungsstrategien

Je nach Anwendung können verschiedene Strategien zum Lösen von Konflikten sinnvoll sein. Insbesondere können bei geeigneter Strategie teilweise Modelleigenschaften über die syntaktische Korrektheit hinaus gewährleistet werden. Um die Betrachtung zu vereinfachen wird im Folgenden von einem *eigenen* Dokument und einem *fremden* Dokument ausgegangen, die eine gemeinsame Ursprungsversion haben und gemischt werden sollen. Dabei sollen keine Änderungen aus der fremden Version verworfen werden, um auf alle vorgestellten Synchronisationsszenarien zuzutreffen.

Die wohl am einfachsten umzusetzende Strategie ist es, alle fremden Änderungen in die neue Version des eigenen Dokuments zu übernehmen und von den eigenen (atomaren) Änderungen diejenigen zu verwerfen, die mit den fremden in Konflikt stehen. Dadurch können nach der Definition der Konflikte keine neuen Konflikte erzeugt werden, da nur Änderungen verworfen werden, aber keine hinzukommen. Danach können also alle eigenen Änderungen angewandt werden und das Mischen ist abgeschlossen. Der Benutzer sollte dann über verworfene Änderungen informiert werden.

Eine weitere Möglichkeit Konflikte zu lösen, die mehr Kontrolle über die semantische Korrektheit suggeriert, ist Benutzeraktionen (beziehungsweise Anwendungsoperationen) in Transaktionen zu fassen und nur komplette Transaktionen zu verwerfen, wenn ein Konflikt erkannt wird. Dies kann aber einerseits noch keine semantische Korrektheit garantieren und wirft zum anderen weitere Probleme auf: Es ist in den meisten Anwendungen nicht sichergestellt, dass eine Aktion ausgeführt werden kann oder darf, wenn eine der vorherigen Aktionen nachträglich verworfen wurde. Zusätzlich können durch das Verwerfen von konfliktfreien Änderungen spätere eigene Änderungen unmöglich gemacht werden. So könnte zum Beispiel eine Änderung, die ein neues Objekt erzeugt, verworfen werden, während dieses Objekt in

einer weiteren Aktion verändert wurde. Diese Art des Vorgehens stellt also recht hohe Anforderungen an die Anwendung, die auch auf diese Sonderfälle reagieren muss.

Wie später noch in Kapitel 6 dargestellt, hat sich der Einsatz von Transaktionen beim Mischen von Modellen nicht bewährt. Ein direktes Mischen der atomaren Änderungen lieferte bereits den Anforderungen genügende Ergebnisse. Es lässt sich jedoch eine kombinierte Strategie implementieren: Transaktionen, die keinen Konflikt enthalten und deren Vorgänger kein Konflikt enthält, können oft sofort angewendet werden. Erst ab dem ersten Konflikt könnten dann die erweiterten Strategien verwendet werden. Allerdings lässt sich so die vollständige kontextsensitive Korrektheit des Modells noch nicht sicherstellen. Modellbedingungen die sich auf die Existenz bestimmer Objekte beziehen, können leicht verletzt werden, wenn zum Beispiel in beiden zu mischenden Replikaten ein Objekt derselben Art eingefügt wird.

In einem UML-Modell beispielsweise darf es keine zwei gleichnamigen Klassen geben (vollqualifiziert). Legen zwei Benutzer in einem CASE-Tool, das CoObRA verwendet, getrennt voneinander je eine neue Klasse an und benennen diese gleich, sind dies für die Bibliothek unterschiedliche Objekte. Nach dem Mischen der Versionen gibt es also ein Modell mit zwei Klassen gleichen Namens. Da es hier keinen syntaktischen Konflikt nach den theoretischen Konzepten gibt, kann diese kontextsensitive Inkonsistenz beim Mischen nicht vermieden werden, obwohl alle Aktionen der Anwendung sicherstellen, dass keine zwei Klassen mit gleichem Namen angelegt werden.

Dennoch liefert der Einsatz von Transaktionen für manche Anwendungen eventuell auch einen weiteren Kompromiss zwischen Komplexität und Korrektheit für das Mischen.

Eine Alternative zum Verwerfen von Änderungen zur Konfliktlösung ist,

das eigene Dokument vor dem Mischen mit Applikationsaktionen zu bearbeiten und Redundanzen zu entfernen (siehe folgender Abschnitt 4.2.6), so dass keine Konflikte mehr bestehen. Die Anwendung könnte dazu zum Beispiel Elemente markieren, deren Änderungen Konflikte erzeugen würden. Es wäre dann an dem Benutzer, geeignet Abhilfe zu schaffen:

Als Beispiel wird der Konflikt aus einem früheren Szenario betrachtet: Beide Benutzer der Vertragsverwaltung änderten den Namen „Meier“, dies führte zum Konflikt: Berater A änderte den Namen auf „Maier“ und Berater B änderte ihn, während er von Berater A getrennt war, auf „Mayer“. Soll nun der Konflikt von Berater A gelöst werden, kann dieser den Namen erneut auf „Meier“ ändern. Sein lokales Änderungsprotokoll enthält dann vier atomare Änderungen bezüglich des Namens:

1. Entfernen des Wertes „Meier“ aus dem Feld „name“ des Kundenobjekts
2. Hinzufügen des Wertes „Maier“ in das Feld
3. Entfernen des Wertes „Maier“ aus dem Feld
4. Hinzufügen des Wertes „Meier“ in das Feld

Alle diese Änderungen stehen in Konflikt mit der von Berater B vorgenommenen Namensänderung. Wendet Berater A nun die Redundanzentfernung an, werden alle vier atomaren Änderungen verworfen. Daher steht nach der Redundanzentfernung keine Änderung mehr in Konflikt mit den Änderungen von Berater B; es kann konfliktfrei gemischt werden.

Mit etwas mehr Aufwand könnte man versuchen von der Applikation automatisch nötige Aktionen ableiten zu lassen. Wurde das eigene Dokument bearbeitet können redundante Änderungen aufgehoben werden (zum Beispiel eigenes Anlegen und Löschen desselben Objekts). Ist die eigene Versi-

on dann konfliktfrei bezüglich der fremden können die Dokumente gemischt werden. Auch mit dieser Strategie ist es jedoch möglich, dass beim Mischen noch Modellbedingungen verletzt werden, die bei anderen Anwendungsaktionen nicht verletzt worden wären.

In der Vertragsverwaltung könnte beispielsweise durch die Aktionen der Applikation sichergestellt werden, dass nie zwei Kunden mit dem gleichen Namen und derselben Adresse angelegt werden können. Legen aber zwei Berater getrennt voneinander jeder einen neuen Kunden an und haben diese den gleichen Namen und die gleiche Adresse, tritt beim Mischen der Versionen kein Konflikt auf, da es sich um unterschiedliche neue Kundenobjekte handelt. Trotzdem ist aber die Bedingung, die die Applikationsaktionen sicherstellen, verletzt: es gibt in der gemischten Version nun zwei Kunden mit gleichem Namen und gleicher Adresse.

Ist es für eine Applikation wichtig jederzeit semantische Modellkorrektheit zu gewährleisten und genügen die Anwendungsaktionen diesen Anforderungen, bietet sich eine weitere Konfliktlösungsstrategie an. Diese ist ähnlich zu dem in CoAct (s. Abschnitt 3.2.1) verwendeten: Es wird die fremde Version als Ausgangspunkt benutzt und die eigenen Änderungen werden nicht direkt angewandt, sondern es wird versucht dieselben Applikationsaktionen auszuführen, die die eigenen Änderungen erzeugt haben. Ist das Ausführen der Aktionen erfolgreich, bekommt man eine neue gemischte Version, die semantisch korrekt ist. Treten bei der Ausführung der Aktionen Bedingungsverletzungen auf, kann die Applikation beziehungsweise der Nutzer entscheiden, welche Konsequenzen daraus gezogen werden sollen und ggf. andere Aktionen durchführen, die nun sinnvoller sind.

Die letztgenannte Strategie garantiert zwar semantische Korrektheit, stellt aber auch die höchsten Anforderungen an die Applikation beziehungswei-

se den Applikationsprogrammierer. Insgesamt bleibt festzuhalten, dass beim Benutzen der Bibliothek ein Mittelweg zwischen Korrektheitsanforderungen und Aufwand, abhängig von der Applikation gewählt werden muss. Für eine Orientierung bezüglich der Auswirkungen dieser Wahl sei auf Abschnitt 6 verwiesen.

### 4.2.6 Redundanzentfernung

Innerhalb einer Änderungsliste kann es redundante Informationen[6] bezüglich eines Modellzustands geben. Dies, sowie die Relevanz der Reihenfolge der Änderungen, wurde bereits bei der Definition der Änderungsliste bemerkt. Geht es also nur um den Endzustand eines Modells, nicht um die Historie, können eventuell Änderungen aus der Liste entfernt werden, ohne das Endresultat zu verändern. Es wird zunächst diskutiert, wie diese Änderungen zu identifizieren sind. Im Anschluss wird geklärt, wann und in welchem Maße eine Redundanzentfernung sinnvoll ist.

Für Redundanzen gibt es zwei Möglichkeiten: Einmal kann eine Kante erst gezogen und dann wieder gelöscht worden sein. In diesem Fall können genau diese beiden Änderungen gelöscht werden. Hierbei ist darauf zu achten, dass sich in der Liste zwischen den beiden Änderungen keine weitere Änderung, die sich auf dieselbe Kante bezieht, befinden darf. Aus der Sequenz „Anlegen, Löschen, Anlegen, Löschen“ dürfen also nicht nur die erste und die letzte Änderung verworfen werden, da sonst die nicht sinnvolle Sequenz „Löschen, Anlegen“ zurückbleibt.

Der zweite Fall für eine Redundanz ist das Anlegen und Löschen desselben Knotens. Tritt dies auf, sind auch alle Änderungen, die sich auf Kanten

[6] Wie in Abschnitt 4.1.3 dargestellt, kann eine Liste von Änderungen beispielsweise das Erzeugen und das Entfernen ein und derselben Kante enthalten. Verwirft man beide Änderungen ist das Resultat bei Anwenden der kompletten Liste unverändert.

zu und von diesem Knoten beziehen, redundant. Es genügt jedoch die beiden Änderungen zum Erzeugen und Entfernen des Knotens zu löschen, da die Änderungen an den Kanten bereits durch die vorherige Regel entfernt werden können, da ein (syntaktisch korrekt) gelöschtes Objekt keine Kanten mehr haben kann (ein Knoten, der Ziel oder Quelle einer Kante ist, ist immer in der Knotenmenge).

Die Durchführung der Redundanzentfernung ist - wie eingangs angedeutet - dann sinnvoll, wenn Speicherplatz oder Bandbreite gespart werden soll (nicht Rechenzeit) und nicht die Historie, sprich die Abfolge der Änderungen, die wichtige Information ist, sondern nur das vereinigte Delta der Liste. Wird die Liste als reines Mittel für die Persistenz benötigt, also zum Speichern eines Modellzustandes, kann die Redundanzentfernung daher uneingeschränkt durchgeführt werden. Wird allerdings Versionierung eingesetzt müssen teilweise bestimmte Zwischenversionen[7] erhalten bleiben. Sind die Listen, die eine solche Version ausmachen jedoch bekannt, können die jeweiligen Listen mittels der Redundanzentfernung gekürzt werden, da immer nur das Delta zwischen den Versionen interessant ist, nicht der genaue Entstehungsweg der Version. So können auch im Versionierungsbetrieb Ressourcen gespart werden.

Als Beispielszenario sei kurz die Redundanzentfernung im Client-Server-Betrieb dargestellt: Jeder Client musste hier zwei Versionen vorhalten: die mit dem Server gemeinsame Ursprungsversion und seine aktuelle Fassung des Modells. Genauer gesagt würde er also die Änderungsliste vorhalten, die er vom Server erhielt, und zusätzlich das Delta zu seiner aktuellen Version. Aus der Änderungsliste, die der Server schickt, kann bereits auf Seiten des Servers die Redundanz entfernt werden. Das Delta des Clients zu dieser Version braucht ebenfalls keine Redundanzen, da die Zwischenversionen zu

[7] s. Synchronisationsstrategien in Abschnitt 4.2.2.

keinem Zeitpunkt zum Server zurückgeschickt werden.

Werden Transaktionen eingesetzt können die Änderungslisten innerhalb der Transaktionen normalerweise durch die Redundanzentfernung laufen (es sei denn sie fungieren zum Beispiel als Zeitprotokoll). Die Redundanzentfernung über Transaktionsgrenzen hinweg macht wenig Sinn, da die Transaktionen dann nicht mehr sinnvoll einzeln verarbeitet werden können. Man würde daher eher Transaktionen durch eine schlichte Liste ersetzen, bevor man übergreifend Redundanzen entfernt.

## 4.3 Besonderheiten

Um die Definitionen und Konzepte übersichtlicher zu halten wurden bisher einige Vereinfachungen angenommen. Zum Einen wurden qualifizierte und geordnete Assoziationen nicht erwähnt. Zum Anderen wurde immer von unveränderlichen Schemata (Metamodellen) ausgegangen. Diese Aspekte werden in den folgenden Abschnitten behandelt.

### 4.3.1 Qualifizierte Assoziationen

*Qualifiziert* (qualified) heißen Assoziationen dann, wenn ein Link nicht nur durch den Namen der Assoziation und die beiden Objekte (Knoten) definiert wird, sondern auch noch ein (oder mehrere) Schlüssel zur Linkdefinition hinzugenommen wird. Aus graphentheoretischer Sicht ist dann ein Link nicht mehr ein 3-Tupel sondern ein 4- beziehungsweise n-Tupel. Sie entsprechen damit von ihrer Notation und der Bedeutung für die Persistenz den n-nären Assoziationen, bei denen nicht zwei sondern drei oder mehr Objekte (Knoten) miteinander verlinkt werden.

Für die Konzeption der Bibliothek in dieser Arbeit lassen sich die qualifizierten und n-nären Assoziationen auf zwei Arten auf die benutzten Konzep-

te zurückführen. Üblich ist die Rückführung der Links auf ein zusätzliches Objekt, das den Link repräsentiert. Dieses Objekt kann dann die n Links zu den eigenlich verlinkten Objekten zusammenführen. Das heißt in diesem Fall wird die n-näre Assoziation auf eine Klasse und n Assoziationen abgebildet. Eine weitere Möglichkeit ist, den qualifizierten Links statt einem Schlüssel einen erweiterten Assoziationsnamen zuzuordnen, beziehungsweise auch die n-nären Links nur durch ein 3-Tupel darzustellen und die fehlenden Objekte in den veränderten Assoziationsnamen zu kodieren. So würde also eine n-näre (oder qualifizierte) Assoziation auf eine endliche Anzahl an Assoziationen abgebildet werden.

Obwohl die zweite Möglichkeit zur Abbildung eher ungewöhnlich ist, soll sie hier zur Erklärung genutzt werden, da der konzeptionelle Unterschied für die Deltas auf diese Weise deutlich geringer ist: Qualifizierte und n-näre Assoziationen entsprechen dann also schlicht einer Menge von Assoziationen, die alle wie gehabt behandelt werden können. Weder die atomare Eigenschaft von Deltas, die diese Assoziationen betreffen, noch die Art der Deltas unterscheidet sich von üblichen Assoziationen. Daher treffen die obigen Konzepte auch für die komplexeren Assoziationen voll zu. Die technische Umsetzung wird durch die Art der Assoziationen etwas verkompliziert, dies wird jedoch erst im Umsatzungskapitel diskutiert.

### 4.3.2 Geordnete Assoziationen

Bei *geordneten* (ordered) Assoziationen liegt Semantik in der Reihenfolge der Links vor, es handelt sich also um Listen[8] - nicht um Mengen oder Multimengen. Es ist also möglich im Modell abzufragen, welche Reihenfolge die Nachbarn eines Objekts haben, die über solche Assoziationen zu erreichen sind. Ebenso ist es oft möglich die Links in einer gewünschten Reihenfol-

[8] in denen allerdings oft jedes Element nur ein Mal enthalten sein darf

ge anzulegen. Für die Konzepte in dieser Arbeit bedeutet die Existenz von geordneten Assoziationen eine zusätzliche Anforderung. Auch hier gibt es Möglichkeiten die geordneten Assoziationen auf bestehende Konzepte abzubilden. Es könnten wieder Hilfsobjekte oder Hilfsassoziationen eingeführt werden. Im Fall der ganzzahlig nummerierten Hilfsassoziationen wären diese jedoch voneinander abhängig: Wird am Anfang einer Liste ein Link (Kante) eingefügt, verschieben sich alle folgenden Links. Die daraus entstehenden Konsequenzen für die bisherigen Herleitungen sind recht umfangreich.

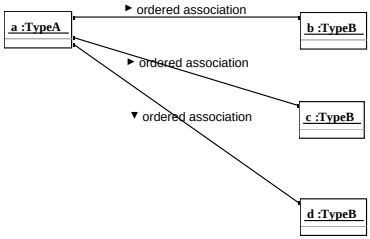


Abbildung 4.5: *Objekte die mit einer geordneten Assoziation verlinkt sind.*

Eine weitere Möglichkeit die geordneten Assoziationen auf Hilfsassoziationen abzubilden, ist, sie mit Brüchen zu nummerieren. Die Konsequenzen für die Herleitung sind gering, da sie sich direkt auf qualifizierte Assoziationen abbilden lassen. Allerdings ist die Nummerierung mit Brüchen nicht mit der diskreten Nummerierung in der gängigen Implementierung von geordneten Assoziationen vergleichbar.

Daher wird hier auf die durchaus übliche Abbildung auf Hilfsobjekte (Hilfsknoten) zurückgegriffen: Ein Link einer geordneten Assoziation (siehe Abbildung 4.5) wird dann auf theoretischer Ebene durch einen zusätzlichen Knoten und mehrere Kanten repräsentiert. Dabei geht von dem Hilfsknoten jeweils eine Kante zu den zu verlinkenden Knoten. Bis zu zwei weitere Kanten gehen zu dem nächsten beziehungsweise vorherigen Hilfsknoten der ge-

ordneten Assoziation (siehe Abbildung 4.6). Auf theoretischer Ebene gelten durch die Abbildung natürlich noch alle zuvor gemachten Annahmen und Schlüsse. Auch für das eigentliche Modell lassen sich die Aussagen zurückführen, dabei sind insbesondere Deltaberechnung und Konflikterkennung sowie syntaktische Korrektheit zu erwähnen:

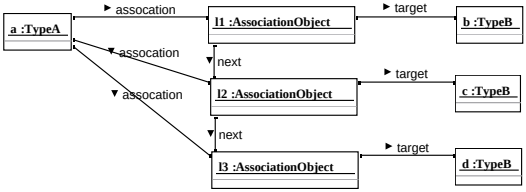


Abbildung 4.6: *Die geordneten Links wurden auf Hilfsobjekte abgebildet.*

Bei der Deltaberechnung fallen statt einem atomaren Delta für eine geänderte geordnete Kante auf theoretischer Ebene mehrere atomare beziehungsweise ein nicht-atomares Delta an. Jedoch stehen entweder alle diese atomaren Deltas mit anderen in Konflikt oder keines der logisch zusammengehörigen Deltas. Wird nun bei der Konfliktlösung entweder das komplette nichtatomare betroffene Delta verworfen oder eben keine der Änderungen die aus einer geänderten Kante hervorgingen, ist weiterhin die syntaktische Modellkorrektheit gewährleistet. Dies lässt sich mit den bereits erwähnten Garantien erklären, bei denen bereits begründet wurde warum stets gemeinsam auftretende Änderungen auch für stets gemeinsam vorhandene Modellelemente beziehungsweise -kanten sorgen. Die Abbildung des theoretischen Modells auf das eigentliche ist somit immer möglich.

### 4.3.3 Schemaevolution

Alle Definitionen, Schlüsse und Erklärungen haben bisher vorausgesetzt, dass sobald zwei Modelle betroffen sind, die Schemata der Modelle gleich sein müssen. Auch in der Praxis ändert sich das Schema beim Anlegen von neuen Versionen eines Dokuments nicht. Daher ist es für ein Dokument normalerweise nicht nötig gleichzeitig mit Modellen mit unterschiedlichen Schemata umgehen zu können. Allerdings kommt es durchaus häufig vor, dass sich Metamodelle im Zuge der Weiterentwicklung eines Programms verändern. Gibt es nach einer solchen Änderung noch Daten, die mit einer vorherigen Version des Metamodells erzeugt wurden - sei es von einer Applikationsinstanz mit alter Version oder eine früher gespeicherte Dateiversion - muss man diese meist trotzdem noch verarbeiten und dabei eine die Version beziehungsweise die Änderungen des Metamodells berücksichtigen.

Es entsteht also die Notwendigkeit, auch von dem Metamodell einer Anwendung explizit Versionen zu verwalten. Nur so kann auf die Änderungen am Metamodell dezidiert eingegangen werden. Idealerweise wird auch die Metamodellversion beim Speichern eines Modells vermerkt.

Dass eine Applikation mit mehreren Versionen von Metamodellen arbeitet ist jedoch meist nicht oder nur sehr aufwändig zu erreichen[9]. Um trotzdem noch mit solchen Daten arbeiten zu können, ist es stattdessen möglich die Daten zunächst zu konvertieren, um ein Modell zu erhalten, das auf dem neuen Metamodell basiert. Auch sind dann alle Operationen wie Persistenz, Mischen und Synchronisierung wieder wohldefiniert. Die Konversion von

[9] In Java ist es zum Beispiel möglich zwei verschiedene Klassen mit gleichem Namen mit speziellen ClassLoadern zu laden. Die Instanzen dieser Klassen können dann eingeschränkt miteinander operieren. Die Zuweisung von Instanzen der einen Klasse zu Feldern vom Typ der anderen Klasse sind aber nicht möglich. Der Einsatz von ClassLoader generell - und insbesondere das Laden von Klassen mit gleichem Namen - bringt jedoch einen hohen Aufwand und Inkompatibilitäten mit anderen Mechanismen, die ClassLoader verwenden, mit sich. Daher wird von derartigen Lösungen meist abgesehen.

Modellen mit einem Schema in ein anderes ist allerdings teilweise aufwändig und stark abhängig von der Semantik des Modells. Die Regeln beziehungsweise Modelltransformationen zur Übersetzung zwischen den Schemata sind daher applikationsspezifisch und müssen vom Designer des Modells spezifiziert werden.

Von Seiten der Bibliothek können jedoch einige Hilfestellungen zur Anwendung dieser Regeln geleistet werden. Zunächst gilt es festzustellen ob und welche anderen Schemata zum Einsatz kommen. Sind die unterschiedlichen Versionsnummern der Schemata bekannt, können die Daten zum älteren Schema in die des neueren konvertiert werden. Zur besseren Wartbarkeit wäre etwa möglich einen Satz von Konvertierungsregeln pro Metamodell-Versionsnummer zu spezifizieren, um jeweils Daten für die vorhergehende Version umzuwandeln. Es sei allerdings angemerkt, dass eventuell effektivere Konversionen möglich werden, wenn auch Regeln spezifiziert werden, um über mehrere Versionen hinweg zu konvertieren.

Eine sehr einfache Änderung, die am Metamodell stattgefunden haben könnte, ist die Umbenennung einer einzelnen Klasse. So könnte etwa die Klasse „Kunde“ in „Person“ umbenannt worden sein. Die Applikation hat daraufhin nur noch die neue Klasse „Person“ zur Verfügung und kann Dateien, die die Klasse „Kunde“ als Typ eines Objektes spezifizieren, zunächst nicht laden - die Klasse kann nicht gefunden werden. Zum erfolgreichen Laden muss jede Änderung, die den alten Klassennamen enthält, zunächst konvertiert werden. In diesem Fall ist die Konvertierungsregel schlicht die Ersetzung des Namens „Kunde“ durch den Namen „Person“.

Teilweise erfordern Änderungen am Schema sogar keine Konvertierungen. So ist zum Beispiel das Hinzufügen von Klassen (Knotennamen) und Assoziationen (Kantennamen) in der Regel unkritisch. In manchen Fällen

wäre es denkbar, dass für bestimmte Situationen auch zusätzliche Objekte oder Links in den Daten erzeugt werden müssen[10]. Ein Beispiel für einfache Konversionen ist außerdem das Entfernen von Informationen aus dem Modell. Wird etwa ein bestimmtes Attribut (Kantenname) nicht mehr benötigt, können alle atomaren Deltas, die dieses Attribut betreffen schlicht verworfen werden.

Teilweise müssen aber auch komplexere Konversionen bei manchen Umstrukturierungen vorgenommen werden. Beispielsweise könnte eine Vererbung durch eine Delegation ersetzt werden und so ein Knoten und dessen Kanten, nun auf zwei Knoten mit einer zusätzlichen Verbindung aufgeteilt werden. Im Falle von wohldefinierten *Refactorings* wäre es hier sogar denkbar, dass ein Modellierungswerkzeug, das das Metamodell bearbeitet, auch im selben Zuge Transformationsregeln für die Daten generiert.

## 4.4 Technische Aspekte

Da die in dieser Arbeit entwickelte Bibliothek mit beliebigen Java-Metamodellen arbeiten soll und nicht selbst die Arbeitsspeicherstrukturen vorgibt, unterscheiden sich die technisch durchführbaren Änderungen auf dem Modell von den theoretisch möglichen. Insbesondere durch *Zugriffsmethoden*[11] oder *Assoziationsklassen*[12] werden die Änderungsoperationen definiert. Die Abstraktion von der konkreten Implementierung des Modells wird durch eine

[10] Beispielsweise könnte ein Textattribut „Adresse“ gegen eine Assoziation auf eine neue Klasse namens „Adresse“ getauscht worden sein. In diesem Fall muss statt dem vorherigen (primitiven) Adresstext ein Adressobjekt angelegt und mit den Daten befüllt werden.

[11] In Java ist es üblich Attribute von Klassen nicht direkt nach außen zugreifbar zu machen, sondern das Verändern und Auslesen der Attribute nur über spezielle Methoden der Klasse zuzulassen.

[12] Solche Klassen sind das Äquivalent zu Zugriffsmethoden für Assoziationen. Da hier oft mehrere Objektreferenzen abgelegt werden müssen, basieren die Assoziationsklassen auf sogenannten Collections.

entsprechende Schicht realisiert, die in Abschnitt 5.2 beschrieben wird. Dennoch sind einige potentielle Unterschiede zu diskutieren.

Entgegen der theoretischen Betrachtung ist es beispielsweise nicht üblich zu-1 Assoziationen genauso wie zu-n Assoziationen zu behandeln. Bei zu-1 Zugriffsmethoden oder Assoziationsklassen wird gängigerweise nicht eine Kante gelöscht und anschließend eine neue gezogen, sondern in einer Operation wird der Link auf ein anderes Objekt umgesetzt. Es werden also die Informationen aus zwei aufeinanderfolgenden atomaren Änderungen benötigt, um die Operation auf dem Modell auszuführen. Aus diesem Grund weicht der Begriff der Änderung in der Implementierung leicht von der theoretischen Betrachtung ab: In einer Änderung kann das Entfernen einer Kante und das Ziehen einer Kante der gleichen Art zu einem anderen Objekt abgelegt sein. Die theoretischen Betrachtungen treffen aber trotzdem zu, da diese zusammengefasste Änderung auch als kleine Änderungsliste gesehen werden kann.

Weiterhin ist es bei zu-1 Assoziationen in der Regel auch temporär nicht möglich, mehr als ein Objekt zu referenzieren. Aus diesem Grund wäre es etwa schwierig ein Protokoll anzuwenden, das zunächst einen weiteren Link einer zu-1 Assoziation anlegt und erst im Nachhinein den alten Link löscht, obwohl dies von theoretischer Seite kein Problem darstellt. Glücklicherweise kann ein solches Protokoll jedoch garnicht erst entstehen, da ein Protokoll ja durchgeführte Änderungen an einem gleichartigen Modell enthält, und sich derartige Änderungen nicht zugetragen haben können.

Eine weitere Eigenschaft mancher Zugriffsmethoden oder Assoziationsklassen sind sogenannte *Nebeneffekte*. Die verträgliche und allgemein akzeptierte Variante dieser zusätzlichen Codezeilen in den Methoden einer Assoziation dient der automatischen *referenziellen Integrität*. Diese stellen bei bidirektionalen Assoziationen sicher, dass ein Link vollständig ist, also

aus zwei wechselseitigen Referenzen besteht. In den theoretischen Betrachtungen wurden bidirektionale Assoziationen als zwei voneinander getrennte Assoziationen behandelt. Daher gibt es auch zwei atomare Änderungen, die das Ziehen eines bidirektionalen Links beschreiben, da der Link in der Theorie aus zwei Kanten besteht. Hier gibt es für die technische Umsetzung mehrere Möglichkeiten: Ist bekannt, dass es sich um eine bidirektionale Assoziation mit automatischer referenzieller Integrität handelt, kann eine der beiden Änderungen schlicht beim Aufzeichnen ignoriert (verworfen) werden. Die zweite Möglichkeit wäre, zu erlauben, dass bereits die Rückrichtung eines Links durch das Ziehen der ersten Linkhälfte geschehen ist, und gegebenenfalls eine Änderungsoperation beim Anwenden auszulassen oder das Modell die doppelte Information erkennen zu lassen. In der Realisierung wurde letzteres Verfahren gewählt, mit dem keine Probleme auftraten.

Steht die Information zur Verfügung wodurch Modelländerungen hervorgerufen wurden, kann ein komplexeres Verfahren zur Behandlung von automatischer referenzieller Integrität implementiert werden. Es ist dann nämlich bekannt, dass für die zweite Änderung, die sich auf einen bidirektionalen Link bezieht, die erste Änderung für diesen Link ursächlich war. Diese Information lässt sich dann auch beim Anwenden der Änderungsliste verwenden, um festzustellen, ob die zweite Änderung noch ausgeführt werden muss, oder nicht. Da dieses Verfahren jedoch für die Modelle in den Fallstudien keinen Mehrwert brachte, aber zusätzlichen Aufwand für das Beschaffen der Abhängigkeitsinformationen bedeutete, wurde es wieder verworfen.

Die für diese Arbeit schlecht kalkulierbaren und auch in der Softwaretechnik wenig akzeptierten Nebeneffekte von Zugriffsmethoden sind solche, die weitere beliebige Modelländerungen durchführen. Es könnte etwa eine Methode zum Setzen eines Attributs „nachbarAnzahl“ auch automatisch

eine entsprechende Anzahl Nachbarobjekte erzeugen und verlinken. Diese Art von Nebeneffekten behindert die korrekte Arbeitsweise der hier vorgestellten Bibliothek sehr[13], da sie nicht vereinbar mit den theoretischen Überlegungen sind. In der Praxis wird Quelltext mit solchen Eigenschaften aber ohnehin eher vermieden. In Fällen bei denen Nebeneffekte trotzdem in der Applikation verbleiben sollen, können sie zum Beispiel während der Verarbeitung durch CoObRA abgeschaltet werden[14].

### 4.4.1 Serialisierung von Referenzen

Neben den Eigenschaften der Metamodelle gibt es noch einen wichtigen Punkt, der nicht in den theoretischen Überlegungen behandelt werden konnte: Identifizieren von Objekten. Sind Objekte graphentheoretisch schlicht Knoten, die direkt identifizierbar sind, stellt sich dies in der Praxis weit komplexer dar, wenn die Lebenszeit eines Objektes über die Laufzeit eines Programms hinaus geht.

Um Änderungsdaten in Dateien oder Strömen zu speichern, ist es notwendig Objektreferenzen geeignet in serialisierbare Daten umzuwandeln. Der übliche Weg ist es, für jedes Objekt einen Identifizierer - eine ID - zu vergeben und an Stelle der Referenz diesen zu schreiben. Solche IDs können Formen von simplen durchgezählen Zahlen bis hin zu komplexen Texten annehmen. In gängigen Dateiformaten wie XMI [17] oder der Java Serialisierung [27] werden lediglich lokale Referenzen kodiert. Daher sind auch die Anforderungen an die IDs in diesen Fällen niedrig. Es reichen lokal eindeutige IDs, die schlicht einer Tag-Nummer beziehungsweise einer Strom-Position

[13] In diesem Fall würden etwa die Nachbarn beim Laden doppelt erzeugt: einmal durch das Setzen des Attributs und einmal durch das Wiederholen der Objekterzeugungsoperationen.

[14] Die Bibliothek bietet dich Möglichkeit abzufragen, ob gerade eine Lade-, Rückgängig- oder Wiederholen-Operation ausgeführt wird.

entsprechen. Sollen die IDs auch über die Lebenszeit des Datenstroms hinweg gültig bleiben, reicht diese Art der Identifizierung nicht mehr aus. Die IDs müssen unabhängig von Positionen in Strömen und Dateien werden. Einfaches Durchzählen von Objekten funktioniert allerdings noch immer. Eine Anforderung, die durch einen simplen Zähler nicht mehr erfüllt werden kann, ist die Forderung nach Eindeutigkeit über Strom- beziehungsweise Dateigrenzen hinweg. Diese Anforderung entsteht wenn dieselbe Referenz in unterschiedlichen Dateien (oder Dateiversionen) gespeichert werden soll - zum Beispiel für Querreferenzen zwischen zwei Modellen.

Um auch die Forderung nach dateiübergreifender Eindeutigkeit erfüllen zu können, bieten sich mehrere Lösungsstrategien an: Die einfachste ist die zufällige Generierung von langen Nummern oder Zeichenketten. Je länger diese sind desto unwahrscheinlicher wird eine doppelt vergebene ID. Ab etwa 64 Bit mehr als es schätzungsweise Objekte geben wird, können Dopplungen quasi ausgeschlossen werden (Wahrscheinlichkeit $< 1/2^{64}$). Dieser kleine Unsicherheitsfaktor kann eventuell in Kauf genommen werden. Diese Methode hat jedoch den gravierenden Nachteil, dass IDs relativ viel Speicherplatz einnehmen. Dies ist der Erfüllung der Forderung nach schnellem Schreiben und Lesen der persistenten Daten abträglich. Daher wurden andere Alternativen evaluiert.

Eine weitere Möglichkeit eindeutige IDs über Modelgrenzen hinweg zu vergeben, ist, eine zentrale Instanz zur ID-Vergabe zu etablieren. Diese kann wiederum schlichte Zähler zum Generieren einer ID verwenden. Eine solche Instanz zur Verfügung zu stellen, ist allerdings nicht in jedem Szenario möglich. Ein wichtiger Vorteil von Versionierung mit optimistischem Sperrkonzept ist, dass zeitweilig auch ohne Verbindung zu einem zentralen Repository weitergearbeitet werden kann. In dieser Zeit werden aber Objekte erzeugt, für die dann je eine ID benötigt wird. Soll während der verbindungslosen

Zeit auch auf die Verbindung zur zentralen ID-Vergabe verzichtet werden, sind teils aufwändige Verfahren nötig. Eine Möglichkeit ist, vorweg eine Menge von IDs zu beantragen, die dann ohne weitere Verbindung vergeben werden kann. Soll eine ausreichende Menge an ID's vorhanden sein, muss eine unverhältnismäßig große Anzahl an freien IDs vorgehalten werden, was oft nicht akzeptabel ist. Eine Alternative zum Vorhalten von IDs ist die temporäre Vergabe von lokalen IDs, die bei der nächsten Verbindung durch zentral vergebene IDs ersetzt werden. Dieses Ersetzen ist jedoch nicht möglich, wenn Persistenzdaten nicht nächträglich verändert werden können (zum Beispiel bei Strömen), aber sofort geschrieben werden müssen.

Da keiner dieser Ansätze zur ID-Vergabe überzeugen kann, wurde eine Kombination dieser Verfahren gewählt. Die IDs bestehen in CoObRA 2 aus einem Präfix und einem Suffix, wobei das Suffix eine schlichte, aufwärts laufend vergebene Zahl ist und das Präfix beliebig komplex werden kann: Beim lokalen Speichern ohne Server zum Beispiel müssen die IDs nur innerhalb eines Repositories eindeutig sein. In diesem Fall ist das Präfix leer. Sollen die IDs möglichst universell eindeutig sein, wird eine zufällige Zeichenkette als Präfix generiert. Dies kommt beim der SCM-Integration zum Einsatz da IDs nicht zentral vergeben werden können. Ist jedoch eine zentrale Instanz zur ID-Vergabe verfügbar (Client-Server Modus) vergibt diese eine Ihrer IDs als Präfix für die Repositories. Dies ist auch hierarchisch organisiert möglich - eine komplette ID eines übergeordneten Knotens kann als Präfix eines untergeordneten Repositories verwendet werden. Die Vorteile dieser Lösung liegen auf der Hand: In allen Situationen werden die Anforderungen erfüllt und es kann je nach aktueller Anforderung die Komplexität variiert werden. Und selbst bei hoher Komplexität des Präfix' ist die ID noch kürzer

als eine komplett zufällig generierte ID[15]. Damit wird die Verarbeitungsgeschwindigkeit nur soweit belastet wie nötig.

[15] Durch die geringere Anzahl an zufällig vergebenen IDs kann bei gleicher Wahrscheinlichkeit für Kollisionen ein kürzerer zufälliger Schlüssel genutzt werden.

# 5 Umsetzung

Die während der Diplomarbeit [22] enstandene Bibliothek CoObRA 1 war nicht auf Flexibilität ausgelegt. Für die Erforschung der eingangs beschriebenen Eignungsarten war diese Implementierung des Basiskonzepts nicht mehr weiterzuverwenden. Auch die in Kapitel 2 ausgeführten Anforderungen wurden von CoObRA 1 nur zu einem rudimentären Teil erfüllt. Aus diesem Grund wurde eine neue Bibliothek entwickelt. In diesem Kapitel werden Überlegungen und Implementierungsdetails, die zur Bibliothek CoObRA 2 geführt haben, eingehend präsentiert.

## 5.1 Architektur

Für die Flexibilität einer Software ist deren Architektur von entscheidender Bedeutung. Insbesondere die Modularisierung beziehungsweise Unterteilung in Softwarekomponenten ist ein bekanntes Mittel aus der Softwaretechnik, um Teilfunktionalitäten austauschen zu können. So soll ein größtmöglicher Wiederverwendungswert der Software - zum Einsatz für verschiedenste Aufgaben - erreicht werden.

Um die Bibliothek in Module zu unterteilen sind zunächst Teilaufgaben zu identifizieren. Ein Modul kann dann eine Menge von Teilaufgaben zusammenfassen. Welche Teilaufgaben zu einem Modul zusammengefasst werden hängt von der Notwendigkeit ab, diese Aufgaben auf unterschiedliche Art und Weise zu erledigen und von der Abhängigkeit zwischen den Tei-

laufgaben. Haben zwei Teilaufgaben starke Abhängigkeiten und werden sie dennoch in unterschiedlichen Modulen untergebracht kann dies die Schnittstellendefinition für die Module stark aufblähen. Um die Software zu modularisieren wurden daher zunächst Teilaufgaben und deren Abhängigkeiten ermittelt:

- Erfassen von Modelländerungen

  Die Quelle für das Erfassen von Modelländerungen ist das Modell der Applikation, somit ist die Erfassung sehr applikationsspezifisch. In der Bibliothek muss also eine Schnittstelle für das Benachrichtigen bei Änderungen bereit stehen. Gegebenenfalls können auch Standard- beziehungsweise Beispielimplementierungen für Aufrufe an diese Schnittstelle zur Verfügung gestellt werden.

- Anwenden von Änderungsdaten auf ein Modell

  Auch die Operationalisierung von Änderungsdaten ist applikations- beziehungsweise modellspezifisch. Die Operaionalisierung sollte daher allgemein implementiert werden und über eine entsprechende auswechselbare Abstraktionsschicht auf das Modell zugreifen. Den allgemeinen Teil der Operationalisierung auszutauschen ist voraussichtlich nicht nötig, er ist eher konfigurierbar zu gestalten. Die Anwendung von Änderungsdaten hängt nicht direkt von der Erfassung der Modelländerung ab.

- Halten von Änderungsdaten im Speicher

  Wie schon die ersten genannten Teilaufgaben hängen alle Teilaufgaben, die sich in irgendeiner Weise mit Änderungsdaten befassen, von der Repräsentation der Änderungsdaten im Speicher ab. Eine gemeinsame Schnittstelle für den Zugriff auf diese Daten erscheint öko-

nomisch. Für die Haltung dieser Daten im Arbeitsspeicher ist eine schlichte Implementierung dieser Schnittstelle erforderlich.

- Durchführung und Verwaltung von Undo/Redo

  Um einen Satz von Modelländerungen rückgängig zu machen oder zu wiederholen nachdem er rückgägig gemacht wurde, ist der Zugriff auf die Änderungsinformationen sowie auf die Operationalisierung nötig. Insbesondere muss auf den Daten navigiert werden können.

- Navigation durch Änderungsdaten im Speicher

  Für Undo, Redo und Laden ist es nötig den jeweils vorherigen respektive nächsten Eintrag in einer Änderungsliste zu erfragen. Werden Daten gefiltert ist es wünschenwert den nächsten Eintrag, der bestimmten Kriterien entspricht, zu erhalten. Insbesondere auf einem Versionierungsserver ist auch die Navigation zu Änderungen einer bestimmten Version beziehungsweise Transaktion nötig.

- Transaktionsverwaltung

  Die Transaktionsverwaltung kann Transaktionen starten, abschließen und abbrechen. Transaktionen gruppieren Änderungen und können benannt werden, um Versionen zu markieren. Somit hängt die Transaktionsverwaltung von Operationalisierung und Änderungsrepräsentation ab.

- Persistente Speicherung von Änderungsdaten

  Wie die Speicherung der Änderungsdaten direkt im Arbeitsspeicher muss auch die persistente Speicherung Navigation auf den Daten zulassen. Je nach Anwendungsszenario unterscheidet sich die Menge an zu unterstützenden Navigationsmöglichkeiten. Für jede Art der persistenten Speicherung gilt es ein Datenformat zu definieren, das von der

Art der Änderungsdaten abhängt und sich daher auf die Repräsentation der Daten im Arbeitsspeicher stützt. Die meisten vorstellbaren Arten der persistenten Speicherung setzen voraus, dass Objektreferenzen durch einen geeigneten Namen oder Schlüssel (ID) repräsentiert werden.

- Vergabe und Verwaltung von IDs

  Die Auswahl eines geeigneten Namens oder Schlüssels (ID) zum Identifizieren eines Modellobjekts beziehungsweise zum Repräsentieren einer Objektreferenz muss - abhängig vom Anwendungsszenario - verschiedene Anforderungen erfüllen. Während es bei schlichter lokaler Persistenz ausreicht, dass für je zwei unterschiedliche Objekte die IDs innerhalb derselben Datei unterschiedlich sind, ist es dagegen bei verteilter Persistenz notwendig, dieses Kriterium für alle Dateien beziehungsweise Speicherbereiche auf unterschiedlichen Systemen zu gewährleisten. In diesem Fall ist die Abstimmung der ID-Vergabe mit einer oder mehreren anderen Anwendungsinstanzen nötig; es gibt also eine potentielle Abhängigkeit zur Kommunikationsschicht.

- Navigation durch Änderungsdaten in Persistenzschicht

  Die Navigation durch die Änderungsdaten in einem Persistenzmedium muss nicht notwendigerweise genauso flexibel sein, wie die Navigation im Arbeitsspeicher; möglichst umfassende Navigationsmöglichkeiten sind jedoch wünschenswert, um zu vermeiden, dass vor der Durchführung einer Operation die gesamte Änderungsliste in den Speicher geladen werden muss. Ebenso wäre ein einheitliches Navigationsinterface hilfreich um die Schnittstellen übersichtlich zu halten. Die tatsächlich zur Verfügung stehenden Navigationsmöglichkeiten werden sich jedoch nach der Art des Speichermediums richten müssen.

- Versand und Empfang von Änderungsdaten über Ströme oder Nachrichten

  Zum Versenden und Empfangen von Änderungsdaten müssen ähnliche Datenströme erzeugt und gelesen werden, wie bei der persistenten Speicherung im Allgemeinen. Zusätzlich wird eventuell mehr Kontrolle über die Granularität der geschriebenen Daten benötigt, sowie über die versendete Datenmenge (in Bytes). Eine Wiederverwendung der Funktionalität für die dateibasierte Speicherung erscheint angebracht.

- Persistente Speicherung in einem SCM-System

  Für die Speicherung in textbasierten SCM-Systemen ist ein Textdateiformat notwendig. Eine Wiederverwendung der Funktionalität für die allgemeine dateibasierte Speicherung erscheint auch hier angebracht. Die Forderung nach kleinen Dateien beziehungsweise wenigen versandten Bytes ist allerdings teilweise konträr zur Forderung nach einem Textformat.

- Lesen von SCM-Rückmeldungen/Daten

  Wird eine Textdatei von SCM-Systemen verwaltet werden im Konfliktfall SCM-Markierungen innerhalb der Datei untergebracht. Diese gilt es zu verarbeiten. Dies stellt einerseits zusätzliche Anforderungen an das verwendete Textformat, andererseits müssen diese Markierungen (die eigentlich nicht Teil des zu schreibenden Formats sind) beim Einlesen von Dateien korrekt behandelt werden.

- Konfliktdetektion

  Unabhängig von der Kommunikationsform (zum Beispiel Strom, Nachrichten oder SCM) können bei nebenläufigen Änderungen an dem replizierten Modell Konflikte zwischen den anzuwendenden Änderungen

auftreten. Diese gilt es zu detektieren und der Applikation zu melden, um eine Konfliktbehandlung möglich zu machen.

- Konfliktbehandlung

  Zur Konfliktbehandlung gilt es eine Schnittstelle zur Applikation anzubieten, da die Behandlung nur applikationsspezifisch sein kann, das heißt dass sie weitgehend von der Anwendung durchgeführt wird. Eine automatische Konfliktbehandlung, die die Modellsemantik der Applikation nicht mit einbezieht, kann jedoch ebenso angeboten werden.

- Redundanz in Änderungsdaten entfernen

  Da die Größe eines Änderungsprotokolls über die Zeit monoton wächst, solange die komplette Historie erhalten bleibt, ist die Funktion zum Entfernen alter Versionen aus einem Protokoll gefragt. Insbesondere für Speicherung und Versand von Änderungslisten, wo nur eine Endversion oder das Delta zwischen zwei Versionen benötigt wird, nicht aber die Historie, ist die Entfernung der dadurch definierten Redundanz wichtig.

- Fehlerbehandlung bei den diversen Teilaufgaben

  Ab einem gewissen Reifegrad einer Anwendung ist es notwendig unerwartete Situationen in den Versionierungsabläufen anwendungsspezifisch zu behandeln. Manche Fehlersituationen sollen ignoriert und der Ablauf wiederholt werden, in anderen Fällen ist der User auf diverse Arten zu informieren, in weiteren eventuell sogar die Applikation zu terminieren. Um dem Anwendungsentwickler die Arbeit zu erleichtern ist hierzu eine zentrale Schnittstelle sinnvoll.

- Filtern von Änderungslisten, -strömen und -nachrichten

Beim Einlesen beziehungsweise Empfangen und beim Schreiben beziehungsweise Senden von Änderungsdaten ist es teilweise notwendig die Daten zu filtern. Dabei soll zum Beispiel für jede Modelländerung ein Veto eingelegt werden können. Es wäre jedoch auch denkbar die Änderungsdaten auf andere Weise zu beeinflussen. Ein möglichst flexibler Filtermechanismus ist wünschenswert.

Schon an Hand der Anzahl der Teilfunktionalitäten wird deutlich, dass ein späterer Nutzer der Bibliothek leicht den Überblick verlieren kann. Eine zumindest bei anfänglicher Arbeit knappe Schnittstelle sollte also geschaffen werden. Als zentrales Element für die Funktionalität wurde daher eine Klasse namens *Repository* eingeführt. Sie bietet direkte Schnittstellen zu den am häufigsten benötigten Funktionen und erlaubt weiterhin den Zugriff auf die Untermodule, um auch die komplette Funktionionaltät erreichbar zu machen. Zudem wird die Klasse genutzt, um zentrale Funktionen, die nicht in ein Modul ausgelagert wurden, zu implementieren.

### 5.1.1 Änderungsdaten

Wie bereits angedeutet ist für viele Funktionen die Verarbeitung von Änderungsdaten notwendig. Es liegt daher nahe eine gemeinsame Schnittstelle für Änderungsdaten festzulegen. Die zu repräsentierenden Informationen wurden in Kapitel 4 vorgestellt. Eine atomare Änderung (*Change*) speichert demnach die Art der Änderung (*kind*), das betroffene Objekt (*affectedObject*) und eventuell Zusatzinformationen über das betroffene Feld (*field*), den alten und/oder den neuen Wert (*oldValue*, *newValue*) und eventuell einen Schlüssel für den Teil des Feldes (*key*) - je nach Art der Änderung. Zusätzlich sind Transaktionen zu repräsentieren. Diese sind schachtelbar und können atomare Änderungen enthalten. In Abbildung 5.1 sind die sich daraus ergebende Kompositstruktur und die erwähnten Attribute und Assoziationen

dargestellt.

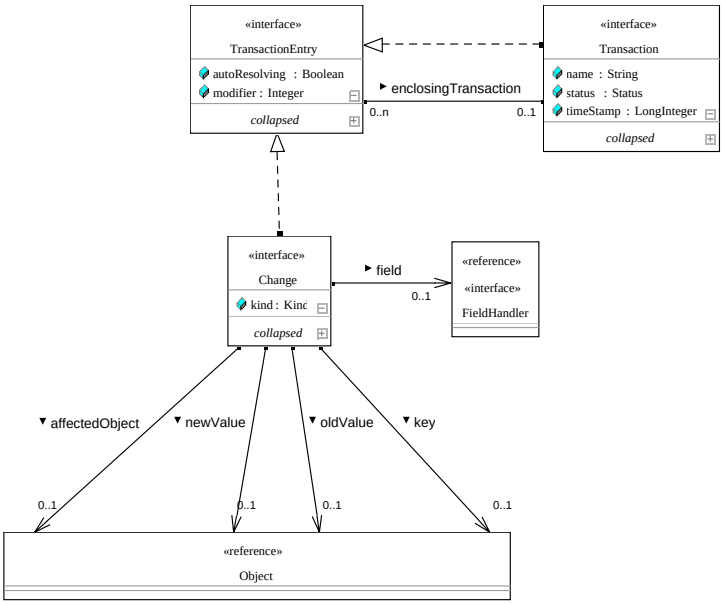


Abbildung 5.1: *Interface zum Zugriff auf Änderungsdaten: Change und Transaction*

Die Speicherung der Daten im Arbeitsspeicher ist nach der Schnittstellendefinition bereits durch eine Implementierung dieser Schnittstelle erledigt. Schon die Navigation auf diesen Änderungsdaten erfordert jedoch bereits eine erweiterte beziehungsweise zusätzliche Schnittstelle. Da für die Navigation das Ermitteln der nachfolgenden Änderung nicht ausreicht, sondern auch noch einige komplexere Navigationsoperationen angeboten werden sollen (siehe Abschnitt 5.3), wurde die Schnittstelle der Changes und Transactions nicht um diese Methoden erweitert. Stattdessen gibt es dafür eine gesonderte Schnittstelle, auf die im Folgenden eingegangen wird.

In engem Zusammenhang mit der Navigation auf Änderungsdaten stehen

weiterhin die Teilfunktionen der persistenten Speicherung, der Versand und Empfang von Änderungsdaten sowie die Redundanzeliminierung und die Filter für Änderungen. All diese Teilfunktionen sollen für den Rest der Bibliothek beziehungsweise die Anwendung transparent sein. Das bedeutet alle sollten die gleiche Schnittstelle bieten. So ist es möglich, dass derselbe Quellcode für den Bibliotheksablauf verwendet werden kann, ohne unterscheiden zu müssen, aus welcher Datenquelle die Änderungen kommen und ob sie gefiltert werden oder nicht. Diese Schnittstelle wurde Persistenzmodul getauft (*PersistencyModule*). Details zu den Persistenzmodul-Implementierungen finden sich in Abschnitt 5.3.

Das Erfassen von Modelländerungen ist weitgehend unabhängig von anderen Funktionen. Aus Sicht der Bibliothek reicht eine einzige Methode im Repository, die einzelne atomare Änderungen entgegennimmt. Abhängig ist das Erfassen von Änderungen insoweit von der Modell-Abstraktionsschicht (siehe Abschnitt 5.2), als dass eine Referenz auf ein Feld aus dem Metamodell zur Änderungsinformation gehört (wenn es sich um eine Feldänderung handelt). Der Einfachheit halber wurde eine Schnittstelle für die erfassende Klasse namens *ChangeRecorder* etabliert. Es ist für eine Applikation jedoch nicht notwendig, diese Schnittstelle exakt zu implementieren, da die Änderungen vom Repository entgegengenommen werden. Die Bibliothek hat aber bereits einige Standardimplementierungen der Schnittstelle für *PropertyChangeEvents*, MOF Modelländerungs-Ereignisse und für EMF.

### 5.1.2 Operationalisierung

Den ersten echten Nutzen bringt die Bibliothek beim Zurückrollen oder (erneutem) Anwenden von Änderungsdaten auf ein Modell. Diese Teilfunktionalität wurde Operationalisierung (*operationalization*) genannt. Der Zugriff auf diese Funktionen von außerhalb der Bibliothek geschieht über das Re-

pository. Dies übernimmt auch die grundsätzliche Verwaltung der anzuwendenden Änderungen. Da die Reihenfolge und Logik der Operationalisierung immer gleich abläuft, wurde sie nicht in ein Modul ausgelagert. Sie verteilt sich stattdessen auf die Transaktionsimplementierung: Beim Anwenden von Änderungen werden gängigerweise komplette Transaktionen verarbeitet, anstatt nur einzelne atomare Änderungen anzuwenden. Da Transaktionen auf ihre Kindelemente - andere Transaktionen und Änderungen - zugreifen können, bot es sich an die Verwaltung der Operationalisierung von Transaktionen (*commit/recommit*, *abort/rollback*) direkt in den Transaktionsklassen zu implementieren. Somit ist auch bereits ein Großteil der Transaktionsverwaltung behandelt. Schnittstelle für Transaktionen ist allerdings auch das *Repository*.

Die tatsächlichen Auswirkungen einer Änderungsoperationalisierung auf das Modell ist modellspezifisch. Um trotzdem die Bibliothek modellunabhängig gestalten zu können, wurde eine Abstraktionsschicht für Modelle etabliert. Diese Schicht bietet Methoden zum Ermitteln von Klassen und Feldarten sowie zum Ändern von Feldwerten. Diese Methoden werden von der Operationalisierung aufgerufen. Details zur Abstraktionsschicht finden sich in Abschnitt 5.2.

### 5.1.3 IDs und Kommunikation

Um die Verwaltung und lokale Vergabe von IDs kümmert sich ein separates Modul, das *IdentifierModule*. Da die Identifizierer zwar von anderen Modulen benutzt werden, die Verwaltung und Vergabe aber nur minimal - in der Vergabe des Präfix - von anderen Funktionen abhängt, fällt die Separation leicht. Nach außen angebotene wichtige Funktionen sind Anfrage einer neuen ID, Abfrage der ID eines Objekts, Abfrage des Objekts zu einer ID und das Setzen des Präfix'.

Die jeweils lokale Verwaltung von Querreferenzen zwischen Repositories kann nicht direkt vom IdentifierModule übernommen werden, da es, wie jedes Modul, nur Zugriff auf sein eigenes Repository hat. Um das Zusammenspiel von mehreren Repositories kümmert sich stattdessen der *IDManager*. Er verwaltet mehrere IdentifierModules und somit mehrere Repositories. Die zentrale Aufgabe des IDManagers ist die Zuordnung von Objekten zu Repositories (beziehungsweise IdentifierModules) und damit die Feststellung, ob es sich bei einer Referenz um eine Querreferenz handelt.

Wie im Konzept in Abschnitt 4.4.1 erläutert, ist es teilweise nötig, bereits zur Vergabe des ID-Präfix mit einer zentralen Instanz zu kommunizieren. Diese Kommunikation sollte jedoch nicht vom IdentifierModule ausgehen, um Abhängigkeiten zu den Kommunikationsmodulen zu vermeiden. Stattdessen ist es möglich das Präfix auf dem IdentifierModule bei Bedarf zu setzen. Ob dies nötig ist, entscheidet die Kommunikationslogik.

Diese befindet sich inklusive der Konfliktdetektion im *ServerModule*, da die Strategien zur Kommunikation und Konflikterkennung eng miteinander zusammenhängen. Die tatsächliche Kommunikation und die dazu notwendige Serialisierung von Daten findet über die Persistenz-Module statt. Die Server-Module entscheiden jedoch über Zeitpunkt und Art der Kommunikation. Welche Server-Module es initial geben soll wird in Abschnitt 5.4 erörtert.

### 5.1.4 Fehlerbehandlung

Während des Einsatzes einer komplexen Bibliothek wie CoObRA treten immer wieder Situationen auf, die im normalen Ablauf der Applikation nicht vorgesehen sind. Dies beginnt mit profanen Dingen, wie nicht gefundenen Dateien, und kann sehr feine Details - wie eine nicht anzuwendende Änderung - betreffen. Weiterhin gibt es unerwartete Ereignisse mit verschieden

schweren Auswirkungen auf den normalen Ablauf. Teilweise können Anomalien erkannt werden, die jedoch den geordneten Ablauf nicht behindern (zum Beispiel eine unerwartete Objekterzeugung während Redo). Andere Vorkommen sind zwar als Fehler einzustufen, der Ablauf kann aber weitergeführt werden (zum Beispiel eine nicht anzuwendende Änderung). Und schließlich gibt es Fehler, die keine weiteren Operationen in diesem Kontext zulassen (zum Beispiel eine nicht gefundene Datei, die geladen werden soll).

Die Behandlung solcher Situationen ist teilweise von vornherein applikationsspezifisch (zum Beispiel Anzeigen von Dialogen), teilweise soll Standardverhalten auch einfach nur angepasst werden. Aus diesem Grund wird die Behandlung an ein *ErrorHandlerModule* delegiert. Dies kann von einer Applikation ausgetauscht werden.

### 5.1.5 Resultierende Architektur

Die aus diesen Überlegungen resultierende Architektur ist in Abbildung 5.2 als Komponentendiagramm dargestellt. Zunächst fällt auf, dass die gestrichelten Pfeile, die Abhängigkeiten zwischen den Modulen anzeigen, nur wenig vorkommen. Dies entspricht den Zielvorgaben. Zu bemerken ist jedoch, dass die Repräsentationsschnittstelle für die Änderungsdaten nicht dargestellt ist, da sie keine Komponente ist. Viele Komponenten hängen natürlich - wie eingangs erläutert - von dieser Schnittstelle ab. Da es sich, wie gesagt, nur um eine Schnittstellendefinition (Interfaces) handelt, ist dies unkritisch.

Abgesehen von den Abhängigkeiten innerhalb der Bibliothek sind außerdem die Abhängigkeiten (beziehungsweise Schnittstellen) zur Applikation dargestellt. Dabei besteht die Applikation aus Sicht der Bibliothek aus einem (oder mehreren) Modell(en), einem Adapter zu diesem Modell für den Schreibzugriff (siehe Abschnitt 5.2) und einer Ablauflogik, die geeignet Bibliotheksaufrufe tätigt, um zum Beispiel Transaktionen zu starten oder

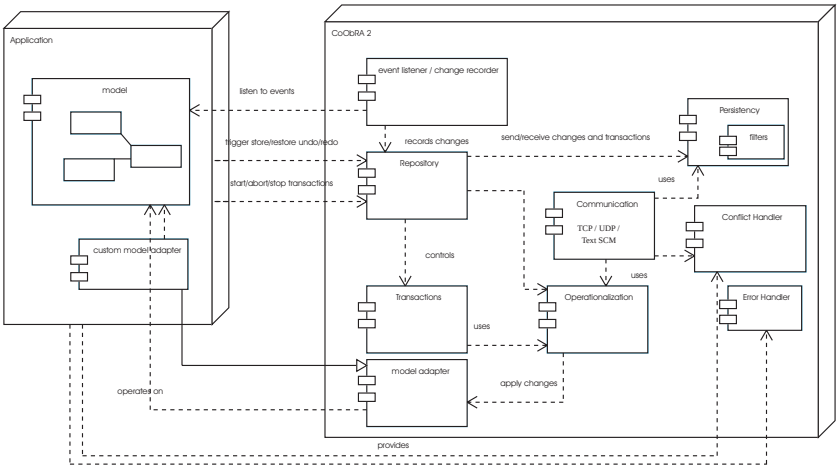


Abbildung 5.2: *Die Software-Architektur von CoObRA 2*

zu beenden. Zusätzlich kann die Applikation Konfliktlösungsstrategien und Fehlerbehandlungsstrategien austauschen. Letzteres gestattet unter anderem die interaktive Konfliktbehandlung.

## 5.2 Abstraktion

Da die CoObRA 2 Bibliothek mit nahezu beliebigen Modellen umgehen soll, ist eine Abstraktion von der Metamodellimplementierung unabdingbar. Die Implementierung der eigenen Bibliothek „Java Feature Abstraction“ wurde daher bei der Entwicklung von CoObRA 2 begonnen. Da diese Abstraktion auch für andere Projekte des Lehrstuhls sinnvoll war, wie zum Beispiel für den Objekt Browser eDOBS [7], den Jar-Import-Mechanismus von FUJABA sowie für die Datenansichten in der Forschungsapplikation OBA [8], wurde eine gemeinsame Lösung entwickelt.

### 5.2.1 Technische Anforderungen

Die wichtigste Anforderung der CoObRA Bibliothek an die Abstraktionsschicht ist der Schreibzugriff auf das Modell. Es müssen Objekte angelegt und gelöscht, Attributwerte gesetzt, sowie Links gezogen und entfernt werden können. Zum Anlegen von Objekten ist es nötig mindestens die Klasse des Objekts zu spezifizieren, zum Anlegen von Links die Assoziation oder Rolle und für das Setzen eines Attributwerts ist das Attribut anzugeben. Daher ist also auch der Zugriff auf das Metamodell unabdingbar. Im Folgenden werden die Details der Anforderungen nach Bibliotheksaufgaben gegliedert gelistet:

- Für den Empfang von Änderungsobjekten für das Protokoll müssen **Felder** (Attribute oder Rollen) eindeutig identifiziert werden können. Die Abstraktionsschicht muss also die Möglichkeit bieten, Handler für Felder in Klassen zu erfragen.

- Für das Speichern von Änderungsdaten ist es nötig die **Klasse eines Objektes** zu erfragen und deren Name (Identifizierer für die Klasse) zu ermitteln. Beim Laden muss dann die **Klasse zu einem Klassennamen** ermittelt werden. (Weiterhin sind zum Speichern natürlich die Objektidentifizierer nötig, diese fallen jedoch nicht in den Aufgabenbereich der Abstraktionsschicht.)

- Um die serialisierten Datenmengen gering zu halten, werden **Rückgabetypen von Feldern** erfragt. In den Persistenzdaten brauchen Datentypen dann nur bei Abweichung vom Feldtyp abgelegt werden.

- Beim Anwenden von Änderungen ist dann der erste Zugriff auf das Modell nötig (bisher nur Metamodell): zu bestimmten Klassen müssen **Objekte erzeugt** werden. Dabei ist zu beachten, ob es sich bei der

Klasse um ein Singleton handelt, denn in diesem Fall wird nur eine Instanz angelegt. Es sollte auch eine API angeboten werden um **Objekte** zu **löschen**, auch wenn dies manche Modelle nicht direkt unterstützen, sondern sich auf die Garbage Collection[1] verlassen. Weiterhin müssen die **Werte beziehungsweise Referenzen in Feldern** gesetzt und entfernt werden.

- Werden im Metamodell **qualifizierte Assoziationen** benutzt muss die Abstraktionsschicht Möglichkeiten anbieten diese zu erkennen und deren Instanzen samt Schlüssel zu setzen.

- Für die Konflikterkennung und Redundanzentfernung ist es weiterhin nötig zwischen zu-1 und zu-n Enden von Assoziationen zu unterscheiden. Es müssen also Informationen über **Kardinalitäten** zu Verfügung stehen.

Alle weiterführenden Bibliotheksfunktionen für Persistenz und Versionierung bauen auf den genannten Basisfunktionen auf und bedingen keine weiteren Anforderungen an die Modell-Abstraktion. Für Kopieren und Einfügen (Abschnitt 6.6) und für die anderen genannten Anwendungen (eDOBS, FUJABA, OBA) sind allerdings weitere Schnittstellen nötig:

- Für die Suche in Objektstrukturen beim Kopieren ist das **Lesen von Feldwerten** beziehungsweise Referenzen notwendig (Modell), sowie das **Auflisten von allen Feldern** einer Klasse (Metamodell). Diese Funktionen werden auch von eDOBS und OBA benutzt, um dem Benutzer Objektstrukturen darzustellen.

[1] In Java ist es der Virtual Machine überlassen Objekte aus dem Speicher zu entfernen, wenn sie nicht mehr erreichbar sind [13]. Einen expliziten Befehl zum Freigeben von Speicher gibt es in Java nicht.

- Weiterhin ist für die property-basierten Entscheidungen beim Kopieren ein Auslesen der Eigenschaften (**composite, aggregation, usage annotation**) von Properties (Rollen) nötig (ebenfalls Metamodell).

- FUJABA nutzt neben den bereits aufgelisteten Informationen noch weitere Metamodellinformationen beim Einlesen von Klassen aus Bytecode: Modifikatoren für Klassen, Methoden und Attribute wie **Sichtbarkeit, Abstract- und Static-Eigenschaft**. **Oberklassen** und -interfaces werden abgefragt. Zu Properties (Rollen) können die jeweiligen **Partner in einer Assoziation** erfragt werden. Felder können mit **Nur-Lesen- und Transient-Eigenschaft** versehen sein.

- Schließlich nutzt eDOBS noch die Möglichkeit **Methoden aufzurufen**.

Diese umfangreichen Anforderungen lassen sich auch ohne Abstraktionsschicht erfüllen, hier wird dies *direkter Zugriff* auf das (Meta-)Modell genannt. Wie im Folgenden dargestellt, ist dies jedoch recht aufwändig.

### 5.2.2 Zugriff auf Modell und Metamodell in Java

Die Java Virtual Machine bietet bereits reflektiven Zugriff auf die Klassen und Objekte einer Java-Anwendung. Es wird jedoch in keiner Weise von Implementierungsdetails abstrahiert. So kann sich beispielsweise für ein Attribut aus dem ursprünglichen Metamodell in einer Klasse der Implementierung nicht nur ein Java-Attribut befinden, sondern auch zwei Zugriffsmethoden. Mit der *Java Reflection API* [25] treten diese drei Artefakte auch gesondert auf und es ist eventuell kein Zusammenhang erkennbar. Das gefundene Java-Attribut kann mit der Reflection API verändert werden, hierzu wird allerdings nicht metamodellkonform die schreibende Zugriffsmethode verwendet, sondern direkt der Wert verändert. Dies ist sogar möglich, wenn

die Sichtbarkeit des Java-Attributs dies eigentlich verbieten würde. Da diese direkte Manipulation zu vielseitigen Problemen in der Applikation führen kann, ist dies nicht akzeptabel.

Die Java Reflection API bietet natürlich die Möglichkeit auch die Zugriffsmethoden aufzurufen, allerdings muss dies explizit geschehen. Auch das Auffinden der Zugriffsmethoden und das Zusammenstellen geeigneter Parameter bleibt dem Aufrufer überlassen. Insbesondere beim Zugriff auf Assoziationen ist die Implementierung jedoch sehr vielfältig und es lässt sich keine allgemeine Strategie angeben, mit der Zugriffsmethoden gefunden und aufgerufen werden können.

Für einige Metamodellimplementierungen stehen andere - teils standardisierte - reflektive Schnittstellen zur Verfügung. So bieten JMI und EMF jeweils Möglichkeiten auf Modell und Metamodell zuzugreifen und abstrahieren dabei schon recht umfassend von der Implementierung. Allerdings muss das begutachtete Modell auch entsprechend ein JMI-Metamodell beziehungsweise MOF-Metamodell sein. Auch diese Ansätze sind daher nicht allgemein genug.

In der hier vorgestellten Modellabstraktionsbibliothek werden für verschiede Metamodellimplementierungen unterschiedliche Module zur Umsetzung der an der Bibliotheksschnittstelle abgegebenen Befehle auf die Schnittstelle des (Meta-)Modells eingesetzt.

### 5.2.3 Abstraktionsschnittstelle

Aus den Anforderungen an die Abstraktionsschicht entwickelte sich die in Abbildung 5.3 dargestellte Schnittstelle für den Zugriff auf Metamodelle und deren Modelle. Die zentralen Klassen sind hier *ClassHandler* und *FeatureHandler*. ClassHandler repräsentiert das Konzept einer Klasse aus dem Metamodell. Demnach kann man eine ClassHandler-Instanz nach dem Namen

der Klasse, die er repräsentiert, fragen und verschiedene andere Eigenschaften ablesen. Ein ClassHandler hat eine Liste von FeatureHandlern. Diese repräsentieren Felder (Properties), Methoden und Konstruktoren, für die es jeweils Unterklassen namens *FieldHandler*, *MethodHandler* und *ConstructorHandler* gibt. Auch diese erlauben es weitere Eigenschaften, wie zum Beispiel den Namen, abzufragen. Die genannten Informationen bezieht die Bibliothek implementierungsabhängig vom Metamodell.

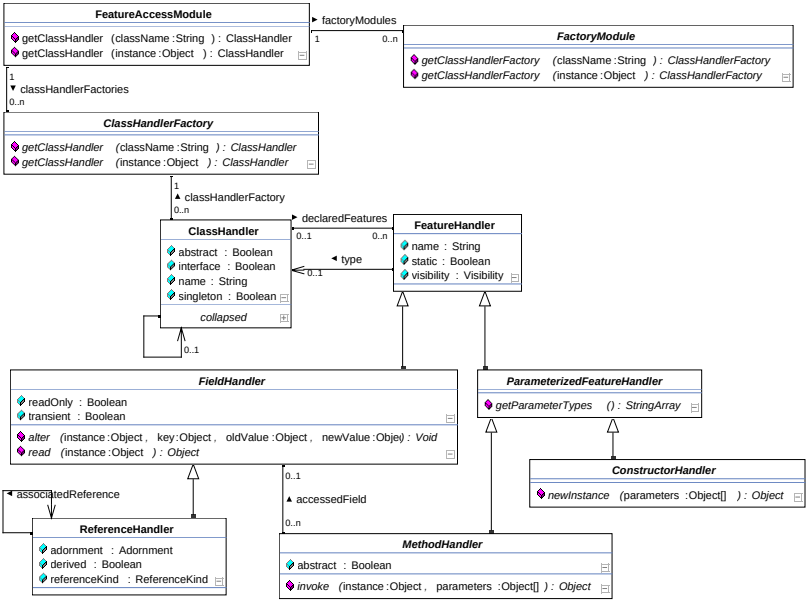


Abbildung 5.3: *Klassendiagramm der Schnittstelle der Java Feature Abstraction*

Es wird nicht explizit markiert, welche Aufrufe auf das Modell zugreifen, es lässt sich jedoch aus deren Beschreibung jeweils ableiten: ClassHandler bieten an, Instanzen über Konstruktoren zu erzeugen (dies erzeugt neue Objekte im Modell) beziehungsweise zu löschen. FieldHandler bieten an

Werte beziehungsweise Referenzen zu lesen und zu verändern, auch dies sind Operationen mit Auswirkungen allein auf das Modell. Das Aufrufen von Methoden über MethodHandler ist ebenso möglich und wird auf Objekten im Modell ausgeführt.

Die ebenfalls in Abbildung 5.3 gezeigte Klasse *FeatureAccessModule* dient dem vereinten Zugriff auf die Handler. Es delegiert die Anfragen nach einem ClassHandler an die *FactoryModule*s, die wiederum *ClassHandlerFactories* anweisen, einen entsprechenden Handler zu erstellen. Diese Delegation wird über eine Chain of Responsibility[2] durchgeführt und ermöglicht so verschiedene Metamodellimplementierungen durch unterschiedliche Implementierungen der Abstraktionsschnittstelle zu verwenden. So kann die Anwendung oder Bibliothek, die die Java Feature Abstraction einsetzt, komplett unabhängig von Modellimplementierungen entwickelt werden.

### 5.2.4 Performanz

Der schnelle Zugriff auf Modell und Metamodell ist insbesondere für CoObRA 2 ein sehr wichtiger Faktor für die Operationsgeschwindigkeit von Speicher- und Ladevorgängen. Daher wurde großer Aufwand betrieben, um die Java Feature Abstraction Bibliothek auch auf Geschwindigkeit zu optimieren. Beim Einsatz verschiedener Profiler (HAT, JProbe und Yourkit) und bei Benchmarks wurden dabei mehrere Arten von Engpässen identifiziert:

- **Suchoperationen**: Nicht nur die Suchoperationen, die selbst zu implementieren waren sondern insbesondere die der ClassLoader[3] aus der

[2] Design Pattern aus [6]

[3] In einer Java VM werden (mit wenigen Ausnahmen) alle Klassen von einem sogenannten ClassLoader in den Speicher geladen. Der reflektive Zugriff auf diese Klassen erfolgt initial ebenfalls über ClassLoader. Selbst die Anfrage nach bereits geladenen Klassen ist in den getesteten Sun VM (1.4 bis 1.6) nicht hoch optimiert.

Java Virtual Machine und anderen Bibliotheken[4] nahmen viel Zeit in Anspruch. Dem wurde weitgehend durch das Caching mit Hashtabellen im FeatureAccessModule begegnet. Auch der reflektive Zugriff auf Properties und Methoden innerhalb der Javaklassen stellte sich als sehr langsam heraus und wurde ähnlich vermieden. Hier mussten allerdings noch Vererbungshierarchien beachtet werden, was die Komplexität der Cachemechanismen in den ClassHandler erhöhte. Der reflektive Zugriff auf EMF[5]-Modelle ist deutlich performanter als die normale Java-Reflection, jedoch konnten auch hier die Caches die Performanz weiter optimieren.

- **Objekterzeugungen**: Weitere Problemstellen im Quellcode, die im Profiler auffielen, waren diverse Objekterzeugungen. Insbesondere die Erzeugung von Zeichenketten (java.lang.String) schlug mit hohem Aufwand zu Buche. Nicht nur die Erzeugung der diversen Hilfsobjekte sondern auch deren Löschung durch den Garbage Collector der Virtual Machine kostete viel Zeit. Durch die Operationen des Garbage Collectors wurde sogar die komplette Virtuelle Maschine kurzzeitig angehalten, was im echtzeitnahen Bereich inakzeptabel ist. Um die Erzeugung von Zeichenketten zu vermeiden wurden die Klassen zur Serialisierung von Daten umgestellt, so dass deren Methoden nicht länger Zeichenketten als Rückgabewerte lieferten, sondern die Resultate direkt in Datenströme schrieben. Eine weitere Optimierungsmöglichkeit bezüglich Objekterzeugungen wurde für Metamodelle identifiziert: Haben Methoden primitive Rückgabewerte (boolean, int, float et cetera) werden beim reflektiven Aufruf Hilfsobjekte erzeugt, die

[4] Die FUJABA RuntimeTools bringen spezielle ClassLoader für das FUJABA Plugin Konzept mit - ebenso die Eclipse-Umgebung.

[5] Metamodellimplementierung aus dem Umkreis des Eclipse-Projektes

diese Rückgabewerte verpacken. Besonders Zugriffsmethoden werden von CoObRA häufig über die Abstraktionsschicht reflektiv aufgerufen. Der Entwickler eines Modells kann also sein Modell optimieren indem Zugriffsmethoden keine unentbehrlichen Werte zurückliefern.

- **Reflektive Methodenaufrufe**: Allgemein sind die reflektiven Aufrufe von Methoden recht teuer. Diesen Umstand hat Sun - Hersteller der verbreitetsten Java Virtual Machine - ebenfalls erkannt und diese Aufrufe in Java 5 (JRE1.5) um einiges beschleunigt. Im Zuge der Optimierungen wurde jedoch noch eine Möglichkeit gefunden die Aufrufe auf einem anderen Weg zu tätigen, als über den noch immer langsamen reflektiven Mechanismus: Mittels der Bibliothek ASM[6] wurden dynamisch Java-Klassen generiert, die einen MethodHandler implementierten. Der einzige Zweck dieses Handlers war der direkte Aufruf der ihm zugeordneten Methode. Diese Art der dynamischen Erstellung von maßgeschneiderten Handlern beschleunigte den Methodenaufruf fast um den Faktor 2.

Neben den genannten großen Kategorien von Optimierungen wurden mit den Profilern natürlich weitere kleine Engpässe entfernt, die aber hier nicht näher beleutet werden sollen, da sie nur jeweils einen kleinen Teil zur Optimierung beitrugen. Als Resultat der kompletten Optimierung der Abstraktionsschicht wurden die in den Beispielanwendungen gemessenen Ladezeiten von CoObRA Dateien um den Faktor 10 verkürzt. Dazu trug auch die bereits kurz oben erwähnte Optimierung bei der Serialisierung bei, die nicht teil der Abstraktionsschicht sondern der Persistenzmodule ist. Da die Persistenzmodule eine zentrale Rolle bezüglich Funktion und Flexibilität einnehmen wird ihnen das nun folgende Unterkapitel gewidmet.

[6] ASM dient der direkten Bearbeitung von Java Bytecode. [18]

## 5.3 Persistenzmodule

Persistenzmodule bieten, wie bereits beschrieben, eine Schnittstelle zum Lesen und Schreiben von Änderungsdaten samt Navigation auf diesen Daten. Die Schnittstelle des Persistenzmoduls schreibt dabei kein Medium vor: Quelle beziehungsweise Ziel dieser Lese- und Schreiboperationen kann zum Beispiel der Arbeitsspeicher, eine Datei, ein Datenstrom oder aber auch ein anderes Persistenzmodul sein. Wie in Abbildung 5.4 dargestellt, bietet die Schnittstelle einerseits Methoden zum Öffnen (open), Schließen (close), komplett Schreiben (flush) und Entfernen (delete) des Moduls. Weiterhin bietet sie mehrere Methoden zum Lesen von Änderungsdaten und zum Navigieren auf den Daten (receive). Konkret kann die erste Änderung gelesen werden und die jeweils nächste beziehungsweise vorherige Änderung zu einer bekannten Änderung. Beim Lesen der nächsten oder vorherigen Änderung kann außerdem ein Filter angegeben werden, um beispielsweise nur Änderungen an einer bestimmten Instanz aus dem Modell auszulesen. Schließlich gibt es noch eine Methode zum sequenziellen Schreiben von Änderungsdaten (send). Diese Methode gibt, aus unten näher erläuterten Gründen, eventuell ein neues Änderungsdatenobjekt zurück, auf dass man sich später beziehen kann, um auf den Daten zu navigieren (Referenzparameter für die lesenden Methoden).

Dieser Ansatz hat sich als sehr flexibel bewährt und gestattet interessante Möglichkeiten. Die Anzahl der in Abbildung 5.4 dargestellten Module lässt bereits erkennen, dass davon intensiv Gebrauch gemacht wurde. Im Folgenden werden nun einige Implementierungen der Persistenzmodulschnittstelle vorgestellt, um die Einsatzmöglichkeiten zu umreißen.

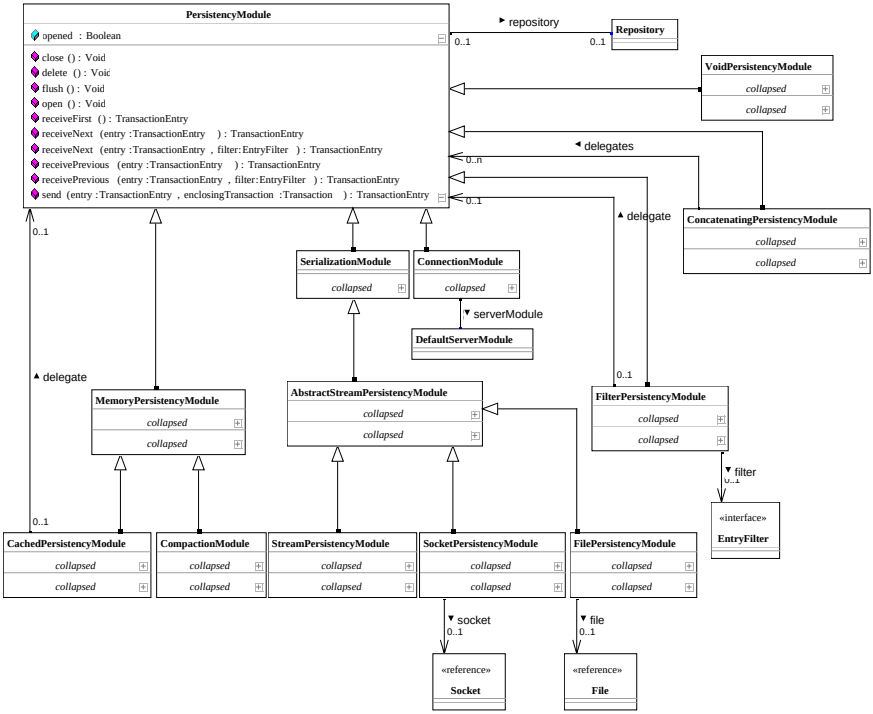


Abbildung 5.4: *Implementierte Persistenz Module*

### 5.3.1 Arbeitsspeicher

Das Vorhalten der Änderungsdaten im Arbeitsspeicher übernimmt das *MemoryPersistencyModule*. Dabei handelt es sich um die denkbar einfachste Speichermöglichkeit. Um einem Persistenzmodul mehr Freiheiten bei der Speicherung der Daten und der Navigation darauf zu geben, muss eine Modulinstanz nur solche Änderungsobjekte als Filter- beziehungsweise Referenzparamter akzeptieren, die es selbst erzeugt hat. Dies erlaubt schon beim MemoryPersistencyModule einen einfachen Kniff: Es wird für die Datenhaltung eine neue Klasse verwendet, die außer den eigentlichen Änderungsdaten auch noch Zeiger auf die vorherige und die nächste Änderung hält. So wird direkt mit den Änderungsdatenobjekten eine verkettete Liste zur Navigation auf denselben erzeugt.

### 5.3.2 Datenströme

Das *StreamPersistencyModule* hat im Vergleich zum MemoryPersistency-Module umfangreichere Aufgaben zu erfüllen. Um Änderungsdaten in einen Strom zu serialisieren, müssen zunächst die Werte aus einem Änderungsobjekt (Change) serialisiert werden. Dazu werden Objektreferenzen durch IDs ersetzt und primitive Datentypen konvertiert. Weiterhin müssen Objekte, die nicht eigentliche Instanzen des Modells sind (zum Beispiel Datenklassen des JDK - Point, Dimension, et cetera), serialisiert werden. Der größte Teil der dazu nötigen Schritte wird an das IdentifierModule (IDs) beziehungsweise die Abstraktionsschicht (konvertieren, serialisieren von Werten) delegiert. Die serialisierten Werte müssen dann in einem geeigneten Datenformat in den Datenstrom geschrieben werden, um vollständige Änderungsdaten zu repräsentieren.

Die offensichtlichste und wichtigste Anforderung ist dabei, dass die Änderungsinformationen aus dem Datenstrom nach Versand oder Speicherung

vollständig wieder hergestellt werden können. Bei der Ausarbeitung des Datenformats fließen aber auch die eingangs beschriebenen Anforderungen ein: Die Daten sollen mit wenig Zeit- beziehungsweise Verarbeitungsaufwand gelesen und geschrieben werden; und ein zeilenbasiertes Textformat ist erforderlich, wenn Konflikterkennung in textbasierten SCM-Systemem möglich sein soll. Es wurde daher zunächst ein Format entwickelt, das auch der letztgenannten Anforderung genügt und auf Geschwindigkeit soweit wie möglich optimiert ist.

Wenn Informationen schnell verarbeitet werden sollen, ist die erste Devise, die Daten möglichst kompakt abzulegen und somit das Datenvolumen zu reduzieren. Rechenaufwändige Algorithmen, die das Datenvolumen weiter reduzieren, können jedoch u.U. von Nachteil sein, da auch die Rechenkapazität (nicht nur die Ein-/Ausgabegeschwindigkeit) beschränkt ist. Es wurde daher ein Datenformat mit einfachen Trenn- und Markierungszeichen gewählt - ohne Kompression. Bei höheren Anforderungen an die Kompaktheit der Daten und mehr verfügbarem Speicher kann ein Kompressionsalgorithmus nachgeschaltet werden (GZip oder Zip). Eine andere Alternative, die weitere Ressourcen spart, wäre ein reines Binärformat. Die Informationsdichte wäre noch weiter steigerbar, ohne mehr Rechenzeit aufwenden zu müssen. Allerdings kann es nicht zum Speichern in Text-SCM-Systemen verwendet werden und wurde daher zurückgestellt. Eine vollständige Beschreibung des Datenformats *CTR* befindet sich im Anhang.

Auch die Kommunikation (zum Beispiel über ein Netzwerk) zwischen Applikationen - die Distribution und Synchronisation derer Modelle - wird über Ströme abgewickelt. Das *ConnectionModule* erweitert dabei die Schnittstelle für das Empfangen und Senden der Änderungsdaten um die Verbindungs- und Kommunikationslogik (zum Beispiel über TCP) und delegiert die sonstigen Aufgaben an das StreamPersistencyModule.

Die Navigation auf den Änderungen beschränkt sich bei reinen Datenströmen (zum Beispiel Empfang über TCP/IP oder lesen von einer URL) auf sequenzielles Lesen. Aus diesem Grund reicht es für ein StreamPersistencyModule sich die zuletzt gelesene Änderung zu merken. Wird für eine Änderung dann die nächste Änderung angefordert, kann das Modul überprüfen, ob tatsächlich die nächste Änderung aus dem Strom gelesen werden soll. Wird eine Änderung vor der zuletzt gelesenen angefordert, schlägt die Operation fehl, da man in Strömen nicht rückwärts navigieren kann.

### 5.3.3 Dateien

Im Gegensatz zu reinen Strömen kann man in Dateien beliebig navigieren. Diese Flexibilität soll daher natürlich auch das *FilePersistencyModule* nach außen zur Verfügung stellen. Um dies zu ermöglichen wurde erneut von der Möglichkeit Gebrauch gemacht, dass ein Persistenzmodul eigene Objekte zur Verwaltung der Änderungen einsetzen kann. Eine Unterklasse von Change wird verwendet, um die Position einer Änderung in der Datei mit zu speichern. Wird dann eine Navigationsoperation relativ zu einer Änderung ausgelöst, kann zunächst zu der darin vermerkten Dateiposition navigiert werden.

Das Feststellen der aktuellen Dateiposition stellte sich allerdings als nicht ganz trivial heraus: Beim Schreiben und Lesen von Textdateien in Java verwendet man gängigerweise Klassen, die vom Kodieren und Dekodieren der Texte in entsprechende Zeichensätze abstrahieren (Writer und Reader). Bei manchen Zeichensätzen (insb. UTF-8, UTF-16) sind einige Zeichen (oder alle) aber nicht genau ein Byte lang, sondern mehrere Bytes. Die Dateiposition wird jedoch in Bytes nicht in Zeichen angegeben. Weiterhin stimmt die aktuelle Dateiposition, die man vom Betriebssystem erfragen kann, nicht mit der des aktuell ausgelesenen (beziehungsweise geschriebenen) Zeichens über-

ein, da Lese- und Schreiboperationen aus Geschwindigkeitsgründen gepuffert werden. Um dieses Problem im FilePersistencyModule zu lösen, wurden zwischen die abstrahierenden Reader (beziehungsweise Writer) und den Zwischenspeichermechanismus jeweils zählende Zwischenschichten eingebaut.

### 5.3.4 Cache

Lädt eine Applikation ein Modell aus einem Strom (zum Beispiel über ein Netzwerk) kann es trotzdem erforderlich sein, auf den Änderungsdaten zu navigieren, um beispielsweise die letzte Änderung wieder rückgängig zu machen. Ein Strom unterstützt allerdings (meist) keine Operation um Daten erneut zu lesen, somit stellt ein StreamPersistencyModule wie oben erwähnt diese Navigation nicht. Doch auch hier kann man im Zuge einer Persistenzmodulimplementierung Abhilfe schaffen: Das *CachedPersistencyModule* delegiert zunächst an ein anderes Persistenzmodul (zum Beispiel das bereits erwähnte StreamPersistencyModule). Die vom Delegaten zurückgelieferten Änderungsdaten werden jedoch (in gewissem Rahmen) vom CachedPersistencyModule im Speicher gehalten. Wird nun eine bereits gelesene Änderung angefordert, werden die zwischengespeicherten Daten verwendet, anstatt zu delegieren. Dies ermöglicht einen flexiblen Umgang mit ansonsten eingeschränkten Modulen, ohne dass der Speicher unnötig belastet wird.

### 5.3.5 Filter

Filter können an ein Persistenzmodul übergeben werden, um die als nächstes zu lesende Änderung näher zu spezifizieren. Soll allerdings die Änderungsliste nicht vom aufrufenden Code gefiltert werden, sondern von außerhalb, helfen die Filter-Parameter allein nicht mehr. Beispielsweise sollen beim Laden einer Datei bestimmte Arten von Modellelementen nicht mitgeladen werden. Dazu soll natürlich nicht der Lademechanismus in der CoObRA

Bibliothek angepasst werden, der Filter an das Persistenzmodul übergibt. Stattdessen hat die Applikation die Möglichkeit ein *FilterPersistencyModule* einzusetzen. Auch dieses delegiert die eigentlichen Aufrufe an an anderes Persistenzmodul, jedoch wird ein zusätzlicher Filter angewandt, den man beim instanzieren des FilterPersistencyModules spezifizieren kann.

### 5.3.6 Redundanzentfernung

Auch die im Konzept-Kapitel diskutierte Redundanzentfernung kann in einem Persistenzmodul realisiert werden. Dieses kann wie ein Filter transparent vor das übliche PersistenzModul geschaltet werden, um Redundanz aus den Änderungslisten beim Lesen oder Schreiben zu entfernen. Dies ermöglicht - insbesondere beim Client-Server-Betrieb - das Datenvolumen auf unkomplizierte Weise zu reduzieren. Und auch beim Speichern in Datei kann auf diese Weise leicht zwischen kompletter Historie und redundanzfreier letzter Version gewählt werden. Details zur Implementierung der eigentlichen Redundanzentfernung finden sich im Abschnitt 5.6.

### 5.3.7 Konkatenation

Im CASE-Tool FUJABA werden geöffnete Projekte in einem sogenannten *Workspace*-Verzeichnis gehalten. Dieses enthält jedoch nicht komplette Projektdateien, sondern nur die Änderungen, die seit dem Öffnen des Projekts aufgetreten sind. Das bedeutet, dass zum Laden eines Projekts aus dem Workspace zunächst die Originaldatei aus dem Dateisystem geladen werden muss und danach die Änderungen aus dem Workspace angewandt werden. Um dies für CoObRA transparent zu machen, wurde das *ConcatenatingPersistencyModule* eingeführt. Dieses vereinigt mehrere Persistenzmodule (durch Konkatenation) zu einem einzigen. Beim Lesen aus dem ConcatenatingPersistencyModule werden die Aufrufe zunächst an das

erste Delegat weitergeleitet, solange dieses noch Änderungen liefert. Sind alle Änderungen aus dem ersten Delegat gelesen, wird an das zweite Persistenzmodul delegiert. In FUJABA kann dem Repository also schlicht ein ConcatenatingPersistencyModule übergeben werden, das zunächst an ein Persistenzmodul für die originale Projektdatei delegiert und danach an ein Modul für die Workspace-Datei.

## 5.4 Server Module

Die Kommunikation und die Auswahl der Mischstrategien wird in den sogenannten *ServerModule*n realisiert. Jeweils eines dieser Module ist einem Repository zugeordnet und bietet eine Schnittstelle für Operationen wie *update*, *commit* und *connect* an. Die Kommunikation wird von den bisher geschriebenen Server-Modulen erneut an ein Kommunikationsmodul delegiert.

### 5.4.1 Standardimplementierung

Als erstes Kommunikationsmodul entstand eine Anbindung an das TCP/IP-Protokoll: Mit serialisierten Befehlsobjekten können über dieses Modul Anfragen zwischen einem Client und einem Server ausgetauscht werden. Nach einem Befehl können noch Binärdaten, wie zum Beispiel Änderungsprotokolle gesendet werden.

Diese Kommunikationsform dient der Standardimplementierung eines Server-Moduls. Dieses erlaubt die Durchführung der als Client-Server-Strategien vorgestellten Synchronisationsarten. Dabei arbeitet sie mit einem proprietären modellunabhängigen CoObRA-Server zusammen: Die Anfragen und Änderungsprotokolle werden vom Server empfangen und verwaltet, ohne dass dafür das Metamodell der entsprechenden Applikation benötigt wird.

Um die Handhabung der Änderungsdaten nicht eigens für den Server abändern zu müssen, arbeitet auch dieser mit der Abstraktionsschicht. Allerdings mit einer speziellen Implementierung, die nicht versucht die Klassennamen auf Java-Klassen abzubilden oder tatsächlich Daten zu deserialisieren. Stattdessen werden nur Platzhalter (NonResolvingClassHandler) als Klassen verwendet und die Werte werden als Texte beziehungsweise Binärdaten beibehalten.

### 5.4.2 Echtzeitimplementierung

Bei echtzeitnahen Simulationen und Spielen sind die Anforderungen - wie eingangs dargestellt - anders gelagert: Es kommt mehr auf die effektive zeitnahe Synchronisation an, als auf Konfliktlösungsstrategien. Schon die Kommunikation über das TCP-Protokoll ist hier nicht mehr die beste Wahl:

Da in Netzwerken mitunter Datenpakete verloren gehen, sieht das TCP-Protokoll vor, in diesem Fall die Daten erneut zu übertragen und auf Empfängerseite auf die Ankunft des fehlenden Paketes zu warten - notfalls wird dieser Vorgang mehrfach wiederholt. Verluste von Datenpaketen sorgen so mit TCP für Wartezeiten, die um mehr als das Doppelte höher sind, als die End-zu-End-Latenz der unterliegenden Schicht.

Das UDP-Protokoll dagegen stellt das Fehlen von Paketen nicht fest und versucht auch nicht die Reihenfolge der Pakete auf Seiten des Empfängers zu korrigieren, sollte diese bei der Übertragung verändert werden[7]. Daraus resultieren einerseits geringere Latenzzeiten für die Kommunikation, andererseits aber auch eine Ungewissheit über die versandten Daten beim Empfänger.

[7] Da nicht unbedingt alle Pakete dieselbe Route zum Empfänger nehmen, ist es möglich, dass einige Pakete, die später versendet wurden, vor Paketen ankommen, die früher versendet wurden.

Um diesem Unterschied gerecht zu werden, ist ein neues Kommunikationsmodul für CoObRA 2 entstanden. Dieses sendet auftretende Änderungen sofort an alle verbundenen Kommunikationspartner. Beim Empfang von Änderungsdaten quittiert es diese mit einer Nachricht an den Absender. Auf Seiten des Absenders werden alle quittierten Änderungsdaten pro Kommunikationspartner markiert. Sind versandte Änderungsdaten nach einer Zeitüberschreitung noch nicht quittiert, werden diese erneut versendet. Hierbei wird allerdings der Vorteil genutzt, dass CoObRA nähere Informationen über die Bedeutung der Daten hat: Mittlerweile nicht mehr aktuelle Änderungen werden nicht erneut versandt, sondern von der Redundanzentfernung ausgefiltert.

Die Einsatzmöglichkeiten dieser UDP-basierten Server-Module werden in Abschnitt 6.5 noch im Detail ausgeführt.

### 5.4.3 SCM Integration

Auch die Integration in etablierte Source-Code-Management-Systeme wie CVS oder Subversion wird von einem Server-Modul organisiert. Hier findet eine „Kommunikation“ zwischen CoObRA und dem Dateisystem statt. Die eigentliche Datenübertragung führt jedoch die entsprechende SCM-Software durch. Nachdem das externe SCM-Programm neue Daten in die Datei geschrieben hat, kontrolliert das SCM-Server-Modul die Datei auf Konflikte und teilt diese der Applikation mit.

Die Dateien, die für die externe SCM-Software auf der Festplatte abgelegt werden, enthalten - entsprechend der Anforderungen - nur Text und keine Binärdaten. Weiterhin sind sie zeilenbasiert, um die Konfliktdetektion und Deltaberechnung in der externen Software nicht auszuhebeln. Wird eine Datei von zwei Benutzern verändert, erkennt daher das SCM-System dies als textzeilenbasierten Konflikt und markiert diesen in der Datei mit speziellen

Markierungszeilen.

Diese Markierungen schließen je eine Änderungsliste von einem der Benutzer (beziehungsweise Applikationsinstanzen) ein. CoObRA's SCM Server-Modul kann daraufhin die Änderungslisten mit den gehabten Verfahren mischen und feingranular Konflikte detektieren. Erst diese werden der Applikation mitgeteilt. Für den Benutzer ist so das Verhalten mit externem SCM-Management analog zu dem proprietären CoObRA-Server mit dem Standard Server-Modul.

## 5.5 Integrationsaufwand

Der Integrationsaufwand für einen Anwendungsentwickler, der CoObRA verwenden möchte, sollte möglichst gering sein. Dazu trägt vor allem die Abstraktionsschicht einen entscheidenden Teil bei. Wird in der Applikation ein standardgerechtes Metamodell verwendet, wie zum Beispiel ein JMI- oder EMF-kompatibles, können die bereits vorhandenen Implementierungen der Java Feature Abstraction benutzt werden. Auch für von FUJABA generierten Code und für die sogenannte JavaBeans Konvention sind Standardimplementierungen vorhanden, so dass mit all diesen Metamodellimplementierungen kein Arbeitsaufwand für die Integration der Modelle anfällt. Sollte ein Mapping des Metamodells auf Java-Klassen verwendet werden dass in all diese Kategorien nicht passt, muss eine neue Adapterschicht programmiert werden. Dies ist jedoch erfahrungsgemäß auch innerhalb weniger Stunden zu erreichen und hängt nicht von der Größe des Softwaresystems ab.

Die wichtigste Entscheidung, die ein Anwendungsentwickler treffen muss, wenn er CoObRA einsetzt ist, für welches lokale Repository ein neues Objekt angemeldet werden soll: In einem Repository sollten sich all die Objekte

befinden, die konzeptionell zu *einem* Modell gehören und zum Beispiel gemeinsam in einer Datei abgespeichert werden. Für kleine Anwendungen ist diese Entscheidung leicht, da schlicht pro Anwendungsinstanz nur ein Repository verwendet wird. Bei größeren Anwendungen gibt es jedoch eventuell mehrere Dateien, Projekte oder andere Dokumente, mit denen ein Anwender arbeitet und so ist auch die Unterteilung in Repositories auf der Seite der Bibliothek notwendig. Da CoObRA Änderungen protokolliert, ist bereits bei Erzeugung des Objekts (bevor Attribute gesetzt oder Kanten gezogen werden) zu entscheiden, welches Repository die Änderungen protokolliert. Dies ist bei vielen Anwendungen leicht zu realisieren, es gibt jedoch Fälle wo die Anwendung zu diesen Zeitpunkten die Zuordnung noch nicht machen kann. Hier steht der Anwendungsentwickler eventuell vor einem konzeptionellen Problem.

Auch hier gibt es Abhilfemöglichkeiten: So können zum Beispiel die Änderungsprotokolle so lange nur temporär gespeichert werden (zum Beispiel in einem Repository pro Objekt), bis die Zuordnung zu dem finalen Repository vorgenommen werden kann. Eine andere Möglichkeit ist, die Änderungsliste im Nachhinein durch Inspizieren der Objekte (statt durch Protokollieren) zu erstellen, wenn sie dem Repository zugeordnet werden.

Wie auch für die Abstraktion vom Zugriff auf das Modell, gibt es für das Protokollieren von Änderungen an erzeugten Objekten schon vorgegebene Implementierungen, die PropertyChangeEvents, den EMF Model Adapter beziehungsweise die MOF/JMI Listener benutzen, um Änderungen am Modell zu bemerken. Diese werden dann automatisch in CoObRA Changes umgewandelt und protokolliert.

Manche Metamodellimplementierungen (zum Beispiel JavaBeans) bieten weiterhin keine Möglichkeit Objekte zu löschen oder schreiben die Löschung der Objekte nicht explizit vor (FUJABA). In diesen Fällen bietet die

CoObRA-Bibliothek die Möglichkeit die Löschung der Objekte automatisch durch den Garbage-Collector zu detektieren. Hier entsteht also im Zweifelsfall kein zusätzlicher Integrationsaufwand beim Verwenden der Bibliothek.

Die sogenannte „Loading on Demand"-Funktionalität wurde bisher nicht eingehend getestet, da in den größeren Anwendungen aus den Testszenarien meist ein komplettes Modell für übliche Operationen benötigt wurde. Es lässt sich aber festhalten, dass die Anforderungen an die Metamodellimplementierung für das inkrementelle Laden eines Modells deutlich höher sind als für alle anderen Funktionalitäten, da das Detektieren von Lesezugriffen auf das Modell erforderlich ist.

Insgesamt hält sich der Integrationsaufwand für die üblichen Funktionen (abgesehen vom inkrementellen Laden) in Grenzen oder ist gar vernachlässigbar, wenn bereits unterstützte Metamodelle verwendet werden. Der tatsächliche Aufwand für einige konkrete Projekte ist im Kapitel 6 dargestellt.

## 5.6 Redundanzentfernung

Wie bereits im Konzept erläutert, enthält eine Liste von Änderungen oft mehr Informationen als nötig sind, um ein Delta zwischen zwei Versionen zu beschreiben. Wird also aus einer Änderungsliste nur die Information benötigt, wie sich ein Modell vor Anwenden der kompletten Liste von dem Modell nach der Anwendung unterscheidet, kann man einzelne Deltas aus der Liste entfernen und so die Liste verkürzen, ohne wichtige Informationen zu verlieren. In Abschnitt 4.2.6 wurde bereits vorgestellt, wie diese Verkürzung theoretisch durchgeführt werden kann. Die effiziente Umsetzung dieses Verfahrens als Persistenzmodul wird nun dargestellt.

Soll die Redundanz aus einer Änderungsliste entfernt werden, die auf ein

leeres Modell angewendet wird, böte sich die Möglichkeit an, die Liste anzuwenden und danach eine neue Liste aus dem entstandenen Modell zu generieren, indem alle Objekte und Kanten aufgelistet werden. Dies ist jedoch nicht möglich, wenn die Änderungsliste ein Delta zwischen zwei nichtleeren Modellen beschreibt. Weiterhin ist das komplette Anlegen des Modells meist unerwünscht.

Die durch die theoretische Betrachtung suggerierte Lösung, immer zwei Änderungen zu suchen, die sich gegenseitig aufheben, wäre mit quadratischem Aufwand verbunden und sollte deswegen nicht naiv implementiert werden. Stattdessen wird die Liste von Änderungen linear abgearbeitet, während alle Änderungen sortiert und in Hashtabellen abgelegt werden. Objekterzeugungen und -löschungen werden nach Objekt abgelegt, Linkerzeugungen und -löschungen nach dem Tupel Objekt-Assoziationsname. Wird eine Änderung bei einem bereits vergebenen Hashwert abgelegt, handelt es sich um eine potentielle Redundanz. Der Aufwand ist so nur linear zur Anzahl der Änderungen in der Liste. Auch die Realisierung als Persistenzmodul, das Änderungen nacheinander empfängt, wird so vereinfacht.

Wurden zwei Änderungen gefunden, die eine Redundanz beinhalten können, in der Regel beide verworfen werden. Für die besonderen Änderungen von zu-1 Kanten, bei denen teilweise das Löschen des einen Links und das Erzeugen des neuen in einem Änderungsobjekt gespeichert sind, werden die zwei teilweise redundanten Änderungsobjekte zu einem einzigen zusammengefasst: Das resultierende Änderungsobjekt beinhaltet den gelöschten Link der ersten Änderung und den erzeugten der zweiten.

Da bei vielen Metamodellimplementierungen Attribute vor dem Löschen eines Objekts nicht explizit gelöscht werden und teilweise auch Links nicht komplett entfernt werden, werden beim Verwerfen des Erzeugens und Löschens eines Objekts auch alle weiteren Änderungen verworfen, die dieses

Objekt betreffen (und zwischen dem Erzeugen und Löschen des Objekts liegen).

Von einem transparenten Persistenzmodul wäre zu erwarten, die Änderungen an ein Delegat möglichst bald weiterzugeben. Bei der Redundanzentfernung kann allerdings jede weitere empfangene Änderung Auswirkungen auf die erste Änderung haben. Daher wird keine Änderungsinformation weitergereicht solange noch empfangen wird. Erst bei Abschluss des Empfangsvorgangs werden die Änderungen an das Delegat übertragen. Diese Eigenschaft kann auch benutzt werden, um zwischen Versionen zu trennen, die nicht miteinander verschmolzen werden sollen.

# 6 Einsatz der Bibliothek

Um CoObRA 2 in der Praxis zu erproben und Konzept- und Implementierungsalternativen zu evaluieren wurde die Bibliothek in diversen Softwareprojekten bereits eingesetzt. Die umfangreichsten Projekte waren dabei das CASE-Tool FUJABA, das im Fachgebiet von Prof. Zündorf mitentwickelt wird, und das MOFLON Projekt, das in Darmstadt im Fachgebiet von Prof. Schürr entwickelt wird. Auch in einem industrienahen Forschungsprojekt für die Volkswagen AG, die Siemens AG und die Volkswagen-Bordnetze-GmbH (jetzt Sumitomo) namens OBA ist CoObRA für Persistenz und Verteilung zuständig. Weitere kleine Testprojekte werden im Anschluss an diese kurz dargestellt.

## 6.1 FUJABA

Um die prinzipielle Eignung des Konzepts einer Persistenz- und Verteilungsbibliothek auf Basis von Änderungsprotokollen zu zeigen, wurde bereits zu Beginn der Arbeiten zu dieser Dissertation die CoObRA 1 Bibliothek in Fujaba 4 integriert. An diesem Beispiel lässt sich der Aufwand, der für die Integration von CoObRA in eine große vorhandene Anwendung zu erwarten ist, sehr gut beschreiben. Nach der rudimentären Anbindung an das Metamodell von FUJABA traten jedoch schon beim Laden von Dateien diverse Anomalien auf. Die Gründe hierfür waren Nebeneffekte in den Zugriffsmethoden und vor allem in den Konstruktoren der Klassen des Modells. So

wurden beispielsweise beim Anlegen von Objekten weitere Objekte erzeugt, die gleichzeitig auch von dem Änderungsprotokoll erzeugt wurden[1]. Dies resultierte in doppelt vorhandenen Objekten. Teilweise konnten wegen der Nebeneffekte und zusätzlichen Zugriffmethoden die Attribute von manchen Objekten nicht wie gewünscht vom Lademechanismus gesetzt werden.

Weiterhin fehlten im Metamodell an einigen Attributen und Rollen die Aufrufe für das Versenden von Änderungsereignissen. Diese wurden bisher in FUJABA nur zum Darstellen der Diagramme verwendet. Für CoObRA wurden diese genutzt, um das Änderungsprotokoll zu erstellen. An einigen Attributen wurden gar zusätzliche Ereignisse erzeugt, die nicht auf ein tatsächliches Attribut zurückzuführen waren. Im Änderungsprotokoll waren dadurch Änderungen an nicht-existenten Attributen zu finden, die herausgefiltert werden mussten.

Weiteren Aufwand erzeugte das Plugin-Konzept von FUJABA: In dem CASE-Tool können sich Plugins registrieren, um zusätzliche Funktionalität bereitzustellen, die nicht im Kern der Anwendung verfügbar ist. Diese Erweiterungen liegen in einem Unterverzeichnis der Anwendungsinstallation und werden nach dem Start des Werkzeugs mit speziellen ClassLoadern[2] geladen. Durch die Eigenschaften der Java Virtual Machine ist es in diesem Fall nötig, für das Anlegen von Instanzen dieser Klassen explizit auf die speziellen ClassLoader zuzugreifen. Enthält ein Plugin auch neue Metamodellelemente, benötigt der CoObRA-Mechanismus also Zugriff auf die korrekten Klassen und damit auf die ClassLoader der Plugins. Hierfür musste das noch recht unflexible CoObRA 1 erst erweitert werden.

Trotz der zahlreichen zu behebenden Anomalien im Metamodell war die Integration und die Programmierung einer Benutzeroberfläche innerhalb ei-

[1] Der bisher von FUJABA verwendete Lademechanismus schrieb direkt in die Java-Attribute und hatte daher diese Probleme nicht.

[2] Diese bieten in Java die Möglichkeit Klassen aus Dateien nachzuladen.

ner Woche soweit abgeschlossen, dass die neuen Funktionen in Tests eingesetzt werden konnten. Die so entstandene Fujaba-Version (Screenshot in Abbildung 6.1) wurde im Anschluss in Übungen mit 40 bis 80 Studenten getestet und nach einem Semester als stabil befunden. Die Handhabbarkeit und Wartbarkeit des FUJABA-Modells wurde durch die Bereinigung von Nebeneffekten und unerwartetem Verhalten im Zuge der Integration ebenfalls verbessert.

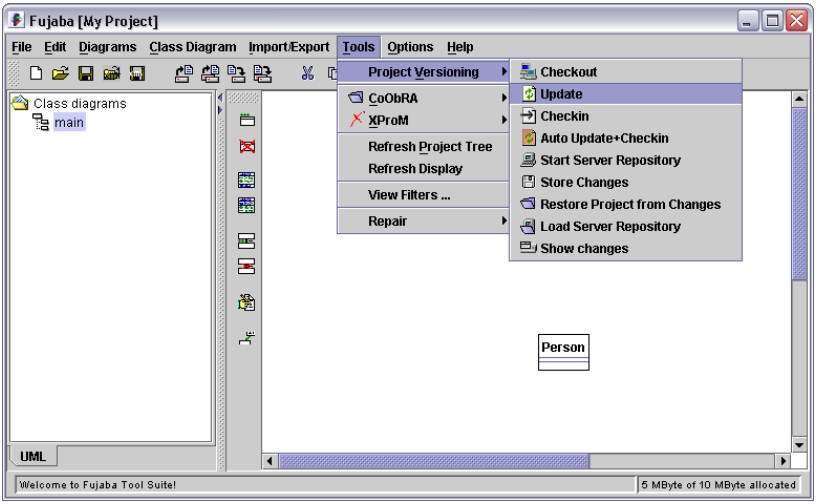


Abbildung 6.1: *Screenshot von FUJABA 4 mit CoObRA 1*

Neben der gewonnenen Erfahrung mit dem Mehrbenutzerbetrieb, die weiter unten in diesem Abschnitt dargestellt wird, trat bei den Tests ein sehr allgemeines Problem zu Tage: Viele Operationen waren zu langsam. Insbesondere das Laden aus Datei und das Herunterladen vom Versionierungsserver nahm viel Zeit in Anspruch. Die Hauptgründe für die langen Ladezeiten waren das verwendete Dateiformat (XML) und der Modellzugriff direkt über Java-Reflection ohne Caching. Dies wurde daher - wie beschrieben - in

CoObRA 2 deutlich verbessert.

### 6.1.1 FUJABA 5

Mit dem Entstehen von FUJABA 5 war auch CoObRA 2 einsatzbereit und wurde innerhalb weniger Stunden in das Werkzeug integriert. Der bereits erwähnte Einsatz von Profilern fand zum großen Teil mit dieser neuen Version als Testapplikation statt. So konnten die Ladezeiten im Gegensatz zu FUJABA 4 mit CoObRA 1 um den Faktor 10 bis 30 (je nach Projektgröße) verringert werden. Auch der Speicherbedarf wurde drastisch reduziert, so dass statt etlichen hundert Megabyte nur noch einige 'zig Megabyte für dieselbe Projektgröße zum Laden benötigt wurden.

Neben der verbesserten Skalierbarkeit wurden in FUJABA 5 zwei wichtige neue Möglichkeiten für den Benutzer eingeführt. Einerseits können nun mehrere Projekte gleichzeitig geöffnet werden, die untereinander auch Querbeziehungen haben dürfen. Andererseits wurde ein sogenannter *Workspace* eingeführt, der auch nach dem Beenden der Anwendung die geöffneten Projekte enthält. Wird die Anwendung erneut gestartet, findet der Benutzer die Projekte so vor, wie er sie beim Schließen verlassen hat - auch wenn er sie nicht explizit gespeichert hat.

Mehrere Projekte werden durch CoObRA 2 mit mehreren Repositories unterstützt und auch Querbeziehungen werden in dem Konzept der CoObRA-Bibliothek berücksichtigt. In FUJABA war daher nur noch eine Benutzeroberfläche für die Verwaltung der Projektabhängigkeiten zu schaffen.

Für die Entwicklung des Workspace wurde zum ersten Mal die Flexibilität der CoObRA 2 Bibliothek auf die Probe gestellt. Es galt die aktuellen Änderungen an den geöffneten Projekten sofort im Workspace zu speichern, dies jedoch möglichst ohne dort eine Kopie der kompletten Projektdaten vorzuhalten. Die Projektdateien sollten also nicht in den Workspace kopiert

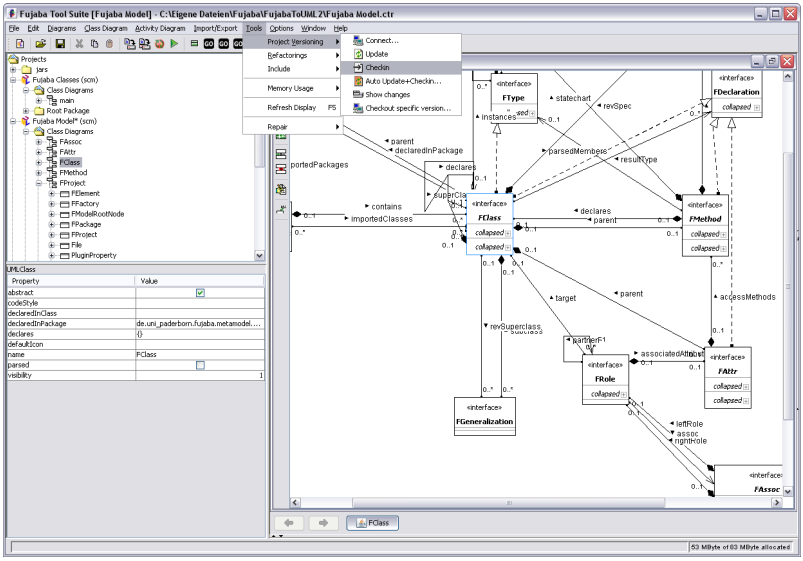


Abbildung 6.2: *Screenshot von FUJABA 5 mit CoObRA 2*

und nach jeder Änderung gespeichert werden. Weiterhin sollten die Projektdateien nicht direkt in den Benutzerverzeichnissen beim Schließen verändert werden, sondern die Projekte sollten weiterhin nach dem Neustart der Applikation als „noch nicht gespeichert“ markiert sein.

Für diesen Zweck wird ein konkatenierendes Persistenzmodul benutzt, das zwei dateibasierte Persistenzmodule vereint. Das Repository eines Projektes erhält zur Verwaltung der Änderungsdaten also ein Modul, das zunächst aus der Projektdatei liest und danach aus einer Datei im Workspace, um die „nicht gespeicherten“ Änderungen zu laden. Das Schreiben von neuen Änderungen wird ebenfalls in den Workspace umgeleitet und passiert sofort, wenn Änderungen auftreten. Löst der Benutzer den Speichervorgang aus, werden lediglich die Daten aus dem Workspace an die Projektdatei angehängt und die Workspacedatei neu angelegt. Diese Lösung sorgt auch dafür, dass sogar bei einem Absturz keine Daten verloren gehen.

### Neues Metamodell

Neben dem Einführen von mehreren Projekten wurde in FUJABA 5 das Metamodell gegenüber der Vorgängerversion intensiv überarbeitet. Viele Klassen wurden in andere Pakete verschoben und umbenannt. Attribute wurden teilweise gelöscht, teils vereinigt oder gar in ihrer Bedeutung leicht geändert. Objekte werden in FUJABA 5 über Fabriken und nicht länger direkt über Konstruktoren angelegt.

Da auch der alte FUJABA-interne Persistenzmechanismus namens FPR nun die Java Feature Abstraction (s. Abschnitt 5.2) benutzt, konnte der Import von FUJABA-4-Projekten schnell realisiert werden: Die Abstraktionsschicht wurde so konfiguriert, dass die alten Klassen- und Attributnamen auf die Handler für die neuen Klassen und Attribute aufgelöst wurden. Für vereinigte oder in der Bedeutung geänderte Attribute wurden neue Handler

erstellt, die die Attributwerte übersetzen. So musste im Inneren des Lademechanismus FPR nichts für den Import aus dem alten Metamodell geändert werden.

Auch die im Abschnitt 6.6 vorgeschlagenen Annotationen an den Rollen von Assoziationen wurden erstmals im FUJABA 5 Metamodell angefügt. So konnte ein zuverlässiger Mechanismus für das Kopieren und Einfügen von Elementen implementiert werden, der nicht mehr länger auf Heuristiken basiert.

### 6.1.2 Erfahrungen im Mehrbenutzerbetrieb

Im Rahmen von verschiedenen Forschungsprojekten wird FUJABA - am Lehrstuhl von Prof. Zündorf - von den Mitarbeitern nicht nur weiterentwickelt sondern auch als CASE-Tool zur Entwicklung eingesetzt. Durch die Unterstützung des Mehrbenutzerbetriebs mit CoObRA ist dies nun auch in Teams möglich. Da die Arbeitsabläufe zur Synchronisation der Arbeitskopien an das Vorgehen bei CVS und ähnlichen Versionsverwaltungen angelehnt sind, fanden sich alle Mitarbeiter sofort zurecht und nutzten den Machanismus intuitiv.

Bei CoObRA 1 und damit in FUJABA 4 war die einzig angebotene Konfliktlösungsstrategie die eigenen konfliktbehafteten Änderungen zu verwerfen und die Änderungen vom Server - also die von den Kollegen, die bereits mit dem Server synchronisiert hatten - zu akzeptieren. Verworfene Änderungen wurden schlicht in einem Synchronisationsprotokoll angezeigt. Diese Konflikte traten jedoch sehr selten überhaupt auf und falls sie auftraten waren die Mitarbeiter immer in der Lage, geeignete Maßnahmen - an Hand des Protokolls - manuell zu ergreifen. Dass syntaktische Modelle gravierende semantische Unstimmigkeiten zeigten, kam - nachdem die bereits genannten Anomalien behoben waren - nicht mehr vor.

Auf Grund dieser guten Erfahrungen mit der einfachen Konfliktlösungsstrategie ist dieses feingranulare Mischen in CoObRA 2 der Standardmechanismus. Er wird auch in FUJABA 5 wieder wie gehabt eingesetzt.

### 6.1.3 Generierte Anwendungen

Nicht nur in FUJABA selbst wird CoObRA 2 eingesetzt, sondern auch jede mit FUJABA entwickelte Anwendung kann auf einfache Weise die Funktionen der Bibliothek nutzen. Der generierte Quelltext, den das Softwareentwicklungswerkzeug erstellt, enthält bei Bedarf bereits die notwendigen Programmteile in den Zugriffmethoden, um PropertyChangeEvents - Ereignisse, die die Änderung einer Objekteigenschaft mitteilen - automatisch zu versenden. Diese Ereignisse werden von einer CoObRA 2 Komponente dann in die Änderungsdaten für die Bibliothek übersetzt. Auch Probleme mit Nebeneffekten in Zugriffsmethoden gibt es gängigerweise nicht, da standardmäßig von FUJABA's Codegenerierung keine Nebeneffekte in Zugriffsmethoden von Attributen oder Assoziationen vorgesehen sind.

## 6.2 MOF

MOFLON [1] ist ein an der TU Darmstadt entwickeltes Meta-CASE-Tool, das zwar auf FUJABA aufsetzt, aber ein eigenes Metamodell mitbringt. Die Implementierung dieses Metamodells ist an den JMI Standard angelehnt und ist MOF-2.0-konform. Um die CoObRA 2 Bibliothek auch für die MOFLON Daten einsetzen zu können, wurde eine entsprechende Schicht für die „Java Feature Abstraction“ erstellt. Diese übersetzt die Anfragen von CoObRA in die reflektiven Zugriffe von JMI, um Metamodell-Informationen auszulesen und das Modell zu verändern. Auf diesem Weg konnten die Persistenzfunktionen sowie Rückgängig- und Wiederholen-Funktionen von einem

Mitarbeiter an der TU Darmstadt in MOFLON integriert werden.

Das MOFLON-Modell wird zum Teil auf die FUJABA-internen Modellschnittstellen übersetzt, so dass vorhandene Editoren und Code-Generatoren weitergenutzt werden können. Dies brachte für die Darmstädter einige Schwierigkeiten mit sich, da die zunächst implementierten Übersetzungsmechanismen sich auf die Reihenfolge bestimmter Modelländerungen verließen. So erzeugten die Übersetzungen teilweise unerwartete Nebeneffekte und Fehlermeldungen während des Ladens. Diese Probleme konnten zum Teil individuell gelöst werden. Der verbliebene Teil wurde dadurch behoben, dass die Übersetzungsvorgänge während des Eingriffs von CoObRA (Laden, Undo, Redo) abgeschaltet wurden.

### 6.2.1 EMF

Die Implementierung der Abstraktionsschicht für EMF[3] Modelle ist in Zusammenarbeit mit der Universität Bayreuth entstanden. Es gestattet jeder Applikation, deren Modell mittels EMF implementiert worden ist, die einfache Nutzung von CoObRA 2. Die Universität Bayreuth konnte CoObRA 2 in ersten Testläufen bereits einsetzen.

## 6.3 OBA

In dem Forschungsprojekt OBA [8], das an der Universität Kassel zusammen mit der Volkswagen AG, Siemens VDO und Sumitomo (ehemals VW Bordnetze) durchgeführt wird, kommt CoObRA 2 ebenfalls zum Einsatz. Da OBA zu großen Teilen eine mit FUJABA generierte Software ist, war

[3] Das Eclipse Modeling Framework (EMF) ist ein Rahmenwerk der Eclipse Open Source Community, das eine standardisierte Abbildung von MOF-ähnlichen Metamodellen auf Java Quelltext bereitstellt.

CoObRA 1 nach wenigen Arbeitsschritten für die Persistenz einsatzbereit und wurde so von Anfang an verwendet. Sobald CoObRA 2 einsatzbereit und getestet war, wurde die OBA Anwendung auf die neue Bibliothek umgestellt, um die Performanzsteigerungen auszunutzen.

Neben dem Einsatz der Abstraktionsschicht über CoObRA 2 wurde die Java Feature Abstraction auch noch anderweitig in OBA eingesetzt: In der Anwendung waren etliche Tabellenansichten für die Daten notwendig. Diese wurden *generisch*[4] programmiert und benutzen für den Zugriff auf die Daten die für CoObRA eingeführte Abstraktionsschicht. So war es möglich Tabellen für alle Arten von Objekten allgemein zu programmieren und der Arbeitsaufwand für das Erstellen der Applikation nahm erneut deutlich ab.

Aus der Nutzung der Abstraktionsschicht in dieser Weise erwuchsen allerdings auch die bereits genannten Zusatzanforderungen (Abschnitt 5.2) an diese Schicht, deren Implementierung auch in der nächsten vorgestellten Anwendung von Nutzen war.

## 6.4 eDOBS

Das „eclipse Dynamic Object Browsing System“ [7] (eDOBS) stellt den Hauptspeicher eines laufenden Java-Programms als UML-Objektdiagramme dar. Es werden also die Daten einer laufenden Applikation ausgelesen und für jedes gefundene Objekt ein Diagrammelement angelegt, welches mit Informationen über Attributbelegungen angereichert wird. Für jeden Link wird auch eine grafische Verbindung zwischen den entsprechenden Diagrammelementen angelegt.

[4] Mit generisch ist hier gemeint, dass die Programmteile sich nicht auf konkrete Klassen des applikationsspezifischen Metamodells beziehen, sondern allgemein mit Objekten arbeiten, deren Klasse sie nur zur Laufzeit kennen. Es gibt also zur Übersetzungszeit keine Abhängigkeiten zum konkreten Metamodell.

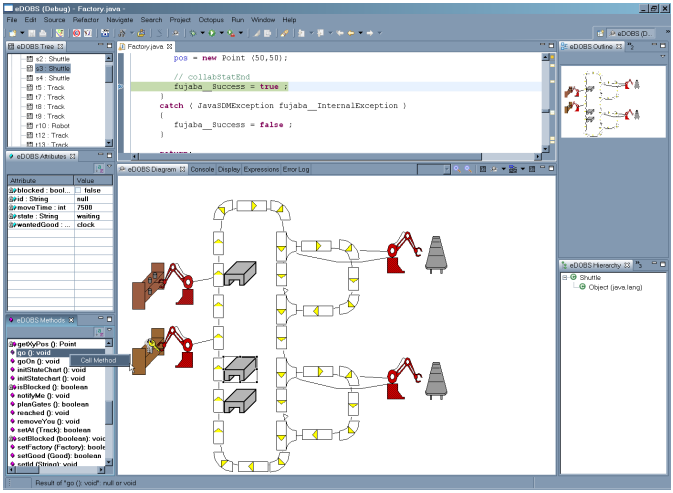

Abbildung 6.3: *Screenshot von eDOBS in Eclipse*

Im Rahmen dieser Arbeit ist die Realisierung von eDOBS insofern interessant, als dass wiederum die Java Feature Abstraction eingesetzt wird, um das Modell einer Applikation zu analysieren[5]. So ist es eDOBS möglich, nahezu jedes Modell zu erfassen und geeignet zu visualisieren. Eine spezielle Erweiterung der Abstraktionsschicht wurde vom Autor des Werkzeugs eigens für eDOBS angefertigt: Eine Implementierung der Abstraktionsschicht, die über die Debugger-Schnittstelle von Eclipse auf ein Modell zugreift. Mit dieser Komponente ist es eDOBS möglich auch über die Grenzen seiner Virtual Machine hinweg Objektstrukturen zu analysieren und den Anwender bei der Fehlersuche in dem analysierten Programm zu unterstützen.

[5] CoObRA kommt bisher in eDOBS nicht zum Einsatz. In einer Diplomarbeit am Lehrstuhl wird im Moment jedoch eine Lösung entwickelt CoObRA-Protokolle als animiertes eDOBS-Diagramm zu visualisieren. Weiterhin ist geplant, für die analysierten Modelle Persistenzfunktionen anzubieten, selbst wenn die dargestellte Applikation dies nicht unterstützt.

## 6.5 Echtzeitnaher Einsatz

Im Gegensatz zu Anwendungen wie CASE-Tools oder anderen Programmen, bei denen in Abständen von vielen Minuten bis hin zu Tagen kommuniziert wird, um Arbeitsergebnisse auszutauschen, ist es bei verteilten Simulationsumgebungen und Spielen wichtig, ein Modell innerhalb kürzester Zeit und mit Abständen von einigen Millisekunden immer wieder zu synchronisieren. Bei dieser Art der Synchronisation stehen Konflikterkennung und -behebung im Hintergrund, da durch die kurzen Synchronisierungszyklen und die oft disjunkten Arbeitsbereiche der Programminstanzen kaum Konflikte entstehen. Wenn Konflikte erkannt und behoben werden sollen verbietet sich weiterhin eine Benutzerinteraktion zur Konfliktlösung, da dies zu viel Zeit kosten würde, stattdessen werden einfache Regeln[6] benutzt, um Konflikte zu beheben.

### 6.5.1 Digitale Fabrik

Primäre Zielapplikation für die echtzeitnahe Kategorie ist das Projekt „Physics Factory“ am Lehrstuhl von Prof. Zündorf. In dieser Anwendung wird mittels OpenGL[7] und ODE[8], einer Physiksimulationsbibliothek, eine Fabrikhalle simuliert. In dieser Umgebung kann Steuerungssoftware für Anlagen und Fahrzeuge getestet werden. CoObRA 2 wurde bereits in Tests eingesetzt, um die Instanzen der Simulation auf mehreren Rechnern miteinander zu synchronisieren und so verteilte Ansicht und Simulation der Fabrik zu erlauben.

[6] wie das Verwerfen der konfliktbehafteten lokalen Änderungen

[7] Die Open Graphics Library ist eine Bibliothek zur Ansteuerung moderner 3D Grafikkarten [16]

[8] Open Dynamics Engine, http://www.ode.org/

### 6.5.2 Spiele mit hohen Anforderungen

Um die Anforderungen von echtzeitnahen Anwendungen an die Bibliothek zu verdeutlichen wird hier jedoch ein einfacheres Beispiel näher betrachtet, das während der Forschungsarbeiten als Konzeptstudie entstand: Ein schnelles Air-Hockey Spiel namens „MultiShufflePuck“. Ein Puck wird von mehreren Spielern mit je einem Quader vom eigenen Tor ferngehalten, um gleichzeitig zu versuchen ihn im Gegenzug einem Gegner in das Tor zu schießen (siehe Abbildung 6.4).

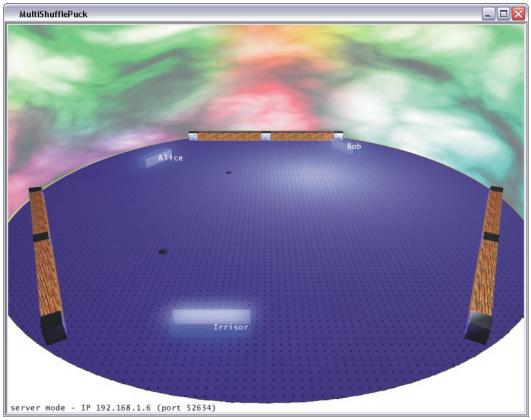


Abbildung 6.4: *Screenshot des verteilten Air-Hockey Spiels MultiShufflePuck.*

In einem derartigen Spiel kommt es für die Spieler darauf an, den aktuellen Spielzustand, hier also die Positionen von Pucks und Gegnern, möglichst aktuell dargestellt zu bekommen. Bereits Verzögerungen von wenigen Zehntelsekunden werden als störend empfunden.

Um diese geringen Latenzen zu erreichen, durfte - wie bereits in Abschnitt 5.4.2 erläutert - nicht auf die bisher verwendete TCP/IP-basierte Kommunikation zurückgegriffen werden. Mit der vorgestellten, auf das UDP Protokoll

zugeschnittenen, Synchronisierungsstrategie lassen sich Modelle jedoch effektiv zeitnah abgleichen. Selbst bei hohen Netzwerklatenzen (150ms pro Weg) und unnatürlich hohen Paketübertragungsfehlerraten (20%) war das Air-Hockey-Spiel noch ohne wahrnehmbare Probleme spielbar. Mit dem TCP-Modul hingegen waren die Ergebnisse bereits bei deutlich kleineren Fehlerraten (5%) inakzeptabel.

## 6.6 Kopieren und Einfügen

Obwohl das Kopieren und Einfügen ursprünglich nicht Anforderung an CoObRA war und da es aber zum Teil inhaltlich mit dem Rest der Arbeit verknüpft ist, folgen hier nun einige Überlegungen zu dieser Problematik. In Abschnitt 6.6.8 wird außerdem die Realisierung der Funktionalität beschrieben.

Auf den ersten Blick erscheint das Kopieren und Einfügen in objektorientierten Anwendungen als triviales Problem. So handelt es sich ja lediglich um das Kopieren von Teilgraphen einer Objektstruktur und dem Einfügen an anderer Stelle desselben Graphen oder eines anderen Graphen. Widmet man sich der Realisierung dieser Funktionalität in einer größeren Anwendung, treten jedoch einige verkomplizierende Feinheiten zu Tage.

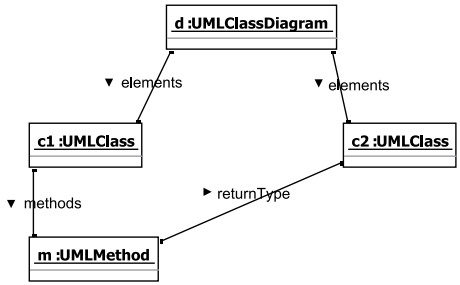


Abbildung 6.5: *Beispielszenario zum Kopieren und Einfügen; Objektdiagramm*

Um die Erläuterungen konkreter formulieren zu können wird in diesem Abschnitt als Beispiel der in Abbildung 6.5 zu sehende Ausschnitt eines FUJABA UML Modells verwendet: ein Klassendiagramm mit zwei Klassen; eine Klasse enthält eine Methode. Dabei ist es wichtig zu bemerken, dass der

Benutzer die Klasse c1 und die Methode m1 als Einheit oder Komposition wahrnimmt; Im Modell jedoch sind es zwei getrennte Objekte, die über Links verbunden sind.

Das erste auftretende Problem ist die Identifizierung des gewünschten Teilgraphen auf Basis einer Benutzeraktion. So erwartet ein Benutzer natürlich nicht nur ein einzelnes kopiertes Objekt, wenn er in seinem CASE-Tool eine Klasse kopiert, sondern auch Kopien der Methoden und Attribute der Klasse und etwaiger für ihn nicht sichtbarer Hilfsobjekte. Diese weiteren Objekte müssen kopiert werden, auch wenn sie nicht explizit durch den Benutzer ausgewählt wurden. Die erwartete Menge an zu kopierenden Objekten zu bestimmen ist also domänen- beziehungsweise modellspezifisch. In FUJABA wurden daher verschiedene Möglichkeiten untersucht diese Menge duch Heuristiken und Annotationen aus der Benutzerselektion abzuleiten. Die vorgestellten Heuristiken haben sich dabei allerdings nicht bewährt.

Das zweite Problem ist die *Kontexteinbettung*: Es müssen Links ausgewählt werden, die die Verbindung der Kopie zu den umgebenden Objekten herstellen. Teilweise müssen Links, die von den zu kopierenden Objekten ausgingen auch von den neu eingefügten Objekten aus gezogen werden; ein anderer Teil darf jedoch nicht neu erstellt werden, da sonst das Modell im schlimmsten Fall ungültig ist[9]. Auch für diese Auswahl waren allein auf Kardinalitäten basierende Heuristiken nicht geeignet.

Um dem Applikationsprogrammierer dennoch weitreichende Hilfestellungen bei der Bestimmung der zu kopierenden Objekte zu geben, kann man jedoch einige Überlegungen anstellen, die im Folgenden dargestellt werden.

[9] Da zum Beispiel Kardinalitäten verletzt werden

### 6.6.1 Auswahl der Menge zu kopierender Objekte

Der Benutzer (beziehungsweise ein Programmteil) selektiert im Modell der Applikation meist nur einige Wurzelobjekte, die er kopieren möchte. Dies könnte zum Beispiel in einem CASE-Tool eine UML-Klasse sein. Erwartet wird jedoch - wie eingangs beschrieben - beim Einfügen nicht nur eine Kopie des einzelnen Objekts zu erhalten, sondern auch Kopien der zugehörigen Objekte wie Methoden, Attribute und eventueller Hilfsobjekte, die der Benutzer garnicht direkt wahrnimmt. Es gilt also an Hand der spezifizierten Wurzelobjekte die Menge an Objekten zu bestimmen, von denen tatsächlich Kopien anzufertigen sind.

Die Menge der zu kopierenden Objekte bildet normalerweise zusammenhängende Graphen, in denen sich mindestens ein Wurzelobjekt befindet. Das bedeutet, jedes dieser Objekte ist durch einen Pfad von Links mit einem Wurzelobjekt verbunden[10]. Man könnte also die Menge der zu kopierenden Objekte dadurch bestimmen, dass man für jeden Link(typ) entscheidet, ob über ihn erreichbare Objekt mitkopiert werden sollen.

Die Betrachtung einiger Beispiele zeigte, dass man an Hand der Assoziation und Richtung eines Links schon entscheiden kann, ob er mitzukopierende Objekte referenziert. Es ist also die *Property*[11] entscheidend die den Typ der Referenz angibt. Kann man also die Menge der Properties $P$ bestimmen, deren Links auf zu kopierende Objekte zeigen, lässt sich ein Algorithmus formulieren, der die Menge $K$ der zu kopierenden Objekte aus den Wurzelobjekten bestimmt:

*Man bilde eine Menge $K_0$, die alle Wurzelobjekte enthält. Nun bildet man*

[10] Für Modelle, bei denen dies nicht zutrifft wird ohne Beschränkung der Allgemeinheit angenommen, dass es dann einen anderen Weg gibt, solche alleinstehenden Objekte (oder Teilgraphen) mit anderen kopierten Objekten zu assoziieren. Dieser Weg wird dann als Link betrachtet.

[11] Die Assoziationsenden - auch Rollen genannt - heißen in UML 2.0 und MOF „Property“.

*rekursiv die Menge $K_{i+1}$ mit Inhalt der Menge $K_i$ sowie aller Objekte, die über eine Referenz (Link) von einem Objekt aus $K_i$ erreichbar sind und deren Property in der Menge $P$ ist. Ist die nächste Menge $K_{n+1}$ gleich der vorherigen $K_n$ ist der Algorithmus abgeschlossen und man setzt $K = K_n$.*

### 6.6.2 Auswahl der Menge zu kopierender externer Links

Ist die Menge der zu kopierenden Objekte bestimmt, stellt sich die nächste Frage: Welche Referenzen auf Objekte außerhalb der Menge - *externe Referenzen* - sollten von den Kopien der Objekte aus ebenfalls gezogen werden? Hierzu wird zunächst das Beispiel des UML Werkzeugs bemüht:

Abbildung 6.5 zeigte ein Klassendiagramm mit zwei Klassen; eine Klasse enthielt eine Methode. Kopiert man nun die Klasse c1, wird auch eine Kopie von der enthaltenen Methode m angefertigt. Die Kopie der Methode sollte dann genau wie sein Original die Klasse c2 als Rückgabetyp referenzieren. Kopiert man hingegen die Klasse c2 (m wird nicht kopiert), sollte die Kopie von c2 nicht m referenzieren, da die Methode natürlich ihren Rückgabetyp c2 behält (und nicht nun die Kopie von c2 zurückgibt).

Wie auch die Entscheidung für die Objekte, lässt sich an der Art der Referenz - der Property - festmachen, ob eine externe Referenz kopiert werden muss. Es gibt also auch eine Menge von Properties, deren Referenzen auch mitkopiert werden, wenn sie auf Objekte außerhalb der Menge zu kopierender Objekte zeigen.

### 6.6.3 Kontextwechsel

Man kann eine weitere Beobachtung bezüglich der Kanten machen, wenn man die Veränderungen betrachtet, die ein Benutzer erwartet, wenn er an

einer anderen Stelle als der ursprünglichen eine Kopie einzufügen versucht. In dieser Arbeit wird von einem *Kontextwechsel* gesprochen. Ein Beispiel für einen solchen Wechsel ist das Kopieren einer UML Klasse von einem Paket in ein anderes. Man erwartet hier als Benutzer, dass die Kopie der Klasse nicht in dem Originalpaket ist, sondern in dem Zielpaket. Auf Modellebene bedeutet dies, dass eine Kante nicht wie gehabt beim Einfügen wiederhergestellt wird, sondern ein anderes Zielobjekt ausgewählt wird. Ein Mechanismus für diese Auswahl wird beim Auflösen der Identifizierer in Abschnitt 6.6.5 vorgeschlagen.

Die wichtige Beobachtung hierbei ist, dass die Auswahl solcher geänderter Kanten ebenfalls von ihrem Typ, also der Property, abhängt. Einige Properties deuten auf eine Beziehung hin, die der Benutzer als Enthaltensein wahrnimmt. Diese können dann erwartungsgemäß geändert werden, wenn alter und neuer Kontext bekannt sind.

### 6.6.4 Abbildung auf die UML

Die drei Eigenschaften von Properties (Rollen) - „Referenzen zeigen auf zu kopierende Objekte“, 'Referenzen werden auch extern kopiert' und „Referenzen zeigen auf Kontextelemente“ - sind nicht unabhängig voneinander. Sorgt eine Property dafür, dass mit ihr referenzierte Objekte kopiert werden, ist die Eigenschaft, ob externe Referenzen kopiert werden, irrelevant, da sie nie externe Objekte referenzieren kann. Weiterhin zeigt das Gegenstück einer Property, deren Instanzen auf zu kopierende Objekte zeigt, üblicherweise auf Kontextelemente.

Bei der näheren Betrachtung der Eigenschaften von Properties und dem Vergleich mit den Attributen, die die UML für Properties definiert, stellte sich heraus, dass diese sich zum Teil aufeinander abbilden lassen: Properties zeigen genau dann auf zu kopierende Objekte, wenn ihr Kompositions-

Attribut gesetzt ist (Composition). Weiterhin zeigen sie auf ein Kontextelement genau dann, wenn die Gegenrichtung eine Komposition oder Aggregation ist; auch werden in diesem Fall externe Referenzen kopiert. Allerdings gibt es weitere externe Referenzen, die kopiert werden müssen - diese ließen sich nicht zuverlässig auf ein gängiges UML-Attribut von Properties oder Assoziationen abbilden. Dem Designer des Metamodells müssen demnach nur noch Methoden an die Hand gegeben werden, solche Properties zusätzlich zu markieren, deren Instanzen weitere zu kopierende externe Referenzen sind.

### 6.6.5 Serialisierung und Identifizierer

Um Kopieren und Einfügen zwischen Applikationen zu ermöglichen müssen die Daten beim Kopieren in die Zwischenablage übertragen werden. Hierzu ist eine Serialisierung nötig. Im Gegensatz zur hier vorgestellten Serialisierung für Persistenzzwecke ist es jedoch nicht wünschenswert, hierfür Objektidentifizierer zu verwenden, die global eindeutig sind. Die beim Kopieren verwendeten Identifizierer (IDs) für externe Referenzen müssen sich im Zielkontext auflösen lassen, auch wenn es sich dort um andere Objekte handelt. Beispielsweise erwartet ein Benutzer, der eine UML-Methode kopiert, dass diese beim Einfügen in eine UML-Klasse eines anderen Projekts noch immer den Typ „String“ hat, auch wenn dies im Zielprojekt ein anderes Modellelement als im Quellprojekt ist.

Um dies zu gewährleisten reicht die automatische Vergabe von IDs nicht aus. Stattdessen muss die Applikation modellspezifische Identifizierer zur Verfügung stellen. Diese sollten es der Applikation später ermöglichen geeignete Objekte im Zielmodell wiederzufinden. Für UML-Klassen eignet sich hierzu zum Beispiel der voll qualifizierte Name der Klasse.

### 6.6.6 Ausschneiden

Beim Ausschneiden sind im Gegensatz zum Kopieren einige Besonderheiten zu beachten. Insbesondere sind beim ersten Einfügen nach dem Ausschneiden *alle* externen Referenzen wiederherzustellen, nicht nur die besonders zum Kopieren ausgezeichneten. Der Benutzer erwartet bei Ausschneiden-Kopieren vom Effekt her ein Verschieben, nicht ein Kopieren, Löschen und Einfügen. Daher sollte auch die Auswirkung auf das Änderungsprotokoll nicht sein, dass Objekte neu erzeugt werden, sondern nur Kanten umgesetzt werden. Dies lässt sich erreichen, indem beim Ausschneiden nur alle externen Referenzen gelöscht werden. Ein isolierter Teil eines Modells wird in den meisten Applikationen dem Benutzer nicht dargestellt und scheint somit für ihn gelöscht zu sein. Beim ersten Einfügen werden dann die Kanten bezüglich des Kontextwechels komplett neu gezogen. Verwirft der Benutzer den Inhalt der Zwischenablage oder fügt er ihn in ein anderes Modell ein, müssen die isolierten Objekte allerdings entgültig gelöscht werden. Beim Einfügen in ein anderes Modell wird zunächst ein normales Kopieren durchgeführt, danach können die Beziehungen zu dem alten Modell als Querbeziehungen - sofern möglich - wiederhergestellt werden. Ist dies nicht möglich, weil zum Beispiel unerlaubte Querbeziehungen entstehen würden, bleibt es beim normalen Kopieren.

### 6.6.7 Zwischenfazit zum Kopieren

Auf Seiten des Applikationsprogrammierers bleiben also zwei Aufgaben zu erledigen: Das Metamodell muss mit Informationen über Kompositionen, Aggregationen und Properties für weitere zu kopierende, externe Referenzen angereichert werden (wobei die zuerst genannten bereits in den meisten Metamodellen enthalten sind). Weiterhin sind Methoden zur Vergabe und Auflösung von modellspezifischen IDs zu implementieren.

Die Realisierung des vorgestellten Vorgehens auf Seiten der Bibliothek ist mit den erarbeiteten Konzepten in dieser Weise bereits möglich und wird im folgenden Abschnitt vorgestellt.

### 6.6.8 Realisierung

Die bisher vorgestellten Konzepte bieten Methoden, um das Ausschneiden, Kopieren und Einfügen allgemein mit CoObRA zu realisieren, wenn modellspezifische Informationen zur Verfügung stehen. Die Informationen über Kompositionen, Aggregationen sowie Annotationen zum Kopieren weiterer externer Referenzen werden von der Abstraktionsschicht (s. Abschnitt 5.2) geliefert. Für die verlangten modellspezifischen Identifizierer (Kontext-IDs) wird für die Applikation eine Schnittstelle vorgegeben, um IDs applikationsspezifisch vergeben und auflösen zu können.

Sind die Informationen über das Metamodell verfügbar, gilt es nun an Daten über die aktuellen Modellelemente zu kommen. Für die Bibliothek gibt es dafür zwei Möglichkeiten. Einerseits kann das Modell ebenfalls über die Abstraktionsschicht direkt ausgelesen werden, was weitere Anforderungen an diese Schicht stellt. Andererseits können die Daten aus dem Änderungsprotokoll gelesen werden. In CoObRA 1 wurde diese zweite Möglichkeit gewählt, da es dort keine Abstraktionsschicht gab, die Zugriff auf das Modell hätte bieten können. Da jedoch eine Navigation in der Objektstruktur zum Auffinden der Nachbarn von kopierten Objekten wiederholt nötig ist, stellt das Verfahren über die Änderungsprotokolle sehr hohe Performanzanforderungen. In CoObRA 2 wurde der Weg über die Abstraktionsschicht gewählt, da die Navigation direkt über die Objektstrukturen schneller ist und die zusätzlichen Anforderungen, mit der Abstraktionsschicht lesen zu können, bereits erfüllt wurden (s. Abschnitt 5.2).

Um möglichst großen Nutzen aus den bereits implementierten Mechanis-

men der Bibliothek ziehen zu können, werden die Daten über die zu kopierenden Objekte - als Änderungsliste - serialisiert. Dabei wird der Mechanismus zum Erzeugen von Identifizierern ausgetauscht, um die Kontext-IDs zu verwenden. Die Liste von Änderungen ergibt sich in diesem Fall allerdings nicht aus dem Protokollieren, sondern wird künstlich aus den Daten über die zu kopierenden Objekte und Links erstellt, die aus der Modellabstraktionsschicht ausgelesen werden.

Das in 6.6.1 vorgestellte Verfahren zur Auswahl der Menge zu kopierender Objekte kann durch den Zugriff auf die Objektstruktur entsprechend leicht implementiert werden, da alle nötigen Operationen durch die Abstraktionsschicht abgedeckt werden. Für jedes Objekt in dieser Menge wird eine entsprechende Änderung zum Erzeugen des Objekts generiert. Danach können alle von dem Objekt ausgehenden Links für das Generieren einer Änderung in Betracht gezogen werden: Ist entweder das Ziel ebenfalls in der Menge der zu kopierenden Objekt oder wird der Link auch nach extern kopiert, wird eine entsprechende Änderung generiert. Beim Ausschneiden werden alle externen Links in Änderungen übersetzt, da diese auch komplett gelöscht und wiederhergestellt werden müssen.

Ist das Protokoll erstellt werden noch die externen Referenzen, die auf Kontextelemente zeigen und deren Properties für die Kontextersetzung ausgezeichnet sind, markiert, um sie beim Einfügen geeignet ersetzen zu können. Danach kann das Protokoll wie gewohnt serialisiert werden, um anschließend in die Zwischenablage gestellt zu werden.

Beim Einfügen wird die Änderungsliste deserialisiert und angewandt. Beim Auflösen der IDs von externen Referenzen wird - wie bei der Vergabe - die Applikation bemüht. Kopierte Objekte verwaltet der normale ID-Mechanismus, da die Objekte neu erstellt werden. Die Ersetzung der Kontextobjekte passiert transparent beim Auflösen der Identifizierer, da sie

bereits zuvor als Kontext markiert wurden. Nach dem Anwenden der Liste ist das Einfügen bereits abgeschlossen.

Beim Sonderfall des ersten Einfügens nach dem Ausschneiden, werden nur die externen Referenzen angewandt, falls im selben Modell eingefügt wurde. Wird in ein anderes Modell ohne Querbeziehungen eingefügt, entspricht das Verfahren einem Kopieren, Löschen und Einfügen und wird normal verarbeitet. Sind Querbeziehungen zwischen Modellen zugelassen, werden nach dem Einfügen die zusätzlichen externen Referenzen auf die eingefügten Objekte wiederhergestellt.

# 7 Ausblick

Als Zusammenfassung der Forschungsergebnisse lässt sich festhalten, dass der Ansatz, Änderungsprotokolle für Persistenz, Verteilung und Versionierung einzusetzen, seine Tragfähigkeit in den vorgesehenen Einsatzgebieten zeigen konnte. Es sind weiterhin problematische Eigenschaften von Metamodellimplementierungen aufgezeigt worden, die den Einsatz der entwickelten Bibliothek erschweren; für diese Fälle, wie zum Beispiel Nebeneffekte in Zugriffsmethoden, unerwartetes Verhalten der Modelle und Anomalien während des Ladens, wurden Abhilfsmöglichkeiten vorgestellt. Auch die Universalität und Flexibilität des Ansatzes sowie der in Java implementierten Bibliothek CoObRA 2 konnten an Hand von diversen Anwendungsszenarien gezeigt werden.

Obwohl die grafische Benutzeroberfläche für FUJABA auf ein Minimum beschränkt wurde und bisher keine Erkennung für semantische (kontextsensitive) Inkonsistenzen in dem CASE-Tool zur Verfügung steht, ist der Einsatz der Bibliothek in der Entwicklungsumgebung sehr erfolgreich. Sie wird bereits für die gemeinsame Entwicklung von Software eingesetzt - sowohl mit CVS als auch mit CoObRA-Servern. Ein Mechanismus für die Untersuchung eines FUJABA-Projektes auf semantische Unstimmigkeiten ist bereits in der Planung und wird auch nach der Konfliktdetektion in Zukunft wertvolle Informationen liefern. Eine Benutzeroberfläche zur Abfrage von Versionen, Erstellung von Branches und dem Mischen derselben wäre für ein CASE-Tool ebenfalls eine Bereicherung. Da die Bibliothek bereits

die nötigen Mechanismen zur Verfügung stellt wäre diese Oberfläche geeignetes Thema für neue Diplom- oder Masterarbeiten. Auf diese wiederum könnten Untersuchungen zum Mischen von Branches aufsetzen: Es gilt zu erforschen welche Probleme das Mischen zweier Modellversionen, die sich potentiell weit voneinander entfernt haben, aufwerfen kann.

Mit der vorgestellten Filterschnittstelle ist es möglich flexible Anfagen an ein Persistenzmodul zu stellen, um gezielt Daten (zum Beispiel über bestimmte Objekte) aus den Protokollen zu lesen. In den bisher implementierten Modulen wird dies durch eine sequenzielle Suche durch die Dateien realisiert. Denkbar wäre jedoch hier eine Brücke zur Domäne der Datenbankmanagementsysteme zu schlagen: Ein Persistenzmodul für Datenbanken könnte Anfragen in SQL oder ähnliche Sprachen übersetzen, um Änderungsprotokolldaten direkt in der Datenbank zu filtern. Von einem solchen Verfahren würde insbesondere die Teilgraphen-Persistenz profitieren, bei der das Filtern intensiv eingesetzt wird.

Der Einsatz in der Echtzeitsimulation ist bisher ebenfalls sehr vielversprechend. Die sonst sehr aufwändige effiziente Synchronisation über ein Netzwerk ist mit einigen Handgriffen in eine modellzentrierte Javaanwendung integriert. Auch hier gibt es jedoch weiteren Forschungsbedarf: So ist die Virtuelle Fabriksoftware, die im Fachgebiet entstanden ist, bisher zwar verteilt auf mehreren Arbeitsrechnern gelaufen und wurde mit CoObRA synchronisiert, der Testlauf auf der CAVE des Fachgebiets Technische Informatik steht jedoch noch aus. Hier ist es nötig die Inhalte framegenau darzustellen, wofür eventuell weitere Mechanismen als die bisher vorgestellten nötig sind, um die Darstellungszeit zu synchronisieren.

Zur Zeit gibt es für das vorgestellte Konzept nur eine Implementierung in Java. Die theoretischen Überlegungen beschränken sich jedoch nicht auf diesen Einsatz. Es ist daher durchaus möglich die CoObRA-Bibliothek für

andere Zielsprachen und Plattformen zu portieren. Dies bedeutet jedoch - ob des Umfangs des aktuellen Quellcodes - einen nicht unerheblichen Arbeitsaufwand: Der Kern von CoObRA 2 umfasst 30.000 Zeilen Quelltext, die Abstraktionsschicht samt einer Standardimplementierung (JMI/MOF, EMF und JDI Implementierungen nicht mitgerechnet) sogar weitere 60.000 Zeilen. Weiterhin gilt es äquivalente Konstrukte für Hilfsklassen wie Listen- und Mengenimplementierungen sowie Assoziationsimplementierungen und Zugriffskonzepte auf die Zielsprache(n) abzubilden. Allein zur Ergänzung der in der Java-Laufzeitumgebung vorhandenen Mengen, Listen und Iteratoren um weitere Hilfsklassen wurden für CoObRA 15.000 Zeilen in einer externen Bibliothek geschrieben.

Die hier vorgestellten Beispielanwendungen belegen die breite Anwendbarkeit der, während dieser Arbeit enstandenen, Konzepte und der CoObRA 2 Bibliothek für eine Vielzahl unterschiedlicher Applikationen. Jede Art von Javaprogramm, in dem Mehrbenutzerbetrieb, Undo/Redo, Verteilung oder Synchronisation benötigt werden, kann durch den Einsatz von CoObRA mit minimalem Arbeitsaufwand von den präsentierten Ergebnissen direkt profitieren. So leistet diese Arbeit einen wertvollen Beitrag dazu, das Ziel der Softwaretechnik, Programme einfacher und effizienter zu machen, zu erreichen.

# A Anhang

## A.1 Literaturverzeichnis

[1] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *A. Rensink, J. Warmer (eds.), Model Driven Architecture - Foundations and Applications: Second European Conference, Heidelberg, Lecture Notes in Computer Science (LNCS), Vol. 4066, 361–375*. Springer Verlag, 2006.

[2] L. DeMichiel, M. Keith, and others. Java Persistence API in JSR 220: Enterprise JavaBeans TM 3.0. http://jcp.org/en/jsr/detail?id=220.

[3] eclipse community. Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/.

[4] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In *Proc. of the* $6^{th}$ *International Workshop on Theory and Application of Graph Transformation*. Paderborn, Germany, 1998.

[5] S. Fordin. Overview of JAXB. https://jaxb.dev.java.net/.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley, 1995.

[7] L. Geiger and A. Zündorf. eDOBS - Graphical Debugging for eclipse. In *3rd International Workshop on Graph-Based Tools (GraBaTs); ICGT 2006, Natal, Brasil*, September 2006.

[8] R. Gemmerich, S. Semmelrodt, A. Zündorf, C. Reckord, J. Leohold, J. Trippler, L. Brabetz, D. Müller, U. Schrey, and H.-G. Weil. Ein ganzheitlicher Ansatz zur Generierung und Optimierung von Fahrzeugbordnetzen. In *12th International Conference and Exhibition Electronic Systems for Vehicles Baden-Baden (VDI Berichte Nr. 1907), pp. 597-608*. VDI Verein Deutscher Ingenieure (Hrsg.), Germany), October 2005.

[9] IBM Corporation. Model Integrator Guide.

[10] U. Kelter, J. Wehren, and J. Niere. A generic difference algorithm for uml models. In P. Liggesmeyer, K. Pohl, and M. Goedicke, editors, *Software Engineering*, volume 64 of *LNI*, pages 105–116. GI, 2005.

[11] N. Kiesel, A. Schürr, and B. Westfechtel. *GRAS: A Graph-Oriented Software Engineering Database System*. Springer Verlag, Berlin, 1996.

[12] J. Klingemann, T. Tesch, and J. Wäsch. Enabling Cooperation among Disconnected Mobile Users. In *Conference on Cooperative Information Systems*, pages 36–46, 1997.

[13] A. Lal and J. Formichelli. The java virtual machine, section 4.6, memory manager. citeseer.ist.psu.edu/lal98java.html.

[14] Microsoft Corporation. Managing projects with Visual SourceSafe. Redmond, Washington, 1997.

[15] M. Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *Lecture Notes in Computer Science*. Springer, 1996.

[16] J. Neider, T. Davis, and M. Woo. Opengl programming guide: The official guide to learning opengl. Addison-Wesley, Reading, MA, 1993.

[17] Object Management Group. MOF 2.0 / XMI Mapping Specification, v2.1. http://www.omg.org/cgi-bin/doc?formal/2005-09-01.

[18] Objectweb. ASM - a Java bytecode manipulation framework. http://asm.objectweb.org/.

[19] D. Ohst. Versionierungskonzepte mit Unterstützung für Differenz- und Mischwerkzeuge. Siegen, Univ., Diss., 2004.

[20] I. Rockel. Versionierungs- und Mischkonzepte für UML Diagramme. Diplomarbeit Universität-GH Paderborn, September 2000.

[21] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wäsch, and P. Muth. Towards a Cooperative Transaction Model - The Cooperative Activity Model. In *The VLDB Journal*, pages 194–205, 1995.

[22] C. Schneider. CASE Tool Unterstützung für die Delta-basierte Replikation und Versionierung komplexer Objektstrukturen (Diploma Thesis, german), 2003.

[23] C. Schneider. CoObRA 2 - light weight object oriented persistency and version support in symbiosis with text-based SCM. noch unveröffentlicht, 2007.

[24] C. Schneider, A. Zündorf, and J. Niere. CoObRA - a small step for development tools to collaborative environments. In *Workshop on Directions in Software Engineering Environments; 26th international conference on software engineering*. ICSE 2004, Scotland, 2004.

[25] Sun Microsystems, Inc. Java Core Reflection API and Specification. http://java. sun. com/j2se/1.3/docs/guide/reflection/.

[26] Sun Microsystems, Inc. Java Metadata Interface (JMI). http://java.sun.com/products/jmi/index.jsp.

[27] Sun Microsystems, Inc. Java object serialization specification. http://java.sun.com/, 2001.

[28] J. Szuba. *Graphs and Graph Transformations in Design in Engineering*. PhD thesis, Polish Academy of Sciences, Warsaw, 2005.

[29] W. F. Tichy. RCS - a system for version control. Purdue University, Department of Computer Sciences, West Lafayette, Indiana 47907, July 1985.

[30] T. Vinayak. Distributed Software Development with Perforce. http://www.perforce.com/perforce/papers/distributed.html.

[31] B. Westfechtel. Revisions- und Konsistenzkontrolle in einer integrierten Software-Entwicklungsumgebung. Informatik Fachberichte 280, Springer-Verlag, Berlin, Germany, 1991.

[32] B. White. Software Configuration Management Strategies and Rational ClearCase. Addison-Wesley, 2000.

[33] J. Wäsch and W. Klas. History Merging as a Mechanism for Concurrency Control in Cooperative Environments. In *RIDE-NDS*, pages 76–85, 1996.

[34] CVS - Concurrent Versions System. http://www.nongnu.org/cvs/, 2006.

[35] Hibernate - Relational Persistence for Java and .NET. http://www.hibernate.org/, 2006.

[36] SVN - Subversion. http://subversion.tigris.org/, 2006.

## A.2 Abbildungsverzeichnis

## A.3 CTR Dateiformat

Durch den Einsatz in textbasierten SCM Systemen ist es vorzuziehen, eine Änderung (Change) pro Textzeile abzulegen (s. Abschnitt 5.4.3). Damit war bereits das Trennzeichen zwischen Änderungen festgelegt: der Zeilenvorschub. Die Wahl der weiteren Trenn- und Markierungszeichen war für die Zweckerfüllung weitgehend irrelevant, und wurden daher in Form von einbuchstabigen Abkürzungen gewählt:

- Attribute der Änderung werden durch Semikolons separiert
- Jede Zeile wird mit einem Markierungszeichen wie folgt eingeleitet:
  - „h“ leitet Metadaten ein - die Zeile enthält Informationen über die Daten oder das Repository
  - „t“ leitet Daten über eine Transaktion ein (Transaction)
  - „c“ leitet Daten über eine Änderung ein (Change)
  - „u“ leitet Daten über eine bereits widerrufene Änderung ein (undone)
  - „]“ markiert das Ende des Stroms
  - „#“ markiert eine Zeile als Kommentar (zu verwerfende Daten)
  - „<“ markiert den Anfang eines vom SCM erkannten Konflikts
  - „=“ markiert die Trennung zwischen zwei Versionen in einem vom SCM erkannten Konflikt
  - „>“ markiert das Ende eines vom SCM erkannten Konflikts
- Handelt es sich bei der Zeile um Änderungsdaten folgt nach dem ersten Zeichen die Ordnungszahl der Art der Änderung. Mögliche Bedeutungen sind in *Change.Kind* definiert:

- Ein Objekt wird erstellt
- Ein Objekt wird gelöscht
- Ein Feld wird geändert (Wert hinzufügen, Wert entfernen oder Wert ändern)
- Ein Schlüssel eines qualifizierten Feldes wird gelöscht
- Verwaltungsinformationen (zum Beispiel zum Speichern des ID Suffix für das IdentifierModule)

Danach folgen, je nach Art der Änderung, die weiteren Daten. Beim Erzeugen von Objekten sind dies „Modifier“[1], Klasse des Objekts, ID des Objekts und die ID der übergeordneten Transaktion.

Für alle anderen Arten von Änderungen folgen „Modifier“, die ID des betroffenen Objekts, das geänderte Feld, der neue/hinzugefügte Wert, der alte/entfernte Wert, der Schlüssel im Fall eines qualifizierten Feldes und schließlich die ID der übergeordneten Transaktion. Alle Werte, die für die Darstellung der Änderung nicht benötigt werden, sind mit „-“ angegeben, was der Parser mit dem Java-Wert „null“ übersetzt.

Im Falle einer Transaktionszeile folgen dem „t“-Marker die ID der Transaktion, der Name der Transaktion, der Zeitstempel als Zahl (Millisekunden seit dem 01.01.1970 00:00 Uhr UTC), die ID der übergeordneten Transaktion und schließlich der „Modifier“.

[1] Zusätzliche Flags für die Änderung. So können zum Beispiel einzelne Änderungen als „lokal“ definiert werden, um sie nicht an Server zu senden.