Konzepte für den Einsatz versionierter Objektmodelle im Bauwesen

Dissertation zur Erlangung des akademischen Grades

Doktor - Ingenieur

an der Fakultät Bauingenieurwesen der Bauhaus-Universität Weimar

vorgelegt von

Torsten Richter

geboren in Gera

Gutachter:

1. Prof. Dr.-Ing. Karl E. Beucke, Bauhaus-Universität Weimar
2. Prof. Dr.-Ing. habil. Carsten Könke, Bauhaus-Universität Weimar
3. PD Dr.-Ing. habil. Volker Berkhahn, Leibniz Universität Hannover

Eingereicht am: 25. März 2009

Tag der Disputation: 6. Oktober 2009

Vorwort

Die vorliegende Arbeit entstand während meiner sechsjährigen Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl „Informatik im Bauwesen“ an der Bauhaus-Universität Weimar. Dort war ich am DFG-Projekt „Entwurf und Verifizierung einer CAD-Systemarchitektur zur Unterstützung der verteilten technischen Bearbeitung im Konstruktiven Ingenieurbau“ beteiligt, das im Rahmen des DFG-Schwerpunktprogramms 1103 mit dem Titel „Vernetzt-kooperative Planungsprozesse im Konstruktiven Ingenieurbau“ gefördert wurde. Daran schloss sich das Transferprojekt „Synchron-kooperative Projektbearbeitung mit strukturierten Objektversionsmengen“ an, um die entwickelten Konzepte in Zusammenarbeit mit einem Industriepartner – der Hochtief Construction AG, IKS – zu verifizieren und der Praxis näherzubringen.

Zum Gelingen dieser Arbeit haben viele Personen beigetragen, bei denen ich mich an dieser Stelle bedanken möchte. Zuerst wäre mein Mentor Herr Prof. Dr.-Ing. Karl Beucke zu nennen, der es mir ermöglichte, im Rahmen des interessanten Forschungsprojekts zu promovieren, und für die idealen Bedingungen am Lehrstuhl sorgte. Nicht zuletzt waren mir seine Anregungen und konstruktiven Diskussionen während der Promotionsphase eine wertvolle Hilfe.

Bei den Herren Prof. Dr.-Ing. Carsten Könke aus Weimar und Prof. Dr.-Ing. Volker Berkhahn aus Hannover bedanke ich mich für die Begutachtung der Arbeit und für die hilfreichen inhaltlichen Anmerkungen.

Weiterhin möchte ich den am DFG-Projekt direkt und indirekt beteiligten Mitarbeitern danken. Herr Prof. Dr.-Ing. Berthold Firmenich hatte als Mitinitiator und erster Bearbeiter des Projekts durch seine reichhaltigen praktischen Erfahrungen, seine methodische Vorgehensweise und seine kritischen, aber stets hilfreichen Hinweise einen wesentlichen Anteil am erfolgreichen Gelingen. Den Kollegen und Freunden Daniel Beer, Christian Koch und Bertie Olivier danke ich für die vielen interessanten fachlichen Diskussionen und die Unterstützung während der Bearbeitung des Themas. Die beiden erstgenannten standen auch für das Korrekturlesen der Arbeit zur Verfügung und gaben viele nützliche Hinweise.

Bei den Kollegen des Lehrstuhls möchte ich mich für die freundliche Arbeitsatmosphäre und die schönen Stunden bei gemeinsamen Veranstaltungen oder Ausflügen bedanken. Herrn Jens-Uwe Wagner bin ich für die Betreuung von Hard- und Software, für die informativen Gesprächen zu technischen Dingen sowie für seine offene und ehrliche Art zu Dank verpflichtet.

Abschließend möchte ich meiner Familie, meinen Freunden und Bekannten für die Unterstützung während der arbeitsintensiven Zeit bedanken. Mein besonderer Dank gilt Katja Witte für das entgegengebrachte Verständnis, den Rückhalt und die Aufmunterung nicht nur während der Promotionsphase.

Für Katja und Kim Sophie.

Kurzfassung

Bauwerke sind in der Regel Unikate, für die meist eine komplette und aufwändige Neuplanung durchzuführen ist. Der Umfang und die Verschiedenartigkeit der einzelnen Planungsaufgaben bedingen ein paralleles Arbeiten der beteiligten Fachplaner. Darüber hinaus ist die Bauplanung ein kreativer und iterativer Prozess, der durch häufige Änderungen des Planungsmaterials und Abstimmungen zwischen den Fachplanern gekennzeichnet ist. Mithilfe von speziellen Fachanwendungen erstellen die Planungsbeteiligten verschiedene Datenmodelle, zwischen denen fachliche Abhängigkeiten bestehen.

Ziel der Arbeit ist es, die Konsistenz der einzelnen Fachmodelle eines Bauwerks sicherzustellen, indem Abhängigkeiten auf Basis von Objektversionen definiert werden. Voraussetzung dafür ist, dass die Fachanwendungen nach dem etablierten Paradigma der objektorientierten Programmierung entwickelt wurden. Das sequentielle und parallele Arbeiten mehrerer Fachplaner wird auf Basis eines optimistischen Zugriffsmodells unterstützt, das ohne Schreibsperren auskommt. Weiterhin wird die Historie des Planungsmaterials gespeichert und die Definition von rechtsverbindlichen Freigabeständen ermöglicht. Als Vorbild für die Systemarchitektur diente das Softwarekonfigurationsmanagement, dessen Versionierungsansatz meist auf einem Client-Server-Modell beruht. Die formale Beschreibung des verwendeten Ansatzes wird über die Mengenlehre und Relationenalgebra vorgenommen, so dass er allgemeingültig und technologieunabhängig ist.

Auf Grundlage dieses Ansatzes werden Konzepte für den Einsatz versionierter Objektmodelle im Bauwesen erarbeitet und mit einer Pilotimplementierung basierend auf einer Open-Source-Ingenieurplattform an einem praxisnahen Szenario verifiziert. Beim Entwurf der Konzepte wird besonderer Wert auf die Handhabbarkeit der Umsetzung gelegt. Das betrifft im Besonderen die hierarchische Strukturierung des Projektmaterials, die ergonomische Gestaltung der Benutzerschnittstellen und der Erzielung von geringen Antwortzeiten. Diese Aspekte sind eine wichtige Voraussetzung für die Effizienz und Akzeptanz von Software im praktischen Einsatz. Bestehende Fachanwendungen können durch geringen Entwicklungsaufwand einfach in die verteilte Umgebung integriert werden, ohne sie von Grund auf programmieren zu müssen.

Abstract

Structures are normally unique which often require a new, complete and complex design. The extensiveness and diverseness of the several planning tasks cause a parallel work of the involved planners. Furthermore, the building planning is a creative and iterative process that is characterised by frequent changes of the planning material and coordinations between the professional designers. The planners create different data models with dependencies between each other.

The aim of this thesis is to ensure the consistency of the particular expert models by defining dependencies on the basis of object versions. A prerequisite is the development of all specialised applications with the established object-oriented programming paradigm. The sequential and parallel work of many planners is supported by an optimistic access model that does not require write locks. The history of the planning material will be stored additionally and the definition of release states is provided. The software configuration management served as a model for the system architecture whose versioning approach often relies on the client-server concept. The formal description of the used approach is done by the usage of the set theory and relational algebra to ensure its generality and independency from technologies.

On the basis of this approach concepts for the application of versioned object models in civil engineering are formulated and verified at a practical scenario with a pilot implementation based on an open source engineering platform. A great importance was attached on the usability of the implementation. This affects especially the structuring of the project material, the ergonomic design of the user interface as well as the achievement of short response times. These aspects are a precondition for the efficiency and acceptance of the software by the practical users. Existing expert applications can be easily integrated in the distributed environment without programming them from scratch.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Ratlosigkeit und Unzufriedenheit
sind die ersten Vorbedingungen
des Fortschritts.

(Thomas Alva Edison,
1847–1931)

1.1 Problembeschreibung

In Deutschland sind im Jahr 2007 durch Planungsfehler ca. 6,5 Mrd. Euro Verlust im Bauwesen entstanden. Diese enorme Geldsumme lässt sich mit folgender Rechnung abschätzen. Das Deutsche Institut für Wirtschaftsforschung in Berlin sagte für das Jahr 2007 ein Bauvolumen von 269 Mrd. Euro voraus (Bartholmai u. Gornig, 2008). Laut (Kochendörfer u. a., 2004) entstehen während der Planung Fehlerkosten in Höhe von 4 % bis 12 % der Investitionskosten, wovon noch einmal 30 % Entwurfs- und Planungsfehler sind (Jungwirth u. a., 1996). Bei einem angenommenen Anteil von 8 % für die Fehlerkosten, die sich dann auf 21,5 Mrd. Euro belaufen, ergibt sich die oben genannte Zahl.

Wodurch entsteht dieser volkswirtschaftlich beträchtliche Schaden? Bauwerke sind in der Regel Unikate oder werden nur in Kleinserien hergestellt. Die Einzelfertigung bedingt einen hohen relativen Kostenanteil für die Planung, da für jedes Bauwerk neue Pläne benötigt werden[1]. Im Gegensatz dazu erlauben in der stationären Massenfertigung einmal erstellte Pläne die Herstellung einer Vielzahl gleicher Produkte, die aus standardisierten Einzelteilen und Baugruppen bestehen. Erschwerend kommt hinzu, dass im Bauwesen verschiedene Gewerke – wie z. B. Architekturplanung, Tragwerksplanung, Technische Gebäudeausrüstung (TGA), Fassadenplanung usw. – am Entwurf und an der Erstellung eines Bauwerks beteiligt sind. Weiterhin gliedert sich die deutsche Bauindustrie in wenige große sowie viele kleine und mittlere Betriebe[2], was den Einsatz unterschiedlicher Planungssoftware zur Folge hat. Der kleinste Nenner des Zusammenarbeitens ist demnach der Austausch von Papier- oder mittlerweile digitalen Dokumenten, vorwiegend in proprietären[3] Formaten. Die Heterogenität der verwendeten Software erhöht dabei den

[1] s. (Hauschild, 2003)

[2] Laut (Bartholmai u. Gornig, 2008) gab es 2005 73930 Betriebe mit bis zu 59 Beschäftigten, die mit 68 % der im Bauhauptgewerbe Beschäftigten (499000 von 733800) 58 % des Gesamtumsatzes (43,94 von 76,33 Mrd. €) erwirtschafteten. Dagegen gab es nur 710 Betriebe mit über 100 Beschäftigten (137200), die 28 % (21,57 Mrd. €) zum Gesamtumsatz beitrugen.

[3] proprietarius ⟨lat., „Eigentümer“⟩

Datenverlust durch die nötige Umwandlung und die jeweils unterschiedliche Interpretation der Austauschformate.

Jede erstellte Zeichnung eines Bauwerks ist ein eigenständiges Dokument und bezieht sich mehr oder weniger auf andere Zeichnungen. Zu Beginn werden die vom Architekten erstellten Entwurfspläne als Kopie an die anderen Gewerke verteilt, die dort als Grundlage für deren zu erstellende Zeichnungen dienen. Später werden die Gewerkepläne zurückgeschickt und müssen mühsam von Hand abgeglichen werden, da mangels eindeutiger Identifikation von Zeichnungsobjekten herkömmliche CAD-Programme[4] keine Vergleichsfunktion bieten. Dabei kommt es oft vor, dass Fehler und Inkonsistenzen zwischen den Zeichnungen unerkannt bleiben und erst beim Bauen zu Tage treten.

Die baufertigen Pläne entstehen in einem schrittweisen Prozess, wobei die Planänderungen verschiedene Ursachen haben können:

- Geänderte Nutzungsanforderungen des Bauherrn,
- Geänderte Voraussetzungen für oder durch die Gewerke,
- Ausprobieren verschiedener Varianten zur Zeit- und/oder Kosteneinsparung,
- Beseitigung von Fehlern.

Durch die nicht vorhandene Modellierung der Abhängigkeiten zwischen den Objekten verschiedener Planungsdokumente ist es nicht möglich, von Änderungen betroffene Dokumente und deren Objekte zu ermitteln, was wiederum zu zeitaufwändiger und fehlerbehafteter manueller Prüfung führt.

Ist ein geprüfter Planungsstand erreicht, kann dieser rechtsverbindlich nach (DIN 6789-5, 1995) freigegeben werden, wobei die Dokumente archiviert und, falls sie digital vorliegen, schreibgeschützt werden müssen. Folglich ist es zweckmäßig und notwendig, die entstehenden Pläne zu versionieren. Dies kann manuell durch Kennzeichnung der Papierpläne, durch Anhängen von Versionsnummern an die Dateien oder durch ein Dokumenten-Management-System (DMS) geschehen.

(Firmenich, 2002) stellt in seiner Arbeit ein Konzept für die Versionierung von Objekten im Gegensatz zur Dokumentversionierung vor. Es setzt voraus, dass die Datenmodelle der Anwendungen mit dem objektorientierten Ansatz verwaltet werden. Die Bestandteile des Datenmodells werden demnach in Objekte mit Eigenschaften und Operationen aufgeteilt. Änderungen an den Objekten erzeugen im Laufe des Planungsprozesses neue Objektversionen, die dauerhaft[5] und unveränderlich gespeichert werden. Dadurch können nun Beziehungen feingranular zwischen Objektversionen definiert werden. Folgende Ziele lassen sich mit dem Konzept der Objektversionierung erreichen:

- Konsistentes[6] Produktmodell für den gesamten Bauplanungsprozess,

[4] CAD = Computer Aided Design ⟨engl., „rechnerunterstützte Konstruktion“⟩

[5] In der Informatik wird mit *Persistenz* das (dauerhafte) Speichern von Daten in nicht-flüchtige Datenspeicher bezeichnet.

[6] con ⟨lat., „zusammen“⟩ + sistere ⟨lat., „halten“⟩ → widerspruchsfrei

- Minimierung und Kontrolle von Redundanz, d. h. der mehrfachen Speicherung ein und derselben Information,
- Detaillierte Nachvollziehbarkeit von Änderungen,
- Variantenbildung,
- Vorhersage, welche Objektversionen von der Änderung bindender Objektversionen betroffen sind,
- Paralleles, verteiltes Arbeiten mit optimistischem Ansatz, der kein Sperren von Dokumenten erfordert.

Firmenich setzt die Objektversionen und die Beziehungen zwischen ihnen allgemeingültig mit der Mengenlehre um und definiert Operationen für die verteilte Bearbeitung mit der Mengenalgebra Feature-Logic. In der Umsetzung wird ein Datenbankentwurf für die Speicherung der Feature-Logic-Daten und die Formulierung der verteilten Operationen mit der Structured Query Language (SQL) vorgestellt. (Beer, 2005) zeigt, wie das Konzept in eine Systemarchitektur mit der klassischen Client[7]-Server[8]-Struktur auf Basis von langen Transaktionen eingebettet werden kann. Bestehende Ingenieuranwendungen werden beibehalten und um einen Workspace erweitert, der sie zur verteilten Bearbeitung befähigt. Der Workspace verfügt über die verteilten Operationen zur Kommunikation mit dem zentralen Server bzw. Project.

1.2 Ziel der Arbeit

Ziel der Arbeit ist es, die Konzepte der Arbeiten von (Firmenich, 2002) und (Beer, 2005) auf dem Gebiet der Objektversionierung fortzuführen und folgende allgemeine Ziele im Zusammenhang mit diesen beiden Arbeiten zu verfolgen:

- Befähigung objektorientierter Ingenieuranwendungen für die verteilte, parallele Bearbeitung von Planungsmaterial,
- Einfache Integration bestehender Anwendungen,
- Client-Server-Modell auf Basis langer Transaktionen,
- Unterstützung des iterativen Bauplanungsprozesses für mehrere Fachdomänen,
- Konsistentes Produktdatenmodell,
- Definition von anwendungsübergreifenden Abhängigkeiten zwischen Objektversionen,
- Zeit- und Kosteneinsparung beim Bauplanungsprozess durch erhöhte Effizienz.

[7] client ⟨engl., „Kunde“⟩
[8] server ⟨engl., „Anbieter“⟩

Der Hauptschwerpunkt dieser Arbeit gilt der Entwicklung von Konzepten für den Einsatz versionierter Objektmodelle im Bauwesen. Um ein rechnergestütztes Konzept erfolgreich in der Praxis anwenden zu können, muss zum einen das Kernproblem gelöst werden und die entwickelte Software für den Nutzer handhabbar sein. Handhabbarkeit bedeutet in diesem Zusammenhang nicht nur die Gestaltung einer nutzergerechten Oberfläche, sondern auch das Erreichen einer akzeptablen Ausführungsgeschwindigkeit im Umgang mit großen Datenmodellen.

Folgende Schwerpunkte sind im Speziellen zu untersuchen:

- **Abhängigkeiten:** Die Modellierung von Abhängigkeiten soll zum schnellen Erkennen inkonsistenter Planungszustände aufgrund von Änderungen dienen. Es stellt sich die Frage, wie Abhängigkeiten bzw. Bindungen zwischen Objektversionen anwendungsübergreifend erzeugt, bearbeitet und dem Nutzer anschaulich angezeigt werden können. Änderungen an bestehenden Klassenschemas von Fachanwendungen sind dabei zu vermeiden.

- **Benutzeroberflächen:** Die Versionsbeziehungen werden mit Versionsgraphen und die Abhängigkeiten zwischen Objektversionen mit Bindungsgraphen modelliert. Für die Darstellung und Navigation im Projekt ist es wegen der Komplexität nicht zweckmäßig, die kompletten Graphen anzuzeigen. Deshalb müssen Benutzeroberflächen entworfen werden, die je nach Aufgabe unterschiedliche Sichten auf das Projekt gewähren. Die Gestaltung von Benutzeroberflächen soll den Regeln der Software-Ergonomie entsprechen.

- **Vergleich und Zusammenführen von Dokumenten:** Der optimistische Ansatz der verteilten Planung erlaubt das parallele Bearbeiten des gleichen Planungsmaterials durch mehrere Akteure. Voraussetzung dafür ist, dass es auf einfache Art und Weise wieder fehlerfrei zusammengeführt und vorher eindeutig verglichen werden kann. Dieses Problem muss mit einem geeigneten Konzept und einer nutzergerechten Oberfläche gelöst werden.

- **Leistung:** Die Akzeptanz von Software bei Anwendern sinkt erheblich, wenn lange Wartezeiten bei der Verwendung auftreten, in denen nicht weiter gearbeitet werden kann. Für den Ansatz der Objektversionierung müssen dahingehend die verteilten Operationen, wie Speichern, Laden, Commit[9] und Update[10], sowie die Kommunikation vom Client zum Server untersucht und eventuell verbessert werden.

- **Systemarchitektur:** Die vorhandene Systemarchitektur ist für die Umsetzung der vorgenannten Punkte zu analysieren und anzupassen.

Bei Abschluss dieser Arbeit soll kein marktreifes Produkt veröffentlicht werden, sondern die Gewissheit bestehen, dass das Konzept tragfähig und eine Weiterentwicklung für den praktischen Einsatz lohnenswert erscheint, um die wirtschaftlichen Verluste durch die bestehenden Arbeitsweisen im Bauplanungsprozess zu verringern.

[9] Senden von Planungsmaterial vom lokalen Rechner zum zentralen Server
[10] Holen von Planungsmaterial vom zentralen Server zum lokalen Rechner

1.3 Lösungsansatz und Vorgehensweise

Das verteilte Arbeiten im Bauplanungsprozess wird auf Basis versionierter Objektmodelle, mit der Speicherung von Objektversionen als kleinster Einheit, umgesetzt. Analog zur Systemarchitektur von (Beer, 2005) werden die Datenmodelle einerseits mit einem textbasierten Versionsverwaltungssystem versioniert und anderseits Metadaten und sonstige benötigte Daten, wie z. B. Versionsbeziehungen, mit der Mengenalgebra Feature-Logic modelliert und gespeichert. Jedoch wird in dieser Arbeit die Bedeutung des Dokuments gegenüber der alleinigen Objektversionierung wieder stärker in den Vordergrund gerückt.

Systemarchitektur: Innerhalb der vorliegenden Systemarchitektur wird für eine bessere Handhabbarkeit sowie für die Unterstützung verschiedener Fachdomänen das Planungsmaterial auf dem zentralen Server und auf den lokalen Rechnern mit einer neuen Hierarchie gegliedert. Weiterhin erfolgt die Kommunikation zwischen den Clients und dem Server nun ausschließlich über standardisierte Protokolle. Im Allgemeinen erfährt die Systemarchitektur eine Überarbeitung des Klassenentwurfs, der Komponenten und der Datenspeicherung.

Abhängigkeiten: Die vorhandenen persistenten Objektversionsidentifikatoren (POVID) dienen zur konsistenten Definition von Abhängigkeiten bzw. Bindungen. Abhängigkeiten werden nicht im Datenmodell der Anwendung, sondern außerhalb in der Feature-Logic gespeichert. Bindungen dürfen gleichwohl innerhalb von Anwendungen erzeugt, bearbeitet und gelöscht werden. Damit Bindungen zwischen Datenmodellen verschiedener Programme definiert werden können, muss der flexible Import eines fremden Datenmodells in eine Anwendung möglich sein. Dahingehend werden neue Serialisierungskonzepte entwickelt, die zudem Geschwindigkeitsvorteile gegenüber der Java-Serialisierung bieten.

Die Erzeugung und Bearbeitung von Abhängigkeiten stellt außerdem Anforderungen an die Benutzeroberfläche. Prototypisch wird das Vorgehen am Beispiel des Lastabtrags gezeigt, der an eine Geometrie gebunden ist.

Benutzeroberflächen: Für die verschiedenen verteilten Operationen werden entsprechende Benutzeroberflächen und Dialoge vorgestellt, die jeweils nur eine gewünschte Sicht auf das versionierte Modell wiedergeben, um die Komplexität zu reduzieren. Größtenteils reichen dann für die Benutzeroberflächen einfache grafische Komponenten, wie Tabellen und Bäume, aus, die in geeigneter Art und Weise kombiniert werden. Die Mengenalgebra Feature-Logic ist für die Benutzeroberflächen um einen neuen Datentyp zur Speicherung des Datums und um eine Selektion auf Basis atomarer Werte zur Definition von Filtern zu erweitern.

Vergleich und Zusammenführen von Dokumenten: Diese beiden Operationen sind für den Anwender nur dann praktikabel, wenn die Dokumente wie im verwendeten Programm üblich dargestellt werden. Die Objektversionsidentifikatoren stellen wieder die Eindeutigkeit und Konsistenz sicher. Für einen einfachen Vergleich verteilt bearbeiteter Dokumente werden hinzugefügte, bearbeitete, gelöschte und unveränderte Objekte farblich hervorgehoben. Das Zusammenführen von Dokumenten geschieht nicht automatisch, da die Planungshoheit des Ingenieurs durch die Software nicht in Frage gestellt werden

soll. Vielmehr wird die Kommunikation zwischen den Planern weiterhin als notwendig angesehen. Eine Unterstützung kann nur beim eigentlichen Vorgang des Zusammenführens durch geeignete Interaktionsmöglichkeiten und Nutzeroberflächen erfolgen.

Leistung: Leistungsverbesserungen durch kürzere Ausführungszeiten von Operationen können an verschiedenen Stellen der Systemarchitektur erreicht werden. Wesentliche Zeiteinsparungen sind durch den Einsatz von bekannten Optimierungsmethoden auf der Datenbankseite und durch die effizientere Datenspeicherung in das Dateisystem auf dem Rechner des Bearbeiters mit Hilfe von neuen Serialisierungsmethoden möglich. Auch die Kommunikation zwischen Client und Server kann durch Verringerung der übertragenen Datenmenge und Anzahl der Zugriffe wesentlich beschleunigt werden, was zum Beispiel die Wartezeiten in Nutzeroberflächen merklich verkürzt.

1.4 Gliederung

Die vorliegende Arbeit gliedert sich in sechs Kapitel und einen Anhang, wobei die Kapitel 3 bis 5 den Schwerpunkt der Arbeit bilden.

Kapitel 2 beleuchtet den aktuellen Stand der Technik und Wissenschaft auf dem Gebiet der kooperativen, verteilten Bearbeitung im Bauwesen, an dem intensiv geforscht wurde und wird. Trotz aller Bemühungen dominiert in Deutschland im Hinblick auf die Bearbeitung technischer Zeichnungen immer noch der dokumentbasierte Datenaustausch auf Basis von zweidimensionalen Plänen, der für eine kooperative Bearbeitung von Planungsmaterial nicht förderlich ist. Weltweit gibt es Initiativen und mittlerweile auch kommerzielle Software, um diesen Zustand zu verbessern – sei es durch dreidimensionale Gebäudeinformationsmodelle oder durch Versionierung von Dokumenten. Diese Konzepte werden analysiert und bewertet.

Als positives Beispiel für die erfolgreiche, parallele Bearbeitung wird das Softwarekonfigurationsmanagement näher erläutert, welches auf der Versionierung von Textdateien im Softwareentwicklungsprozess beruht. Die Konzepte und Methoden sind mittlerweile so ausgereift und allgemein akzeptiert, dass diese den Softwareerstellungsprozess positiv beeinflusst haben und dort nicht mehr wegzudenken sind. Im Gegensatz zu den Bauwerksmodellen in zwei- oder dreidimensionaler Form besitzen die Quellcodedateien eine einfache sequentielle Struktur, die vom Menschen leicht gelesen werden kann. Mit einfachen Algorithmen unterstützt der Rechner den Programmierer beim Vergleichen und Zusammenführen zweier Dateien, wobei das Zeichen das kleinste Element darstellt. Soll dieses Konzept auf objektorientierte Modelle angewendet werden, so darf nicht das Dokument als kleinstes Element angesehen werden, sondern das Objekt. Somit wird aus der Dokumentversionierung die Objektversionierung, die näher beschrieben wird und als Grundlage für die folgenden Kapitel dient.

Abschließend werden Grundbegriffe der Software-Ergonomie aufgeführt und für den Entwurf von Benutzerschnittstellen näher betrachtet. Ein Reihe internationaler Normen stellt

Anforderungen zur Konstruktion grafischer Benutzeroberflächen auf, um die Gebrauchstauglichkeit von Software zu erhöhen. Diese werden mit ihren wesentlichen Begriffen und Konzepten vorgestellt. Die Umsetzung grafischer Benutzeroberflächen mit der Programmiersprache Java wird danach allgemein und an einem Beispiel erläutert.

Kapitel 3 beschreibt die Erweiterung des mathematischen Modells zur Objektversionierung, das in zwei Disserationen am Lehrstuhl „Informatik im Bauwesen“ entwickelt wurde. Die mengentheoretische Beschreibung fokussiert in der vorliegenden Form stärker auf die Zusammenfassung von Objekten zu Dokumenten im Hinblick auf gewohnte Arbeitsweisen im Bauplanungsprozess. Gleichwohl bleibt das Objekt die kleinste versionierbare Einheit. Weiterhin hat die Modellierung der Abhängigkeiten durch die Hinwendung zu Dokumenten eine Weiterentwicklung erfahren. Ausgehend vom mathematischen Modell werden die wesentlichen verteilten Operation beschrieben.

Kapitel 4 stellt die wesentlichen Konzepte und Umsetzungen der Systemarchitektur vor, die unabhängig von einer speziellen Fachanwendung sind. Zunächst werden die wichtigsten Klassen und Schnittstellen zur Umsetzung der Client-Server-Architektur spezifiziert und danach einzelne Themengebiete angesprochen. Aufgrund der Vielzahl von Objekten in einem Modell muss ein geeignetes Serialisierungsverfahren für die persistente Speicherung im Arbeitsbereich des Planers und auf dem Versionierungsserver gefunden werden. Darauf aufbauend wird ein Ansatz vorgestellt, der Abhängigkeiten zwischen Datenmodellen anwendungsunabhängig auf Basis von Objektversionen definiert. Außerdem spielt die Handhabbarkeit versionierter Objektmodelle eine wichtige Rolle, was sowohl die notwendigen Benutzeroberflächen als auch den Umgang mit großen Datenmodellen betrifft. Die technische Umsetzung erfolgt in der plattformunabhängigen Programmiersprache Java.

Kapitel 5 zeigt, basierend auf dem vorhergehenden Kapitel, die prototypische Anwendung auf den geometrischen Lastabtrag in der Vorbemessung von Bauwerken integriert in die Open-Source-Ingenieurplattform CADEMIA (Firmenich u. a., 2008). Die vorhandenen Anwendungen werden zur verteilten Bearbeitung befähigt, indem sie die entworfene Systemarchitektur um anwendungsspezifische Funktionen erweitern. Auf dieser Basis wird der Objektversionierungsansatz verifiziert.

Schlussendlich fasst das *Kapitel 6* die Arbeit in Kurzform zusammen und diskutiert die gewonnenen Ergebnisse und leitet daraus Erkenntnisse ab. Die ermittelten Vor- und Nachteile sind entscheidend für eine praktische Verwertbarkeit des Objektversionierungskonzepts in Bauplanungsprozess. Abschließend werden Anknüpfungspunkte für weitere Forschungsarbeiten aufgezeigt.

2 Stand der Technik und Wissenschaft

Technik ist Mittel zum Zweck, nicht Selbstzweck.

(Carl Friedrich von Weizsäcker, 1984)

2.1 Grundlagen und Begriffe

2.1.1 Dokumentation

Information (Claus u. Schwill, 1993)**:** Der Begriff Information[1] wird von vielen Wissenschaftsbereichen in teils unterschiedlichen Bedeutungen verwendet. Die Informatik versteht sich als *„Wissenschaft von der systematischen Verarbeitung von Informationen, trotzdem wurde der Begriff Information bisher kaum präzisiert.“*. Eine Information besteht aus mindestens drei Teilen:

- einem syntaktischen Teil, der die zulässige Struktur der Bausteine, aus denen sich die Information zusammensetzt, beschreibt,
- einem semantischen Teil, der die Bedeutung der Information angibt,
- einem pragmatischen Teil, aus dem sich der Zweck der Information und die erhofften Handlungen ergeben.

Daten (Fischer u. Hofer, 2008)**:** *„Alles, was sich in einer für die Datenverarbeitungsanlage, den Computer, erkennbaren Weise codieren, speichern und verarbeiten lässt, also abstrahierte und ‚computergerecht‘ aufbereitete Informationen.“*

Datenträger (Fischer u. Hofer, 2008)**:** *„Gesamtheit aller nicht flüchtigen, physikalischen Medien zur dauerhaften und nicht auf ständige Energiezufuhr angewiesenen Speicherung von Daten.“*

Datei: Eine Datei im Sinne der elektronischen Datenverarbeitung besteht aus einer Folge von Bytes, die dauerhaft auf einem Datenträger in einem Container abgelegt und mit einem Dateinamen versehen ist. Ohne Kenntnis über die Struktur in der Datei lässt sich der eigentliche Inhalt nicht wiederherstellen. Üblicherweise zeigt die Dateierweiterung – ein am Ende des Dateinamens durch einen Punkt abgetrennter Namensbestandteil – die Art von Informationen in der Datei an.

[1] informare ⟨lat., „bilden, eine Form geben“⟩

(Claus u. Schwill, 1993) unterscheiden drei verschiedene Dateiorganisationen:

- **Sequentielle Dateien:** *„Die Daten sind fortlaufend in der Reihenfolge ihrer Eingabe gespeichert und können nur in dieser Reihenfolge wieder abgerufen werden.“*
- **Direkte Dateien:** *„Der Zugriff erfolgt über einen Schlüssel, aus dem mit einer Tabelle oder mit Hilfe eines Algorithmus die Adresse des zugehörigen Datensatzes bestimmt wird (Hash-Verfahren). Der Algorithmus ist Teil der Dateidefinition.“*
- **Indexsequentielle Dateien:** *„Sie bilden eine Mischform aus den beiden Dateiorganisationen. Eine Folge von Datensätzen, die sequentiell nach einem Merkmal geordnet ist, wird in gleich große Einheiten unterteilt, die unter einem Index zusammengefasst werden. Als Index dient jeweils das größte (bzgl. der Ordnung) in der Einheit vorkommende Merkmal. Innerhalb einer Einheit können die Datensätze nur sequentiell durchlaufen werden.“*

Dateiformat: Das Dateiformat gibt die Struktur der Daten innerhalb einer Datei vor, um Informationen speichern und lesen zu können. Nach der Offenheit kann man Dateiformate einteilen in:

- **Proprietäre Dateiformate:** Sie werden in der Regel von einem Hersteller entwickelt und sind nach außen nicht oder schlecht dokumentiert. Für die Verwendung solcher Formate in Fremdprodukten sind meistens Lizenzgebühren zu entrichten. Erreicht das Format einen mehr oder weniger ausgeprägten Monopolstatus im Markt, so ist der Urheber meist daran interessiert, diesen Status zu bewahren, um die Kunden an sein Produkt zu binden.[2]
- **Offene Dateiformate:** Sie können durch jedermann ohne rechtliche Einschränkungen verwendet und implementiert werden. Offene Dateiformate vermeiden die strikte Bindung an die Software eines bestimmten Herstellers, da eine Vielzahl von Programmen die Daten lesen und schreiben können.

(Beucke, 2002) führt zwei wesentliche Forderungen für Dateiformate auf.

- Der Zugriff auf die Daten muss durch Einhaltung von Integrität und Konsistenz sicher sein.
- Die Daten müssen auch noch mit späteren Programmversionen, die durch funktionale Erweiterungen und konzeptionelle Änderungen gekennzeichnet sind, gelesen werden können. In der Informatik wird dieses Prinzip mit Abwärtskompatibilität bezeichnet (Wikipedia, 2008a).

[2] Beispielsweise führte die Firma Autodesk 2006 die sogenannte TrustedDWG®-Technologie in das Format DWG ein, die dem Nutzer beim Öffnen einer Zeichnung in einem Autodesk-Produkt eine Meldung anzeigt, falls die Zeichnung nicht mit einem Autodesk-Produkt oder mit einer RealDWG®-Lizenz erstellt wurde (Autodesk, 2008). Als die Open Design Alliance (ODA) – eine Organisation, die ihren Mitgliedern die Bibliothek *OpenDWG* zum Lesen und Schreiben von DWG-Dateien zur Verfügung stellt – den TrustedDWG-Abschnitt in ihre DWG schrieb, wurde sie von Autodesk am 22.11.2006 wegen angeblich unberechtigter Nutzung der Marke *AutoCAD* verklagt. Im April 2007 wurde die Anklage fallengelassen als die ODA die Unterstützung für TrustedDWG aus ihrer Bibliothek entfernte. Im Gegenzug entschärfte Autodesk die Warnmeldung in AutoCAD 2008. (Quelle: (Wikipedia, 2008c))

Textdatei: Eine Textdatei speichert Text in einer Datei. (Gumm u. Sommer, 2006) definieren Text als Folge von Buchstaben, Ziffern, Satz- und Spezialzeichen. Einige Spezialzeichen, auch Sonderzeichen genannt, steuern den Textfluss und sind nicht druckbar. Dazu gehören u. a. das CR-Zeichen[3] für einen Zeilenumbruch und das Tabulatorzeichen für eine Einrückung. Wichtigstes Merkmal einer Textdatei ist, dass sie ohne spezielle Software vom Menschen gelesen und interpretiert werden kann.

Als Zeichensatz wird häufig noch ASCII[4] und mittlerweile verstärkt Unicode verwendet. Der ursprüngliche 7-Bit-ASCII-Zeichensatz definiert 128 Zeichen, welche das lateinische Alphabet, arabische Ziffern, Satzzeichen und Steuerzeichen umfassen, und wurde um mehrere 8-Bit-Varianten mit jeweils 256 Zeichen durch die International Organization for Standardization (ISO) erweitert, von denen die Variante Latin-1 zusätzliche Zeichen für die westeuropäischen Sprachen enthält (ISO/IEC 8859-1, 1998). Unicode wurde schließlich wegen der vorhandenen Zeichensatzvielfalt als ein generischer Zeichensatz vom *Unicode Consortium* entworfen, der jedem mehr oder weniger gebräuchlichen Zeichen einen digitalen Code zuweist. Aktuell ist die Version 5.1.0 (Unicode Consortium, 2008), in der über 100.000 Zeichen enthalten sind. Die ISO standardisierte zeitgleich den Universal Character Set (UCS), engl. *universeller Zeichensatz*, der praktisch dem Unicode entspricht (ISO/IEC 10646-4, 2008). Ursprünglich für 65536 Zeichen[5] ausgelegt, umfasst Unicode heute 17 Ebenen zu je 16 Bit, womit sich insgesamt 1.114.112 verschiedene Zeichen speichern lassen. Aus Kompatibilitätsgründen entsprechen die ersten 256 Zeichen dem Latin-1-Zeichensatz. Es existieren verschiedene Kodierungen, um Unicode-Zeichen durch Bytefolgen abzubilden. Die beiden bekanntesten sind UTF-8[6] und UTF-32. UTF-32 verwendet generell für jedes Zeichen 4 Byte und benötigt dadurch sehr viel Speicherplatz. UTF-8 erlaubt eine variable Anzahl von Bytes: 1 Byte für die ersten 128 Zeichen des ASCII-Zeichensatzes, 2 bis 4 Byte für alle weiteren Zeichen. Mit 7-Bit-ASCII kodierte Texte gleichen damit denen, die in UTF-8 gespeichert wurden.

XML-Datei: Die Extensible Markup Language (XML) ist eine Auszeichnungssprache, die hierarchische Strukturen abbildet und in einer Textdatei speichert. Das World Wide Web Consortium (W3C) ist verantwortlich für die Definition der Spezifikation und hat am 29. September 2006 die zur Zeit aktuelle vierte Version veröffentlicht (W3C, 2006). Die Grammatik kann durch eine Document Type Definition (DTD) oder durch das modernere *XML Schema* (W3C, 2004) vorgegeben werden. Die Strukturierung innerhalb einer XML-Datei erfolgt durch XML-Elemente, deren Name frei vergeben werden kann. Binärdaten sind innerhalb einer XML-Datei nicht erlaubt. Ziel von XML ist es, neue Datenformate zu definieren, die maschinenlesbar, firmen- und plattformübergreifend sind (Fischer u. Hofer, 2008). XML-Parser sind Programme, die XML-Dateien lesen, interpretieren und teilweise auch verifizieren können.

Dateisystem (Fischer u. Hofer, 2008)**:** Das Dateisystem ist als Teil eines Betriebssystems ein *„Modell zur Verwaltung und Ablage von Dateien auf einem Sekundärspeicher“*. Mit Se-

[3] CR = carriage return ⟨engl., „Wagenrücklauf“⟩
[4] ASCII = American Standard Code for Information Interchange
[5] 16 Bit = 2^{16} = 65536
[6] UTF = Unicode Transformation Format

kundärspeicher sind hier Plattenspeicher mit hoher Kapazität, mäßiger Zugriffszeit und relativ geringen Kosten pro Byte gemeint, die Daten persistent speichern. Festplatten, CD, DVD und Flashspeicher fallen in diese Kategorie. *„Dateisysteme vermitteln zwischen der Speicherungs-Hardware, die in Zylinder, Sektoren und Cluster aufgeteilt ist, und dem Betriebssystem.“* Die äußere Strukturierung ist meist hierarchisch ausgelegt und unterteilt sich ausgehend von einem Wurzelverzeichnis in Unterverzeichnisse und Dateien. Verbreitete Dateisysteme sind FAT, FAT32 und NTFS für Windows-Betriebssysteme sowie ext3, ext4, ReiserFS, HFS+ und XFS für Unix-ähnliche Betriebssysteme.

Dokument: (Fischer u. Hofer, 2008) definieren ein Dokument unter zwei Gesichtspunkten:

1. allgemein: Schriftstück;
2. speziell im Umfeld der Datenverarbeitung: Ergebnis der anwenderseitigen Arbeit mit einer Applikation sowie Datei mit den entsprechenden Ergebnissen: Schriftstück, Tabelle, Datenbank, Quellenprogramm, Objektprogramm, Grafik, ...

Nach (DIN ISO 11442, 2006) ist ein Dokument *„eine festgelegte und strukturierte Menge von Informationen, die als Einheit verwaltet und zwischen Anwendern und Systemen ausgetauscht werden kann.“*

Ein Dokument ist in der Regel rechtsverbindlich, was zum Beispiel durch den Begriff *Dokumentenechtheit* zum Ausdruck kommt. Das Justizkommunikationsgesetz aus dem Jahr 2005 (JKomG, 2005) erlaubt nun auch die Verwendung von elektronischen Dokumenten in der Justiz durch Einfügen des Paragrafs 130a in die Zivilprozessordnung (ZPO, 2008):

> § 130a Abs. 1. ZPO: *„Soweit für vorbereitende Schriftsätze und deren Anlagen, für Anträge und Erklärungen der Parteien sowie für Auskünfte, Aussagen, Gutachten und Erklärungen Dritter die Schriftform vorgesehen ist, genügt dieser Form die Aufzeichnung als elektronisches Dokument, wenn dieses für die Bearbeitung durch das Gericht geeignet ist. Die verantwortende Person soll das Dokument mit einer qualifizierten elektronischen Signatur nach dem Signaturgesetz versehen.“*

Die Beweiskraft elektronischer Dokumente stellt Paragraf 371a der Zivilprozessordnung durch die formale Gleichstellung mit Urkunden sicher:

> § 371a Abs. 1 ZPO: *„Auf private elektronische Dokumente, die mit einer qualifizierten elektronischen Signatur versehen sind, finden die Vorschriften über die Beweiskraft privater Urkunden entsprechende Anwendung. [...]“*

Dokumentation (DIN 6789-1, 1990)**:** *„Eine Dokumentation ist die Summe der für einen bestimmten Zweck vollständig zusammengestellten Dokumente.“*

Dokumentensatz (DIN 6789-2, 1990)**:** *„Ein Dokumentensatz ist eine für einen bestimmten Zweck als Einheit gehandhabte Zusammenfassung von Dokumenten.“*

Freigabe (DIN 6789-5, 1995)**:** *„Die Freigabe ist einen bestimmten Anweisungen entsprechende Genehmigung nach abgeschlossener Prüfung. Freigaben können objekt- oder*

tätigkeitsbezogen sein, z. B. Zeichnungsfreigabe (Freigabe der fertiggestellten Zeichnung) → Konstruktionsfreigabe (Freigabe zum Konstruieren). [...]“

Abbildung 2.1 zeigt den Freigabeprozess. Das Freigabeobjekt, hier als Gegenstand bezeichnet, wird einer Prüfung auf Freigabe unterzogen und gegebenenfalls mehrfach geändert. Als Ergebnis erhält man eine Freigabe (Release), die einem freigegebenen Dokumentenstand entspricht.

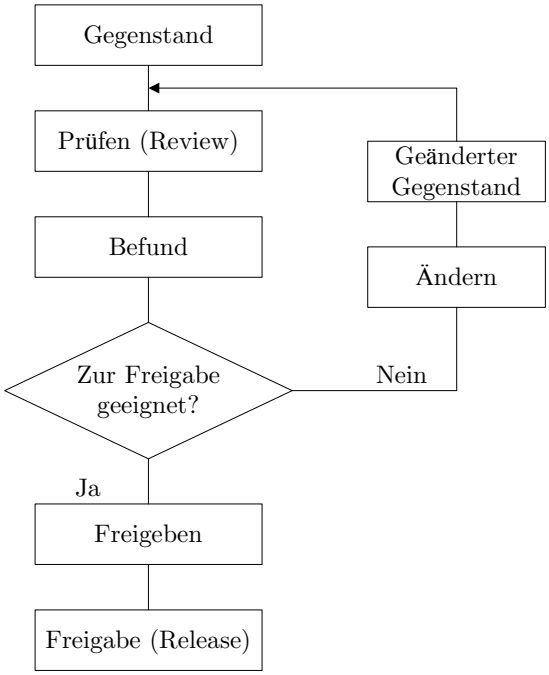


Abbildung 2.1: Ablauf einer Freigabe nach (DIN 6789-5, 1995)

„Bei der rechnergestützten Dokumentenverwaltung müssen mit jeder Freigabestufe [...] die Rechte für den Schreibzugriff auf die betroffenen Dokumente durch entsprechende Systemsteuerung eingeschränkt werden. Nach der Generalfreigabe müssen die Dokumente für alle Schreibzugriffe gesperrt sein, während Lesezugriffe von allen Freigaben unabhängig sein können oder nach der Generalfreigabe sogar einem größeren Anwenderkreis gestattet werden.“

Versionierung: Eine Version in der Informatik ist ein eindeutiger Zustand eines Dokuments, eines Objekts oder eines sonstigen Datencontainers. Versionierung ist das Aufzeichnen bzw. Archivieren[7] von Versionen sowie das Schützen dieser vor nachträglichen Änderungen. Für jede versionierte Einheit entsteht ein gerichteter, azyklischer Graph[8], der im Fall der linearen Versionierung keine Verzweigungen enthält. Versionen sind dann *Revisionen* bezüglich der Vorgängerversion. Gehen von einem Knoten mehrere parallele Versionen aus, so entstehen *Varianten*. Diese können später durch einen Merge-Vorgang[9] zusammengeführt werden. Diese Form der Versionierung wird als nichtlinear bezeichnet.

[7] archivum ⟨lat., „Regierungs-, Amtsgebäude“⟩
[8] s. (Pahl u. Damrath, 2000)
[9] to merge ⟨engl., „vereinigen, vermischen, verbinden“⟩

2.1.2 Objektorientierung

Objektorientierung: Die Objektorientierung ist ein Programmierparadigma, bei dem die Daten und die dazugehörigen Methoden in Einheiten zusammengefasst werden. Diese Einheiten heißen zur Laufzeit Objekte und werden durch Klassen beschrieben.

Ziele der Objektorientierung:

- **Realität:** Man verspricht sich durch die Objektorientierung eine bessere Modellierung realer Probleme gegenüber anderen Programmierparadigmen.
- **Hierarchien:** Das Vererbungskonzept erlaubt die Abbildung hierarchischer Zusammenhänge, wobei die übergeordneten Klassen Gemeinsamkeiten definieren und die untergeordneten Klassen Spezialisierungen darstellen. Eine Schnittstelle schreibt Methodensignaturen für implementierende Klassen vor und ermöglicht so den einfachen Austausch verschiedener Implementierungen oder Programmmodule.
- **Datenkapselung:** Als privat deklarierte Attribute kann nur das Objekt selbst lesen und schreiben, die als geschützt (protected) markierten Attribute nur die abgeleiteten Objekte. Ein Zugriff ist nur indirekt über die Methoden möglich. Damit besitzen die Objekte die Hoheit über ihre Daten und können ihre Konsistenz sicherstellen.
- **Wiederverwendung:** Da Objekte eine Einheit aus Daten und Logik bilden, lassen sie sich einfacher in anderer Software wiederverwenden, die die gleiche Funktionalität benötigt.

Klasse: Eine Klasse in der Objektorientierung ist ein komplexer Datentyp, der Attribute (Eigenschaften) und Methoden (Operationen) definiert. Attribute sind entweder einfache oder komplexe Datentypen. Klassen abstrahieren und modellieren reale Objekte eines Fachproblems und können voneinander abgeleitet sein, wobei die Kindklasse von der Vaterklasse alle nicht-privaten Attribute und Methoden erbt. Dieses Prinzip ist als Vererbung bekannt. Klassen werden in Textdateien als Quelltext eines Programms gespeichert.

Generischer Datentyp (Gumm u. Sommer, 2006)**:** Ein generischer Datentyp verwendet über Typparameter abstrakte Datentypen, die erst zur Laufzeit festgelegt werden. Der Einsatz ist innerhalb von Methoden, Klassen und Schnittstellen möglich und erlaubt die Umsetzung der typsicheren Programmierung. Häufig werden sie bei Behälter-Datentypen (Container-Klassen), wie z. B. Sets, Maps[10], Listen usw., eingesetzt, damit die Einschränkung auf einen Datentyp vorgenommen werden kann, ohne jeweils eine eigene Container-Klasse für jeden Datentyp zu schreiben.

Objekt: Ein Objekt in der Programmierung ist ein Exemplar einer Klasse. Von einer Klasse können zur Laufzeit des Programms eine endliche, von der Größe des Arbeitsspeichers[11] begrenzte Anzahl von Objekten erzeugt bzw. instanziiert werden. Jedes Objekt besitzt einen eindeutigen Zustand, der von der Belegung der Attribute abhängig ist, und

[10] In der Literatur findet man auch den Oberbegriff *Assoziatives Array* für diese Datenstruktur. Sie speichert Schlüssel-Wert-Paare.

[11] Mit Arbeitsspeicher ist ein sehr schneller und flüchtiger Speicher gemeint, der oft auch als Random Access Memory (RAM) bezeichnet wird.

ein durch seine Methoden definiertes Verhalten (Booch, 1995). Objekte sind transient, das bedeutet, dass sie mit Beendigung des Prozesses aus dem Arbeitsspeicher gelöscht werden oder bei einer Stromunterbrechung unwiederbringlich verloren gehen.

Abbildung 2.2: Darstellung eines Objekts

Objekthandle: Das Objekthandle ist als temporärer Identifikator ein Zeiger auf den Bereich im Arbeitsspeicher, in dem das Objekt abgelegt ist, und dient dem Zugriff auf das Objekt. Die Eindeutigkeit muss im Identifikatorenraum sichergestellt sein. Zur Laufzeit des Programms ist der Identifikatorenraum gleich dem zur Verfügung stehenden Adressbereich. Nach Beendigung des Programms oder des Prozesses verfällt das Objekthandle ebenso wie das zugehörige Objekt.

Objektmodell: Objekte stehen zueinander in Beziehung, da sie sich untereinander über die Attribute und Methoden referenzieren können. Meist gibt es ein Wurzelobjekt, das innerhalb des betrachteten Objektmodells nicht referenziert wird. Die Menge zueinandergehöriger Objekte heißt Objektmodell. Objektorientiert programmierte Anwendungen bilden demnach auch ein Objektmodell.

Objektmodelle lassen sich durch einen Graphen abbilden, wobei die Knoten Objekte und die Kanten Objektreferenzen repräsentieren. Die Kanten werden durch einen durchgezogenen Pfeil mit offener Pfeilspitze dargestellt. Abbildung 2.3 zeigt beispielhaft ein Objektmodell als Graph mit dem Objekt *a* als Wurzelobjekt.

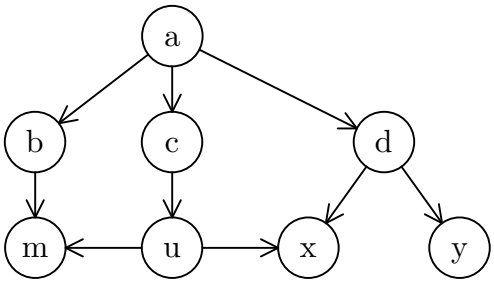
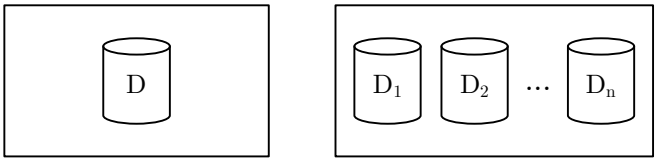


Abbildung 2.3: Objektmodell mit Objektreferenzen

Reflexion (Ullenboom, 2008)**:** Die Reflexion ermöglicht einem objektorientierten Programm, zur Laufzeit Informationen über die Struktur seiner Klassen und Objekte zu gewinnen. Die Werte von Attributen können ausgelesen und unter bestimmten Bedingungen auch verändert werden.

Datenmodell: Objektorientierte Anwendungen verwalten die vom Nutzer erzeugten Daten in einem Objektmodell, das ein Wurzelobjekt als Startpunkt enthält. Dieses spezielle Objektmodell soll im Rahmen dieser Arbeit als Datenmodell bezeichnet werden.

Eine Anwendung mit einem Single Document Interface (SDI) kann nur ein Datenmodell gleichzeitig verwalten, eine Anwendung mit einem Multiple Document Interface (MDI) hingegen eine endliche Anzahl in Abhängigkeit vom freien Arbeitsspeicher (s. Abbildung 2.4).



(a) SDI-Anwendung *(b)* MDI-Anwendung

Abbildung 2.4: Anwendungstypen nach Anzahl der verwalteten Datenmodelle

Informationsspeicherung: Das Datenmodell liegt während der Programmausführung flüchtig (transient) im Arbeitsspeicher und geht schon bei kurzfristiger Stromunterbrechung, wie sie bei einem Stromausfall ohne eine vorgeschaltete Unterbrechungsfreie Stromversorgung (USV) auftritt, verloren. Das während der Bearbeitung am Rechner geschaffene Ergebnis soll aber dauerhaft (persistent) gespeichert und bei Wiederaufnahme der Arbeit weiter bearbeitet werden können. Die Informationsspeicherung überführt das Datenmodell vom Arbeitsspeicher auf einen nicht flüchtigen Datenspeicher. Drei Formen der Speicherung bieten sich an:

- **Speicherung in eine Datei mit einem bestimmten Dateiformat:** Dieser Weg wird bevorzugt gewählt, da ein leichter Datenaustausch zwischen verschiedenen Programmen erfolgen kann, sofern die Dateiformatspezifikation allen zugänglich und als Standard akzeptiert ist. Der Programmierer muss bei fehlenden oder nicht zugänglichen Bibliotheken selbst das korrekte Schreiben, Lesen und Interpretieren der Datei implementieren, was bei komplexen Datenmodellen und Dateiformaten einen erheblichen Zeitaufwand bedeutet.

- **Serialisierung:** Das komplexe Objektmodell wird in eine speicherfähige, sequentielle Form überführt. Das Gegenstück dazu ist die Deserialisierung, die nur dann erfolgreich sein kann, wenn sich das zugrundeliegende Klassenschema nicht geändert hat. Voraussetzung für die Serialisierung ist ein eindeutiger persistenter Identifikator für jedes Objekt, der in der Regel automatisch vergeben wird. Wenn die Serialisierung bereits von der Programmiersprache, wie z. B. in Java, unterstützt wird, ist die Implementierung in wenigen Schritten abgeschlossen. Nachteilig wirkt sich die Bindung an das Klassenschema aus – einerseits durch die verhinderte Weiterentwicklung des internen Datenmodells und andererseits durch den praktisch unmöglichen Datenaustausch mit anderen Programmen.

- **Externalisierung:** Die Externalisierung ist eine Variante der Serialisierung, bei der der Programmierer festlegt, in welchem Format das Objekt gespeichert werden soll. Innerhalb der Klasse ist eine Methode für das Schreiben und eine für das Einlesen hinzuzufügen.

Persistenter Identifikator: Für die Überführung des persistenten Datenmodells in den Arbeitsspeicher müssen die Objekte auf dem Datenträger identifiziert werden können. Dies geschieht mit einem persistenten Identifikator, der im Identifikatorenraum nur einmal vorkommen darf. Zwei Arten der Erzeugung existieren:

- **Global eindeutiger Identifikator:** (ISO/IEC 11578, 1996) und (ISO/IEC 9834-8, 2005) definieren für die Softwareentwicklung, insbesondere für verteilte Systeme, den Universally Unique Identifier (UUID) zur Erzeugung eindeutiger Identifikatoren ohne zentrale Vergabe. Ein UUID besteht aus 128 Bit und wird als 16-Byte-Zahl in Hexadezimalform dargestellt. Damit stehen $2^{128} = 3,4 \cdot 10^{38}$ mögliche Identifikatoren zur Verfügung. Im Laufe der Zeit wurden fünf verschiedene Versionen entwickelt. Version 1 bezieht die MAC[12]-Adresse, eine quasi-eindeutige Hardwarekennung des Netzwerkadapters, und die Zeit zur Berechnung ein. Sie wird aber nicht mehr benutzt, da auf die Identität des Rechners geschlossen werden kann. Version 4 verlässt sich ausschließlich auf generierte Zufallszahlen, während Version 3 und 5 einen MD5[13]-/SHA-1[14]-Hash aus einer URL[15], einem vollständigen Domänennamen, einem Objektidentifikator und einem Namen aus einem Verzeichnisdienst oder aus anderen Namensräumen berechnen. Der Globally Unique Identifier (GUID) ist eine Implementierung der UUID, die hauptsächlich von der Firma Microsoft in ihren Produkten verwendet wird.

- **Zentral vergebener Identifikator:** Eine zentrale Stelle verwaltet alle Identifikatoren und vergibt eindeutige Identifikatoren an die anfragenden Klienten. Im einfachsten Fall wird ein Zähler bei jeder Anfrage um eins erhöht. In einer verteilten Umgebung muss sichergestellt sein, dass die Vergabestelle bei Bedarf immer erreichbar ist. Als praktisches Beispiel für eine zentrale Registratur soll an dieser Stelle die Vergabe der MAC-Adressen durch das Institute of Electrical and Electronics Engineers (IEEE) erwähnt werden. Die MAC-Adresse ist 48 Bit lang und wird häufig in Hexadezimalform notiert. Das IEEE vergibt die ersten 24 Bit kostenpflichtig als Herstellerkennung[16] und veröffentlicht diese in einer Datenbank. Die restlichen 24 Bit können die Hersteller frei vergeben.

Java (Ullenboom, 2008; Krüger, 2007)**:** Java ist eine von der Firma Sun Microsystems[17] entwickelte und erstmals 1995 veröffentlichte moderne objektorientierte Sprache, die vor allem Schwerpunkte auf Plattformunabhängigkeit, Sicherheit und Unterstützung von Computernetzen legt. Die gewünschten Ziele wurden mit der Gestaltung von Java als hybride Sprache erreicht. Der Quelltext wird mit einem Compiler[18] in Bytecode übersetzt, der dann in einer *Virtuellen Maschine (VM)* ausgeführt wird (s. Abbildung 2.5). Für die am häufigsten verwendeten Plattformen stellt Sun eine Java VM (JVM) zur Verfügung.

[12] MAC = Media Access Control ⟨engl., „Medienzugriffskontrolle“⟩
[13] MD5 = Message-Digest Algorithm 5
[14] SHA = Secure Hash Algorithm
[15] URL = Uniform Resource Locator ⟨engl., „einheitlicher Ressourcen-Positionsanzeiger“⟩
[16] OUI = Organizationally Unique Identifier
[17] Der Name Sun leitet sich von *Stanford University Network* ab, da drei der vier Firmengründer an der Stanford University studiert haben.
[18] compiler ⟨engl., „Übersetzer“⟩

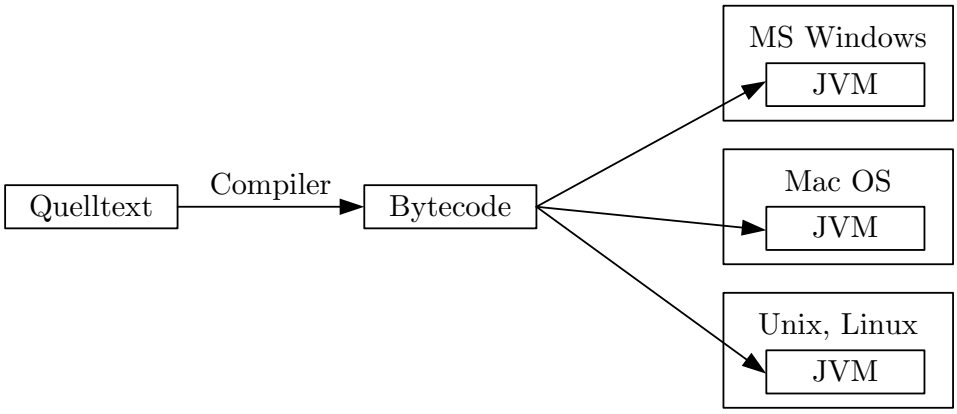


Abbildung 2.5: Java als hybride Sprache

Im Jahr 2007 wurden große Teile von Java unter der Open Source Lizenz *GNU General Public License* (GPL) veröffentlicht.

Generics: Die generische Programmierung wird in Java mit Generics, gleichbedeutend mit dem Begriff *Typparametern*, umgesetzt. Bei der Definition von Methoden, Klassen oder Schnittstellen können einmalig ein oder mehrere Typparameter festgelegt werden, die in der Regel mit einem Großbuchstaben bezeichnet werden. Der oder die Typparameter werden zur Deklaration nach dem Einheitenbezeichner in spitze Klammern gesetzt. Die Listings 2.1 und 2.2 zeigen die Definition einer generischen Schnittstelle mit dem Typparameter `E` und einer abgeleiteten Klasse – jeweils mit den zwei Methoden *setAttribut*() und *getAttribut*().

```
1 public interface GenerischeSchnittstelle<E> {
2     public void setAttribut(E param);
3     public E getAttribut();
4 }
```

Listing 2.1: Generische Schnittstelle

```
1 public class GenerischeKlasse<E> implements
2         GenerischeSchnittstelle<E> {
3     E m_attribut;
4
5     public void setAttribut(E param){m_attribut = param;}
6     public E getAttribut(){return m_attribut;}
7 }
```

Listing 2.2: Generische Klasse

Listing 2.3 zeigt die Anwendung der generischen Klasse. In Zeile 3 wird ein Objekt *myClass* als Instanz der Klasse *GenerischeKlasse* mit dem Typparameter `Integer` erzeugt und in der nächsten Zeile wird der Methode *setAttribut()* ein Integer-Objekt mit dem Wert 42

übergeben. An dieser Stelle prüft der Compiler, ob der übergebene Parameter eine Instanz der Klasse *Integer* ist. Der Ausgabebefehl schreibt in Zeile 6 den Wert 42 in die Konsole.

```
1 public class AnwendungGenerics {
2     public static void main(String[] args){
3         GenerischeSchnittstelle<Integer> myClass =
4             new GenerischeKlasse<Integer>();
5         myClass.setAttribut(new Integer(42));
6         System.out.println(myClass.getAttribut());
7     }
8 }
```

Listing 2.3: Instanziierung der generischen Klasse

2.2 Kooperation im Bauwesen

2.2.1 Einführung

Grundlagen: Im Anhang B auf Seite 225 sind die wesentlichen Grundlagen zu verteilten Systemen als Basis für verteilte Anwendungen zusammengefasst.

Verteilte Anwendung: (Pahl u. Beucke, 2000) definieren eine Anwendung als verteilt, wenn *„der Zugriff auf mindestens ein Objekt der Anwendung den offenen oder versteckten Einsatz eines Kommunikationsdienstes erfordert.“* Im Hinblick auf die verteilten Systeme lässt sich eine verteilte Anwendung auch als eine Anwendung beschreiben, deren Softwarekomponenten an verschiedenen Orten eines verteilten Systems ausgeführt werden und Informationen untereinander austauschen.

Zusammenarbeit: (Bair, 1989) unterscheidet vier Arten von Zusammenarbeit, je nach Grad der Interaktion in der Gruppe (Abbildung 2.6). Informieren bedeutet den Austausch von Informationen, ohne dass sich die Beteiligten zwangsläufig kennen müssen. Koordinieren[19] meint das Teilen von Ressourcen, jedoch können die Beteiligten auch an zwei unterschiedlichen Projekten arbeiten. Bei der Kollaboration[20] arbeiten die Beteiligten zwar auf ein gemeinsames Ziel hin, jeder löst aber eine eigene Teilaufgabe. Die Interaktion und das Zusammenarbeiten in der Gruppe ist bei der Kooperation am größen, da eigene Ziele zurückgestellt werden und das Erreichen des Gruppenziels in den Vordergrund rückt. Die Gruppenmitglieder müssen aber nicht gleichberechtigt sein.

[19] Koordination ⟨lat., „Zuordnung“⟩
[20] co- ⟨lat., „zusammen“⟩, laborare ⟨lat., „arbeiten“⟩

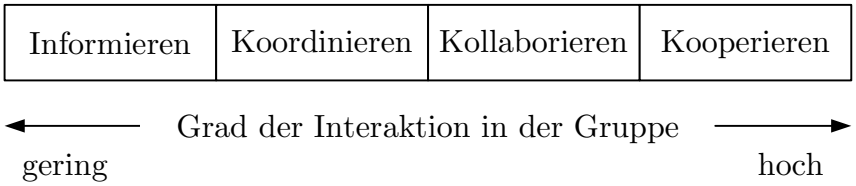


Abbildung 2.6: Arten der Zusammenarbeit nach (Bair, 1989)

(Bretschneider, 1998) bezeichnet Kooperation als *„die Zusammenarbeit mehrerer Personen, Gruppen oder Institutionen in einem Team [...] an einem gemeinsamen Material und auf ein gemeinsames Ziel hin.“* Bretschneider unterscheidet Kooperation hinsichtlich Zeit und dem bearbeiteten Planungsmaterial. Bearbeiten Planer das Material zeitlich nacheinander bzw. asynchron, dann nennt er es *sequentielles Arbeiten*. Arbeiten sie zwar gleichzeitig (synchron), aber an verschiedenen Teilen des Materials, so arbeiten sie *parallel*. Das Kooperationsbewusstsein entsteht durch den Austausch ihrer Ergebnisse. Beim *wechselseitigen Arbeiten* bearbeiten die Planer zeitgleich dasselbe Material, *„um unmittelbar ein gemeinsames Ergebnis zu erzielen“*. Dies deutet darauf hin, das alle Planer gleichzeitig an demselben Modell Änderungen vornehmen und sofort die Änderungen der anderen Planer sehen können. Bretschneider weist darauf hin, dass diese Kooperationsform hauptsächlich dem Beseitigen von Planungskonflikten dient.

	Identische Teile des Materials	Verschiedene Teile des Materials
Verschiedene Zeiten (asynchron)	Sequentielles Arbeiten	
Zur gleichen Zeit (synchron)	Wechselseitiges Arbeiten	Paralleles Arbeiten

Tabelle 2.1: Einordnung von Kooperation hinsichtlich Zeit und Material

Lange Transaktionen: (Firmenich, 2002) unterteilt den Bauplanungsprozess in drei sich wiederholende Phasen. Ausgehend von einem konsistenten Stand des gemeinsamen Arbeitsmaterials werden in Phase 1 durch die Planer Teilmengen gebildet, die nicht disjunkt[21] sein müssen, und auf den eigenen Rechner geladen. In Phase 2 bearbeiten sie synchron oder asynchron und unabhängig voneinander die Teilmengen mit ihren Fachanwendungen. Während dieser Phase ist davon auszugehen, dass die Client-Rechner nicht ständig mit einem Netzwerk verbunden sind. In Phase 3 speichern die Planer ihre Ergebnisse zurück in das gemeinsame Arbeitsmaterial. Im Fall von nicht-disjunkten Teilmengen sind diese automatisch oder manuell zusammenzuführen, um wieder einen konsistenten Zustand des gemeinsamen Arbeitsmaterials herzustellen. Entsprechend Tabelle 2.1 lässt sich diese Kooperationsform als *parallel* ohne Berücksichtigung der Zeit einstufen.

Kommunikation: Kommunikation[22] ist ein wichtiger Bestandteil von und Voraussetzung für erfolgreiche Kooperation. Kommunikation bedeutet immer den Austausch von

[21] disiunctum ⟨lat., „getrennt“⟩
[22] communicare ⟨lat., „teilen, mitteilen, teilnehmen lassen, gemeinsam machen“⟩

Informationen, die sehr vielfältiger Natur sein können. Somit fallen unter Kommunikation der Austausch von verbalen Informationen per Telefon (synchron), von schriftlichen Informationen per E-Mail (asynchron) oder von binären Informationen beim Dokumentenaustausch. Die Beschaffung von Information kann auch durch neuartige Technologien, wie z. B. Software-Agenten, erfolgen (Rüppel, 2007). Im Rahmen von Bauprojekten ist es sinnvoll, Dokumente zentral an einer Stelle zu verwalten und zur Verfügung zu stellen. Dokumenten-Management-Systeme (DMS) leisten dabei gute Dienste.

Integratives Kooperationsmodell: Als Ergebnis des DFG[23]-Schwerpunktprogramms 1103 „Vernetzt-kooperative Planungsprozesse im Konstruktiven Ingenieurbau“ stellt (Rüppel, 2007) ein *integratives Kooperationsmodell* auf, das sich aus vier Schichten zusammensetzt. Die Kommunikationsschicht beinhaltet den synchronen und asynchronen Informationsaustausch zwischen den Planungsbeteiligten. Die Organisationsschicht bildet die Fachplaner (Akteure) und ihre Hoheiten (Rollen) ab. Sie besitzen sowohl Fähigkeiten als auch Rechte. Die Koordinationsschicht versucht mit Hilfe von Prozessmodellen, sequentielle und parallele Arbeitsvorgänge in der Bauplanung zu koordinieren und dadurch die Effizienz und Qualität der Kooperation zu verbessern. Die Ressourcenschicht enthält Modelle, Verarbeitungsmethoden, die auf den Modellen operieren, und Wissen, das formalisiert vorliegt. Zu den Modellen zählen verteilte Produkt- und Simulationsmodelle.

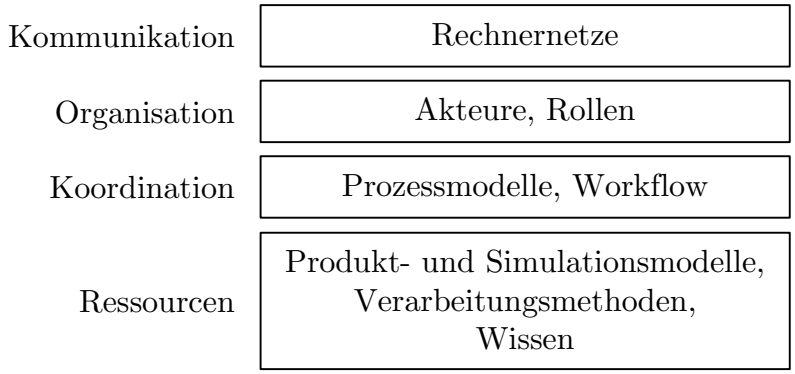


Abbildung 2.7: Integratives Kooperationsmodell nach (Rüppel, 2007)

Produktmodell: Ein Produktmodell im Bauwesen beschreibt einen Teil eines Bauwerks, wobei jede Domäne bzw. jedes Gewerk in der Regel eigene Modelle erstellt. Erst die Summe aller Produktmodelle ergibt ein mehr oder weniger wirklichkeitsgetreues Abbild des zu planenden Bauwerks. Theoretisch ließen sich alle Teilmodelle in einem vereinigen, (Firmenich u. Rank, 2007) bezweifeln aber die praktische Umsetzbarkeit, zumal Informationen zum ganzen Lebenszyklus – von der Planung bis zum Rückbau – anfallen. Zwischen den einzelnen Teilmodellen bestehen Abhängigkeiten, die nicht immer explizit formuliert sind. Werden die Teilmodelle während des Bauplanungsprozesses nicht sorgfältig aufeinander abgestimmt, treten Konflikte spätestens bei der Errichtung des Bauwerks auf. Ziel von verteilten Produktmodellen ist die Sicherstellung der Konsistenz durch geeignete Methoden, um spätere kostenintensive Korrekturen zu vermeiden.

[23] DFG = Deutsche Forschungsgemeinschaft

Prozessmodell: Mit geeigneten Prozessmodellen wird bei mittleren bis großen Projekten versucht, den Bauplanungsprozess, der durch eine hohe Komplexität, eine Vielzahl beteiligter Fachplanern und häufige Umplanungen gekennzeichnet ist, optimal zu gestalten. (Berkhahn u. a., 2007) schlagen hierfür ein relationales Prozessmodell vor, das aus drei Teilmodellen besteht (s. Abbildung 2.8).

- **Organisationsstruktur:** Die Organisationsstruktur entsteht bzw. wird festgelegt, weil die Planungsakteure unterschiedliche Zuständigkeiten und Befugnisse haben.
- **Prozessstruktur:** Die Prozessstruktur verwaltet die Planungsaktivitäten. Jede Aktivität muss unabhängig von einer anderen ausführbar sein und benötigt Zeit sowie Ressourcen (Akteure), was durch Termin- und Ressourcenpläne koordiniert wird.
- **Gebäudestruktur:** Für die Gebäudestruktur wird das Gebäude zunächst in die beiden Komponenten Raum und Bauteil aufgeteilt. Danach werden noch die Verbindungen zwischen ihnen modelliert, da hier oft Planungskonflikte auftreten. Jeder Komponente ist ein Bearbeitungszustand zugeordnet. Ist eine Komponente fertig geplant, enthält sie Verweise auf die entsprechenden Planungsobjekte im Produktmodell.

Zwischen den drei Teilmodellen bestehen Abhängigkeiten, die durch äußere Verknüpfungen abgebildet werden. Diese stellen die Konsistenz des gesamten Prozessmodells sicher.

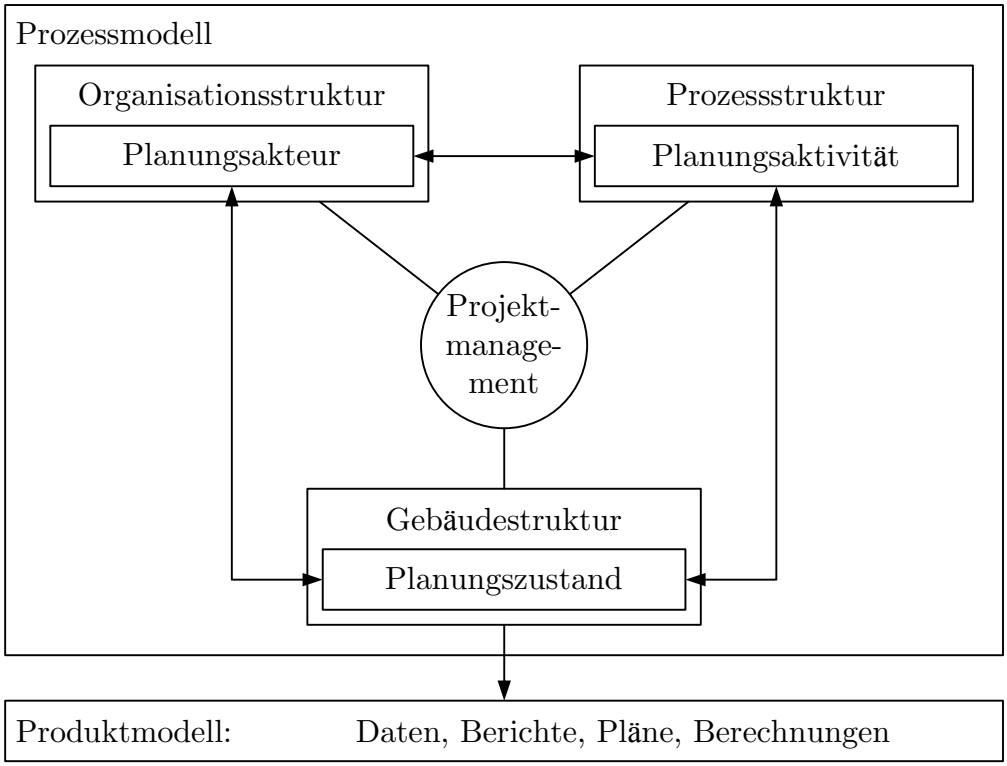


Abbildung 2.8: Relationales Prozessmodell nach (Berkhahn u. a., 2007)

2.2.2 Dokumentbasierter Datenaustausch

Überblick: Digitale Dokumente sind in der Praxis der meistgenutzte Informationscontainer zur Beschreibung von Produktmodellen in Form von Berichten, Berechnungen, Plänen oder anderen Inhalten. Dokumente haben sich vor der Einführung von Computern in den Firmen bewährt und fanden in den Dateien ihr digitales Gegenstück. Für viele Anwender bedeutete der Umstieg von der analogen in die digitale Welt zwar das Erlernen neuer Werkzeuge, aber die gewohnten Arbeitsweisen blieben bestehen. So wurden die Pläne zwar mit CAD-Programmen gezeichnet, am Ende aber doch ausgedruckt und an die beteiligten Projektpartner per Post verschickt. Betrachtet man die aktuelle Situation in Deutschland, so werden die Dokumente mittlerweile digital und meist per E-Mail ausgetauscht.

Formate: Es haben sich im Bauplanungsprozess Standardformate – oft aus der Monopolstellung einzelner Anwendungen heraus – etabliert.

- **Office-Dokumente:**
  - *.doc* als Standardformat der Textverarbeitung Microsoft Word
  - *.xls* als Standardformat der Tabellenkalkulation Microsoft Excel
  - *Office Open XML* wird von Microsoft als offenes Nachfolgeformat seit Office 2007 eingesetzt und wurde von der ISO als Norm (ISO/IEC 29500, 2008) aufgenommen. Die Aufnahme dieses Formats als ISO-Norm wird von einigen Seiten kritisiert, da die Spezifikation mit 6000 Seiten sehr umfangreich ist und mit *OpenDocument* schon vorher ein ISO-Standard existierte.
  - *OpenDocument (ODF)* ist ein offenes, XML-basiertes Format für Büroanwendungen, das von der Firma Sun entwickelt und von der ISO als Norm aufgenommen wurde (ISO/IEC 26300, 2006). Es ist das Standardformat des offenen und lizenzfreien Office-Pakets OpenOffice. Der IT-Rat des Bundesministeriums des Innern empfiehlt OpenDocument für Textdokumente im deutschen E-Government (SAGA 4.0, 2008).
- **PDF:** Das Portable Document Format (PDF) ist ein plattform-, anwendungs- und hardwareunabhängiges Format für elektronische Dokumente, das von der Firma Adobe 1993 veröffentlicht wurde. PDF kann Texte, Vektor-, Rastergrafiken sowie interaktive Formularelemente enthalten und bindet alle verwendeten Schriftarten ein. Die Seitengeometrie ist explizit beschrieben, so dass die Dokumente auf jedem Rechner gleich aussehen. Die ISO hat die Version PDF 1.7 als offenes Format in der Norm (ISO 32000-1, 2008) spezifiziert. Zuvor erschien schon die Norm (ISO 19005-1, 2008), die PDF 1.4 für die Langzeitarchivierung verwendet. Dort sind einige für die Archivierung wichtige Forderungen aufgeführt.
  - Metadaten müssen über den offenen Standard *Extensible Metadata Platform* (XMP) angegeben sein.
  - Alle Bilder und Schriftarten müssen eingebunden sein. Alternative Bilder sind nicht erlaubt.

  - Der vormals patentierte Lempel-Ziv-Welch-Algorithmus (LZW) darf nicht zur Komprimierung eingesetzt werden.
  - Farben und die Kodierungen in den Schriftarten müssen eindeutig sein.
  - Verschlüsselungen und das Sperren von Funktionen sind verboten.
  - JavaScript darf nicht verwendet werden.
  - Audio- und Videoelemente dürfen nicht eingebettet sein.

- **CAD-Dokumente:** CAD-Dokumente enthalten zwei- oder dreidimensionale Geometriedaten. Aufgrund der Vielzahl am Markt erhältlicher CAD-Programme existieren viele CAD-Formate, die den Austausch von Plänen erschweren. Daraufhin haben sich einige wenige Formate durchgesetzt.
  - *DWG*[24] ist das native und proprietäre Dateiformat des Programms AutoCAD der Firma Autodesk. Zusammen mit DXF stellt das binäre Format den Defacto-Standard für Austausch von CAD-Plänen im deutschen Bauwesen dar.
  - *DXF*[25] ist eine offene, textbasierte Spezifikation, die von Autodesk geschaffen wurde und ständig weiterentwickelt wird.
  - *STEP*[26] ist ein von der ISO definierter Standard zur Beschreibung von Produktdaten (DIN ISO 10303). Die Norm untergliedert sich in viele Teile, von denen Teil 11 zum Beispiel die Modellbeschreibungssprache EXPRESS näher erläutert (s. auch Abschnitt 2.2.6 auf Seite 28).
  - *STEP-CDS*[27] ist ein CAD-neutrales Format zur Beschreibung von 2D-Geometrie, das auf der Norm (ISO 10303-214, 2005) aufbaut. Neben der Geometrie sind noch einige andere Informationen enthalten, wie Annotationen (Bemaßung, Beschriftung, ...), Modellstrukturen (Gruppen, Layer), Zeichnungslayout, Sachdaten (Material, Kosten, ...) und externe Referenzen. STEP-CDS wird vom Verband der Automobilindustrie (VDA) für die Planung bei der Errichtung von Betriebsstätten empfohlen (VDA, 2006).

- **Bauinformationen:** GAEB DA XML dient zum Austausch von Bauinformationen von der Ausschreibung bis zur Durchführung eines Bauvorhabens. Der Standard wird vom Gemeinsamen Ausschuss Elektronik im Bauwesen (GAEB) definiert und unterstützt mehrere Datenaustauschphasen.
  - D81 Leistungsbeschreibung
  - D82 Kostenanschlag
  - D83 Angebotsaufforderung
  - D84 Angebotsabgabe
  - D85 Nebenangebot
  - D86 Auftragsvergabe
  - D89 Rechnung
  - D93 Preisanfrage
  - D94 Preisangebot
  - D96 Bestellung
  - D97 Auftragsbestätigung

[24] Drawing ⟨engl., „Zeichnung“⟩
[25] Drawing Interchange Format ⟨engl., „Zeichnungsaustauschformat“⟩
[26] STEP = Standard for the Exchange of Product model data
[27] CDS = Construction Drawing Subset

Organisation: Obwohl der Datenaustausch elektronischer Dokumente per E-Mail einfach durchzuführen ist, stellt sich die Frage, wie diese Dateien innerhalb der Firmen verwaltet werden. Stehen nur die Mittel des Betriebssystems zur Verfügung, lässt sich im Dateisystem eine hierarchische Struktur anlegen, um die Dateien später schneller wiederzufinden. Um z. B. aus rechtlichen Gründen auf einen früheren Stand der Planung zurückzugreifen, ist es notwendig, die Historie von Dokumenten zu speichern. Im Dateisystem ist dies nur möglich, wenn der Bearbeiter selbst eine aufsteigende Nummer dem Dateinamen hinzufügt, was aber Disziplin und Aufmerksamkeit erfordert. Zudem ist der Schutz vor Datenverlust auf Einzelplatzrechnern kaum gegeben.

Das Unternehmen könnte sich einen zentralen Dateiserver zulegen und die Berechtigungen der Mitarbeiter für bestimmte Bereiche des Verzeichnisbaums festlegen. Auch eine Sicherung der Daten wäre einfach umzusetzen. Allerdings ist das Wechseln zu einem früheren Projektstand mit einigem Aufwand verbunden und entspricht keiner transparenten Lösung.

Ein paralleles Arbeiten wird von den meisten Anwendungen nicht unterstützt, da sie Dokumente weder vergleichen noch zusammenführen können.

2.2.3 Dokumenten-Management-Systeme

Dokumenten-Management: Ursprünglich lagen Dokumente ausschließlich in Papierform vor und mit Einführung des Dokumentenmanagements wurde zunächst nur die Verwaltung von gescannten Schriftstücken verbunden. Dadurch, dass später ohnehin viele Dokumente auf dem Rechner erstellt und bearbeitet wurden, musste für diese nicht der Umweg über Drucker und Scanner gegangen werden. Im Laufe der Entwicklung kamen zusätzliche Aufgaben dazu. Die Dokumente sollten nicht nur erfasst, sondern auch mit Metadaten versehen, versioniert, schnell wiedergefunden oder vom System dem richtigen Bearbeiter zum richtigen Zeitpunkt wieder vorgelegt werden.

Dokumenten-Management-System: (Kampffmeyer u. Rogalla, 1997) nehmen eine Einordnung von Dokumenten-Management-Systemen (DMS) im weiteren und engeren Sinn vor. Zu ersterem zählen verschiedene Systeme, die zusammenarbeiten.

- Dynamisches Dokumenten-Management,
- Bürokommunikation,
- Document Imaging (Erfassung von gescannten Dokumenten),
- Workflow,
- Groupware,
- Elektronische Archivierung.

Zu den zweiten zählen klassische Dokumenten-Management-Systeme zur Verwaltung von Dokumenten bzw. Dateien. Sie nutzen zur Datenspeicherung in der Regel einen Datenbankserver und sind durch folgende Merkmale gekennzeichnet.

- Zusammenfassung von Einzeldokumenten zu Containern,
- Check-in: *„Kontrolliertes Einbringen von Dokumenten mit Vergabe von Attributen in Dokumenten-Management-Systeme.“*,
- Check-out: *„Kontrollierter Export von Dokumenten aus Dokumenten-Management-Systemen in andere Umgebungen, z. B. zur erneuten Bearbeitung.“*,
- Versionsmanagement:*„Verwaltung und konsistente Speicherung der unterschiedlichen Versionen von Dokumenten [. . .].“* Die Dokumente werden linear versioniert, wobei nur ein Bearbeiter die aktuelle Version auschecken, bearbeiten und wieder einchecken kann. Während dieses Zeitraums ist das Dokument für die anderen Nutzer schreibgeschützt. Dieses *pessimistische Zugriffsmodell* erlaubt nur sequentielles Arbeiten.
- Selbstbeschreibende Dokumente: Sie bestehen aus einem Header und einer Inhaltskomponente mit dem eigentlichen Dokument. Der Header enthält die Selbstbeschreibung des Dokuments (Metadaten), wozu u. a. zählen: Eindeutiger Identifikator, Autor, Datum, Uhrzeit, Dokumentklasse, Zugriffsschutzklasse und Prüfsummen.

Ein Archivsystem ist ein integraler Bestandteil eines Dokumenten-Management-Systems, der für die langfristige und unveränderbare Speicherung von Dokumenten zuständig ist. Der Verband Organisations- und Informationssysteme e. V. (VOI) hat zehn Merksätze zur revisionssicheren elektronischen Archivierung in (Kampffmeyer u. Rogalla, 1997) veröffentlicht.

1. Jedes Dokument muss unveränderbar archiviert werden.
2. Es darf kein Dokument auf dem Weg ins Archiv oder im Archiv selbst verloren gehen.
3. Jedes Dokument muss mit geeigneten Retrievaltechniken wiederauffindbar sein.
4. Es muss genau das Dokument wiedergefunden werden, das gesucht worden ist.
5. Kein Dokument darf während seiner vorgesehenen Lebenszeit zerstört werden können.
6. Jedes Dokument muss in genau der gleichen Form, wie es erfasst wurde, wieder angezeigt und gedruckt werden können.
7. Jedes Dokument muss zeitnah wiedergefunden werden können.
8. Alle Aktionen im Archiv, die Veränderungen in der Organisation und Struktur bewirken, sind derart zu protokollieren, dass die Wiederherstellung des ursprünglichen Zustandes möglich ist.
9. Elektronische Archive sind so auszulegen, dass eine Migration auf neue Plattformen, Medien, Softwareversionen und Komponenten ohne Informationsverlust möglich ist.

10. Das System muss dem Anwender die Möglichkeit bieten, die gesetzlichen Bestimmungen (BDSG, HGB/AO etc.) sowie die betrieblichen Bestimmungen des Anwenders hinsichtlich Datensicherheit und Datenschutz über die Lebensdauer des Archivs sicherzustellen.

2.2.4 Zentrale Zeichnungsdatenbank (Autodesk Revit)

Autodesk Revit: Revit, das 2002 von Autodesk aufgekauft wurde, ist ein parametrisches CAD-System, mit dem Bauwerksinformationsmodelle (BIM)[28] erstellt werden können. Ein BIM beschreibt den kompletten Lebenszyklus eines Bauwerks und fasst dazu räumliche Geometriedaten und weitere Informationen über das Bauwerk zusammen (s. Abschnitt 2.2.6).

Kooperationsansatz: Wie in (Revit, 2005) beschrieben, unterstützt Revit paralleles Zusammenarbeiten mehrerer Projektbeteiligter auf Basis eines pessimistischen Zugriffsmodells. Zu Beginn der Kooperation muss eine zentrale Datei auf einem Rechner angelegt werden. Jeder Nutzer, der an der Bearbeitung teilnehmen will, checkt sich das Modell in eine lokale Datei auf seinem Arbeitsplatzrechner aus. Um das Sperren des gesamten Planungsmaterials durch einen Nutzer zu vermeiden, lassen sich vorher Bereiche (Worksets) definieren, die bestimmte Bauteile enthalten. Ein Workset kann nur durch genau einen Planer bearbeitet werden, für die anderen ist dieser Bereich schreibgeschützt. Jederzeit sind die Änderungen eines Planers in seiner lokalen Datei speicherbar, ohne das zentrale Modell zu beeinflussen. Ein Check-in der lokalen Änderungen in die zentrale Datei übernimmt der Befehl *Speichern in die Zentraldatei*, der projektweit nur von einem Nutzer gleichzeitig ausgeführt werden darf. Um die veröffentlichten Arbeitsergebnisse der anderen Planer zu erhalten, ist ein Update durchzuführen.

Fernzugriff: Für den Zugriff auf die zentrale Datei ist immer ein direkter Zugriff auf das Intranet, z. B. durch ein gesichertes VPN[29], erforderlich. Im Whitepaper (Revit, 2005) wird explizit darauf hingewiesen, dass ein Arbeiten über VPN – insbesondere bei großen Projekten – Performanceprobleme verursachen kann. Alternativ wird vorgeschlagen, über eine Terminalsitzung (Remote-Desktop) auf einen Rechner im Intranet zuzugreifen, auf dem sich die lokale Datei befindet.

2.2.5 Dokumenten-Management mit optimistischem Zugriffsmodell (Bentley ProjectWise)

Bentley ProjectWise: Bentley ist ein Anbieter für Konstruktionssoftware, der mit MicroStation eine CAD-Anwendung im Produktportfolio hat. Darüber hinaus bietet Bentley mit ProjectWise eine umfassende Kollaborationslösung an, die einem Dokumenten-Management-System mit weitergehenden Funktionen entspricht, so dass dieses Framework

[28] BIM = Building Information Modeling
[29] VPN = Virtual Private Network

in das Enterprise-Content-Management (ECM) eingeordnet werden kann. Je nach Projektgröße und gewünschter Funktionalität lässt sich das Framework aus verschiedenen Servern zusammenstellen.

Kooperationsansatz: Das besondere an ProjectWise ist, dass für das firmeneigene CAD-Format DGN[30] ab Version 8 ein optimistisches Zugriffsmodell implementiert ist. Dieses *Konzept der verteilten DGN* erlaubt das gleichzeitige Auschecken und parallele Bearbeiten der letzten (aktiven) Version eines Dokuments durch mehrere Benutzer. In den lokalen Dateien wird eine Zeichnungshistorie aktiviert, die das Hinzufügen, Ändern und Löschen von Zeichnungskomponenten aufzeichnet. Die lokalen Dateien werden später nacheinander durch die Planer wieder in ProjectWise eingecheckt. Falls schon eine neuere Version auf dem Server vorliegt, muss der Planer eine Aktualisierung seiner lokalen Kopie vornehmen, wobei Konflikte, die durch die gleichzeitige Bearbeitung von Zeichnungskomponenten durch andere Planer entstanden sind, in der CAD-Anwendung angezeigt werden. Konflikte müssen geprüft und vor dem Einchecken der Zeichnung als gelöst markiert werden.

Im aktuellen Handbuch von ProjectWise (Bentley, 2008) wird vom produktiven Einsatz verteilter DGN abgeraten. Da dieser Vermerk schon 2006 im Handbuch stand, zeigt sich die Komplexität des optimistischen Zugriffsmodells für objektorientierte Modelle. Durch das Abschalten der Option *Gemeinsam bearbeitbar* verhalten sich die Dokumente wie in einem normalen Dokumenten-Management-System mit pessimistischem Zugriffsmodell.

Abhängigkeiten: Abhängigkeiten zwischen Zeichnungskomponenten verschiedener Dokumente können nicht explizit definiert werden. Es können lediglich mehrere Dokumente zu einem *Logischen Set* zusammengefasst werden. Ein als Masterdokument bezeichnetes Dokument referenziert dabei andere Dokumente, was vom Planer festgelegt wird. Zeichnungskomponenten sind wahrscheinlich nicht projektweit eindeutig adressierbar, da nur zeichnungsintern eindeutige IDs durch eine fortlaufende Nummer vergeben werden.

2.2.6 Bauwerksinformationsmodelle

Bauwerksinformationsmodell: Ein Bauwerksinformationsmodell bzw. Produktdatenmodell fasst alle Informationen zum gesamten Lebenszyklus eines Bauwerks in einem Modell zusammen. Konstruktionsanwendungen verwenden intern ein eigenes BIM, das in der Regel inkompatibel zu denen in Anwendungen anderer Hersteller ist. Für den Datenaustausch ist demnach ein standardisiertes Bauwerksinformationsmodell erforderlich.

IFC: Das bekannteste Bauwerksinformationsmodell bilden die von der 1995 gegründeten Industrieallianz für Interoperabilität e.V. (IAI) entworfenen *Industry Foundation Classes* (IFC). Das semantische Objektmodell, welches auf Klassendefinitionen in der Modellbeschreibungssprache EXPRESS basiert, soll in Zukunft alle Fachdomänen abdecken. Die aktuelle Version IFC 2x3 TC1[31] umfasst hauptsächlich die Bereiche Architekturplanung und Technische Gebäudeausrüstung (TGA)[32]. Enthaltene Objekte sind

[30] DGN = Microstation CAD drawing
[31] http://www.iai-tech.org/products/ifc_specification/ifc-releases/ifc2x3-tc1-release
[32] Die TGA wird alternativ durch die Begriffe Heizung-Klima-Lüftung-Sanitär (HKLS) umschrieben.

z. B. semantische Elemente (Bauteile), Prozesse und geometrische Formen. Das Release 2x wurde als Plattformspezifikation als Publicly Available Specification (PAS)[33] aufgenommen (ISO/PAS 16739, 2005). Der Austausch von IFC-Daten erfolgt entweder über STEP-Dateien (ISO 10303-21, 2002) oder über einen IFC-Modellserver (Kiviniemi u. a., 2005).

Obwohl die IFC für den verbesserten Datenaustausch im Bauplanungsprozesse sorgen soll, treten einige Probleme zu Tage, die den Einsatz der IFC behindern.

- Laut (Weise, 2006) entsteht *„durch die modulare Datenstruktur ein höherer Implementierungsaufwand [...], da die Modelldefinitionen nicht auf fachspezifische Anforderungen optimiert werden kann.“*.
- Die verschiedenen Anwendungen können intern nur mit einem bestimmten Teil des Modells umgehen, so dass beim Laden und Speichern Datenverluste auftreten. Dieser Fakt hat bei Software verschiedener Fachdomänen eine umso stärkere Bedeutung.
- Das Gesamtmodell eines Bauwerks führt zu einer großen Datenmenge, die beim Austausch über Dateien hinderlich ist. Außerdem wird beim Bearbeiten des Modells nur ein kleiner Teil verändert. Eine Lösung besteht in der Aufteilung in Partialmodelle, wodurch wiederum Probleme bezüglich der referentiellen Integrität auftreten.
- Eine Versionierung und die Umsetzung von Benutzerrechten wird beim Dateiaustausch nicht unterstützt.

(Kiviniemi u. a., 2005) versuchen mit dem Konzept eines IFC-Modellservers, die angesprochenen Probleme zu lösen. Das IFC-Modell soll in Partialmodelle aufgeteilt werden, die über Objektreferenzen aufeinander verweisen. Die Serverarchitektur stützt sich auf Datenbankserver und Webservices sowie Standardtechnologien: XML, SOAP[34], WSDL[35]. Obwohl einige kommerzielle Umsetzungen existieren, fordern die Autoren eine standardisierte Programmierschnittstelle für Modellserver. Der *ModelServer*[36] der Firma Eurostep unterstützt laut Produktinformationsseite ein Versionsmanagement für Modellinstanzen.

Aktuelle Situation: Die Entwicklung und der Einsatz von Bauwerksinformationsmodellen wird vor allem in den skandinavischen Ländern vorangetrieben. So müssen z. B. in Dänemark Bauvorhaben der öffentlichen Hand über 5,3 Mill. Euro zusätzlich mit der IFC modelliert werden. (Kiviniemi u. a., 2008) berichten in ihrer Umfrage, dass in Dänemark, Finnland und Norwegen ca. 10 % und in Schweden ca. 30 % der Firmen Bauwerksinformationsmodelle ohne IFC-Bezug einsetzen. Der Einsatz IFC-kompatibler BIM teilt sich wie folgt auf: Dänemark 11 %, Finnland 19 %, Norwegen 4 % und Schweden 2 %. Über 60 % der Firmen nutzen reine CAD-Programme und ca. 8 % zeichnen noch mit Hand.

[33] PAS = Publicly Available Specification, ⟨engl., „öffentlich verfügbare Spezifikation“⟩. Eine PAS hat nicht den Status einer Normen, soll aber laut DIN *„die Lücke zwischen der konsensbasierten Normung und Industriestandards [...] schließen.“* Diese technischen Spezifikationen durchlaufen eine schnellere Veröffentlichungsphase von 6 bis 8 Wochen und werden allein durch die Verfasser verantwortet.

[34] SOAP = Simple Object Access Protocol

[35] WSDL = Web Services Description Language

[36] http://www.eurostep.com/global/solutions/complementary-tools/eurostep-modelserver-for-ifc.aspx

Die Akzeptanz von BIM ist bei Architekten mit 30 % am höchsten, danach folgen Ingenieure mit 18 % und Subunternehmer mit 12 %. Ihre Pläne von Hand zeichnen 10 % der Architekten, 5 % der Ingenieure und immerhin noch 27 % der Subunternehmer.

Domänenspezifische Teilmodelle: (Willenbacher, 2002) trennt in seiner Arbeit das zentrale Produktdatenmodell in domänenspezifische Teilmodelle auf. Dadurch können die Teilmodelle besser an die Anforderung der spezialisierten Fachanwendungen ausgelegt werden. Um die entstehenden Redundanzen widerspruchsfrei zu halten, müssen Verknüpfungen zwischen den Teilmodellen definiert werden, die wie die Teilmodelle dynamisch modifizierbar sind. Willenbacher weist im Fazit auf die Problematik der komplexen Verknüpfungsdefinition hin:

> „*An dieser Stelle muss jedoch ausdrücklich auf die Problemhaftigkeit der Verknüpfungstypdefinition und der Verknüpfungsspezifikation durch die Fachplaner hingewiesen werden. In Abhängigkeit von der inhaltlichen Ausrichtung und der Strukturierung der Partialmodelle kann sich die Erstellung der Verknüpfungen als ein sehr komplexer Prozess gestalten, welcher ein hohes Maß an partialmodellübergreifendem Wissen und Verständnis erfordert.*“

2.2.7 Modellierung von Abhängigkeiten

Ziel: Durch die Bearbeitung entstehen Änderungen am Modell, die zu Inkonsistenzen an diesem oder anderen Modellen führen können. Konsistenzsicherung ist das Beheben von Inkonsistenzen, die (Laabs, 1998) in zwei Typen einteilt.

- **Manuelle Konsistenzsicherung:** Das System lässt beliebige Aktionen zu, so dass danach immer noch Inkonsistenzen vorliegen. Der Benutzer muss die Inkonsistenz selbst erkennen und beseitigen, ohne vom System unterstützt zu werden.
- **Automatische Konsistenzsicherung:** Das System nimmt die Beseitigung der Inkonsistenzen vor und lässt nur Aktionen zu, die zu diesem Ziel beitragen.

(Laabs, 1998): Laabs schlägt für Ingenieuranwendungen einen Halbautomatismus mit einer Kombination beider Methoden vor, bei dem der Nutzer den Zeitpunkt zur Wiederherstellung der Konsistenz festlegt. Parallel zum objektorientierten Modell führt er in seiner Arbeit eine externe Verwaltung zur Aktualisierung und Konsistenzsicherung ein, die auf Methoden der Mengenlehre aufbaut und einen Beobachtermechanismus zur Erkennung von Objektänderungen verwendet. Laabs deutet im Fazit an, dass dieser Ansatz für große Modelle kaum anwendbar ist, da der Überblick über Aktionen und Reaktionen verloren geht.

Verzögerte Aktualisierung: (Hanff, 2003) setzt auf eine verzögerte Aktualisierung der Objekte auf Basis der Abhängigkeiten, die vom Nutzer gestartet wird. Dazu beschreibt er das zu entwerfende objektorientierte System formal mit der Graphentheorie. Der Zugriff auf die Attribute, Ein- und Ausgabeparameter der Methoden sowie Aktualisierungsmethoden für die Attribute wird für jedes Objekt in der sprachunabhängigen Schnittstellenbeschreibungssprache IDL zwingend festgelegt. Das Klassenschema wird in XML formuliert,

aus dem mit einem Generator die Klassen für eine Programmiersprache generiert werden. Für den Aktualisierungsalgorithmus erhalten die Objekte, Attribute und Methoden transiente, ganzzahlige Versionsnummern, die bei einer erfolgten Aktualisierung um eins erhöht werden. Der Ansatz von Hanff ist nur für Neuimplementierung interessant und lässt sich nicht für bestehende Fachanwendungen einsetzen.

Semantische Abhängigkeiten: (Olivier, 2007) stellt ein Konzept zur Kopplung von zwei unabhängigen Ingenieurmodellen auf der Basis von semantischen Abhängigkeiten vor. Am Beispiel der Kopplung von CAD- und FEM[37]-Modellen entwirft er ein Structural Engineering Model (SEM), das durch den Tragwerksplaner mit Hilfe von Strukturkomponenten erzeugt wird. Der Tragwerksplaner entscheidet, welche Strukturkomponenten von ausgewählten geometrischen Komponenten abhängig sind. Nach Festlegung weiterer für die FEM notwendigen Parameter kann das Finite-Elemente-Modell aus dem SEM erzeugt werden. Entscheidend ist, dass die ursprünglichen Modelle nicht erweitert werden, sondern nur das SEM Kenntnisse über die Abhängigkeiten zwischen den Komponenten beider Modelle besitzt.

Olivier unterscheidet bei den Abhängigkeiten zwischen Intra- und Intermodellbindungen. Intramodellbindungen sind Abhängigkeiten zwischen Objekten innerhalb des Datenmodells einer Anwendung, während Intermodellbindungen Abhängigkeiten zwischen Objekten in zwei getrennten Datenmodellen sind. In der Implementierung befindet sich zwischen einer Menge von bindenden Objekten und dem zugehörigen gebundenen Objekt ein Binder-Objekt, das wiederum ein Update-Objekt kennt. Zur Erkennung von Objektänderungen eignet sich die Versionierung am besten, da andere Methoden wie Zeitstempel oder Prüfsummen nicht immer korrekte Ergebnisse liefern. Im Falle von Objektänderungen im ersten Modell stellt ein Aktualisierungsvorgang die Konsistenz zwischen beiden Modellen wieder her. Ausgehend von einem Bindungsgraphen wird die korrekte Aktualisierungsreihenfolge durch eine topologische Sortierung bestimmt. Das Update-Objekt führt die Aktualisierung des gebundenen Objekts abhängig vom Zustand der bindenden Objekte durch.

2.2.8 Änderungsorientierter Ansatz

Konzept: Die meisten Ansätze verwenden ausgewertete Modelle für den Datenaustausch zwischen verschiedenen Anwendungen. Wie schon mehrfach erwähnt, gehen Informationen durch die Transformation der nativen Modelle in Standardformate verloren. (Koch, 2008) stellt ein Konzept vor, bei dem die Befehle durch ein Journaling in Änderungsdateien aufgezeichnet und zwischen den Planungsbeteiligten ausgetauscht werden. Dadurch werden kumulierende Informationsverluste vermieden. Eine Analogie besteht zur geometrischen Volumenmodellierung, wo entweder die Geometrie durch die Außenflächen (B-Rep[38]) oder durch die Ausführung von booleschen Operationen auf Körper (CSG[39]) beschrieben wird.

[37] FEM = Finite-Elemente-Methode. Ein numerisches Verfahren zur näherungsweisen Lösung von Differentialgleichungen.
[38] B-Rep = Boundary Representation
[39] CSG = Constructive Solid Geometry

Bei der CSG-Modellierung entsteht ein CSG-Baum, dessen Blätter einem geometrischen Primitiv und die Knoten einer Operation bzw. dem Zwischenergebnis dieser Operation entsprechen. Das Endergebnis ist im Wurzelknoten enthalten.

Das objektorientierte Modell wird um eine Änderungssemantik zur Sicherstellung der Konsistenz erweitert, wobei eine operative Modellierungssprache die Schnittstelle zwischen Fachplaner und Fachanwendung darstellt. Für verschiedene Fachdomänen lassen sich standardisierte Modellieroperationen definieren, die idealerweise von allen Anwendungen aus diesem Bereich verstanden werden. Eine eindeutige Identifizierbarkeit der Objekte ist für diesen Ansatz zwingend notwendig und wird durch persistente Objektidentifikatoren sichergestellt.

Die Speicherung von ausgewerteten Modellen ist zu bestimmten Meilensteinen weiterhin empfehlenswert, da eventuelle Datenverluste in den Änderungsdateien eine Lücke in der Befehlshistorie bedeuten. Außerdem würde das Abspielen der kompletten Befehlshistorie von großen Modellen viel Zeit benötigen. Eine Kombination des änderungsorientierten Ansatzes mit einer versionierten Umgebung bietet sich an, da ein Meilenstein genau einer Version des ausgewerteten Modells entspricht.

2.3 Software Configuration Management

2.3.1 Konfigurationsmanagement

Das Konfigurationsmanagement (KM) entstand in den 1950er aus dem Wunsch heraus, die ansteigende Produktkomplexität in der Luft- und Raumfahrtindustrie zu beherrschen. Diese Probleme betrafen nach und nach auch andere Branchen bis hin zur Softwareentwicklung. Nachfolgend sind einige der in der Norm (DIN ISO 10007, 2004) definierten Begriffe aufgeführt.

Produktkonfigurationsangaben (DIN ISO 10007, 2004)**:** *„Anforderungen an Entwicklung, Realisierung, Verifizierung, an Funktionstüchtigkeit und Unterstützung des Produkts.“*

Konfiguration (DIN ISO 10007, 2004)**:** *„Miteinander verbundene funktionelle und physische Merkmale eines Produkts, wie sie in den Produktkonfigurationsangaben beschrieben sind.“*

Konfigurationseinheit (DIN ISO 10007, 2004)**:** *„Einheit innerhalb einer Konfiguration, die eine Endgebrauchsfunktion erfüllt.“*

Konfigurationsmanagement (DIN ISO 10007, 2004)**:** *„Koordinierte Tätigkeiten zur Leitung und Lenkung der Konfiguration. Das Konfigurationsmanagement konzentriert sich üblicherweise auf technische und organisatorische Tätigkeiten, die die Lenkung eines Produkts und der dazugehörigen Produktkonfigurationsangaben in allen Phasen des Produktlebenszyklus einleiten und aufrechterhalten.“*

Konfigurationsmanagement-Prozess: Weiterhin geht die Norm auf den Konfigurationsmanagement-Prozess ein, für den Verantwortlichkeiten, Befugnisse und auszuführende Tätigkeiten festzulegen sind.

2.3.2 Software Configuration Management

Definition (Estublier u. a., 2005)**:** Software Configuration Management (SCM) ist ein auf Softwaresysteme angewandtes Konfigurationsmanagement. Das Ziel von SCM ist es, einen systematischen und nachvollziehbaren Softwareentwicklungsprozess sicherzustellen, in dem alle Änderungen korrekt verwaltet werden, so dass ein Softwaresystem zu jeder Zeit in einem wohldefinierten Zustand ist. Im Gegensatz zu anderen Anwendungen des Konfigurationsmanagement ist die Hauptaufgabe die Verwaltung von Verzeichnissen und Dateien.

Aufgabe eines SCM-Systems: (Estublier u. a., 2005) stellen drei Hauptaufgaben für ein SCM-System heraus, die sich bis heute nicht geändert haben.

1. Verwalten von Dateien, die für die Erstellung des Softwareprodukts notwendig sind.
2. Aufzeichnen von Änderungen an diesen Dateien in Hinsicht auf die resultierenden Versionen.
3. Das Bauen eines ausführbaren System aus den Dateien.

Geschichtliche Entwicklung: Die Entwicklung von SCM-Systemen begann Mitte der 1970er als einzelne Programmierer ein Versionierungssystem für die Entwicklung kritischer Software auf Großrechnern (Mainframe) benötigten. Das erste SCM-System war das Source Code Control System (SCCS), das von Rochkind für ein IBM System/370 entwickelt und später auf Unix portiert wurde (Rochkind, 1975). Danach folgte 1980 das Revision Control System (RCS) von (Tichy, 1982), das zur platzsparenden Speicherung Rückwärts-Deltas verwendet. Dazu wird die letzte Version vollständig und zusätzlich die Änderungen zu den Vorgängerversionen in einer Datei gespeichert. Die Berechnung der Änderungen erfolgt durch den Diff-Algorithmus von (Hunt u. McIllroy, 1976). Diese Methode ist aber nur für textbasierte Dateien sinnvoll, bei denen sich von Version zu Version wenig ändert. Beide SCM-Systeme versionieren nur einzelne Dateien und bieten keine Team-Unterstützung. RCS setzt eine Sperre auf die Datei, wenn sie gerade bearbeitet wird.

Die nächste Generation diente zur Entwicklung großer Softwareprojekte auf Unix-Rechnern in einem lokalen Team. Die Versionen aller Dateien werden in einem zentralen *Repository*[40] gespeichert und jeder Programmierer, der an dem Projekt mitarbeiten will, muss sich zunächst den aktuellen Stand als *Arbeitskopie* in den *Workspace*[41] auf seinem lokalen Rechner auschecken. Im Workspace kann der Bearbeiter beliebig Dateien anlegen, löschen oder ändern, ohne dass der Stand auf dem Repository davon beeinflusst wird. Erst durch einen *Commit* (Check-in) veröffentlicht der Programmierer das Ergebnis seiner Arbeit, das

[40] repository ⟨engl., „Depot, Lager“⟩
[41] workspace ⟨engl., „Arbeitsbereich“⟩

dann für alle zugänglich ist und in deren Workspaces übernommen werden kann. Wenn mehrere Programmierer eine Datei bearbeiten, können Konflikte beim Commit entstehen, die entweder präventiv durch Sperren vermieden oder erst in Kauf genommen und dann durch einen manuellen *Merge* behoben werden. Der zweite Ansatz wird auch als optimistisches Zugriffsmodell bezeichnet. Ein bekannter Vertreter ist das anfangs von (Grune, 1986) als Shell-Skripte entwickelte Concurrent Versions System (CVS), das später von (Berliner, 1990) erweitert und als Software veröffentlicht wurde. Es benutzt das Dateiformat von RCS und implementiert ein optimistisches Zugriffsmodell, welches bei (Honda u. Miller, 1989) als Copy-Modify-Merge-Szenario beschrieben ist.

Mit der dritten Stufe von SCM-Systemen kann jede Art von Software durch viele Beteiligte, die beliebige Plattformen nutzen und weltweit verteilt sind, entwickelt werden. Für einen erfolgreichen Einsatz müssen sie den gesamten Entwicklungsprozess unterstützen, was über die reine Versionierung hinausgeht. Zu den erweiterten Aufgaben zählen unter anderem: Benutzerverwaltung und -organisation, Fehlerverfolgung, Kundenwünsche sowie die automatische Generierung neuer Versionsstände (Builds). Versionsverwaltungssysteme[42] werden daher als Teilgebiet des Software Configuration Management angesehen.

(Estublier u. a., 2005) fassen die geschichtliche Entwicklung in folgender Tabelle zusammen. Die Grenzen zwischen der zweiten und dritten Generation verwischen teilweise, da z. B. CVS inzwischen Standardsoftware ist, auf mehreren Plattformen läuft und von räumlich weit entfernten Entwicklern gleichzeitig genutzt werden kann.

	1970	1980	1990	2000
System:	Eigenbau	Ad-hoc-System	Standard	
Für	kritische Software	große Software	jede Software	
Durch	Versionierung	Workspace	Prozessmodellierung	
Mit	einer Person	lokalem Team	jedem überall	
Auf	Mainframe	Unix	jedem Rechner	

Tabelle 2.2: Evolution des Kontexts von SCM-Systemen nach (Estublier u. a., 2005)

Begriffe und Konzepte

Repository: In einem Repository werden alle erstellten Versionen von Dateien und/oder Verzeichnissen und zugehörige Metainformationen gespeichert. Bei einer zentralen Versionsverwaltung, die immer eine Client-Server-Architektur umsetzt, existiert nur ein Repository, das sich auf einem Server befindet. Seit ca. 2005 gibt es verteilte Versionsverwaltungssysteme, bei denen es kein zentrales Repository mehr gibt. Stattdessen besitzt jeder Nutzer auf seinem Rechner ein eigenes Repository, das zur lokalen Versionierung dient. Dadurch ist kein Serverzugriff beim täglichen Arbeiten mehr nötig und es entstehen keine Konflikte. Die Zusammenarbeit wird erreicht, indem sich das eigene Repository mit

[42] VCS = Version Control System

dem Repository eines anderen Nutzers abgleichen lässt. Die bekanntesten Vertreter mit Open-Source-Lizenz sind Bazaar, Git[43], Mercurial[44] und Monotone.

Workspace: Der Workspace enthält die Arbeitskopie vom Repository, in der der Nutzer Änderungen vornehmen kann. In der hierarchische Struktur im Dateisystem bestehend aus Verzeichnissen und Dateien werden zusätzlich Metadaten zu den Versionseinheiten gespeichert. Nach dem Anlegen der Arbeitskopie mit einem *Check-out* erfolgt die Synchronisation durch ein *Update* vom Repository zum Workspace oder durch ein *Commit* in die Gegenrichtung. Ein weiterer häufig verwendeter Begriff für Workspace ist *Sandbox*[45].

Diff: Ein Vergleichen (Diff[46]) ist immer dann erforderlich, wenn ein Update oder ein Commit geänderter Dateien vorgenommen wird. Mit dem Diff-Algorithmus von (Hunt u. McIllroy, 1976), der von (Myers, 1986) noch verbessert wurde, werden die Änderungen bestimmt.

Delta: Die Delta-Kodierung wird in der Versionsverwaltungssystem verwendet, um die Dateiversionen platzsparend zu speichern. Damit ist es eine Möglichkeit zur Datenkompression. Beim Vorwärtsdelta-Verfahren wird die erste Version komplett und für die Folgeversionen jeweils nur die Änderung zur Vorgängerversion (Delta) gespeichert. Da im Entwicklungsprozess häufiger auf die letzte statt auf die alten Versionen zugegriffen wird, müsste jedes Mal die komplette Berechnung durchgeführt werden. Deshalb ist es bei Verwendung dieses Verfahrens sinnvoll, vollständige Zwischenversionen abzulegen. Ein alternativer Ansatz ist das Rückwärtsdelta, bei dem die letzte Version immer vollständig und die Änderung zur Vorgängerversion als Delta vorliegt.

Merge: Ein Merge ist das Zusammenführen zwei verschiedener Zustände einer Datei. Die Voraussetzung dafür ist das Bestimmen der Änderungen durch die Operation Diff. Praktisch bewährt hat sich der Drei-Wege-Merge, bei dem zusätzlich die Elternversionen miteinbezogen werden. Bereiche, die sich nicht überschneiden, werden dann automatisch zusammengeführt. Jedoch ist dies kein Garant für absolute semantische Korrektheit des Quellcodes. Ein Konflikt besteht, wenn an der gleichen Position in beiden Dateien Änderungen vorgenommen wurden. Dieser muss dann vom Entwickler manuell beseitigt werden, indem er sich für eine Variante entscheidet. Sehr hilfreich beim Diff und Merge sind grafische Benutzeroberflächen, die beide Dateien nebeneinander anzeigen und die Änderungen hervorheben.

Branch: Normal erfolgt die Entwicklung auf dem Hauptzweig (trunk[47]) durch Verwendung einer linearen Versionierung für die einzelnen Dateien. Stehen komplexe Änderungen an oder ist ein Freigabestand (Release) erreicht, ist es günstig, einen Nebenzweig

[43] Git wurde von Linus Torvalds für die Entwicklung des Linux-Kernels initiiert, der ca. 22.000 Dateien umfasst. Er wollte ein System, das viele unabhängige Entwickler einbindet, flexible Verzweigungen erlaubt, Sicherheit gegen (böswillige) Veränderungen durch Hashes bietet und vor allem performant ist.

[44] s. (Mackall, 2006)

[45] sandbox ⟨engl., „Sandkasten“⟩

[46] difference ⟨engl., „Unterschied“⟩

[47] trunk ⟨engl., „Baumstamm“⟩

(branch[48]) zu eröffnen. Auf diesem Zweig können im ersten Fall der neue Programmcode ausgiebig getestet oder im zweiten Fall Patches z. B. für Sicherheitslücken entwickelt werden, ohne den Hauptzweig zu beeinflussen. Wahlweise können die Änderungen durch einen Merge in den Hauptzweig übernommen werden.

Tag: Ein Tag[49] ist eine Menge von Dateiversionen, die einen Stand des Softwareprodukts bilden. Von einer Datei darf dabei nur eine Version enthalten sein. Wenn wie bei CVS jede Datei eine vom Gesamtprojekt unabhängige Versionsnummerierung verwendet, ist es praktisch unmöglich, die zu einem Stand passenden Dateiversionen später wiederzufinden. Der Tag bekommt einen eindeutigen Namen zugewiesen und lässt sich effizient im Repository speichern. Meistens werden Tags für Freigabestände oder wichtige Zwischenstände vergeben.

Fazit

(Estublier u. a., 2005) stellen fest, dass das Software Configuration Management die erfolgreichste Software-Entwicklungs-Disziplin ist und die Forschung sowohl im akademischen als auch im unternehmerischen Bereich vorangetrieben wurde. Die Basistechnologien wurden vor 20 Jahren entwickelt und haben sich bis heute kaum verändert. Es ist zu unterscheiden zwischen den einfacheren SCM-Systemen, wie CVS, die verständliche Konzepte verwenden und dadurch sehr erfolgreich sind, und komplexe SCM-Systeme, die ausgefeiltere, aber auch entsprechend aufwändigere Methoden anwenden, um große Projekte zu bewältigen. Laut (Conradi u. Westfechtel, 1998) hat sich die Prozessmodellierung im Software Configuration Management als sehr erfolgreich erwiesen. (Estublier u. a., 2005) führen drei Kriterien eines SCM-Systems für die Akzeptanz beim Kunden auf, die sich problemlos auf andere Produkte übertragen lassen.

1. Der Wunsch des Kunden zur Lösung eines bestimmten Problems oder der Verfügbarkeit einer bestimmten Funktionalität bzw. Produkteigenschaft.

2. Die Fähigkeit des Herstellers eines SCM-Werkzeugs, die gewünschte Funktionalität bereitzustellen.

3. Die Bereitschaft des Kunden, den möglichen Mehraufwand, der durch die neue Funktionalität entsteht, zu akzeptieren.

Weiterhin äußern sich die Autoren kritisch zu neueren Forschungsansätzen, wie z. B. von (Zeller u. Snelting, 1997):

> „*From a researchers' point of view, research has continuously improved the state of the art by offering new or alternative modeling capabilities. From a practitioners' point of view, however, a sizeable portion of these approaches are overkill.*“

[48] branch ⟨engl., „Abzweig, Zweig“⟩
[49] tag ⟨engl., „Etikett, Marke, Schild“⟩

2.3.3 Subversion

Motivation für die Neuentwicklung: Im Jahr 2000 wurde die Entwicklung von Subversion durch die Firma CollabNet gestartet, die eigens dafür Karl Fogel einstellten, der weitreichende Erfahrungen mit dem Einsatz von CVS besaß (Fogel u. Bar, 2002). Die Motivation für die Neuentwicklung war, ein verbessertes CVS zu programmieren, das einige konzeptionelle Schwächen von CVS beseitigen sollte. CVS besaß damals im Open-Source-Umfeld eine weite Verbreitung. Im Jahr 2004 wurde mit Subversion 1.0 das erste Release für den produktiven Einsatz vorgestellt. Das aus der internen Dokumentation entstandene Buch (Collins-Sussman u. a., 2008) ist frei erhältlich[50].

Unterschiede zu CVS

Repository: Ursprünglich verwendete Subversion zur Speicherung der Versionsdaten die Datenbank Berkeley DB. Seit Version 1.1 ist das auch direkt im Dateisystem möglich. Der Zugriff ist über verschiedene Methoden möglich.

Schema	Zugriffsmethode
`file:///`	Direkter Zugriff auf ein Repository auf dem lokalen Rechner
`http:///`	Zugriff auf einen Apache-Server über das WebDAV[51]-Protokoll
`https:///`	Wie `http://`, jedoch mit SSL-Verschlüsselung[52]
`svn:///`	Zugriff über ein eigenes Protokoll auf einen `svnserve`-Server
`svn+ssh:///`	Wie `svn://`, jedoch über einen SSH-Tunnel[53]

Tabelle 2.3: Zugriffsmethoden auf ein Subversion-Repository nach (Collins-Sussman u. a., 2008)

Im Gegensatz zu CVS lassen sich nun Dateien verschieben, ohne dass die Historie verloren geht, sowie Verzeichnisse verwalten. Commits sind darüber hinaus – in Anlehnung an das ACID[54]-Prinzip bei Datenbanken – atomar[55] und isoliert[56]. Das bedeutet, Commits werden ganz oder gar nicht ins Repository übernommen und nicht durch parallel ablaufende Commits beeinflusst.

[50] Veröffentlicht unter der *Creative Commons Attribution License*. `http://creativecommons.org/licenses/by/2.0/`

[51] WebDAV = Web-based Distributed Authoring and Versioning. WebDAV ist eine Erweiterung des HTTP-Protokolls, um Dateien über ein Netzwerk bearbeiten und verwalten zu können. Eine Versionsverwaltung ist enthalten.

[52] SSL = Secure Sockets Layer. Siehe Absatz Java/RMI auf Seite 237.

[53] SSH = Secure Shell. SSH ist ein Netzwerkprotokoll zur Herstellung sicherer Verbindungen zwischen zwei Rechnern.

[54] ACID = Atomicity, Consistency, Isolation, Durability. s. (Gray, 1981), (Härder u. Reuter, 1983)

[55] atomos ⟨griech., „unteilbar“⟩

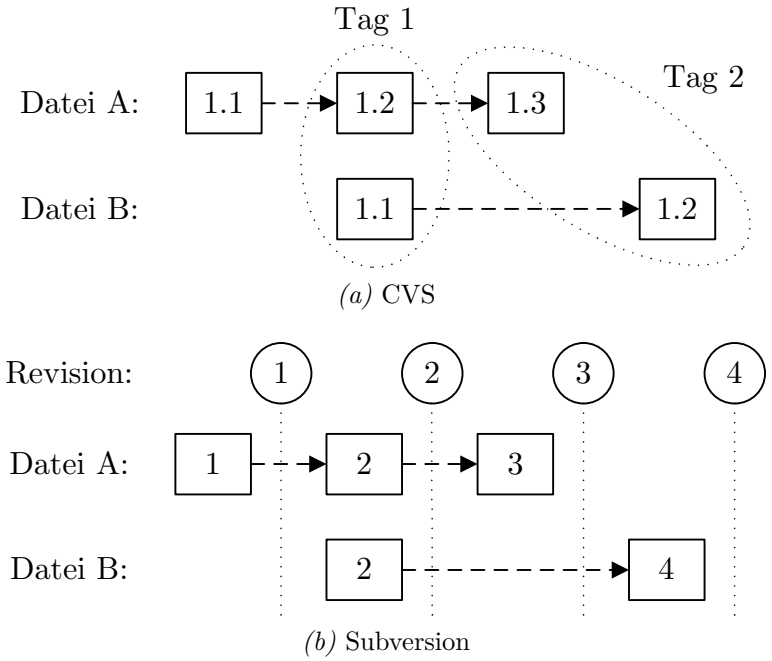
[56] isola ⟨ital., „Insel“⟩

Sandbox: Subversion speichert von jeder Datei eine zweite Kopie im Verzeichnis *.svn* parallel zur Datei. So verdoppelt sich zwar der Speicherbedarf, aber so werden auch Serverzugriffe bei einigen Operationen vermieden. Jederzeit lassen sich nun offline die Änderungen seit dem letzten Update oder Commit ermitteln sowie die Änderungen einfach zurücknehmen. Vor einem Commit berechnet Subversion die Änderungen aus der Arbeitskopie und braucht nur die Deltas zum Server schicken.

Tags und Branches: Subversion speichert den Hauptentwicklungszweig eines Projekts in dem Verzeichnis *trunk*. Das Erzeugen eines Tags und eines Branches wird durch das Setzen von Verweisen auf die zugehörigen Dateiversionen in der internen Datenbank erreicht. Die Subversion-Entwickler bezeichnen dies als „billige Kopie“. Neben dem trunk-Verzeichnis existieren noch die Verzeichnisse *tags* und *branches*, die die Kopien aufnehmen.

Binärdateien: Subversion kann automatisch erkennen, ob eine Text- oder Binärdatei vorliegt. In CVS mussten vorher Dateierweiterungen mit dem entsprechenden Typ verknüpft werden. Im Gegensatz zu CVS speichert Subversion Binärdateien über das Deltaverfahren ab und spart dadurch Platz im Repository ein.

Versionsnummerierung: Während CVS die Versionsnummer einer Datei nur dann hochzählt, wenn sie committet wurde (s. Abbildung 2.9a), erhöht Subversion bei *jedem* Commit die Revisionsnummer von *allen* Dateien im Projekt (s. Abbildung 2.9b). Die Zuweisung von Dateiversionen zu einer Konfiguration muss in CVS über einen Tag erfolgen. In Subversion reicht dazu die Angabe einer Revisionsnummer aus, trotzdem ist es empfehlenswert, darüber hinaus Tags mit selbsterklärenden Namen zu verwenden.



(a) CVS

(b) Subversion

Abbildung 2.9: Versionsnummerierung

CVS nutzt als Versionsnummernformat Ganzzahlen, getrennt durch einen Punkt, beginnend bei `1.1`. Für Branches werden drei Ganzzahlen kombiniert, von denen die ersten

zwei der Ausgangsversion entsprechen, z. B. Ausgangsversion = `1.5`, Branch = `1.5.2`. Die erste Dateiversion im Branch erhält dann die Nummer `1.5.2.1`. In einem Branch kann wieder ein neuer Branch erstellt werden, der vielleicht die Nummer `1.5.2.14.2` erhält. Hier wird ersichtlich, dass dieses Konzept aufgrund der Unübersichtlichkeit für den Nutzer seine Grenzen hat.

In Subversion setzt sich die Versionsummer nur aus einer Ganzzahl zusammen. Beim Anlegen eines Branches vergibt der Entwickler einen schlüssigen Namen, wie z. B. `Patch_R1.0`. Der Zugriff erfolgt dann über die URL `/project/branch/Patch_R1.0`.

2.4 Objektversionierung

2.4.1 Definitionen

Die nächsten drei Abschnitte führen diejenigen Definitionen und Erkenntnisse der Dissertation von (Firmenich, 2002) auf, die für die vorliegende Arbeit von Bedeutung sind und eine einheitliche Bezeichnung sicherstellen sollen. Einzig die Mengen Ω und M wurden in Q und Ω umbenannt. Wenn Mengen oder Relationen mit einem Überstrich angeben sind, z. B. $\overline{Q}$, dann gelten sie nur innerhalb des Workspace.

Objekt: Die Definition und Darstellung ist im Abschnitt 2.1.2 auf Seite 14 zu finden. Objekte werden mit lateinischen Kleinbuchstaben bezeichnet und jedem Objekt wird ein projektweit eindeutiger und persistenter Identifikator, der Persistente Objektidentifikator (POID), zugeordnet.

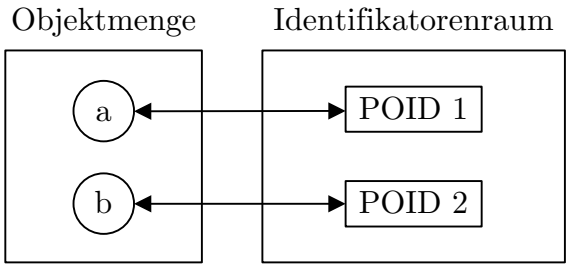


Abbildung 2.10: Bijektive Relation zwischen Objekt und persistentem Objektidentifikator

Objektmenge: Die Objektmenge Q speichert alle Objekte eines Projekts.

Objektversion: Eine Objektversion a_i ist ein eindeutiger Zustand eines Objekts a, der dauerhaft im Projekt gespeichert wird.

Objektversionsmenge: Die Objektversionsmenge Ω enthält alle Objektversionen eines Projekts. Die Abbildung $\omega : \Omega \rightarrow Q$ weist jeder Objektversion $a_i \in \Omega$ ein Objekt $a \in Q$ zu. Objekt und Objektversion bilden die Knoten eines Graphen, die Abbildung ist eine gerichtete Kante, dargestellt durch einen strichpunktierten Pfeil.

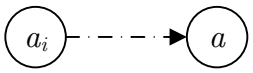


Abbildung 2.11: Objektversionsabbildung

Objektänderung: Eine Objektänderung (a_i, a_j) liegt vor, wenn sich eine neue Objektversion a_j aus einer Änderung einer bestehenden Objektversion a_i ergibt. Die Objektänderung wird als gerichtete, gestrichelte Kante dargestellt.

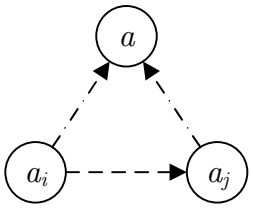


Abbildung 2.12: Objektänderung

Das Erzeugen und Löschen von Objektversionen ließe sich so nicht darstellen, da es keinen Vorgänger bzw. Nachfolger gibt. Aus diesem Grund führte (Firmenich, 2002) virtuelle Objektversionen $\delta_i \in \Delta$ ein. Im Beispiel aus Abbildung 2.13 ist a_1 die Ursprungsversion des Objekts a, da die Objektänderungsbeziehung $(\delta_0, a_1) \in \Delta \times \Omega$ existiert. Die Objektversion a_2 wurde gelöscht, da die Objektänderungsbeziehung $(a_2, \delta_1) \in \Omega \times \Delta$ existiert.

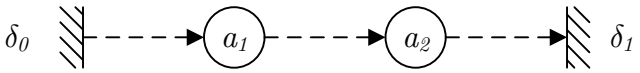


Abbildung 2.13: Virtuelle Objektversionen

Objektversionsgraph: Abbildung 2.13 enthält die grafische Darstellung eines einfachen Objektversionsgraphen. Der Objektversionsgraph zeigt die Entwicklungsgeschichte eines Objekts im Bearbeitungsprozess und besteht aus den Objektversionen als Knoten sowie den Objektänderungen als Kanten. Er ist gerichtet und azyklisch. Abbildung 2.14 zeigt im Vergleich zu vorher einen etwas komplexeren Objektversionsgraph, da hier nicht nur eine lineare Versionierung vorkommt, sondern auch parallele Varianten des Objekts. Anhand der Anzahl von Ausgangskanten eines Knotens lässt sich das Entstehen der Revisionen von dem der Varianten unterscheiden. Besitzt ein Knoten mehr als eine Eingangskante, dann führt er Varianten wieder zusammen. Diese als *Merge* bezeichnete Operation wird später noch von Bedeutung sein. Die virtuelle Objektversion δ_0 und die Kante zur Ursprungsversion des Objekts werden im Graphen oft weggelassen, da diese Version oft leicht zu erfassen ist.

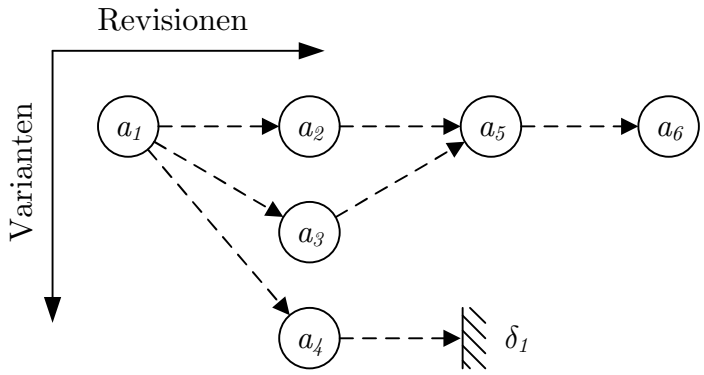


Abbildung 2.14: Objektversionsgraph mit Varianten

Objektversionsrelation: Die Objektversionsrelation $V \subseteq \Omega_\Delta \times \Omega_\Delta$ mit $\Omega_\Delta := \Omega \cup \Delta$ enthält alle Beziehungen zwischen den Objektversionen.

Objektbeziehungen: (Pahl u. Beucke, 2000) bemessen der *„strukturierten und formalisierten Definition von Objektbeziehungen“* im vernetzt-kooperativen Planungsprozess eine hohe Bedeutung bei, damit im *„Update-Prozess getrennt bearbeiteter Informationsmengen“* die Änderungen an Objekten eindeutig verfolgt werden können. Sie führen dazu die Begriffe Bindung und Abhängigkeit ein.

Bindung: Ein Objekt b ist an ein Objekt a gebunden, wenn mindestens eine der folgenden Bedingungen erfüllt ist:

- Methoden von a ändern Attribute von b.
- Methoden von b verwenden Attribute von a.

Eine sofortige Aktualisierung von Attributen einer gebundenen Version lässt sich für die Objektversionierung nicht anwenden, da einmal im Projekt gespeicherte Objektversionen nicht mehr geändert werden dürfen. Als Lösung kommt nur die Erzeugung eines neuen Nachfolgers der gebundenen Objektversion in Betracht.

Bindungsrelation: Ist eine Objektversion $b_j \in \Omega$ an eine Objektversion $a_i \in \Omega$ gebunden, so ist das geordnete Paar (a_i, b_j) ein Element der Bindungsrelation $B \subseteq \Omega \times \Omega$:

$$B := \{(a_i, b_j) \in \Omega \times \Omega \mid a_i \text{ bindet } b_j\} \tag{2.1}$$

Die geometrische Darstellung ist ein Graph mit einer von a_i nach b_j durchgezogenen gerichteten Kante.

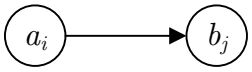


Abbildung 2.15: Bindungsrelation

Abhängigkeit: Eine Objektversion $c_k \in \Omega$ heißt abhängig von einer Objektversion $a_i \in \Omega$, wenn im Bindungsgraphen mindestens ein Weg von a_i nach c_k existiert. Abhängigkeiten werden durch einen durchgezogenen Pfeil mit offener Pfeilspitze dargestellt.

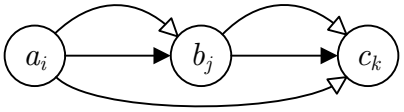


Abbildung 2.16: Abhängigkeit

Die Relation H aller möglichen Wege wird als transitive Hülle bezeichnet.

$$H := \{(a_i, b_j) \in \Omega \times \Omega \mid b_j \text{ ist von } a_i \text{ abhängig } \} \tag{2.2}$$

Bei der Definition von Abhängigkeiten zwischen Objektversionen müssen zwei Bedingungen eingehalten sein:

- Eine Objektversion a_i darf nicht gleichzeitig von zwei Versionen des Objekts b abhängig sein.

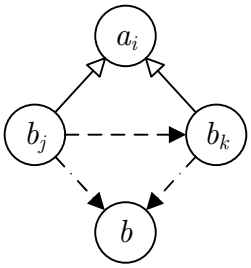


Abbildung 2.17: Ungültige Abhängigkeit (Bedingung 1)

- Objektversionen eines Objekts a dürfen nicht voneinander abhängig sein.

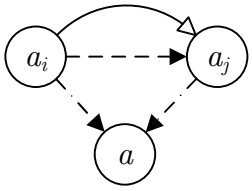
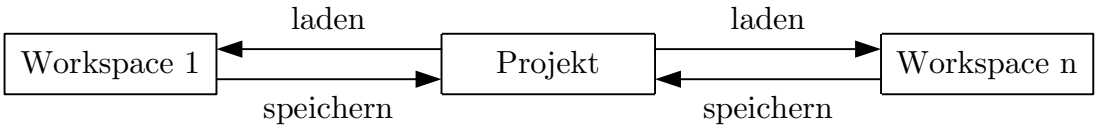


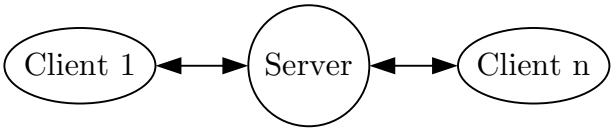
Abbildung 2.18: Ungültige Abhängigkeit (Bedingung 2)

2.4.2 Verteilte Bearbeitung

Workspace: Die strukturierte Menge aller Objektversionen und die Abhängigkeiten zwischen ihnen werden zentral im *Projekt* gespeichert. Jeder Bearbeiter besitzt einen eigenen *Workspace*, in dem er Teile des zentral abgelegten Planungsmaterials laden, bearbeiten und zurückspeichern kann. Von einem Objekt kann maximal eine Version in den Workspace geladen werden, was zu einer unversionierten Sicht führt. Der Projekt-Workspace-Ansatz lehnt sich zweckmäßigerweise an die Client-Server-Architektur an.



(a) Projekt und Workspace



(b) Client-Server-Architektur

Abbildung 2.19: Verteilte Bearbeitung

Operationen: (Firmenich, 2002) beschreibt für die verteilte Bearbeitung eine Reihe notwendiger Operationen. Sie erlauben eine sehr flexible Bildung von Teilmengen $S \subseteq \Omega$, die in den Workspace geladen werden. Einschränkungen bestehen darin, dass nur eine Version je Objekt und die davon abhängigen Objektversionen übertragen werden dürfen. An dieser Stelle soll nur ein kurzer Überblick gegeben werden. Die ausführlichen Beschreibungen sind in der Dissertation von Firmenich zu finden.

Operation	Beschreibung
Selektion	Die Selektion ist notwendig, um aus dem zentralen, versionierten Modell ein konsistentes und unversioniertes Teilmodell auszuwählen.
Laden	Die Operation Laden überträgt das Teilmodell, inklusive Objektversionen und Bindungen, vom Projekt in den lokalen Workspace des Clients unter Beachtung einiger Vorschriften. Von jeder Objektversion a_i wird eine neue Objektversion a_j erzeugt und in die Menge W eingetragen. Weiterhin enthält die Bearbeitungsrelation $\overline{V}$ nach dem Laden alle Paare (a_i, a_j). Die Relation $\overline{B}$ enthält nach Ausführen des Ladens alle Bindungen des Workspace.
Bearbeitung	Objektversionen können erzeugt, geändert und gelöscht werden. Eine Änderung ist erfolgt, wenn sich mindestens ein Attribut des Objekts geändert hat. Die aktuellen Zustände der Objektversionen während der Bearbeitung werden durch Eintragung in entsprechende Mengen und Relationen kenntlich gemacht.
Entladen	Geänderte, ausgetragene und ungeänderte Objektversionen können aus der Bearbeitungsrelation $\overline{V}$ ausgetragen werden. Die Bindungen müssen dann entsprechend angepasst werden.

Zusammenführen	Leitet entweder ein Benutzer nacheinander oder mehrere Benutzer durch gleichzeitige Bearbeitung mehr als einen Nachfolger derselben Objektversion ab, so entstehen Varianten. Die Operation *Zusammenführen* lädt alle zusammenführbaren Objektversionen in den Workspace und erstellt durch manuelle Attributfestlegung die Zwischenversion z_r. Die Bindungen muss der Nutzer in der Regel auch selbst anpassen.
Modernisieren	Überholte Objektversionen sind gebundene Objektversionen, deren bindende Objektversionen Nachfolger besitzen. Die Operation *Modernisieren* erzeugt eine neue Version der gebundenen Objektversion und bindet diese an eine Version des bindenden Objekts ohne Nachfolger.
Reduktion	Ungeänderte Objektversionen müssen nicht im Projekt gespeichert werden und können aus dem Workspace entfernt werden. Die Bindungen werden durch die Operation entsprechend angepasst.
Speichern	Der Nutzer hat nach der Bearbeitung die Wahl, die Ergebnisse zu verwerfen oder im Projekt zu speichern. Beim Verwerfen werden alle Mengen und Relationen des Workspace geleert. Vor dem Speichern ist zu prüfen, ob neuere Objektversionen der geladenen Objekte im Projekt vorliegen, falls ja, ist der Workspace zu aktualisieren. Danach werden alle Änderungen durch Anlegen neuer Objektversionen und Bindungen im Projekt übernommen.

Tabelle 2.4: Operationen der verteilten Bearbeitung nach (Firmenich, 2002)

Mengenalgebra: Die verteilten Operationen sind mengentheoretisch formal beschrieben und ihnen liegt eine algebraische Struktur[57] zugrunde. Es stehen die drei Grundoperationen Vereinigung $\sqcup$, Durchschnitt $\sqcap$, Komplement $\sim$ zur Verfügung, von denen sich alle anderen Mengenoperation ableiten lassen. Die Grundoperationen, auch *innere Verknüpfungen* genannt, erlauben die Bildung von Teilmengen aus der Menge Ω, aus denen sich wiederum Teilmengen berechnen lassen.

2.4.3 Feature-Logic

Feature-Logic: Die Feature-Logic ist eine Mengenalgebra, die zusätzlich die Teilmengenbildung über die Eigenschaften der Mengenelemente unterstützt. Die Eigenschaften (*Features*) dienen dabei als Operatoren und stellen *äußere Verknüpfungen* dar. (Smolka, 1992) hat die *Feature-Constraint-Logic* im Bereich der logischen Programmierung und Wissensrepräsentation vorgestellt. (Zeller, 1997) zeigte die Anwendbarkeit der nun Feature-Logic genannten Algebra auf das Software Configuration Management.

[57] s. (Scheid, 1994)

Ein Feature f ist eine Eigenschaft eines Elements und weist ihm einen Wert zu, der selbst ein Element ist. Formal gesehen, ist ein Feature die Menge aller geordneten Paare $(Element, Wert)$. Die Domäne D^I ist eine Menge, die alle Elemente enthält und sich in die Menge D^I_A der primitiven und in die Menge D^I_E der nicht-primitiven Elemente unterteilt. Primitiven Elemente ist ein atomarer Wert zugeordnet und sie besitzen keine Features. Nicht-primitive Elemente sind nicht atomar und können Features besitzen. Die Feature-Algebra I ist ein geordnetes Paar $(D^I, \cdot^I)$, wobei die Interpretationsfunktion $\cdot^I$ jedem primitiven Element einen atomaren Wert und jedem Feature f eine Menge geordneter Paare $f^I \subseteq D^I \times D^I$ unter bestimmten Bedingungen zuweist (Zeller, 1997).

Der Feature-Graph stellt die Feature-Algebra grafisch dar. Die Elemente werden als Knoten und die Features als beschriftete, gerichtete Kanten dargestellt. Atomare Werte erscheinen in einem Rechteck, das mit einer ungerichteten Kante mit dem nicht-primitiven Element verbunden wird. Im Feature-Graph der Abbildung 2.20 hat das Element x zwei Features f und g. Das Feature f zeigt auf das nicht-primitive Element y, das Feature g auf das primitive Element a, dem der atomare Wert 2 zugeordnet ist. Das Element y besitzt seinerseits das Feature f, welches auf das Element z verweist. Die Domäne ist folgendermaßen belegt: $D^I = \{a, x, y, z\}$ mit $D^I_A = \{a\}$ und $D^I_E = \{x, y, z\}$.

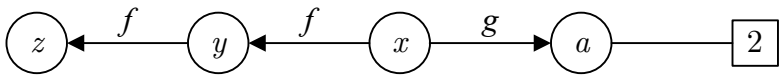
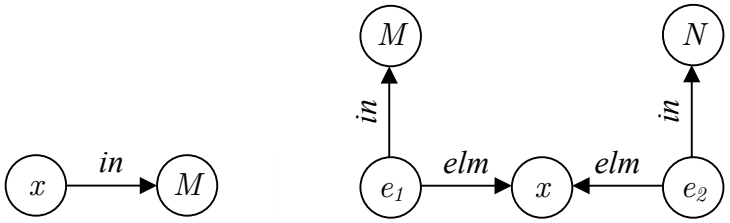


Abbildung 2.20: Beispiel eines Feature-Graphen

Mit der Feature-Logic lassen sich Mengen und Relationen abbilden. Wie im Feature-Graph der Abbildung 2.21a zu sehen, verdeutlicht das Feature *in* die Zugehörigkeit des Elements x zur Menge M : $M = \{x\}$. Da das Element x kein weiteres Feature *in* besitzen darf, kann es in keiner weiteren Menge eingetragen werden. Zu diesem Zweck werden Stellvertreterelemente e_i eingeführt (Abbildung 2.21b), die mit den Features *in* und *elm* die Zuordnung der Elemente zu den Mengen vornehmen: $M = \{x\}, N = \{x\}$.



(a) Exklusive Mengenzugehörigkeit

(b) Mehrfache Mengenzugehörigkeit

Abbildung 2.21: Feature-Graph: Modellierungen der Mengenzugehörigkeit

Weiterhin können durch die Stellvertreterelemente und die Feature *in*, *src*, *dst* Relationen wie in der Abbildung 2.22 modelliert werden: $R = \{(x, y), (y, z)\}$

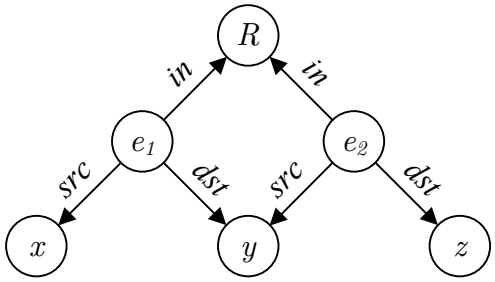


Abbildung 2.22: Feature-Graph: Modellierung von Relationen

Feature-Term (Zeller, 1997)**:** Ein Feature-Term beschreibt eine Menge von Elementen $S^I \subseteq D^I$ der Feature-Algebra I. Nachstehende Tabelle führt die vorhandenen Operationen der Feature-Logic auf, wobei S und T jeweils einen Feature-Term repräsentieren. Firmenich fügte noch die Operation *Extraktion* hinzu, die den Wert des Features eines Elements zurückgibt. Die Operationen lassen sich ineinander verschachteln, wodurch lange Ausdrücke entstehen können. Firmenich ordnete außerdem jedem mathematischem Symbol ein Textsymbol zu, auf deren Basis er einen Interpreter für Feature-Terme entwarf.

Operator	Mathem. Symbol	Text-symbol	Erklärung
Domäne	$\top$	`[]`	Alle Elemente in der Domäne.
Leere Menge	$\bot$	`{}`	–
Extraktion	$f(S)$	`S.f`	Wert des Features f der Elemente in S.
Selektion	$f : S$	`f:S`	Elemente, deren Feature f den Wert von S besitzen.
Existenz	$f : \top$	`f:[]`	Elemente, deren Feature f existiert.
Divergenz	$f \uparrow$	`f^`	Elemente, deren Feature f nicht existiert.
Übereinstimmung	$f \downarrow g$	`f==g`	Elemente, deren Features f und g gleich sind.
Verschiedenheit	$f \uparrow g$	`f!=g`	Elemente, deren Features f und g unterschiedlich sind.
Komplement	$\sim S$	`~S`	Differenz zwischen der Domäne und der Menge S.
Durchschnitt	$S \sqcap T$	`[S,T]`	Elemente, die sowohl in S als auch in T vorkommen.
Vereinigung	$S \sqcup T$	`{S,T}`	Elemente, die entweder in S, T oder beiden vorkommen.
Implikation	$\rightarrow T$	`S->T`	Wenn Elemente in S vorkommen, müssen sie auch in T vorkommen.
Äquivalenz	$S \leftrightarrow T$	`S<->T`	Elemente sind entweder in S und T oder weder in S und T dafür aber in der Domäne.
Term-Äquivalenz	$S = T$	–	S und T sind äquivalent, wenn sie die gleichen Elemente beschreiben.

Tabelle 2.5: Feature-Logic-Operationen

Die Anwendung der Feature-Terme soll an den vorherigen Beispielen kurz gezeigt werden.

Operation	Beispiel	Feature-Term mathematisch	textuell	Ergebnis
Domäne	Abb. 2.21a	$\top$	`[]`	`{x,M}`
Leere Menge	Abb. 2.21a	$\bot$	`{}`	`{}`
Extraktion	Abb. 2.21a	$in(x)$	`x.in`	`{M}`
Selektion	Abb. 2.21a	$in : [x]$	`in:M`	`{x}`
Selektion	Abb. 2.21b	$in : [M]$	`in:M`	`{e1}`
Selektion, Extraktion	Abb. 2.21b	$elm(in : [M])$	`[in:M].elm`	`{x}`
Vereinigung	Abb. 2.21b	$\{in : [M], in : [N]\}$	`{in:[M],in:[N]}`	`{e1,e2}`
Durchschnitt	Abb. 2.21b	$[in : [M], in : [N]]$	`[in:[M],in:[N]]`	`{}`
Vereinigung, Extrakt.	Abb. 2.21b	$elm(\{e1, e2\})$	`{e1,e2}.elm`	`{x}`
Selektion, Extraktion (Nachfolger von x)	Abb. 2.22	$dst([src : [x], in : [R]])$	`[src:x, in:R].dst`	`{y}`

Tabelle 2.6: Anwendung der Feature-Terme

Anwendung der Feature-Logic auf die Objektversionierung: Die Feature-Logic eignet sich gut, um die Mengen und Relationen der Objektversionierung zu modellieren und Teilmengen durch geeignete Feature-Terme abzufragen. Die Abbildung $\omega : \Omega \to Q$ für die Zuordnung der Objektversionen zu ihren Objekten wird beispielhaft wie im nächsten Feature-Graphen umgesetzt. Jede Objektversion a_i erhält ein Feature obj, das auf das Objektelement a zeigt.

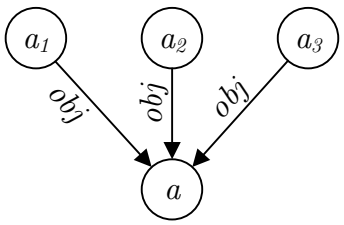


Abbildung 2.23: Feature-Graph: Zuordnung der Objektversionen zu den Objekten

Die Objektversionsrelation V kann durch das Hinzufügen von zusätzlichen Elementen v_i umgesetzt werden. Das Beispiel zeigt die Historie des Objekts b mit der Erzeugung, Änderung und Löschung. Die Elemente v_i modellieren den Übergang von einer Objektversion zur nächsten mit den Features src und dst sowie die eigene Zugehörigkeit zu V mit dem Feature in.

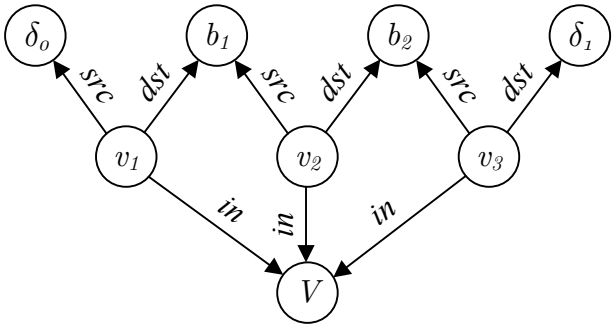


Abbildung 2.24: Feature-Graph: Objektversionsrelation

Für die Modellierung der Bindungsrelation B und der transitiven Hülle $H(B)$ müssen zusätzliche Elemente b_i und h_i eingeführt werden. Die transitive Hülle wird nur für Wege mit der Länge größer als 1 in die Feature-Logic eingetragen. Im Beispiel ist die Objektversion y_1 an x_1 und z_1 an y_1 gebunden. Die Objektversion z_1 ist von x_1 abhängig, da ein Weg zwischen beiden im Bindungsgraphen existiert.

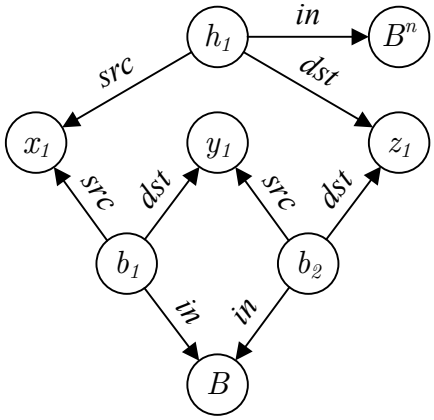


Abbildung 2.25: Feature-Graph: Bindungsrelation

Datenspeicherung der Feature-Logic: Die Daten der Feature-Logic lassen sich prinzipiell in beliebige Datencontainer speichern. (Firmenich, 2002) präsentiert einen Ansatz zur Ablage in relationalen Datenbanken (RDB)[58]. Die Relation *Domain* enthält alle Elemente und die Relation *Atom* alle primitiven Elemente, denen ein atomarer Wert zugeordnet wird. Der Name der primitiven Elemente wird in der Form _i generiert, wobei i ein automatisch erhöhter Zähler ist. Die Relation *Feature* speichert alle Features und die Relation *Relslot* – als das Herzstück der Feature-Logic – die Zuordnung *Element-Feature-Element*. Der Wert in der dritten Spalte (*value*) entspricht einem Element aus der Domäne und nicht einem atomaren Wert aus der gleichnamigen Spalte von *Atom*.

Domain

element
...

Feature

feature
...

Atom

element	value
...	...

Relslot

object	feature	value
...	...	...

[58] s. (Codd, 1970), (Kemper u. Eickler, 2006)

In (Richter, 2003) wurde ein Umsetzungskonzept für die Feature-Logic vorgestellt, um Implementierungen für weitere Datencontainer einfach einzubinden. Die Schnittstelle *FeatureLogic* definiert Methoden zum Hinzufügen und Löschen von Elementen und Features sowie Methoden für die Operationen der Feature-Logic. Der *FeatureLogicAdapter* implementiert als abstrakte Klasse nur einen Teil der Methoden, um den Feature-Logic-Interpreter einzubinden. Die Klasse *FeatureLogicMyContainer* ist vom Adapter abgeleitet und muss für jeden Container vollständig ausprogrammiert werden. Da die relationalen Datenbanken geringfügig unterschiedliche Sprachvarianten der SQL verwenden und geschachtelte SQL-Abfragen unterschiedlich unterstützen, muss für jede Datenbank eine eigene Implementierung geschrieben werden.

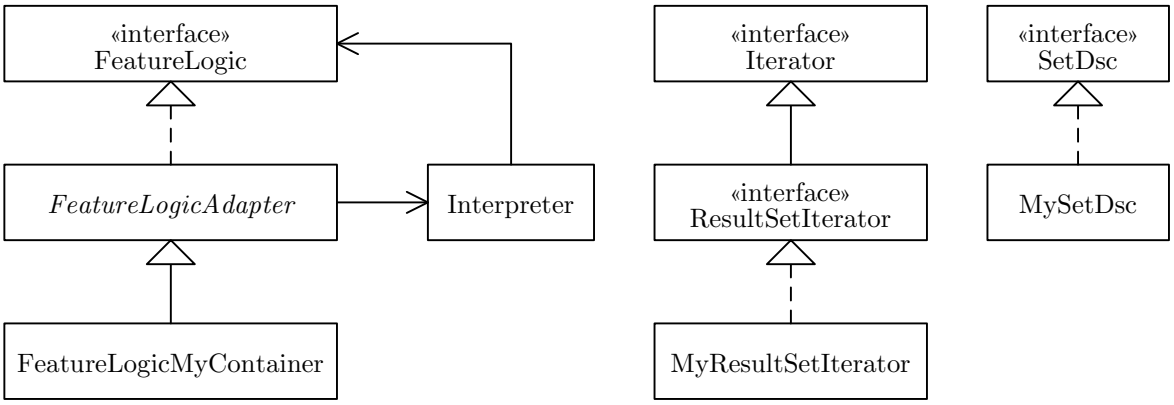


Abbildung 2.26: UML-Klassendiagramm der Feature-Logic

Die Schnittstelle *ResultSetIterator* ist von *java.util.Iterator* abgeleitet und dient dem Traversieren der vom Datencontainer zurückgelieferten Ergebnismenge. Die Schnittstelle *SetDsc*, eine Abkürzung von *set descriptor*, dient zur Beschreibung von Teilmengen der Datenquelle. Da dies sehr unterschiedlich geschehen kann, besitzt die Schnittstelle für eine flexible Handhabung keine Methoden. Im Falle von relationalen Datenbanken enthält die Klasse *MySetDsc* einen SQL-Ausdruck.

2.4.4 Versionierung strukturierter Objektmengen mit Hilfe von textbasierten Versionsverwaltungssystemen

In (Firmenich u. a., 2005) stellen die Autoren einen Ansatz vor, wie sich objektorientierte Modelle mit textbasierten Versionsverwaltungssystemen (VCS) versionieren lassen und nennen diese Umgebung *objectVCS*. Abbildung 2.27 zeigt die grundlegende Architektur, bei der auf die zusätzliche Modellierung der Versionen und Bindungen durch die Feature-Logic verzichtet wird. Stattdessen wird die Verwaltung der Objektversionen einem VCS übertragen, die sich in der Praxis für die Versionierung von Software bestens bewährt haben und effizient mit Speicherplatz umgehen.

Die kleinste versionierbare Einheit eines VCS ist die Datei. Da Objekte versioniert werden sollen, muss jedes Objekt in eine Datei gespeichert werden. Zu diesem Zweck wurde ein

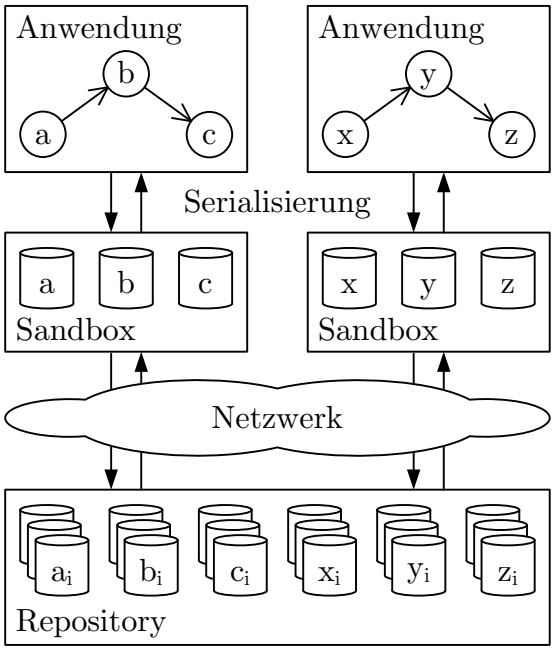


Abbildung 2.27: objectVCS

XML-Serialisierer erstellt, der jedes Objekt in eine XML-Datei schreibt. Der Dateiname entspricht der POID des Objekts – zusammengesetzt aus dem Klassennamen, Benutzernamen sowie einem aufsteigenden Zähler – und der Dateiendung *xml*. Bei der Speicherung der Attribute muss zwischen einfachen und komplexen Datentypen unterschieden werden. Die einfachen können in der XML-Datei des Objekts als Wert abgelegt werden, für die komplexen wird die POID hinterlegt, mit der auf die XML-Datei des referenzierten Objekts verwiesen wird. objectVCS sorgt nach dem Deserialisieren eines Dokuments dafür, dass die POIDs den transienten Objekten im Arbeitsspeicher zugeordnet und für die Dauer der Bearbeitung in einer Tabelle (assoziatives Array) gespeichert werden. Während des Serialisierens wird dann ein transientes Objekt in die XML-Datei mit dem zugehörigen Dateinamen entsprechend der POID geschrieben.

Jedes Dokument wird in einem separaten Verzeichnis unterhalb des Projektverzeichnisses gespeichert, um die Objekte eines Dokuments von denen anderer Dokumente zu trennen. Von jedem Objekt darf sich maximal eine Version in Form der XML-Datei im Verzeichnis befinden. Weiterhin müssen die Objektversionen zueinander passen, sonst würden Fehler beim Laden des Dokuments auftreten. Versionsverwaltungssysteme verfügen über die Funktionalität des *Taggings*, mit der zueinander gehörende Versionen zu einer Konfiguration zusammengefasst und im Repository mit einem Tag i versehen werden (s. Abschnitt 2.3.2 auf Seite 36). Formell ergibt sich die Menge T getaggter Dateiversionen:

$$T_i := \{f \in V \mid f \text{ hat den Tag } i\} \tag{2.3}$$

mit f als Dateiversion.

Im folgenden Beispiel soll das Tagging verdeutlicht werden. In der ersten Dokumentversion existieren zwei Objekte a und b, wobei a von b referenziert wird. Während des Commits

zum Repository vergibt das VCS Versionsnummern für beide Objekte, so dass sie unter a_1 und b_1 gespeichert werden. Nach abgeschlossenem Commit veranlasst objectVCS, diese beiden Objektversionen unter dem Tag T_1 zusammenzufassen. Danach verändert der Nutzer nur das Objekt b und startet einen weiteren Commit. Die Objektversion b_2 wird erzeugt und zusammen mit a_1 durch das Tag T_2 gekennzeichnet. Das Objekt a muss nicht noch einmal zum Server übertragen werden, da es den gleichen Zustand wie beim ersten Commit besitzt. Es ist leicht zu erkennen, dass nach jedem Commit eine neue Dokumentversion entsteht, deren Name dem Tag entspricht.

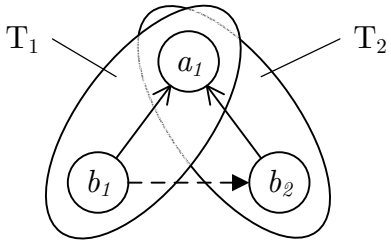


Abbildung 2.28: Beispiel für das Taggen von Objektversionen

Die meisten zentralen Versionsverwaltungssysteme unterstützen keine beliebige Variantenbildung, so dass es sinnvoll ist, diesen Ansatz auf die lineare Versionierung zu begrenzen. Dessen ungeachtet bietet objectVCS die Möglichkeit, verteilt ohne das Sperren von Dokumenten zu arbeiten, wenn geeignete Werkzeuge zum einfachen Vergleichen und Zusammenführen zur Verfügung stehen.

2.4.5 Systemarchitektur für verteilte Bearbeitung und Modelle

Um die Möglichkeiten des Objektversionierungsansatzes auszuschöpfen und trotzdem die textbasierten Versionsverwaltungssysteme weiterzuverwenden, bedarf es einer erweiterten Systemarchitektur. (Beer, 2005) entwirft eine solche, indem er *objectVCS* mit dem Ansatz von Firmenich in einer weiterentwickelten Version kombiniert. Abbildung 2.29 zeigt die vereinfachte Sicht auf die Systemarchitektur mit den drei Hauptkomponenten *Project, Workspace* und *Anwendung*. Bestehende Ingenieuranwendungen, die ihr Objektmodell in native Dateiformate speichern, werden um einen Workspace mit der Funktionalität der verteilten Bearbeitung erweitert. Jeder Workspace besitzt eine *Sandbox* als lokalen Speicherort für die verteilte Anwendung. Auf der Serverseite existiert ein zentrales Project, das für die Verwaltung der versionierten Objektmodelle und weiterer Modellierungsdaten zuständig ist und seine Daten im *Repository* speichert.

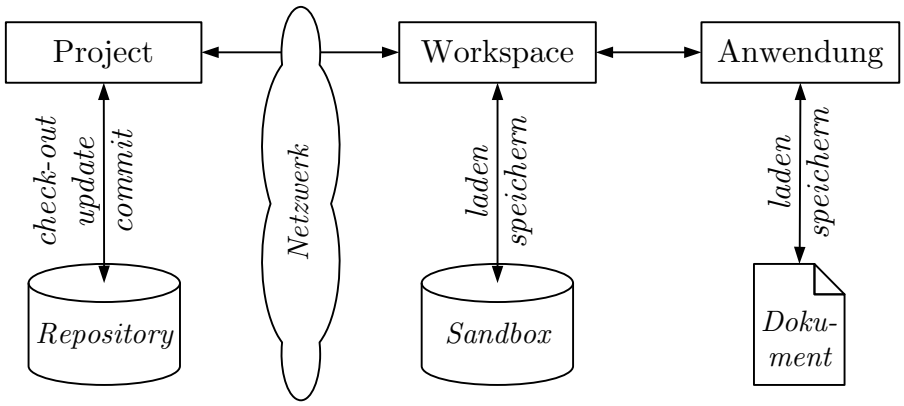


Abbildung 2.29: Vereinfachte Darstellung der Systemarchitektur nach (Beer, 2005)

Abbildung 2.30 verfeinert die Systemarchitektur in Hinsicht auf die zwei wesentlichen Datenströme zwischen Project und Workspace sowie die dafür benötigten Komponenten. Zum einen übernimmt weiterhin ein VCS in seiner effizienten und ausgereiften Art und Weise die persistente Speicherung der Objektversionen. Zum anderen wird eine zusätzliche Modellierung für die flexible Selektion von Mengen anhand von Eigenschaften und für die Definition von Bindungen benötigt, die mit Hilfe der im Abschnitt 2.4.3 auf Seite 44 vorgestellten Feature-Logic realisiert wird. Von der Feature-Logic gibt es je eine Instanz auf dem Server und auf den Client-Rechnern. Die Klassen *FLServer* und *FLClient* sorgen für die Kommunikation zwischen ihnen. Das VCS-Repository ist als Teil des Repositorys der zentrale Speicherort des VCS.

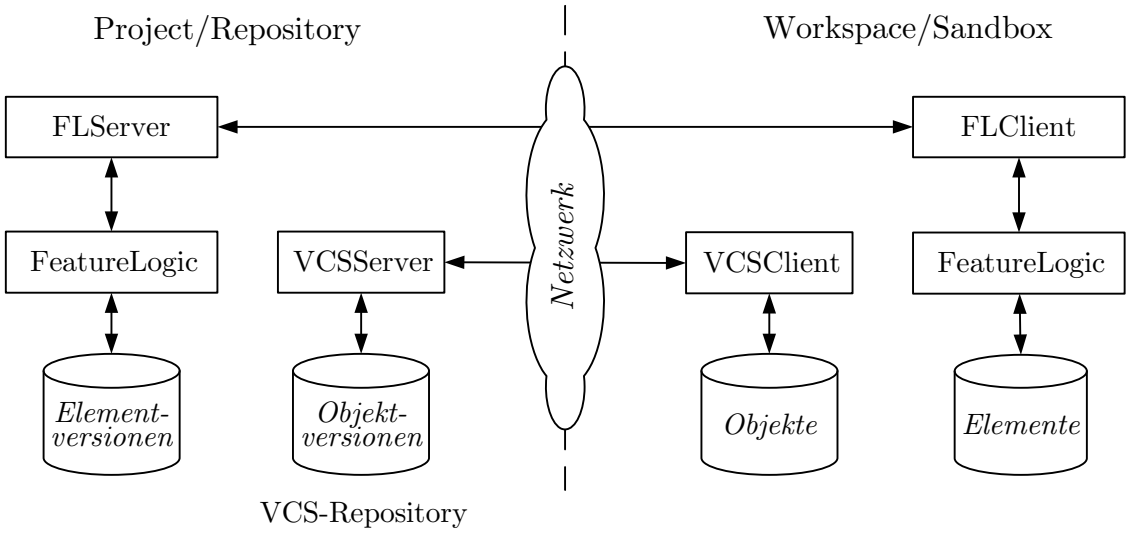


Abbildung 2.30: Systemarchitektur nach Datenströmen (Beer, 2005)

Im Folgenden werden die wesentlichen Begriffe für die mathematische Beschreibung des Datenmodells aufgeführt. Die verwendeten Bezeichner unterscheiden sich teilweise von denen, die Beer einführte, da sie mit den in dieser Arbeit verwendeten übereinstimmen sollen.

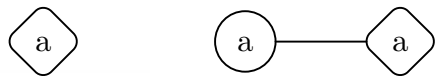
Sandbox

Element: In der Feature-Logic stellen die Elemente, die über Features verknüpft werden, die Basis dar. Beer führt zusätzlich in der Sandbox Elemente ein, die als Stellvertreter von Objekten unabhängig von den Anwendungen dienen. Elemente spiegeln den Objektzustand wider und besitzen deshalb entsprechend den Objekteigenschaften Objekt-Features sowie optional vergebene Klassen-, Anwendungs- und Nutzerfeature. Jedem Objekt ist genau ein Element zugeordnet, dessen Name der projektweit eindeutigen POID des Objekts entspricht. Die Menge E enthält alle Elemente.

$$E := \{e \mid e \text{ ist ein Element}\} \tag{2.4}$$

Ein auf der Spitze stehendes Quadrat mit abgerundeten Ecken dient als grafische Repräsentation eines Elements (Abbildung 2.31a). Die Elementzuordnung η ist eine bijektive Abbildung und ordnet einem Element $e \in E$ ein Objekt $a \in Q$ zu und wird grafisch als durchgezogene Linie dargestellt.

$$\eta : E \leftrightarrow Q := \{(e, a) \in E \times Q \mid \text{Element } e \text{ ist Objekt } a \text{ zugeordnet}\} \tag{2.5}$$



(a) Grafische Darstellung

(b) Elementzuordnung

Abbildung 2.31: Element

Teilmodell: Beer fasst Elemente, die zu Objekten eines eigenständigen Datenmodells gehören, zu einem Teilmodell zusammen. Diese Modellierung bewirkt eine Zerlegung der Elementmenge in disjunkte Teilmengen. Zwischen Objekten, deren Elemente unterschiedlicher Teilmodelle angehören, bestehen keine Objektreferenzen, so dass ein Teilmodell komplett in eine Anwendung geladen werden kann. Alle Teilmodelle sind in der Teilmodellmenge T enthalten.

$$T := \{T_i \mid T_i \text{ ist ein Teilmodell}\} \tag{2.6}$$

Ein Teilmodell T_i besteht aus einer Menge von Elementen.

$$T_i := \{e \in E \mid e \text{ ist Element des Teilmodells } T_i\} \tag{2.7}$$

Bindungen: Die Elemente eignen sich für die teilmodell- und anwendungsübergreifende Definition von Bindungen. Alle Bindungen zwischen zwei Elementen werden in der Sandbox in die Bindungsrelation B_S eingetragen.

$$B_S := \{(e, f) \in E \times E \mid e \text{ bindet } f\} \tag{2.8}$$

Ein Element e darf nicht von sich selbst abhängig sein, d. h. keine transitive Hülle von e nach e im Bindungsgraph existieren.

$$\underset{e \in E}{\forall}(e, e) \notin H(B_S) \tag{2.9}$$

Abbildung 2.32 zeigt wie Bindungen zwischen Teilmodellen definiert werden. Anstatt sie direkt im instanziierten Objektmodell zu speichern, werden Elemente aneinander gebunden und extra gespeichert.

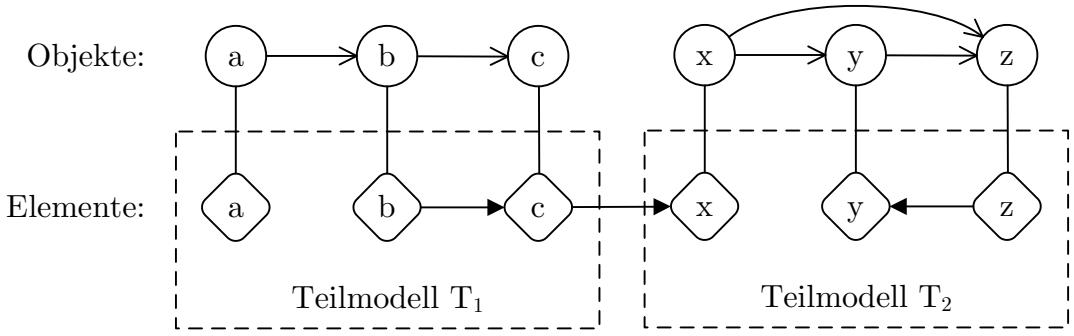


Abbildung 2.32: Definition von Bindungen

Repository

Objektversion: Das Versionsverwaltungssystem liefert nach einem Commit für jedes übertragene Objekt eine Versionsnummer zurück. Durch Anhängen der Versionsnummer an die POID entsteht der projektweit eindeutige *persistente Objektversionsidentifikator* POVID[59] für jede Objektversion $a_i \in \Omega$.

Elementversion: Analog zur Bildung von Objektversionen aus Objekten entstehen die zugehörigen Elementversionen aus den Elementen. Die Abbildung $\xi : \Xi \to E$ ordnet jeder Elementversion $e_i \in \Xi$ ein Element $e \in E$ zu und wird Elementversionsabbildung genannt. Alle Elementversionen werden im Repository in der Elementversionsmenge Ξ gespeichert.

$$\Xi := \{e_i \mid e_i \text{ ist eine Elementversion}\} \tag{2.10}$$

Die Elementversionszuordnung ε ist eine bijektive Abbildung, die einer Elementversion e_i genau eine Objektversion a_i zuordnet. Beide Versionen erhalten die gleiche POVID.

$$\begin{aligned}\varepsilon : \Xi \leftrightarrow \Omega := \{&(e_i, a_i) \in \Xi \times \Omega \mid \\ &\text{Elementversion } e_i \text{ ist Objektversion } a_i \text{ zugeordnet}\}\end{aligned} \tag{2.11}$$

Elementversionsrelation: Für die Modellierung der Versionshistorie werden nun ausschließlich Elementversionen herangezogen, die indirekt Objektversionen adressieren. Die

[59] Beer verwendete hierfür den Begriff *Persistenter Versionsidentifikator* (PVID). Dieser wird im Kapitel 4.5.2 in einem anderen Zusammenhang gebraucht.

Abbildung der Historie geschieht mit Hilfe der Elementversionsrelation V_E analog zur Objektversionsrelation V (s. Seite 41).

Teilmodellversion: Eine Teilmodellversion $T_{i,j}$ setzt sich aus Elementversionen e_i zusammen und legt damit den Zustand eines Teilmodells T_i im Repository fest.

$$T_{i,j} := \{e_i \in \Xi \mid e_i \text{ ist Elementversion der Teilmodellversion } T_{i,j}\} \subseteq \Xi \tag{2.12}$$

Alle Teilmodellversionen sind Bestandteil der Teilmodellversionsmenge Θ.

$$\Theta := \{T_{i,j} \mid T_{i,j} \text{ ist eine Teilmodellversion}\} \tag{2.13}$$

Die Teilmodellversionsabbildung $\vartheta : \Theta \to T$ ordnet jeder Teilmodellversion $T_{i,j}$ ein Teilmodell T_i zu.

$$\vartheta : \Theta \to T := \{(T_{i,j}, T_i) \in \Theta \times T \mid T_{i,j} \text{ ist eine Version des Teilmodells } T_i\} \tag{2.14}$$

Die Modellierung der Versionshistorie geschieht mit Hilfe der Teilmodellversionsrelation V_T analog zur Objektversionsrelation V.

Bindungen: Die Bindungsrelation B_R speichert alle Paare von Elementversionen, die aneinander gebunden sind.

$$B_R := \{(e_i, f_j) \in \Xi \times \Xi \mid e_i \text{ bindet } f_j\} \tag{2.15}$$

Eine Elementversion darf dabei nicht an eine Version des gleichen Elements und nicht an mehrere Versionen eines anderen Elements gebunden sein.

Freigabe: Nach Definition der Freigabe im Abschnitt 2.1.1 auf Seite 12 entspricht eine Freigabe L_i einem bestimmten Planungszustand, der einer Menge von Teilmodellversionen und der darin enthaltenen Elementversionen entspricht.

$$L_i := \{e_i \in \Xi \mid e_i \text{ gehört zur Freigabe } L_i\} \subseteq \Xi \tag{2.16}$$

Folgende Eigenschaften müssen für eine konsistente Freigabe erfüllt sein.

- Höchstens eine Version eines Elements e darf in der Freigabe L_i enthalten sein.
- Alle Elementversionen e_i die eine in der Freigabe enthaltene Elementversion e_j binden, müssen in der Freigabe L_i enthalten sein.
- Es dürfen nur vollständige Teilmodellversionen $T_{i,j}$ in der Freigabe L_i enthalten sein.

Alle Freigaben L_i sind in der Freigabemenge L enthalten.

$$L := \{L_i \mid L_i \text{ ist eine Freigabe}\} \tag{2.17}$$

Umsetzung

Zur Erklärung der Umsetzung soll nur die Datenspeicherung in der Sandbox und im Repository betrachtet werden. Die Umsetzung der Operationen kann bei (Beer, 2005) nachgeschlagen werden.

Sandbox: Der im Rahmen von *objectVCS* vorgestellte XML-Serialisierer wird in der Systemarchitektur von Beer weiterhin verwendet, um die Datenobjekte der Anwendung in XML-Dateien zu speichern. Ein textbasiertes Versionsverwaltungssystem dient dann zur Versionierung der serialisierten Teilmodelle. Die Aufteilung der Sandbox in Verzeichnisse unterliegt kaum Einschränkungen durch das VCS, so dass die in Abbildung 2.33 zu sehende hierarchische Unterteilung in der Sandbox gewählt wurde. Direkt unterhalb des Sandboxverzeichnisses erhält jedes Teilmodell ein eigenes Verzeichnis, das wiederum je ein Verzeichnis für die XML-Dateien der Objekte und Elementdateien enthält. Im Objekteverzeichnis speichert das VCS zusätzlich Daten zur Versionsverwaltung ab. Eine Elementdatei im Elementeverzeichnis setzt sich aus Zeilen mit Schlüssel-Wert-Paaren zur Speicherung der Elementeigenschaften zusammen.

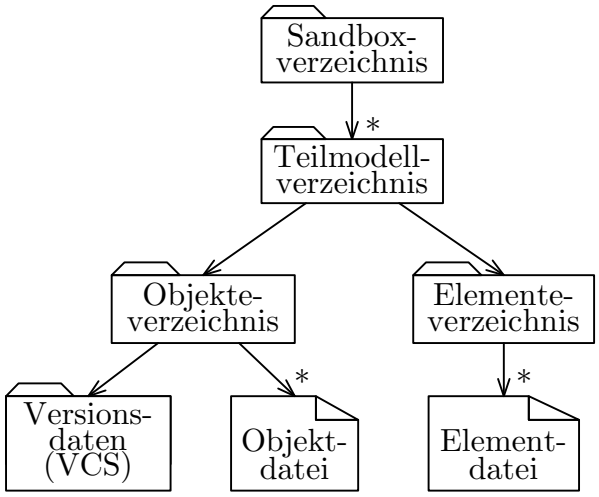


Abbildung 2.33: Umsetzung der Sandbox (Beer, 2005)

Repository: Das Repository verwendet zwei Datencontainer. Zum einen werden die Dateiversionen der Sandbox im VCS-Repository gespeichert, zum anderen die Elemente mit ihren Eigenschaften durch die Feature-Logic in einer relationalen Datenbank. Die hierarchische Struktur der Sandbox wird im VCS-Repository unverändert abgebildet, nur die Art der internen Speicherung und Verwaltung unterscheidet sich zwischen den einzelnen Versionsverwaltungssystemen.

2.4.6 Weitere Versionierungsansätze in der Forschung

Rückblick: Ab Anfang der 1980er befassten sich Forschergruppen damit, wie man CAD-Datenmodelle mit relationalen Datenbanken abbilden kann (Haskin u. a., 1982). Wenig

später erschienen dann Artikel[60], die sich mit der Versionierung von CAD-Modellen mit ihren Objekten befassten. In diesem Zusammenhang gewann auch die objektorientierte Methode an Bedeutung, die gut Objektbeziehungen und Vererbungen abbilden kann. In den Versionierungsartikeln werden typische Begriffe des Software-Configuration-Managements verwendet: Version, Konfiguration, Versionsserver, Workspace, Check-out, Check-in. Viele Ansätze verwenden Datenbanken zu Modellierung der Versionierungsbeziehungen. (Chou u. Kim, 1986) setzen die Versionierungsarchitektur mit zwei zentralen Datenbanken auf dem Server und jeweils einer privaten Datenbank auf den Clientrechnern um. Die erste Datenbank auf dem Server enthält unveränderbare, öffentliche Versionen aus allen Projekten, die für jeden CAD-Bearbeiter zugänglich sind, und die zweite enthält Daten aus einem Projekt, auf die nur die kooperierenden Bearbeiter eines Projekts zugreifen können. Des Weiteren wird ein Benachrichtigungsmechanismus für untereinander referenzierte Objektversionen entwickelt.

Versionierung von IFC-Modellen: (Nour u. a., 2006) stellen einen Versionierungsansatz für IFC-Modelle auf der Basis von *objectVCS* vor. Die von einer IFC-Anwendung gespeicherten STEP[61]-Dateien werden durch ein Java-Programm mit einem STEP-Parser eingelesen und für jedes Objekt wird eine XML-Datei in der Sandbox abgelegt. Das objectVCS-Framework kann dann diese XML-Dateien versionieren. Das Problem an diesem Konzept ist, dass nicht alle IFC-Objekte eine eindeutige GUID besitzen. Nur für Objekte, die von *IfcRoot* abgeleitet sind, trifft das zu. Die GUID eines IFC-Objekts sollte sich während der Bearbeitung nicht ändern, bei manchen Anwendungen ist das aber nicht von vornherein sichergestellt. Das Konzept der Objektversionierung für die IFC wurde in (Tulke u. a., 2008) aufgegriffen.

Versioniertes Meta-Modell mit Änderungssemantik: Aufgrund der Vielzahl verschiedener Datenmodelle im Bauwesen verwendet (Weise, 2006) ein Meta-Modell, das eine Strukturierung in Objekte, Vererbungsbeziehungen und Assoziationen unterstützt. Das Meta-Modell wird mit der Sprache EXPRESS beschrieben, die aber keine eindeutige Objektidentifikation bietet. Der Ansatz setzt auf lange Transaktionen mit sich wiederholenden, dreistufigen Planungsschritten. Das Gesamtmodell wird (1) in Teilmodelle aufgeteilt und (2) von den Planern synchron auf ihren Rechnern bearbeitet. Danach werden (3) alle Teilmodelle zu einem zeitlichen Koordinierungspunkt wieder zusammengeführt.

Für die Versionierung werden in Phase 2 die Änderungen aus den Zuständen der Objekte unter Berücksichtigung der Historie berechnet und als Vorwärtsdelta auf dem Versionierungsserver gespeichert. Die Änderungssemantik unterscheidet zwischen Erzeugen, Verändern, Löschen, Teilen, Zusammenlegen und Evolution von Datenobjekten. Die fehlende Objektidentifikation soll durch heuristische[62] Verfahren ausgeglichen werden. Diese sind bei einer hohen Anzahl von Änderungen nicht immer eindeutig. Anstatt Objekte falsch zuzuordnen, werden sie als nichtzuordenbar gekennzeichnet.

[60] s. (Chou u. Kim, 1986), (Katz u. a., 1986), (Katz, 1990)
[61] (ISO 10303-21, 2002)
[62] heurísko ⟨altgriech.., „ich finde“⟩, heuriskein ⟨altgriech.., „(auf-)finden, entdecken“⟩ → Heuristische Methoden versuchen, mit geringem Rechenaufwand akzeptable Lösungen für ein Problem zu finden.

2.5 Vergleich und Zusammenführung von Dokumenten

Die essentielle Bedeutung des Vergleichens und Zusammenführens von Dokumenten bzw. Dateien in versionierten Systemen mit einem optimistischen Zugriffsmodell wurde bereits im Abschnitt 2.3.2 auf Seite 34 herausgestellt. In den nächsten Abschnitten sollen Konzepte für verschiedene Dateiarten sowie Objektmodelle untersucht werden.

2.5.1 Textdateien

Diff-Algorithmus: Textdateien besitzen einen relativ einfachen Aufbau und bestehen aus Zeilen, die sich wiederum aus Zeichen zusammensetzen. Der von (Hunt u. McIllroy, 1976) vorgestellte und bis heute kaum veränderte Diff-Algorithmus findet Änderungen durch Lösung des „Longest common subsequence problem“. Das bedeutet, dass beim zeilenweisen Vergleichen die größten Bereiche mit Übereinstimmungen in beiden Dateien gesucht werden.

Nutzerunterstützung: Ausgereifte grafische Benutzeroberflächen helfen dem Anwender, die gefundenen Änderungen schnell zu erfassen. Abbildung 2.34 zeigt das Fenster zum Vergleichen von Textdateien der integrierten Entwicklungsumgebung (IDE)[63] Eclipse[64]. Die weißen Bereiche enthalten unveränderte Daten und die grau hinterlegten neue, veränderte oder gelöschte Daten. Die Leiste am rechten Rand zeigt in einer Schnellübersicht die Stellen mit den Änderungen und kann zur Navigation innerhalb der verglichenen Dateien genutzt werden.

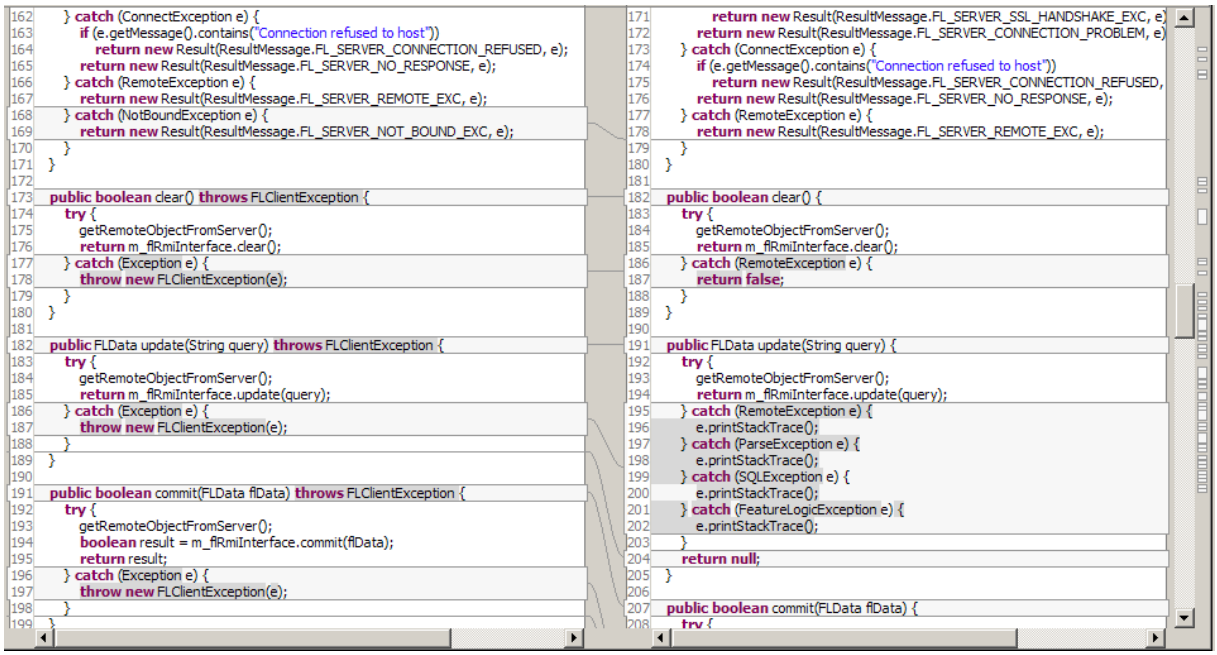


Abbildung 2.34: Textdateivergleich in der IDE Eclipse

[63] IDE = Integrated Development Environment
[64] http://www.eclipse.org/

Merge: Geänderte Bereiche, die sich nicht überschneiden, werden von den Programmen meist automatisch übernommen. Der Nutzer muss nur dann eingreifen, wenn an gleichen Stellen in beiden Dateien Änderungen vorgenommen wurden und für das Merge-Werkzeug ein Konflikt besteht. Das Vergleichsfenster aus Abbildung 2.34 wird für den Konfliktfall in Eclipse um Schaltflächen zum teilweisen oder kompletten Übernehmen der Änderungen vom Server erweitert. Außerdem kann die lokale Datei in diesem Fenster editiert werden.

2.5.2 Plotdateien

Plotdatei: Eine Plotdatei dient unter Verwendung einer Seitenbeschreibungssprache zur Ansteuerung von Stiftplottern. Die bekannteste Seitenbeschreibungssprache ist die Hewlett Packard Graphic Language (HPGL), die auch von anderen Plotter- und Druckerherstellern verwendet wird. Die häufigsten genutzten Kommandos sind: *Wähle Stift, Senke Stift, Bewege Stift relativ* und *Bewege Stift absolut*. Neben Linie und Punkt sind auch andere geometrische Formen Bestandteil der Sprache: Bogen, Kreis, Rechteck. Mit dem Aufkommen von Tintenstrahlplottern war es notwendig, die Version HP-GL/2 zu veröffentlichen, die verschiedene Strichstärken unterstützt.

Austauschformat: Wegen der Universalität der Kommandos hatte sich HP-GL als Datenaustauschformat für Vektorgrafik etabliert, so dass viele CAD-Programme eine Ex- und/oder Importfunktion für HPGL-Dateien besitzen. Jedoch lässt sich aus den Plotdateien nicht wieder der originale Zustand des CAD-Modells wiederherstellen, da beim Export die komplexen CAD-Objekte in geometrische Primitive zerlegt werden und Zeichnungsstrukturen, wie z. B. Layer, verloren gehen. Ein Zusammenführen von CAD-Dokumenten ist über diesen Weg damit hinfällig.

PlanDiffViewer: Das Programm PlanDiffViewer[65] der Firma WeltWeitBau vergleicht zwei HPGL-Plotdateien miteinander, indem es sie im Ausgabefenster übereinanderlegt und hinzugefügte, geänderte und gelöschte Elemente farbig hervorhebt. Verglichen mit früheren Arbeitsweisen ohne Computer entspricht das dem Übereinanderlegen von Plänen aus Transparentpapier. Das Problem bei Plotdateien besteht in der fehlenden Identifikation der grafischen Elemente. So wird zum Beispiel das Löschen einer Linie und das Hinzufügen einer neuen Linien mit gleichen Abmessungen nicht als Änderung erkannt. In der aktuellen Version unterstützt PlanDiffViewer den Vergleich von Plänen im STEP-CDS-Format sowie SVG-Vektorgrafiken. Scalable Vector Graphics (SVG) ist ein Standard zur Beschreibung von Vektorgrafiken mit XML und wurde vom W3C ursprünglich für das World Wide Web (WWW)[66] entwickelt (W3C, 2003).

[65] `http://www.wwbau.de/de/PlanNet/PlanDiffViewer/PlanDiffViewer.html`
s. auch (Ramunno, 2005)

[66] Das World Wide Web ist ein Hypertext-System für das Internet, das am CERN in Genf entworfen und 1993 veröffentlicht wurde.

2.5.3 Anwendungsdokumente

Allgemein: Anwendungsdokumente liegen oft in Binärform vor und enthalten ein Datenmodell der Anwendung in persistenter Form. Es nicht sinnvoll, die Dateien direkt zu vergleichen, sondern sie einzulesen und ein transientes Modell zu instanziieren. Für den Vergleich müssen alle relevanten Daten berücksichtigt und sollten dem Benutzer in der gewohnten oder in einer nachvollziehbaren Form dargestellt werden. Da Anwendungen zur Bearbeitung verschiedener Aufgabenbereiche eingesetzt werden, unterscheidet sich die Struktur und Präsentation der Daten in den Anwendungen.

CAD-Dokumente: CAD-Dokumente enthalten nicht nur Zeichnungskomponenten mit verschiedenen Linienattributen, sondern auch Informationen zur Blattaufteilung, Layer, Zellen, Symbole, Schraffuren und externe Referenzen. Assoziative Bemaßungen und Schraffuren können an vorhandene Zeichnungskomponenten geknüpft werden und passen sich bei Änderung von diesen automatisch an.

Vergleich von DWG-Dateien: Das Produkt CompareDWG 2007 der Firma furix[67] ist ein ObjectARX[68]-Modul zur Erweiterung von AutoCAD um den Vergleich von zwei DWG-Dateien. Im Basismodus wird nur die Geometrie unter Vernachlässigung der Objektidentifikatoren entsprechend der Funktionsweise des PlanDiffViewers mit den bekannten Nachteilen verglichen. Im detailierten Modus bezieht CompareDWG die Objektidentifikatoren mit ein, die in AutoCAD als Handle bezeichnet werden und nur innerhalb einer Zeichnung eindeutig sind. Es ist nicht auszuschließen, dass sie sich – beispielsweise durch das Zusammenlegen von Zeichnungen – ändern. Außerdem wird beim Vergleichen die interne Repräsentation der Zeichnung mit berücksichtigt. Jedoch bewirkt schon das Ändern der Zeichnungsskalierung, dass alle Objekte fälschlicherweise als geändert angezeigt werden. CompareDWG erzeugt als Zwischenergebnis sechs temporäre Dateien mit verschiedenen Objektzuständen aus Zeichnung 1 und 2 sowie zwei textuelle Ergebnisberichte. Abschließend kann sich der Nutzer eine kombinierte Zeichnung mit ausgewählten und farblich gekennzeichneten Objektzuständen erstellen lassen.

2.5.4 Objektmodelle

Nichtinstanziierte Objektmodelle: In (Richter, 2005) wurde für objectVCS (s. Abschnitt 2.4.4 auf Seite 49) zuerst ein Vergleich und Zusammenführen von Objektmodellen auf Basis der serialisierten XML-Dateien entwickelt. Während des Vergleichsprozesses wird im Konfliktfall ein externes Textvergleichswerkzeug[69] mit den zwei XML-Dateien des betroffenen Objekts aufgerufen und die Unterschiede dort grafisch dargestellt (s. Abbildung 2.35). Der Merge muss danach durch einen manuellen Eingriff des Nutzers in einem Texteditor durchgeführt werden. Diese Vorgehensweise ist sehr umständlich, fehleranfällig und nur als Studie für Entwickler gedacht. Durch das direkte Ändern der Objektattribute kann zudem die Konsistenz des Objektmodells nicht sichergestellt werden.

[67] http://www.furix.com/
[68] AutoCAD Runtime Extension
[69] ExamDiff der Firma prestoSoft. http://www.prestosoft.com/edp_examdiff.asp

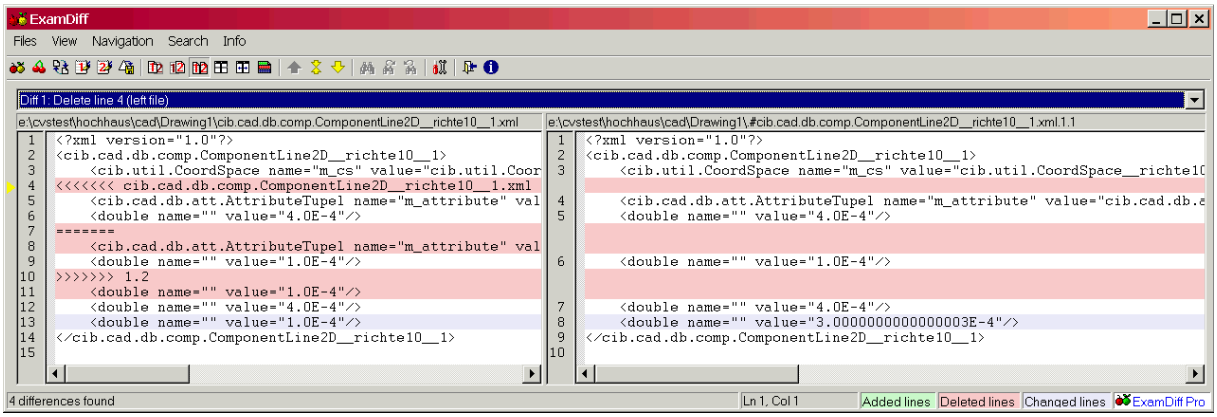


Abbildung 2.35: XML-Datei-Vergleich zweier Zustände eines serialisierten Objekts

Ansatz für beliebige instanziierte Objektmodelle: Für dieses Verfahren müssen beide Objektmodelle transient im Arbeitsspeicher vorliegen und deshalb vorher deserialisiert werden. In (Richter, 2005) wurde eine Umsetzung für beliebige Objektmodelle vorgestellt, die das gleiche Klassenschema benutzen und persistente Objektidentifikatoren bereitstellen. Bei den Objektattributen ist zum einen zwischen ein- und mehrwertigen und zum anderen zwischen primitiven Attributen und Objektreferenzen zu unterscheiden. Ausgehend von einem Wurzelobjekt wird durch Reflexion das Objektmodell traversiert und in einem Objektgraph abgebildet. Die Merge-Operation wird durch eine grafische Benutzeroberfläche unterstützt, die im oberen Bereich den Weg vom Wurzelobjekt zum betrachteten Objekt und im unteren Bereich den Zustand des Objekts im ersten und zweiten Objektmodell anzeigt. Abbildung 2.36 zeigt den Dialog für den Merge einer Map. Durch Selektieren von Objekten in der linken und mittleren Liste werden diese in das resultierende Objektmodell übernommen. Auch dieser Ansatz ist in dieser Form nicht praxistauglich und fehleranfällig, da der Bezug zur Nutzersicht auf die Modelle fehlt und durch das direkte Manipulieren der Objekte leicht Inkonsistenzen entstehen.

Anwendungsbezogener Diff und Merge: (Schäfer, 2006) geht der Frage nach, wie das Vergleichen und Zusammenführen von Ingenieurmodellen durch anwendungsspezifische Werkzeuge mit Kenntnissen über das Modell implementiert werden kann. Dazu definiert er die Schnittstelle *Differentiator*, die für jede zu vergleichende Objektklasse extern in einer eigenen Klasse implementiert werden muss. Das bestehende Klassenmodell muss so nicht erweitert werden. Die Differentiator-Klassen wissen, wie ein Vergleich zweier Objekte vorzunehmen ist. Dazu werden Methoden der Anwendung zur Bestimmung der Objektzustände benutzt, ohne auf die Reflexion zurückgreifen zu müssen. Die Verwendung von POIDs garantiert ein eindeutiges Ergebnis.

Die Anwendung des Konzepts wurde am Beispiel von CAD für der Anwendung CADEMIA (s. Abschnitt 5.2 auf Seite 174) implementiert und überprüft. Integraler Bestandteil ist ein Merge-Dialog (s. Abbildung 2.37), der links die POIDs der Zeichnungskomponenten und rechts zwei Grafikbereiche mit den CAD-Modellen in Realansicht angezeigt. Der untere Grafikbereich zeigt das Modell auf dem Server und der obere das aktuelle

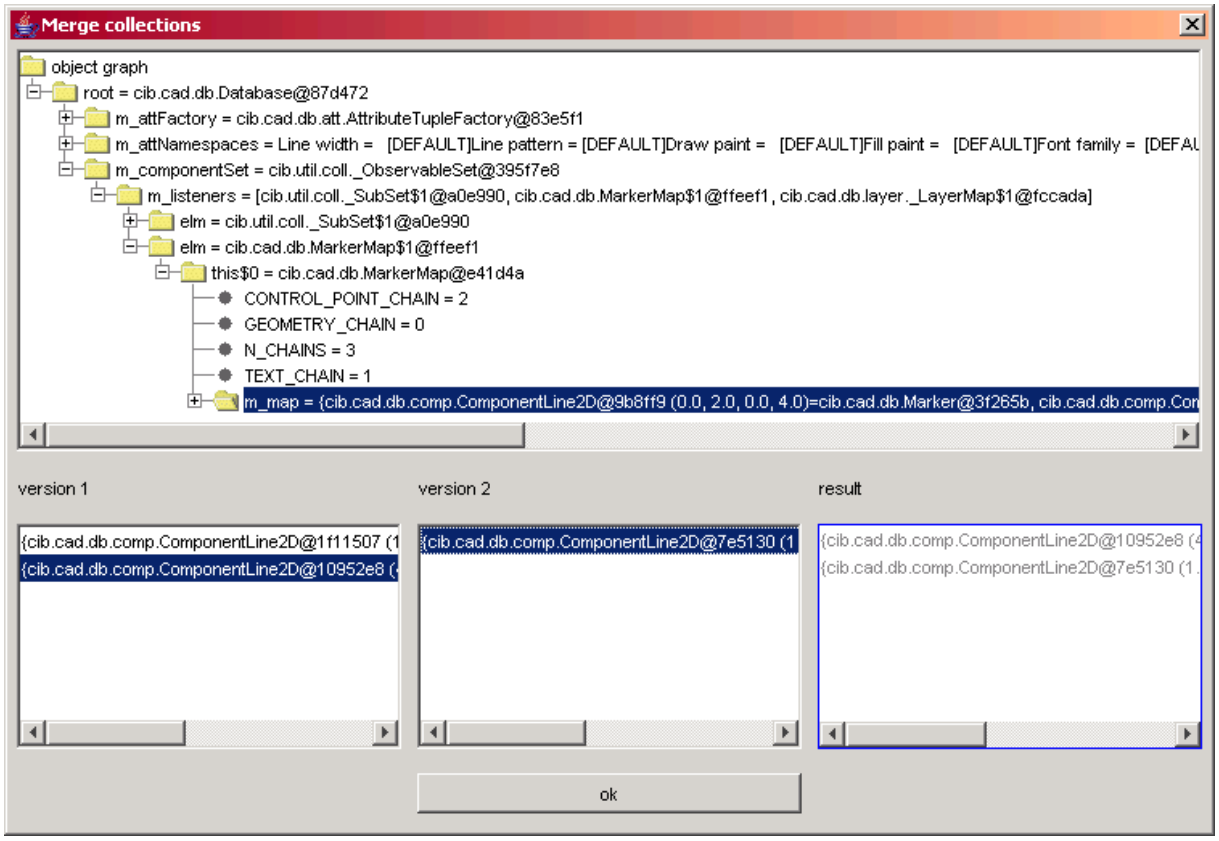


Abbildung 2.36: Dialog zum Merge einer Map eines instanziierten Objektmodells

Modell in der Sandbox. Die Grafikbereiche benutzen weiterhin Farben, um die verschiedenen Objektzustände hervorzuheben: schwarz = ungeändert, pink = geändert, grün = hinzugefügt, rot = gelöscht. Durch Filter auf der linken Seite lassen sich die angezeigten Objekte je nach ihrem Zustand von der Anzeige entfernen. Der Merge-Vorgang geschieht schrittweise durch das Anklicken und Ziehen[70] einzelner Zeichnungskomponenten in die geöffnete Anwendung, die nicht Bestandteil der Abbildung ist. Den betroffenen Attributen konfliktbehafteter Komponenten lassen sich über ein Eigenschaftsfenster Werte zuordnen, wobei zum Setzen der Attribute Methoden der Anwendung aufgerufen und nicht direkt die Objektattribute geändert werden.

Der vorgestellte Ansatz stellt gegenüber den anderen einen wesentlichen Fortschritt dar. Er integriert sich nahtlos in die Anwendung und präsentiert dem Nutzer die zu vereinigenden Modelle in der gewohnten Form. Konsistenzprobleme werden vermieden, da die Objekte nicht direkt über Reflexion, sondern über die bereitgestellten Methoden der Anwendung geändert werden. Dafür ist ein höherer Implementierungsaufwand in Kauf zu nehmen, um die Objektverwaltung in der Anwendung zu berücksichtigen. Eine geeignete grafische Benutzeroberfläche beschleunigt den Merge-Prozess und führt zu Ergebnissen mit weniger Fehlern.

[70] Drag and Drop ⟨engl., „Ziehen und Fallenlassen“⟩

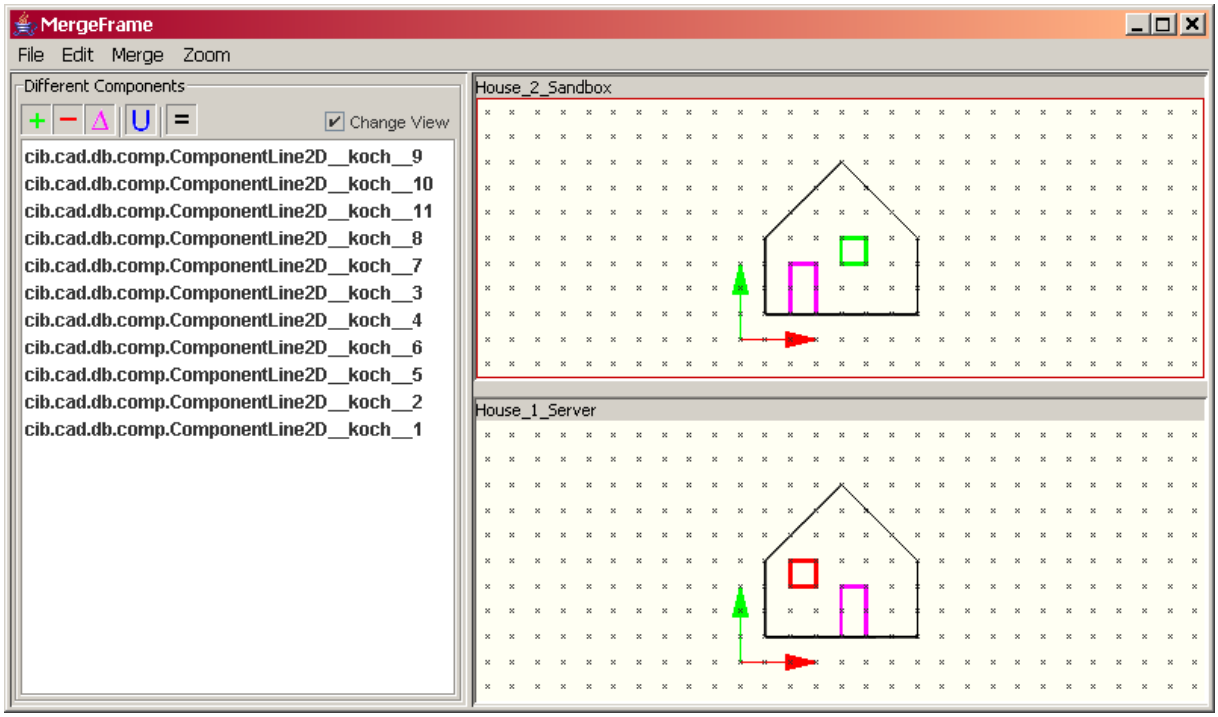


Abbildung 2.37: Dialog eines anwendungsspezifischen Merges im CAD-Bereich

2.6 Benutzerschnittstellen

2.6.1 Grundbegriffe

Normen: Die weltweit gültige und vom DIN Deutsches Institut für Normung e.V. (DIN) unverändert übernommene Norm DIN EN ISO 6385:2004-05 (ISO 6385, 2004) befasst sich allgemein mit der ergonomischen Gestaltung von Arbeitssystemen. Nach der Einführung von Begriffen beschreibt sie Gestaltungsgrundsätze sowie das Vorgehen für Entwurf, Realisierung und Validierung. Hauptziel ist es, die Leistung und Effizienz des Arbeitssystems einschließlich der Arbeitenden ohne nachteilige Wirkungen für deren Gesundheit, Wohlbefinden oder Sicherheit zu optimieren. Für die Gestaltung von Bildschirmarbeitsplätzen wird auf die Normenreihe DIN EN ISO 9241 „Ergonomie der Mensch-System-Interaktion“ verwiesen, die sich in 17 Teile gliedert und Anforderungen sowohl an Hardware als auch an Software aufführt.

Arbeitssystem (ISO 6385, 2004)**:** *„System, welches das Zusammenwirken eines einzelnen oder mehrerer Arbeitender/Benutzer mit den Arbeitsmitteln umfasst, um die Funktion des Systems innerhalb des Arbeitsraumes und der Arbeitsumgebung unter den durch die Arbeitsaufgaben vorgegebenen Bedingungen zu erfüllen.“*

Arbeitsmittel (ISO 6385, 2004)**:** *„Werkzeuge, einschließlich Hardware und Software, Maschinen, Fahrzeuge, Geräte, Möbel, Einrichtungen und andere im Arbeitssystem benutzte (System-)Komponenten.“*

Arbeitender/Benutzer (ISO 6385, 2004)**:** *„Person, die innerhalb des Arbeitssystems eine oder mehrere Arbeitsaufgaben durchführt.“*

Benutzerschnittstelle: Ein Mensch-Maschine-System liegt vor, wenn eine Aufgabe gemeinsam von einer menschlichen und einer technischen Komponente erledigt wird. Eine Benutzerschnittstelle steht in diesem System gleichwertig für den Begriff Mensch-Maschine-Schnittstelle (MMS) und dient dem Ablesen von Zustandsinformationen oder dem Bedienen der Maschine.

Ergonomie (ISO 6385, 2004)**:** *„Die Ergonomie*[71] *ist eine wissenschaftliche Disziplin, die sich mit dem Verständnis der Wechselwirkungen zwischen menschlichen und anderen Elementen eines Systems befasst, und der Berufszweig, der Theorie, Prinzipien, Daten und Methoden auf die Gestaltung von Arbeitssystemen anwendet mit dem Ziel, das Wohlbefinden des Menschen und die Leistung des Gesamtsystems zu optimieren.“*

Menschliche Informationsverarbeitung: Da für die ergonomische Gestaltung von Arbeitssystemen der Mensch im Mittelpunkt steht, ist die Kenntnis seiner Informationsverarbeitung erforderlich. (Dahm, 2006) fasst diese anschaulich in einem Blockschaltbild zusammen (s. Abbildung 2.38), welches vereinfachend auf das Sehen beschränkt bleibt. Von den fünf menschlichen Sinnen Sehen, Hören, Riechen, Schmecken und Tasten fallen auf das Sehen 80 % und auf das Hören 15 % der verarbeiteten Informationen. Nach (Dahm, 2006) ergeben alle Sinne zusammen einen Informationsstrom von geschätzten 8 MBit/s.

Jeder Sinn wird ausgehend durch einen Umgebungsreiz von einem oder mehreren Sensoren wahrgenommen und in Nervenimpulse umgewandelt. Nach Durchlaufen eines Filters werden diese mit Hilfe der Informationen aus dem Kurz- und Langzeitgedächtnis erkannt und interpretiert. Dieser Prozess der Wahrnehmung wird in der Wissenschaft als Perzeption bezeichnet. Da durch den Menschen in erster Linie das Sehen zur Informationsgewinnung Verwendung findet, sind die Kenntnisse über die Fähigkeiten und Einschränkungen dieses Sinnes bei der Gestaltung von Software-Nutzeroberflächen von Bedeutung.

In der nächsten Stufe, der kognitiven[72] Verarbeitung, erkennt das Gehirn Dinge, Vorgänge und Beziehungen. Der Mensch verfügt über vielfältige kognitive Fähigkeiten, wie z. B. Aufmerksamkeit, Erinnerung, Kreativität, Orientieren, Planen, Vorstellungskraft und Wille. Die Kognitionswissenschaft erforscht diese Fähigkeiten als interdisziplinäres Fach. Auch hier spielen das Kurz- und Langzeitgedächtnis eine wichtige Rolle. Das Kurzzeitgedächtnis ist analog zum Arbeitsspeicher im Computer sehr schnell, besitzt aber nur eine sehr geringe Kapazität von 7+/-2 sinntragenden Einheiten. Dies gilt beim Entwurf von Benutzerschnittstellen zu beachten, indem z. B. innerhalb einer logischen Gruppe nicht zu viele Elemente angeordnet werden. Das Langzeitgedächtnis ist der permanente Wissensspeicher des Menschen mit einer sehr langen Speicherdauer und fast unbegrenzter Kapazität. Die Informationen werden durch Ausbildung von Verbindungen zwischen den Neuronen des Gehirns gespeichert. Das Langzeitgedächtnis lässt sich in zwei Arten einteilen:

[71] ergon ⟨griech., „Arbeit, Werk“⟩ + nomos ⟨griech., „Gesetz, Regel“⟩
[72] cognoscere ⟨lat., „erkennen, erfahren, kennen lernen“⟩

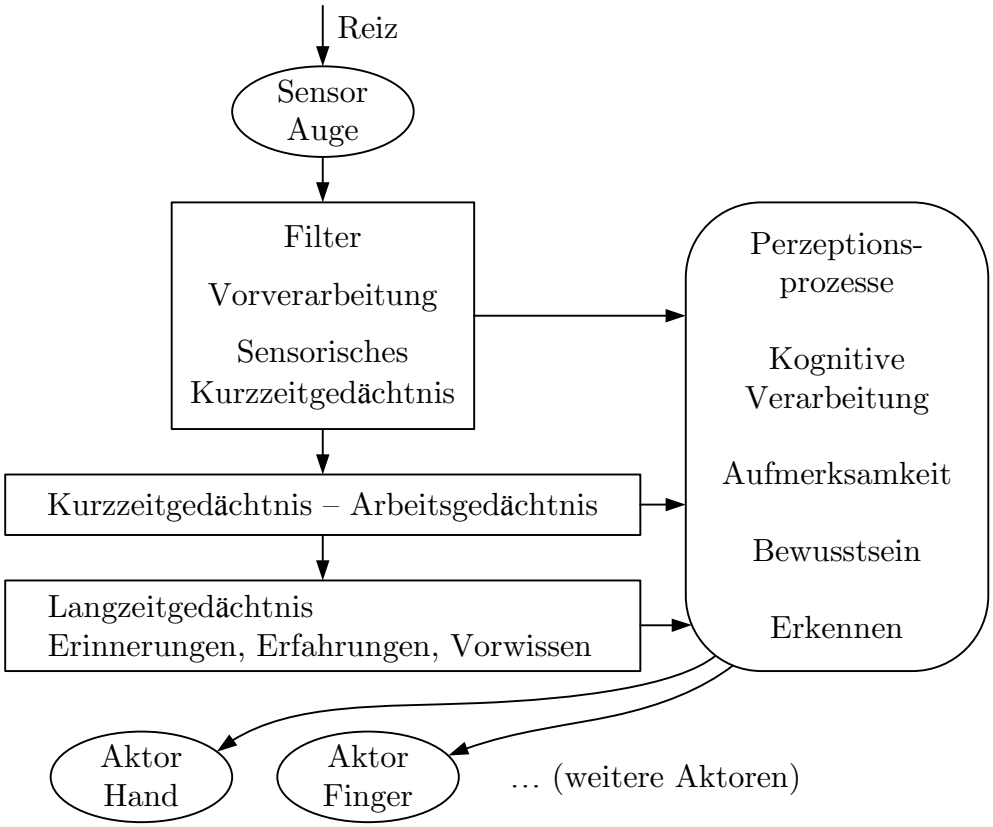


Abbildung 2.38: Blockschaltbild der menschlichen Informationsverarbeitung nach (Dahm, 2006)

- **Deklaratives Gedächtnis**: Speichert Fakten, Daten und Konzepte sowie Bilder und Vergleiche. Wissen ist die Gesamtheit aus Daten und den Verknüpfungen zwischen ihnen.
- **Produktionen-Gedächtnis**: Speichert Fähigkeiten und Abläufe, die das Gedächtnis später automatisiert ablaufen lassen kann.

Am Ende der Wahrnehmungs- und kognitiven Verarbeitungskette stehen Handlungen. Voraussetzung für eine erfolgreiche Ausführung ist die Aufmerksamkeit gerade bei Handlungen, die unser volles Bewusstsein erfordern. In der Regel kann nur eine Handlung bewusst ausgeführt werden. Je nach Grad des nötigen Bewusstseins lassen sich die Handlungen nach (Hacker, 1986) in drei Ebenen einteilen. Dadurch ist es auch möglich, mehrere Handlungen gleichzeitig auszuführen.

- Bewusste Handlungen
- Routinehandlungen
- Vollständig automatisierte Handlungen

(Norman, 2001) stellt ein Modell von Handlungsschritten auf, die bis zum Erreichen eines Ziels durchgeführt werden. Dieses lässt sich auf alle Arten von Handlungen anwenden. Durch Veränderung ein oder mehrerer Stellgrößen wird Einfluss auf eine Steuergröße genommen, die sich durch die Aktionen dem Zielwert nähert oder von ihm entfernt. Während

der Ausführung erfolgt durch die Sensoren eine Rückkopplung, die durch die kognitive Verarbeitung in eine Änderung der Handlung münden kann. Somit tritt ein Regelungsprozess in Gang.

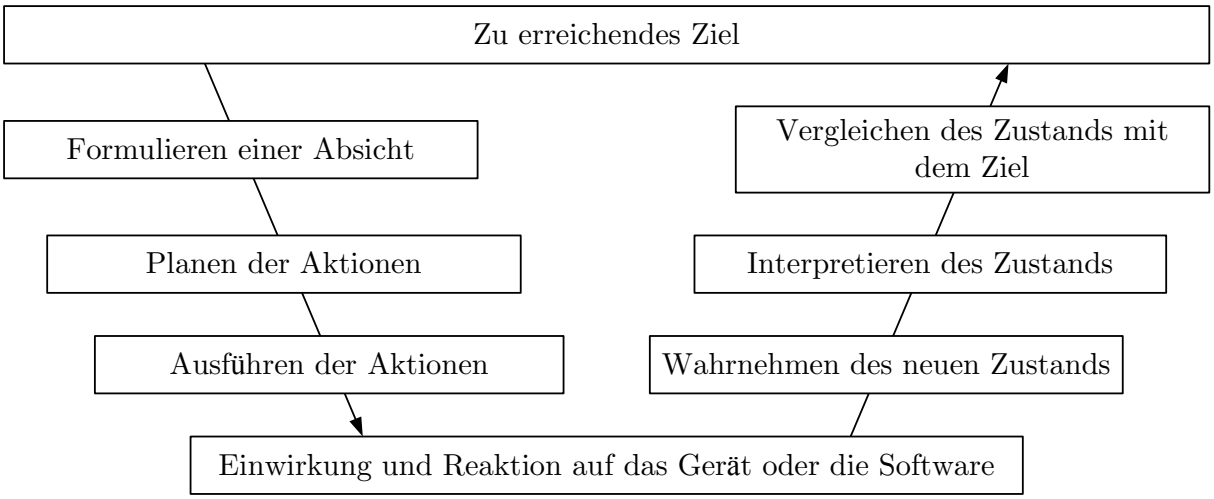


Abbildung 2.39: Handlungsschritte nach (Norman, 2001)

2.6.2 Software-Ergonomie

Mensch-Computer-Interaktion: Im Bereich der Informatik wird das Wort Maschine durch Computer ersetzt, woraus sich das Wort Mensch-Computer-System ergibt. Die Wechselwirkung zwischen Mensch und Computer wird folglich als Mensch-Computer-Interaktion, engl. Human-Computer Interaction (HCI), bezeichnet.

Vergleich: (Herczeg, 2005) stellt Computer als moderne Werkzeuge den traditionellen Werkzeugen hinsichtlich der Eigenschaften gegenüber und fasst das Ergebnis in einer Tabelle zusammen (s. Tabelle 2.7). Computerbasierte Werkzeuge scheinen danach zwar vorteilhafter zu sein, jedoch besitzen sie durch die hohe Komplexität eine mangelnde Bedienbarkeit. Bedingt durch die langen Entwicklungsprozesse wurden traditionelle Werkzeuge schrittweise an die menschlichen Bedürfnisse angepasst und weisen dadurch eine gute Bedienbarkeit bei eingeschränkter Vielseitigkeit auf.

Traditionelles Werkzeug	Computerbasiertes Werkzeug
vorgegebene starre Form	gestaltbare und dynamische Form
passiv	aktiv und reaktiv
erklärungsbedürftig	selbsterklärend
kleine überschaubare Funktionalität	große und komplexe Funktionalität
optimiert in der Bedienung	optimiert in der Funktionalität

Tabelle 2.7: Eigenschaften traditioneller und computerbasiertes Werkzeuge nach (Herczeg, 2005)

Gebrauchstauglichkeit: (ISO 9241-11, 1999) führt zur Bewertung der Software-Ergonomie den Begriff Gebrauchstauglichkeit (engl. usability) ein, der den unscharfen Begriff *Benutzerfreundlichkeit* ersetzt. Die Norm definiert drei Maße zur Bestimmung der Gebrauchstauglichkeit.

- **Effektivität** ist *„die Genauigkeit und Vollständigkeit, mit der Benutzer ein bestimmtes Ziel erreichen.“*
- **Effizienz** ist *„der im Verhältnis zur Genauigkeit und Vollständigkeit eingesetzte Aufwand, mit dem Benutzer ein bestimmtes Ziel erreichen.“*
- **Zufriedenstellung** ist *„die Freiheit von Beeinträchtigungen und positive Einstellungen gegenüber der Nutzung des Produkts.“*

Um die Gebrauchstauglichkeit eines Produkts oder einer Software bestimmen zu können, müssen angestrebte Ziele vorher festgelegt und die oben genannten Maße weiter zerlegt werden. Jedem Ziel wird so ein Teilmaß der Effektivität, Effizienz und Zufriedenstellung zugeordnet. Außerdem sind Nutzungskontexte wie Benutzer, Arbeitsaufgaben, Arbeitsmittel (Hardware, Software) und Umgebung näher zu beschreiben. Durch den Vergleich der definierten Anforderungen mit der tatsächlich erreichten Gebrauchstauglichkeit soll des Produkt schrittweise überarbeitet und verbessert werden.

DIN EN ISO 9241-110: Teil 110 der Normenreihe ISO 9241 befasst sich ausschließlich mit der Gestaltung von Dialogen und gibt eine Reihe von Empfehlungen. Unterstützt wird er von weiteren Teilen und anderen Quellen, die in Tabelle 2.8 aufgeführt sind.

ISO 9241-110, Grundsätze der Dialoggestaltung		
ISO 9241-12, Informationsdarstellung		
Benutzerführung: ISO 9241-13	Dialogführung: ISO 9241-14, Menüs ISO 9241-15, Kommandosprachen ISO 9241-16, Direkte Manipulation ISO 9241-17, Bildschirmformulare	Weitere Quellen: • andere Normen • wiss. Literatur • Hersteller-richtlinien • Unternehmens-richtlinien

Tabelle 2.8: Teile der ISO 9241 und andere Quellen für Gestaltungsempfehlungen nach (ISO 9241-110, 2008)

Auch Teil 110 definiert neue Begriffe, wobei die wichtigsten in den nächsten drei Absätzen näher erklärt sind.

Interaktives System (ISO 9241-110, 2008)**:** *„Kombination von Hardware- und Softwarekomponenten, die Eingaben von einem Benutzer empfangen und Ausgaben zu einem Benutzer übermitteln, um ihn bei der Ausführung einer Arbeitsaufgabe zu unterstützen.“*

Dialog (ISO 9241-110, 2008)**:** *„Interaktion zwischen einem Benutzer und einem interaktiven System in Form einer Folge von Handlungen des Benutzers (Eingaben) und Antworten des interaktiven Systems (Ausgaben), um ein Ziel zu erreichen.“*

Benutzungsschnittstelle (ISO 9241-110, 2008)**:** *„Alle Bestandteile eines interaktiven Systems (Software oder Hardware), die Informationen und Steuerelemente zur Verfügung stellen, die für den Benutzer notwendig sind, um eine bestimmte Arbeitsaufgabe mit dem interaktiven System zu erledigen.“*

Andere Begriffe, die im Software-Bereich bedeutungsgleich verwendet werden, sind Benutzerschnittstelle und Benutzeroberfläche. Der englische Begriff dafür ist *User interface*, abgekürzt durch UI.

Kriterien: (ISO 9241-110, 2008) gibt Empfehlungen zur Dialoggestaltung anhand verschiedener Kriterien, die hier genannt und nur mit einer treffenden Aussage aus der Norm näher erläutert werden. Weitere Informationen sind in der Norm nachzuschlagen.

- **Aufgabenangemessenheit**: *„Ein interaktives System ist aufgabenangemessen, wenn es den Benutzer unterstützt, seine Arbeitsaufgabe zu erledigen, d. h., wenn Funktionalität und Dialog auf den charakteristischen Eigenschaften der Arbeitsaufgabe basieren, anstatt auf der zur Aufgabenerledigung eingesetzten Technologie.“*
- **Selbstbeschreibungsfähigkeit**: *„Ein Dialog ist in dem Maße selbstbeschreibungsfähig, in dem für den Benutzer zu jeder Zeit offensichtlich ist, in welchem Dialog, an welcher Stelle im Dialog er sich befindet, welche Handlungen unternommen werden können und wie diese ausgeführt werden können.“*
- **Erwartungskonformität**: *„Ein Dialog ist erwartungskonform, wenn er den aus dem Nutzungskontext heraus vorhersehbaren Benutzerbelangen sowie allgemein anerkannten Konventionen entspricht.“*
- **Lernförderlichkeit**: *„Ein Dialog ist lernförderlich, wenn er den Benutzer beim Erlernen der Nutzung des interaktiven Systems unterstützt und anleitet.“*
- **Steuerbarkeit**: *„Ein Dialog ist steuerbar, wenn der Benutzer in der Lage ist, den Dialogablauf zu starten sowie seine Richtung und Geschwindigkeit zu beeinflussen, bis das Ziel erreicht ist.“*
- **Fehlertoleranz**: *„Ein Dialog ist fehlertolerant, wenn das beabsichtigte Arbeitsergebnis trotz erkennbar fehlerhafter Eingaben entweder mit keinem oder mit minimalem Korrekturaufwand seitens des Benutzers erreicht werden kann.“*
- **Individualisierbarkeit**: *„Ein Dialog ist individualisierbar, wenn Benutzer die Mensch-System-Interaktion und die Darstellung von Informationen ändern können, um diese an ihre individuellen Fähigkeiten und Bedürfnisse anzupassen.“*

Erreicht werden die Kriterien unter anderem durch folgende charakteristische Eigenschaften nach (ISO 9241-12, 2000).

- **Klarheit:** *„Der Informationsgehalt wird schnell und genau vermittelt.“*

- **Unterscheidbarkeit:** *„Die angezeigte Information kann genau unterschieden werden.“*
- **Kompaktheit:** *„Den Benutzern wird nur jene Information gegeben, die für das Erledigen der Aufgabe notwendig ist.“*
- **Konsistenz:** *„Gleiche Information wird innerhalb der Anwendung entsprechend den Erwartungen des Benutzers stets auf gleiche Art dargestellt.“*
- **Erkennbarkeit:** *„Die Aufmerksamkeit des Benutzers wird zur benötigten Information gelenkt.“*
- **Lesbarkeit:** *„Die Information ist leicht zu lesen.“*
- **Verständlichkeit:** *„Die Bedeutung ist leicht verständlich, eindeutig, interpretierbar und erkennbar.“*

(Herczeg, 2005) listet noch weitere Kriterien auf, die je nach Anwendungssituation gültig sind. Diese Liste ist, wie der Autor betont, nicht vollständig.

- Übersichtlichkeit
- Direktheit
- Einbezogenheit
- Bediensicherheit
- Wahrnehmbarkeit
- Natürlichkeit

Arten von Benutzerschnittstellen:

- **Kommandozeile** (engl. Command Line Interface (CLI)): Die Kommandozeile war die erste Form der Benutzerschnittstelle für Computer und ist heute immer noch in Anwendungen und vor allem in Betriebssystemen vorzufinden. Der Benutzer muss zur Steuerung von Software Befehle in die Kommandozeile eingeben, die dann von einem Kommandozeileninterpreter verarbeitet werden. Andere Begriffe für diese Form der Benutzerschnittstelle sind, wenn eine Textausgabe integriert ist, Terminal oder Shell[73].
- **Zeichenorientierte Benutzerschnittstelle** (engl. Text User Interface (TUI)): Diese Art der Benutzerschnittstelle ist zwar auch textbasiert, bedarf aber keiner Befehlseingabe durch den Nutzer. Dafür existieren Menüs und Dialoge, die mit Tastenkürzeln und teilweise mit der Maus bedient werden können. Die Gestaltung ist wegen der maximal möglichen 256 Zeichen des verwendeten Zeichensatzes und wegen des vorhandenen Rasters von meist 80 mal 25 Zeichen eingeschränkt. Ein bekannter Vertreter dieser Benutzerschnittstelle ist der 1986 veröffentlichte Norton Commander von John Socha.

[73] shell ⟨engl., „Hülle, Außenhaut“⟩

- **Grafische Benutzeroberfläche** (engl. Graphical User Interface (GUI)): Der erste Rechner mit einer grafischen Benutzeroberfläche war der 1973 am Xerox PARC[74] entwickelte Xerox Alto. Voraussetzungen dafür waren die Verwendung einer Rastergrafik für das freie Zeichnen von grafischen Elementen und die Maus als Eingabegerät zur zweidimensionalen Steuerung des Mauszeigers. Durch das Betätigen der Tasten oder des Mausrades werden Aktionen in der Software ausgelöst. Für die Darstellung stehen unter anderem die folgenden grafischen Elemente und Steuerelemente zur Verfügung.

  - Fenster
  - Menü
  - Symbolleiste
  - Icon
  - Bildlaufleiste
  - Tabelle
  - Baum
  - Dialogfenster
  - Registerkarten
  - Gruppe
  - Schaltfläche (Button)
  - Beschriftung (Label)
  - Checkbox (Auswahlkästchen)
  - Radiobutton (Optionsfeld)
  - Listenfeld (Auswahlliste)
  - Textbox
  - Combobox (Kombinationslistenfeld)
  - Schieberegler (Slider)

  Grafische Benutzeroberflächen benötigen zwar eine höhere Rechenleistung und Speicherkapazität als die vorgenannten zwei Arten, haben diese aber weitgehend abgelöst.

- **Sprachbasierte Benutzerschnittstelle** (engl. Voice User Interface (VUI)): Die Kommunikation zwischen Nutzer und System erfolgt hauptsächlich über Sprache. Dafür ist zum einen eine Spracherkennung zum Deuten der Nutzereingaben und zum anderen eine Ausgabe durch aufgezeichneten Ton oder durch eine synthetische Sprache, die aus Texten erzeugt wird, notwendig.

2.6.3 Grafische Benutzeroberflächen mit Java

AWT: Aufgrund der verwendeten Programmiersprache Java für die spätere Umsetzung fiel die Wahl auf die im JDK integrierte Grafikbibliothek Swing. Sie setzt auf dem älteren Abstract Window Toolkit (AWT) auf, das zur Darstellung des GUIs die grafischen Elemente des Betriebssytems heranzieht. Dadurch entsprechen AWT-Anwendung zwar dem Erscheinungsbild des Betriebssystems, aber die verwendbaren Elemente beschränken sich auf den kleinsten gemeinsamen Nenner zwischen allen Systemen. AWT-Komponenten heißen schwergewichtig, da sie Ressourcen des Betriebssystems referenzieren.

[74] Xerox PARC = Xerox Palo Alto Research Center

Swing: Im Gegensatz dazu zeichnet Swing alle grafischen Komponenten selbst und erreicht auf diese Weise die Plattformunabhängigkeit. Anfangs hatte es durch die Performanceschwächen einen schlechten Ruf, was sich aber durch Verbesserungen an der Klassenbibliothek und durch die Verfügbarkeit schnellerer Hardware gebessert hat. Ein großer Vorteil von Swing-Anwendungen ist, dass sie durch Verwendung leichtgewichtiger Komponenten auf jeder Plattform – zumindest theoretisch – gleich funktionieren. Die Anpassung an die Optik verschiedener Betriebssysteme erfolgt durch das Setzen sogenannter Look-and-Feels. Swing ist modular aufgebaut und eignet sich sehr gut für die Entwicklung flexibler GUIs.

Grafische Elemente in Swing

Hauptfenster: Das Hauptfenster einer GUI mit Swing repräsentiert die Klasse *JFrame*. Ihm kann durch die Klassen *JMenuBar* und *JToolBar* ein Menü und eine Symbol- bzw. Werkzeugleiste hinzugefügt werden. Grafische Komponenten können nicht direkt auf dem Hauptfenster platziert werden, dazu dienen Container. *JFrame* enthält selbst den Container *content pane*, der für alle Kindkomponenten außer dem Menü zuständig ist. Die Methode *add()* fügt diese automatisch der *content pane* hinzu.

Dialoge: Dialoge sind Fenster, die für eine bestimmte Aufgabe entworfen wurden. Ist ein Dialog modal, muss er erst geschlossen werden, bevor zu anderen Fenstern oder Dialogen gewechselt werden kann. Bei nicht-modalen Dialogen ist dies auch ohne Beenden möglich. Swing stellt einige Standarddialoge bereit, wie z. B. einen Farbauswahl- und einen Dateiauswahldialog. Für die Erstellung eigener Dialoge steht die Klasse *JDialog* zur Verfügung.

Container: Container können die im nächsten Absatz beschriebenen Komponenten und selbst wiederum Container aufnehmen. In Swing existieren vier Containertypen.

- **JPanel:** Die Klasse *JPanel* ist der Standardcontainer in Swing, die durch zuweisbare Layoutmanager eine verschiedene Anordnung der enthaltenen Kindelemente ermöglicht. Wahlweise ist Doppelpufferung (engl. double buffering) einschaltbar, bei der alle Zeichenoperationen auf ein Hintergrundbild angewendet werden, das zu einem bestimmten Zeitpunkt im Vordergrund angezeigt wird (Ullenboom, 2008). Damit wird ein flüssigerer Bildaufbau erreicht.
- **JScrollPane:** Wenn Swing-Komponenten, wie oftmals Bäume und Tabellen, in der Darstellung größer als der Bildschirm sind, lassen sie sich in einer *JScrollPane* platzieren. Die Klasse zeigt bei Bedarf automatisch eine horizontale und vertikale Bildlaufleiste an.
- **JTabbedPane:** Mit dieser Klasse können Komponenten thematisch in Registerkarten bzw. Reitern angeordnet werden. Dies ist vor allem sinnvoll, wenn zu viele Komponenten in einem Fenster die Bedienbarkeit beeinträchtigen.
- **JSplitPane:** Der letzte Container dient dazu, zwei Swing-Komponenten durch einen horizontalen oder vertikalen Teiler, der zudem verschiebbar ist, zu trennen. Durch

die Schachtelung von *JSplitPane*-Objekten können auch mehr als zwei Bereiche erstellt werden.

Komponenten: Komponenten sind diejenigen GUI-Elemente, die für die vorrangige Interaktion mit dem Nutzer zuständig sind. Swing-Komponenten sind alle von der abstrakten Klasse *JComponent* abgeleitet, die eine umfangreiche Basisfunktionalität bereitstellt. Dazu gehören: Größe, Position, Tooltips, Rahmen, Fokus, Navigation, Ereignisse. Die Klasse ist von so zentraler Bedeutung, dass sich sogar die Container-Klassen von ihr ableiten.

Für jede Komponente steht in Swing eine eigene Klasse zur Verfügung, die alle in Tabelle 2.9 aufgeführt sind.

GUI-Komponente	Klasse
Gruppe	JPanel, ButtonGroup (für Radiobuttons)
Schaltfläche	JButton
Beschriftung	JLabel
Checkbox	JCheckBox
Radiobutton	JRadioButton
Listenfeld	JList
Textbox (einzeilig)	JTextField, JFormattedTextField, JPasswordField
Textbox (mehrzeilig)	JTextArea
Editorkomponente	JEditorPane
Drehfeld	JSpinner
Combobox	JComboBox
Schieberegler	JSlider
Fortschrittsbalken	JProgressBar

Tabelle 2.9: Klassen für Swing-Komponenten

JComponent-Objekte generieren von sich aus Ereignisse (Events) für verschiedene Situationen. Zum Beispiel wird ein *KeyEvent* bei einem Tastendruck erzeugt, das durch einen spezialisierten *KeyListener* an andere Java-Objekte, die sich an der Komponente registriert haben, weitergeleitet wird. Eine andere Variante ist das direkte Reagieren auf Ereignisse durch das Schreiben einer anonymen inneren Klasse, deren Objekt mit der Methode *addXXXListener()* der Komponente zugewiesen wird.

Listing 2.4 erklärt an einem Minimalbeispiel die generelle Funktionsweise von Swing-Anwendungen. Die Klasse *SwingExample* leitet sich von der Klasse *JFrame* ab und erbt somit deren Methoden. Alle Einstellung werden direkt im Konstruktor vorgenommen. Auf das Hauptfenster sollen ein mehrzeiliges Textfeld zur Eingabe beliebiger Zeichen und eine Schaltfläche zum Beenden des Programms angeordnet werden. Zeilen 5 bis 17 erzeugen ein JTextArea-Objekt der Größe 400 mal 300 Pixel und weisen ihm ein KeyListener-Objekt zur Reaktion auf Tastaturereignisse zu. Bei Drücken einer Taste (keyPressed) wird der

Tastencode und bei Loslassen (keyTyped) das eingebene Zeichen in die Konsole geschrieben. Das Textfeld enthält nur die Zeichen. Analog dazu erzeugen die Zeilen 18 bis 25 eine Schaltfläche, die beim Betätigen das Programm beendet. Zusätzlich werden Mausklicks auf die Schaltfläche mit einem *MouseListener* überwacht (Zeile 26-35). Die restlichen Zeilen des Konstruktors ordnen die zwei Komponenten mit Hilfe des Layoutmanagers *BorderLayout* auf dem Hauptfenster an und blenden es auf dem Bildschirm ein.

```
1  public class SwingExample extends JFrame{
2      public SwingExample(){
3          setDefaultCloseOperation(EXIT_ON_CLOSE);
4
5          JTextArea textArea = new JTextArea();
6          textArea.setPreferredSize(new Dimension(400,300));
7          textArea.addKeyListener(new KeyListener(){
8              public void keyTyped(KeyEvent e){
9                  System.out.println("Eingegebenes Zeichen: " +
10                     e.getKeyChar() );
11             }
12             public void keyPressed(KeyEvent e){
13                 System.out.println("Gedrueckte Taste: " +
14                     e.getKeyCode() );
15             }
16             public void keyReleased(KeyEvent e){}
17         });
18         JButton button = new JButton("Beenden");
19         button.addKeyListener(new KeyListener(){
20             public void keyTyped(KeyEvent e){
21                 System.exit(0);
22             }
23             public void keyPressed(KeyEvent e){}
24             public void keyReleased(KeyEvent e){}
25         });
26         button.addMouseListener(new MouseListener(){
27             public void mouseClicked(MouseEvent e){
28                 System.out.println("Maus geklickt.");
29                 System.exit(0);
30             }
31             public void mouseEntered(MouseEvent e){}
32             public void mouseExited(MouseEvent e){}
33             public void mousePressed(MouseEvent e){}
34             public void mouseReleased(MouseEvent e){}
35         });
36         setLayout(new BorderLayout());
37         getContentPane().add(textArea, BorderLayout.CENTER);
38         getContentPane().add(button, BorderLayout.SOUTH);
39         pack();
40         setVisible(true);
```

```
41      }
42
43      public static void main(String[] args){
44          new SwingExample();
45      }
46  }
```

Listing 2.4: Minimalbeispiel einer Swing-Anwendung

Abbildung 2.40 zeigt das Programm nach der Ausführung und der Eingabe des Wortes `Swing` in das Textfeld. Die Konsolenausgabe ist in Listing 2.5 enthalten. Der Tastencode 16 steht für die Umschalttaste und die anderen für die jeweilige Buchstabentaste. Im letzten Schritt wurde das Programm mit einem Mausklick auf die Schaltfläche *Beenden* geschlossen.

Abbildung 2.40: Screenshot: Minimalbeispiel einer Swing-Anwendung

```
1  Gedrueckte Taste: 16
2  Gedrueckte Taste: 83
3  Eingegebenes Zeichen: S
4  Gedrueckte Taste: 87
5  Eingegebenes Zeichen: w
6  Gedrueckte Taste: 73
7  Eingegebenes Zeichen: i
8  Gedrueckte Taste: 78
9  Eingegebenes Zeichen: n
10 Gedrueckte Taste: 71
11 Eingegebenes Zeichen: g
12 Maus geklickt.
```

Listing 2.5: Konsolenausgabe des Swing-Beispiels

Bäume: Zur Darstellung von Bäumen steht die Swing-Klasse *JTree* zur Verfügung. Die Klasse *DefaultMutableTreeNode* beschreibt die Eigenschaften eines Knotens und stellt Methoden zur Abfrage und Manipulation bereit, die teilweise von der Schnittstelle *TreeNode* übernommen und implementiert wurden. Wenn Änderungen am Baum vorgenommen werden sollen, bietet sich die Nutzung eines *TreeModels* an, das entsprechende Ereignisse generieren kann. Bäume mit vielen Knoten sollten in eine *JScrollPane* gesetzt werden.

Tabellen: Swing behandelt die Präsentation/Steuerung und das Modell von Tabellen in zwei unterschiedlichen Klassen und wendet damit das Architekturmuster Model-View-Controller (MVC)[75] an. Für ersteres ist die Klasse *JTable* verantwortlich und für zweiteres die Schnittstelle *TableModel*. JTable bietet die Anzeige der Daten in Zeilen und Spalten, Selektion, Tastaturnavigation und das Ändern der Spaltenbreiten. TableModel schreibt Methoden zur Manipulation und Zustandsabfrage des zugrundeliegenden Tabellenmodells vor. Die abstrakte Klasse *AbstractTableModel* nimmt schon viele Implementierungen vor, so dass eine eigene Klasse für das Modell davon abgeleitet werden kann. Eine Sortierung der Tabellenspalten übernimmt die Klasse *TableRowSorter*, der das Tabellenmodell übergeben und dem JTable-Objekt über die Methode *setRowSorter()* zugewiesen wird.

[75] Model-View-Controller ⟨engl., „Modell/Präsentation/Steuerung“⟩ → Das MVC-Konzept wurde zuerst von (Reenskaug, 1979) erwähnt, der damals am Xerox PARC arbeitete.

3 Formale Beschreibung der Grundlagen

Zur Erforschung der Wahrheit bedarf es notwendig der Methode.

(René Descartes, 1628)

3.1 Erweiterung des mathematischen Modells

3.1.1 Allgemein

Mengen: Für Mengen, die unversionierte Elemente enthalten, existieren unter dem gleichen Bezeichner zwei Varianten, eine in der Sandbox und eine im Repository. Wenn die Menge nur Elemente einer Sandbox enthält, wird der Mengenbezeichner mit einem Überstrich gekennzeichnet. Wenn die Menge im Repository gemeint ist, entfällt der Überstrich. Der Sachverhalt wird im Beispiel 3.1 vertieft.

Beispiel 3.1: Unterschied zwischen Sandbox und Repository am Beispiel der Objekte

Die Menge $\overline{Q}$ enthält alle Objekte, die sich in einer Sandbox befinden. Der Zustand eines Objekts wird im abgespeicherten Zustand durch seine Attribute bzw. Eigenschaften wiedergegeben, die atomar und nicht atomar sein können.

In der Abbildung 3.1 befindet sich zum Zeitpunkt 1 ausschließlich das Objekt a mit dem atomaren Attribut 5.0 in der Sandbox des Nutzers A. Beim Übertragen des Objekts in das Repository wird eine neue Objektversion a_1, die den Zustand von a übernimmt, angelegt und in die Objektversionsmenge Ω eingetragen. Das Objekt a seinerseits wird zur Menge Q des Repositorys hinzugefügt und besitzt keine Attribute mehr. Die Beziehung zwischen Objektversion und Objekt wird durch die Objektversionsabbildung $\omega : \Omega \to Q$ hergestellt.

Zum Zeitpunkt 2 hat Nutzer A den Wert des Attributs von 5.0 auf 6.0 geändert und überträgt das Objekt in das Repository. Dort wird eine neue Version a_2 angelegt, die das Attribut übernimmt, und ferner das Paar (a_2, a) in ω eingetragen. Gleichzeitig hat Nutzer B das Objekt b mit dem Zeichenfolgenattribut vom Wert "xyz" erzeugt und als b_1 eingetragen (committet). Die Menge Q enthält nun die Elemente $\{a, b\}$, während die

Menge $\overline{Q}$ in Sandbox A nur $\{a\}$ und $\overline{Q}$ in Sandbox B nur $\{b\}$ enthält. Die Menge Ω besteht aus den Objekttversionen $\{a_1, a_2, b_1\}$.

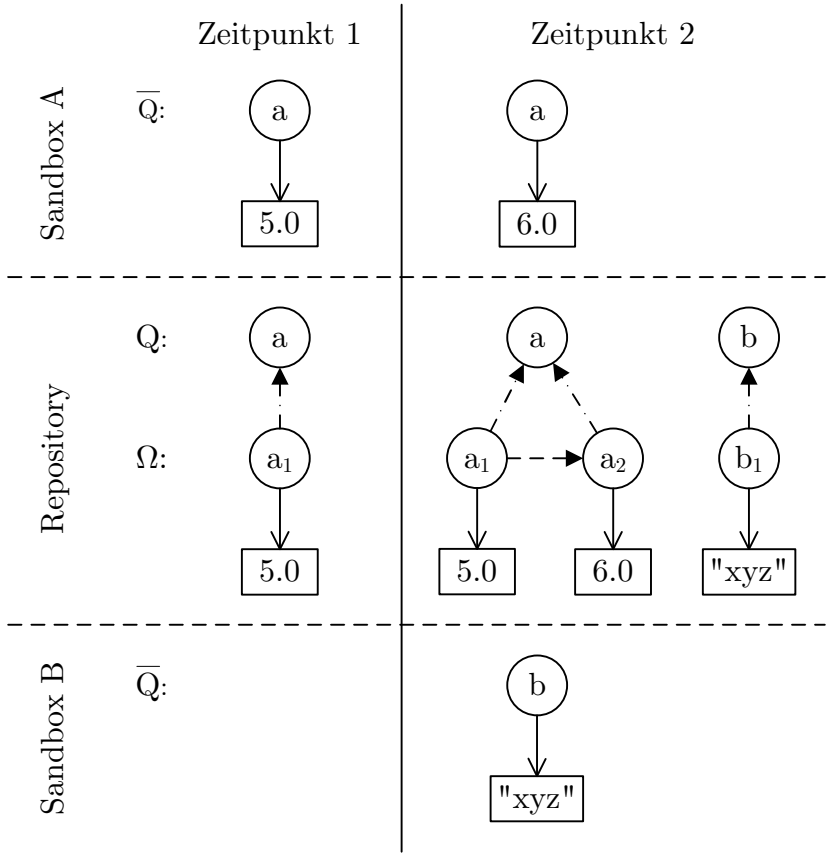


Abbildung 3.1: Objekte und Objektversionen in Sandbox und Repository

3.1.2 Unversioniert

Dokument: Im Abschnitt 2.1.1 auf Seite 9 wurde die Bedeutung des Dokuments für die technische Dokumentation herausgestellt. Ebenso besteht die gewohnte Arbeitsweise vieler Planer am Rechner in der Erzeugung und Bearbeitung von digitalen Dokumenten verschiedenster Art. Jedes Dokument enthält eine Teilmenge von Informationen über das zu planende Bauwerk und wird mit einer zugehörigen Fachanwendung bearbeitet. Jede Anwendung verwendet ein natives Format für die Speicherung der Informationen, das meistens für die Objektversionierung ungeeignet ist. Deshalb wird das Dokument einer verteilten Anwendung aus einer Menge von Objekten des Datenmodells und den zugehörigen Objektelementen zusammengesetzt.

Somit enthält ein Dokument D_i die zwei disjunkten Teilmengen $D_i^O \subseteq Q$ und $D_i^E \subseteq E$, die als Dokumentobjekt- und Dokumentelementmenge bezeichnet werden. Jedes Element $e \in E$ besitzt ein über die bijektive Abbildung η zugeordnetes Objekt $a \in Q$ mit $\eta(e) = a$. Der von (Beer, 2005) verwendete und auf Seite 53 beschriebene Begriff *Teilmodell* wird

des Weiteren nicht mehr verwendet und geht indirekt über die Menge D_i^E im Dokument auf.

$$D_i := D_i^O \cup D_i^E \tag{3.1}$$

$$D_i^O := \{a \in Q \mid a \text{ ist Objekt des Dokuments } D_i\} \cup Q \tag{3.2}$$

$$D_i^E := \{e \in E \mid e \text{ ist Element des Dokuments } D_i\} \cup E \tag{3.3}$$

mit

$$\underset{a \in D_i^O}{\forall} \underset{e \in D_i^E}{\exists} \eta(e) = a \tag{3.4}$$

Abbildung 3.2 zeigt exemplarisch die Modellierung eines Dokuments D_1, das sowohl die Objekte a, b als auch die zugehörigen und gleichnamigen Elemente a, b enthält.

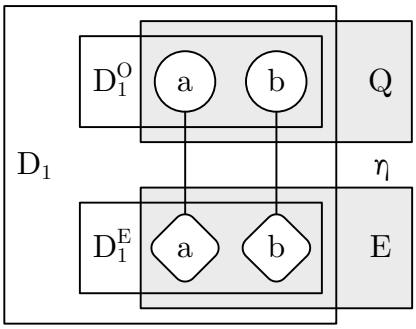


Abbildung 3.2: Mathematische Modellierung des Dokuments

Die Dokumentmenge D enthält alle Dokumente D_i.

$$D := \{D_i \mid D_i \text{ ist ein Dokument}\} \tag{3.5}$$

Anwendung: Mit einer Anwendung A_i werden Dokumente D_i bearbeitet. Die Anwendungsmenge A enthält alle Anwendungen, die in allen Projekten verwendet werden.

$$A := \{A_i \mid A_i \text{ ist eine Anwendung}\} \tag{3.6}$$

Anwendungsabbildung: Mit einer Anwendung A_i können verschiedene Dokumente D_i bearbeitet werden. Die Anwendungsabbildung α ordnet jedem Dokument eine Anwendung zu.

$$\alpha : D \to A := \{(D_i, A_i) \in D \times A \mid D_i \text{ wird mit } A_i \text{ bearbeitet}\} \tag{3.7}$$

Projekt: Ein Projekt P_i enthält eine bestimmte Anzahl von Dokumenten eines in sich geschlossenen Bauprojekts. Ein Dokument D_i gehört zu genau einem Projekt P_i.

$$P_i := \{D_i \in D \mid \text{Das Dokument } D_i \text{ ist Element des Projekts } P_i\} \tag{3.8}$$

Die Projektmenge P enthält alle Projekte P_i.

$$P := \{P_i \mid P_i \text{ ist ein Projekt}\} \tag{3.9}$$

Beispiel 3.2: Mengentheoretische Modellierung in der Sandbox

Das folgende Abbildung zeigt die mengentheoretische Modellierung für drei Dokumente in der Sandbox. Das Dokument D_1 beinhaltet die Objekte a, b und c; Dokument D_2 die Objekte d und e; Dokument D_3 die Objekte f und g. Das Dokument D_1 wurde mit Anwendung A_1 und das Dokument D_2 mit Anwendung A_2 innerhalb des Projekts P_1 bearbeitet. Das zweite Projekt P_2 enthält nur das Dokument D_3, welches ebenso mit Anwendung A_2 bearbeitet wurde. Die Großbuchstaben auf der linken Seite bezeichnen jeweils eine Menge innerhalb der Sandbox.

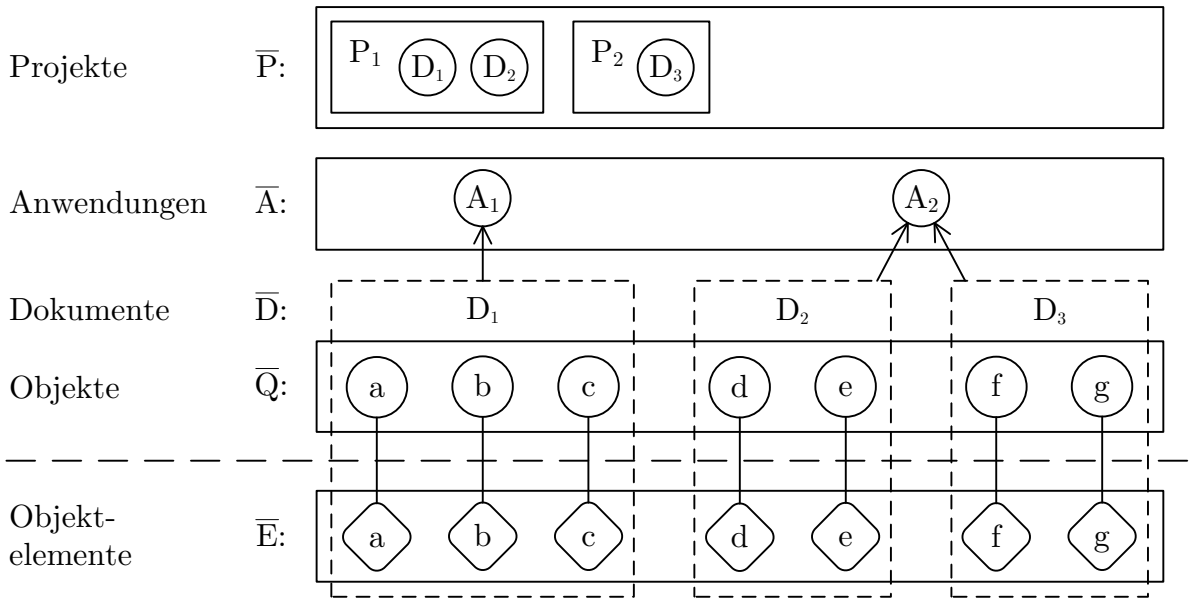


Abbildung 3.3: Beispiel für die Modellierung in der Sandbox

Bindungen: Die von Beer getroffenen Festlegungen für Bindungen gelten weiterhin, mit der Ausnahme, dass Bindungen nur zwischen Elementen *eines* Projekts definiert werden dürfen.

$$\underset{e \in D_i}{\forall} \underset{f \in D_j}{\forall} (e, f) \in B_S \Rightarrow D_i \in P_i \wedge D_j \in P_i \tag{3.10}$$

3.1.3 Versioniert

Dokumentversion: Eine Änderung an einem Dokument führt zu einem neuen Dokumentzustand in der Sandbox, der den Commit als neue Dokumentversion $D_{i,j}$ in das Repository zulässt. Eine Dokumentänderung kann zwei Ursachen haben:

- Die Änderung des Zustands eines Objekts a, das Teil des Dokuments D_i ist.
- Das Hinzufügen oder Löschen einer Bindung $(e, f) \in B_S$, die ein Element f des Dokuments D_i als gebundenes Element enthält.

Die Menge Γ speichert alle Dokumentversionen $D_{i,j}$.

$$\Gamma := \{D_{i,j} \mid D_{i,j} \text{ ist eine Dokumentversion}\} \tag{3.11}$$

Dokumentversionsabbildung: Die Dokumentversionsabbildung γ ordnet jeder Dokumentversion $D_{i,j}$ das zugehörige Dokument D_i zu.

$$\gamma : \Gamma \to D := \{(D_{i,j}, D_i) \in \Gamma \times D \mid D_{i,j} \text{ ist eine Version des Dokuments } D_i\} \tag{3.12}$$

Virtuelle Versionen: Virtuelle Versionen δ_i markieren die Erzeugung oder Löschung einer Version im Versionsgraphen. (Firmenich, 2002) verwendete sie für Objekte und Beer für Elemente. Generell ist es ausreichend, sie allgemein für Versionen zu definieren.

$$\Delta := \{\delta_i \mid \delta_i \text{ ist eine virtuelle Version}\} \tag{3.13}$$

Versionsrelation: Die Versionsrelation V stellt allgemein Beziehungen zwischen Versionen in einem Versionsgraphen dar, was die Erzeugung, Änderung und Löschung beinhaltet. Für eine beliebige Menge W von Versionen lässt sich die Versionsrelation mit der Menge $W_\Delta := W \cup \Delta$ wie folgt formulieren.

$$V := \{(w_i, w_j) \in W_\Delta \times W_\Delta \mid \text{Version } w_j \text{ ist Nachfolger von Version } w_i\} \tag{3.14}$$

Die Versionsrelation V teilt sich in drei disjunkte Mengen auf.

$$V = V_N \cup V_A \cup V_L \tag{3.15}$$

$$V_N := \{(\delta_i, w_j) \in \Delta \times W \mid w_j \text{ ist die erste Version}\} \tag{3.16}$$

$$V_A := \{(w_i, w_j) \in W \times W \mid w_j \text{ ist die Nachfolgeversion von } w_i\} \tag{3.17}$$

$$V_L := \{(w_i, \delta_j) \in W \times \Delta \mid \text{ Der Nachfolger von } w_i \text{ wurde gelöscht}\} \tag{3.18}$$

Im Repository entstehen Versionen aus Objekten, Elementen und Dokumenten. Dementsprechend gibt es für jede Menge eine eigene Versionsrelation, namentlich mit V_O, V_E und V_D gekennzeichnet.

Freigabe: Eine Freigabe L_i setzt sich – im Gegensatz zur Definition auf Seite 55 – aus Dokumentversionen zusammen, die einen Projektstand repräsentieren. Die Freigabemenge L fasst alle Freigaben zusammen.

$$L_i := \{D_{i,j} \in \Gamma \mid D_{i,j} \text{ gehört zur Freigabe } L_i\} \subseteq \Gamma \tag{3.19}$$

$$L := \{L_i \mid L_i \text{ ist eine Freigabe}\} \tag{3.20}$$

Die Bedingungen für konsistente Freigaben ändern sich durch den Bezug auf Dokumentversionen.

- Höchstens eine Version des Dokuments D_i ist in der Freigabe L_i enthalten.

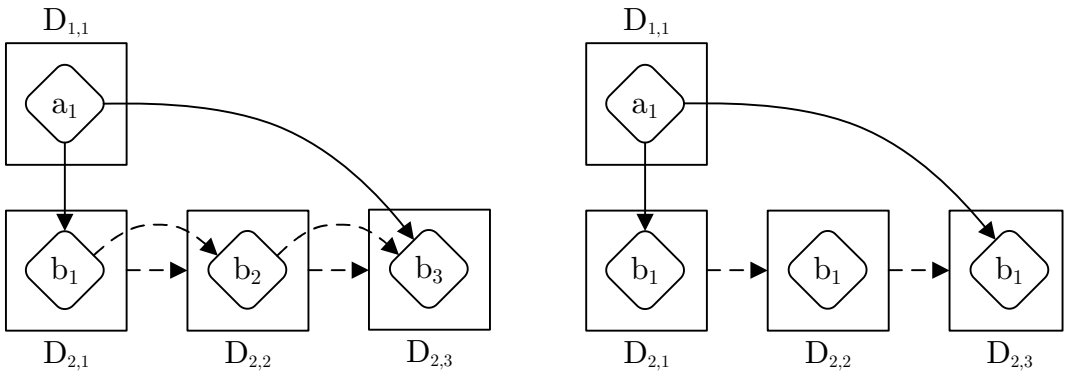
$$\underset{D_{i,j} \in L_i}{\forall} \underset{D_{k,l} \in L_i}{\forall} \gamma(D_{i,j}) \neq \gamma(D_{k,l}) \tag{3.21}$$

- Ist eine Dokumentversion $D_{k,l}$ mit einer gebundenen Elementversion f_j enthalten, muss die Dokumentversion $D_{i,j}$, die die bindende Elementversion e_i enthält, in der Freigabe L_i enthalten sein.

$$\underset{D_{i,j} \in \Gamma}{\forall} \underset{D_{k,l} \in L_i}{\forall} \underset{e_i \in D_{i,j}}{\exists} \underset{f_j \in D_{k,l}}{\exists} (e_i, f_j) \in B_R \Rightarrow D_{i,j} \in L_i \tag{3.22}$$

3.1.4 Bindungen

Problem: Bei (Firmenich, 2002; Beer, 2005) speichert die Bindungsrelation B_R Paare von Elementversionen. Eine Bindung kann nur entfernt werden, wenn eine Nachfolgeversion des gebundenen Elements erzeugt wird und die Bindung zum bindenden Element ausgetragen wird. Für ein erneutes Eintragen muss wiederum eine Nachfolgeversion des gebundenen Elements hinzugefügt werden (s. Abbildung 3.4a).



(a) Modellierung nach (Firmenich, 2002; Beer, 2005)

(b) Gewünschte Modellierung

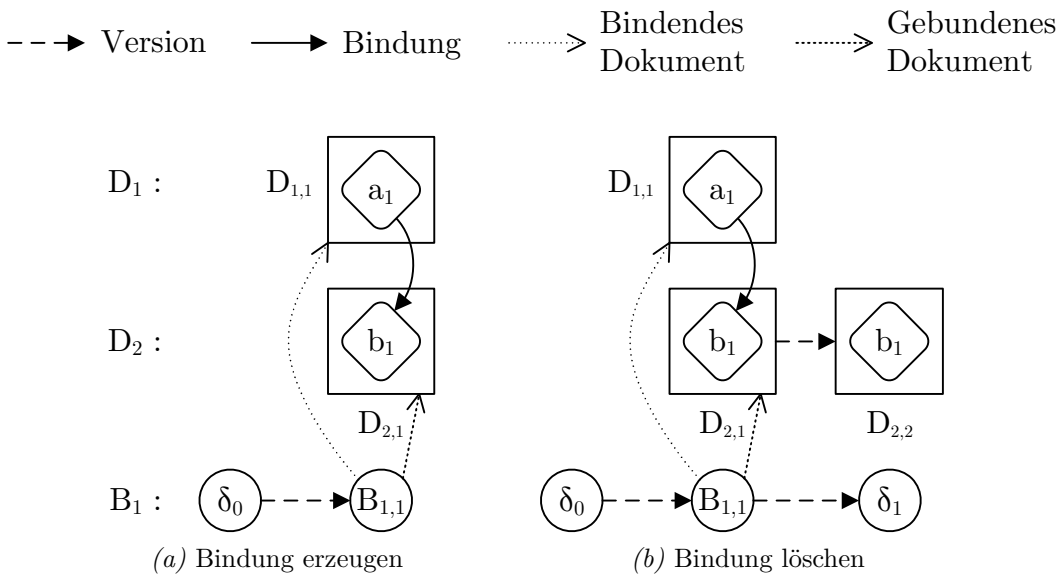
Abbildung 3.4: Erzeugen und Löschen von Bindungen

Das Hinzufügen einer neuen Elementversion setzt eine Änderung des zugrundeliegenden Objekts voraus, obwohl dies hier nicht zutrifft. Die gewünschte Modellierung entspricht der in Abbildung 3.4b, wo keine neuen Objekt- bzw. Elementversionen erzeugt werden müssen. Jedoch lässt sich das Bindungspaar (a_1, b_1) nicht eindeutig den Dokumentversionen $D_{2,1}$ und $D_{2,3}$ zuordnen. Zur Lösung des Problem muss die Bindung die zugehörige Dokumentversion des bindenden und gebundenen Elements speichern.

Beispiel 3.3: Modellierung der Bindung

Die neue Modellierung soll an einem Beispiel, das in Abbildung 3.5a beginnt, erläutert werden. In einem ersten Schritt werden jeweils die Ursprungsversionen $D_{1,1}$ und $D_{2,1}$ der zwei Dokumente D_1 und D_2 ins Repository übertragen. Beide Dokumente enthalten jeweils ein Element, repräsentiert durch die Elementversionen a_1 bzw. b_1. Zwischen a_1 und b_1 besteht eine Abhängigkeit in Form einer Bindung, die mit B_1 bezeichnet wird. Neu an dieser Stelle ist, dass die Bindung die bindende Dokumentversion $D_{1,1}$ und die gebundene Dokumentversion $D_{2,1}$ kennt. Neu ist weiterhin, dass die Bindung einen zeitlichen Aspekt hinzugewinnt und versioniert wird. So entsteht die erste Version $B_{1,1}$ der Bindung B_1 aus der virtuellen Version δ_0.

Abbildung 3.5b zeigt das Modell nach dem Löschen der Bindung B_1 durch einen Bearbeiter. Vom Dokument D_2 existiert eine neue Version $D_{2,2}$, die die unveränderte Elementversion b_1 enthält. Die Bindung B_1 geht in die virtuelle Version δ_1 über und endet in ihrer Historie.



(a) Bindung erzeugen *(b)* Bindung löschen

Abbildung 3.5: Beispiel: Versionierte Bindung, Schritt 1 und 2

Im nächsten Schritt wird wieder eine Bindung von a_1 nach b_1 eingetragen, die jetzt den Namen B_2 erhält (s. Abbildung 3.6). Wie zuvor entspringt die erste Version $B_{2,1}$ der virtuellen Version δ_0 und zeigt diesmal auf $D_{1,1}$ und $D_{2,3}$.

Der vierte und letzte Arbeitsschritt demonstriert den Fall, dass die Bindung unverändert bleibt, aber der Inhalt des Dokuments D_2 verändert wird, indem ein Element c hinzukommt. Im Repository entsteht die Elementversion c_1 und eine neue Version $B_{2,2}$ der Bindung B_2.

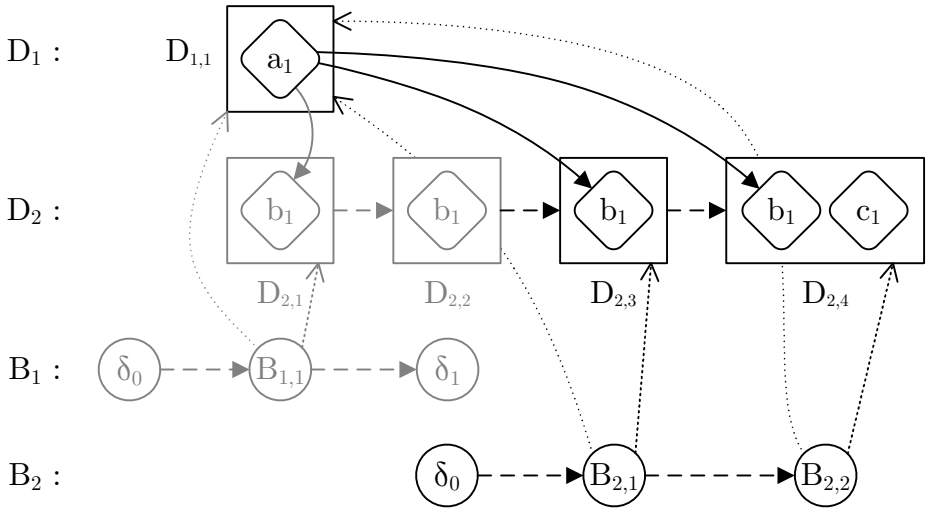


Abbildung 3.6: Beispiel: Versionierte Bindung, Schritt 3 und 4

Repository: Das mathematische Modell muss um eine Menge B erweitert werden, die Bindungselemente B_i enthält und als Bindungsmenge bezeichnet wird.

$$B := \{B_i \mid B_i \text{ ist eine Bindung}\} \tag{3.23}$$

Eine neue Version $B_{i,j}$ einer nicht gelöschten Bindung B_i entsteht, wenn eine neue Version $D_{k,l}$ des Dokuments committet wird, welche das gebundene Element f von B_i enthält. Alle Bindungsversionen werden in der Menge Ψ gespeichert.

$$\Psi := \{B_{i,j} \mid B_{i,j} \text{ ist eine Bindungsversion}\} \tag{3.24}$$

Die Bindungsversionsrelation V_B bildet den Bindungsversionsgraph durch Speicherung der Beziehungen zwischen den Bindungsversionen analog zu den bekannten Versionsrelationen aus Abschnitt 3.1.3.

Die Bindungsversionsabbildung β ordnet jeder Bindungsversion $B_{i,j}$ die zugehörige Bindung B_i zu.

$$\beta : \Psi \to B := \{(B_{i,j}, B_i) \in \Psi \times B \mid B_{i,j} \text{ ist eine Version der Bindung } B_i\} \tag{3.25}$$

Die Abbildung $\beta_{B,R} : B_{i,j} \to B_R$ stellt die Verknüpfung zwischen der Bindungsversion $B_{i,j}$ und dem zugehörigen Paar $(e_i, f_j) \in B_R$ her.

$$\begin{aligned}\beta_{B,R} : B_{i,j} \to B_R := \{&(B_{i,j}, (e_i, f_j)) \in \Psi \times B_R \mid \\ &\text{Bindungspaar } (e_i, f_j) \text{ gehört zur Bindungsversion } B_{i,j}\}\end{aligned} \tag{3.26}$$

Der Verweis einer Bindungsversion $B_{i,j}$ auf die Dokumentversionen $D_{i,j}$ und $D_{k,l}$, die die bindende Elementversion e_i bzw. die gebundene Elementversion f_j enthalten, werden durch die Abbildungen $\beta_{u,R}$ und $\beta_{z,R}$ übernommen. Die Indizes u und z stehen für *U*rsprung und *Z*iel.

$$\begin{aligned}\beta_{u,R} : \Psi \to \Gamma := \{&(B_{i,j}, D_{i,j}) \in \Psi \times \Gamma \mid D_{i,j} \text{ enthält die bindende Elementversion } e_i \\ &\text{der Bindungsversion } B_{i,j}\}\end{aligned} \tag{3.27}$$

$$\begin{aligned}\beta_{z,R} : \Psi \to \Gamma := \{&(B_{i,j}, D_{k,l}) \in \Psi \times \Gamma \mid D_{k,l} \text{ enthält die gebundene Elementversion } f_j \\ &\text{der Bindungsversion } B_{i,j}\}\end{aligned} \tag{3.28}$$

Sandbox: Die Dokumente mit ihren Objekten erscheinen in der Sandbox in einer unversionierten Sicht und können durch eine Bearbeitung in ihrem Zustand geändert werden. Für die Definition einer konsistenten Bindung in der Sandbox ist es wichtig, dass die Ursprungsdokumente mit den bindenden Objekten in ihrem Zustand eindeutig adressierbar sind. In der Sandbox ist es jedoch ausdrücklich erlaubt, Dokumente unbeschränkt zu bearbeiten, wobei nicht jeder zwischendurch gespeicherte Zustand wiederherstellbar ist.

Eindeutig definierte und adressierbare Zustände von Dokumenten existieren in der Sandbox nur direkt nach dem Übertragen zum oder Aktualisieren vom Server. Die Zuordnung kann dann über die Versionsnummer erfolgen, die in der Sandbox bekannt sein muss.

Die Definition einer Bindung setzt somit ein unverändertes Dokument seit einem Commit oder Update voraus.

$$\underset{e \in D_i}{\forall} e \in \overline{E_U} \Rightarrow \text{Dokument } D_i \text{ ist unverändert} \tag{3.29}$$

Die Abbildung $\beta_{B,S} : \overline{B} \to B_S$ stellt die Verknüpfung zwischen der Bindung B_i und dem zugehörigem Paar $(e, f) \in B_S$ her.

$$\begin{aligned}\beta_{B,S} : \overline{B} \to B_S := \{&(B_i, (e, f)) \in \Psi \times B_S \mid \\ &\text{Bindungspaar } (e, f) \text{ gehört zur Bindung } B_i\}\end{aligned} \tag{3.30}$$

Der Verweis einer Bindung B_i auf die Dokumente D_i und D_j, die das bindende Element e bzw. das gebundene Element f enthalten, werden durch die Abbildungen $\beta_{u,S}$ und $\beta_{z,S}$ übernommen.

$$\begin{aligned}\beta_{u,S} : \overline{B} \to D := \{&(B_i, D_i) \in \overline{B} \times D \mid D_i \text{ enthält das bindende Element } e \\ &\text{der Bindung } B_i\}\end{aligned} \tag{3.31}$$

$$\begin{aligned}\beta_{z,S} : \overline{B} \to D := \{&(B_i, D_j) \in \overline{B} \times D \mid D_j \text{ enthält das gebundene Element } f \\ &\text{der Bindung } B_i\}\end{aligned} \tag{3.32}$$

Die Bindung B_i wird je nach Bearbeitungszustand in einer der drei disjunkten Teilmengen von $\overline{B}$ gespeichert.

$$\overline{B} = \overline{B}_U \cup \overline{B}_N \cup \overline{B}_L \tag{3.33}$$

mit

$$\overline{B}_U := \{B_i \mid B_i \text{ ist eine unveränderte Bindung}\} \subseteq \overline{B} \tag{3.34}$$

$$\overline{B}_N := \{B_i \mid B_i \text{ ist eine neu erzeugte Bindung}\} \subseteq \overline{B} \tag{3.35}$$

$$\overline{B}_L := \{B_i \mid B_i \text{ ist eine gelöschte Bindung}\} \subseteq \overline{B} \tag{3.36}$$

3.1.5 Zusammenfassung

Tabelle 3.1 fasst alle für das mathematische Modell wesentlichen Mengen und Relationen zusammen. Sie können entweder nur in der Sandbox, nur im Repository oder auch in beiden gleichzeitig Verwendung finden. Die letzte Spalte der Tabelle führt jeweils ein Beispielelement der mathematischen Struktur auf. Es ist zu unterscheiden, dass die Sandbox eine unversionierte Sicht auf das Modell verwaltet und die Versionierung erst im Repository durchgeführt wird.

Bezeichnung	Sandbox		Repository	Beispiel-elemente
Objektmenge	$\overline{Q}$		Q	a, b, c
Dokumentmenge	$\overline{D}$		D	D_1, D_2
Elementmenge	$\overline{E}$		E	e, f
Anwendungsmenge	$\overline{A}$		A	A_1, A_2
Projektmenge	$\overline{P}$		P	P_1, P_2
Dokument	D_i		s. u.	a, b
Projekt	P_i		s. u.	D_1
Elementzuordnung		$\eta : E \leftrightarrow Q$		$(e, a), (f, b)$
Anwendungsabbildung		$\alpha : D \to A$		(D_1, A_1)
Objektversionsmenge	–		Ω	a_1, a_2, b_1
Dokumentversionsmenge	–		Γ	$D_{1,1}, D_{1,2}$
Elementversionsmenge	–		Ξ	e_1, e_2, f_1
Dokument	s. o.		D_i	$\{\}$
Projekt	s. o.		P_i	D_1, D_2
Freigabe	–		L_i	$D_{1,2}, D_{3,1}$
Freigabemenge	–		L	L_1, L_2
Elementversionszuordnung	–		$\varepsilon : \Xi \leftrightarrow \Omega$	(e_1, a_1)
Objektversionsabbildung	–		$\omega : \Omega \to Q$	(a_1, a)
Elementversionsabbildung	–		$\xi : \Xi \to E$	(e_1, e)
Dokumentversionsabbildung	–		$\gamma : \Gamma \to D$	$(D_{1,1}, D_1)$
Objektversionsrelation	–		V_O	(a_1, a_2)
Elementversionsrelation	–		V_E	(e_1, e_2)
Dokumentversionsrelation	–		V_D	$(D_{1,1}, D_{1,2})$
Bindungsmenge	$\overline{B}$		B	B_1, B_2
Bindungsrelation Sandbox (SB)	B_S		–	(e, f)
Bindungsrelationsabbildung SB	$\beta_{B,S} : \overline{B} \to B_S$		–	$(B_1, (e, f))$
Bindungsursprungsabbildung SB	$\beta_{u,S} : \overline{B} \to \overline{D}$		–	(B_1, D_1)
Bindungszielabbildung SB	$\beta_{z,S} : \overline{B} \to \overline{D}$		–	(B_1, D_2)
Bindungsversionsmenge	–		Ψ	$B_{1,1}, B_{1,2}, B_{2,1}$
Bindungsversionsrelation	–		V_B	$(B_{1,1}, B_{1,2})$
Bindungsversionsabbildung	–		$\beta : \Psi \to B$	$(B_{1,1}, B_{1,2}, B_1)$
Bindungsrelation Repository (RP)	–		B_R	(e_1, f_1)
Bindungsrelationsabbildung RP	–		$\beta_{B,R} : \Psi \to B_R$	$(B_{1,1}, (e_1, f_1))$
Bindungsursprungsabbildung RP	–		$\beta_{u,R} : \Psi \to \Gamma$	$(B_{1,1}, D_{1,1})$
Bindungszielabbildung RP	–		$\beta_{z,R} : \Psi \to \Gamma$	$(B_{1,1}, D_{2,1})$

Tabelle 3.1: Mathematisches Modell: Mengen und Relationen

3.2 Operationen

3.2.1 Hinweis

Die Operationen des mathematischen Modells wurden von (Firmenich, 2002, S. 23ff) und (Beer, 2005, S. 95ff) grundlegend beschrieben. Im Folgenden werden nur Ergänzungen zum erweiterten mathematischen Modell dieser Arbeiten aufgeführt, um Redundanz zu vermeiden.

3.2.2 Check-out (Projekt beitreten)

Diese neue Operation ist nötig, um als Bearbeiter einem existierenden Projekt $P_i \in P$ beizutreten und dort Dokumente $D_i \in D$ zu bearbeiten. Bevor Nutzer an Projekten teilnehmen können, muss ein Verantwortlicher in einem vorgelagerten Schritt das Projekt im Repository bekannt machen und in die Menge P eintragen.

$$P = P \cup \{P_i\} \tag{3.37}$$

Das Projekt P_i wird in der Sandbox in die Menge $\overline{P}$ und die Anwendung A_i in die Menge $\overline{A}$ eingetragen.

$$\overline{P} = \overline{P} \cup \{P_i \mid P_i \in P \wedge P_i \notin \overline{P}\} \tag{3.38}$$

$$\overline{A} = \overline{A} \cup \{A_i\} \tag{3.39}$$

3.2.3 Selektion

Die Selektion auf Elementbasis nach Beer setzt voraus, dass von den Objekten abgeleitete Elementeigenschaften existieren, um mit einer geeigneten Anfragesprache die Auswahl sinnvoll einzugrenzen. Dies ist oftmals zu speziell, da die Nutzer eher dokumentorientiert arbeiten und Dokumenteigenschaften, wie Name, Autor oder Datum, zur Auswahl heranziehen. Dementsprechend wird die Operation *Selektion* auf die Auswahl von Dokumentversionen ausgelegt. Dokumentversionen besitzen die zugeordneten Eigenschaften Tag, Autor, Übertragungsdatum und Revisionsnummer, die für die Selektion genutzt werden können. Ein Tag entspricht dem Namen für eine Dokumentversion $D_{i,j}$, da die Eigenschaft *Name* schon für das Dokument D_i reserviert ist.

In einem ersten Schritt werden die Dokumentversionen $D_{i,j}$ in die Dokumentversionsselektionsmenge S_D eingetragen, die den vom Nutzer vorgegebenen Eigenschaften entsprechen.

$$S_D := \{D_{i,j} \mid D_{i,j} \text{ ist eine selektierte Dokumentversion}\} \subseteq \Gamma \tag{3.40}$$

Im zweiten Schritt muss geprüft werden, ob Elementversionen $f_j \in D_{k,l}$ von anderen Elementversionen $e_i \in D_{i,j}$ abhängig sind. Wenn das der Fall ist, müssen die Dokumentversionen $D_{i,j}$ in die Menge S_D aufgenommen werden, sofern sie noch nicht in der Sandbox vorhanden sind.

$$S_D = S_D \cup \left\{ D_{i,j} \in \Gamma \mid \underset{D_{k,l} \in S_D}{\forall} \underset{e_i \in D_{i,j}}{\exists} \underset{f_j \in D_{k,l}}{\exists} (e_i, f_j) \in H(B) \wedge D_{i,j} \notin \overline{\Gamma} \right\} \tag{3.41}$$

3.2.4 Holen (Update New Documents)

Die Operation *Holen* lädt maximal eine Version je Dokument, das sich noch nicht in der Sandbox befindet, vom Repository und speichert es unversioniert in die Sandbox ab. Gleiches gilt für die enthaltenen Objekt- und Elementversionen. Die zu übertragenden Dokumentversionen $D_{i,j}$ sind in der Menge S_D verzeichnet.

$$\overline{D} = \overline{D} \cup \{\gamma(D_{i,j})\} \qquad \text{mit } D_{i,j} \in S_D \tag{3.42}$$
$$\overline{Q} = \overline{Q} \cup \{\omega(a_i)\} \qquad \text{mit } a_i \in D_{i,j} \tag{3.43}$$
$$\overline{E} = \overline{E} \cup \{\xi(e_i)\} \qquad \text{mit } e_i \in D_{i,j} \tag{3.44}$$

Bindungen: Im Unterschied zu Beer werden unselektierte Dokumentversionen immer komplett in die Sandbox übertragen, anstatt nur die bindenden Elemente abzulegen und sie mit einem Schreibschutz zu versehen. Das Schreibschutzkonzept einzelner Elemente in der Sandbox wird generell nicht weiter verwendet. Folgende Fälle sind bei der Operation Holen zu berücksichtigen, wenn sich außer dem gebundenen Dokument auch das bindende Dokument in der Sandbox befinden soll.

(1) Enthält die Selektionsmenge S_D Dokumentversionen $D_{i,j}$ ohne gebundene Elementversionen f_j, so sind keine weiteren Dokumente zu holen.

(2) Enthält die Selektionsmenge S_D Dokumentversionen $D_{i,j}$ mit bindenden Elementversionen e_i und sind die Dokumentversionen $D_{k,l}$ mit den gebundenen Elementversionen f_j nicht in S_D, so sind keine weiteren Dokumente zu holen.

(3) Enthält die Selektionsmenge S_D Dokumentversionen $D_{k,l}$ mit gebundenen Elementversionen f_j und sind die bindenden Elementversionen e_i Bestandteil von Dokumentversionen $D_{i,j} \in S_D$, so sind keine weiteren Dokumente zu holen.

(4) Enthält die Selektionsmenge S_D Dokumentversionen $D_{k,l}$ mit gebundenen Elementversionen f_j und sind die bindenden Elementversionen e_i Bestandteil von Dokumentversionen $D_{i,j} \notin S_D$, so sind diese Dokumente in S_D aufzunehmen und zu holen.

$$S_D = S_D \cup \left\{ D_{i,j} \in \Gamma \mid \underset{(e_i, f_j) \in B_R}{\exists} e_i \in D_{i,j} \wedge f_j \in D_{k,l} \wedge D_{i,j} \notin S_D \wedge D_{k,l} \in S_D \right\} \tag{3.45}$$

Nach dem Holen aller Dokumentversionen $D_{i,j} \in S_D$ können die Bindungen vom Repository zur Sandbox übertragen werden.

$$B_S = B_S \cup \left\{ (\xi(e_i), \xi(f_j)) \mid \underset{(e_i,f_j)\in B_R}{\exists} e_i \in D_{i,j} \wedge f_j \in D_{k,l} \wedge D_{i,j} \in S_D \wedge D_{k,l} \in S_D \right\} \tag{3.46}$$

$$\beta_{B,S} = \beta_{B,S} \cup \left\{ (\beta(B_{i,j}), (\xi(e_i), \xi(f_j))) \mid \underset{B_{i,j}\in\Psi}{\exists} \beta_{u,R}(B_{i,j}) \in S_D \wedge \beta_{z,R}(B_{i,j}) \in S_D) \right\} \tag{3.47}$$

$$\beta_{u,S} = \beta_{u,S} \cup \left\{ (\beta(B_{i,j}), D_{i,j}) \mid \underset{B_{i,j}\in\Psi}{\exists} D_{i,j} \in S_D \wedge \beta_{u,R}(B_{i,j}) = D_{i,j}) \right\} \tag{3.48}$$

$$\beta_{z,S} = \beta_{z,S} \cup \left\{ (\beta(B_{i,j}), D_{k,l}) \mid \underset{B_{i,j}\in\Psi}{\exists} D_{k,l} \in S_D \wedge \beta_{z,R}(B_{i,j}) = D_{k,l}) \right\} \tag{3.49}$$

3.2.5 Laden

Die Operation *Laden* lädt ein Dokument aus der Sandbox in die Anwendung und überführt es für die Bearbeitung vom persistenten in den transienten Zustand. Die Beschreibung von (Beer, 2005, S. 100) ist hier unverändert gültig.

3.2.6 Bearbeiten

Durch das Bearbeiten in der Anwendung werden nur Objekte, Dokumente oder Bindungen direkt erzeugt, geändert oder gelöscht. Die Aktualisierung der zugehörigen Elemente übernimmt der Workspace. Folgende Ergänzungen zu den Vorarbeiten werden vorgenommen.

Dokument: Ein erzeugtes Dokument wird in die Menge $\overline{D}$ eingetragen.

$$\overline{D} = \overline{D} \cup \{D_i\} \tag{3.50}$$

(3.51)

Ein Dokument D_i gilt als geändert, wenn

- sich der Zustand mindestens eines Objekts $a \in D_i$ geändert hat oder
- mindestes ein Objekt zum Dokument hinzugefügt oder vom Dokument gelöscht wurde oder
- mindestes eine Bindung hinzugefügt oder gelöscht wurde, die ein Objekt des Dokuments bindet (s. Abschnitt 3.1.4 auf Seite 82).

Nach dem Löschen eines Dokuments müssen alle Objekte $a \in D_i$ und die zugeordneten Elemente $e \in D_i$ gelöscht werden.

$$\overline{Q} = \overline{Q} \setminus \left\{ a \mid \underset{a \in \overline{Q}}{\forall} a \in D_i \right\} \tag{3.52}$$

$$\overline{E} = \overline{E} \setminus \left\{ e \mid \underset{e \in \overline{E}}{\forall} e \in D_i \right\} \tag{3.53}$$

$$\overline{D} = \overline{D} \setminus \{D_i\} \tag{3.54}$$

$$\tag{3.55}$$

Bindungen: Eine neu erzeugte Bindung B_i zwischen den Elementen e und f ist in die Menge $\overline{B}$ einzutragen. Weiterhin sind die Abbildungen $\beta_{B,S}$, $\beta_{u,S}$ und $\beta_{z,S}$ anzupassen.

$$B = B \cup \{B_i\} \tag{3.56}$$

$$B_S = B_S \cup \{(e, f) \mid f \text{ ist von } e \text{ abhängig}\} \tag{3.57}$$

$$\beta_{B,S} = \beta_{B,S} \cup \{(B_i, (e, f)) \mid f \text{ ist von } e \text{ abhängig}\} \tag{3.58}$$

$$\beta_{u,S} = \beta_{u,S} \cup \{(B_i, D_i) \mid e \in D_i\} \tag{3.59}$$

$$\beta_{z,S} = \beta_{z,S} \cup \{(B_i, D_j) \mid f \in D_j\} \tag{3.60}$$

Eine Bindung $B_i \in B$ wird mit folgendem Formalismus aus der Sandbox entfernt.

$$\overline{B} = \overline{B} \setminus \{B_i\} \tag{3.61}$$

$$\overline{B_L} = \overline{B_L} \cup \{B_i\} \tag{3.62}$$

$$B_S = B_S \setminus \{\beta_{B,S}(B_i)\} \qquad \text{mit } \beta_{B,S}(B_i) = (e, f) \tag{3.63}$$

$$\beta_{B,S} = \beta_{B,S} \setminus \{(B_i, \beta_{B,S}(B_i))\} \tag{3.64}$$

$$\beta_{u,S} = \beta_{u,S} \setminus \{(B_i, D_i) \mid e \in D_i\} \tag{3.65}$$

$$\beta_{z,S} = \beta_{z,S} \setminus \{(B_i, D_j) \mid f \in D_j\} \tag{3.66}$$

Nach dem Löschen eines gebundenen Objekts b aus einem Dokument können alle Bindungen gelöscht werden, die das zugehörige Element $f = \eta^{-1}(b)$ als bindendes Element enthalten.

$$B_S = B_S \setminus \{(e, f) \in B_S \mid \eta(f) = b \wedge b \text{ wurde gelöscht}\} \tag{3.67}$$

Bindungen dürfen nicht automatisch austragen werden, wenn ein bindendes Element gelöscht wird, da der Bearbeiter beim Prüfen des Dokuments, welches das abhängige Element enthält, darauf hingewiesen werden soll.

3.2.7 Speichern

Die Operation *Speichern* ist die Umkehroperation zum *Laden*.

3.2.8 Vergleichen (Diff)

Die Operation *Vergleichen* vergleicht zwei Dokumente und ist Voraussetzung für die Operationen *Zusammenführen von Varianten* und *Aktualisieren*. Je nach Anwendungsfall wird entweder eine Dokumentversion des Repositorys mit dem aktuellen Zustand in der Sandbox oder zwei Dokumentversionen des Repositorys verglichen. Beide Dokumente müssen sich auf dem lokalen Rechner befinden, so dass entfernt liegende Versionen erst in einen temporären Bereich der Sandbox geholt werden.

Objekte und Elemente: Zunächst werden die jeweils enthaltenen Objekte miteinander verglichen. Einer der fünf Fälle aus Tabelle 3.3 trifft für jedes Objekt zu.

Fall		Dokument 1	Dokument 2
1	Objekt ist	–	–
2	Objekt ist	enthalten	–
3	Objekt ist	–	enthalten
4	Objekt ist	enthalten und gleich	
5	Objekt ist	enthalten und ungleich	

Tabelle 3.3: Vergleich von Dokumenten: Objekte

Der Vergleich zweier Objekte erfolgt durch das Vergleichen der Objektattribute, die sich in einfache und komplexe unterscheiden. Referenzen zu anderen Objekten werden durch deren POID repräsentiert und sind damit eindeutig vergleichbar.

Bindungen: In einem weiteren Schritt werden nur die Bindungen (e, f) betrachtet, deren gebundene Elemente f Bestandteil der zu vergleichenden Dokumente sind. Tabelle 3.4 zeigt die vier möglichen Fälle.

Fall		Dokument 1	Dokument 2
1	Bindung ist	–	–
2	Bindung ist	vorhanden	–
3	Bindung ist	–	vorhanden
4	Bindung ist	vorhanden	

Tabelle 3.4: Vergleich von Dokumenten: Bindungen

3.2.9 Aktualisieren (Update)

Objekte und Elemente: Die Operation *Aktualisieren* gleicht ein Dokument D_i in der Sandbox mit der aktuellen Dokumentversion $D_{i,j}$ im Repository ab. Zur Bestimmung der aktuellen Version lässt sich das bei (Beer, 2005, S. 105) zur Bestimmung der aktuellen Elementversion erklärte Verfahren analog zur Bestimmung der aktuellen Dokumentversion

anwenden. Für jedes Objekt in D_i wird der Zustand in der Sandbox mit dem Zustand der Objektversion in $D_{i,j}$ verglichen.

(Beer, 2005, S. 106) verwendet zum Abgleich von Objekten eine Matrix, die stellvertretend die Elementzustände im Repository und in der Sandbox betrachtet. *Neu, Geändert, Gelöscht* und *Ungeändert* sind dabei die möglichen Zustände, die für jedes Objekt bestimmt werden müssen. Unter Ausschluss von sieben unmöglichen Fällen bleiben noch neun Kombinationsmöglichkeiten übrig. Diese lassen sich auf fünf reduzieren, wenn bei Objekten mit gleicher POID nur geprüft wird, *ob* statt *wo* sich das Objekt geändert hat. Die neue Abgleichsmatrix vereinfacht sich zu Tabelle 3.5, die mit der Operation *Vergleichen* eingeführt wurde. In der letzten Spalte steht die Aktion, die zum Zusammenführen eines Objekts auszuführen ist. Das entstehende Objekt bekommt die gleiche POID zugewiesen.

Fall		Dokument in der Sandbox	Dokumentversion im Repository	Aktion
1	Objekt ist	–	–	Keine
2	Objekt ist	enthalten	–	Löschen ?
3	Objekt ist	–	enthalten	Übernehmen ?
4	Objekt ist	enthalten und gleich		Übernehmen
5	Objekt ist	enthalten und ungleich		Zusammenführen

? Entscheidung des Planers

Tabelle 3.5: Aktualisieren von Dokumenten: Objekte

Im Gegensatz zu Beer wird kein automatisches Übernehmen oder Löschen von Objekten zugelassen. Der Planer soll die Hoheit über das Datenmodell behalten und selbst entscheiden, aus welchen Objekten sich das aktualisierte Dokument zusammensetzt.

Bindungen: Die Operation *Vergleichen* ermittelt das Vorkommen der in Frage kommenden Bindungen, was zu den Aktionen aus Tabelle 3.6 führt.

Fall		Dokument in der Sandbox	Dokumentversion im Repository	Aktion
1	Bindung ist	–	–	Keine
2	Bindung ist	vorhanden	–	Löschen?
3	Bindung ist	–	vorhanden	Übernehmen?
4	Bindung ist	vorhanden		Übernehmen

? Entscheidung des Planers

Tabelle 3.6: Aktualisieren von Dokumenten: Bindungen

3.2.10 Zusammenführen von Varianten

Beim Erstellen von Dokumentvarianten verzweigt sich der Versionsgraph, so dass mehrere parallele Dokumentversionen existieren. Die Operation *Zusammenführen von Varianten* soll diese Varianten wieder zusammenführen. Theoretisch ließe sich eine endliche Anzahl von Dokumentversionen zusammenführen, praktisch ist das für den Nutzer nur für zwei überschaubar. Beide Dokumentversionen $D_{1,1}$ und $D_{1,2}$ müssen in die Sandbox übertragen werden, bevor sie verglichen und vereinigt werden können. Gegenüber der Operation *Aktualisieren* sind beide Versionen gleichwertig, so dass sich die Aktionen leicht unterscheiden (s. Tabelle 3.7).

Fall		Dokument-variante 1	Dokument-variante 2	Aktion
1	Objekt ist	–	–	Keine
2	Objekt ist	enthalten	–	Übernehmen ?
3	Objekt ist	–	enthalten	Übernehmen ?
4	Objekt ist	enthalten und gleich		Übernehmen
5	Objekt ist	enthalten und ungleich		Zusammenführen oder Löschen ?

? Entscheidung des Planers

Tabelle 3.7: Zusammenführen von Dokumentvarianten: Objekte

Bindungen: Wegen der Gleichwertigkeit gibt es auch keine Aktion *Löschen* für Bindungen, sondern nur das *Übernehmen*.

Fall		Dokument-variante 1	Dokument-variante 2	Aktion
1	Bindung ist	–	–	Keine
2	Bindung ist	vorhanden	–	Übernehmen?
3	Bindung ist	–	vorhanden	Übernehmen?
4	Bindung ist	vorhanden		Übernehmen

? Entscheidung des Planers

Tabelle 3.8: Zusammenführen von Dokumentvarianten: Bindungen

Versionierung: Nach dem Zusammenführen in der Sandbox muss das resultierende Dokument in das Repository übertragen werden, wo neue Versionen für die Objekte, Elemente und Bindungen sowie eine neue Version für das Dokument angelegt werden. Im Versionsgraph des Dokuments entsteht die neue Dokumentversion $D_{1,3}$ aus den beiden Ursprungsversionen $D_{1,1}$ und $D_{1,2}$.

3.2.11 Übertragen (Commit)

Wenn das Dokument D_i in der Sandbox einen Stand erreicht hat, der für andere zugänglich sein soll, kann der Bearbeiter das Dokument in das Repository als neue Dokumentversion $D_{i,k}$ übertragen. Voraussetzung dafür ist, dass Änderungen an Objekten oder Bindungen, die auf ein Objekt des Dokuments verweisen, vorgenommen wurden.

Dokument: Falls die Anwendung A_i noch nicht im Repository bekannt ist, muss sie dort aufgenommen werden.

$$A = A \cup \{A_i \in \overline{A} \mid A_i \notin A \wedge \alpha(D_i) = A_i\} \tag{3.68}$$

Ist das Dokument D_i noch nicht im Repository vorhanden, wird es in die Dokumentmenge D aufgenommen und als Ursprungsversion $D_{i,k}$ in die Versionsrelation V_D eingetragen. Andernfalls erscheint es als Nachfolgeversion von $D_{i,j}$. Weiterhin muss die neue Dokumentversion $D_{i,k}$ in die Dokumentversionsmenge Γ und die Zuordnung von $D_{i,k}$ zu D_i in die Dokumentversionsabbildung γ eingetragen werden.

$$D = D \cup \{D_i \mid D_i \in \overline{D} \wedge D_i \notin D\} \tag{3.69}$$
$$\Gamma = \Gamma \cup \{D_{i,k}\} \tag{3.70}$$
$$\gamma = \gamma \cup \{(D_{i,k}, D_i) \mid D_i \in \overline{D} \wedge D_i \in D\} \tag{3.71}$$
$$V_D = V_D \cup \{(\delta_0, D_{i,k})\} \quad \text{Ursprungsversion} \tag{3.72}$$
$$V_D = V_D \cup \{(D_{i,j}, D_{i,k})\} \quad \text{Geänderte Version} \tag{3.73}$$

Bindungen: Von nicht gelöschten Bindungen $B_i \in \overline{B}$ wird grundsätzlich bei jedem Übertragen einer Dokumentversion eine neue Version $B_{i,k}$ angelegt und in die Bindungsversionsmenge Ψ eingetragen. Neu erzeugte Bindungen müssen zusätzlich in die Bindungsmenge B sowie als Nachfolgeversion der virtuellen Version δ_0 in die Bindungsversionsrelation V_B aufgenommen werden. Weiterhin verzeichnet V_B gelöschte und ungeänderte Bindungen. Die Bindungsrelation $B_{B,R}$ speichert das Elementversionspaar (e_i, f_j). Die Zuordnung des Elementversionspaares (e_i, f_j), der bindenden Dokumentversion $D_{i,j}$ und der gebundenen Dokumentversion $D_{k,l}$ zur Bindungsversion $B_{i,k}$ übernehmen die Abbildungen $\beta_{B,R}, \beta_{u,R}$ und $\beta_{z,R}$.

$$B = B \cup \{B_i \mid B_i \in \overline{B} \wedge B_i \notin B\} \tag{3.74}$$
$$\Psi = \Psi \cup \{B_{i,k}\} \tag{3.75}$$
$$\beta = \beta \cup \{(B_{i,k}, B_i) \mid B_i \in \overline{B} \wedge B_i \in B\} \tag{3.76}$$
$$B_{B,R} = B_{B,R} \cup \{(e_i, f_j) \mid e_i \in D_{i,j} \wedge f_j \in D_{k,l} \wedge f_j \text{ ist von } e_i \text{ abhängig}\} \tag{3.77}$$
$$\beta_{B,R} = \beta_{B,R} \cup \{(B_{i,k}, (e_i, f_j)) \mid e_i \in D_{i,j} \wedge f_j \in D_{k,l} \wedge f_j \text{ ist von } e_i \text{ abhängig}\} \tag{3.78}$$
$$\beta_{u,R} = \beta_{u,R} \cup \{(B_{i,k}, D_{i,j}) \mid e_i \in D_{i,j}\} \tag{3.79}$$
$$\beta_{z,R} = \beta_{z,R} \cup \{(B_{i,k}, D_{k,l} \mid f_j \in D_{k,l})\} \tag{3.80}$$
$$V_B = V_B \cup \{(\delta_0, B_{i,k}) \mid \beta(B_{i,k}) \in \overline{B}_N\} \tag{3.81}$$
$$V_B = V_B \cup \{(B_{i,j}, B_{i,k}) \mid \beta(B_{i,k}) \in \overline{B}_U\} \tag{3.82}$$
$$V_B = V_B \cup \{(B_{i,k}, \delta_1) \mid \beta(B_{i,k}) \in \overline{B}_L\} \tag{3.83}$$

3.2.12 Freigeben

Eine Freigabe setzt sich aus zueinander widerspruchsfreien Dokumentversionen zusammen und repräsentiert einen Projektstand zu einem bestimmten Zeitpunkt. Für das mathematische Modell werden die zugehörigen Dokumentversionen $D_{i,j}$ zu einer Freigabe L_i zusammengefasst, wobei von einem Dokument D_i nur eine Version enthalten sein darf. Ist eine Dokumentversion mit gebundenen Elementversionen enthalten, müssen alle Dokumentversionen, die die bindenden Elementversionen enthalten, in der Freigabe enthalten sein. Formell sind die Konsistenzbedingungen für eine Freigabe auf Seite 81 beschrieben.

4 Konzepte und Umsetzung der Systemarchitektur

> Ich stelle fest, dass es zwei Wege gibt, ein Software-Design zu erstellen, entweder so einfach, dass es offensichtlich keine Schwächen hat, oder so kompliziert, dass es keine offensichtlichen Schwächen hat.
>
> *(Tony Hoare, 1980)*

4.1 Vorgehensweise

Systemarchitektur: Die generelle Systemarchitektur von Beer, wie sie im Abschnitt 2.4.5 auf Seite 51 vorgestellt wurde, besitzt für diese Arbeit weiterhin ihre Gültigkeit. So übernimmt ein textbasiertes VCS die Versionierung und Verwaltung der Objekte und die Feature-Logic dient als zusätzliche Modellierungsschicht, z. B. für die Definition von Objektabhängigkeiten oder Freigabeständen. Darüber hinaus dient die Feature-Logic mit der von Firmenich entwickelten Abfragesprache als flexibles Selektionswerkzeug.

Das in diesem Kapitel vorgestellte Umsetzungskonzept weist für die einzelnen Komponenten jedoch Unterschiede auf, was zum einen der Änderung des mathematischen Modells und zum anderen der verstärkten Ausrichtung auf die Praxistauglichkeit im Hinblick auf Leistung und Handhabbarkeit geschuldet ist. Die wesentlichen Änderungen betreffen die folgenden Punkte.

Kommunikation: Wie in Abbildung 2.30 auf Seite 52 dargestellt, existieren zwei Datenströme zwischen Project und Workspace, und zwar zwischen FL-Server und FL-Client sowie zwischen VCS-Server und VCS-Client. Für den ersteren verwendete Beer eine von (Firmenich, 2000) vorgestellte und auf Nachrichten basierende Lösung. Diese Kommunikation wird auf das standardisierte und einfach anwendbare Java/RMI[1] umgestellt. Die Kommunikation des zweiten Datenstroms beschränkt sich auf die vom VCS angebotenen Protokolle. Da in der Umsetzung auch Wert auf die Sicherheit der Daten gelegt wird, werden beide Datenströme mit dem SSL- bzw. SSH-Protokoll verschlüsselt.

Feature-Logic: Die Feature-Logic-Umsetzung wird einigen notwendigen Änderungen unterzogen.

[1] s. Absatz *Java/RMI* auf Seite 237

- **Datenspeicher:** Der verwendete Datenspeicher soll von vornherein nicht auf ein Datenbanksystem eines Herstellers festgelegt, sondern transparent austauschbar sein. Außerdem ist es nicht sinnvoll, auf den Clientrechnern ein großes Relationales Datenbankmanagementsystem (RDBMS) zu installieren. Als Alternative werden eine direkte Speicherung in das Dateisystem und eine rein transiente Variante untersucht.

- **Leistung:** Datenbanksysteme bieten dem Programmierer verschiedene Methoden zur Leistungsverbesserung. Diese werden in die Datenbankumsetzung der Feature-Logic übernommen.

- **Datentypen:** Bisher fehlte ein atomarer Datentyp zum direkten Speichern eines Datums. Für die Selektion von Dokumentversionen ist dieser aber notwendig und wird in die Feature-Logic aufgenommen.

- **Selektion:** Eine weitere Funktionalität, die für die Selektion gebraucht wird, ist das Vergleichen von atomaren Werten, um zum Beispiel Dokumente zeitlich einzugrenzen. Dafür ist unter anderem der Feature-Logic-Interpreter aus (Firmenich, 2002, S. 102ff) anzupassen.

Project: Die Verwendung von RMI als Kommunikationsprotokoll bedingt das Implementieren und Bereithalten eines RMI-Servers.

Workspace: Der Workspace beinhaltet den Großteil der Implementierungen, so dass hier wesentliche Änderungen und Ergänzungen erfolgen.

- **VCS-Anbindung:** Für die bekannten Versionsverwaltungssysteme existieren Bibliotheken für die Anbindung an Java, die die Programmierung wesentlich erleichtern. Gleichwohl muss über einen speziellen VCS-Client die Kopplung zwischen Workspace und der Bibliothek hergestellt werden.

- **Feature-Logic-Anbindung:** Die Umstellung auf RMI führt auch zu einer Neuimplementierung des Feature-Logic-Clients.

- **Objektserialisierung:** Der bisher verwendete XML-Serialisierer weist Nachteile hinsichtlich Leistung und Flexibilität auf. Daraufhin wurden alternative Serialisierungskonzepte untersucht.

- **Objektabhängigkeiten:** Die Abhängigkeiten werden durch Bindungen realisiert, die im Project nun auch versioniert werden. Die Verwaltung erfolgt sowohl transient in der laufenden verteilten Anwendung als auch persistent im Workspace und im Project.

- **Grafische Benutzeroberfläche:** Geeignete grafische Benutzeroberflächen vereinfachen dem Bearbeiter den Umgang mit dem neuen Konzept der Objektversionierung innerhalb der gewohnten Anwendung. Für jede verteilte Operationen ist eine Nutzeroberfläche zu entwerfen, wobei manche Dialoge auch mehrfach verwendet werden können.

Softwareentwicklung: Das objektorientierte Design und die Softwareentwicklung wird in drei voneinander unabhängigen Softwareprojekten vorgenommen, damit unbeabsichtigte Abhängigkeiten zwischen Server und Client vermieden werden.

- *INTERCAD*: Dieses Projekt enthält die Pakete, die sowohl auf der Client- als auch auf der Serverseite benötigt werden und eine gewisse Unabhängigkeit aufweisen. Dazu gehören Datenbankanbindung, Feature-Logic, RMI-Schnittstelle und allgemeine Hilfsklassen.
- *INTERCAD_PROJECT*: Dieses Projekt für die Serveraufgaben fällt sehr klein aus, da es nur die RMI-Implementierung und einen RMI-Server enthält, der seinerseits eine Instanz der Feature-Logic startet. Der VCS-Server muss nicht extra gestartet werden, da er als Dienst ständig läuft.
- *INTERCAD_WORKSPACE*: Dieses Projekt ist sehr umfangreich, da es die anwendungsunabhängigen Funktionen des Workspace enthält. Dazu gehören der Workspace als integraler Bestandteil, VCS-Client, eine Implementierung für ein spezielles Versionsverwaltungssystem, Feature-Logic-Client, grafische Benutzeroberflächen und ein Paket für die Objektserialisierung in der Sandbox.

Für jede Anwendung ist es ratsam, ein eigenes Softwareprojekt anzulegen, das einen vom allgemeinen Workspace abgeleiteten und für die Anwendung erweiterten Workspace enthält. Hinzu kommen die Implementierungen der Kommandos für die verteilte Bearbeitung, Erweiterungen für die Objektserialisierung und die grafischen Nutzeroberflächen.

4.2 Entwurf und Umsetzung grundlegender Klassen

4.2.1 Allgemeine Klassen

Datenbankanbindung: Für die Feature-Logic wird eine Anbindung zu einer relationalen Datenbank benötigt. Im Java Development Kit (JDK) ist die Datenbankschnittstelle Java Database Connectivity (JDBC) enthalten, die eine einheitliche Schnittstelle zu Datenbanken verschiedener Hersteller bietet. Für jede Datenbank muss ein vom Hersteller angebotener JDBC-Treiber verwendet werden, der zwischen dem JDBC-API und der Datenbank vermittelt. Für einen Zugriff auf die Datenbank muss zuerst der Treiber bei einem Treibermanager registriert und von diesem eine Verbindung angefordert werden (Zeile 1 bis 7 im Listing 4.1). Danach kann, wie in den Zeilen 9 bis 12 zu sehen, eine SQL-Anweisung erzeugt, ausgeführt und das Ergebnis über eine *ResultSet* abgefragt werden. Die Zeilen 14 bis 16 enthalten den Code für das Schließen des ResultSets, der SQL-Anweisung und der Datenbankverbindung.

```
1 String driverName = "oracle.jdbc.driver.OracleDriver";
2 Driver d = (Driver)Class.forName(driverName).newInstance();
3 DriverManager.registerDriver(d);
4 String dbHost = "dbserver.uni-weimar.de";
```

```
String dbService = "interCAD";
String dbUrl = "jdbc:oracle:thin:@" + dbHost + ":1521:" +
   dbService;
Connection con = DriverManager.getConnection(dbUrl, "Nutzer", "
   Passwort");

Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM Domain");
while (rs.next())
    System.out.println(rs.getString(1));

rs.close();
stmt.close();
con.close();
```

Listing 4.1: Ausführen einer SQL-Abfrage mit JDBC auf einer Oracle-Datenbank

Das Auf- und Abbauen von Datenbankverbindungen ist sehr zeitintensiv. Bei der Ausführung von mehreren SQL-Anweisungen auf der gleichen Datenbank sollte die Verbindung aufrechterhalten werden. In der Literatur wird mit dem *Connection Pool* ein Verfahren beschrieben, das ständige Verbindungen zur Datenbank vorhält und diese, falls sie frei sind, an einen Client weiterleiten kann.

Aus diesem Grund wurde eine Schnittstelle *DatabaseConnector* und ein davon abgeleiteter *DatabaseConnectorAdapter* geschrieben, die diese Funktionalität für eine aktive Verbindung umsetzen (s. Abbildung 4.1). Weiterhin wurde die Treiberregistrierung und Anmeldeprozedur in den Adapter aufgenommen, so dass für jede neue Datenbank nur ein kleiner, vom Adapter abgeleiteter Connector erzeugt werden muss, der über seinen Konstruktor nur die Anmeldedaten entgegennimmt und die Datenbank-URL zusammensetzt.

Die Anwendung das OracleConnectors zeigt folgender Quellcode. Dem Connector werden Servername, Datenbankname, Nutzername und Passwort übergeben und über die Methode *getConnection()* wird eine Datenbankverbindung angefordert.

```
DatabaseConnector connector =
    new OracleConnector(dbHost, dbService, dbUser, dbPwd);
Connection con = connector.getConnection();
```

Listing 4.2: Anwendung des OracleConnectors

RMI-Schnittstelle: Die RMI-Schnittstelle definiert diejenigen Methoden, die ein Client auf dem Server aufrufen kann. Sie muss demzufolge auf beiden vorhanden sein, es existiert aber nur auf dem Server eine implementierende Klasse. Für die Kommunikation mit dem Feature-Logic-Server wurde die Schnittstelle *FLRmiInterface* mit den entsprechenden Methoden geschrieben (Abbildung 4.2). Das einzige Attribut der RMI-Schnittstelle ist eine Konstante, die den Port der RMI-Registrierung auf den Standardwert 1099 festlegt.

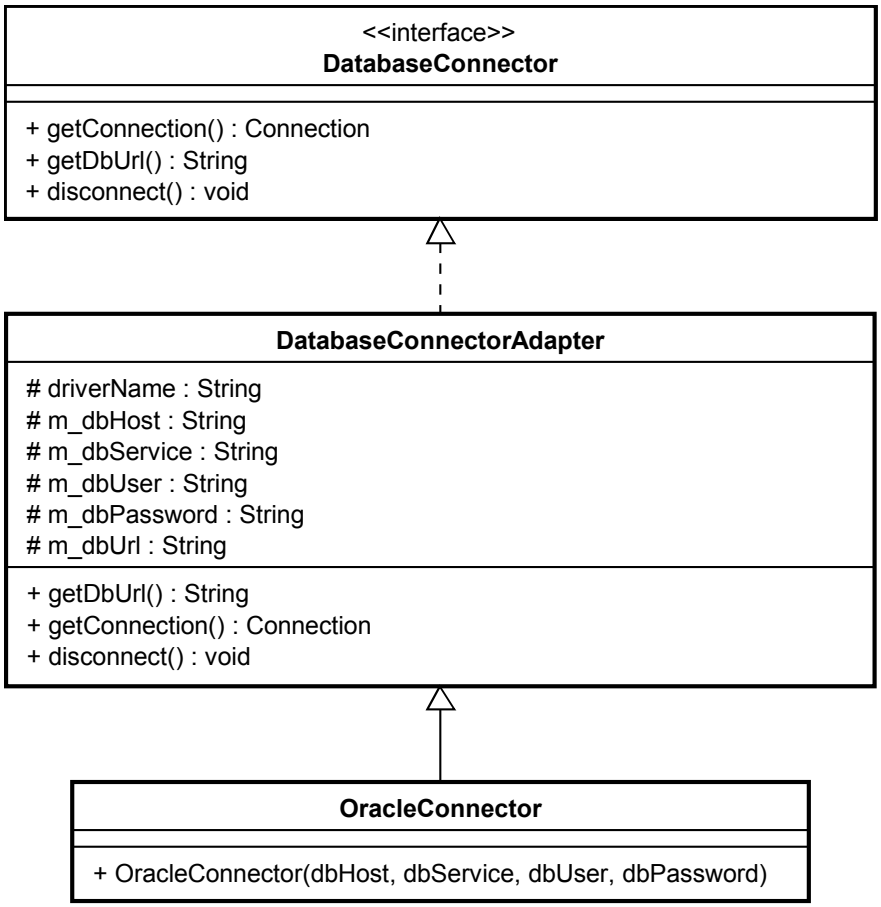


Abbildung 4.1: UML-Klassendiagramm: DatabaseConnector

Die erste Methode *checkConnection()* prüft, ob eine Verbindung zum Feature-Logic-Server besteht, und sollte vor Aufruf einer der folgenden Methoden aufgerufen werden. Die zweite Methode *clear()* löscht den kompletten Inhalt der Feature-Logic und dient nur zu Testzwecken. Eine Datenlöschung während oder nach der Bearbeitung eines realen Projekts verstößt gegen die Archivierungsgrundsätze.

Innerhalb der Operation *Übertragen (Commit)*, s. Abschnitt 3.2.11 auf Seite 94, wird zuerst das Dokument im VCS versioniert und nach Erhalt der Revisionsnummer die zusätzliche Modellierung in der zentralen Feature-Logic gespeichert. Diese Aufgabe übernimmt die Methode *commit()*, die zur Übergabe der Daten ein Objekt der Klasse *FLDataImpl* verwendet, welche die Schnittstelle *FLData* implementiert (s. Abschnitt D.5 auf Seite 253). *FLData* verfügt über Methoden für das Hinzufügen von Elementen, Features und Verknüpfungen zum *FLDataImpl*-Objekt sowie zum Speichern dieser Daten in eine Feature-Logic-Instanz nach dem erfolgreichen Senden an den RMI-Server.

In umgekehrter Richtung arbeitet die Methode *askQuery()*, da sie für einen vorgegebenen Feature-Term nach Auswertung mit dem FL-Interpreter die resultierende Menge an Elementen zurückgibt. Die restlichen sechs Methoden greifen direkt auf den Feature-Logic-Datenbestand zu und sind durch die Darstellung im Klassendiagramm selbsterklärend.

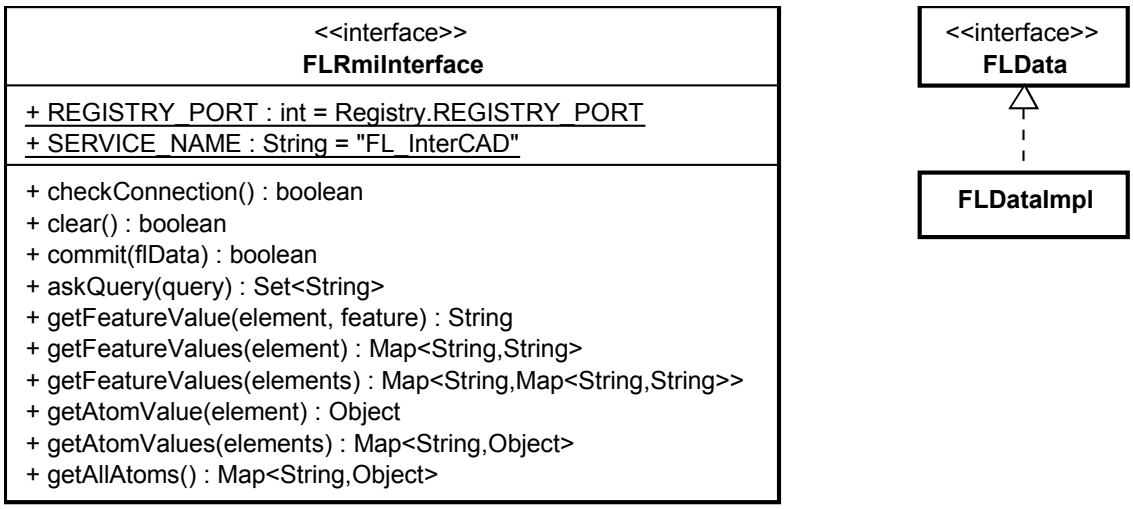


Abbildung 4.2: UML-Klassendiagramm: RMI- und FLData-Schnittstelle

4.2.2 Project

RMI-Server: Auf der Serverseite muss ein RMI-Server entworfen und programmiert werden, der die Methoden der RMI-Schnittstelle implementiert, bei Client-Anfragen ausführt und Rückgabewerte zurückliefert. An zentraler Stelle im Klassendiagramm (Abbildung 4.3) steht die Klasse *Project*, die zuerst über den *DatabaseConnector* eine Datenbankverbindung (Connection) herstellt und diese an eine Feature-Logic-Instanz übergibt. Danach startet sie den Feature-Logic-Server (*FL-Server*), der auch Zugriff auf die Feature-Logic erhält.

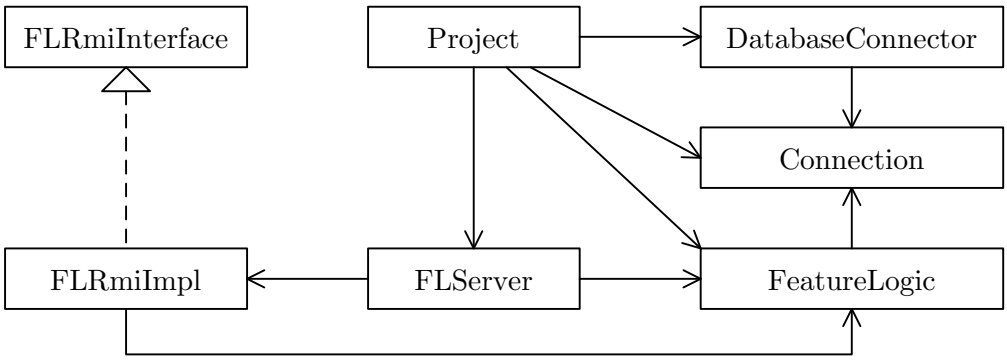


Abbildung 4.3: UML-Klassendiagramm: Project

Der Programmablauf innerhalb der Klasse *FL-Server* sieht folgendermaßen aus.

1. Setzen von Servereinstellungen über Schlüssel-Wert-Paare, u. a. die Serverberechtigungen, durch Zuweisen einer Policy-Datei,

2. Zuweisen eines *Keystore*, der eine selbstunterzeichnete Signatur und das Schlüsselpaar für die SSL-Verschlüsselung enthält, sowie des Passworts für den Zugriff auf den *Keystore*,
3. Setzen des *RMISecurityManager*,
4. Einrichten von SSL-Sockets (*SslRMIClientSocketFactory, SslRMIServerSocketFactory*),
5. Erzeugen der RMI-Registrierung und setzen das Standardports 1099,
6. Erzeugen und exportieren des Remote-Objekts (Stub) in die Registry unter einem Alias,
7. Umleiten des Ausgabe- und Fehlerstroms in separate Dateien zur besseren Nachverfolgung bei auftretenden Fehlern.

Der komplette Quellcode der Klasse *FLServer* ist im Anhang (Abschnitt D.1 auf Seite 241) aufgeführt.

Implementierung der RMI-Schnittstelle: Die Klasse *FLRmiImpl* enthält die Implementierung der RMI-Schnittstelle. Vom *FL-Server* erhält sie bei der Instanziierung die Referenz auf das Feature-Logic-Objekt, die sie im Attribut *m_featureLogic* speichert, denn alle wichtigen Methoden benötigen einen Zugriff auf die Feature-Logic. Beispielhaft soll an dieser Stelle die Methode *askQuery()* aus Listing 4.3 erklärt werden.

```
1  public Set<String> askQuery(String query) throws
2          RemoteException, ParseException, SQLException,
3          FeatureLogicException {
4      Interpreter interpreter =
5          new Interpreter(new StringReader(query));
6      interpreter.setFeatureLogic(m_featureLogic);
7      SetDsc setDsc = interpreter.start();
8      ResultSetIterator elmIt =
9          m_featureLogic.resultSetIterator(setDsc);
10
11     Set<String> resultSet = new HashSet<String>();
12     if (elmIt == null){
13         return resultSet;
14     }
15     while (elmIt.hasNext())
16         resultSet.add(elmIt.next());
17     elmIt.close();
18     return resultSet;
19 }
```

Listing 4.3: Methode *askQuery()* der Klasse *FLRmiImpl*

Der im Methodenparameter *query* gespeicherte Feature-Term wird durch einen *StringReader* in einen Zeichenstrom umgewandelt und an den Feature-Logic-Interpreter übergeben.

Dieser interpretiert den Term und liefert das Ergebnis in einem Set-Descriptor (*SetDsc*) zurück. Mit Hilfe des *ResultSetIterator* wird in einer while-Schleife die Ergebnismenge in die Variable *resultSet* vom Typ `Set<String>` übertragen und an den aufrufenden Client zurückgegeben. Die vollständige Implementierung ist im Abschnitt D.2 auf Seite 244 zu finden.

4.2.3 Workspace

Übersicht: Der Workspace ist das zentrale Element der Systemarchitektur auf der Client-Seite. Er verwaltet organisatorische Daten und besitzt das anwendungsunabhängige Wissen für die Ausführung der verteilten Operationen. Einen ersten Eindruck vom grundlegenden Klassenentwurf des Workspace vermittelt Abbildung 4.4.

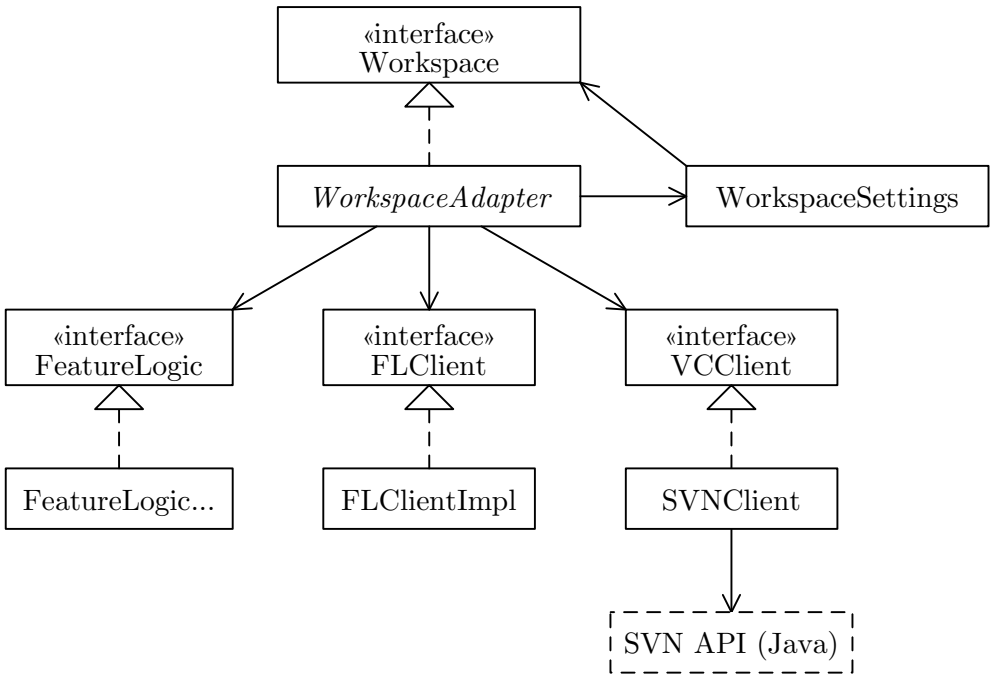


Abbildung 4.4: UML-Klassendiagramm: Workspace

Die Schnittstelle *Workspace* definiert alle diejenigen Methoden, die für die Unterstützung des verteilten Arbeitens auf Basis der Objektversionierung elementar sind. Sie lassen sich in sieben Gruppen einteilen.

1. Zugriff auf die lokale Feature-Logic-Instanz sowie den Feature-Logic- und Versionsverwaltungs-Client,
2. Generieren, Zuordnen und Verwalten der persistenten Objekt-IDs (POID),
3. Eintragen, Verwalten und Löschen von Objektabhängigkeiten (Bindungen),
4. Ausführen lokaler Operationen wie Dokument erzeugen, laden und speichern oder Wechseln des aktuellen Projekts,

5. Ausführen verteilter Operationen mit Serverzugriff wie Projekt beitreten, Holen, Übertragen (Commit) und Aktualisieren (Update),

6. Definieren von und Umschalten auf verschiedene Freigabestände,

7. Hilfsmethoden zur Unterstützung der vorgenannten Operationen.

An dieser Stelle ist es nicht sinnvoll, die sehr umfangreiche Schnittstelle zu erklären. Stattdessen soll sie im Laufe des Kapitels thematisch um die entsprechenden Methoden erweitert werden. Die abstrakte Klasse *WorkspaceAdapter* implementiert große Teile der Schnittstelle, so dass für eine spezielle verteilte Anwendung eine neue Klasse *Workspace-MyApplication* vom Adapter abgeleitet werden kann, die nur wenige Methoden hinzufügen muss. Im Wesentlichen betrifft das die Operationen für das Erzeugen, Laden und Speichern von Dokumenten.

Die Klasse *WorkspaceSettings* steht der Klasse *Workspace* zur Seite und speichert den aktuellen Namen des Projekts, der Anwendung und des Dokuments. Da das Planungsmaterial in der Sandbox, wie im Abschnitt 4.5 auf Seite 130 beschrieben, strukturiert im Dateisystem abgelegt wird, ist diese Klasse unter anderem für die Existenzprüfung der Verzeichnisse und die Rückgabe der absoluten Pfade verantwortlich. Das Klassendiagramm ist in Abbildung C.1 dargestellt.

Für den technologieunabhängigen Zugriff auf die lokale Feature-Logic, die Feature-Logic auf dem Server und das Versionsverwaltungssystem stehen dem Workspace die drei Schnittstellen *FeatureLogic, FLClient* und *VCClient* zur Verfügung, wie sie im Klassendiagramm in Abbildung 4.4 zu sehen sind.

FLClient: Die Schnittstelle *FLClient* lehnt sich stark an die RMI-Schnittstelle an, da sie das Gegenstück zu dieser darstellt. Im Vergleich der Abbildungen 4.2 und 4.5 ist das gut zu erkennen.

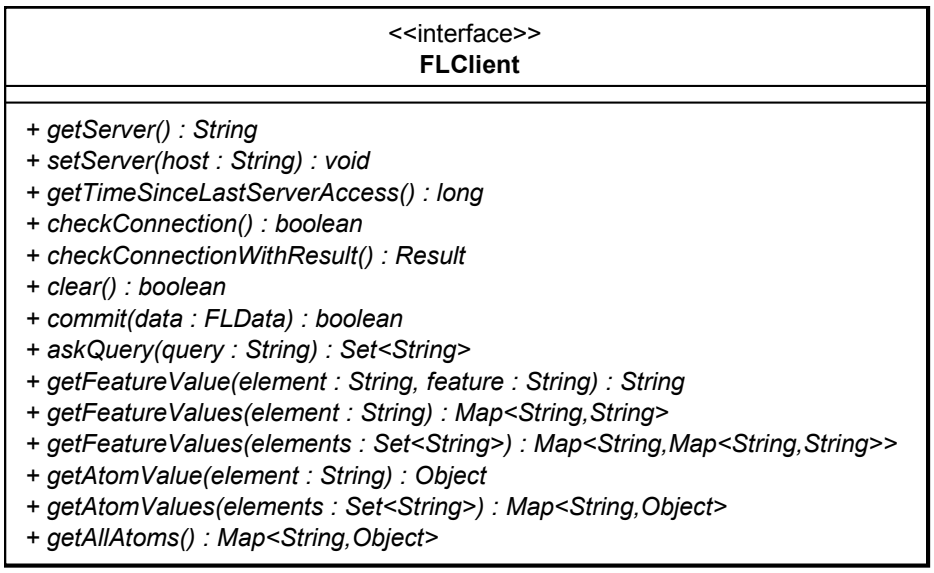


Abbildung 4.5: UML-Klassendiagramm: Schnittstelle *FLCLient*

Bevor eine SSL-Verbindung mit dem RMI-Server hergestellt werden kann, sind in der implementierenden Klasse *FLClientImpl* Einstellungen vorzunehmen (s. Listing 4.4). In dem vereinfachten Auszug des Konstruktors wird dem System zuerst eine Policy-Datei zugewiesen, die dem Client alle Berechtigungen erteilt. Als nächstes wird der sogenannte Truststore zugewiesen, der vorher von einem Administrator erstellt werden muss und das X.509-Zertifikat[2] des Servers enthält. Durch das SSL-Handshake-Protokoll authentifizieren sich Client und Server gegenseitig über den im Zertifikat enthaltenen öffentlichen Schlüssel des Servers. Mit diesem verschlüsselt der Client außerdem einen zufälligen und temporären Sitzungsschlüssel, der dann für die schnellere, symmetrische Verschlüsselung der Daten verwendet wird. Im letzten Schritt wird ein *RMISecurityManager* erzeugt und zugewiesen.

```
String clientPackage =
    FLClientImpl.class.getPackage().getName();
String clientPackagePath = clientPackage.replace(".", "/");
URL url = FLClientImpl.class.getResource("/" +
    clientPackagePath + "/rmi_client_unsecure.policy");
System.setProperty("java.security.policy",
    url.toExternalForm());
String pathTrustStore = System.getProperty("user.dir") +
    FileOperations.getFileSeparator() + "Client_Truststore";
System.setProperty("javax.net.ssl.trustStore", pathTrustStore);
if (System.getSecurityManager() == null)
    System.setSecurityManager(new RMISecurityManager());
```

Listing 4.4: Klasse *FLClientImpl*:

Der Aufruf einer entfernten Methode soll als Fortführung des Listings 4.3 erklärt werden. Der FLClient besitzt wie die RMI-Schnittstelle die Methode *askQuery()*, die in implementierter Form wie in Listing 4.5 aussieht und lediglich aus drei Zeilen besteht. In der ersten wird die private Methode *getRemoteObjectFromServer()*, die eine Referenz zum Remote-Objekt auf dem Server herstellt. Dazu sucht sie auf dem Server nach der RMI-Registry und lässt sich von dieser anhand des Service-Namens das Remote-Objekt zurückgeben, welches im Objektattribut *m_flRmiInterface* gespeichert wird. In der zweiten Zeile ruft *askQuery*() die entfernte Methode auf und gibt das erhaltene Set mit den Feature-Logic-Elementen zurück.

```
private boolean getRemoteObjectFromServer()
    if (m_flRmiInterface == null){
        // Getting registry and remote object from server
        Registry registry = LocateRegistry.getRegistry(m_host,
            FLRmiInterface.REGISTRY_PORT);
        Remote object =
            registry.lookup(FLRmiInterface.SERVICE_NAME);
```

[2]X.509 ist ein Standard der International Telecommunication Union (ITU) für digitale Zertifikate innerhalb einer Public-Key-Infrastruktur (PKI).

```
8           FLRmiInterface flRmiInterface = (FLRmiInterface)object;
9           if (flRmiInterface == null)
10              return false;
11          m_flRmiInterface = flRmiInterface;
12          return m_flRmiInterface.checkConnection();
13      }
14  }
15
16  public Set<String> askQuery(String query){
17      getRemoteObjectFromServer();
18      Set<String> set = m_flRmiInterface.askQuery(query);
19      return set;
20  }
```

Listing 4.5: Methode *askQuery()* der Klasse *FLClientImpl*

VCClient: Die Schnittstelle *VCClient* abstrahiert Methoden, die unabhängig vom verwendeten VCS gültig sind. Für die Umsetzung der Systemarchitektur wurde Subversion als modernes zentrales VCS gewählt (s. Abschnitt 2.3.3 auf Seite 37). Mit *SVNKit* existiert eine rein in Java geschriebene Bibliothek, die einen einfachen Zugang zum Subversion-API auf Client- und Serverseite bereitstellt (SVNKit, 2008). Die Klasse *SVNClient* implementiert die Schnittstelle *VCClient* und ist das Bindeglied zwischen ihr und SVNKit (s. Abbildung 4.4).

Aufgrund der Größe der Schnittstelle *VCClient* wurde sie in den Anhang D.4 auf Seite 250 verschoben. An erster Stelle stehen Dateinamen-Konstanten für abzuspeichernde Verwaltungsdaten, die an späterer Stelle erklärt werden. Dann folgen Set- und Get-Methoden für das Setzen und Auslesen der Zugangsdaten sowie projektspezifischer Informationen. Daran schließen sich Methoden zum Prüfen der Verbindung zum VCS-Server und der Sandbox- und Repositorystruktur an. Im abschließenden Hauptteil sind die wesentlichen Methoden zur Anwendung des Versionsverwaltungssystems enthalten.

Der Zugriff auf Subversion erfolgt verschlüsselt über SSH. Neben seinem eigenen Protokoll `svn://` definiert Subversion die Variante mit SSH und leitet es mit `svn+ssh://` ein. In SVNKit ist für eine erfolgreiche Verbindung ein Authentifizierungsmanager zu erzeugen, dem die gültigen Zugangsdaten zu übergeben sind.

4.3 Persistente Speicherung transienter Objektmodelle

4.3.1 Verwaltung der POIDs

Problem: Jedem in der Anwendung erzeugten Objekt muss bei der persistenten Speicherung eine POID zur projektweit eindeutigen Identifikation zugewiesen werden, die sich auch bei späterer Bearbeitung nicht mehr ändern darf. Anwendungen bzw. Interpreter ordnen dagegen jedem transienten Objekt einen Identifikator zu, der nur innerhalb des

Adressbereichs des Programms oder der virtuellen Maschine eindeutig ist und mit dem Löschen des Objekts aus dem Arbeitsspeicher verfällt.

Lösungsansatz: Zur Lösung des Problems wird im *WorkspaceAdapter* eine bijektive Map eingeführt, die jeder POID eines geladenen Objekts eineindeutig die zugehörige transiente ID zuweist. So kann sowohl zu einer POID das zugehörige Objekt als auch von einem Objekt die entsprechende POID schnell ermittelt werden. Da aktuelle Programmiersprachen in der Regel keine bijektiven Maps unterstützen, wird eine eigene Klasse entwickelt, die intern zwei Maps verwaltet. Die erste speichert wie bisher Schlüssel-Wert-Paare und die zweite zusätzlich Wert-Schlüssel-Paare (s. Abbildung 4.6). Die Klasse stellt die Konsistenz sicher, indem immer in beiden Maps zueinandergehörende Paare eingetragen oder gelöscht werden.

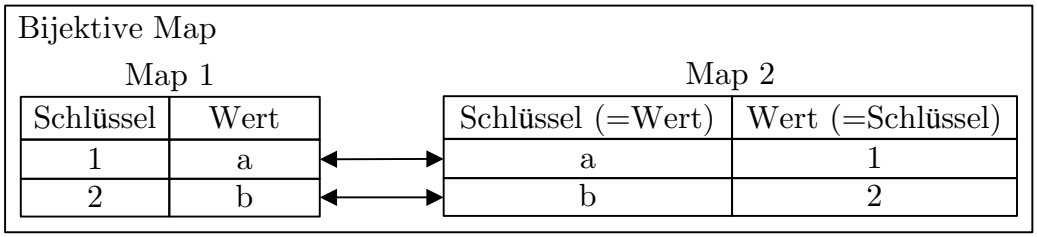


Abbildung 4.6: Schematische Umsetzung einer bijektiven Map

POIDs werden für neu erzeugte Objekte erst dann vergeben, wenn das Datenmodell persistent in die Sandbox gespeichert wird. Die POID dient dabei als Bezeichnung für den Speicherort eines Objekts, was zum Beispiel eine Datei sein kann. Beim Laden wird nach der Erzeugung des Objekts das Paar aus POID und transienter ID in die bijektive Map eingetragen, so dass beim nächsten Speichern die passende POID zu einem transienten Objekt gefunden werden kann.

Workspace: In der Klasse *WorkspaceAdapter* wird ein Attribut *m_persistentIdMap* eingeführt, das die bijektive Map zur Verwaltung der POIDs vorhält. Weiterhin wird die Schnittstelle *Workspace* um Methoden, wie im nächsten Listing zu sehen, erweitert. Die ersten zwei Methoden liefern für ein Objekt entweder eine schon existierende POID zurück oder erzeugen eine neue und tragen diese in die Map ein. *isPoid*() prüft, ob eine gegebene POID existiert, für die dann im positiven Fall das Objekt mit *getObjectFromPoid*() abgerufen werden kann. Die Prüfung, ob ein Objekt eine POID besitzt, und die anschließende Ermittlung decken die Methoden *objectHasPoid*() und *getPoid*() ab. Als letzte Methode löscht *clearPoidMap*() im Bedarfsfall den kompletten Inhalt der Map `m_persistentIdMap`.

```
1 public String getOrGenerateNewPoid(Object object)
2     throws WorkspaceException;
3 public String getOrGenerateNewPoid(Object object,
4     String prefix) throws WorkspaceException;
5 public boolean isPoid(String poid);
6 public Object getObjectFromPoid(String poid)
```

```
7       throws WorkspaceException;
8   public boolean objectHasPoid(Object object);
9   public String getPoid(Object object);
10  public void clearPoidMap();
```

Listing 4.6: Schnittstelle *Workspace*: Methoden zur Verwaltung von POIDs

4.3.2 Automatische Serialisierung in XML-Dateien

Funktionsweise: Für objectVCS[3] wurde ein XML-Serialisierer entwickelt, der in die Java-Serialisierung eingebettet wurde und dadurch beliebige Objektmodelle, ausgehend von einem Wurzelobjekt, mit allen Referenzen persistent speichern kann. Jedem Objekt ist eine XML-Datei mit der POID als Dateiname zugeordnet. Beim Deserialisieren werden die transienten Objekte wieder in ihrem Ursprungszustand wiederhergestellt.

Beispiel 4.1: XML-Serialisierer: Verwaltung der POIDs

Die Zuordnung der persistenten POIDs zu den transienten IDs der Objekte illustriert Abbildung 4.7 an einem Beispiel. Nach dem Start der Anwendung erzeugt der Bearbeiter zwei Objekte, die die transienten IDs `1` und `2` besitzen. Die Map `m_persistentIdMap` ist zu diesem Zeitpunkt noch leer. Erst beim Speichern in das Dokument D_1 wird jedem Objekt eine POID zugewiesen und in der Map zusammen mit der ID gespeichert. Objekt `1` erhält die POID `a` und Objekt `2` die POID `b`. Außerdem werden die XML-Dateien `a.xml` und `b.xml` angelegt, die den Zustand der Objekte dauerhaft speichern. Sollte die Anwendung zwischenzeitlich geschlossen, erneut gestartet und das Dokument geladen worden sein, stellt sich der gleiche transiente Zustand wie nach dem Speichern ein (Zeitpunkt 2). Während der Bearbeitung löscht der Nutzer nun das Objekt `2` und fügt ein neues mit der ID `3` ein. Der Inhalt der Map ändert sich erst nach dem Speichern, indem das Paar `(2,b)` ausgetragen und das Paar `(3,c)` eingetragen wird. Im Dateisystem befinden sich jetzt die Dateien `a.xml` und `c.xml`. Zeitpunkt 4 zeigt den transienten Zustand nach dem Speichern bzw. einem erneuten Laden.

Merkmale: Das Datenmodell einer Anwendung enthält nicht nur die für den Nutzer sichtbaren Objekte, sondern auch solche, die für die Verwaltung notwendig sind. So wird für vorhandene Arrays, Maps und Sets je eine eigene Datei angelegt. In Abhängigkeit von der Umsetzung entsteht eine vielfache Anzahl von Dateien gegenüber der Anzahl von Nutzerobjekten. Für große Modelle hat das eine negative Auswirkung auf die Leistung bei lokalen Lese-/Schreibzugriffen und bei der Ausführung verteilter Operationen, wie Commit oder Update.

Ein Änderung des Klassenschemas des Datenmodells in der Anwendung verhindert die Deserialisierung älterer persistenter Modelle, die dadurch nicht aufwärtskombatibel sind und einem Grundsatz für Dateiformate widersprechen (s. Seite 10). Die Bindung an das

[3] s. Abschnitt 2.4.4

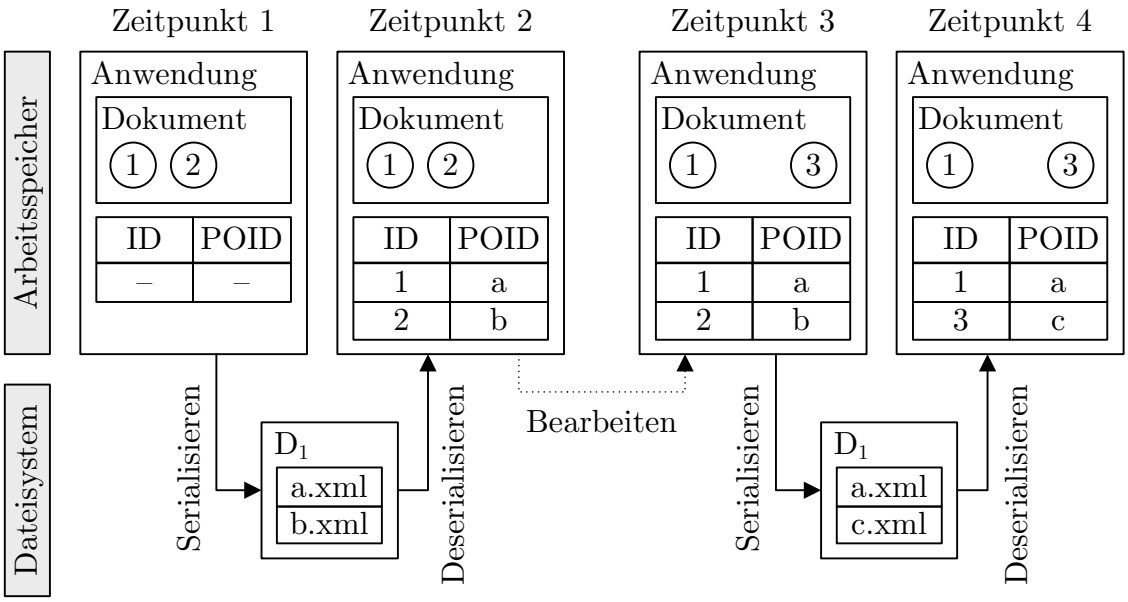


Abbildung 4.7: XML-Serialisierer: Verwaltung der POIDs

Klassenschema weist noch einen weiteren Nachteil auf, da diese Modelle nicht ohne Weiteres in andere Anwendungen importiert werden können. Bevor der nächste Ansatz vorgestellt wird, sollen nochmal die Vor- und Nachteile der automatischen XML-Serialisierung aufgeführt werden.

Vorteile

- Anwendbar für beliebige Objektmodelle
- Keine nachfolgenden Implementierungen notwendig

Nachteile

- Es werden Objekte serialisiert, die für die Versionierung uninteressant sind.
- Eine hohe Anzahl von XML-Dateien verschlechtert die Leistung bei Lese- und Schreibzugriffen.
- Die Traversierung großer Objektmodelle ist speicheraufwändig.
- Eine Änderung des Klassenschemas verhindert das Laden älterer Modelle.
- Die Modelle können nicht in andere Anwendungen importiert werden.

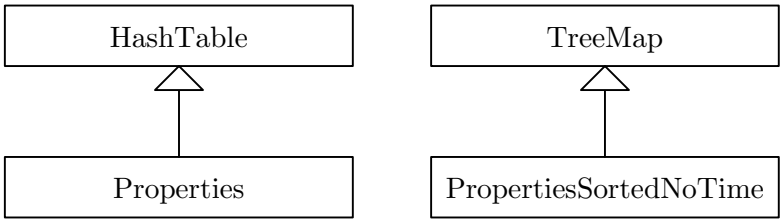
4.3.3 Manuelle Serialisierung in Textdateien

Speicherformat: Bis auf den zweiten lassen sich alle oben genannten Nachteile durch die manuelle Serialisierung in Textdateien ausräumen. Wie die Daten in den Textdateien strukturiert abgelegt werden, ist für das Verfahren unerheblich. Einzige Voraussetzung

ist, dass die Serialisierung desselben Modells die gleichen Dateien mit jeweils unverändertem Inhalt erzeugt, da das Versionsverwaltungssystem anhand der Prüfsumme eine Änderung der Dateien erkennt. Die im Folgenden vorgestellten Varianten finden häufig für die Speicherung beliebiger Daten Verwendung und eignen sich auch für die Speicherung von Attributwerten eines Objekts.

- **Folge von Werten:** Wenn nur die Attributwerte abgespeichert werden sollen, muss die Reihenfolge beim Speichern und späteren Einlesen bekannt sein, damit die Werte den richtigen Attributen zugeordnet werden. Die Trennung der Werte kann durch fast beliebige Zeichen – wie Leerzeichen, Komma, Tabulator, Zeilenumbruch u. a. – erfolgen, die aber von den Werten unterscheidbar sein müssen.

- **Schlüssel-Wert-Paare:** Dieses häufig verwendete Format speichert Daten in Zeilen mit der Form *Schlüssel=Wert* ab, wobei jeder Schlüssel nur einmal Verwendung finden darf. Java stellt zu diesem Zweck die Klasse *Properties* zur Verfügung, die auf der *HashTable*-Implementierung beruht (s. Abbildung 4.8a). Da die Schlüssel in zufälliger Reihenfolge in die Datei geschrieben werden, ändern sich die Dateiprüfsummen trotz gleicher Objektzustände. Außerdem wird ein Zeitstempel vom Zeitpunkt des Speicherns am Anfang der Datei eingefügt. Beide Fakten stehen der Verwendung für die Objektversionierung im Wege, weshalb eine eigene Klasse *PropertiesSortedNoTime* geschrieben wurde, die von der Klasse *TreeMap* mit inhärenter Schlüsselsortierung abgeleitet ist (s. Abbildung 4.8b).



(a) Ohne Schlüsselsortierung, mit Zeitstempel

(b) Mit Schlüsselsortierung, ohne Zeitstempel

Abbildung 4.8: Klassen für die Speicherung von Schlüssel-Wert-Paaren und einer Serialisierungsmöglichkeit in Textdateien

- **XML:** Die Struktur innerhalb von XML-Dateien kann vom Programmierer frei definiert und die Grammatik in einer DTD festgeschrieben werden. Java enthält seit Version 1.4 das Java API for XML Processing (JAXP) zum Schreiben und Lesen von XML-Dateien. Sofern die oben genannten Bedingungen eingehalten werden, ist diese Variante uneingeschränkt einsetzbar.

Umsetzung: Im Gegensatz zur automatischen Serialisierung muss bei der manuellen Serialisierung für eine spezielle Anwendung vorher festgelegt werden, welche Objekte für eine spätere vollständige Rekonstruktion des Datenmodells gespeichert werden müssen.

Die Dateianzahl lässt sich dabei teilweise erheblich reduzieren. Jedoch muss für jede Klasse entweder ein Stück Quellcode für das Schreiben und Lesen eingefügt oder an externer Stelle implementiert werden. Für die klasseninterne Variante bietet sich in Java die Schnittstelle *Externalizable* an, die die zwei Methoden *writeExternal*() und *readExternal*() vorschreibt (s. Seite 16).

Wenn die Klassen nicht verändert werden können oder sollen, müssen sie von außerhalb serialisiert werden. Für ein einheitliches Vorgehen bietet sich die Definition einer Schnittstelle an, die im Listing 4.7 unter dem Namen *IOObjectHandler* aufgeführt ist. Sie umfasst vier Methoden, von denen die erste die behandelte Klasse zurückliefert. Die folgenden drei Methoden dienen zum Schreiben, Laden und Importieren von Objekten der angegebenen Klasse. Der Parameter *file* enthält ein File-Objekt mit dem Ort der Textdatei und der Parameter *context* zum Beispiel den Anwendungskernel, der für die Durchführung der Operationen zuständig ist.

```
1  public interface IOObjectHandler<E> {
2      public Class<E> getHandledClass();
3
4      public void store(E object, File file, Object context)
5              throws IOException;
6      public E load(File file, Object context)
7              throws IOException;
8      public E import(File file, Object context)
9              throws IOException;
10 }
```

Listing 4.7: Schnittstelle *IOObjectHandler*

Angewandt auf die fiktive Klasse *MyClass* könnte die Implementierung des IOObject-Handlers unter Verwendung der neue Properties-Klasse für Schlüssel-Wert-Paare folgendermaßen in der Grundstruktur aussehen.

```
1  public class IOObjectHandlerMyClass implements
2          IOObjectHandler<MyClass>{
3      public Class<MyClass> getHandledClass(){
4          return MyClass.class;
5      }
6      public void store(E object, File file, Object context)
7              throws IOException{
8          PropertiesSortedNoTime props =
9              new PropertiesSortedNoTime();
10         // Das Objekt props mit Schluessel-Wert-Paaren fuellen
11         // ...
12         FileOutputStream out = new FileOutputStream(file);
13         props.store(out);
14         out.close();
15     }
```

```
16      public E load(File file, Object context)
17              throws IOException{
18          PropertiesSortedNoTime props =
19              new PropertiesSortedNoTime();
20          FileInputStream in = new FileInputStream(file);
21          props.load(in);
22          in.close();
23          // Objekt myObject der Klasse MyClass erzeugen
24          // ...
25          return myObject;
26      }
27      public E import(File file, Object context)
28              throws IOException;
29          // Entspricht im Wesentlichen der Load-Methode, mit
30          // dem Unterschied, dass das Objekt in der Anwendung
31          // dem transienten Datenmodell hinzugefuegt wird.
32      }
33  }
```

Listing 4.8: Klasse *IOObjectHandlerMyClass*

Die Namen der erzeugten Textdateien setzen sich aus der POID des serialisierten Objekts und einer frei wählbaren Dateiendung zusammen. Vorgeschlagen wird hierfür die Dateiendung *.obj* als Hinweis auf die Objektversionierung. Zusammenfassend sollen noch einmal die Vor- und Nachteile der manuellen Serialisierung in Textdateien aufgeführt werden.

Vorteile

- Es werden nur Objekte serialisiert, die für die Versionierung interessant sind.
- Keine aufwändige Traversierung großer Objektmodelle.
- Eine Änderung des Klassenschemas verhindert nicht zwangsläufig das Laden älterer Modelle.
- Die Modelle können auch in andere Anwendungen importiert werden.

Nachteile

- Für jede zu serialisierende Klasse muss eine zusätzliche Implementierung vorgenommen werden.
- Eine immer noch hohe Anzahl an Dateien bei großen Datenmodellen verschlechtert die Leistung bei Lese- und Schreibzugriffen im Dateisystem.

4.3.4 Manuelle Serialisierung in ein ZIP-Archiv

Problem: Aktuelle Dateisysteme können zwar eine große Anzahl von Dateien verwalten, jedoch sinkt die Leistung bei Lese-/Schreibzugriffen erheblich, wenn innerhalb einer Operation auf viele kleine Dateien zugegriffen wird. Dieses Defizit wirkt sich auch negativ

auf die zuvor beschriebene manuelle Serialisierung eines Objektmodells in viele einzelne Textdateien aus.

Lösungsansatz: Zur Lösung des Problems muss die Dateianzahl reduziert werden. In der Praxis stehen dafür Datenkompressionsprogramme zur Verfügung, die eine nach oben begrenzte Anzahl von Dateien zu einer Archivdatei zusammenfassen und gleichzeitig den Speicherplatz der Daten mit Hilfe von verlustfreien Kompressionsalgorithmen reduzieren. Bevor dieses Verfahren auf die Serialisierung von Objekten angewandt werden kann, müssen erst die Besonderheiten der Objektversionierung auf Basis von textbasierten Versionsverwaltungssystemen näher betrachtet und folgende Forderungen erfüllt werden.

1. Wird ein unverändertes Datenmodell mehrmals serialisiert, darf sich der Inhalt im Dokumentverzeichnis nicht ändern, das heißt, die Archivdatei an sich darf sich nicht ändern. Andernfalls würde das Versionsverwaltungssystem einen neuen Zustand des persistenten Objektmodells erkennen, was nicht der Fall wäre.

2. Wenn sich nur eine der zu komprimierenden Dateien geringfügig ändert, kann sich bei einer erneuten Kompression je nach Format und Algorithmus ein Großteil das Archivs im Gegensatz zum vorher gespeicherten ändern. VCS sind aber darauf ausgelegt, kleine Änderungen am Inhalt zu erkennen und effizient zu speichern. Deshalb sollten kleine Änderungen in den Ausgangsdateien auch nur kleine Änderungen in der Archivdatei zur Folge haben.

ZIP-Format: Bei der Wahl des Datenkompressionsformats bietet sich das weit verbreitete ZIP-Format an, da es von Java über das Paket *java.util.zip* unterstützt wird und die oben genannten Forderung damit realisiert werden können. Es wurde von Phil Katz im Jahr 1989 entworfen und als freies Format veröffentlicht, wodurch es sich auch aufgrund besserer Eigenschaften gegenüber damaligen Konkurrenzformaten weltweit verbreitet hat und mittlerweile von vielen Betriebssystemen nativ unterstützt wird. Die Spezifikation ist auf der Webseite der Firma PKWARE Inc. frei verfügbar (ZIP, 2007).

Grundsätzlich wird jede Datei separat als ZIP-Eintrag behandelt und komprimiert. Es stehen mehrere Kompressionsalgorithmen zur Verfügung, von denen der DEFLATE-Algorithmus der gebräuchlichste ist. Außerdem ist das umkomprimierte Speichern möglich, was als STORE-Methode bezeichnet wird. Jeder ZIP-Eintrag wird durch einen *Local file header* eingeleitet, der die in Tabelle 4.1 beschriebenen Metadaten enthält.

Der CRC-32-Wert ist das Ergebnis einer Prüfsummenberechnung, die als *Cyclic Redundancy Check*[4] bezeichnet und auf Basis einer Polynomdivision durchgeführt wird (Wikipedia, 2008b). Prüfsummen dienen der einfachen Fehlererkennung bei der Datenübertragung und -speicherung, jedoch können sie nicht wie Hashfunktionen[5] die Datenintegrität sicherstellen. Der CRC-32-Wert wird später beim Vergleich von Objektmodellen noch von Bedeutung sein (s. Abschnitt 4.8.2 auf Seite 159).

Im Anschluss an die Metadaten folgt der eigentliche Dateiinhalt in komprimierter oder unkomprimierter Form. Die Reihenfolge der ZIP-Einträge ist nicht vorgeschrieben, was das

[4] CRC = Cyclic Redundancy Check ⟨engl., „Zyklische Redundanzprüfung“⟩
[5] z. B. MD5 und SHA-1

Beschreibung	Größe	Anmerkung
local file header signature	4 Bytes	(0x04034b50)
version needed to extract	2 Bytes	
general purpose bit flag	2 Bytes	
compression method	2 Bytes	
last mod file time	2 Bytes	
last mod file date	2 Bytes	
crc-32	4 Bytes	
compressed size	4 Bytes	
uncompressed size	4 Bytes	
file name length	2 Bytes	
extra field length	2 Bytes	

Tabelle 4.1: Local file header eines ZIP-Eintrags

Hinzufügen und Löschen von Dateien wesentlich vereinfacht. Am Ende eines ZIP-Archivs befindet sich das zentrale Verzeichnis (*central directory*), das alle ZIP-Einträge referenziert und ihnen zusätzliche Informationen, wie den Dateinamen, zuordnet. Die Reihenfolge der ZIP-Einträge muss nicht mit der im zentralen Verzeichnis übereinstimmen.

Umsetzung: Für die Verwendung von ZIP-Archiven für die Objektversionierung müssen die beiden erläuterten Forderungen erfüllt werden. Die erste Forderung lässt sich einhalten, wenn zum einen die Objekte in der Reihenfolge sortiert nach ihren POIDs serialisiert werden und zum anderen immer der gleiche Zeitstempel für alle ZIP-Einträge verwendet wird. Das Weglassen der Kompression ist die zweckmäßigste und einfachste Lösung zur Erfüllung der zweiten Forderung. Durch die Kombination der manuellen Serialisierung von Textdateien mit der anschließenden Speicherung in ein umkomprimiertes ZIP-Archiv ergibt sich ein Verfahren zur Speicherung von Objektmodellen, das sehr gut zur Charakteristik eines VCS passt und trotzdem eine gute Leistung bei großen Datenmodellen verspricht.

Im Paket *java.uti.zip* existieren vier Klassen, die für die Behandlung von ZIP-Archiven zuständig sind. Für des Schreiben wird ein *ZipOutputStream* erzeugt, der im Konstruktor ein *FileOutputStream*-Objekt mit dem Verweis auf eine Datei entgegennimmt. Die Methode *setMethod()* setzt danach den gewünschten Kompressionsalgorithmus, in diesem Fall auf STORED. Für jeden ZIP-Eintrag wird ein *ZipEntry*-Objekt angelegt, die Metadaten gesetzt und die eigentlichen Dateidaten an einen Strom übergeben. Dieser Vorgang wird für jede Datei wiederholt bis zuletzt der *ZipOutputStream* geschlossen wird.

Für das Auslesen eines ZIP-Archivs muss ein *ZipFile*-Objekt angelegt werden, das über die Methode *entries()* alle enthaltenen ZIP-Einträge zurückliefert. Über die Methode *getInputStream(ZipEntry zipEntry)* erhält man vom *ZipFile* einen *InputStream*, der zum Auslesen des Dateiinhalts dient.

Die Schnittstelle aus Listing 4.7 kann leicht an die Serialisierung in ein ZIP-Archiv angepasst werden, indem die *load-, store-,* und *import*-Methode entsprechend geändert werden.

```
1  public interface IOObjectHandler<E> {
2      public Class<E> getHandledClass();
3
4      public long store(E object, ZipOutputStream zipOut, String
5              entryName, Object context) throws IOException;
6
7      public E load(ZipFile zipFile, String entryName,long[] crc,
8              Object context)  throws IOException;
9
10     public E importObject(ZipFile zipFile, String entryName,
11             Object context) throws IOException;
12 }
```

Listing 4.9: Schnittstelle *IOObjectHandler* für die ZIP-Serialisierung

Ebenso sind in der Beispielklasse *IOObjectHandlerMyClass* die Methoden zu aktualisieren. Der Quellcode in den Zeilen 10 bis 17 zeigt des Schreiben eines ZIP-Eintrags. Die *Properties*-Klasse besitzt die Methode *crc()*, die die Größe und Prüfsumme des *ZipEntry* berechnet. Das gesetzte Datum entspricht dem `29.02.2008 12:00:00 UTC`. Über die Methode *store()* serialisiert sich das Properties-Objekt in den Ausgabestrom des *ZipEntry*. Das Laden geschieht in umgekehrter Art und Weise, indem das Properties-Objekt mit den Daten des Eingabestroms aus dem *ZipEntry* gefüllt wird.

```
1  public class IOObjectHandlerMyClass implements
2          IOObjectHandler<MyClass>{
3      ...
4      public void store(E object, ZipOutputStream zipOut, String
5              entryName, Object context) throws IOException{
6          PropertiesSortedNoTime props =
7              new PropertiesSortedNoTime();
8          // Das Objekt props mit Schluessel-Wert-Paaren fuellen
9          // ...
10         ZipEntry zipEntry = new ZipEntry(entryName);
11         long[] result = props.crc();
12         zipEntry.setSize(result[0]);
13         zipEntry.setCrc(result[1]);
14         zipEntry.setTime(1204286400968L);
15         zipOut.putNextEntry(zipEntry);
16         props.store(zipOut);
17         zipOut.closeEntry();
18         return;
19     }
20     public E load(ZipFile zipFile, String entryName,long[] crc,
21             Object context) throws IOException{
22         ZipEntry zipEntry = zipFile.getEntry(entryName);
23         crc = new long[1];
```

```
24          crc[0] = zipEntry.getCrc();
25          PropertiesSortedNoTime props =
26              new PropertiesSortedNoTime();
27          props.load(zipFile.getInputStream(zipEntry));
28          // Objekt myObject der Klasse MyClass erzeugen
29          // ...
30          return myObject;
31      }
32      ...
33  }
```

Listing 4.10: Klasse *IOObjectHandlerMyClass* für die ZIP-Serialisierung

An zentraler Stelle im *Workspace* wird das Schreiben und Lesen von Objektmodellen gesteuert. Das allgemeine Vorgehen für das Schreiben von Objekten demonstriert folgendes Listing. In Zeile 4 wird die Methode STORED ohne Kompression gesetzt. Die Menge *objectPoids* muss sortiert sein, damit die ZIP-Einträge in einer festgelegten Reihenfolge geschrieben werden. Die Klasse *IOUtils* verwaltet alle *IOObjectHandler*, die dort zur Startzeit des Programms für jede zu serialisierende Klasse zu registrieren sind.

```
1  File documentFile = new File("Pfad/MyDocument.zip");
2  FileOutputStream fos = new FileOutputStream(documentFile);
3  ZipOutputStream zipOut = new ZipOutputStream(fos);
4  zipOut.setMethod(ZipOutputStream.STORED);
5  for (String poid : objectPoids){
6      Object obj = getObjectFromPoid(poid);
7      IOObjectHandler ioObjectHandler =
8          IOUtils.getIObjectHandler(obj.getClass());
9      ioObjectHandler.store(comp, zipOut, poid + ".obj", krnl);
10 }
11 zipOut.close();
```

Listing 4.11: Workspace: Serialisierung in ein ZIP-Archiv

Für das Laden werden erst alle Namen der ZIP-Einträge ausgelesen und in einem Set zwischengespeichert. Danach wird für jeden Eintrag die Klasse, die in der POID kodiert ist, bestimmt, der passende IOObjectHandler geholt und das Objekt im Arbeitsspeicher angelegt.

```
1  ZipFile zipFile = new ZipFile(documentFile);
2  Set<String> entryNames = new HashSet<String>();
3  Enumeration<? extends ZipEntry> e = zipFile.entries();
4  while (e.hasMoreElements())
5      entryNames.add(e.nextElement().getName());
6
7  for (String entryName : entryNames){
8      Class clazz = FLUtilities.getClassOfObjectPoid(entryName);
```

```
9       IOObjectHandler ioObjectHandler =
10          IOUtils.getIObjectHandler(clazz);
11      Object obj =
12          ioObjectHandler.load(zipFile, entryName, crc, krnl);
13  }
14  zipFile.close();
```

Listing 4.12: Workspace: Deserialisierung aus einem ZIP-Archiv

Verwaltung der Versionsnummern: Wenn Objekte jeweils in einer Datei gespeichert werden, ordnet das Versionsverwaltungssystem jeder Datei in der Sandbox nach einem Commit eine Versionsnummer zu und verwaltet die Dateiversionen auf dem Server. Mit dem Serialisieren aller Objekte eines Dokuments in ein ZIP-Archiv entfällt zwangsläufig diese Funktionalität. Die Objektversionsnummern werden aber zur Modellierung in der Feature-Logic benötigt. Das Ableiten der Objektversionsnummern aus der Dokumentversionsnummer wäre falsch, da eine Objektversion in mehreren Versionen des Dokuments vorkommen kann.

Die Objektversionsnummern müssen demnach unabhängig vom Versionsverwaltungssystem vorgehalten werden. Zur Speicherung bietet sich eine Textdatei mit Schlüssel-Wert-Paaren an, die jeder Datei eine Versionsnummer zuweist, parallel zum ZIP-Archiv des Dokuments liegt und zusammen mit der Dokumentversion auf dem VCS-Server gespeichert wird. Auch hier ist zu beachten, dass sich die Datei bei unverändertem Dokumentinhalt und mehrmaligem Speichern nicht ändert. Dies wird durch Verwendung der Klasse *PropertiesSortedNoTime* sichergestellt. Für ein neues Dokument mit drei Objekten könnte die unter dem Namen *revisions.txt* abgespeicherte Datei folgendermaßen aussehen.

```
1  KlasseA_Richter_POID1.obj=42
2  KlasseA_Richter_POID2.obj=42
3  KlasseB_Richter_POID3.obj=42
```

Listing 4.13: Verwaltung der Objektversionsnummern in der Datei *revisions.txt*

Die Versionsnummern neuer und geänderter Objekte werden mit der globalen VCS-Revisionsnummer des ZIP-Archivs gleichgesetzt, wobei die Datei *revisions.txt* erst bei Kenntnis dieser Nummer geschrieben kann. Dazu muss der Commit-Vorgang in vier Schritten durchgeführt werden.

1. Übertragen des Dokument-ZIP-Archivs, z. B. *MyDocument.zip*, an den VCS-Server, der dieser Datei eine globale Revisionsnummer zuweist. Für das Beispiel wäre das die 42.
2. Zuordnen der Revisionsnummer an neue sowie geänderte Objekte und Erzeugen bzw. Aktualisieren der Datei *revisions.txt*.
3. Übertragen von *revisions.txt* an den VCS-Server, die dann die nächsthöhere Revisionsnummer 43 zugewiesen bekommt, welche aber nicht weiter von Bedeutung ist.

4. Taggen des ZIP-Archivs und der *revisions.txt* mit einem gemeinsamen, vom Nutzer vorher angegebenen Tag, der später zum Beispiel für das Update verwendet wird.

Falls im Beispiel das Objekt mit der POID `"KlasseA_Richter_POID1"` bearbeitet und geändert wird, so ändert sich nach einem Speichern nur das ZIP-Archiv *MyDocument.zip*. Erst der zweite Schritt der Commit-Operation aktualisiert die Datei *revisions.txt*, die dann wie im Listing 4.14 aussieht. Das Objekt erhält demnach die neue Versionsnummer 44, die der Revisionsnummer des ZIP-Archiv gleicht.

```
1 KlasseA_Richter_POID1.obj=44
2 KlasseA_Richter_POID2.obj=42
3 KlasseB_Richter_POID3.obj=42
```

Listing 4.14: Datei *revisions.txt* nach Änderung eines Objekts

Platzbedarf im Dateisystem: Die kleinste Einheit eines Dateisystems ist ein Cluster, der beispielsweise für das in Microsoft Windows verwendete NTFS in der Regel 4 kByte groß ist. Einem Cluster darf höchtens eine Datei zugeordnet sein, so dass Dateien zwischen 1 Byte und 4096 Byte Größe den gleichen Speicherplatz im Dateisystem belegen. Dieser Effekt wird in der Informatik als interne Fragmentierung oder Verschnitt bezeichnet und wirkt sich bei vielen kleinen Dateien stärker aus als bei wenigen großen. Eine Datei mit 1 Byte erzeugt demzufolge eine interne Fragmentierung von 4095 Byte.

Vergleicht man die drei Serialisierungmethoden anhand eines praktischen Beispiels in Form eines CAD-Dokuments mit 11960 Zeichnungsobjekten, so ergibt sich der in Tabelle 4.2 aufgeführte Platzbedarf im Dateisystem NTFS. Es ist zu erkennen, dass von links nach rechts die Dateianzahl und der tatsächlich benötigte Bruttoplatzbedarf abnehmen. Im Fall des XML-Serialisierers steigt er fast um das Achtfache gegenüber dem Nettoplatzbedarf und bei der manuellen Serialisierung in Textdateien sogar um das Zwanzigfache, da alle Dateien kleiner als 4 kByte und im Schnitt nur ca. 200 Byte groß sind. Das ZIP-Archiv belegt zwar netto mehr Platz als die einzelnen Textdateien, was sich durch die unkomprimierte Form und die zusätzlichen Metadaten erklären lässt, real aber nur ein Achtel. Im Vergleich zu den XML-Dateien benötigt das ZIP-Archiv nur 2,7 % des Speicherplatzes.

	XML	Manuell in Textdateien	Manuell in ein ZIP-Archiv
Anzahl Dateien	50 923	11 983	1
Cluster	52 447	11 983	1 424
Platzbedarf netto	27 494 kB	2 373 kB	5 694 kB
Platzbedarf brutto	209 788 kB	47 932 kB	5 696 kB

Tabelle 4.2: Platzbedarf eines serialisierten CAD-Dokuments

Zusammenfassung: Trotz des erhöhten Implementierungs- und Verwaltungsaufwands lässt sich durch die Speicherung der Dokumente in einzelne ZIP-Archive die Serialisie-

rungsgeschwindigkeit wesentlich erhöhen, was durch die Zeitmessungen im nächsten Abschnitt bestätig wird. Nebenbei verringert sich durch den Übergang von vielen kleinen Datei zu einer großen der tatsächliche Platzbedarf im Dateisystem. Wie bei den beiden Serialisierungsmethoden zuvor sollen noch einmal die Vor- und Nachteile aufgeführt werden.

Vorteile

- Es werden nur Objekte serialisiert, die für die Versionierung interessant sind.
- Keine aufwändige Traversierung großer Objektmodelle.
- Eine Änderung des Klassenschemas verhindert nicht zwangsläufig das Laden älterer Modelle.
- Die Modelle können auch in andere Anwendungen importiert werden.
- Durch die Reduzierung der Dateianzahl auf zwei pro Dokument werden praxistaugliche Zeiten für das Speichern, Laden und Übertagen zum/vom Server erreicht.
- Der Speicherplatzbedarf ist gering.

Nachteile

- Für jede zu serialisierende Klasse muss eine zusätzliche Implementierung vorgenommen werden.

Zeitbedarf beim Speichern: Für den Vergleich der Zeiten wurde für jede Serialisierungsmethode eine bestimmte Anzahl von Objekten in der Anwendung CADEMIA erzeugt und danach das transiente Datenmodell in ein Dokument serialisiert. Die gemessenen Zeiten korrespondieren mit der Anzahl der erzeugten Dateien, was eindeutig für die Speicherung in ein ZIP-Archiv spricht. Beim XML-Serialisierer kommt noch das Traversieren des Objektmodells hinzu, dessen Leistung sich bei der vorliegenden Implementierung mit steigender Objektanzahl überproportional verschlechtert.

Anzahl Objekte	Automatisch in XML-Dateien			Manuell in Textdateien	Manuell in ZIP-Archiv
	Traversieren	Schreiben	Gesamt		
1000	9,7	12,8	22,5	2,6	0,56
2000	21,8	36,5	58,3	5,3	0,71
3000	46,3	51,4	97,7	7,5	1,1
4000	78,4	74,2	153	10,0	1,4
5000	–	–	–	–	1,7
10000	–	–	–	24,7	3,4

Tabelle 4.3: Serialisierungszeiten für die verschiedenen Methoden in [s]

Zeitbedarf beim Commit: In Tabelle 4.4 sind die benötigten Zeiten für eine Übertragung des Dokument zum VCS-Server ausschließlich für die manuellen Serialisierungsmethoden aufgeführt. Der Commit vieler kleiner Textdateien benötigt schon bei relativ wenigen Objekten praxisuntaugliche Zeiten. Es wurden deshalb keine Messungen bei Dokumenten mit mehr als 1000 Objekten vorgenommen. Der Commit eines ZIP-Archivs mit 50000 Objekten und einer Größe von ca. 45 MB ist nach 10 Sekunden abgeschlossen und damit erheblich schneller. Zu beachten ist aber, dass noch zusätzliche Zeit für die Übertragung von Daten zum Feature-Logic-Server hinzukommt, was im Abschnitt 4.9.2 auf Seite 165 diskutiert wird.

Anzahl Objekte	Manuell in Textdateien	Manuell in ZIP-Archiv
100	18,4	3,9
200	33,8	3,0
1000	329	2,8
10000	–	3,6
20000	–	5,8
30000	–	7,3
50000	–	9,6

Tabelle 4.4: Commit-Zeiten für die manuellen Serialisierungsmethoden in [s]

4.4 Erweiterung der Feature-Logic

4.4.1 Datums-Datentyp

Feature-Logic-Datentypen: Die Feature-Logic unterstützt die Datentypen *Integer, Long, Double* und *String* für atomare Elemente. Die Atome werden in der Atom-Tabelle nur in einer textuellen Form gespeichert, so dass es keine Unterscheidung zwischen den Datentypen gibt. Für eine eindeutige Ermittlung des Datentyps in der Feature-Logic müssen die atomaren Werte in eine vorher festgelegte Form überführt werden, wie sie beispielhaft in Tabelle 4.5 zu sehen sind.

Datum: Das Speichern eines Datums war bisher nicht explizit vorgesehen. Für die Verwaltung von Dokumenten ist es jedoch sinnvoll, diesen ein Datum zuzuordnen und einen Datentyp in der Feature-Logic einzuführen. Java speichert für ein Datum die Anzahl der vergangenen Tausendstelsekunden seit dem 1.1.1970 als *Long*-Wert ab. Um in der Feature-Logic ein Datum von einem *Long* zu unterscheiden, wird statt einem `L` die Zeichenkette `DATE` angehängt.

Datentyp	Beispiel	Feature-Logic	Unterscheidungsmerkmal
Integer	123	`123`	Ganzzahlen innerhalb des Wertebereichs
Long	1213000282109	`1213000282109L`	`L` am Ende
Double	5.34	`5.34`	Dezimalpunkt
String	Richter	`"Richter"`	Anführungszeichen
Date	09.06.2008 10:31:13	`1213000282109DATE`	`DATE` am Ende

Tabelle 4.5: Speicherung atomarer Werte in der Feature-Logic

4.4.2 Vergleichsoperatoren für atomare Werte

Ziel: (Firmenich, 2002) entwarf für die Feature-Logic einen Interpreter, mit dem sich Feature-Terme parsen und Anfragen an den zugrundeliegenden Datenspeicher stellen lassen. Für die Anfragen können nur Features und Elemente verwendet werden, jedoch nicht direkt atomare Werte. Für den Dialog *ProjektExplorer*, der im s. Abschnitt 4.8.1 auf Seite 152 beschrieben wird, sollen im Selektionsteil Constraints definiert werden können, die atomare Werte im Nummern- oder Datumsformat verwenden. Das erlaubt Einschränkungen, wie z. B.: „Gebe alle Dokumentversionen zurück, die nach dem 16.08.2008 um 16:00 Uhr angelegt wurden“.

Begriffe

Compiler (Aho u. a., 2002)**:** Ganz allgemein betrachtet, übersetzt ein Compiler ein Programm, das in einer Quellsprache geschrieben ist, in ein gleichwertiges Programm mit einer Zielsprache und gibt bei Auftreten von Fehlern entsprechende Meldungen aus.

Interpreter (Aho u. a., 2002)**:** Die Aufgabe eines Interpreters ähnelt der des Compilers, mit dem Unterschied, dass er kein Zielprogramm erzeugt, sondern die Operationen gleich ausführt.

Analyse der Quellsprache (Aho u. a., 2002)**:** Compiler und Interpreter analysieren die Quellsprache in drei Arbeitsschritten:

- **Lexikalische Analyse:** Ein Scanner[6] zerlegt den Ausdruck in Terminale und Nichtterminale, wobei Terminale für die Syntax[7] der Sprache von Bedeutung sind.
- **Syntaktische Analyse:** Ein Parser[8] prüft, ob der Ausdruck der Syntax der Sprache entspricht und erzeugt einen Syntaxbaum, der die Ausführung des Ausdrucks eindeutig beschreibt. Die Blätter des Baumes enthalten die Terminale und die inneren Knoten die Nichtterminale.

[6] scanner ⟨engl., „Abtaster, Leser“⟩
[7] syntaxis ⟨griech., „Zusammenordnung, Anordnung, Zusammenstellung“⟩
[8] parser ⟨engl., „Analysierer“⟩

- **Semantische Analye:** Der letzte Schritt besteht in der Prüfung der Semantik[9]. Die Semantik ist die *„Bedeutungslehre bzw. der Bedeutungsinhalt sprachlicher Gebilde“* (Fischer u. Hofer, 2008). Bei Programmiersprachen wird zum Beispiel geprüft, ob die Datentypen zu den Operatoren passen.

Scanner- und Parsergenerator: Scanner- und Parsergeneratoren sind Softwarewerkzeuge, die aus einer vorgegebenen Grammatik einer Sprache entweder den Scanner oder den Parser erzeugen. Die Grammatik G einer formalen Sprache ist ein 4-Tupel (N,T,P,S) mit N als Menge der Nichtterminale, T als Menge der Terminale, P als Menge der Grammatik-Produktionen und S als Startsymbol der Grammatik.

Backus-Naur-Form: Die Backus-Naur-Form (BNF) beschreibt durch reguläre Ausdrücke kontextfreie Grammatiken, indem sie lexikalische und syntaktische Regeln formuliert. Reguläre Ausdrücke stellen nach (Fischer u. Hofer, 2008) eine *„Syntax zur Beschreibung von Zeichenketten“* dar. Niklaus Wirth schuf mit der Erweiterten Backus-Naur-Form (EBNF) eine Erweiterung zur Syntaxbeschreibung für die von ihm entwickelte Programmiersprache Pascal. Inzwischen existieren mehrere Varianten der EBNF, die von der ISO auch in der Norm (ISO/IEC 14977, 1996) veröffentlicht wurde.

JavaCC: JavaCC (Java Compiler Compiler) ist ein Scanner- und Parsergenerator, der in Java geschrieben ist und für eine Grammatik, die in einer Datei mit der Endung *jj* definiert ist, den Scanner und Parser erzeugt. Die Grammatik muss in der EBNF vorliegen.

Umsetzung

Operatoren: Zusätzlich zu den in Tabelle 2.5 auf Seite 46 aufgeführten Operatoren, soll die Grammatik um die Operatoren aus Tabelle 4.6 erweitert werden. Das im Ziel genannte Beispiel einer Einschränkung des Datums würde dann mit dem Feature-Term `date>16.08.2008_16:00:00` beschrieben werden.

Operator	Nummer/Datum
gleich	=
ungleich	<>
kleiner als	<
kleiner gleich	<=
größer gleich	>=
größer als	>

Tabelle 4.6: Feature-Logic: Vergleichsoperatoren für atomare Werte

Der entsprechende Bereich in der Grammatik wird um die neuen Operatoren erweitert (s. Listing 4.15).

[9] semainein ⟨griech., „bezeichnen“⟩

```
TOKEN : /* OPERATORS */
{
    < SELECTION:                 ":"     >
|   < SELECTION_EQUAL:           "="     >
|   < SELECTION_NOT_EQUAL:       "<>"    >
|   < SELECTION_LESS:            "<"     >
|   < SELECTION_LESS_EQUAL:      "<="    >
|   < SELECTION_GREATER:         ">"     >
|   < SELECTION_GREATER_EQUAL:   ">="    >
|   < COMPLEMENT:                "~"     >
|   < DIFFERENCE:                "~~"    >
|   < NOT_DEFINED:               "^"     >
|   < EQUAL:                     "=="    >
|   < DIFFERENT:                 "!="    >
|   < IMPLICATION:               "->"    >
|   < EQUIVALENCE:               "<->"   >
|   < EXTRACTION:                "."     >
}
```

Listing 4.15: Grammatik der Feature-Logic: Erweiterung um Vergleichsoperatoren

Literale: In der bestehenden Grammatik fehlt bisher ein Literal für das Datum. Listing 4.16 enthält die Definition für das Literal, welches sich aus drei oder sechs Literalen, die je einem Bestandteil des Datums entsprechen, zusammensetzt. Das Verkettungszeichen `"|"` steht hier für *oder*.

```
...
  < DATE_LITERAL: < DAY > "." < MONTH > "." < YEAR >
      | < DAY > "." < MONTH > "." < YEAR > "_"
        < HOUR > ":" < MINUTE > ":" < SECOND >  >
|
  < #DAY: ["1"-"9"] | ["0"]["1"-"9"] | ["1"-"2"] ["1"-"9"] |
          ["3"] ["0"-"1"] >
|
  < #MONTH: ["1"-"9"] | ["0"]["1"-"9"] | ["1"] ["0"-"2"] >
|
  < #YEAR: ["2"] ["0"-"9"] ["0"-"9"] ["0"-"9"] >
|
  < #HOUR: ["0"-"9"] | "0" ["0"-"9"] | ["1"]["0"-"9"] |
           ["2"] ["0"-"3"] >
|
  < #MINUTE: ["0"-"5"] ["0"-"9"] >
|
  < #SECOND: ["0"-"5"] ["0"-"9"] >
}
```

Listing 4.16: Grammatik der Feature-Logic: Erweiterung um Operatoren

Produktionen: Die Produktionen beschreiben Nichtterminalsymbole mit regulären Ausdrücken und geben vor, wie der Parser vorzugehen hat. JavaCC erlaubt die Vermischung von Produktionen mit Java-Quellcode, wie es im Listing 4.17 zu sehen ist. Trifft der Parser auf ein Date-Literal, speichert er es als `String` in einem Token x. Der folgende Java-Quellcode erzeugt aus dem Literal ein Date-Objekt.

```
Date dateLiteral() : {
    indent("DateLiteral");
    Token x;
}
{
    x = <DATE_LITERAL> {
        String dateString = x.image;
        DateFormat dfDate;
        Date date;
        try{
            if (dateString.contains("_")){
                dateString = dateString.replace("_", " ");
                dfDate = DateFormat.getDateTimeInstance(
                    DateFormat.MEDIUM, DateFormat.MEDIUM);
                date = new FormattedDate(
                    dfDate.parse(dateString).getTime());
            }
            else {
                dfDate = DateFormat.getDateInstance(
                    DateFormat.MEDIUM);
                date = new FormattedDate(
                    dfDate.parse(dateString).getTime());
            }
        } catch(java.text.ParseException e) {
            throw new ParseException(e.getMessage());
        }
        return date;
    }
}
```

Listing 4.17: Produktion zur Umwandlung eines Datumsliterals in ein Java-Objekt

Erweiterung der Feature-Logic: Nach dem erfolgreichen Parsen eines Ausdrucks ruft der Feature-Logic-Interpreter Methoden der Schnittstelle *Feature-Logic* auf. Zur Ermittlung von Elementen mit den neuen Vergleichsoperatoren müssen noch Methodensignaturen zur Schnittstelle hinzugefügt werden. Listing 4.18 zeigt diese ohne die Ausnahmebehandlungen (Exceptions).

```
public SetDsc getAllEqual(String ftr, Number value);
public SetDsc getAllNotEqual(String ftr, Number value);
public SetDsc getAllLess(String ftr, Number value);
```

```
4  public SetDsc getAllLessOrEqual(String ftr, Number value);
5  public SetDsc getAllGreater(String ftr, Number value);
6  public SetDsc getAllGreaterOrEqual(String ftr, Number value);
7
8  public SetDsc getAllEqual(String ftr, Date date);
9  public SetDsc getAllNotEqual(String ftr, Date date);
10 public SetDsc getAllLess(String ftr, Date date);
11 public SetDsc getAllLessOrEqual(String ftr, Date date);
12 public SetDsc getAllGreater(String ftr, Date date);
13 public SetDsc getAllGreaterOrEqual(String ftr, Date date);
```

Listing 4.18: Schnittstelle *Feature-Logic*: Neue Methoden für die Vergleichsoperatoren

Wie leicht zu erkennen ist, müssen die Methoden einmal für Nummern- und einmal für Datumsobjekte implementiert werden. Zum Vergleichen von zwei Datumsobjekten wird jeweils der gespeicherte Long-Wert herangezogen und vorher durch 1000 geteilt, um sekunden- statt millisekundengenau zu vergleichen. Für den Vergleich von Zahlen sind aufgrund unterschiedlicher Genauigkeiten vorher die Datentypen der *Number*-Objekte zu bestimmen. *Number* ist die abstrakte Vaterklasse aller Zahlenklassen in Java. Tabelle 4.7 führt die vier möglichen Fälle für den größten Ganzzahl-Datentyp *Long* und den größten Gleitkommazahl-Datentyp *Double* auf.

Datentyp 1	Datentyp 2	Vergleichsmethode
Long	Long	long-Wert
Double	Double	double-Wert auf 5 signifikante Stellen gerundet
Long	Double	Falls der double-Wert beider Zahlen kleiner als $10^{15} - 1$ ist, wird dieser auf 5 signifikante Stellen gerundet und verglichen.
Double	Long	dto.

Tabelle 4.7: Feature-Logic: Vergleich von Zahlenwerten unterschiedlicher Datentypen

4.4.3 Feature-Logic für das Dateisystem

Einleitung: Sowohl für das Repository als auch die Sandbox ist, wie in den nächsten beiden Abschnitten beschrieben, eine Instanz der Feature-Logic zur Modellierung von Mengen, Relationen und Graphen notwendig. Für das Repository als zentrale Stelle der Systemarchitektur lohnt sich der Aufwand, einen Datenbankserver für die Feature-Logic aufzusetzen und zu unterhalten. Für die Sandbox ist vorerst auch eine weniger aufwändige Lösung ausreichend, die ohne eine separate Installation und Verwaltung auskommt.

Die Umsetzung muss ausreichend performant sein und die Daten dauerhaft speichern können. Die erste Anforderung lässt sich erfüllen, wenn das gesamte Feature-Logic-Modell während der Ausführung des Programms im Arbeitsspeicher vorgehalten wird, und die

zweite, indem jede der vier Feature-Logic-Relationen in eine Textdatei im Dateisystem gespeichert wird. Die Umsetzung setzt sich somit aus zwei Teilen zusammen: Die Klasse *FeatureLogicOOModel* ist für den transienten Teil und die Klasse *FeatureLogicFileSystem* für den persistenten Teil zuständig. Das UML-Klassendiagramm in Abbildung 4.9 zeigt die Anbindung an den Adapter *FeatureLogicAdapter*, wie er auf Seite 49 beschrieben wurde.

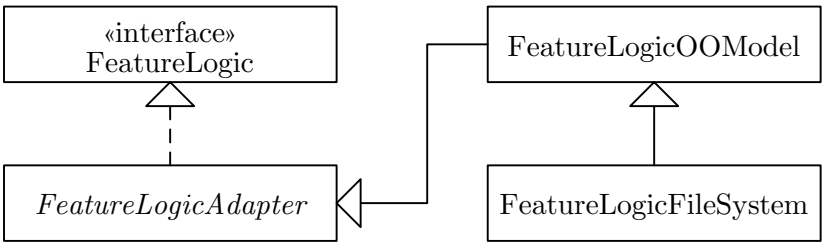


Abbildung 4.9: UML-Klassendiagramm: Feature-Logic für das Dateisystem

Transiente Feature-Logic: Zunächst soll die Klasse *FeatureLogicOOModel* vorgestellt werden. Die Mengen *Domain* und *Feature* können in Java einfach durch die Container-Klasse *Set* und die Relation *Atom* durch die Container-Klasse *Map* realisiert werden. Die Relation *Relslot* enthält drei Spalten und ist dadurch nicht direkt in einer vorhandenen Datenstruktur abbildbar. Denkbar wäre, stellvertretend für die Zeile eine Klasse mit drei Textattributen zu schreiben und die Objekte dieser Klasse in einem Set zu speichern. Jedoch müsste für Anfragen an die Feature-Logic über alle Zeilen iteriert werden, was die Leistungsfähigkeit enorm einschränken würde. Besser ist es, jedem unterscheidbaren Element, das in der ersten Spalte eingetragen ist, eine Map für die Feature-Wert-Paare aus der zweiten und dritten Spalte zuzuordnen. Die Speicherung erfolgt durch eine Schachtelung von einer äußeren und mehreren inneren Maps. Somit werden die Attribute der Klasse wie im Listing 4.19 definiert.

```
1 Set<String> m_domain          = new HashSet<String>();
2 Set<String> m_features        = new HashSet<String>();
3 Map<String,String> m_atom     = new HashMap<String,String>();
4 Map<String, Map<String,String>> m_relslot
5                     = new HashMap<String,Map<String,String>>();
```

Listing 4.19: Attribute der Klasse *FeatureLogicOOModel*

Beispiel 4.2: Umsetzung der transienten Feature-Logic

Die Umsetzung der transienten Feature-Logic soll an einem kleinen Beispiel verdeutlicht werden. Es existieren zwei geometrische Objekte, und zwar eine Linie, die von (0,0) nach (4,0) verläuft, und ein Kreis mit dem Mittelpunkt in (0,0) und dem Radius 2. Für die Modellierung in der Feature-Logic werden die vier nicht-primitiven Elemente *Linie, Kreis, Punkt1* und *Punkt2* benötigt und die atomaren Werte `2`, `0` und `4` den primitiven

Elementen `_1`, `_2`, und `_3` zugeordnet. Den nicht-primitiven Elementen können über Features primitive oder nicht-primitive Elemente zugewiesen werden. Den entsprechenden Feature-Graph für das Beispiel zeigt Abbildung 4.10 auf der linken Seite. Rechts ist nur für die beiden Attribute *m_atom* und *m_relslot* der Klasse *FeatureLogicOOModel* die Belegung im Arbeitsspeicher dargestellt. Während *m_atom* nur aus einem Map-Objekt besteht, verweisen die Schlüssel der äußeren Map von *m_relslot* auf jeweils ein inneres Map-Objekt, das Feature-Wert-Paare enthält.

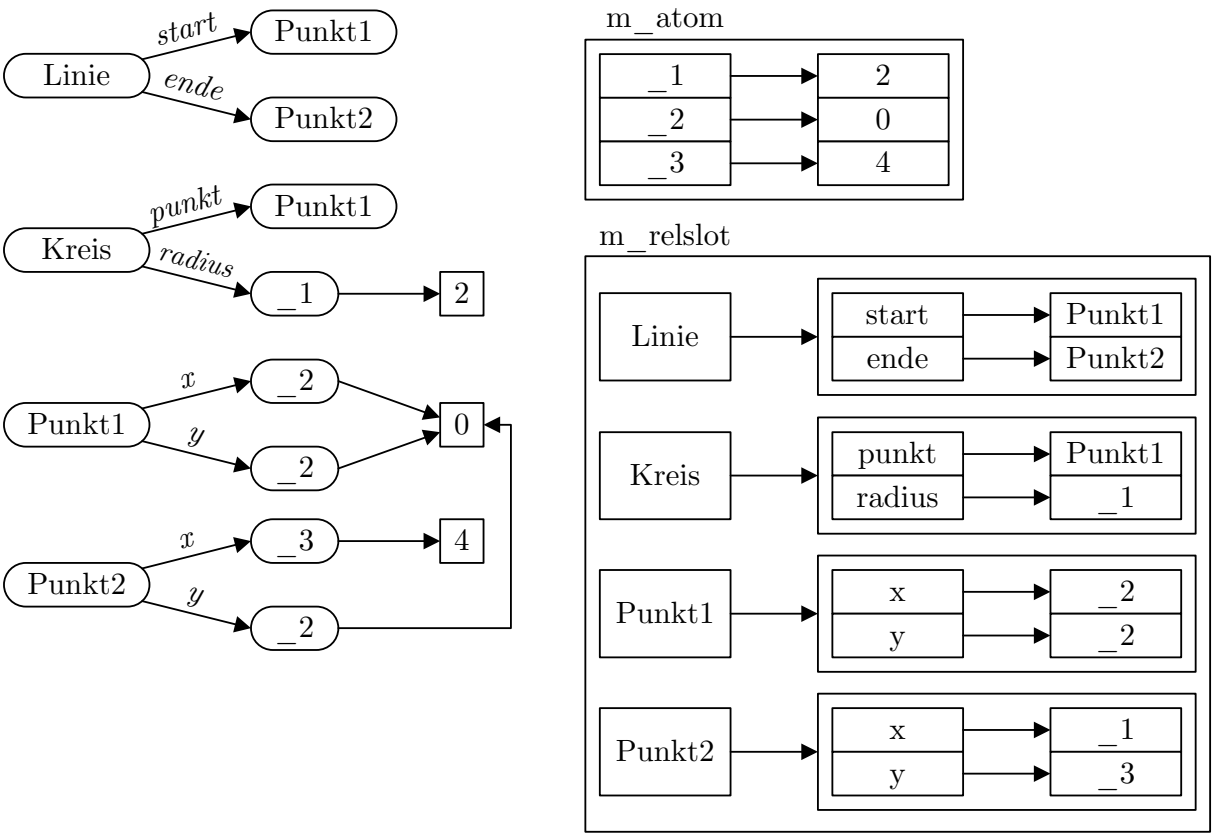


Abbildung 4.10: Transiente Feature-Logic: Beispiel mit geometrischen Objekten

Feature-Logic für das Dateisystem: Der persistente Teil der Feature-Logic für das Dateisystem ist in der Klasse *FeatureLogicFileSystem* implementiert, die von der Klasse *FeatureLogicOOModel* abgeleitet ist. Sie hat Zugriff auf die oben beschriebenen Datenstrukturen, da die Attribute mit der Sichtbarkeit *protected* definiert wurden. Jede der vier Datenstrukturen wird in eine eigene Textdatei gespeichert. In den Dateien `Domain` und `Feature` belegt jedes Element bzw. Feature eine eigene Zeile, wohingegen sich in der Datei `Atom` in jeder Zeile ein Paar aus einem primitiven Element und seinem atomaren Wert, getrennt durch ein Tabulator-Zeichen (ASCII-Code 9), befindet. Die Relation `Relslot` wird derart in einer gleichnamigen Datei gespeichert, dass für jedes Element erst der Name und dann seine Feature-Wert-Paare aufgeführt werden. Eine Leerzeile trennt zwei verschiedene Elemente voneinander.

Beispiel 4.3: Persistente Feature-Logic: Beispiel mit geometrischen Objekten

Für das obige Beispiel 4.2 soll an dieser Stelle der Inhalt der Textdateien aufgelistet werden.

Domain

```
Linie
Kreis
Punkt1
Punkt2
_1
_2
_3
```

Feature

```
start
ende
punkt
radius
x
y
```

Atom

```
_1    2
_2    0
_3    4
```

Relslot

```
Linie
start     Punkt1
end       Punkt2

Kreis
punkt     Punkt1
radius    _1

Punkt1
x         _2
y         _2

Punkt2
x         _3
y         _2
```

In Abbildung 4.11 sind die zusätzlich eingeführten Methoden für die Klasse *FeatureLogicFileSystem* aufgeführt. Wenn eine Instanz der Feature-Logic erzeugt wird, muss dem Konstruktor ein Verzeichnis übergeben werden. Falls die vier Textdateien schon existieren, werden sie mit der Methode *init()* geparst und die ausgelesenen Daten in den transienten Datenstrukturen gespeichert. Die Methode *checkState()* prüft vor jeder Operation, die die Daten manipulieren will, ob sich die Textdateien auf dem Datenträger seit dem letzten Einlesen verändert haben. Es kann nämlich nicht ausgeschlossen werden, dass der Nutzer mit mehreren Anwendungen gleichzeitig in der Sandbox arbeitet. In solch einem Fall veranlasst die Methode ein erneutes Einlesen. Ein Abspeichern aller Feature-Logic-Daten wird durch die Methode *writeToDisk()* eingeleitet, die vorher erst prüft, ob sich die transienten Daten verändert haben. Falls ja, wird eine exklusive Schreibsperre durch Anlegen der temporären Datei `WriteLock` gesetzt, die nach dem Schreibvorgang wieder gelöscht wird.

Zusätzlich wurde ein weiterer Sperrmechanismus implementiert, der einem Workspace erlaubt, während der Ausführung von verteilten Operationen in der Sandbox die Feature-Logic im Dateisystem exklusiv für sich mit der Methode *setExternalLock()* zu sperren. Zu diesem Zweck wird die Datei `ExternalLock` angelegt. Die Methode *isExternalLocked()* prüft, ob diese Datei existiert, und liefert den aktuellen Zustand der von außerhalb gesetzten Sperre zurück.

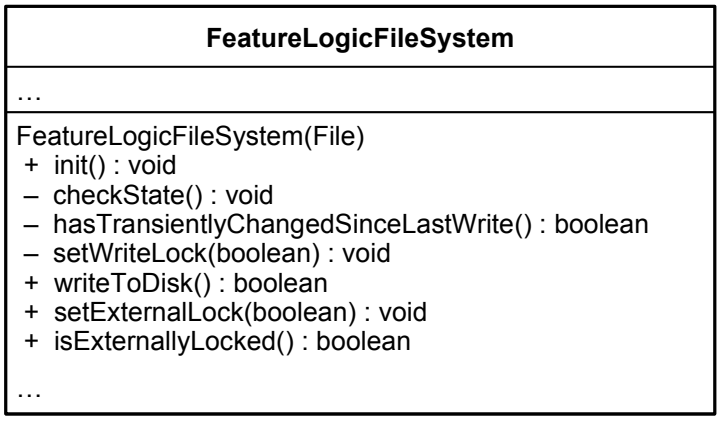


Abbildung 4.11: UML-Klassendiagramm: FeatureLogicFileSystem

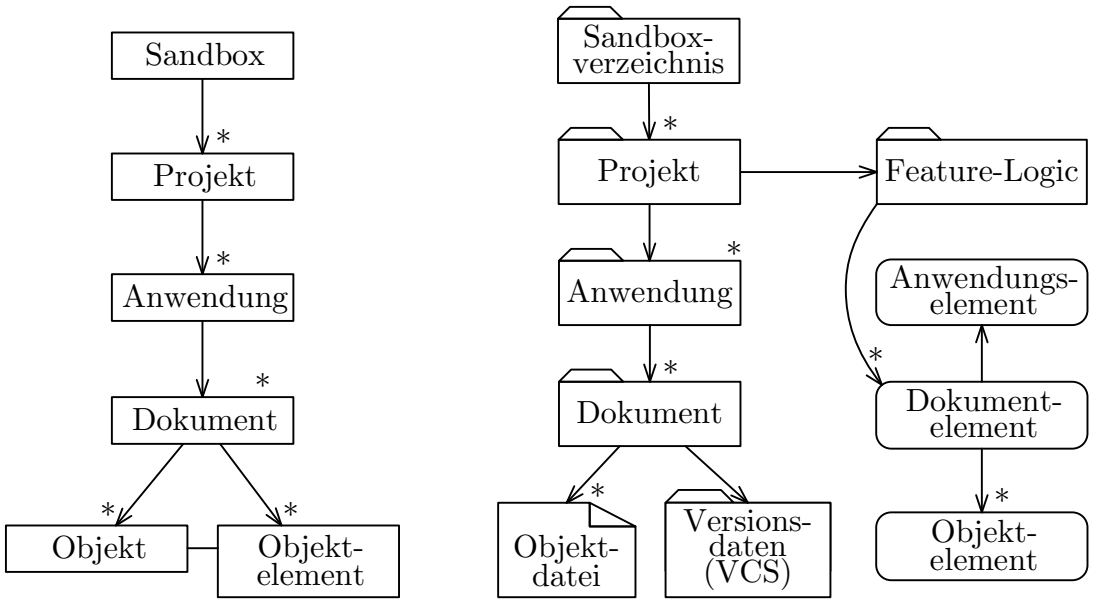
4.5 Strukturierung und Modellierung des Planungsmaterials

4.5.1 Sandbox

Struktur und Umsetzung: Das Planungsmaterial lässt sich in einer hierarchischen Struktur durch Aufteilung in Projekte, Anwendungen und Dokumente verwalten. Die Dokumente enthalten ihrerseits Objekte, denen – wie in Abbildung 4.12a ersichtlich – je ein gleichnamiges Element zugeordnet ist. Die Umsetzung der Sandbox erfolgt im Dateisystem des Rechners (s. Abbildung 4.12b). An oberster Stelle steht das Sandboxverzeichnis, gefolgt von den Projekt-, Anwendungs- und Dokumentverzeichnissen. Jedes Dokumentverzeichnis enthält für jedes Objekt eine serialisierte Datei sowie ein Verzeichnis, in welches das textbasierte Versionsverwaltungssystem Versionsdaten zu den Objekten speichert. Die Daten der Feature-Logic werden für jedes Projekt getrennt in der Feature-Logic-Umsetzung für das Dateisystem abgelegt.

Identifikation: Alle Teile der Sandboxstruktur werden mit einem persistenten Identifikator (PID) versehen, der zur Modellierung innerhalb der Feature-Logic verwendet wird. Über die PID ist zum Beispiel ein Objekt indirekt mit seinem Element verknüpft. Für die Vergabe der PID wird die hierarchische Struktur genutzt. Elemente der ersten Stufe, die Projekte, erhalten ihren Namen als PID, ebenso die Anwendungen als Elemente der zweiten Stufe. Anwendungen werden projektübergreifend verwendet und müssen in der PID nicht auf das Projekt verweisen. Die PID der nächsten Stufe, die Dokumente, setzt sich aus dem Projekt, der Anwendung und dem Namen des Dokuments getrennt durch das Zeichen „@“ zusammen.

Objekte verwenden als Vertreter der vierten Stufe folgende PID: Dokument-PID + „@“ + Klassenname + „___“ + eindeutiger Name des erzeugenden Bearbeiters + „___“ + UUID. Der Klassenname wird für die Bestimmung der Objektklasse bei der Deserialisierung verwendet. Die UUID ist für die eindeutige Identifikation nötig, da in einem Dokument mehrere Objekte der Klasse auftreten können. Ein iterativer Zähler wäre nur



(a) Struktur *(b)* Umsetzung

Abbildung 4.12: Umsetzungskonzept der Sandbox

über eine zentrale Vergabestelle zu realisieren, was zusätzlichen Aufwand bedeuten würde. Tabelle 4.8 zeigt ein Beispiel für die Vergabe der PIDs in der Sandbox. Die PID der Objekte wird als Persistenter Objektidentifikator (POID) bezeichnet, um die Bedeutung der Objekte zu betonen.

Stufe	Name	Beispiel	PID (Stufe 1-3)/POID (Stufe 4)
1	Projekt	Hochhaus	`Hochhaus`
2	Anwendung	CADEMIA	`cademia`
3	Dokument	Grundriss_EG	`Hochhaus@cademia@Grundriss_EG`
4	Objekt	Linie	`Hochhaus@cademia@Grundriss_EG@ + cib.cad.db.comp.ComponentLine2D__ + richter__ + 1079d8f1-c205-4538-9305-0dfe1c57ae0e`

Tabelle 4.8: Identifikation in der Sandbox

Die Syntax der Elemente unterliegt festgelegten Regeln. Der Name von Projekten, Anwendungen und Dokumenten darf aus Buchstaben, Ziffern, Binde- und Unterstrichen bestehen und muss mit einem Buchstaben oder einer Ziffer beginnen. Für den Nutzername gilt die gleiche Festlegung, mit der Ausnahme, dass er nur mit einem Buchstabe beginnen darf. Der Klassenname wird durch die Programmiersprache vorgegeben und enthält im Fall von Java vorangestellte und durch einen Punkt getrennte Paketnamen.

Für die Überprüfung der PID-Syntax eignen sich reguläre Ausdrücke. Das folgende Listing 4.20 zeigt die Definition der verwendeten regulären Ausdrücke im Quellcode. Die

Zeichenkettenklasse *String* in Java stellt die Methode `boolean matches(String regex)` bereit, die `true` zurückliefert, wenn der Text dem übergebenen regulären Ausdruck entspricht.

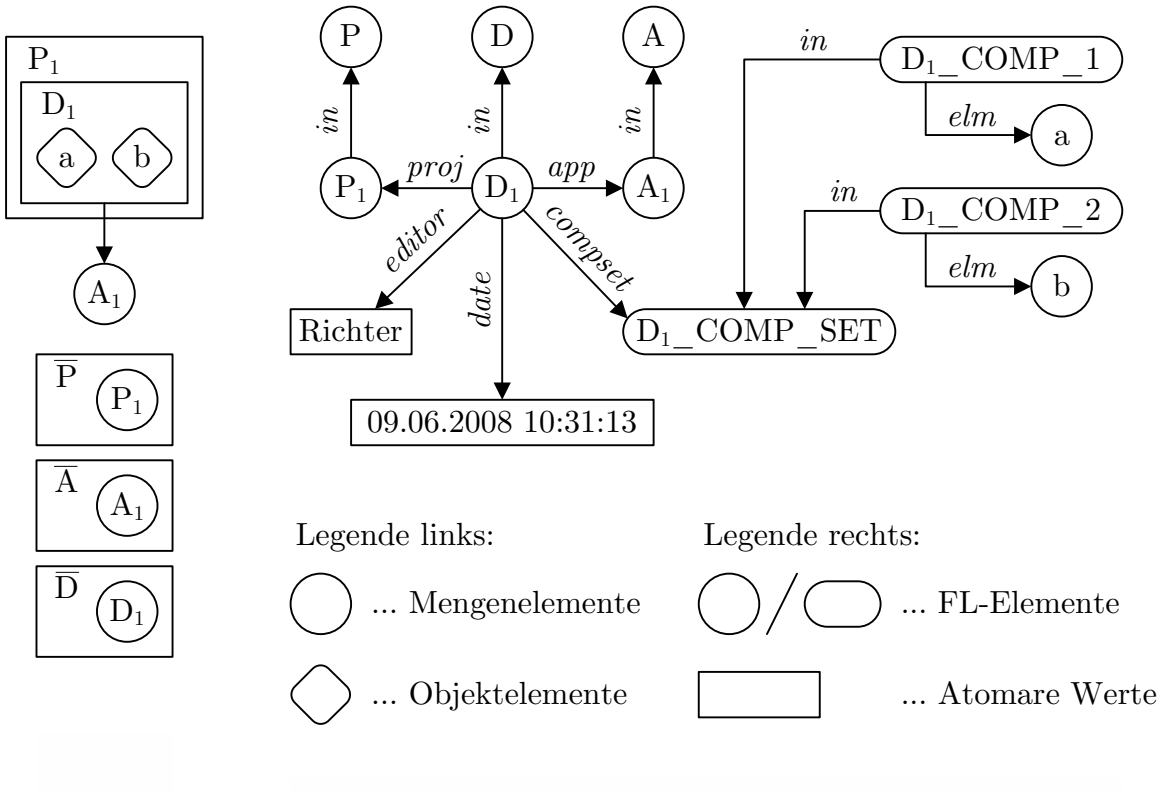
```
String SEPARATOR = "@";
String SEPARATOR_LINE = "__";

//\w = [A-Za-z0-9_], \u002D = "-", \u0020 = " "
String REG_EXP_USER = "[A-Za-z][\\w\\u002D]*";
String REG_EXP_PROJECT = "[A-Za-z0-9][\\w\\u002D]*";
String REG_EXP_APP = REG_EXP_PROJECT;
String REG_EXP_DOC = REG_EXP_PROJECT;
String REG_EXP_DOC_PID = REG_EXP_PROJECT +
          SEPARATOR + REG_EXP_APP + SEPARATOR + REG_EXP_DOC;
String REG_EXP_OBJECT_POID = REG_EXP_DOC_PID +
          SEPARATOR + "[^@]+" + SEPARATOR_LINE +
          REG_EXP_USER + SEPARATOR_LINE + "[^@]+";
```

Listing 4.20: Reguläre Ausdrücke zur Syntaxüberprüfung von PIDs

Modellierung des Dokuments in der Feature-Logic: Als erläuterndes Beispiel dient Abbildung 4.13, die im linken Teil das mathematische Modell und rechts das gleiche Modell, umgesetzt in der Feature-Logic, enthält. Im Mittelpunkt der Modellierung in der Feature-Logic steht das Dokument, im Beispiel mit D_1 bezeichnet. D_1 gehört zum Projekt P_1 und wurde mit der Anwendung A_1 erstellt. Die Zuordnung erfolgt über die Features *proj* und *app*. Aus Platzgründen wurden in diesem Beispiel keine realen PIDs für P_1, A_1 und D_1 verwendet. Die Zugehörigkeit dieser drei Elemente zu den entsprechenden Mengen P, A und D – Projekte, Anwendungen, Dokumente – wird durch das Feature *in* modelliert. Im Gegensatz zum mathematischen Modell wird der Überstrich bei den Mengenbezeichnern P, A und D weggelassen.

Die Features *editor* und *date* weisen dem Dokument indirekt über primitive Elemente, die hier nicht dargestellt sind, atomare Werte für den Bearbeiter und das letzte Speicherdatum zu. Die Zuordnung der Anwendungsobjekte zum Dokument wurde in einer frühen Entwurfsphase sehr flexibel ausgelegt, da zu diesem Zeitpunkt geplant war, dass Objekte Bestandteil mehrerer Dokumente sein durften. In der aktuellen Umsetzung ist ein Objekt maximal in einem Dokument enthalten, die ursprüngliche Modellierung wurde aber beibehalten. Das Dokument D_1 besitzt die Menge `D1_COMP_SET`, die wiederum die Objektelemente enthält. Aufgrund der ursprünglich geplanten Flexibilität muss die Mengenzugehörigkeit, analog zu Abbildung 2.21b auf Seite 45, mit zusätzlichen Elementen `D1_COMP_X` modelliert werden. Das `X` steht für das n-te Objekt innerhalb des Dokuments. Im Beispiel enthält das Dokument die zwei Objekte `a` und `b`.



(a) Mathematisches Modell

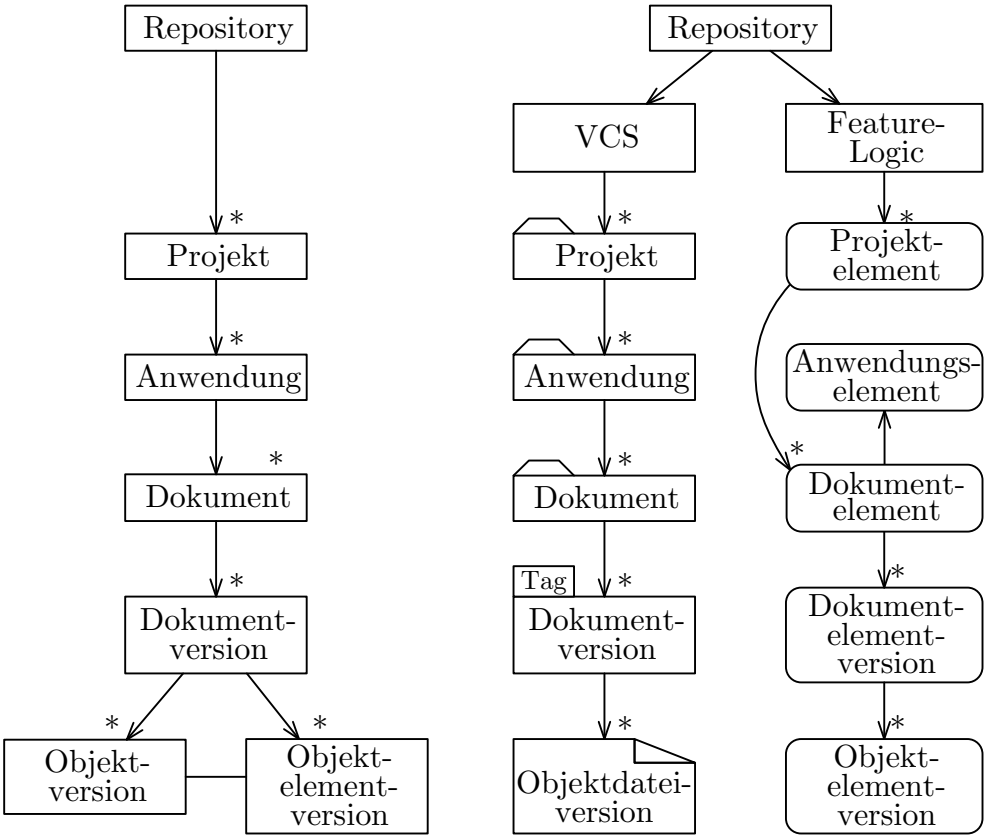
(b) Feature-Logic

Abbildung 4.13: Modellierung des Dokuments in der Sandbox

4.5.2 Repository

Struktur und Umsetzung: Die Struktur des Repositorys entspricht weitestgehend der hierarchischen Struktur der Sandbox, mit dem Unterschied, dass noch eine weitere Stufe hinzukommt. Unterhalb der Dokumente gibt es zusätzlich Versionen von ihnen, die selbst aus Objekt- bzw. Elementversionen bestehen (s. Abbildung 4.14a auf der nächsten Seite). Das Repository unterteilt sich in den VCS-Server, der die serialisierten Objektmodelle speichert und versioniert, und den Feature-Logic-Server, der das Feature-Logic-Modell auf Basis der versionierten Elemente speichert. Zwischen beiden Servern existiert keine direkte Beziehung (s. Abbildung 4.14b). Nur die gleichlautenden Identifikatoren für zusammengehörende Objekt- und Objektelementversionen stellen eine indirekte Kopplung her.

Über die interne Verwaltungsstruktur des VCS kann keine allgemeingültige Aussage getroffen werden, da sie unterschiedlich realisiert wurde. Die logische Struktur entspricht aber den hierarchischen Stufen, wie sie im Dateisystem der Sandbox angelegt sind. Direkt versioniert werden nur die Dateien der Objekte. Die Dokumentversionen entstehen dadurch, dass zusammengehörige Objektversionen mit einem Tag versehen werden, ohne



(a) Struktur *(b)* Umsetzung

Abbildung 4.14: Umsetzungskonzept des Repositorys

dass eine Kopie der Version erstellt werden muss. Diese effiziente Methode wurde schon erfolgreich beim Projekt *objectVCS* angewendet (s. Abschnitt 2.4.4 auf Seite 49).

Als Datencontainer für die Feature-Logic eignet sich ein ausgereifter und performanter Datenbankserver mit einem RDBMS, da dieser Transaktionen und Mehrbenutzerzugriff unterstützt sowie die Dauerhaftigkeit und Konsistenz der Daten sehr gut gewährleistet.

Identifikation: Alle PID und POID, die für unversionierte Elemente sowohl in der Sandbox als auch im Repository verwendet werden, bleiben unverändert. Nur an den Elementen, die versioniert werden, wird getrennt durch ein „@“ die jeweilige Revisionsnummer des VCS angehängt. Der versionierte Identifikator für Objekte wird als *Persistenter Objektversionsidentifikator* (POVID) und für die anderen Elemente als *Persistenter Versionsidentifikator* (PVID) bezeichnet. Die fünf hierarchischen Stufen des Repositorys werden nach folgendem Schema in Anlehnung an Tabelle 4.8 auf Seite 131 mit einer persistenten ID versehen.

Stufe	Name	Beispiel	PVID (Stufe 1-4)/POVID (Stufe 5)
1	Projekt	Hochhaus	`Hochhaus`
2	Anwendung	CADEMIA	`cademia`
3	Dokument	Grundriss_EG	`Hochhaus@cademia@Grundriss_EG`
4	Dokumentversion	Rev. 42	`Hochhaus@cademia@Grundriss_EG@42`
5	Objektversion	Linie, Rev. 23	`POID@23`

Tabelle 4.9: Identifikation im Repository

Wie an den letzten beiden Zeilen zu erkennen ist, werden nur die Dokumente und Objekte versioniert. Die PVIDs lassen sich genau wie PIDs durch reguläre Ausdrücke überprüfen. Zeile 1 des Listings 4.21 enthält den Ausdruck für eine Revisionsnummer im Long-Format, wie sie z. B. im Versionsverwaltungssystem Subversion verwendet wird. Mit den nächsten beiden Ausdrücken kann die PVID von Dokumenten und die POVID für Objekte geprüft werden. Der letzte Ausdruck ist für den Tag einer Dokumentversion vorgesehen, der aus Buchstaben, Zahlen, Unter-, Bindestrichen und Leerzeichen bestehen darf.

```
1 String REG_EXP_REV = "[1-9][0-9]*";  // SVN -> Long
2 String REG_EXP_DOC_PVID = REG_EXP_DOC_PID + SEPARATOR +
     REG_EXP_REV;
3 String REG_EXP_POVID = REG_EXP_OBJECT_POID + SEPARATOR +
     REG_EXP_REV;
4 String REG_EXP_TAG = "[A-Za-z0-9][\\w\\u0020\\u002D]*";
```

Listing 4.21: Reguläre Ausdrücke zur Syntaxüberprüfung von PVIDs

Modellierung von Dokumentversionen in der Feature-Logic: Das in Abbildung 4.13 auf Seite 133 verwendete Beispiel soll zur Erklärung der Dokumentversionsmodellierung auf Basis der Elemente aufgegriffen und in den Abbildungen 4.15 und 4.16 weiterentwickelt werden.

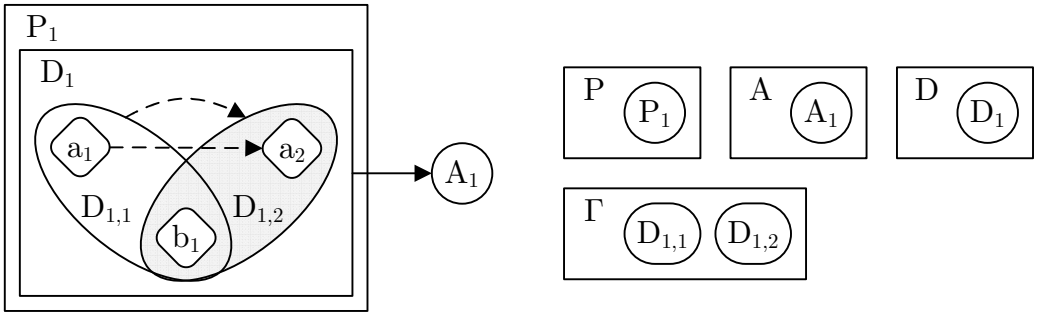


Abbildung 4.15: Mathematische Modellierung der Dokumentversionen

Das Dokument D_1 wird mit seinen beiden Objektelementen a und b unter dem Tag *„V1 Entwurf“* als erste Version unter dem Namen $D_{1,1}$ an das Repository übertragen.

Dabei wird in einem ersten Schritt der Commit-Befehl des VCS aufgerufen, der das Dokumentverzeichnis mit seinem Inhalt auf dem VCS-Server speichert und die Revisionsnummer 1 bei erfolgreicher Ausführung zurückliefert.

Im zweiten Schritt werden alle nötigen Einträge in der Feature-Logic vorgenommen. Das Feature *doc* ordnet der Dokumentversion $D_{1,1}$ das Dokument D_1 und die Feature *revision, tag, date* sowie *editor* die entsprechenden Werte zu. Abbildung 4.16 zeigt auf der linken Seite die Modellierung der Elemente und Features für $D_{1,1}$. Die Objektelementversionen erhalten aufgrund der Ersteintragung die gleiche Versionsnummer wie das Dokument, so dass a_1 und b_1 entstehen. Die Zuordnung der Objektelementversionen zur Dokumentversion erfolgt auf bekannte Art und Weise. Die Elemente `V_UUID1`, `V_UUID2` und `V_UUID3` aus Abbildung 4.17 modellieren die Kanten von der virtuellen Objektversion δ_0, die durch das Element `start` repräsentiert wird, zu $D_{1,1}, a_1$ und b_1 im Versionsgraph.

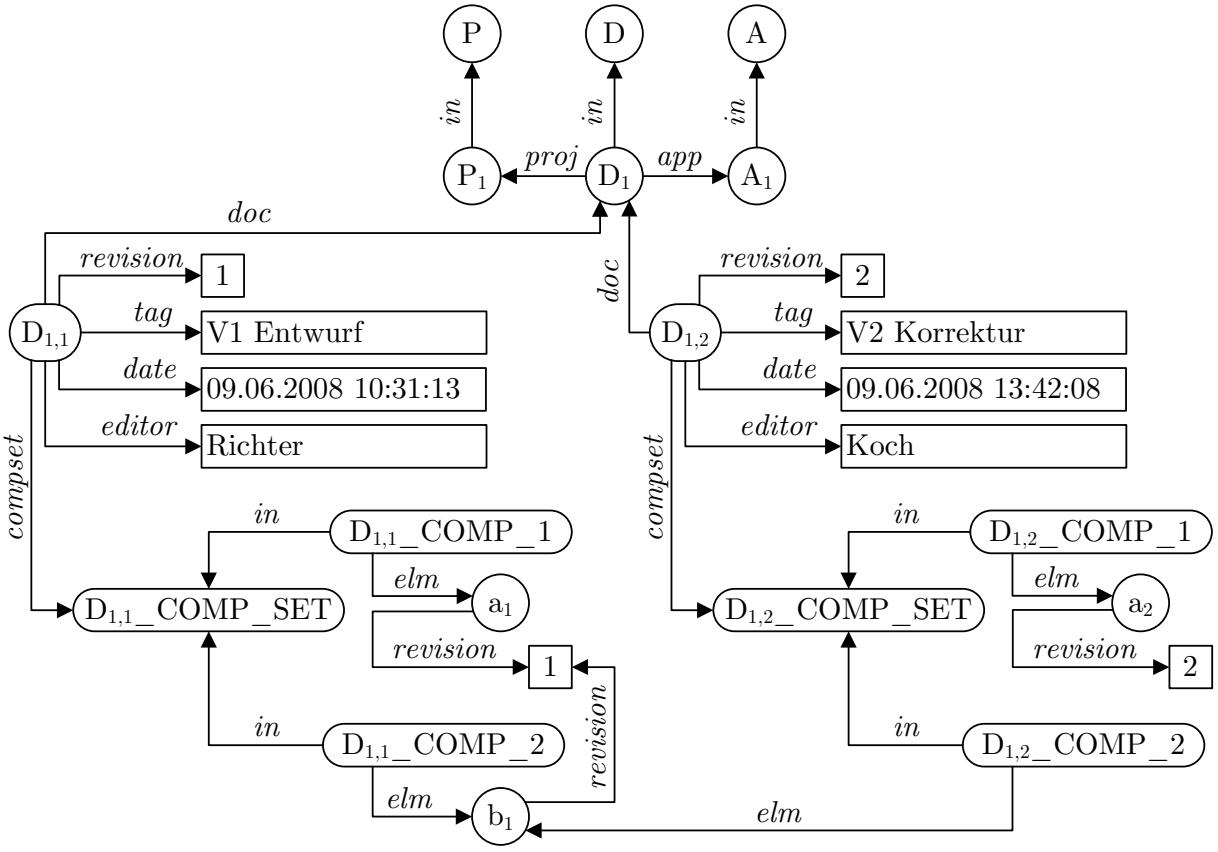


Abbildung 4.16: Modellierung der Dokumentversionen in der Feature-Logic

Nach einer Änderung des Zustands von a in der Sandbox überträgt der Bearbeiter eine zweite Version des Dokuments unter dem Tag *„V2 Korrektur“* an den Server, die den Namen $D_{1,2}$ zugewiesen bekommt. Sie enthält ebenso wie $D_{1,1}$ die Objektelementversion b_1 und die neu entstandene Objektelementversion a_2. Die gestrichelten Pfeile in Abbildung 4.15 verdeutlichen den Versionsübergang. Ihre Entsprechung in der Feature-Logic finden sie durch die Elemente `V_UUID4` und `V_UUID5` sowie deren Features in Abbil-

dung 4.17. Die Modellierung von $D_{1,2}$ erfolgt analog zu $D_{1,1}$, wobei `D1,2_COMP_2` auf die schon existierende Objektelementversion b_1 zeigt.

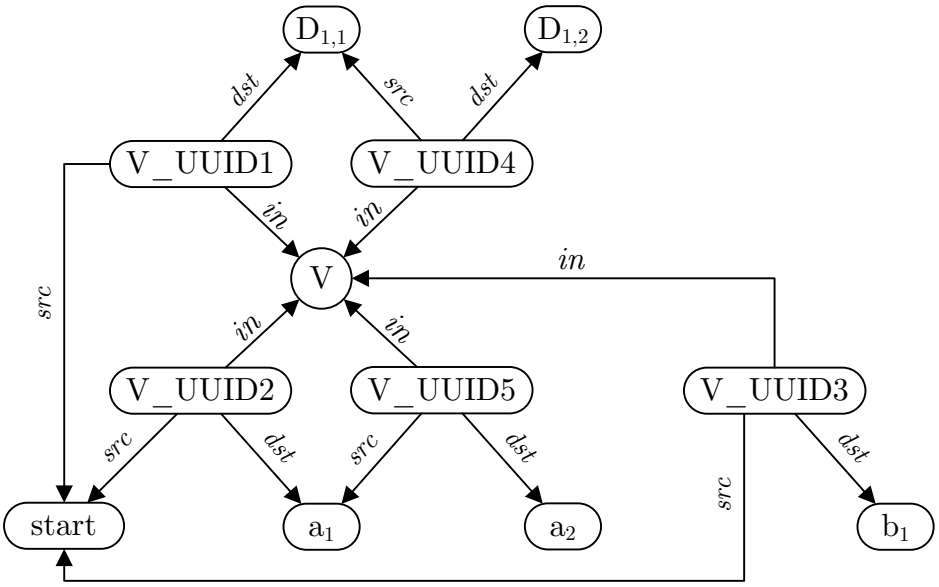


Abbildung 4.17: Modellierung der Dokumentversionen in der Feature-Logic, Versionsgraph für Dokument D_1 sowie Objekte a und b

4.6 Realisierung von Objektabhängigkeiten

4.6.1 Modellierung und Speicherung in der Sandbox

Klassen für die Modellierung von Bindungen: Bindungen stellen im mathematischen Modell eine Abhängigkeit zwischen zwei Objekten dar. Für die Definition einer konsistenten Bindung in der Sandbox sind die Bedingungen im Absatz *Sandbox* auf Seite 84 einzuhalten. Wie im Abschnitt 3.1.4 beschrieben, werden Bindungen in der Sandbox über Elemente $e \in \overline{E}$ definiert, die wiederum den Objekten $a \in \overline{Q}$ über die gemeinsame POID zugeordnet sind. Im Gegensatz zum auf Seite 31 beschriebenen Ansatz von (Olivier, 2007) erfolgt die Modellierung nicht auf Basis von Objektreferenzen, sondern über die POIDs. Damit steht eine sehr einfach zu verwaltende und flexible Lösung zur Verfügung.

Olivier führte für die Implementierung pro Bindung ein Binder-Objekt ein, das für ein gebundenes Objekt die Bindungen von beliebig vielen bindenden Objekten verwaltet. Für die vorliegende Umsetzung werden die wesentlichen Methoden in die Schnittstelle *Binder* verlagert und mit einem generischen Typ versehen (s. Abbildung 4.18). Der Update-Mechanismus wurde nicht übernommen, da keine automatische Aktionen bei Änderung eines bindenden Objekts durchgeführt werden sollen.

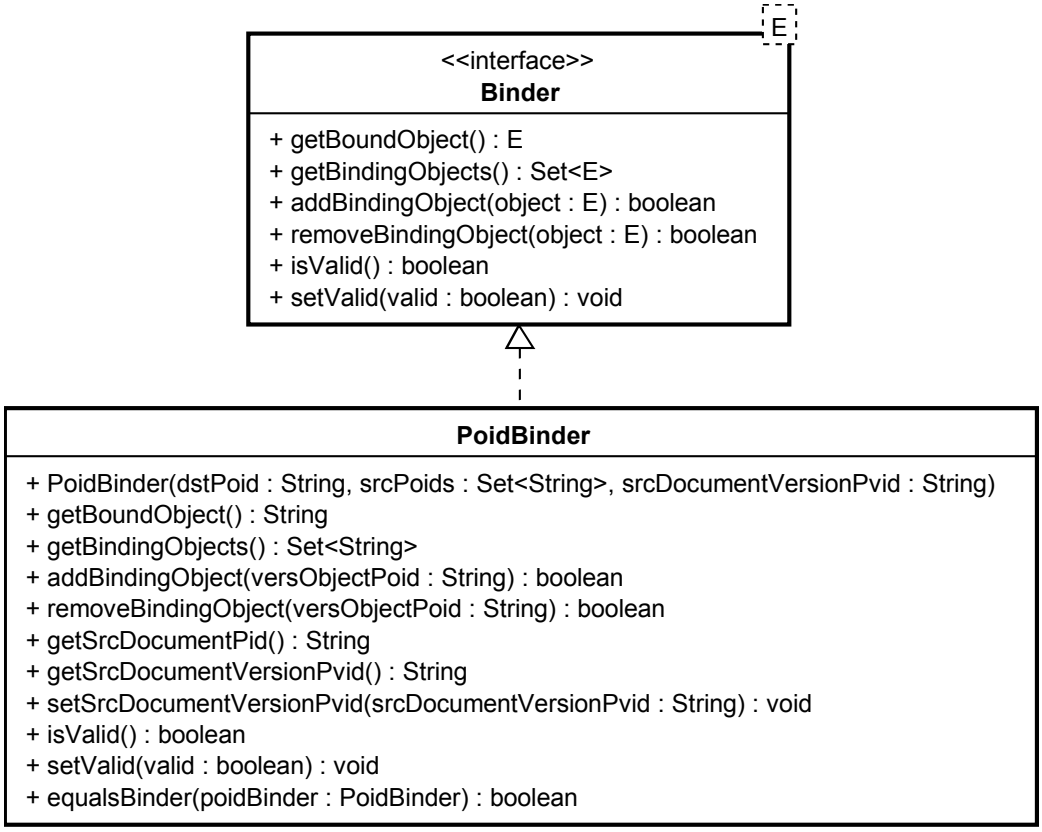


Abbildung 4.18: UML-Klassendiagramm: Schnittstelle *Binder* und Klasse *POIDBinder*

Das gebundene Objekt wird im Konstruktor implementierender Klassen übergeben und kann später nicht mehr geändert werden. Bindende Objekte können dagegen später beliebig hinzugefügt oder gelöscht werden. Außerdem lässt sich die Gültigkeit einer Bindung über die Methode *setValid()* setzen und mit *isValid*() abfragen. Wann eine Abhängigkeit als gültig betrachtet wird, ist durch die nutzende Anwendung festzulegen. Im Rahmen der im Kapitel 5 auf Seite 173 vorgestellten Pilotimplementierung muss der Planer entscheiden, wann eine Bindung zwischen Objekten widerspruchsfrei ist.

Von der Schnittstelle wird die Klasse *POIDBinder* abgeleitet, die die Bindungen über die POIDs verwaltet. Der generische Typ `E` wird demzufolge durch den Textdatentyp `String` ersetzt. Im Gegensatz zum mathematischen Modell gehören alle bindenden Objekte, die zu einer Dokumentversion gehören und auf das gleiche Objekt eines Dokuments in der Sandbox verweisen, zu einer Bindung. Dies hilft, den Verwaltungsaufwand im Workspace zu verringern. Der Konstruktor nimmt die POID des gebundenen Objekts, eine Menge von bindenden Objekten und die PVID der Dokumentversion, die alle bindenden Objekte enthält, entgegen. Ergänzend zur Schnittstelle *Binder* fügt die Klasse weitere für die Objektversionierung spezifische Methoden hinzu. *getSrcDocumentPid*() liefert die PID und *getSrcDocumentVersionPvid*() die PVID der bindenden Dokumentversion zurück. Beide lassen sich aus der POVID der bindenden Objekte aufgrund der hierarchischen Struktur der IDs ableiten. Nach der Ausführung der Operation *Update* ist mit der Methode *setSrcDocumentVersionPvid()* ein Aktualisieren der Dokument-PVID möglich.

Transiente Verwaltung im Workspace: In der Klasse *WorkspaceAdapter* wird ein Attribut *m_binders* vom Typ `Set<PoidBinder>` eingefügt, das Objekte vom Typ *POIDBinder* aufnehmen kann. Für das Hinzufügen und Entfernen sowie den Zugriff auf Binder-Objekten wurde die Schnittstelle *Workspace* um die Methoden des Listings 4.22 ergänzt. Die Methode *getBinders()* gibt alle POIDBinder-Objekte zurück und Methode *getBinders(String destPoid)* nur diejenigen, die auf ein gebundenes Objekt mit der angegebenen POID verweisen.

```
1 public boolean addBinder(PoidBinder b, boolean newBinding);
2 public boolean removeBinder(PoidBinder b);
3 public void clearAllBinders();
4 public Iterator<PoidBinder> iterateBinders();
5 public Set<PoidBinder> getBinders();
6 public Set<PoidBinder> getBinders(String destPoid) throws
     WorkspaceException;
```

Listing 4.22: Schnittstelle *Workspace*: Methoden zur Verwaltung von POIDBinder-Objekten

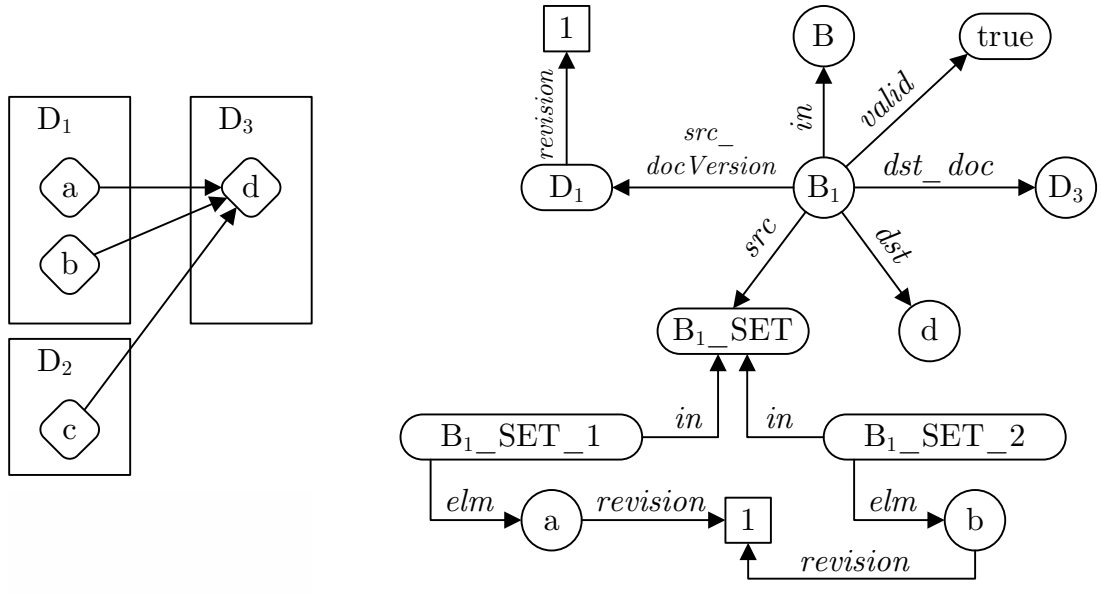
Modellierung von Bindungen in der Feature-Logic: Die Modellierung der Bindung in der Feature-Logic erfolgt über die POID der Objekte als persistenter Schlüssel. Wie bei der transienten Speicherung werden alle Bindungen, die auf dasselbe gebundene Objekt verweisen und bei denen die bindenden Objekte Teil eines Dokuments sind, zu einer Bindung zusammengefasst. Zu beachten ist, dass die Objekte in der Feature-Logic durch Objektelemente repräsentiert werden.

Wie im Beispiel der Abbildung 4.19a dargestellt, ist das Objekt d an die beiden Objekte a und b des Dokuments D_1 und an das Objekt c des Dokuments D_2 gebunden. Diese drei voneinander unabhängigen Bindungen werden für die Speicherung in der Feature-Logic zu zwei Bindungen mit den Bezeichnungen B_1 und B_2 zusammengefasst. Eine Bindung B_i enthält alle Bindungen, deren bindende Objekte sich in einem Dokument befinden und auf dasselbe gebundene Objekt verweisen. Somit enthält B_1 die Paare (a, d) sowie (b, d) und B_2 das Paar (c, d).

B_1 steht in Abbildung 4.19b als Abkürzung für eine PID der Form `B__Bearbeiter__UUID` und gehört zur Menge B, die der lokalen Bindungsmenge $\overline{B}$ nach (3.23) gleicht. Weiterhin zeigt B_1 mit dem Feature *src_docVersion* auf das Dokument mit den bindenden Objektelementen und mit dem Feature *dst_doc* auf das Dokument des gebundenen Objektelements. Ähnlich wie bei der Dokumentmodellierung werden die bindenden Objektelemente einer Menge `B1_SET` zugeordnet. In diesem Fall ist die aufwändigere Modellierung notwendig, da Objektelemente in mehreren Bindungen als bindendes Objektelement auftreten können. Das Feature *src* verknüpft diese Menge mit der Bindung B_1, wohingegen das Feature *dst* das gebundene Objektelement d mit B_1 verknüpft.

Für die Definition der Bindung muss die Versionsnummer des bindenden Dokuments und der bindenden Objektelemente in der Sandbox bekannt sein, deshalb wird sie über das Feature *revision* abgespeichert. Bindungen können in der Sandbox im Gegensatz zum Repository den Status gültig oder ungültig besitzen, deshalb wird dem Bindungselement B_1

ein weiteres Feature *valid* mit dem angenommenen Wert `true` zugeordnet. Für das Bindungspaar (c, d) muss in gleicher Art und Weise eine Bindung B_2 modelliert werden, die aber nicht in Abbildung 4.19b aufgenommen wurde.



(a) Mathematisches Modell *(b)* Feature-Logic (für (a, d) und (b, d))

Abbildung 4.19: Modellierung der Bindung in der Sandbox

Erweiterung für das VCS: Wie schon in Abbildung 3.4b auf Seite 82 gezeigt, sollen Änderungen an Bindungen keine Auswirkungen auf den Zustand gebundener Objekte haben. Um eine geänderte Bindung zum Server übertragen zu können, wird eine neue Revisionsnummer benötigt, die vom Versionsverwaltungssystem vergeben wird. Da Bindungen außerhalb der Dokumente in der Feature-Logic gespeichert werden, haben sie keinen Einfluss auf den Inhalt serialisierter Dateien in den Dokumentverzeichnissen. Solange aber keine Änderungen an den Dateien vorgenommen werden, kann kein Commit durchgeführt und keine neue Revisionsnummer angefragt werden. Folglich muss das Dokumentverzeichnis auch Informationen zu den Bindungen enthalten, die auf das enthaltene Dokument zeigen.

Zu diesem Zweck wird beim Speichern des Dokuments zusätzlich eine Datei mit dem Namen *bindings.txt* geschrieben, die für alle gebundenen Objekte Bindungsinformationen enthält. Wie im schematischen Aufbau in Listing 4.23 zu sehen, gehören dazu die PID des Binders, die POID des gebundenen Objekts, die PVID des bindenden Dokuments und eine Liste mit POVIDs der bindenden Objekte. Wichtig ist, dass die Binder und die bindenden Objekte pro Binder immer in der gleichen Reihenfolge gespeichert werden, sonst würde das VCS trotz unveränderten Planungsmaterials einen Unterschied feststellen.

```
PID_Binder_1
POID_Gebundenes_Objekt
PVID_Dokumentversion_mit_den_bindenden_Objekten
POVID_Bindendes_Objekt_1
POVID_Bindendes_Objekt_2

PID_Binder_2
POID_Gebundenes_Objekt
PVID_Dokumentversion_mit_den_bindenden_Objekten
POVID_Bindendes_Objekt_1
```

Listing 4.23: Schematischer Aufbau der Datei *bindings.txt*

4.6.2 Modellierung und Speicherung im Repository

Modellierung von Bindungen in der Feature-Logic: Die Modellierung von Bindungen im Repository wird anhand des Beispiels aus den Abbildungen 4.20 und 4.21 erläutert. Es existieren zu Beginn zwei Dokumente in der Sandbox des Nutzers: D_1 mit den Objektelementen a, b und D_2 mit dem Objektelement c. D_1 liegt zusätzlich schon als Version $D_{1,1}$ mit den Objektelementversionen a_1, b_1 im Repository vor. Der Nutzer definiert eine Bindung B_1 zwischen den Objektelementen a und c und committet danach das Dokument D_2. Da von diesem noch keine Version existiert, wird eine neue Version $D_{2,1}$ mit der Objektelementversion c_1 angelegt.

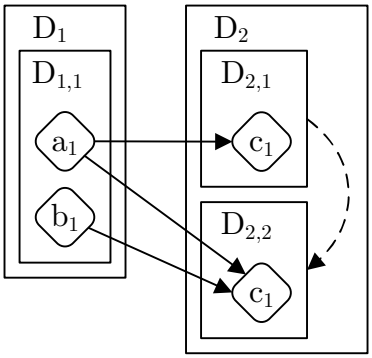


Abbildung 4.20: Mathematische Modellierung von Bindungen im Repository

Die Modellierung der Bindung im Repository ist ähnlich zu der in der Sandbox mit dem Unterschied, dass sich auf Dokument- und Objektelementversionen bezogen wird. Da Bindungen, wie im Abschnitt 3.1.4 auf Seite 82 beschrieben, nun auch versioniert sind, wird eine Bindungsversion $B_{1,1}$ angelegt und ihr die bindende Objektelementversion a_1, die gebundene Objektelementversion c_1 und deren Dokumentversionen $D_{1,1}, D_{2,1}$ zugeordnet.

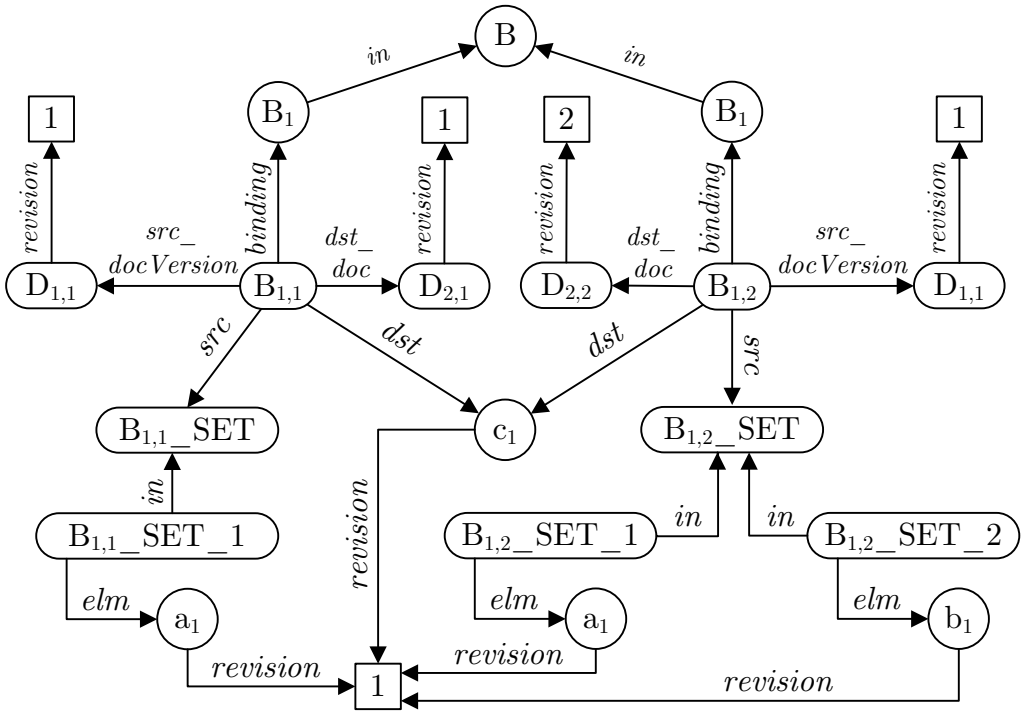


Abbildung 4.21: Modellierung von Bindungen in der Feature-Logic des Repositorys

In einem zweiten Bearbeitungsgang bindet der Nutzer das Objektelement c zusätzlich an das Objektelement b von D_1. Obwohl sich am eigentlichen Dokumentinhalt nichts geändert hat, besitzt das Dokument wegen der Änderung der Bindungen, die auf das Objektelement c des Dokuments D_2 zeigen, einen neuen Zustand, der einen Commit erlaubt. Nach der Übertragung von D_2 an den Server ist eine neue Version $D_{2,2}$ entstanden, die genau wie $D_{2,1}$ die Objektelementversion c_1 enthält. Weiterhin gibt es eine neue Bindungsversion $B_{1,2}$, die die beiden Bindungspaare (a_1, c_1) und (b_1, c_1) modelliert. Abbildung 4.22 enthält die Umsetzung des Versionsgraphen in der Feature-Logic für die Bindung B_1. Die Hilfselemente `V_UUID1` und `V_UUID2` modellieren je eine Kante.

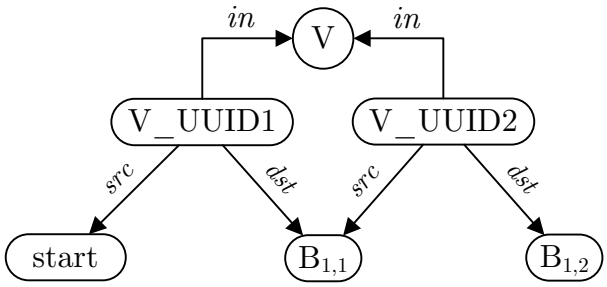


Abbildung 4.22: Modellierung von Bindungen in der Feature-Logic des Repositorys, Versionsgraph für Bindung B_1

4.6.3 Bearbeiten von Objektabhängigkeiten

Allgemein: Die bisherigen Abschnitte zeigen, wie sich Objektabhängigkeiten transient und persistent in Sandbox und Repository modellieren lassen. Jedoch wurde noch nicht der Prozess der Bearbeitung näher beleuchtet. Die Bindungen selbst müssen durch den Planer in der Anwendung erzeugt, bearbeitet und gelöscht werden können. Die in Tabelle 4.10 gezeigten fünf möglichen Fälle werden nachfolgend näher erläutert.

Ohne Anwendungs-unterstützung	Mit Anwendungsunterstützung			
	Bindung innerhalb eines Dokuments	Bindung zwischen zwei Dokumenten		
		In einer Anwendung		In zwei Anwendungen
		Ein Dokument	Zwei getrennte Dokumente	
Fall 1	Fall 2	Fall 3	Fall 4	Fall 5

Tabelle 4.10: Mögliche Fälle bei der Bearbeitung von Objektabhängigkeiten

- **Ohne Anwendungsunterstützung (Fall 1):** Unabhängig vom Ladezustand und von der Visualisierung der Objekte in der Anwendung könnten Bindungen allein durch die Angabe der POID von Ursprungs- und Zielobjekt durch den Planer definiert werden. Genauso könnten alle existierenden Bindungen für eine Bearbeitung in einer Liste mit ihren PIDs und den POIDs der zugehörigen Objekte angezeigt werden. Augenscheinlich ist dieser Ansatz dem Nutzer nicht zumutbar, da er anhand der POID kaum das richtige Objekt zuordnen kann.
- **Mit Anwendungsunterstützung:** Es ist anzunehmen, dass der Nutzer Bindungen in der Anwendung definieren will, mit der er die Datenmodelle bearbeitet. Von Vorteil ist dabei, dass die Objekte in ihrer grafischen Repräsentation vorliegen und für die Erzeugung einer Bindung einfach ausgewählt werden können. Weiterhin könnten die Bindungen grafisch in der Anwendung, z. B. durch Pfeile, dargestellt werden. Je nachdem, ob die Objektabhängigkeiten dokumentübergreifend sind oder nicht, ergeben sich folgende Ansätze.
  - **Bindungen innerhalb eines Dokuments (Fall 2):** Abhängigkeiten zwischen Objekten eines Datenmodells bezeichnet (Olivier, 2007) als *intra-model binding*. Sie werden meist durch die Fachanwendung selbst verwaltet und im Dokument gespeichert. Gleichwohl ließen sich diese Bindungen unabhängig von der Anwendung über den Workspace modellieren und speichern. Da alle Bindungen nur für Objekte innerhalb eines Dokuments existieren, lässt sich die Bearbeitung der Abhängigkeiten relativ einfach innerhalb der Anwendung umsetzen, sofern ein Zugriff auf die Präsentationsschicht besteht.
  - **Bindungen zwischen zwei Dokumenten:** Diese Art der Bindung wird am häufigsten anzutreffen sein, da Abhängigkeiten zwischen zwei unterschiedlichen Datenmodellen innerhalb einer Fachdomäne oder fachdomänenübergreifend helfen, die Konsistenz des Planungsmaterials sicherzustellen. Olivier defi-

niert diese Form als *inter-model binding*. Für das Bearbeiten ist zu unterscheiden, ob sich die Dokumente in eine Anwendung laden lassen oder nicht.

- **In einer Anwendung:** Voraussetzung dafür ist, dass beide Modelle mit dieser Anwendungen angezeigt werden können. Im **Fall 3** werden beide Datenmodelle in einem Dokument angezeigt und verwaltet. Das bedeutet, dass zu einem geladenen Dokument ein weiteres Dokument schreibgeschützt importiert und unterscheidbar vom ursprünglichen Inhalt dargestellt wird. Bindungen lassen sich dann sehr intuitiv erzeugen und anzeigen. Falls diese Variante nicht möglich ist, müssen beide Modelle in getrennten Dokumenten geladen werden, was eine MDI-Anwendung erfordert (**Fall 4**). Die Bindungsvisualisierung ist dann zwar weniger intuitiv oder ausgeschlossen, dafür besteht immer noch der einheitliche Zugriff auf beide Datenmodelle gegenüber dem letzten Fall 5.
- **In zwei Anwendungen (Fall 5):** Eine nutzerfreundliche und durchgängige Lösung ist bei zwei unterschiedlichen Anwendungen entweder schwierig und gar nicht zu implementieren. Die Anwendung mit dem Dokument der gebundenen Objekte braucht einen Zugriff auf das Objektmodell der anderen Anwendung, um zum Beispiel den Selektionsstatus der bindenden Objekte auszulesen oder zu setzen. Wie die Kommunikation erfolgt, ist im Einzelfall zu entscheiden.

Unterstützung durch den Workspace für den Fall 3: Der Workspace soll dem Programmierer eine Unterstützung für das Importieren eines weiteren Datenmodells in ein geöffnetes Dokument bieten. Dazu werden in der Klasse *WorkspaceAdapter* zwei Mengen zur Verwaltung der PIDs bzw. POIDs importierter Dokumente und deren Objekte zur Verfügung gestellt. Die Namen der Attribute vom Typ `Set<String>` sind `m_importedDocumentPids` und `m_importedObjectPoids`. Außerdem werden Methoden zur Bearbeitung dieser Mengen und für die Abfrage des Status zur Klasse *WorkspaceAdapter* (Listing 4.24), letztere auch zur Schnittstelle *Workspace* (Listing 4.25), hinzugefügt.

```
1 protected void addPoidToImportedSet(String poid){...}
2 protected void removePoidFromImportedSet(String poid){...}
3 protected void addDocumentPidToImportedSet(String docPid){...}
4 protected void removeDocumentPidFromImportedSet(String docPid)
     {...}
5 public boolean isImportedPoid(String poid){...}
6 public boolean isImportedDocument(String docPid){...}
7 public int numberOfImportedDocuments(){...}
8 public Set<String> getImportedDocuments(){...}
```

Listing 4.24: Klasse *WorkspaceAdapter*: Methoden zum Verwalten von PIDs bzw. POIDs importierter Dokumente und Objekte

```
1 public boolean isImportedPoid(String poid);
2 public boolean isImportedDocument(String docPid);
3 public int numberOfImportedDocuments();
4 public Set<String> getImportedDocuments();
```

Listing 4.25: Schnittstelle *Workspace*: Methoden zur Statusabfrage importierter Dokumente und Objekte

Diese Methoden bieten allein noch keine ausreichende Unterstützung für das Importieren, da noch Operationen fehlen, die den Import, das Auffrischen und das Entfernen importierter Dokumente durchführen. Dafür wird die Workspace-Schnittstelle noch um die Methoden des Listings 4.26 erweitert, die bis auf die letzte im speziellen Workspace einer Anwendung implementiert werden müssen.

Bevor Bindungen erzeugt werden können, müssen ein oder mehrere Dokumente mit *importDocument*() importiert werden. Sollten zu einem geladenen Dokument schon Bindungen existieren, lädt *importBindingDocuments*() die bindenden Dokumente nach. *dropImportedDocument*() entfernt ein per PID angegebenes Dokument aus der Anwendung und *dropAllImportedDocuments*() alle importierten Dokumente. Die im *WorkspaceAdapter* befindliche Methode *refreshImportedDocuments()* ruft nacheinander für alle importierten Dokumente *dropImportedDocument*() und *importDocument*() auf. So können Änderungen, die von außerhalb an den Dokumenten in der Sandbox vorgenommen wurden, in der Anwendung aktualisiert werden.

```
1 public int[] importDocument(String docPid, Object context)
2     throws WorkspaceException;
3 public int[] importBindingDocuments(String application,
4     Object context) throws WorkspaceException;
5 public int dropImportedDocument(String docPid, Object context)
6     throws WorkspaceException;
7 public int dropAllImportedDocuments(Object context);
8 public int refreshImportedDocuments(Object context)
9     throws WorkspaceException;
```

Listing 4.26: Schnittstelle *Workspace*: Operationen zur Durchführung des Imports von Dokumenten

4.6.4 Gültigkeitsprüfung

Zustände: Objektabhängigkeiten in der Sandbox können unter anderem ungültig werden, wenn neuere Dokumentversionen von bindenden Objekten in die Sandbox durch ein Update geladen werden und sich diese Objekte geändert haben. Es gibt aber noch weitere Fälle, die Bindungen ungültig werden lassen. Tabelle 4.11 führt diese zusammen mit den

nötigen Nutzeraktionen auf. Hat sich weder an den Dokumenten noch an den Bindungen etwas geändert, sind die Bindungen weiterhin gültig.

Gebundenes Objekt in der Sandbox	Zustand und Aktion.
Hat sich geändert.	Bindung ist weiterhin gültig. Keine Aktion nötig. Es wird davon ausgegangen, dass der Bearbeiter bewusst diese Änderung vorgenommen hat.
Fehlt.	Bindung ist zwecklos. Bindung löschen.
Bindende Objekte in der Sandbox	
Mindestens eins hat sich geändert.	Bindung ist ungültig. Konsistenz prüfen.
Mindestens eins hat sich nicht geändert, aber eine höhere Versionsnummer als in der Bindung verzeichnet.	Bindung ist ungültig. Konsistenz prüfen.
Mindestens eins hat sich nicht geändert, aber eine niedrigere Versionsnummer als im PoidBinder verzeichnet.	Bindung ist ungültig. Update des bindenden Dokuments.
Mindestens eins fehlt.	Bindung ist ungültig. Konsistenz prüfen. Falls keine bindenden Objekte mehr vorhanden sind, ist die Bindung zu löschen.

Tabelle 4.11: Status von Objektabhängigkeiten bzw. Bindungen

Erweiterung des Workspace: Für die Gültigkeitsprüfung von Bindungen wird in der Klasse *WorkspaceAdapter* die Methode *checkBinder()* eingeführt, deren vollständiger Kopf in Listing 4.27 dargestellt ist. Sie ist für jede zu prüfende PoidBinder-Instanz aufzurufen. Die zu übergebende Menge *objectsInDocument* enthält alle POIDs des aktuellen Dokuments, die mit der in der Workspace-Schnittstelle enthaltenen Methode *objectsInLoadedDocument*() ermittelt werden können. Die übrigen Parametermengen werden während der Abarbeitung der Methode mit den entsprechenden POIDs gefüllt.

```
1 protected boolean checkBinder(PoidBinder poidBinder,
2     Set<String> objectsInDocument,
3     Set<String> missingBoundObject,
4     Set<String> missingBindingObjects,
5     Set<String> changedSrcObjects,
6     Set<String> wrongSrcObjects,
7     Set<String> olderSrcObjects,
8     Set<String> newerSrcObjects, boolean reinitFeatureLogic)
9     throws WorkspaceException {...}
```

Listing 4.27: Klasse *WorkspaceAdapter*: Methode *checkBinder()* für die Gültigkeitsprüfung einer Bindung

Um zu bestimmen, ob sich ein Objekt in der Sandbox verändert hat, muss der aktuelle Stand mit dem des letzten Updates oder Commits verglichen werden. Günstigerweise speichert das verwendete VCS *Subversion* eine zweite Kopie jeder Datei nach einem Checkout, Commit und Update, so dass trotz des doppelten Speicherbedarfs in der Sandbox auf langsame Netzwerkzugriffe verzichtet werden kann. Bei Verwendung der manuellen Serialisierung in eine ZIP-Datei aus Absatz 4.3.4 auf Seite 113 lässt sich nur eine Änderung des Dokuments direkt feststellen. Für die Prüfung einer Objektänderung müssen die CRC-32-Werte des entsprechenden ZIP-Eintrags vom aktuellen und vom hinterlegten Dokument-ZIP-Archiv verglichen werden. Dazu dient die Methode *objectHasChangedSinceLastUpdateOrCommit()* aus Listing 4.28, die sich wiederum der beiden Methoden *getCurrentCRCOfZipEntry()* und *getCRCOfZipEntryOfLastCommit()* der Schnittstelle *VCClient* von Seite 250 bedient.

Die Methode *checkBindings()* prüft für die übergebenen POIDs der gebundenen Objekte die Bindungen, indem sie für jedes der übergebenen POIDBinder-Objekte die Methode *checkBinder()* aufruft. Die sechs möglichen Ergebnismengen werden für die Rückgabe in ein Feld vom Typ `Set<String>` gespeichert. Abschließend gibt es noch die Methode *checkBindingsInSandbox()*, die für ein nicht geladenes Dokument die Bindungen untersucht. Dafür muss sie aus der Feature-Logic die beteiligten Objekte ermitteln und POIDBinder-Instanzen erzeugen, mit denen sie dann *checkBindings*() aufruft. Aufgrund der globalen Untersuchung ist hier die Rückgabe einfacher gestaltet, indem ein Integer-Wert verwendet wird. Ist er `0` sind keine Bindungen definiert, ist er `1` sind alle Bindungen gültig und bei `-1` existieren ungültige Bindungen.

```
1 public Set<String> objectsInLoadedDocument(Object context);
2 public int objectHasChangedSinceLastUpdateOrCommit(String poid)
3     throws WorkspaceException;
4 public Set<String>[] checkBindings(
5     Set<String> objectsInDocument, Set<PoidBinder> binders)
6     throws WorkspaceException;
7 public int checkBindingsInSandbox(String documentPid)
8     throws WorkspaceException;
```

Listing 4.28: Schnittstelle *Workspace*: Methoden für die Gültigkeitsprüfung von Objektabhängigkeiten

4.6.5 Konfliktbeseitigung

Methoden: An dieser Stelle sollen nicht die einzelnen Konfliktfälle behandelt werden, die aus der Gültigkeitsprüfung resultieren, sondern nur allgemeine Lösungsmöglichkeiten. Die nachfolgende Auflistung beschreibt drei mögliche Varianten.

- **Einzelfallprüfung:** Diese Methode ist sehr aufwändig, da jede ungültige Bindung sorgfältig vom Planer kontrolliert und einzeln bestätigt werden muss. Falls die Prüfung nicht mittels der Importfunktion in einem Dokument durchgeführt werden

kann, erhöht sich der Zeitbedarf nochmals. Das Setzen der Gültigkeit von `false` auf `true` innerhalb des übergegebenen *PoidBinder*-Objekts übernimmt die Methode *markBinderAsValid()* der Workspace-Schnittstelle aus Listing 4.29. Sie kann dabei nur auf formale nicht aber auf fachliche Gültigkeit prüfen.

- **Globales Setzen der Gültigkeit:** Im Gegensatz zur Einzelfallprüfung lassen sich mit dieser Methode alle Bindungen des gerade bearbeiteten Dokuments in einem Schritt unabhängig von der fachlichen Konsistenz auf gültig setzen. Empfehlenswert ist dieses Vorgehen lediglich, wenn der Planer die Änderungen an den bindenden Objekten überschaut und die Auswirkungen der Abhängigkeiten richtig einschätzt.

- **Löschen von Bindungen:** Wenn einzelne Objektabhängigkeiten im Laufe der Bearbeitung nicht mehr von Bedeutung sind, kann der Planer sie löschen. Andererseits müssen sie gelöscht werden, wenn das gebundende Objekt oder alle bindenden Objekte fehlen. Dazu steht im Workspace die Methode *deleteBindings()* zur Verfügung, die eine Menge von *PoidBinder*-Instanzen entgegennimmt und diese löscht.

Die vorgenommenen Änderungen werden erst durch die Operation *Speichern* persistent in die Sandbox übernommen und dann durch die Operation *Commit* auf dem Server eingetragen.

```
1 public void markBinderAsValid(PoidBinder poidBinder)
2     throws WorkspaceException;
3 public int markBindersAsValid() throws WorkspaceException;
4 public void deleteBindings(Set<PoidBinder> binders,
5     Object context) throws WorkspaceException;
```

Listing 4.29: Schnittstelle *Workspace*: Methoden für die Konfliktbeseitigung von Objektabhängigkeiten

4.7 Realisierung von Freigabeständen

4.7.1 Transiente Modellierung und Speicherung

Definition: Eine Freigabe ist eine Zusammenfassung von zueinander widerspruchsfreien Dokumentversionen, die einen bestimmten Stand im Projekt darstellen. Die Konsistenzbedingungen sind auf Seite 81 beschrieben. In der aktuellen Umsetzung ist es vorgesehen, dass Freigabestände in der Sandbox definiert werden. Dazu müssen hinzuzufügende Dokumente seit ihrem letzten Commit oder Update unverändert sowie, falls Objekte gebunden sind, die Bindungen gültig und alle bindenden Dokumente vorhanden sein. Realisierbar wäre auch eine Definition eines Freigabestandes ohne Umweg über den Workspace, nur wären dann Dokumente nicht so leicht einsehbar. Nicht betrachtet wurde die Einführung von Regeln und Rollen für verschiedene hierarchische Stufen von Nutzern.

Klassen für die Modellierung von Freigabeständen: Für das transiente Speichern von Freigabeständen wurde die im Klassendiagramm der Abbildung 4.23 dargestellte Klasse *ProjectState* entworfen. Als Attribute enthält sie die PID, den frei vergebbaren Namen, den Bearbeiter und das Datum des Freigabestands sowie die PVIDs der enthaltenen Dokumentversionen. Das Attribute *m_isMainRelease* legt fest, ob es ein nicht zur Veröffentlichung gedachter Freigabestand oder ein Hauptfreigabestand ist. Alle Attributwerte können mit den entsprechenden Get-Methoden abgefragt werden.

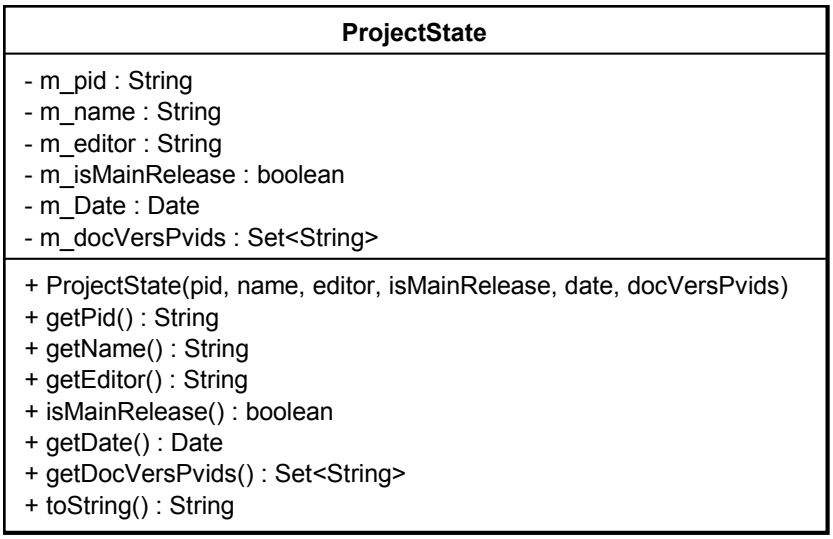


Abbildung 4.23: Klasse *ProjectState*

4.7.2 Modellierung und Speicherung im Repository

Feature-Logic: Freigabestände werden nicht in der Feature-Logic des Workspace gespeichert, sondern direkt in die Feature-Logic des Repositorys eingetragen. Abbildung 4.24 zeigt beispielhaft die Modellierung in der Feature-Logic. Der Freigabename L_1 steht dabei stellvertretend für eine PID mit dem Aufbau `L__Bearbeiter__UUID`. Das Feature *in* ordnet die Freigabe L_1 der Menge L zu und das Feature *isRelease* markiert sie durch Setzen des Wertes `true` als Hauptfreigabe. Die atomaren Features *name, editor* und *date* speichern die Metadaten für den Namen, den Bearbeiter und das Datum. Die Menge der zugehörigen Dokumentversionen $D_{1,4}$ und $D_{2,2}$ wird über das Mengenelement $\texttt{L}_1\texttt{\_SET}$ und die Hilfselemente $\texttt{L}_1\texttt{\_SET\_1}$ und $\texttt{L}_1\texttt{\_SET\_2}$ modelliert sowie über das Feature *docset* der Freigabe zugewiesen. Da im Repository mehrere Projekte gespeichert sind, muss noch über das Feature *project* das Projekt – in diesem Fall *Hochhaus_1* – zugeordnet werden.

4.7.3 Verwaltung im Workspace

Konzept: Freigabestände dienen einerseits dazu, um einen aktuellen Planungsstand als rechtskräftige Planungsgrundlage freizugeben, und andererseits, um später einen Pla-

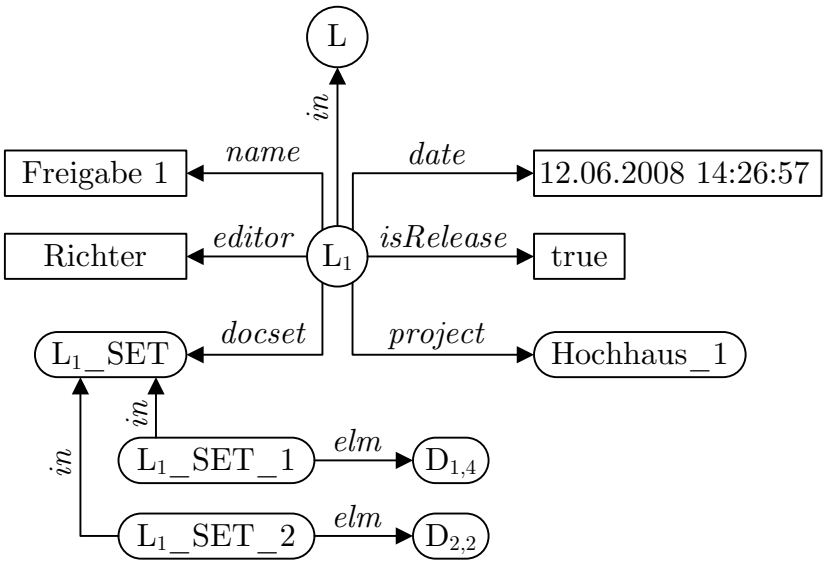


Abbildung 4.24: Modellierung einer Freigabe im Repository

nungsstand eines früheren Zeitpunkts, z. B. aus Beweisgründen, vorübergehend wiederherzustellen. Zu beachten ist dabei, dass Freigabestände nur angezeigt, aber nicht mehr verändert werden können. Das heißt, der Workspace muss die Definition von Freigabeständen sowie das Umschalten auf Freigabestände und zurück auf den aktuellen Planungsstand unterstützen.

Umsetzung: Normalerweise befinden sich im Workspace die aktuellen Versionen der Dokumente, die je nach Aufgabengebiet des Planers von ihm bearbeitet werden. Bevor er einen Freigabestand festlegen kann, müssen alle Dokumente, die enthalten sein sollen, versioniert und seit dem letzten Commit oder Update unverändert sein. Zusätzlich müssen die Gültigkeitsbedingungen für die Objektabhängigkeiten eingehalten werden.

Aus den in der Sandbox vorhandenen Dokumenten, die durch die eben genannten Forderungen in einer eindeutig adressierbaren Version vorliegen, wählt der Planer diejenigen Dokumente aus, die zum Freigabestand gehören sollen. Danach legt er den Namen und den Status des Freigabestands fest. Im Workspace wird dann die Methode *defineProjectState()* aus Listing 4.30 mit diesen Parametern aufgerufen, welche dann die Eintragung in die Feature-Logic des Repositorys vornimmt.

```
1 public Result defineProjectState(String stateName, boolean
    mainRelease, Set<String> docVersPvids) throws
    WorkspaceException;
2 public TreeMap<String,String> getProjectStates() throws
    WorkspaceException;
3 public Set<ProjectState> getProjectStatesComplete() throws
    WorkspaceException;
4 public void switchToProjectState(ProjectState projectState,
    Object context) throws WorkspaceException;
5 public boolean isProjectState();
```

```
6 public String getCurrentProjectStatePoid();
7 public String getCurrentProjectStateName();
8 public void switchProjectStateToHead(Object context) throws
     WorkspaceException;
9 public String getPvidOfLastDocumentVersion(String documentPid)
     throws WorkspaceException;
```

Listing 4.30: Schnittstelle *Workspace*: Methoden zur Verwaltung von Freigabeständen

Für das Umschalten auf einen bestimmten Freigabestand muss der Planer den gewünschten aus einer Tabelle aussuchen, wobei das System vorher mit der Methode *getProjectStatesComplete()* alle existierenden Freigabestände mit ihren Eigenschaften ermittelt. Nach der Auswahl wird eine Instanz der Klasse *ProjectState* erzeugt und an die Methode *switchToProjectState()* übergeben. Diese führt für alle betroffenen Dokumente – falls nötig – ein Update auf die vorgegebene Version durch und legt das ProjectState-Objekt im Attribut *m_projectState* der Klasse *WorkspaceAdapter* ab (Listing 4.31). In der Sandbox muss außerdem die Information über den aktuellen Freigabestand persistent hinterlegt und bei Bedarf wieder ausgelesen werden. Dies übernehmen die zwei Methoden *writeLocalProjectState*() und *readLocalProjectState*(), indem sie mittels der Java-Serialisierung das Attribut *projectState* in die Datei `projectstate` schreiben und wieder auslesen. Die Datei befindet sich im Verzeichnis `.project` unterhalb des Wurzelverzeichnisses der Sandbox.

```
1 protected ProjectState m_projectState;
2
3 protected void writeLocalProjectState() throws
     WorkspaceException {...}
4 protected void readLocalProjectState() throws
     WorkspaceException {...}
5 protected boolean resetLocalProjectState(){...}
```

Listing 4.31: Klasse *WorkspaceAdapter*: Attribute und Methoden zur Verwaltung von Freigabeständen

Das Wechseln zwischen Freigabeständen und die Rückkehr zum aktuellen Planungsstand ist jederzeit möglich. Letzteres übernimmt die Methode *switchProjectStateToHead()* aus der Workspace-Schnittstelle. Sie fragt für jedes im Freigabestand enthaltene Dokument mit der Methode *getPvidOfLastDocumentVersion()* die PVID der letzen Dokumentversion vom Server ab und aktualisiert auf diese, sofern sie nicht in der Sandbox vorliegt. Abschließend löscht die Methode *resetLocalProjectState()* die Datei `.project/projectstate`.

4.8 Handhabbarkeit versionierter Objektmodelle

Einleitung: Die vorangegangenen Abschnitte dieses Kapitels beschreiben unabhängig von einer bestimmten Anwendung die transiente und persistente Modellierung und Speicherung des verteilten Planungsmaterials sowie die dazu notwendigen Erweiterungen der Systemarchitektur von (Beer, 2005). Somit lassen sich objektorientierte Anwendungen, die eine dokumentierte Programmierschnittstelle zur Verfügung stellen, mit mehr oder weniger Aufwand für den verteilten Planungsprozess ertüchtigen. Jedoch kann eine Anwendung nur dann von Nutzen sein, wenn die Planer diese auch sinnvoll bedienen können.

4.8.1 Benutzerschnittstellen für verteilte Operationen

Konzept: Die verteilten Operationen wurden zunächst unabhängig von einer vorher festgelegten Benutzerschnittstelle implementiert, so dass unterschiedliche gewählt werden könnten (s. Seite 69). Für Operationen, die nur wenige textuelle Eingaben erfordern, ist durchaus eine Abfrage in der Kommandozeile (CLI) der Anwendung – sofern vorhanden – oder in einem einfachen Dialog ausreichend. Komplexe Zusammenhänge hingegen lassen sich am besten durch gut strukturierte grafische Benutzeroberflächen (GUI) darstellen und steuern.

In Tabelle 4.12 sind die verteilten Operationen mit der empfohlenen Benutzerschnittstelle aufgeführt. Die Operationen *Check-out, Laden, Speichern* und *Übertragen* nehmen nur ein Argument entgegen, so dass eine Abfrage in der Kommandozeile ausreichend ist. Für die anderen werden eigene grafische Dialoge entworfen, wobei der *Projektexplorer* günstigerweise für vier Operationen verwendbar ist. Die letzten drei Zeilen der Tabelle enthalten Operationen, die nicht zwingend für die Funktion der versionierten Umgebung erforderlich sind, aber dem Nutzer Informationen zur Dokumenthistorie sowie über den aktuellen Zustand des Projekts auf dem Repository und in der Sandbox anschaulich zeigen.

Umsetzung der grafische Dialoge: Die entworfenen Dialoge sollen unabhängig von einer bestimmten Anwendung sein und den für ihre Aufgabe vollständigen Funktionsumfang aufweisen. Später können sie durch Ableiten von Kindklassen an die Anforderungen spezieller Anwendungen angepasst werden. Für die Umsetzung wird die im Abschnitt 2.6.3 auf Seite 70 beschriebene Grafikbibliothek Swing verwendet.

Dialog Projektexplorer

Ziel: Mit Hilfe dieses Dialogs soll der Nutzer in die Lage versetzt werden, einen Überblick über das komplexe zentrale Projekt zu gewinnen. Durch die hierarchische Strukturierung, wie sie in Abschnitt 4.5 auf Seite 130 beschrieben ist, lässt sich das Projektmaterial in einem Baum darstellen. Jeder Knoten des Baumes entspricht einem Element in der Feature-Logic und besitzt bestimmte Eigenschaften oder Metadaten, die durch Feature-Wert-Paare beschrieben werden. Weiterhin soll innerhalb des Dialogs ein Selektionsteil die Filterung von Dokumentversionen anhand ihrer Metadaten erlauben.

Operation	UI	Dialogname bzw. Anzahl der Argumente
Check-out (Projekt beitreten)	CLI	1 Argument: Projekt
Selektion	GUI	Dialog Projektexplorer
Holen (Update New Documents)	GUI	Dialog Projektexplorer
Laden	CLI	1 Argument: Dokument
Speichern	CLI	1 Argument bei erstmaligem Speichern oder *Speichern unter*: Dokument
Vergleichen (Diff)	GUI	Dialog Diff
Aktualisieren (Update)	GUI	Dialoge Projektexplorer und Diff & Merge
Übertragen (Commit)	CLI	1 Argument: Tag
Freigeben	GUI	Dialog Freigabestand definieren
Freigabestand wechseln	GUI	Dialog Freigabestand auswählen
Informationsoperationen		
Projektüberblick	GUI	Dialog Projektexplorer
Dokumenthistorie	GUI	Dialog Dokumenthistorie
Sandboxstatus	GUI	Dialog Sandboxinfo

Tabelle 4.12: Benutzerschnittstellen für verteilte Operationen

Der Dialog soll sich außerdem bei Verwendung für die Operationen *Holen* und *Aktualisieren* unterschiedlich verhalten. Beim Holen wird nur die letzte Version aller noch nicht in der Sandbox befindlichen Dokumente eines Projekts angezeigt und der Nutzer kann daraus beliebige Dokumentversionen auswählen. Beim Aktualisieren werden alle Versionen eines Dokuments angezeigt, von denen der Nutzer nur eine auswählen kann.

Vorentwurf: Abbildung 4.25 zeigt den grafischen Vorentwurf des Dialogs mit insgesamt vier Bereichen. Links oben ist ein Baum mit den folgenden hierarchischen Stufen enthalten: Projekte, Anwendungen, Dokumente, Dokumentversionen, Objektversionen. Rechts daneben befindet sich eine Tabelle, die zum angewählten Knoten alle Eigenschaften in den zwei Spalten *Feature* und *Wert* anzeigt. Der linke untere Bereich dient der Definition von Constraints[10] zur Filterung von Dokumentversionen aus deren Metadaten *Bearbeiter, Versionsnummer, Tag* und *Datum*. Die entsprechenden Feature-Namen sind *editor, revision, tag* und *date*. Zur Definition eines Constraints ist die Auswahl eines Features, eines Operators und eines Werts notwendig. Zu diesem Zwecke wurde die Feature-Logic um Vergleichsoperatoren für atomare Werte erweitert, was im Abschnitt 4.4.2 auf Seite 122 beschrieben ist. Nach dem Festlegen von bis zu vier Constraints zeigt der rechte, untere Bereich des Dialogs alle Dokumentversionen an, die den Randbedingungen entsprechen.

Umsetzung: Abbildung 4.26 zeigt verwendete Swing-Komponenten und neu entworfene Klassen. Zentrale Klasse ist das von *JPanel* abgeleitete *ProjectExplorerPanel*. Die Aufteilung des Panels in vier Bereiche wird durch drei geschachtelte *JSplitPanes* erreicht, wobei

[10] constraint ⟨engl., „Einschränkung, Randbedingung“⟩

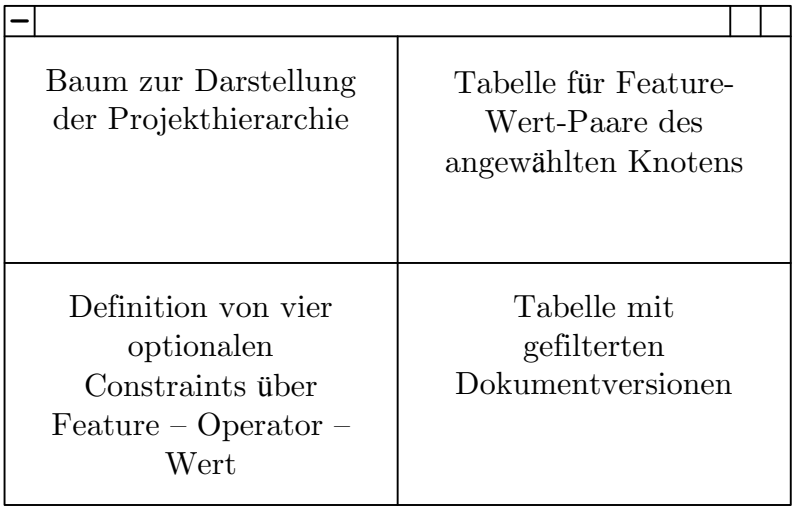


Abbildung 4.25: Dialog Projektexplorer: Vorentwurf

der Selektionsteil nicht Bestandteil des ProjectExplorerPanels ist. Dieser wird durch Ableiten einer Klasse *SelectionPanel* vom ProjectExplorerPanel hinzugefügt. Im oberen Bereich befindet sich das *ProjectTreePanel* mit dem Projektbaum und das *FeatureTablePanel* mit Feature-Wert-Tabelle.

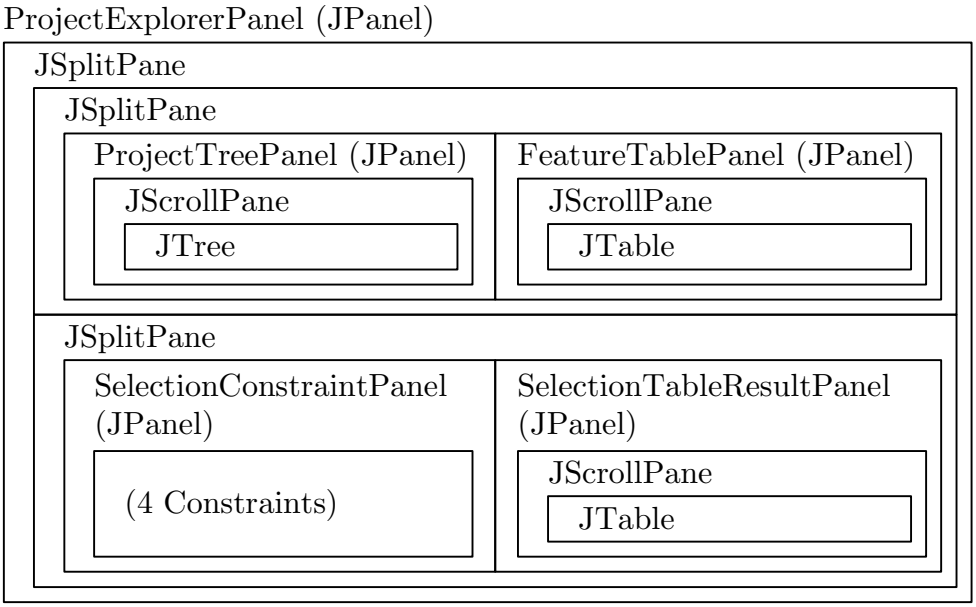


Abbildung 4.26: Dialog Projektexplorer: Verwendete Klassen für die Umsetzung

Die Daten für den Dialog ermittelt die Methode *populateTree()* von der zentralen Feature-Logic, wobei durch Parameter im Konstruktor der Klasse das Einlesen auf bestimmte Projekte, Anwendungen und Dokumente beschränkt werden kann. Die Feature-Terme `in:P` und `in:A` liefern alle Projekte und Anwendungen zurück. Alle Dokumente eines Projekts, die mit einer bestimmten Anwendung bearbeitet wurden, ermittelt der Feature-Term `[in:D,proj:Projektname,app:Anwendungsname]`. Die Dokumenthistorie soll im

Baum in der korrekten zeitlichen Reihenfolge angezeigt werden. Dazu wird zuerst die Startversion mit `[doc:DokumentPID,[[in:V,src:start].dst]]` und danach Schritt für Schritt die Folgeversionen mit `[in:V,src:DokumentPVID].dst` ermittelt. Zum Nachvollziehen der Versionsabfragen eignet sich Abbildung 4.17 auf Seite 137. Schlussendlich werden für jede Dokumentversion die enthaltenen Objektversionen mit dem Feature-Term `[in:[DokumentPVID.compset]].elm` abgefragt und vor dem Hinzufügen zum Baum nach ihrer POVID sortiert.

An dem kleinen Projektbeispiel in Abbildung 4.27 ist der Aufbau des Projektexplorerdialogs gut zu erkennen. Der Baum links oben enthält ein Projekt *Hochhaus* mit den zwei Anwendungen *cademia* und *loadTakeDownCADEMIA*. Die erste Anwendung enthält die zwei Dokumente *Geo* sowie *Geometrie* und die zweite Anwendung das Dokument *LTD*. Bei *Geo* und *LTD* sind drei Versionen mit den Namen *V1, V2, V3* unterhalb des Knotes zu sehen. Für den angewählten Knoten *V1* von *Geo* sind in der Tabelle rechts daneben die Eigenschaften dargestellt. Die Version *V1* enthält zwei Objekte, und zwar einen Kreis und eine Linie. Die POID-Teil vor dem Klassennamen wurde zur besseren Übersichtlichkeit abgeschnitten.

Die Funktionsweise der Selektion soll am gleichen Beispiel erklärt werden. Alle Daten werden zusätzlich in der Klasse *ProjectExplorerPanel* in einer transienten Feature-Logic-Instanz gespeichert, um die Selektionsabfragen ohne Netzzugriff durchführen zu können. Zunächst ist zu erkennen, dass im linken unteren Bereich des Dialogs bis zu vier Constraints definiert werden können. Die ersten zwei sind aktiv geschaltet, was an den Häkchen in den Checkboxen zu erkennen ist. Ein Constraint besteht immer aus einem Feature, einem Operator und einem Wert. Zur einfachen Festlegung durch den Nutzer wurden pro Constraint drei Comboboxen erzeugt, wobei die erste mit den Features *date, editor, revision* und *tag* vorbelegt ist. Die anderen beiden werden dynamisch mit den je nach Datentyp verfügbaren Operatoren und passenden Werten gefüllt.

Falls keine Constraints aktiviert sind, zeigt die Tabelle unten rechts alle existierenden Dokumentversionen des Projekts an und ermöglicht wahlweise die Sortierung nach einer der vier Spalten. Der Feature-Term zur Ermittlung aller Dokumentversionen ist in diesem Fall `doc:[in:D]`. Der erste Constraint aus dem Beispiel schränkt die Dokumentversionen auf diejenigen ein, die genau um 10:13:41 Uhr am 9.6.2008 oder später zum Server committet wurden. Der Feature-Term des Constraints lautet dann: `date>=09.06.2008_10:13:41`. Der zweite Constraint soll nur die Dokumentversionen erlauben, die vom Autor `Torsten Richter` erstellt wurden. Da der Datentyp des Autors ein `String` ist, muss intern zunächst das primitive Element für das Atom `Torsten Richter` ermittelt werden. Angenommen, dass es den Namen `_1` trägt, ergibt sich für den Feature-Term: `editor:_1`. Nach Ermittlung aller Feature-Terme müssen diese mittels des Operators *Schnittmenge* zu einem verknüpft werden, der dann als Abfrage an die transiente Feature-Logic gestellt werden kann: `[doc:[in:D],date>=09.06.2008_10:13:41,editor:_1]`.

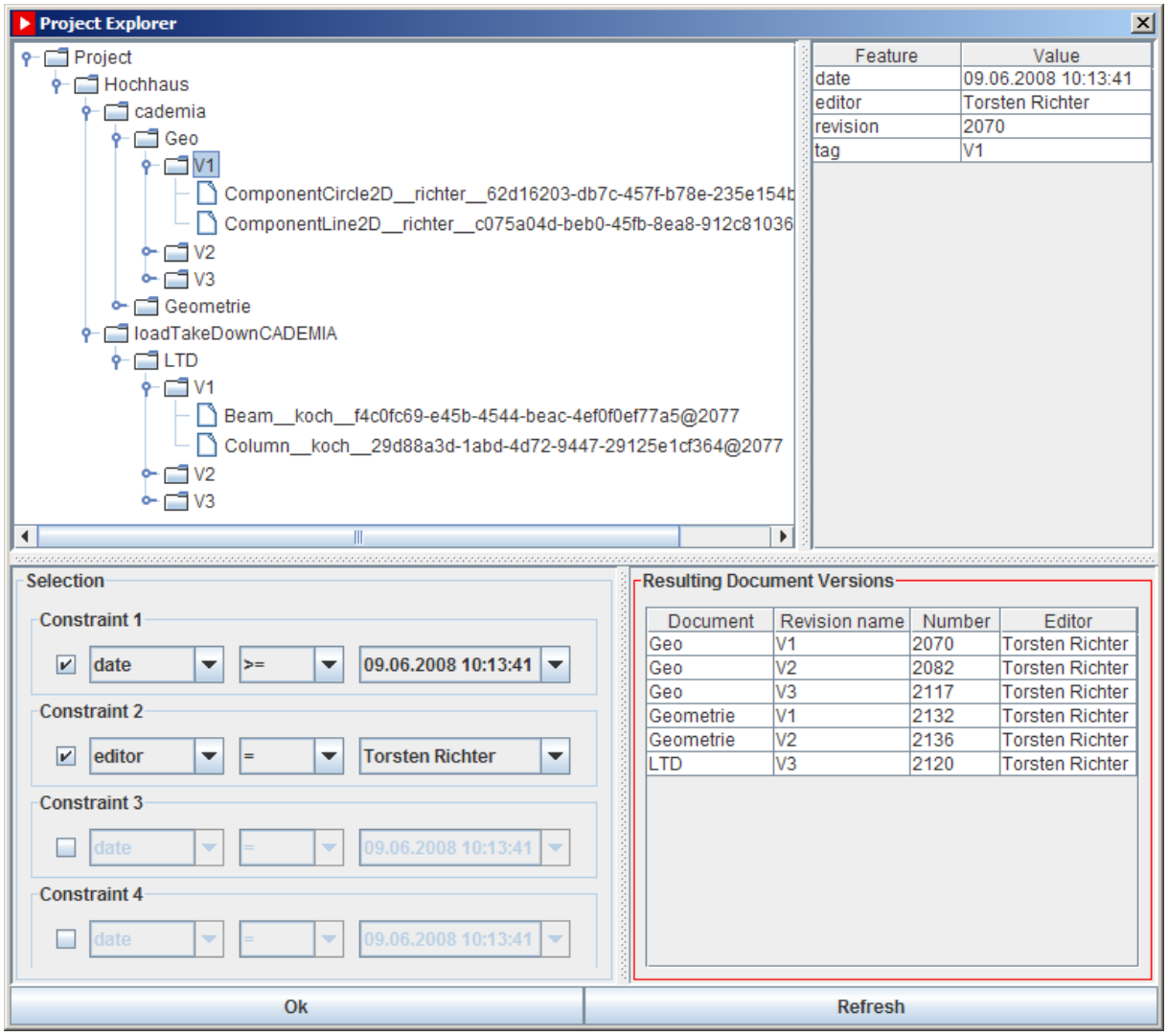


Abbildung 4.27: Dialog Projektexplorer: Screenshot

Dialog Dokumenthistorie

Ziel: Mit Hilfe dieses Informationsdialogs soll die Geschichte des Dokuments veranschaulicht werden. Im Falle der linearen Versionierung ist eine Tabelle ausreichend, die wieder die bekannten Metadaten der Dokumentversionen enthält.

Umsetzung: Abbildung 4.28 zeigt den einfachen Aufbau dieses Dialogs. Für das schon bekannte Dokument *Geo* sind die Metadaten der drei auf dem Server vorhandenen Dokumentversionen dargestellt. Die letzte ist farblich leicht hervorgehoben, da sie sich in der Sandbox befindet. Zusätzlich gibt die vorletzte Spalte an, wie viele Dokumentversionen an die jeweilige Dokumentversion gebunden sind, und die letzte Spalte, von wie vielen sie abhängig ist.

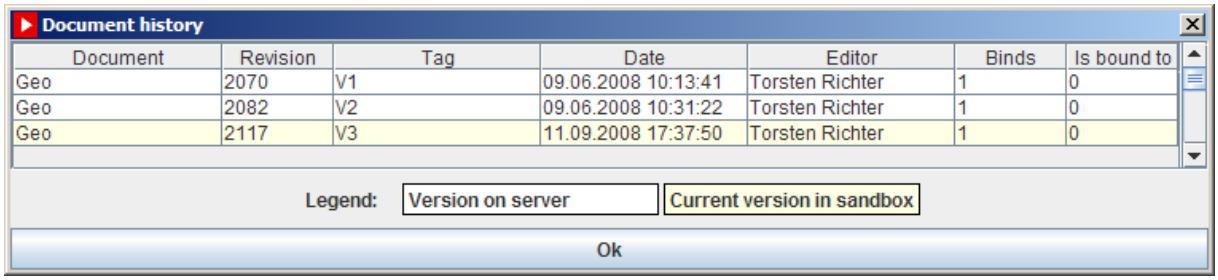


Abbildung 4.28: Dialog Dokumenthistorie: Screenshot

Dialog Sandboxinfo

Ziel: Ziel dieses Dialoges ist es, einen Überblick über den Zustand aller Dokumente in der Sandbox zu gewinnen. Für den Zustand eines Dokuments sind der Versionierungs- und Änderungszustand sowie die Gültigkeit eventuell vorhandener Bindungen von Bedeutung.

Umsetzung: Im Gegensatz zu den vorherigen zwei Dialogen ist für die Datenermittlung kein Netzzugriff erforderlich, da alle Daten aus der Sandbox abgefragt werden. Auch hier ist die Dialoggestaltung einfach gehalten, indem eine Tabelle für die Präsentation der Daten genutzt wird. Zunächst wird für jedes Dokument eine Zeile angelegt und in jede der acht Spalten der entsprechende Wert eingetragen (s. Abbildung 4.29).

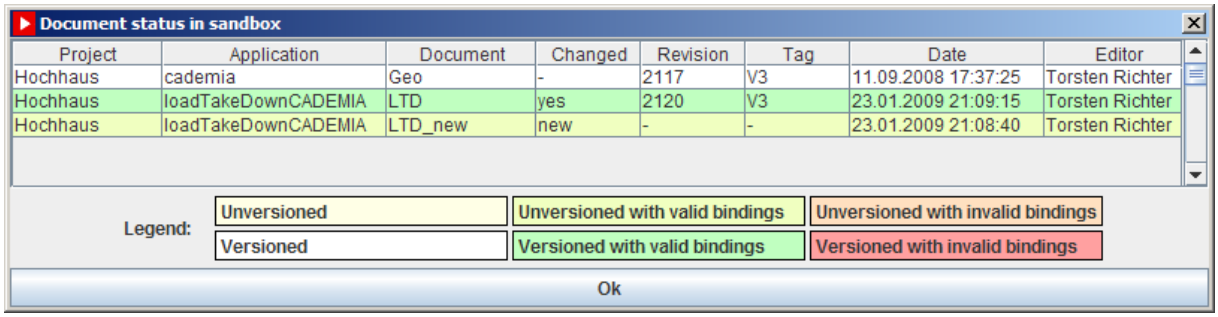


Abbildung 4.29: Dialog Sandboxinfo: Screenshot

Die farbliche Gestaltung der Zeilen dient als zusätzliches Gestaltungselement zur Darstellung des Versionierungs- und Bindungszustands. Für den Bindungszustand spielen alle die Bindungen eine Rolle, die auf Objekte des Dokuments verweisen. Das Farbkodierungsschema der Zeilen kann Tabelle 4.13 entnommen werden.

	Keine Bindungen	Alle Bindungen gültig	Mind. eine Bindung ungültig
Unversioniert	hellgelb	gelb-grün	orange
Versioniert	weiß	grün	rot

Tabelle 4.13: Dialog Sandboxinfo: Farbkodierungsschema der Zeilen

Dialog Freigabestand definieren

Ziel: Dieser Dialog dient zur Definition von Freigabeständen auf Basis unveränderter Dokumentversionen in der Sandbox, wie es im Abschnitt 4.7 auf Seite 148 beschrieben ist.

Umsetzung: Vor der Anzeige des Dialogs werden versionierte Dokumente, die Änderungen in der Sandbox erfahren haben, aussortiert und dem Nutzer in einer Meldung angezeigt. Alle anderen Dokumentversionen erscheinen in einer Tabelle, wobei diejenigen mit gültigen Bindungen grün und mit ungültigen Bindungen rot markiert werden (s. Abbildung 4.30). Die darunterliegende Combobox enthält alle bisher definierten Freigabestände, wie sie von der Methode *getProjectStates()* der Schnittstelle *Workspace* zurückgeliefert wurden. Somit lässt sich sowohl für eine schnelle Eingabe ein alter Freigabestandname einblenden und modifizieren als auch einfach kontrollieren, ob der neue Freigabestandname noch nicht im Projekt existiert. Der Dialog bietet weiterhin die Möglichkeit, den Freigabestand als Haupt- oder Nebenfreigabestand über die Checkbox festzulegen. Nach dem Drücken der Ok-Taste prüft der Dialog, ob die Bindungen zu Dokumentversionen mit gebundenen Objekten gültig und die zugehörigen Dokumentversionen mit den bindenden Objekten markiert bzw. im Freigabestand enthalten sind. Falls die Prüfung fehlschlägt, erscheint eine Fehlermeldung mit entsprechenden Hinweisen.

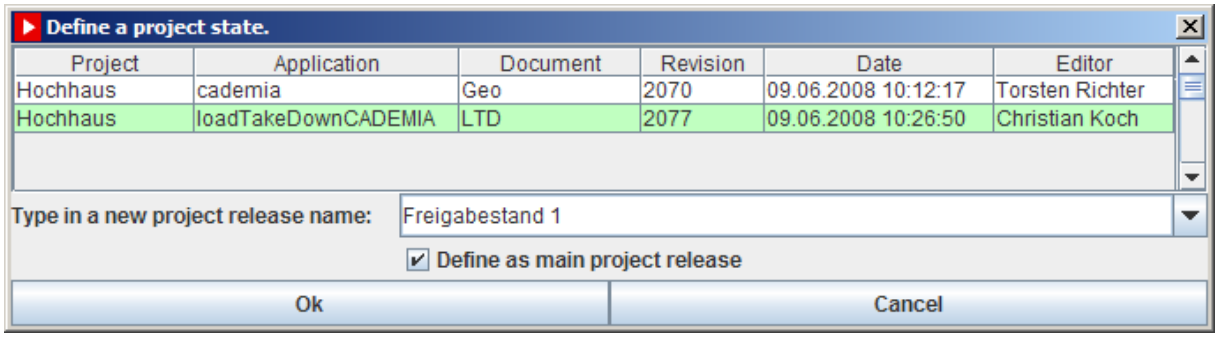


Abbildung 4.30: Dialog Freigabestand definieren: Screenshot

Dialog Freigabestand auswählen

Ziel: Für den Wechsel zu einem Freigabestand des Projekts ist dieser Dialog vorgesehen. Der Nutzer wählt aus den verfügbaren Freigabeständen einen aus, dessen Dokumentversionen dann vom Server in die Sandbox geladen werden.

Umsetzung: Die Umsetzung dieses Dialogs ist sehr einfach, da nur eine Tabelle für die Freigabestände und ihren Metadaten benötigt wird (s. Abbildung 4.31). Die Auswahl ist auf eine Zeile begrenzt, die mit Betätigen der Schaltfläche *Ok* übernommen wird.

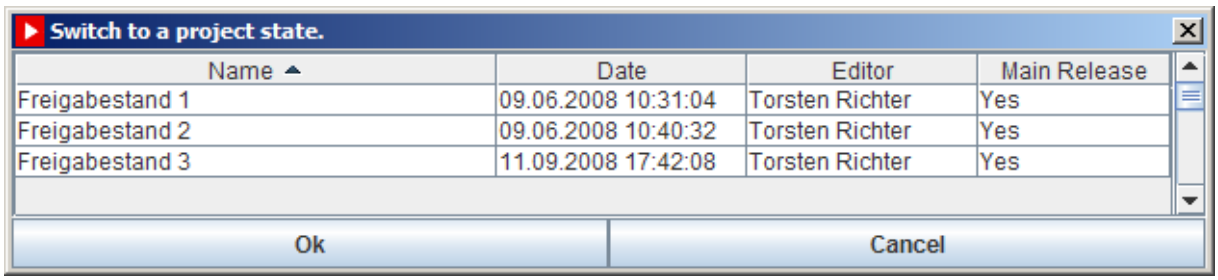


Abbildung 4.31: Dialog Freigabestand auswählen: Screenshot

4.8.2 Vergleich und Zusammenführung versionierter Objektmodelle

Vergleich: Wie in Abschnitt 2.5.4 auf Seite 60 erläutert wurde, ist nur ein anwendungsbezogener Vergleich sowie eine nachfolgende Zusammenführung sinnvoll. Deshalb sollen an dieser Stelle nur die Grundlagen beschrieben werden. Ausgehend von der manuellen Serialisierung in ZIP-Dateien aus Abschnitt 4.3.4 auf Seite 113 besteht die Möglichkeit, die CRC-Prüfsummen der ZIP-Einträge zum Vergleich heranzuziehen. Die auftretenden Fälle dokumentiert Tabelle 3.5 auf Seite 92.

Als Zwischenschritt zum anwendungsbezogenen Vergleich wurde ein Dialog zum allgemeinen Vergleichen von ZIP-Dateien entworfen. Abbildung 4.32 zeigt ihn beim Vergleich von zwei Versionen eines CAD-Dokuments. Die Tabellen zeigen für jedes ZIP-Archiv die Namen der Einträge, deren Größe in Byte und den CRC-Wert. Der Name entspricht im Fall eines Objekts seiner POID. Die verschieden Zustände sind für eine schnellere Erfassung farbig hervorgehoben

Zustand eines ZIP-Eintrags in beiden ZIP-Archiven	Farbe
Unverändert	Schwarz
Geändert	Rot
Nur links vorhanden	Blau
Nur rechts vorhanden	Gelb

Tabelle 4.14: Farbige Hervorhebung von ZIP-Einträgen beim Vergleichen

Ein Anklicken eines ZIP-Eintrags auf der linken oder rechten Seite öffnet jeweils ein nichtmodales Textfenster mit dem serialisierten Inhalt. Im Beispiel sind diese beiden Fenster im unteren Bereich für ein Kreis-Objekt zu sehen. Der einzige Unterschied besteht in der x-Koordinate des Mittelpunkts, angegeben in Weltkoordinaten.

Zusammenführung: Auf der Ebene der ZIP-Einträge würde die Änderung des Inhalts dem direkten Manipulieren des Objektzustands ohne Prüfung auf Konsistenz entsprechen. Für das Zusammenführen kann auch das Zusammenstellen eines neuen ZIP-Archivs aus

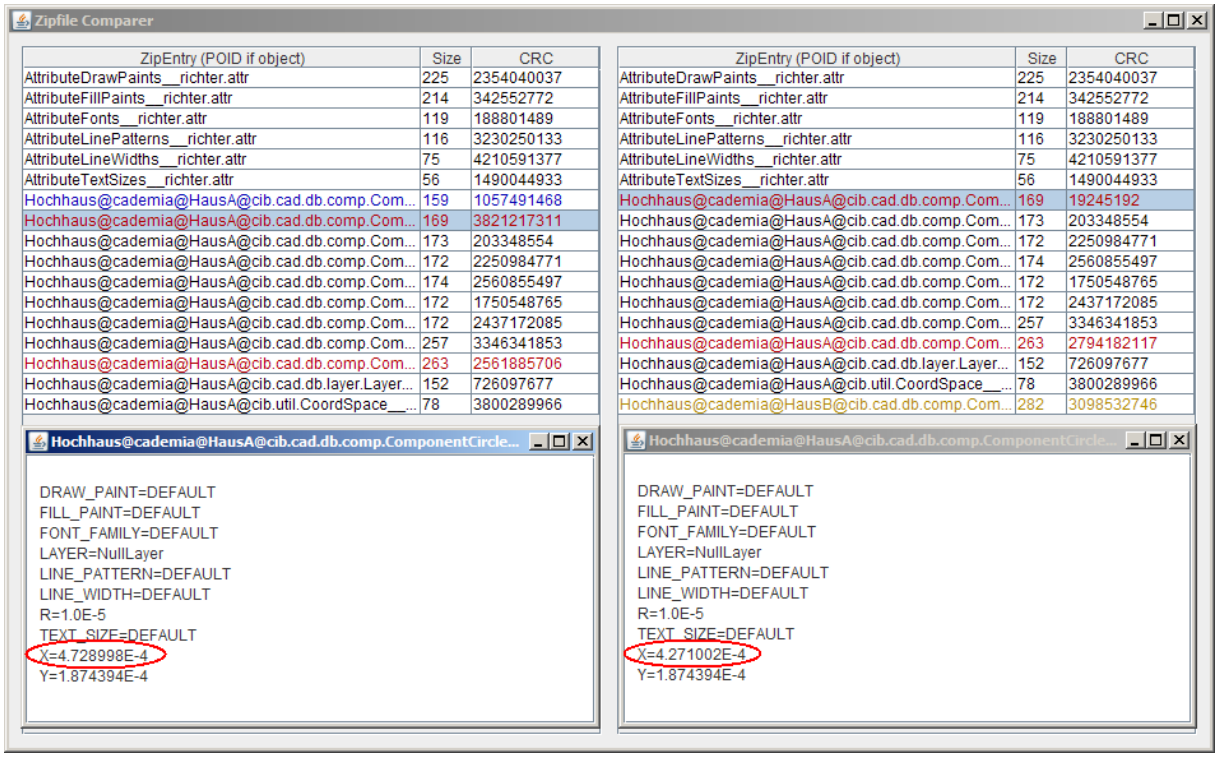


Abbildung 4.32: Vergleichsdialog für serialisierte ZIP-Dateien

beliebigen ZIP-Einträgen zu Widersprüchen innerhalb des Datenmodells führen. Deshalb sollte diese Operation unter Einbeziehung der Anwendungslogik durchgeführt werden.

Bindungen sind in der Sandbox zum einen in der lokalen Feature-Logic und zum anderen in der Datei *bindings.txt* im Verzeichnis des gebundenen Dokuments modelliert. Ein Zusammenführen sollte ebenfalls nur innerhalb der Anwendung erfolgen.

4.9 Leistungsverbesserung

4.9.1 Begriffe

Leistung: Leistung⟨auf engl., „performance“⟩ ist in der Informatik nach (Claus u. Schwill, 1993) die *„Geschwindigkeit und Qualität mit der ein Auftrag oder eine Menge von Aufträgen von einer Datenverarbeitungsanlage verarbeitet wird.“*. Wichtige Messgrößen zur Bestimmung der Leistung sind Durchsatz, Antwortzeit und Verfügbarkeit. Zum objektiven Leistungsvergleich verschiedener Rechner werden Bewertungsprogramme, besser bekannt als *benchmark programs*, herangezogen. Mit ihnen werden zum Beispiel die Anzahl der Gleitkommaoperationen pro Sekunde (FLOPS)[11] gemessen.

[11] FLOPS = Floating Point Operations Per Second

Die Entwicklung der Rechenleistung von Prozessoren verdeutlicht Tabelle 4.15. Der erste funktionsfähige Digitalrechner der Welt, die Zuse Z3, erreichte gerade 2 Gleitkommaoperationen pro Sekunde, was aber schon ein Fortschritt gegenüber der mühseligen Handrechnung bedeutete[12]. Eine aktuelle CPU für Arbeitsplatzrechner kann beachtliche 23,3 Mrd. Operationen pro Sekunde durchführen. Der momentan schnellste Supercomputer ist der IBM Roadrunner am Los Alamos National Laboratory in den USA mit einer Leistung von 1,026 Peta-FLOPS ($= 1,026 \cdot 10^{15}$ FLOPS), der aus 12.960 Cell- und 6.480 AMD-Opteron-Prozessoren aufgebaut ist. Die Cell-Prozessoren tragen ca. 96 % zur Rechenleistung bei, da die Opteron-Kerne nur für die Verwaltung der Cell-Prozessoren verantwortlich sind.

Prozessor	Baujahr	Gleitkommaoperationen
Zuse Z3	1941	2 FLOPS
Intel 80286 8 MHz	1986	1 MFLOPS
Intel 80486DX4 100 MHz	1994	35 MFLOPS
Intel Pentium 4 650 3,2 GHz	2004	9.700 MFLOPS
Intel Core 2 Duo E8400 Wolfdale 3,0 GHz	2008	23.300 MFLOPS
IBM Roadrunner	2008	1,026 Peta-FLOPS

Tabelle 4.15: Rechenleistung ausgewählter Prozessoren

Algorithmus: Ein Algorithmus ist eine Verarbeitungsvorschrift, die die auszuführenden Schritte genau vorgibt. Algorithmen sind durch bestimmte Eigenschaften gekennzeichnet.

- **Determiniertheit:** Für gleiche Eingangsparameter muss das gleiche Ergebnis erzielt werden.
- **Finitheit:** Statische Finitheit bedeutet, das die Handlungsvorschrift in seiner Länge begrenzt ist, während dynamische Finitheit während der Ausführung einen begrenzten Speicherbedarf fordert.
- **Terminierung:** Ein Algorithmus muss nach für jedem Eingabe nach einer endlichen Anzahl von Schritten zu einem Ergebnis kommen. Eine Ausnahme bilden zum Beispiel Steuerungs- und Betriebssysteme, die bewusst in einer Endlosschleife laufen.
- **Determinismus:** Ein Algorithmus ist deterministisch, wenn zu jedem Zeitpunkt der Ausführung der nächste Schritt eindeutig festgelegt ist.

[12] Die Z3 wurde von dem Bauingenieur Konrad Zuse zusammen mit Helmut Schreyer gebaut. Vorgänger war das mechanische Rechenwerk Zuse Z1, das schon viele Konzepte der Z3 enthielt und für die aufwändige und monotone Berechnung in der Flugstatik zum Einsatz kam. Zuse arbeitete zu dieser Zeit bei den Henschel Flugzeug-Werken in Berlin. Die Z1 verwendete das binäre Zahlensystem und las die Programme mittels Lochstreifen ein. Die Mechanik erwies sich aber im praktischen Einsatz als unzuverlässig.

Komplexität: Laut (Claus u. Schwill, 1993) ist die Komplexität einer berechenbaren Funktion f *„der zu ihrer Berechnung erforderliche Aufwand an Betriebsmitteln wie Speicherplatz, Rechenzeit, [...]. Die Komplexität einer Funktion ist die Komplexität des bestmöglichen aller Algorithmen.“*. Die Komplexität eines Algorithmus A ist dagegen *„der erforderliche Rechenaufwand bei einer konkreten Realisierung des Algorithmus innerhalb des Berechnungsmodells.“*. Die Komplexität von A ist somit die obere Schranke der Komplexität von f und die Komplexität von f ist die untere Schranke der Komplexität von A.

Interessant für die Komplexität von Algorithmen ist oft nur die Komplexitätsklasse, welche durch die *Ordnung* beschrieben wird. Die Ordnung (Größenordnung) wird nur qualitativ mit dem *Landauschen Symbol O* für eine Eingabe der Länge n angegeben (s. Tabelle 4.16). Der Ausdruck $t(n) = O(n)$ steht dann für die Aussage: Der Algorithmus hat eine lineare Laufzeit. Der Funktion $s(n)$ steht stellvertretend für den Speicherplatzbedarf. Weiterhin wird bei der Ausführung von Algorithmen zwischen den Fällen *best case*, *average case* und *worst case* unterschieden.

Komplexitätsklasse	Beispiel für die Ordnung
konstant	$O(1)$
logarithmisch	$O(\log(n))$
logarithmisch-linear	$O(n \cdot \log(n))$
linear	$O(n)$
quadratisch	$O(n^2)$
kubisch	$O(n^3)$
exponentiell	$O(e^n)$
Fakultät	$O(n!)$

Tabelle 4.16: Ausgewählte Komplexitätsklassen

Messmethoden: Zur Erfassung von Zeit- und Speicherbedarf stehen unterschiedliche Methoden zur Verfügung.

- **Handmessung:** Diese Variante ist von allen die fehleranfälligste. Für die Zeitmessung ist die Genauigkeit durch die Reaktionsfähigkeit des Messenden eingeschränkt. Diese Methode ist daher nur für eine erste Einschätzung brauchbar. Die Erfassung des Speicherbedarfs kann von außen nur über Software stattfinden, die den aktuellen Speicherbedarf des Programms wiedergibt. Die Berechnung der Speicherveränderung über einem Zeitraum muss durch Subtraktion der beiden zu Beginn und am Ende abgelesenen Werte erfolgen.
- **Automatisch im Quellcode:** Java bietet verschiedene Methoden zur Messung der Zeit und des Speicherbedarfs an. Die Methode *System.currentTimeMillis()* ermittelt die vergangene Zeit in Millisekunden seit dem 1.1.1970. Durch die Ermittlung und Subtraktion von End- und Startwert erhält man die vergangene Zeit. Die Methode *System.nanoTime()* erreicht eine höhere Genauigkeit basierend auf dem Zeitgeber des Rechners (System-Timer) und gibt einen Zeitwert in Nanosekunden zurück.

Für die Messung des Speichers stellt Java die Methoden *Runtime.getRuntime().totalMemory()* und *Runtime.getRuntime().freeMemory()* zur Verfügung. Die Differenz aus dem Wert der ersten und der zweiten Methode ergibt den tatsächlich benutzten Speicher in der Virtuellen Maschine.

- **Profiler:** Profiler sind Software-Werkzeuge, die das Laufzeitverhalten von Programmen untersuchen. Sie verlangsamen zwar die Ausführungsgeschwindigkeit des Programms, geben aber den prozentualen Anteil einzelner Methoden an der Laufzeit in einem festlegbaren Intervall an. Durch eine Baumdarstellung können ausgehend von einer Startmethode alle aufgerufenen Methoden eingesehen werden. Profiler eignen sich sehr gut, um zeit- und speicherkritische Stellen in einer Software zu finden.

4.9.2 Leistungsverbesserung der Systemarchitektur

Einführung: Für die Akzeptanz von Software in der Praxis spielt neben der eigentlichen Problemlösung und einer ergonomischen Bedienung auch die performante Ausführung eine nicht unwesentliche Rolle. Zum einen sind dafür die absoluten Ausführungszeiten einzelner Operationen und zum anderen die Komplexität der Operationen von Bedeutung. Außerdem sollte der benötigte Arbeitsspeicher im Rahmen durchschnittlich ausgestatteter Rechner bleiben, die für das Aufgabengebiet ausgelegt sind. Für die vorliegende Systemarchitektur zur Objektversionierung sind im Wesentlichen folgende Operationen zeitkritisch und daher zu untersuchen und im Bedarfsfall zu optimieren.

- Speichern und Laden von Dokumenten in der Sandbox
- Übertragung von Dokumentversionen zwischen Server und Client
- Abfrage von Daten aus der zentralen Feature-Logic auf dem Server

Speichern und Laden

Serialisierungsverfahren: Dieser Punkt wurde schon im Abschnitt 4.3.4 auf Seite 113 ausführlich diskutiert, was zu dem Konzept der manuellen Serialisierung in ZIP-Archive führte, das eine lineare Komplexität aufweist.

Objekt-Features: Im ursprünglichen Ansatz für die Objektversionierung von (Firmenich, 2002) war eine sehr flexible Selektion von Objekten anhand ihrer Eigenschaften (Features) zur Bildung von Teilmodellen vorgesehen. Für geometrische Datenmodelle im Bauwesen sollte so zum Beispiel eine Auswahl aller Objekte, die sich in einer bestimmten Etage befinden, möglich sein. Mit der Einführung von Dokumenten ist diese Art der Objekt-Selektion nicht mehr von elementarer Bedeutung.

Der größte Nachteil besteht beim Speichern der Objekteigenschaften in einem erhöhten Speicherbedarf und einer Beeinträchtigung der Feature-Logic-Performance, wenn viele große Modelle in der Sandbox vorliegen. Tabelle 4.17 zeigt für die Feature-Logic im Dateisystem (FLFS) die Anzahl der geschriebenen Zeilen für jede Relation bei Speicherung eines

Zeichnungsdokuments mit 10.000 Linien. Bei der verwendeten CAD-Anwendung CADEMIA sind für jedes Objekt im Durchschnitt 10 Features zu speichern. Durch das Weglassen der Objekt-Features reduziert sich die Anzahl der primitiven Elemente von 5.005 auf 3, was direkt in den Relationen *Domain* und *Atom* zu sehen ist. Außerdem reduziert sich die Anzahl der Tupel in der Relation *Relslot* auf fast ein Drittel. Pro Objekt werden nur noch 3 Tupel benötigt: Eins für die Zuweisung des Datums und zwei zur Modellierung der Zugehörigkeit zur Dokumentmenge `COMP_SET` über ein Hilfselement (s. Abschnitt 4.5 auf Seite 130). Durch die auf Seite 128 beschriebene Strukturierung der Textdatei `Relslot` werden pro Objekt 7 Zeilen benötigt, was bei 10.000 Objekten 70.000 Zeilen ergibt. Die drei gespeicherten Atome in der rechten Variante entsprechen dem Dokumentnamen, dem Namen des Bearbeiters und dem Datum.

Anzahl Zeilen in der Datei	Mit Speicherung der Objekteigenschaften	Ohne Speicherung der Objekteigenschaften
Domain	25.016	20.014
Feature	31	18
Atom	5.005	3
Relslot	200.014	70.014
Speicherzeit	9,3 s	8,1 s
Speicherbedarf	10.989 kB	4.402 kB
Ladezeit ZIP	1,50 s	1,47 s
Ladezeit insg.	3,27 s	3,25 s

Tabelle 4.17: Speicherbedarf der FLFS im Dateisystem für 10.000 Objekte

Speicher- und Ladeperformance: Zur Bewertung der Speicher- und Ladezeiten wurden Dokumente mit der in Tabelle 4.18 angegebenen Objektanzahl verglichen[13]. Die Komplexität zeigt in beiden Fällen einen linearen Verlauf. Das Gleiche trifft für die Entwicklung des Speicherplatzbedarfs der ZIP-Archive zu. Die Ladezeiten sind sowohl für das Deserialisieren des Objektmodells aus dem ZIP-Archiv und als auch für den gesamten Vorgang inklusive Visualisierung angegeben.

Anzahl Objekte	100	1.000	10.000	100.000
Speicherzeit insgesamt	0,42 s	0,97 s	8,14 s	72,3 s
Ladezeit ZIP	0,11 s	0,25 s	1,47 s	13,7 s
Ladezeit insgesamt	0,20 s	0,48 s	3,25 s	34,0 s
Größe des ZIP-Archivs	45 kB	438 kB	4.384 kB	44.023 kB

Tabelle 4.18: Speicher- und Ladezeiten für verschiedene Dokumentgrößen

[13] Die Messungen wurden auf einem Rechner mit einem Intel Pentium 4 3,2 GHz, 3 GB RAM und einer 160 GB Festplatte (ST3160023AS, 7.200 U·min^{-1}, 12,8 ms mittlere Zugriffszeit) vorgenommen.

Das Speichern bestehender Dokumente wird durch Nutzung der im ZIP-Archiv enthaltenen CRC-Werte optimiert. Während des Ladens speichert die Map *m_mapPoidCrc* vom Typ `Map<String,Long>` im Workspace die Paare aus Objekt-POID und CRC-Wert aller ZIP-Einträge. Beim Speichern wird für jedes Objekt der neu ermittelte CRC-Wert mit dem alten CRC-Wert aus der Map verglichen und nur bei einer Änderung muss das Datums-Feature des Objekts in der Feature-Logic aktualisiert werden.

Speicherbedarf: Für den Speicherbedarf im Arbeitsspeicher und im Dateisystem wurden zwei CAD-Dokumente mit 10.000 und 100.000 Linien verglichen. Anhand der Ergebnisse in Tabelle 4.19 ist sowohl für den Speicherbedarf als auch für die Speicherzeiten eine lineare Komplexität zu erkennen. Nur das Hinzufügen der Linien ohne eine Visualisierung scheint überproportional mehr Arbeitsspeicher zu benötigen, was aber eine Frage der Umsetzung in der Anwendung ist.

Die Größe des ZIP-Archivs ist mit ca. 42,9 MB bei 100.000 Linien noch annehmbar, zumal das Versionsverwaltungssystem Subversion spätere Änderungen an den Dokumenten effizient speichert (s. Abschnitt 4.3.4 auf Seite 113). Hinzu kommen noch 46,0 MB für die persistente Speicherung der Feature-Logic. Zum Vergleich wurden in den letzten beiden Zeilen der Tabelle der Platzbedarf einer DXF-Datei und der in ein komprimiertes ZIP-Archiv serialisierten FLFS aufgeführt.

Nach dem Speichern liegen die Daten der Dateisystem-Feature-Logic zusätzlich transient im Arbeitsspeicher vor, was bei 100.000 Linien 186,6 MB benötigt. Bei größeren Datenmodellen übersteigt damit der Arbeitsspeicherbedarf die momentane Ausstattung aktueller Bürorechner. Als Alternative wurde die Datenbank JavaDB, die im JRE enthalten ist und auf dem Produkt Apache Derby basiert, als lokaler Ersatz getestet. Jedoch beträgt die Speicherzeit bei 1000 Linien 16,1 s, was ca. 22 mal länger dauert als die bisherige Umsetzung. Abhilfe könnte das Kapseln durch einen Wrapper[14] bringen, der die Relationen *Domain, Feature* und *Atom* für einen schnelleren Zugriff transient vorhält und den Zugriff für mehrere Anwendungen durch einen RMI-Server ermöglicht. Diese Lösung würde zusätzlich ein inhärentes Transaktionskonzept bieten.

Datenübertragung zwischen Server und Client

Feature-Logic auf dem Server: Ein modernes Relationales Datenbankmanagementsystem bietet mehrere Möglichkeiten, den Zugriff auf die Datenbank zu beschleunigen, die im Folgenden vorgestellt und auf die Feature-Logic angewendet werden.

- **Prepared Statement:** Normalerweise werden SQL-Anweisungen komplett mit Parametern an die Datenbank geschickt, die dort geparst, kompiliert und ausgeführt wird. Ein Prepared Statement enthält noch keine Parameter, sondern den Platzhalter `"?"` für jeden Parameter. Das Statement wird einmal bei der Erzeugung kompiliert und braucht bei einem Aufruf nur die Parameter entgegenzunehmen. Danach kann die SQL-Anweisung gleich ausgeführt werden. Listing 4.32 zeigt ein Beispiel zum Hinzufügen eines Elements in die Relation *Domain* der Feature-Logic. Zunächst

[14] wrap ⟨engl., „umhüllen, verpacken“⟩

Anzahl Objekte (Linien)	10.000	100.000
Arbeitsspeicher		
CADEMIA	47,3 MB	47,3 MB
Hinzufügen ohne Visualisierung	+ 7,0 MB	+ 157,9 MB
Visualisierung	+ 10,3 MB	+ 100,0 MB
Nach dem Speichern (inkl. FLFS)	+ 20,1 MB	+ 186,6 MB
Dateisystem		
ZIP-Archiv	4.384 kB	44.023 kB
Feature-Logic	4.637 kB	47.147 kB
Speicherzeit	7,3 s	71,1 s
Ladezeit (ohne Visualisierung)	1,4 s	12,6 s
DXF	1.531 kB	15.616 kB
Feature-Logic in ZIP-Archiv	772 kB	7.702 kB

Tabelle 4.19: Vergleich des Speicherbedarfs für 10.000 und 100.000 Objekte

wird im Konstruktor das Prepared Statement definiert (Zeile 1-3) und später in der Methode *_storeElement()* mit dem Parameter *element* aufgerufen (Zeile 5-6).

```
1 PreparedStatement pstmtAddElement =
2     m_con.prepareStatement(
3         "INSERT INTO Domain VALUES (?)");
4
5 pstmtAddElement.setString(1, element);
6 pstmtAddElement.execute();
```

Listing 4.32: Beispiel eines Prepared Statements für die Feature-Logic

- **Index:** Ohne einen Index muss die Datenbank zum Finden eines Spaltenwerts alle Zeilen einer Tabelle nacheinander durchsuchen, was bei großen Datenmengen selbst mit moderner Hardware enorm zeitaufwändig ist. Daher kann durch eine separate Indexstruktur der Zugriff auf eine Spalte beschleunigt werden. Ein anschauliches Beispiel ist der Index am Ende eines Buches, der zum schnellen Auffinden bestimmter Begriffe innerhalb des Buches dient. Durch Festlegen eines Primärschlüssels wird automatisch ein Index für die entsprechende Spalte erzeugt. Zusätzlich werden in der Feature-Logic auf dem Server zwei weitere Indizes angelegt (s. Listing 4.33). Zeile 1 indiziert in der Relation *Atom* zusätzlich zur Spalte *element* die Spalte *value*, um zu einem atomaren Wert schnell das primitive Element zu finden. Zeile 2 erstellt in der Relation *Relslot* einen kombinierten Index für die Spalten *feature* und *value*, um die Operation *Selektion* zu beschleunigen.

```
1 CREATE UNIQUE INDEX ATOM_IDX ON Atom (value)
2 CREATE INDEX Relslot_Ftr_Val_IDX ON Relslot (feature,value)
```

Listing 4.33: Definition von Datenbankindizes für die Feature-Logic in SQL

- **Batch-Modus:** Im Batch-Modus[15] (Stapelverarbeitung) werden erst mehrere SQL-Anweisungen gesammelt und später zusammen ausgeführt. Dies verringert die Kommunikation zwischen Programm und Datenbank und führt zu einer schnelleren Ausführung der SQL-Anweisen. Jedoch können im Batch-Modus keine Ergebnisse zurückgeliefert werden. Listing 4.34 zeigt die überarbeitete Feature-Logic, damit sie Elemente in die Relation *Domain* im Batch-Modus entgegennehmen kann.

  Von außen kann die Feature-Logic mit der Methode *setBatchMode()* in den Batch-Modus gesetzt werden (Zeile 1-3). Beim Hinzufügen eines Elements wird die interne Methode *_storeElement()* aufgerufen, die zuerst prüft, ob das Element schon vorhanden ist, und falls nicht, die SQL-Anweisung dem Stapel übergibt (Zeile 7-22). Die Prüfung muss durchgeführt werden, da sonst die Eindeutigkeit des Primärschlüssels verletzt und eine SQL-Exception geworfen wird. Ursprünglich wurde die Existenz des Elements mit einer SQL-Anweisung ermittelt, was aber im Batch-Modus nicht möglich ist. Daher wird ein Attribut *m_domain* vom Typ `Set<String>` eingeführt, das eine transiente Menge mit existierenden Elementen vorhält. Abschließend kann der SQL-Stapel durch Aufrufen der Methode *executeBatch()* ausgeführt werden (Zeile 29-35).

```
3  public void setBatchMode(boolean batchMode) {
4      m_batchMode = batchMode;
5  }
6
7  private boolean _storeElement(String element) throws
8          SQLException {
9      if (! isElement(element)) {
10         pstmtAddElement.setString(1, element);
11         if (m_batchMode) {
12             pstmtAddElement.addBatch();
13             countPstmtAddElement++;
14         }
15         else
16             pstmtAddElement.executeUpdate();
17
18         m_domain.add(element);
19         return true;
20     }
21     return false;
22 }
23
```

[15] batch ⟨engl., „Stapel“⟩

```
24 public boolean isElement(String element) throws
25         SQLException {
26     return m_domain.contains(element);
27 }
28
29 // Auszug der Methode executeBatch();
30 m_con.setAutoCommit(false);
31 if (countPstmtAddElement > 0) {
32     pstmtAddElement.executeBatch();
33 }
34 // ... Feature und Atome hinzufuegen
35 m_con.commit();
36 m_con.setAutoCommit(true);
```

Listing 4.34: Feature-Logic : Hinzufügen von Elementen im Batch-Modus

Analog dazu wird der Batch-Modus auch auf die Relationen *Feature* und *Atom* angewendet und die Attribute

- `Set<String> m_feature` und
- `ReverseAccessMap<String,String> m_atom`

hinzugefügt. In der Methode *executeBatch()* werden mehrere Prepared Statements, die sich im Batch-Modus befinden, aufgerufen. Damit alle zusammen eine Transaktion bilden, wodurch sich wiederum Ausführungszeit einsparen lässt, wird zu Beginn der Methode das *AutoCommit* ausgeschaltet (Zeile 30) und nach dem Ausführen aller Statements wieder eingeschaltet (Zeile 35-36).

Leistungsmessung: Für die Zeitmessung stand ein dedizierter[16] Server zur Verfügung, der mit einem Intel Pentium 4 2,6 GHz, 2 GB RAM und einem RAID-1-System[17] aus zwei 74 GB Festplatten (WDC WD740GD-75FLA1, 10.000 U·min^{-1}, 8,8 ms mittlere Zugriffszeit) ausgerüstet ist. Die Übertragungsgeschwindigkeit der Netzwerkverbindung betrug zwischen Client und Server praktisch ca. 7.000 kB/s. Auf dem Server ist die kostenfreie Oracle Datenbank 10g Express-Edition (XE) installiert, die auf eine CPU, 1 GB RAM und 4 GB Datenbankgröße beschränkt ist.

Zur Leistungsbestimmung wurden für den Commit einer Zeichnungsdatei mit 1.000 Linien die Zeiten gemessen. Tabelle 4.20 enthält die Ergebnisse für drei unterschiedliche Optimierungsstufen der Feature-Logic. Im ersten Fall waren die Relationen mit einem Index des Primärschlüssels versehen und es wurde eine Zeit von 82,7 s benötigt. Danach wurden die zwei oben beschrieben Indizes für die Relationen *Atom* und *Relslot* erzeugt sowie transiente Datenstrukturen als Cache[18] hinzugefügt, was in einer Senkung der Zeit

[16] dedicare ⟨lat., „jemandem etwas zusprechen, weihen, widmen“⟩ → Ein dedizierter Server ist nur für eine spezielle Aufgabe vorgesehen.

[17] RAID = Redundant Array of Independent Disks

[18] cacher ⟨franz., „verbergen“⟩ → Ein Cache ist ein schneller Pufferspeicher, der einmal benutzte Daten zwischenspeichert, um sie bei einem nachfolgenden Zugriff schnell zurückgeben zu können.

auf 36,9 s resultierte. Im letzten Fall wurde der Batch-Modus integriert und die Methode *store()* in der Klasse *FLDataImpl* angepasst (s. Abschnitt 4.2.1 auf Seite 99). Hier war im Besonderen darauf zu achten, welche Zeilen in den Relationen neu anzulegen oder wie im Fall der Relation *Relslot* durch Änderung eines Feature-Wertes anzupassen sind. Durch diese Optimierungen konnte die Commit-Zeit auf 8,6 s gesenkt werden.

Optimierung	Zeit des Commits
Index auf Primärschlüssel	82,7 s
+ PStmt[a] + Indizes + `m_domain, m_feature, m_atom`	36,9 s
+ Batch-Modus	8,6 s

[a] Prepared Statement

Tabelle 4.20: Commit-Zeiten für 1000 Linien

Interessant in diesem Zusammenhang ist die Aufteilung der gesamten Commit-Zeit auf die einzelnen Schritte beim Commit-Vorgang, die in Tabelle 4.21 für eine verschiedene Anzahl von Objekten aufgeführt sind. Neben der Datenbank wurde testweise eine Feature-Logic für das Dateisystem auf dem Server installiert. Zwar werden kürzere Zeiten erreicht, aber unter Verzicht auf das bewährte ACID-Konzept und auf andere Funktionalitäten, die Datenbanken bieten.

Schritt	Oracle			FLFS	
Anzahl Objekte:	1.000	10.000	50.000	10.000	50.000
Commit zum VCS Subversion	3,30 s	4,31 s	9,59 s	3,50 s	9,86 s
Sammeln der lokalen FL-Daten	0,56 s	2,67 s	13,0 s	2,06 s	12,0 s
Übertragen der Daten in die FL	7,14 s	15,1 s	55,4 s	2,47 s	22,7 s
Setzen eines Tags in Subversion	0,13 s	0,13 s	0,16 s	0,13 s	0,14 s
Aktualisieren der lokalen FL	0,94 s	0,89 s	4,33 s	0,66 s	3,45 s
Gesamtzeit	8,61 s	24,0 s	83,2 s	9,56 s	48,9 s
Größe des FLData-Objekts	0,4 MB	4,4 MB	22 MB	4,4 MB	22 MB

Tabelle 4.21: Commit-Zeiten im Detail

Ergebnisse und Diskussion: Die Verwendung einer Datenbank als Datencontainer für die Feature-Logic ist unter Berücksichtigung der Leistung kritisch zu bewerten. Ohne Optimierung wird bei 1000 Objekten – eine Anzahl, die im Bauwesen in Datenmodellen eher gering ist – eine Commit-Zeit von 82,7 s benötigt. Durch Ausschöpfen aller Optimierungsmöglichkeiten kann diese zwar auf ca. ein Zehntel gesenkt werden, jedoch steigt der Speicherbedarf im Arbeitsspeicher durch die parallel verwendeten transienten Datenstrukturen. Unter praktischen Gesichtspunkten dürfte bei großen Projekten nur eine

Datenbank mit Prepared Statements und Indizes eingesetzt werden, wofür jedoch lange Ausführungszeiten in Kauf zu nehmen sind. Der Einsatz der Feature-Logic für das Dateisystem auf dem Server stellt keine praxisgerechte Alternative dar.

Problematisch an der Feature-Logic-Umsetzung ist der ständige Wechsel zwischen abfragenden und eintragenden SQL-Anweisungen, der für eine performante Lösung ein Caching erfordert. Die Zeiten für die Speicherung in das Versionsverwaltungssystem erreichen hingegen akzeptable Werte.

Bei der Übertragung sehr großer Modelle zum oder vom Server ist die Bandbreite der vorhandenen Netzwerkverbindung von Bedeutung. Ausgehend von einem Datenmodell mit 50.000 Objekten beträgt die Größe des ZIP-Archivs und des FLData-Objekts zusammen ca. 44 MB. Bei einer heutzutage üblichen asymmetrischen DSL-6000/640-Verbindung[19] mit praktisch erreichbaren 720 kB/s im Download und 75 kB/s im Upload würde der Commit ungefähr 10 Minuten und ein Update ungefähr 60 Sekunden dauern. Eine Komprimierung der Daten ist im Falle des ZIP-Archivs aus den in Abschnitt 4.3.4 auf Seite 113 beschriebenen Gründen nicht ratsam, da für nachfolgende Commits nur die Änderungen am ZIP-Archiv zu übertragen sind, wodurch eine wesentlich geringere Datenmenge anfällt. Eine Komprimierung des FLData-Objekts ist nicht sinnvoll, weil es eine binäre Struktur besitzt und sich deshalb keine signifikante Einsparung erreichen lässt. Überlegenswert ist eine Reduzierung der Größe, indem vor dem Commit nicht die Feature-Logic-Struktur modelliert wird, sondern nur die wesentlichen Daten wie POVIDs, PVIDs, Dokumentmetadaten und Bindungsdaten zum Server übertragen und erst dort für die Feature-Logic aufbereitet sowie gespeichert werden. Weiterhin ist zu hinterfragen, ob für jede Objektversion das Änderungsdatum und die Revisionsnummer modelliert werden muss. Beide Werte lassen sich aus den anderen Daten indirekt ermitteln. Für geänderte Dokumente wäre es ausreichend, neben den Bindungsdaten nur die POVIDs neuer, geänderter und gelöschter Objekte zu übertragen. Unabhängig von der Verringerung der Datengröße ist es für Unternehmen empfehlenswert, einen Netzzugang mit gleicher Upload- und Downloadbandbreite zu unterhalten, was zum Beispiel bei symmetrischem DSL (SDSL) der Fall ist.

Anfragen an die zentrale Feature-Logic

Einzelanfragen: Über die Methode *askQuery(String query)* der Schnittstelle *FLClient* lassen sich Feature-Terme an die Feature-Logic auf dem Server schicken (s. Abschnitt 4.2.3 auf Seite 104). Dabei muss die Abfrage als String-Objekt über RMI an den Server geschickt, die Ergebnismenge berechnet und ebenfalls über RMI als Set-Objekt zurückgeschickt werden. Gleichfalls ist es zum Beispiel möglich, den Wert eines Features direkt ohne einen Feature-Term über die Methode in Zeile 1 des Listings 4.35 auszulesen. In einem schnellen Netzwerk benötigt jeder RMI-Aufruf ca. 12 ms. Wenn wie im Dialog *ProjektExplorer*, der im Abschnitt 4.8.1 auf Seite 152 beschrieben ist, für den Aufbau des Projektbaumes viele Abfragen zu stellen sind, verschlechtert sich die Leistung erheblich.

[19] DSL = Digital Subscriber Line. ⟨engl., „Digitaler Teilnehmeranschluss“⟩

Tausend Abfragen benötigen allein durch die RMI-Aufrufe schon mindestens 12 Sekunden.

Anfragenminimierung: Um die Zahl der Anfragen zu reduzieren, wurden im *FLClient* Methoden eingeführt, die die Feature-Werte-Paare von mehreren Elementen oder die Werte von mehreren Atomen gleichzeitig zurückliefern (Zeile 3-5). Die Methode *getAllAtoms()* liefert die atomaren Werte aller primitiven Elemente zurück. Dadurch, dass zu jedem Objekt nur die Version und das Datum gespeichert wird, besitzt die Relation *Atom* wegen der Mehrfachverwendung der Atome nur eine geringe Größe.

```
1 public String getFeatureValue(String element, String feature);
2
3 public Map<String,Map<String,String>> getFeatureValues(
4     Set<String> elements);
5 public Map<String,Object> getAtomValues(Set<String> elements);
6 public Map<String,Object> getAllAtoms();
```

Listing 4.35: Methoden der Schnittstelle *FLClient* zur Anfragenminimierung

5 Pilotimplementierung

> Man versteht etwas nicht wirklich, wenn man nicht versucht, es zu implementieren.
>
> *(Donald E. Knuth, 2002)*

5.1 Notwendige Implementierungen für spezielle Anwendungen

Methoden: Der Workspace stellt den Zugriff für alle wesentlichen verteilten Operationen über die Schnittstelle *Workspace* sowie die Implementierung über die davon abgeleitete Klasse *WorkspaceAdapter* zur Verfügung. Eine Anwendung, die zur verteilten Objektversionierung befähigt werden soll, muss lediglich um Methoden erweitert werden, die Operationen in der Sandbox ausführen (s. Tabelle 5.1).

Operation	Methode
(Erzeugen der Workspace-Instanz)	Konstruktor
Anwendungsname abfragen	*getApplicationName()*
Dokument erzeugen	*createNewDocument()*
Objekte des aktuellen Dokuments	*objectsInLoadedDocument()*
Dokument speichern	*store()*
Dokument laden	*load()*
Dokument importieren	*importDocument()*
Bindende Dokumente importieren	*importBindingDocuments()*
Importiertes Dokument entladen	*dropImportedDocument()*
Alle importierten Dokumente entladen	*dropAllImportedDocuments()*
Bindungen löschen	*deleteBindings()*

Tabelle 5.1: Durch Anwendungen zu implementierende Methoden des Workspace

Der Konstruktor legt die Superklasse für zu versionierende Objekte des Datenmodells fest und registriert die *IOObjectHandler* in der Klasse *IOUtils*, falls diese verwendet werden (s. Abschnitt 4.3.4 auf Seite 113). Zu Beginn ruft er den Konstruktor des *WorkspaceAdapters* mit *super()* auf und übergibt ihm den *VCSClient*, den *FLClient* und die *WorkspaceSettings*.

Benutzerschnittstelle: Falls die Anwendung über eine grafische Benutzerschnittstelle verfügt, ist es sinnvoll, die Operationen der Systemarchitektur für einen schnellen Zugriff in Menüs und Werkzeugleisten anzuordnen. Außerdem ist je nach Anwendungskontext der Entwurf grafischer Vergleichs- und Zusammenführungswerkzeuge empfehlenswert.

Offenheit: Für einen externen Software-Entwickler ist es vorteilhaft, wenn die zu erweiternde Anwendung im Quellcode verfügbar ist, da er auf das Objektmodell der Anwendung sowie auf das Datenmodell zugreifen kann. Bei geschlossenen Anwendungen ist der Entwickler immer vom Hersteller abhängig. Das bedeutet, dass nur vorhandene Schnittstellen und Dokumentationen genutzt werden können. Häufig kommunizieren Softwarehersteller getätigte Änderungen an neueren Programmversionen nicht oder verzögert an Dritte.

(Fahrig, 2007) beschreibt dies am Beispiel des Produkts Architectural Desktop (ADT) des Herstellers Autodesk, das auf AutoCAD basiert und zur bauteilorientierten Planerstellung dient. Die in C++ programmierte Schnittstelle ObjectARX bietet einen Zugriff auf die Grundfunktionen von AutoCAD, aber nicht auf das ADT-Modell. Dazu ist das Object Modeling Framework (OMF) vorgesehen, welches eine Erweiterung von ObjectARX darstellt. Fahrig stuft die Einarbeitungszeit in OMF als hoch ein und kritisiert die veraltete und unvollständige Dokumentation. Weiterhin weist der ADT viele Inkompatibilitäten auf.

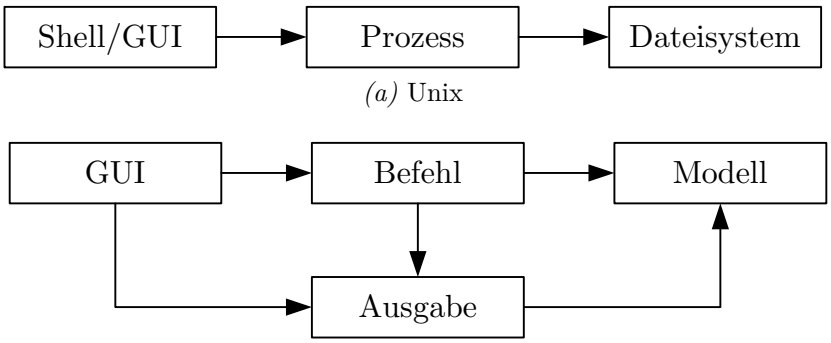
5.2 Open-Source-Ingenieurplattform CADEMIA

Allgemein: CADEMIA ist eine offene und kostenfreie Open-Source-Ingenieurplattform, die ursprünglich für die Lehre an der Bauhaus-Universität Weimar entwickelt wurde[1]. Seit der Version 1.0 ist sie öffentlich verfügbar[2] und unter der GNU General Public License (GPL)[3] lizenziert. CADEMIA ist in der objektorientierten Sprache Java programmiert und dient als Grundlage für die Entwicklung geometrisch orientierter Fachanwendungen. Vorteilhaft ist die leichte Erweiterbarkeit über definierte Schnittstellen und die durch Java gegebene Plattformunabhängigkeit.

Systemarchitektur: Die Systemarchitektur von CADEMIA lehnt sich an den Entwurf eines Betriebssystems an, das wie im Fall von UNIX in die Teilsysteme Shell/GUI, Prozess und Dateisystem gegliedert ist (Bourne, 1988). Shell und GUI repräsentieren die Ein-/Ausgabe und greifen auf laufende Prozesse zu, die unter anderem im Dateisystem operieren (s. Abbildung 5.1a).

In der GUI von CADEMIA werden Nutzeraktionen in Befehle umgewandelt, die außerdem über eine Kommandozeile direkt eingegeben werden können. Die Befehle verändern das Modell, dessen Zustand dem Planer über den View (Ausgabe) grafisch angezeigt wird (s. Abbildung 5.1b).

[1] s. (Firmenich u. a., 2008), (Firmenich, 2006)
[2] http://www.cademia.org/
[3] http://www.gnu.org/copyleft/gpl.html



(a) Unix

(b) CADEMIA

Abbildung 5.1: Systemarchitektur

Die wesentlichen Bestandteile von CADEMIA sind im UML-Diagramm der Abbildung 5.2 dargestellt. Die zentrale Klasse ist der *Kernel*, der Zugriff auf den Befehlsmanager (Klasse *CmdMgr*), die Schnittstelle *UserInterface* und die Klasse *Database* als zentralen Punkt des Modells hat. Die Database verwaltet den Namensraum, das Koordinatensystem, den Attributmanager, die Layer und die Modellkomponenten. Das *LayerSet* enthält alle definierten Layer und die *ReverseAccessMap* weist jeder Komponente genau einen Layer zu.

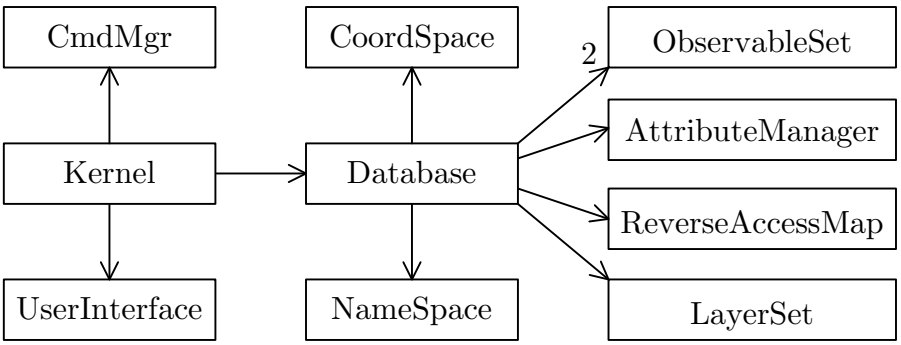


Abbildung 5.2: CADEMIA: Wesentliche Bestandteile

Komponenten: Die Komponenten des Modells müssen die Schnittstelle *Component* implementieren, die sich wiederum von den Schnittstellen *Serializable, Cloneable* und *ObservableObject* ableitet (s. Abbildung 5.3). Die ersten zwei sind Standardschnittstellen von Java, die die Serialisierung und das Klonen von Objekten erlauben. Ein ObservableObject meldet Zustandsänderungen und ein Klonen seiner selbst an registrierte Listener. Komponenten können sich auch von der Klasse *ComponentAdapter* ableiten, die schon wesentliche Methoden implementiert. Dadurch entsteht weniger Aufwand für den Entwickler. Komponenten werden in ein `ObservableSet<Component>` mit dem Namen *m_componentSet* in der Database eingetragen. Ein ObservableSet benachrichtigt angemeldete Listener, wenn Elemente der Menge hinzugefügt, gelöscht oder geklont wurden sowie wenn sie sich geändert haben. Ein zweites ObservableSet mit dem Namen *m_selectSet* enthält alle selektierten Komponenten des Modells.

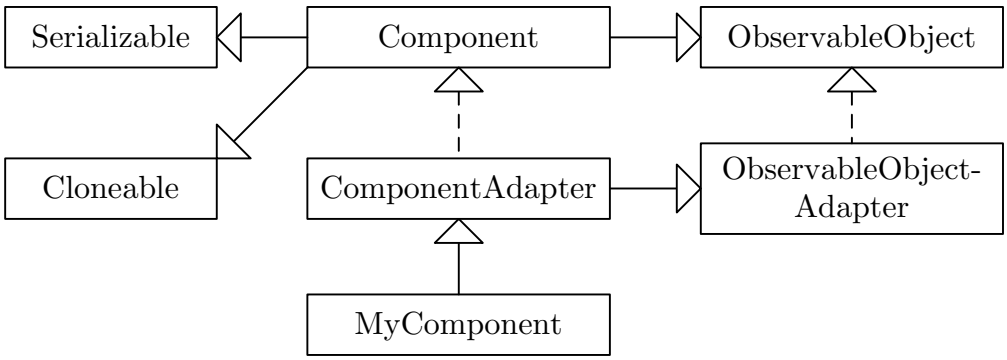


Abbildung 5.3: CADEMIA: Schnittstelle *Component*

Befehle: Objekte der Schnittstelle *Cmd* stehen stellvertretend für Befehle (s. Abbildung 5.4). Die Schnittstelle schreibt Methoden vor, die die Undo-/Redo-Verwaltung ermöglichen, um Befehle zurücknehmen und danach wieder ausführen zu können. Damit die Befehlshistorie erhalten bleibt, verwaltet die Klasse *CmdMgr* des Kernels alle Befehlsobjekte in einer Liste sowie einen Cursor[4], der die aktuelle Position anzeigt.

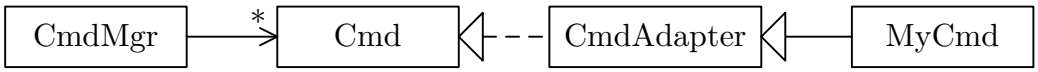


Abbildung 5.4: CADEMIA: Schnittstelle *Cmd*

Dateischnittstellen: CADEMIA speichert das Datenmodell durch Serialisierung der Database in eine binäre Datei mit der Dateierweiterung *.cademia*, die nur von CADEMIA wieder deserialisiert werden kann. Darüber hinaus unterstützt CADEMIA den Im- und Export von DXF-Dateien (s. Abschnitt 2.2.2 auf Seite 23).

Benutzerschnittstelle: *UserInterface* ist die Schnittstelle für alle Benutzerschnittstellen in CADEMIA, von der sich nur die Klasse *GraphicalUserInterface* ableitet. Abbildung 5.5 zeigt die grafische Benutzerschnittstelle von CADEMIA, die in Swing programmiert ist (s. Abschnitt 2.6.3 auf Seite 70). Oberhalb der Zeichenfläche befindet sich eine Menüleiste sowie ein Bereich für Werkzeugleisten und unterhalb sind ein Ausgabefenster, eine Befehlszeile und ein variables Befehlsmenü angeordnet.

CADEMIA unterstützt beliebige Eingabegeräte über die Schnittstelle *InputDevice*. Dazu muss das *UserInterface* einen *InputDevice.Listener* am *InputDevice* registrieren, um über Aktionen informiert zu werden. Die Klasse *AddMouseStrokeInputDevice* fügt die Interpretation von Mausgesten hinzu, die eine Kombination von Mausbewegungen und -klicks darstellen und in Befehle umgewandelt werden. Befehle können dadurch sehr schnell eingegeben werden, ohne die Hand von der Maus nehmen zu müssen. Die Tastatur als zweites physisches Eingabegerät wird durch Swing-Komponenten direkt unterstützt.

View: Der geometrische View setzt auf das Java 2D API auf, das ein Bestandteil der JRE ist (Hardy, 2000). Die Schnittstelle *Shape* ist der Ausgangspunkt für alle geometrischen

[4]cursor ⟨lat., „Läufer“⟩

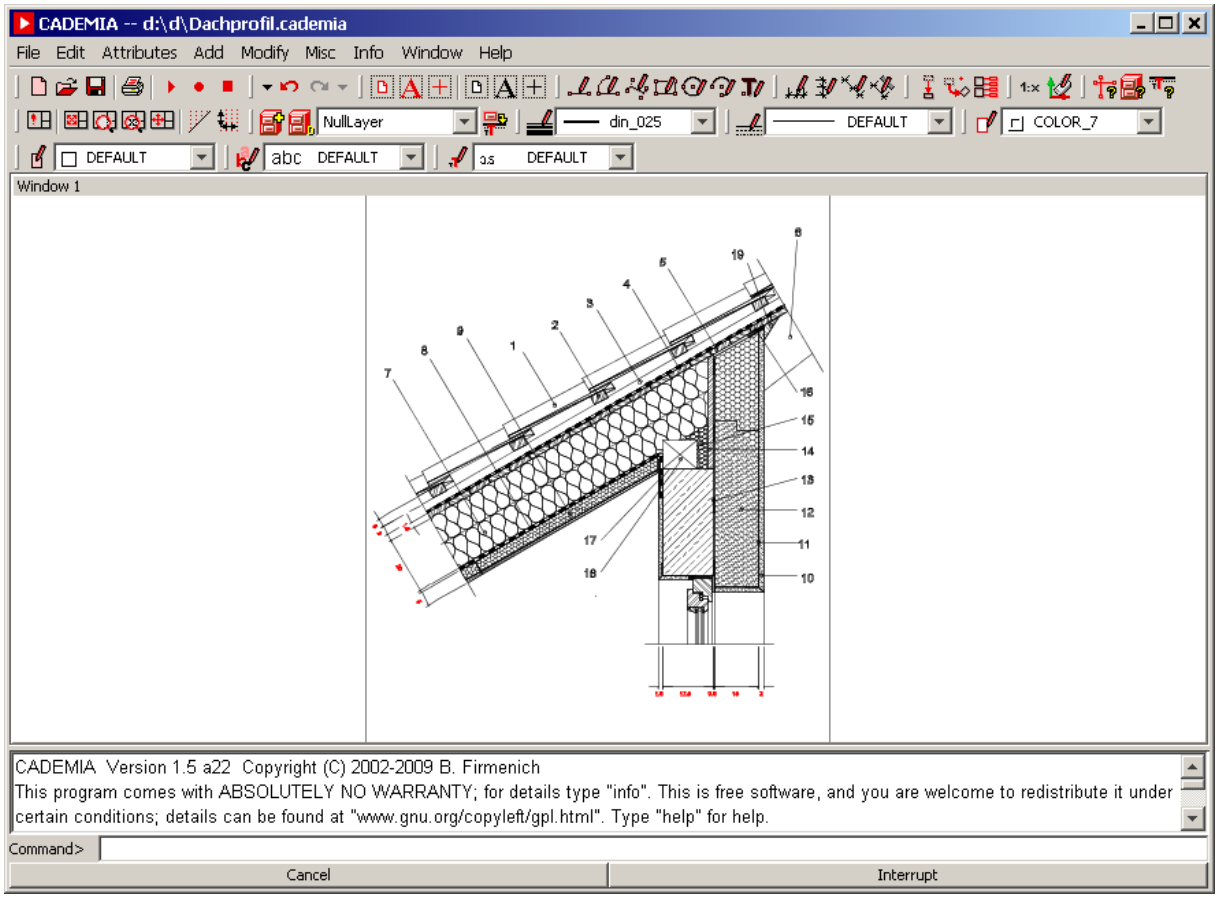


Abbildung 5.5: CADEMIA: Screenshot

Objekte, von den einige Standardformen bereits implementiert sind: Bogen, Ellipse, Linie, quadratische und kubische Kurve, Rechteck, Pfad u. a. Die Verwaltung von formatiertem Text erfolgt in der Klasse *AttributedString*. Die Zeichenfläche wird durch die Klasse *Graphics2D* bereitgestellt, die ein geräteunabhängiges Koordinatensystem besitzt und für den Render-Prozess zuständig ist. Rendern bedeutet das Umwandeln einer Vektorgrafik in eine Rastergrafik, weshalb dieser Prozess im Deutschen auch als Rasterung bezeichnet wird. Der Graphics2D führt Renderoperationen für Shapes, Texte und Bilder aus. Das Ergebnis kann dann an verschiedene Ausgabegeräte, wie Bildschirm oder Drucker, weitergeleitet werden.

5.3 Erweiterung von CADEMIA für die Objektversionierung

5.3.1 Workspace

Klassen: Da CADEMIA eine Plattform für darauf aufbauende Ingenieuranwendungen ist, erfolgt die Implementierung des Workspace in zwei getrennten Klassen (Abbildung 5.6). Die Klasse *WorkspaceCademiaAdapter*, die vom WorkspaceAdapter abgeleitet ist, enthält die Methoden, die für alle CADEMIA-basierten Anwendungen Funktionalitäten bereitstellen, während die Klasse *WorkspaceCademia* speziell für CADEMIA ausgelegt ist. Das betrifft im Wesentlichen das Erzeugen, Laden und Speichern von Dokumenten. Die Methode *getApplicationName()* liefert den für die persistenten Identifikatoren verwendeten Anwendungsnamen `cademia` zurück.

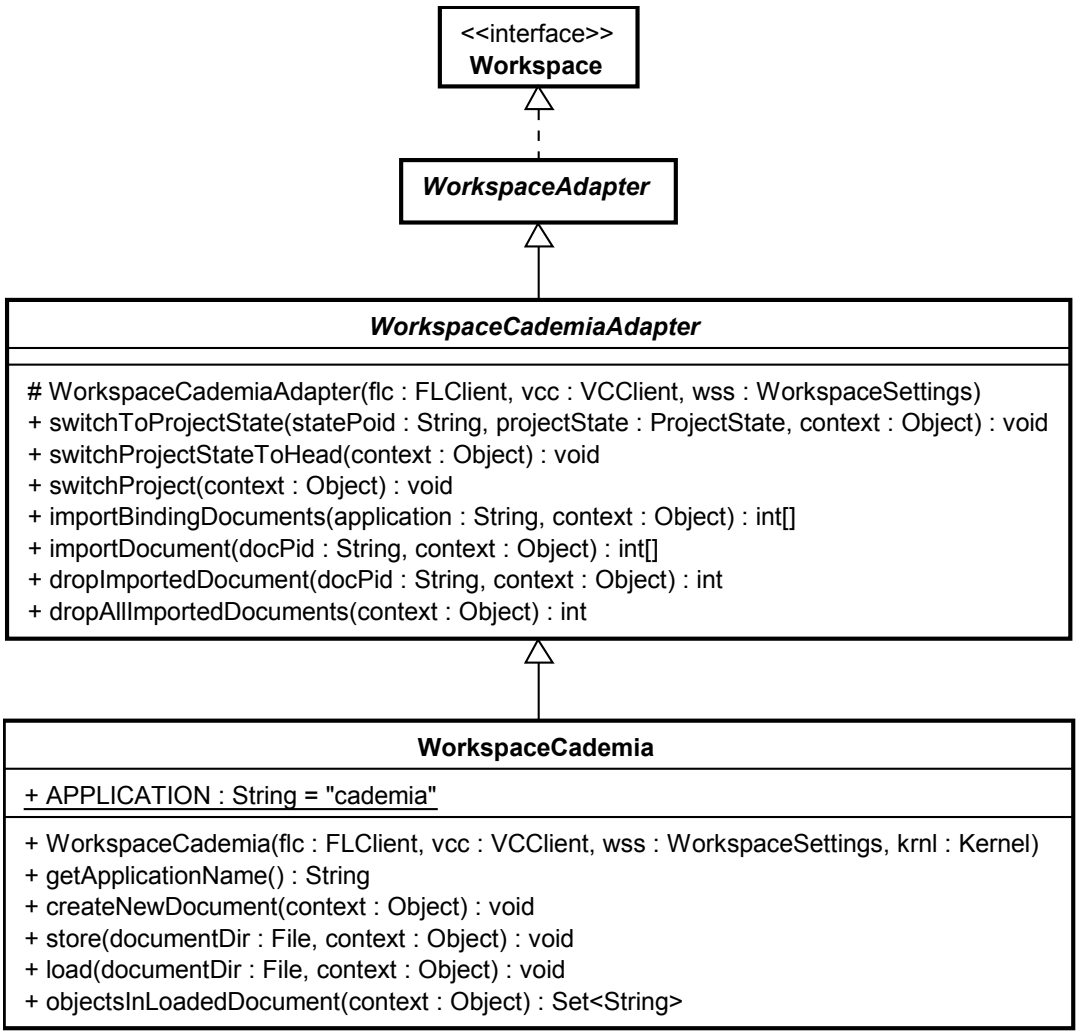


Abbildung 5.6: UML-Klassendiagramm: Workspace für CADEMIA-Anwendungen

5.3.2 Komponenten

Serialisierung: CADEMIA besitzt Standardkomponenten zur Darstellung zweidimensionaler geometrischer Objekte. Dafür ist jeweils ein *IOObjectHandler* zum Speichern, Laden und Importieren zu schreiben. Zusätzlich müssen Zeichnungsattribute, wie Linienstärke, Linienfarbe usw., Layer und das Koordinatensystem gespeichert werden, die in gesonderten Klassen modelliert sind. Tabelle 5.2 zeigt die zu schreibenden IOObjectHandler-Klassen.

Komponente/Attribut	IOObjectHandler
Attribut Füllfarbe	`IOAttrFillPaintsHandler`
Attribut Linienfarbe	`IOAttrDrawPaintsHandler`
Attribut Linienmuster	`IOAttrLinePatternsHandler`
Attribut Linienstärke	`IOAttrLineWidthsHandler`
Attribut Schriftarten	`IOAttrFontsHandler`
Attribut Schriftgrößen	`IOAttrTextSizesHandler`
Koordinatensystem	`IOCoordSpaceHandler`
Layer	`IOLayerDHandler`
Bemaßung	`IODimSingleHandler`
Bogen	`IOArc2DHandler`
Kreis	`IOCircle2DHandler`
Linie	`IOLine2DHandler`
Polylinie	`IOPath2DHandler`
Schriftfeld	`IOTitleBlockHandler`
Text	`IOText2DHandler`

Tabelle 5.2: CADEMIA: Zu implementierende IOObjectHandler-Klassen

Bestimmte Blöcke wiederholen sich in allen IOObjectHandler-Klassen, welche deshalb in die abstrakte Klasse *IOCademiaObjectHandler* ausgelagert werden. Von dieser Klasse leiten sich alle speziellen IOObjectHandler ab. Abbildung C.2 auf Seite 240 zeigt dies am Beispiel des *IOLine2DHandlers*. In die zu speichernden Textdateien werden Schlüssel-Wert-Paare geschrieben, die transient durch ein Objekt der Klasse *PropertiesSortedNoTime* repräsentiert werden. Die speziellen IOObjectHandler müssen drei Methoden bereitstellen:

- *getHandledClass()* gibt die Klasse des zu behandelnden Objekts an.
- *createPropertyObject()* erzeugt für das Speichern das Properties-Objekt mit den Schlüssel-Wert-Paaren aus dem CADEMIA-Modell.
- *createObject()* erzeugt das CADEMIA-Objekt aus den Schlüssel-Wert-Paaren zum Laden in das Modell.

Die Klasse *IOCademiaObjectHandler* übernimmt das Schreiben und Lesen der ZIP-Einträge, das Auslesen und Setzen von Attributen sowie das Umwandeln von Welt- in Nutzerkoordinaten und umgekehrt.

5.3.3 Befehle

Standardoperationen: Zusätzlich zu den im Workspace implementierten Operationen zur verteilten Bearbeitung müssen in CADEMIA Klassen für Befehlsobjekte geschrieben werden, die unter anderem Nutzereingaben entgegennehmen und die Methoden des Workspace aufrufen. Die Implementierung der Klassen soll an dieser Stelle nicht im Detail erläutert werden, stattdessen zeigt folgende Übersicht die Befehlsnamen mit den zugeordneten Operationen.

Befehl	Operation
`checkoutProject`	Check-out (Projekt beitreten)
`switchProject`	Projekt in der Sandbox wechseln
`svnNewDrawing`	Neues Dokument erzeugen
`svnLoad`	Laden
`svnStore`	Speichern
`svnStoreAs`	Speichern unter
`svnSandboxInfo`	Sandboxstatus
`svnDocumentHistory`	Dokumenthistorie
`svnInfo`	Dokumentinformationen (lokal)
`projectExplorer`	Projektüberblick (Projektexplorer)
`defineProjectState`	Freigeben (Freigabestand definieren)
`updateProjectState`	Zu einem Freigabestand wechseln
`resetProjectState`	Rückkehr zum aktuellen Planungsstand
`svnCommit`	Übertragen (Commit)
`svnUpdate`	Aktualisieren (Update)
`svnUpdateOverride`	Aktualisieren mit Überschreiben
`svnUpdateNewDrawings`	Holen (Update New Documents)
`svnCleanUp`	Sandbox bereinigen (von Fehlern und Sperren)
`svnServerSettings`	Einstellungen für den Zugriff auf die Server
`export`	Exportieren in das native CADEMIA-Format

Tabelle 5.3: CADEMIA: Befehle für die verteilten Operationen

Die Standardbefehle von CADEMIA zum Erzeugen, Laden und Speichern von Dokumenten wurden deaktiviert und durch die neuen Befehle ersetzt, die direkt in der Sandbox arbeiten. Demzufolge wurden die zugehörigen Menüeinträge und Schaltflächen entfernt. Einzig der Speichern-Befehl wurde in *Export* umbenannt. Das Importieren und Hinzufügen von externen Geometriemodellen im DXF-Format ist über den Befehl *ImportDXF* möglich.

Bindungsoperationen: Für die Behandlung von Objektabhängigkeiten sind die Befehle aus Tabelle 5.4 zuständig. Die Befehle sind momentan so ausgelegt, dass Objekte des aktuellen Dokuments an geometrische Komponenten importierter Dokumente gebunden werden. Bei Verwendung anderer Fachmodelle in CADEMIA lassen sich die Befehle entsprechend anpassen.

Befehl	Operation
`bindToGeometry`	Objekt an Geometriekomponente(n) binden.
`checkBindingsLocally`	Bindungen im aktuellen Dokument prüfen.
`markBindingsAsValid`	Alle Bindungen im aktuellen Dokument als gültig markieren.
`importGeometry`	Geometrie aus Dokument importieren.
`importBindingGeometry`	Alle bindenden Dokumente importieren.
`refreshImportedGeometry`	Importierte Dokumente aktualisieren.
`dropImportedGeometry`	Importierte Dokumente entladen.
`updateBindingDocumentsTo-LatestVersion`	Alle bindenden Dokumente auf den aktuellen Stand aktualisieren.
`updateBindingDocumentsTo-OriginalVersion`	Alle bindenden Dokumente auf den Stand zur Zeit der Bindungsdefinition aktualisieren.
`dumpBindings`	Interne Informationen zu den Bindungen ausgeben.
`deleteBindings`	Bindungen der markierten gebundenen Objekte löschen.

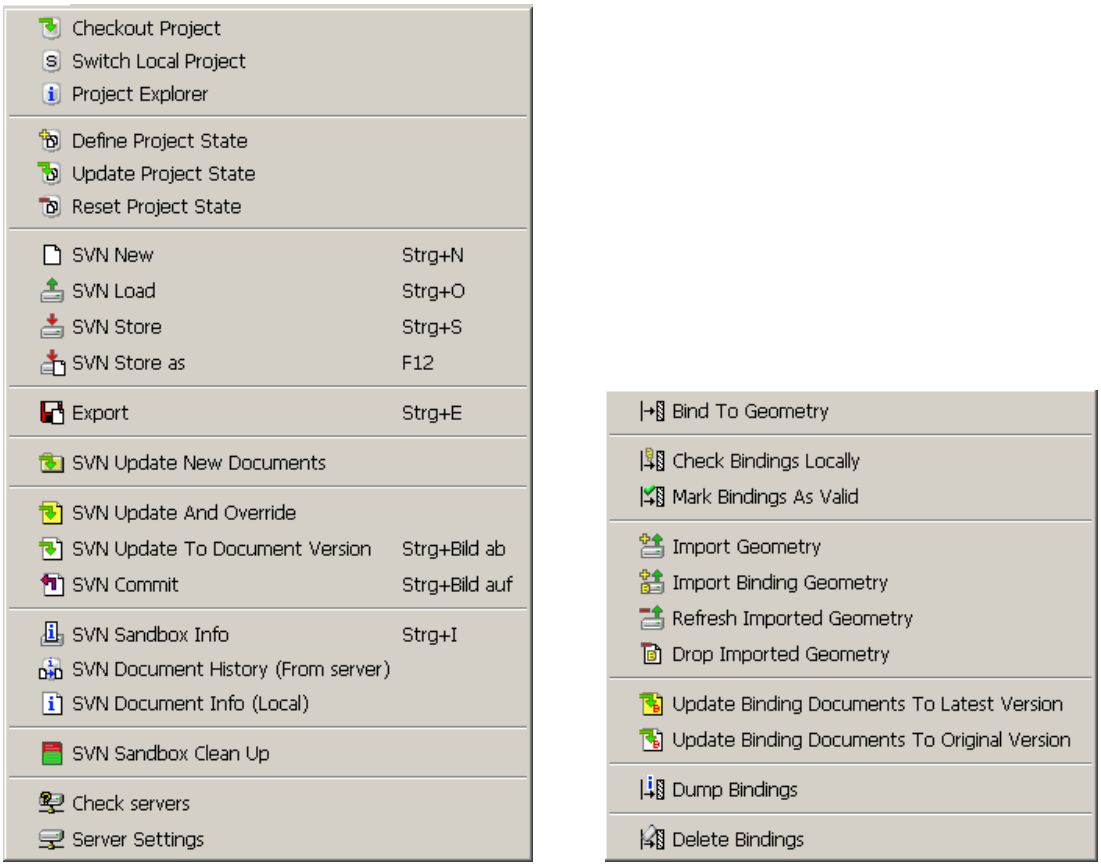
Tabelle 5.4: CADEMIA: Befehle für die Behandlung von Objektabhängigkeiten

5.3.4 Grafische Benutzerschnittstellen

Menüs: Die Menüleiste von CADEMIA wurde für einen schnellen Zugriff auf die Befehle um das Menü *Team* (Abbildung 5.7a) und für Anwendungen, die Objektabhängigkeiten unterstützen, um das Menü *Bindings* (Abbildung 5.7b) erweitert.

Dialoge: Die in Abschnitt 4.8.1 auf Seite 152 entwickelten Dialoge für die verteilten Operationen werden jeweils durch eine Klasse erweitert, die die Eingaben des Nutzers in Befehle für den Interpreter des CADEMIA-Kernels unwandelt.

Vergleichen und Zusammenführen: Auf Basis des ZIP-Archiv-Vergleichsdialogs wurde ein grafischer Dialog zum Vergleichen und Zusammenführen von zweidimensionalen geometrischen Objekten entworfen. Die Objektidentifizierung erfolgt über die Namen der Zip-Einträge, die die POIDs enthalten. Der Dialog enthält vier Zeichenflächen, wie sie in Abbildung 5.8 zu sehen sind. Sie unterstützen dynamisches Zoomen und Verschieben mit der Maus. Der Dialog besitzt das Wissen, wie aus den ZIP-Einträgen je nach Objektklasse ein *Shape* zur grafischen Anzeige erzeugt werden kann. Ausgehend von der Operation *Aktualisieren* sind die Zeichenflächen wie folgt belegt.



(a) Menü *Team* *(b)* Menü *Bindings*

Abbildung 5.7: CADEMIA: Menüs für verteilte Operationen

- **Project:** Diese Zeichenfläche enthält den Zustand der Dokumentversion auf dem Server.
- **Workspace:** Diese Zeichenfläche enthält den aktuellen Zustand des Dokuments in der Sandbox des Planers.
- **Select:** Diese Zeichenfläche dient zur Darstellung beider Zustände eines geänderten Objekts, von denen dann einer ausgewählt werden kann. Bereits übernommene Zeichnungsobjekte werden grau dargestellt.
- **Final:** Diese Zeichenfläche enthält das zusammengeführte Dokument, das beim Abschluss des Merge-Vorgangs die aktuelle Version in der Sandbox ersetzt.

Beispiel 5.1: Vergleichen und Zusammenführen von Zeichnungsversionen

Die Funktionsweise des Dialogs soll an einem kleinen Beispiel vorgestellt werden. Der Planer möchte seine Zeichnung auf den Server übertragen und muss einen Merge durchführen, da vor ihm schon ein anderer Planer eine neue Version des Dokuments committet hat. Die Dokumentversion auf dem Server enthält ein kleines Haus in der Seitenansicht mit einer Tür, einem Fenster und einem runden Dachfenster. Das aktuelle Dokument des Planers

unterscheidet sich von dem auf dem Server in drei Punkten, die farblich hervorgehoben werden.

- Das runde Dachfenster wurde durch ein quadratisches ersetzt. Das gelöschte wird links rot und das hinzugefügte rechts grün dargestellt.
- Das große Fenster wurde nach rechts verschoben und ist in beiden Zeichenflächen pink markiert.
- Die Tür wurde auf der anderen Seite eingehängt, was an dem nach rechts verschobenen Türknauf zu erkennen ist. Dieser wird als eigenständiges Zeichnungsobjekt ebenso pink hervorgehoben.

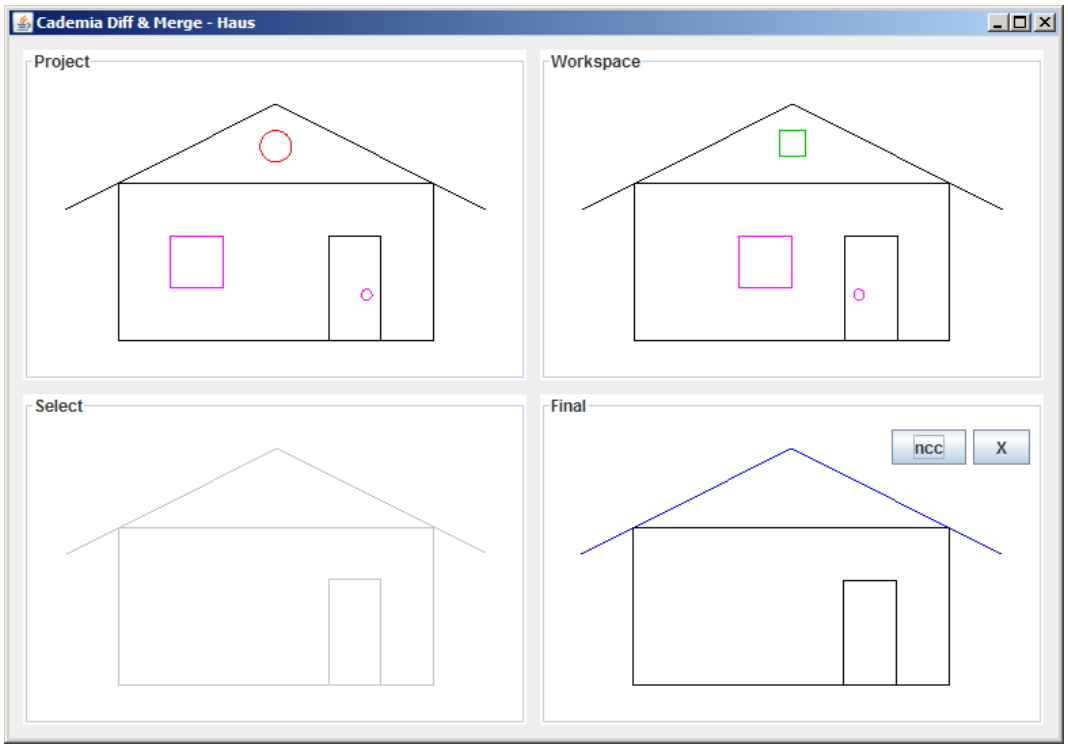
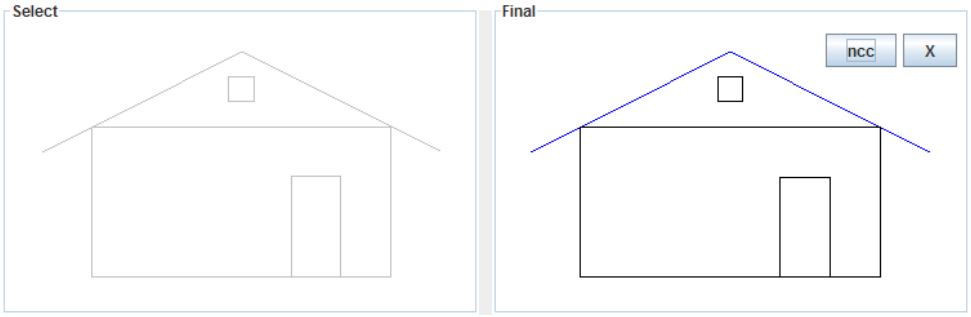


Abbildung 5.8: CADEMIA: Dialog Diff & Merge - Schritt 1

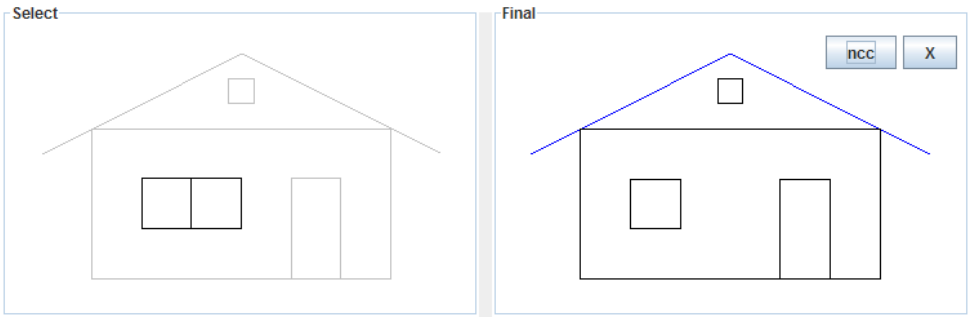
Die restlichen Zeichnungsobjekt blieben unverändert und sind in beiden Zeichenflächen schwarz dargestellt. Der Merge-Vorgang wird in den folgenden fünf Schritten durchgeführt.

- **Schritt 1:** Durch Anklicken der Schaltfläche *ncc* (non-conflicting components) werden alle unveränderten Komponenten in die Zeichenflächen *Select* und *Final* übernommen. Die farbliche Darstellung links unten ist in grau gehalten, da der Teilvorgang abgeschlossen ist. Die farbliche Darstellung rechts unten entspricht den realen Farben in der Zeichnung, wie z. B. das blaue Dach (s. Abbildung 5.8).
- **Schritt 2:** Als nächstes wird das quadratische Dachfenster übernommen, indem es rechts oben angeklickt wird und ebenso in den beiden unteren Zeichenflächen auftaucht (s. Abbildung 5.9a).

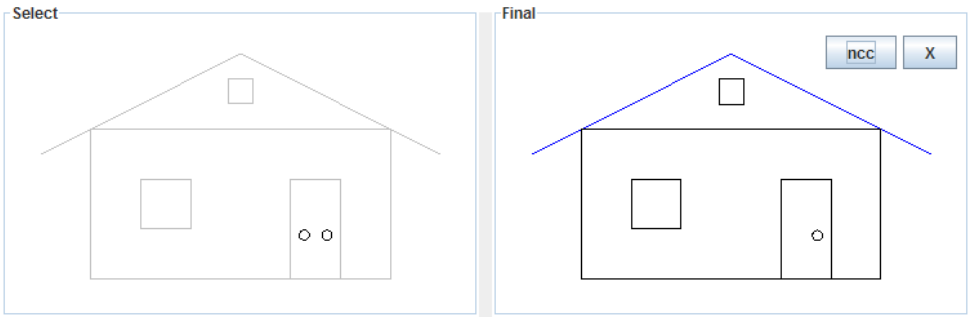
- **Schritt 3:** Danach soll das linke Fenster in die fertige Zeichnung aufgenommen werden. Dazu klickt der Planer es an, wodurch es zunächst zusammen mit dem rechten Fenster nur in der Select-Zeichenfläche in schwarzer Farbe erscheint. Hier wählt der Planer das linke aus und schließt diesen Vorgang ab (s. Abbildung 5.9b).
- **Schritt 4:** Auf gleiche Art und Weise wird der rechte Türknopf übernommen. Zu beachten ist, dass in der Select-Zeichenfläche das rechte Fenster verschwindet und das linke jetzt grau dargestellt ist. Jederzeit können einzelne Objekte aus der finalen Zeichnung entfernt werden, indem sie dort angeklickt werden (s. Abbildung 5.8).
- **Schritt 5:** Ein Klick auf die Schaltfläche *X* exportiert schließlich das Resultat des Zusammenführens in ein neues ZIP-Archiv aus den ZIP-Einträgen der ausgewählten Objekte und überschreibt das aktuelle ZIP-Archiv in der Sandbox. Der Dialog erlaubt in dieser Umsetzung kein Ändern der Objektzustände.



(a) Schritt 2



(b) Schritt 3



(c) Schritt 4

Abbildung 5.9: CADEMIA: Dialog Diff & Merge - Schritt 2 bis 4

5.4 Lastabtragsanwendung

5.4.1 Bestehende Verfahren

Einführung: Nach Abschluss des DFG-Projekts zum Entwurf einer CAD-Systemarchitektur für den verteilten Planungsprozess, dessen Ergebnisse und Erkenntnisse in (Beucke u. a., 2007) dokumentiert sind, folgte eine zweijährige Transferphase in Zusammenarbeit mit einem Praxispartner. Ziel des Projekts war, das Konzept der Objektversionierung auf ein praktisches Beispiel – den Lastabtrag – anzuwenden und zu verifizieren.

Lastabtrag: Ein Lastabtrag wird zu Beginn der Tragwerksplanung im Hochbau durchgeführt, um mit einer einfachen statischen Berechnung die Lasten, die auf die Bodenplatte wirken, zu bestimmen. Dadurch kann der Beginn gegenüber einer erst komplett durchgeführten Statik vorgezogen und so Kosten gesenkt werden. Die benutzten Methoden für einen Lastabtrag reichen von der Handrechnung über den Einsatz von Tabellenkalkulationen bis zur kompletten Modellierung in einer 3D-FEM-Anwendung. Nachteilig bei der 3D-FEM-Berechnung ist, dass der Lastfluss nicht nachvollzogen werden kann und zu Beginn des Projekts die Steifigkeiten der einzelnen Bauteile noch nicht bekannt sind.

Bestehendes Verfahren des Praxispartners: Beim Praxispartner ist ein Verfahren im Einsatz, um die Größe, Verteilung und Weiterleitung der vertikalen Lasten zu bestimmen (Hansen u. Binar, 2005). Die Modellierung erfolgt über statische Grundsysteme unter Vernachlässigung der Steifigkeiten und Durchlaufwirkungen. Die Lasten werden bei der Berechnung erst für jede Ebene horizontal zu den vertikalen Bauteilen weitergeleitet und dann nach unten bis oberhalb der Bodenplatte aufsummiert. Günstigerweise können die Ergebnisse des Verfahren für weitere Aufgaben in der Planung verwendet werden.

- Berechnung der mittleren Vertikallasten für Stützen und Wände für die nachfolgende Statik,
- Massenermittlung für die dynamische Berechnung und den Nachweis der Bauwerksaussteifung,
- Ungefähre Mengenermittlung für den Bedarf an Beton, Stahl und Schalung.

Die Durchführung der Berechnung kann manuell über Formblätter, in einer Tabellenkalkulation oder über ein beim Praxispartner entwickeltes Modul für AutoCAD erfolgen. Die Handrechnung ist das fehleranfälligste und aufwändigste Verfahren, da z. B. bei einer Änderung des Bauwerksentwurfs die komplette Berechnung wiederholt werden muss. Bei Verwendung einer Tabellenkalkulation können immerhin Abhängigkeiten zwischen einzelnen Zellen über Formeln hergestellt werden, so dass bei Geometrieänderungen nur ein Teil der Tabellen überarbeitet werden muss. Die Umsetzung im CAD-System bietet den Vorteil, dass die Lastabtragskomponenten direkt auf Basis der Geometrie erzeugt und so Fehler verringert werden. Jedoch besteht auch hier keine Verknüpfung zwischen den Geometrie- und Lastabtragsobjekten, was bei Vorliegen geänderter Pläne eine aufwändige Nach- oder Neubearbeitung bedeutet.

Für die Identifikation besitzen die Lastabtragskomponenten eine eindeutige Bezeichnung bestehend aus einem Buchstaben für den Bauteiltyp, ein oder zwei Buchstaben für die Bauteilorientierung und eine fortlaufende Nummer. Tabelle 5.5 zeigt alle verwendbaren Bauteile, Lasten und Lasteinwirkungsarten. Eine rechteckige Decke wird in trapezförmige Deckenflächen mit einem Winkel von 45° aufgeteilt. Die Höhe der Lastweiterleitung von Last zu Bauteil bzw. Bauteil zu Bauteil kann durch Lastweiterleitungsfaktoren angepasst werden. Die Lasten eines Bauteils sind erst dann vollständig abgetragen, wenn die Summe der Faktoren 1,0 ergibt.

Bauteiltyp	Bauteilorientierung (Kennung der Ebene)	Bauteilunabhänige Lasten	Lasteinwirkungs-arten
Deckenfläche	A, B, ...	Punktlast	Eigengewicht
Unterzug	AA, BB, ...	Linienlast	Ständige Last
Stütze	AB, BC, ...	Reaktionslast	Nichtständige Last
Wand	dto.		

Tabelle 5.5: Lastabtragskomponenten (Bauteile, Lasten) und Lasteinwirkungsarten

5.4.2 Konzept für eine Neuentwicklung

Rahmenbedingungen: Im Hinblick auf die spätere Objektversionierung unter Verwendung der in Java programmierten Systemarchitektur war es einfacher, den Lastabtrag ebenfalls in Java neu zu implementieren. In Absprache mit dem Praxispartner sollte eine intuitive und vor allem durchgängige Lösung gefunden werden. Bei dem bisherigen CAD-basierten Verfahren müssen zum Beispiel die Lastabtragsdaten erst in eine Datenbank zur Auswertung über Abfragen exportiert werden.

Modellierung: Das entworfene Lastabtragsmodell enthält zum einen eine Menge mit Lastabtragskomponenten und zum anderen eine Menge mit Verknüpfungsobjekten, die die Abhängigkeiten zwischen den Lastabtragskomponenten modellieren und so die Topologie bzw. den Lastfluss abbilden. Die Lastabtragskomponenten speichern alle eingehenden Lasten und können ihren Anteil an den jeweiligen Lasteinwirkungsarten zurückgeben. Die Abhängigkeiten werden vom Tragwerksplaner von Hand angelegt, wobei ein Lastweiterleitungsfaktor anzugeben ist. Bei diesem Vorgang spielen das Fachwissen und die Erfahrung des Planers eine nicht zu unterschätzende Rolle.

Durch die Modellierung entsteht ein Graph, dessen Knoten aus Lastabtragskomponenten und dessen Kanten aus Verknüpfungen bestehen. Für die Lastabtragsberechnung wird die Reihenfolge der Lastweiterleitung mit einer topologischen Sortierung des Graphen bestimmt[5]. Die Last einer Komponente kann erst dann weitergeleitet werden, wenn alle eingehenden Lasten der Komponente bekannt sind.

[5] s. (Pahl u. Damrath, 2000), (Olivier, 2007)

Umsetzung: Ziel der Umsetzung ist es, Modell und Ein-/Ausgabe voneinander zu trennen. Somit wird eine eigenständige Lastabtragsanwendung erstellt, die das Modell verwaltet, die Berechnung der Lastweiterleitung vornimmt und das Modell grafisch darstellt. Die Bearbeitung des Modells soll stattdessen in CADEMIA stattfinden. Aus diesem Grund werden die Lastabtragsobjekte in CADEMIA durch Proxy-Objekte[6] entsprechend dem Proxy-Entwurfsmuster (Gamma u. a., 2004) gekapselt.

Abbildung 5.10 zeigt dies am Beispiel des Balkens. Von der Schnittstelle *INamedObject* leiten sich alle Objekte der Lastabtragsanwendung ab, damit sie mit einem eindeutigen Namen innerhalb des Modells versehen werden. Die abstrakte Klasse *LoadTakeDownComponent* leitet sich von INamedObject ab und stellt gemeinsame Attribute und Methoden für alle Lastabtragskomponenten bereit, so auch für die Klasse *Beam*. Auf der Seite von CADEMIA gibt es die Schnittstelle *Proxy*, die die bekannte Schnittstelle *Component* erweitert und über die Methode *getWrappedComponent()* Zugriff auf das gekapselte INamedObject gewährt. Die Klasse *BeamProxy* implementiert die Schnittstelle *Proxy* und leitet sich vom *ComponentAdapter* ab. Die Proxy-Objekte verfügen über die Funktionalität zum Bearbeiten und Anzeigen der zugehörigen Lastabtragskomponente in CADEMIA.

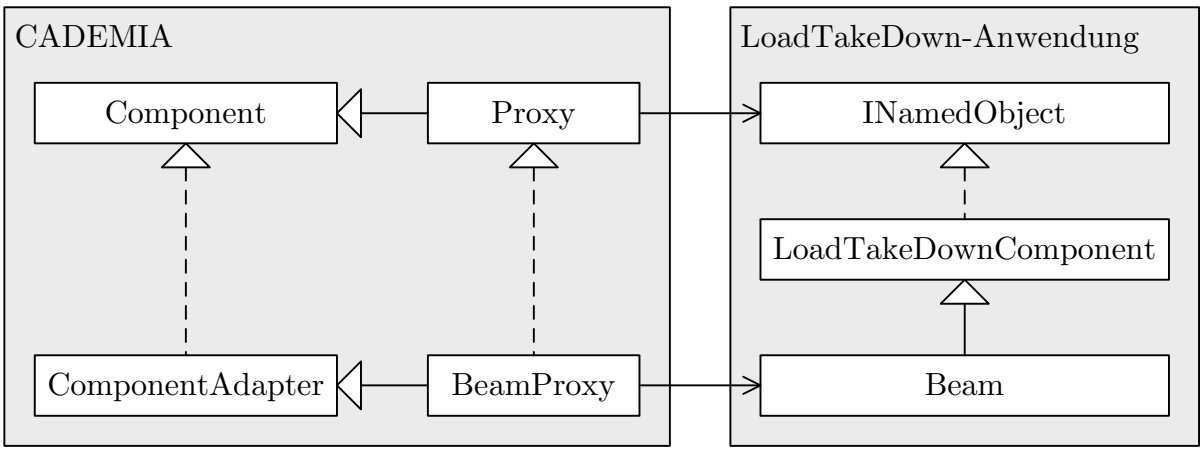


Abbildung 5.10: Lastabtragsanwendung: Proxy-Entwurfsmuster

Zusätzlich müssen für CADEMIA noch Kommando-Klassen zum Hinzufügen der Lastabtragskomponenten sowie zur Definition der Verknüpfungen geschrieben werden. Eine gute Übersicht über alle Befehle bietet das in Abbildung 5.11 dargestellte Menü *LTD-Components*. In einem Dokument kann gleichzeitig für alle Ebenen das Bauwerks zunächst der horizontale Lastabtrag durchgeführt werden. Danach muss jedem Grundriss ein eigenes Koordinatensystem mit einer absoluten Höhe zugeordnet werden, damit die Definition von vertikalen Verknüpfungen möglich ist. Abbildung 5.12 zeigt die Bearbeitung in CADEMIA. Horizontale Lastweiterleitungen sind dabei mit roten Pfeilen und vertikale Lastweiterleitungen mit blauen Pfeilen gekennzeichnet.

[6] proximus ⟨lat., „der Nächste“⟩ → Stellvertreter

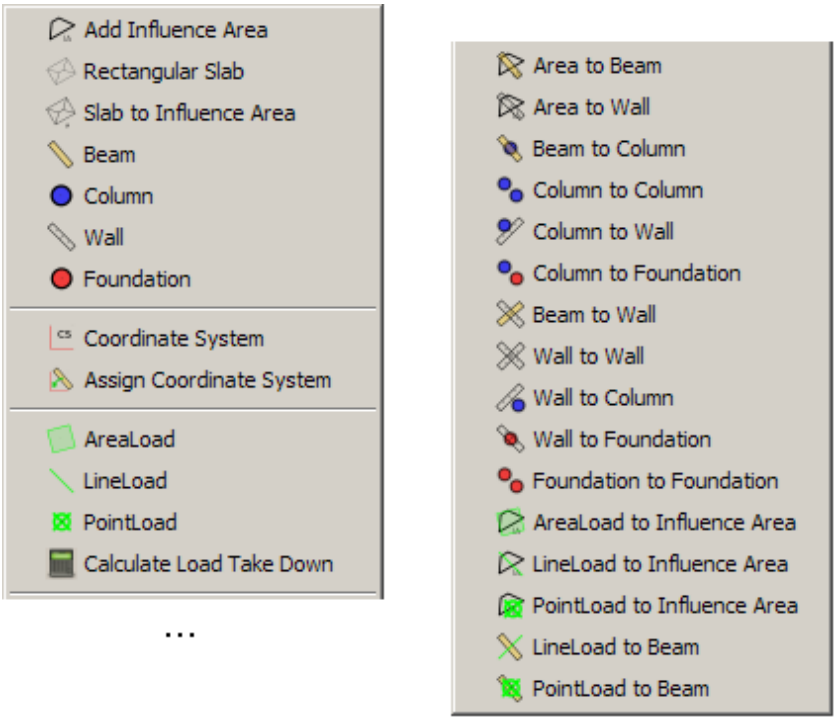


Abbildung 5.11: Lastabtragsanwendung: Menü

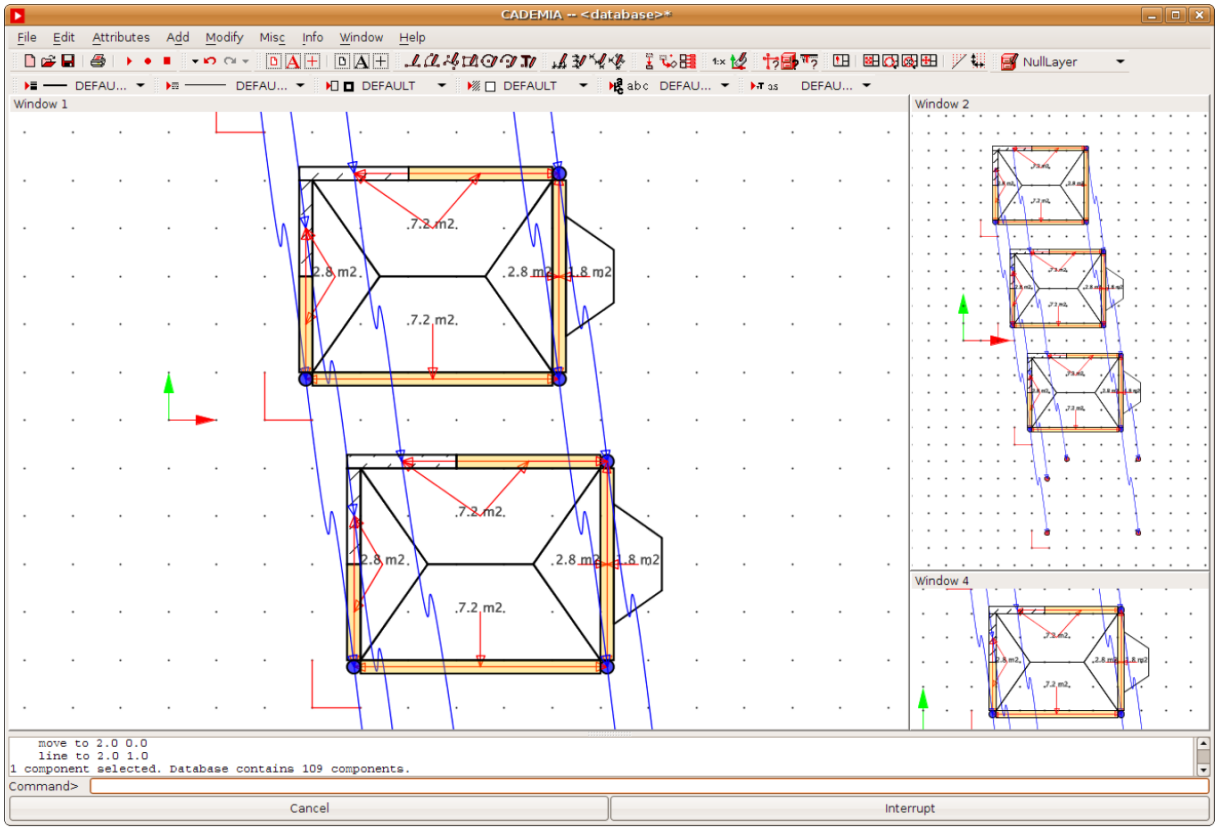
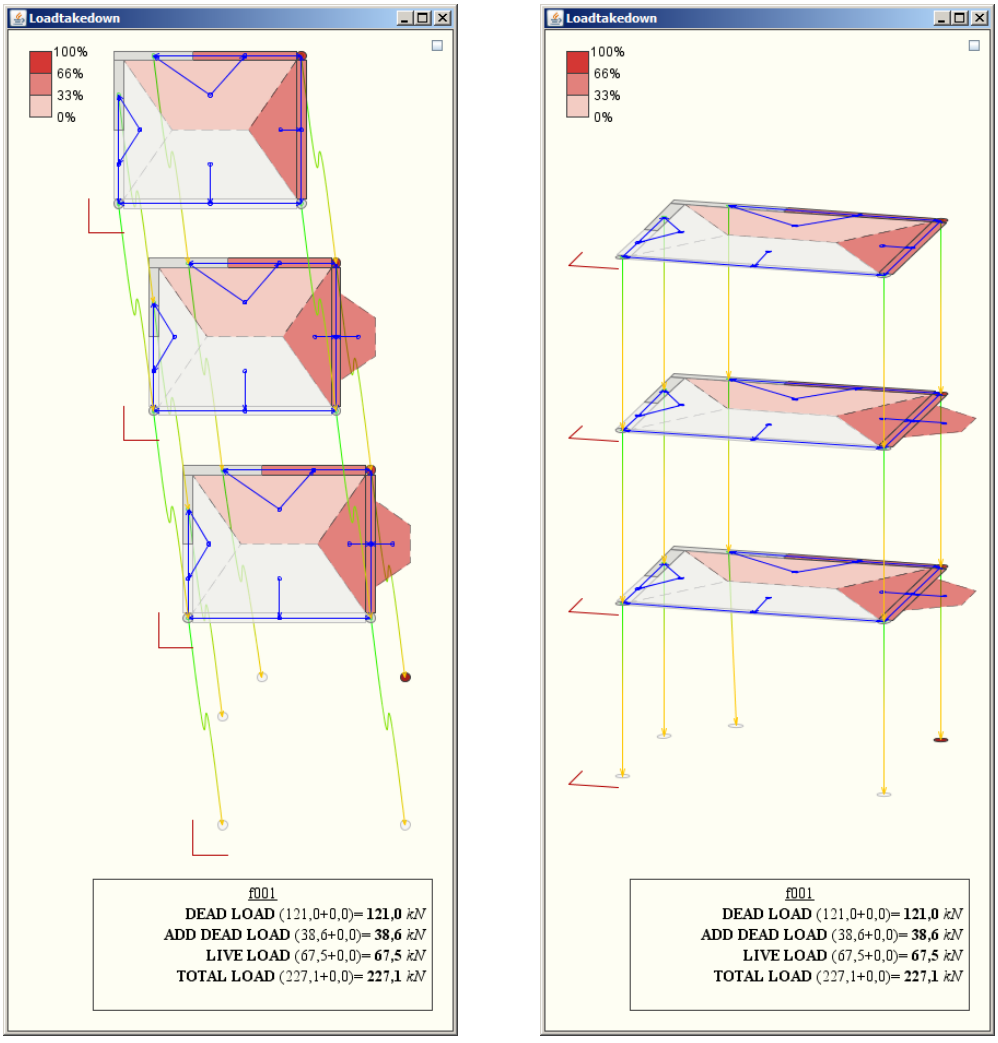


Abbildung 5.12: CADEMIA: Screenshot eines integrierten Lastabtrags

Aus CADEMIA heraus kann die Lastabtragsanwendung zusammen mit dem Modell in eine ausführbare JAR-Datei[7] exportiert und als unabhängiger Betrachter auf jedem Rechner gestartet werden. Das Modell ist in einer 2D-Darstellung (s. Abbildung 5.13a) und in einer 3D-ähnlichen Explosionsdarstellung ohne vertikale Elemente (s. Abbildung 5.13b) darstellbar.

Durch Anwählen einer Lastabtragskomponente werden die Lastabtragskomponenten farbig abgestuft hervorgehoben, die Lasten dorthin abtragen. Die Zusammenfassung im unteren Bereich enthält die berechneten Gesamtlasten der selektierten Komponente aufgeteilt nach Lastfällen. Weiterhin wird bei Überfahren einer Verknüpfung mit der Maus eine Lastübersicht für diese in einem kleinen Fenster (Tooltip) angezeigt.



(a) 2D-Darstellung *(b)* Explosionsdarstellung

Abbildung 5.13: Lastabtragsanwendung: Grafische Ausgabe

[7] JAR = Java Archive. Eine JAR-Datei ist ein komprimiertes ZIP-Archiv zum Speichern von Java-Klassen, das zusätzlich ein Manifest enthält, in dem auf die Startklasse verwiesen wird. Auf Rechnern mit einem installierten JRE sind diese Dateien, sofern sie mit Java verknüpft sind, direkt ausführbar.

5.4.3 Einbindung der Lastabtragsanwendung in die Systemarchtitektur

Workspace: Analog zu Abschnitt 5.3.1 wird die Klasse *WorkspaceLTD*, die spezielle Methoden für die Lastabtragsanwendung implementiert, von der abstrakten Klasse *WorkspaceCademiaAdapter* abgeleitet. Abbildung 5.14 zeigt das Klassendiagramm des neuen Workspace, dessen einziges Attribut *APPLICATION* den Anwendungsnamen auf `loadTakeDownCADEMIA` setzt.

Die in CADEMIA integrierte Lastabtragsanwendung soll es ermöglichen, die Lastabtragsobjekte an Geometrieobjekte von importierten Dokumenten zu binden. Dazu werden implementierte Methoden der Schnittstelle *Workspace* benötigt, die dies unterstützen. Bis auf die Methode *deleteBindings()*, die die Klasse *BinderProxy* benötigt und darauf verweist, sind alle im WorkspaceCademiaAdapter enthalten.

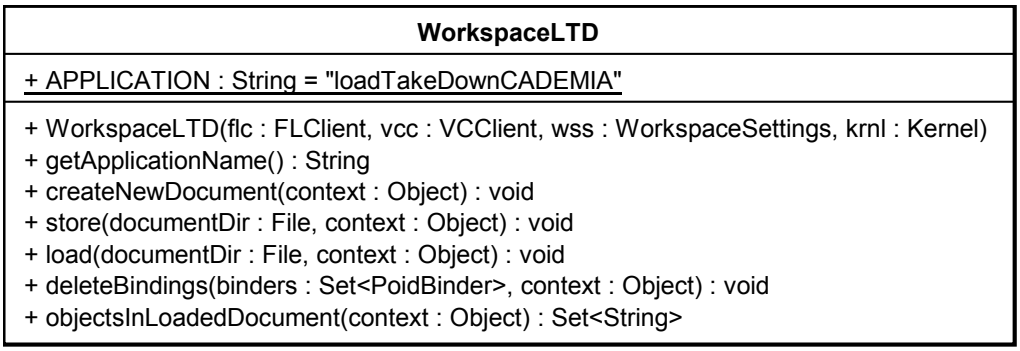


Abbildung 5.14: UML-Klassendiagramm: Workspace für die Lastabtragsanwendung in CADEMIA

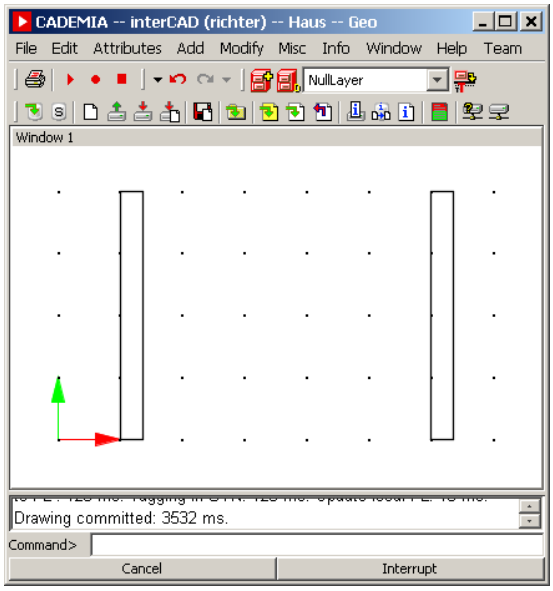
Komponenten: Weder an den Lastabtragsobjekten noch an den Proxy-Objekten muss eine Erweiterung vorgenommen werden, da die Lastabtragsobjekte schon einen eingebetteten Serialisierungmechanismus besitzen. Das Speichern und Laden ist hier nicht über *IOObjectHandler* realisiert, sondern durch zwei Methoden innerhalb der Lastabtragsobjekte. Die Methode *toStream()* schreibt für das Speichern die Objekteigenschaften in einen Ausgabestrom, während die Methode *fromStream()* beim Laden die Eigenschaften wieder aus einem Eingabestrom ausliest.

Befehle: Die bestehenden Befehle für die verteile Bearbeitung können unverändert aus Abschnitt 5.3.3 übernommen und müssen nicht durch neue ergänzt werden.

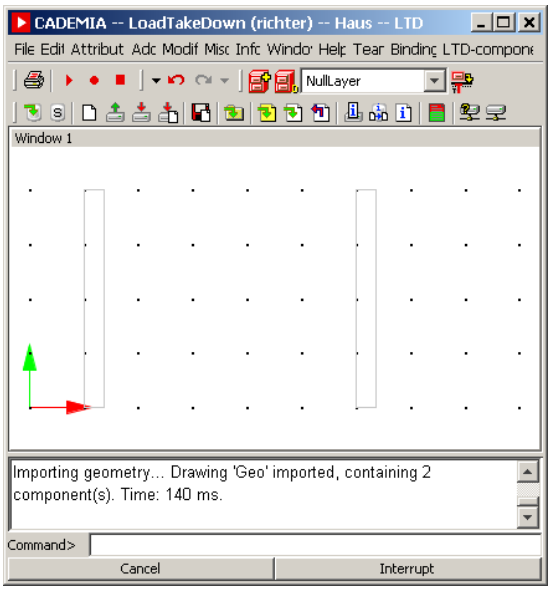
Grafische Darstellung der Bindungen: Anhand eines kleinen Beispiels wird die grafische Darstellung von Bindungen erläutert, das zur Einfachheit nur in einer Sandbox stattfindet. Abbildung 5.15a zeigt zwei Rechtecke in CADEMIA, die Wände symbolisieren sollen. Diese Zeichnung wird unter dem Namen *Geo* auf den Server übertragen und somit versioniert. Auf Basis der nun vorhandenen und eindeutigen POVIDs können Bindungen in der Lastabtragsanwendung definiert werden. Dazu wird das Dokument *Geo* in

das Dokument *LTD* importiert und die Rechtecke auf den Layer *Imported geometry* gelegt sowie grau eingefärbt (Abbildung 5.15b).

Es werden zwei Wände hinzugefügt, wobei die erste genau über dem linken Rechteck und die zweite neben dem rechten Rechteck positioniert wird (Abbildung 5.15c). Im ersten Fall ist es von Vorteil, dass die Wand durch den Objektfang genau die Maße der Geometrie übernimmt. Danach werden mit dem Befehl *bindToGeometry*, der eine Menge von geometrischen Objekten und eine Lastabtragskomponente entgegennimmt, die Bindungen erzeugt. Da in der Lastabtragsanwendung Pfeile schon den Lastfluss anzeigen, werden die Bindungen stattdessen mit einer Umrahmung der Objekte dargestellt (Abbildung 5.15d). Die Farbe variiert in Abhängigkeit von der Gültigkeit der Bindung. Da die eben erzeugten Bindungen gültig sind, wird das bindende Geometrieobjekt mit einer grünen und das gebundene Lastabtragsobjekt mit einer blauen strichlierten Linie umrandet.



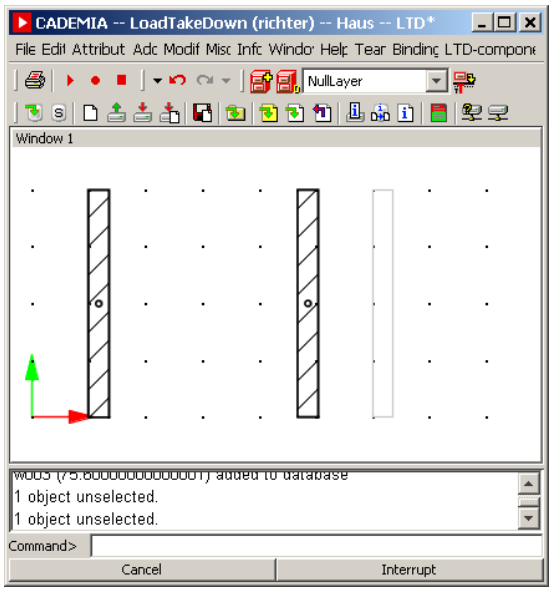
(a) Geometrie 1



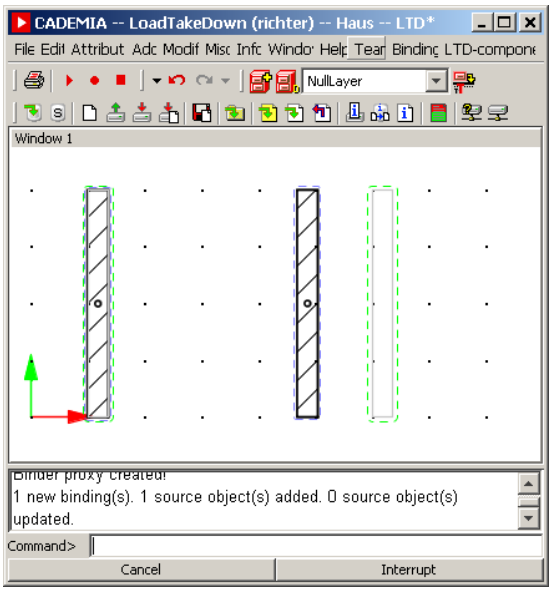
(b) Lastabtrag 1

Abbildung 5.15: Beispiel für die grafische Darstellung von Bindungen, Teil 1

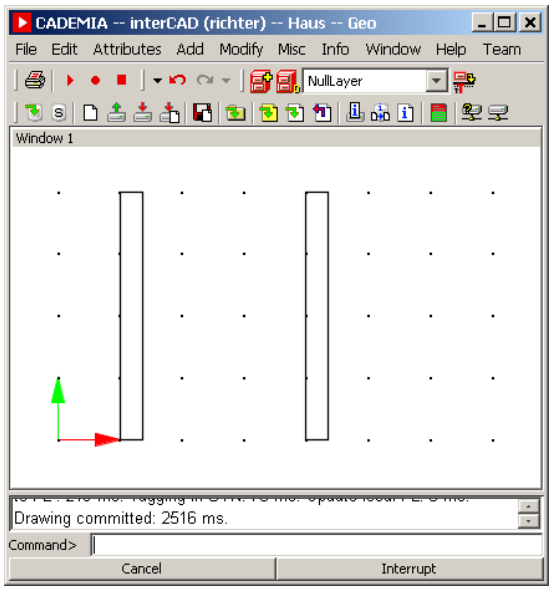
Wie in Abbildung 5.15e zu sehen ist, wird das rechte Rechteck in *Geo* nach links verschoben und das Dokument als nächste Version committet. Nach Ausführen des Befehls *importGeometry* oder *updateBindingDocumentsToLatestVersion* in der Lastabtragsanwendung erscheint das Rechteck an seiner neuen Position. Dadurch, dass die Versionsnummer nicht mehr mit der, die in der Bindung gespeichert ist, übereinstimmt, wird das Rechteck orange und die Wand rot umrandet (Abbildung 5.15f). Nach Prüfung und eventueller Anpassung der Wand lässt sich die Bindung mit dem Befehl *markBindingsAsValid* wieder in einen gültigen Zustand bringen.



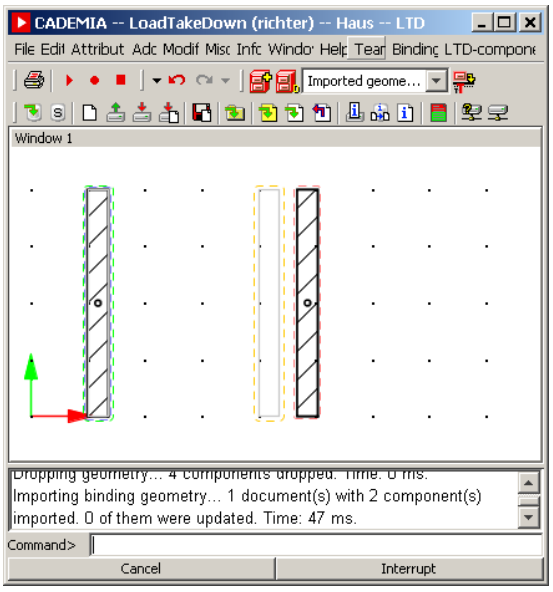
(c) Lastabtrag 2



(d) Lastabtrag 3



(e) Geometrie 2



(f) Lastabtrag 4

Abbildung 5.15: Beispiel für die grafische Darstellung von Bindungen, Teil 2

5.5 Beispielszenario

5.5.1 Situation

Ein Architekt und ein Tragwerksplaner bearbeiten zusammen ein Projekt, in dem ein zweietagiges Haus erstellt werden soll. Sie arbeiten räumlich getrennt und ihre verteilten Fachanwendungen sind an dieselbe Systemarchitektur angeschlossen. Der Architekt entwirft die Geometrie des Hauses mit CADEMIA und der Tragwerksplaner führt passend dazu die Lastabtragsberechnung mit der in CADEMIA integrierten Lastabtragsanwendung durch. Der Entwurf wurde zur besseren Übersichtlichkeit bewusst einfach gehalten.

5.5.2 Ablauf

Geometrieentwurf: Zuerst muss ein Administrator das Projekt *Haus* im Versionsverwaltungssystem Subversion mit einem dazu entworfenen Werkzeug anlegen. Danach können beide Planer das leere Projekt mit der Operation *Projekt beitreten* in ihren Workspace auschecken und bearbeiten. Der Architekt erzeugt für das Erd- und Obergeschoss zwei Grundrisse und speichert sie in die Dokumente *EG* und *OG* (s. Abbildung 5.16 und 5.17).

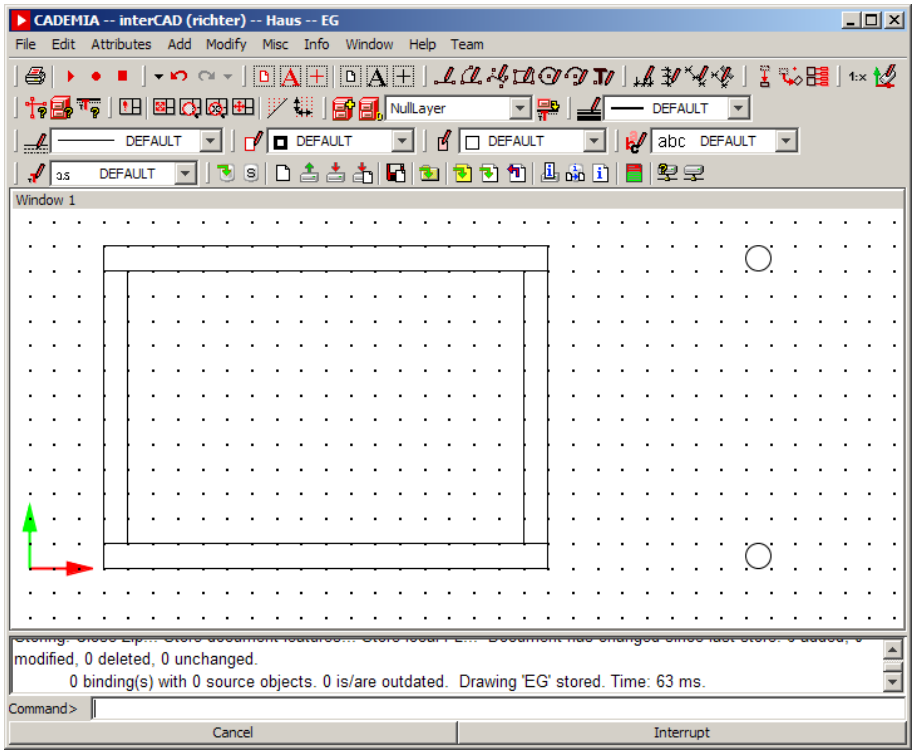


Abbildung 5.16: Szenario: Geometrieentwurf Erdgeschoss

Das Erdgeschoss besteht aus einem Raum und zwei Stützen, die die Lasten aus dem Obergeschoss übernehmen. Das Obergeschoss setzt sich indessen aus zwei Räumen zusammen.

Nach Abschluss des Entwurfs überträgt der Architekt beide Grundrisse mit dem Tag *V1* in das gemeinsame Repository, was in Abbildung 5.18 zu sehen ist.

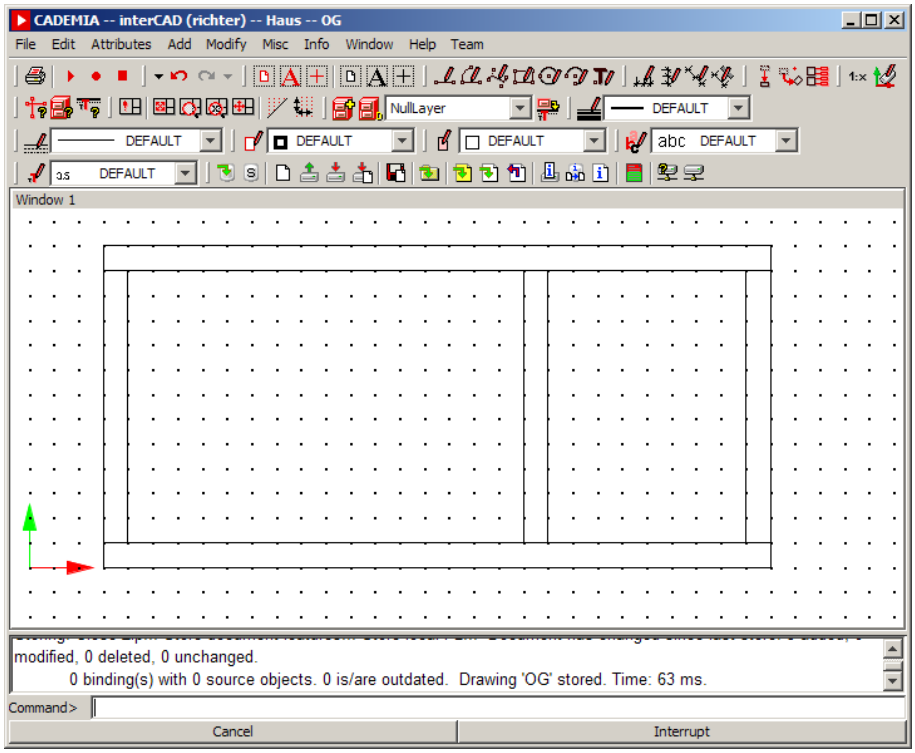


Abbildung 5.17: Szenario: Geometrieentwurf Obergeschoss

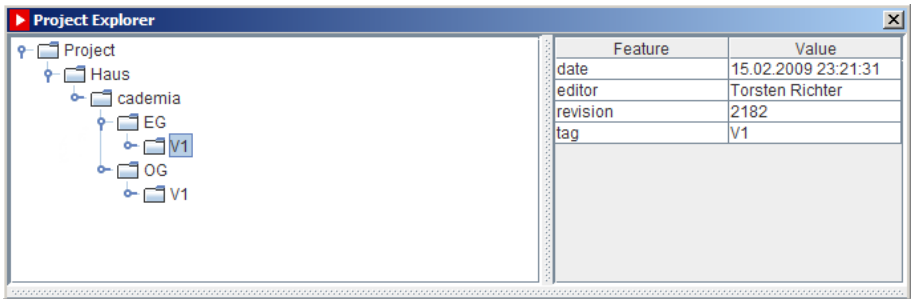


Abbildung 5.18: Szenario: Projektbaum des Projektexplorers nach dem Geometrieentwurf

Lastabtragsberechnung: Als nächstes muss sich der Tragwerksplaner mit dem Befehl *Holen (Update New Documents)* die Zeichnungen in seine Sandbox laden. Er legt in seiner Anwendung ein neues Dokument an und speichert es vor dem Import der Geometrie unter dem Namen *Lastabtrag* ab. Zunächst importiert er das Obergeschoss und schiebt es nach oben, da EG und OG sonst übereinander liegen würden. Danach importiert er das Erdgeschoss und beginnt mit dem Lastabtrag. Als erstes werden die Deckenflächen sowie die Wände und Stützen eingezeichnet und durch Bindungen mit der Geometrie verknüpft. Im Anschluss daran ist die horizontale und vertikale Verknüpfung der Elemente für die

Lastabtragsberechnung vorzunehmen. Jeder Ebene wird im letzten Schritt ein Koordinatensystem mit einer definierten Höhe zugewiesen. Das Ergebnis ist in Abbildung 5.19 zu sehen.

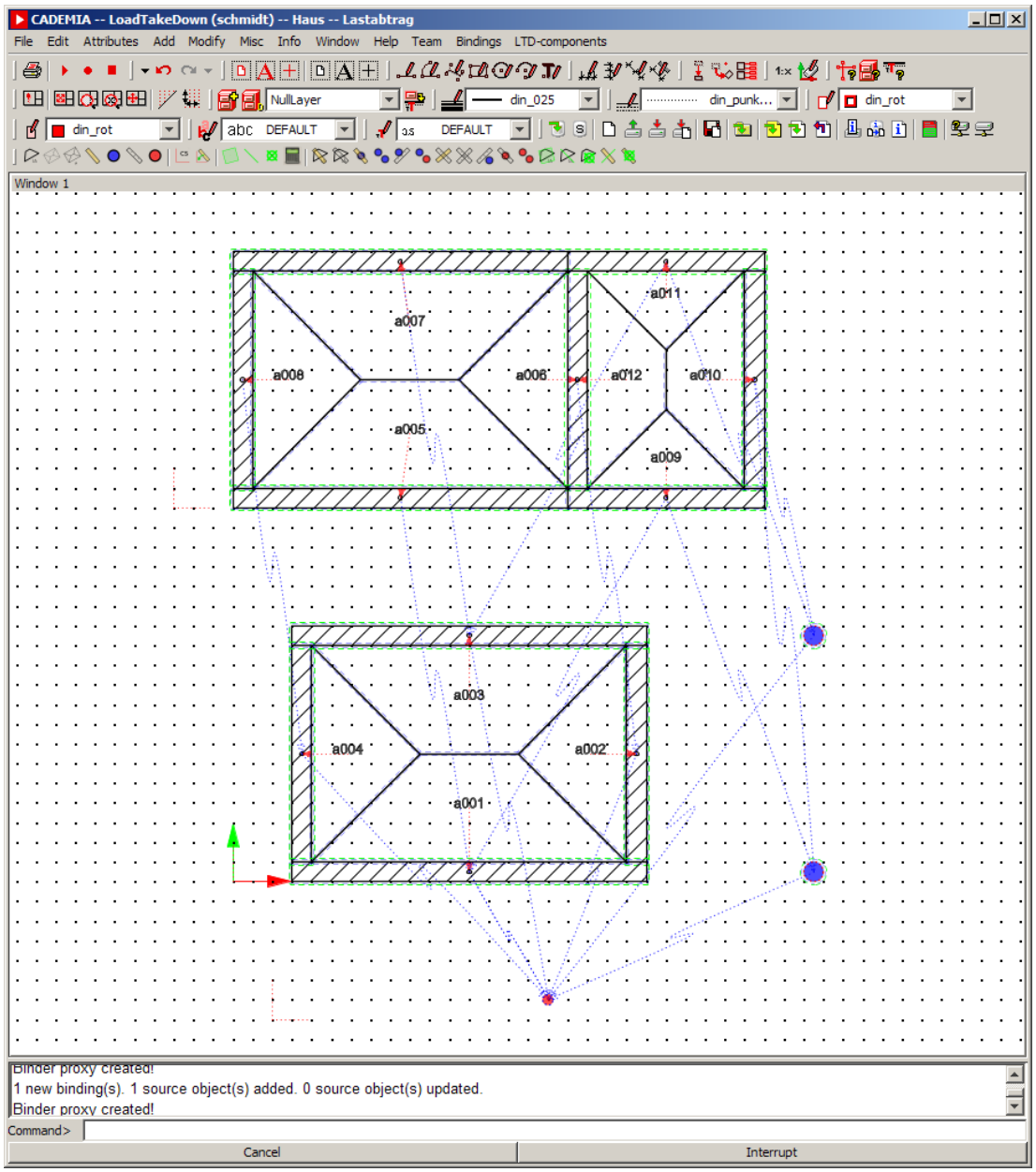


Abbildung 5.19: Szenario: Durchführen des Lastabtrags

Der Lastabtragsbetrachter zeigt in einem eigenen Fenster das Ergebnis wahlweise in der Draufsicht oder in einer dreidimensionalen Explosionsdarstellung mit interaktiver Auswertemöglichkeit an (s. Abbildung 5.20). Das Fundament, dargestellt durch den Kreis in der virtuellen untersten Ebene, fasst alle vertikalen Lasten in einem zentralen Punkt zusammen, um zu kontrollieren, ob alle Lasten erfasst wurden. Abschließend committet der Tragwerksplaner seine Arbeit als erste Version mit dem Tag *V1* auf den Server.

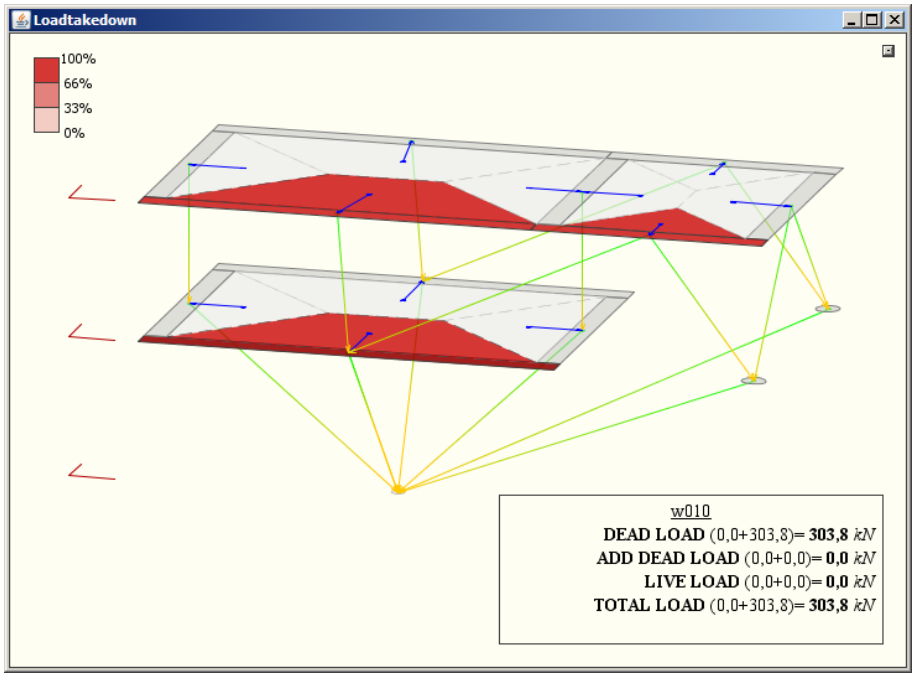


Abbildung 5.20: Szenario: Lastabtragsbetrachter mit der Explosionsdarstellung des Modells

Änderung des Geometrieentwurfs: Der Architekt überarbeitet parallel zum Tragwerksplaner den Geometrieentwurf und verkleinert durch ein Verschieben der rechten Außenwand und der Stützen nach links die Grundfläche. Diese Änderungen überträgt er als neue Dokumentversionen jeweils unter dem Tag *V2* an den Server. Für gewöhnlich wird er seine Änderungen den betroffenen Projektbeteiligten mitteilen. Dies ist eine Frage der Projektorganisation, ob Änderungen aktiv übermittelt (Bringschuld) oder ob das Vorhandensein von Änderungen geprüft werden muss (Holschuld). Die Systemarchitektur könnte diesbezüglich um eine E-Mail-Benachrichtigung unter Festlegung bestimmter Regeln erweitert werden.

Aktualisieren des Lastabtrags: Der Tragwerksplaner prüft durch Ausführen des Befehls *updateBindingDocumentsToLatestVersion* aus dem geladenen Dokument *Lastabtrag* heraus, ob neue bindende Dokumentversionen vorliegen. Dieser Befehl führt gleichzeitig eine Aktualisierung der Geometriedokumente *EG* und *OG* auf die Versionen mit dem Tag *V2* durch. Abbildung 5.21 zeigt für das Erdgeschoss, dass die Kreise nach links verschoben wurden und demzufolge orange markiert sind. Die zwei Stützen als daran gebundene Objekte sind rot umrandet. Das Gleiche trifft für die geänderten Wände und die abhängigen Lastabtragskomponenten im Obergeschoss zu. Hier wird vor allem deutlich, dass die mehrfach überlagerten Komponenten das Erkennen und Lösen von Konflikten erschweren. Eine Lösung wäre, die Komponenten je nach Typ auf verschiedene Layer zu legen und wahlweise auszublenden. Die verwendete Version 1.5a22 von CADEMIA unterstützt dies noch nicht.

Nach der Behebung aller Konflikte werden alle Bindungen mit dem Befehl *markBindingsAsValid* in einen gültigen Zustand versetzt. Das Dokument wird nun vom Tragwerksplaner durch einen Commit als Version *V2* veröffentlicht.

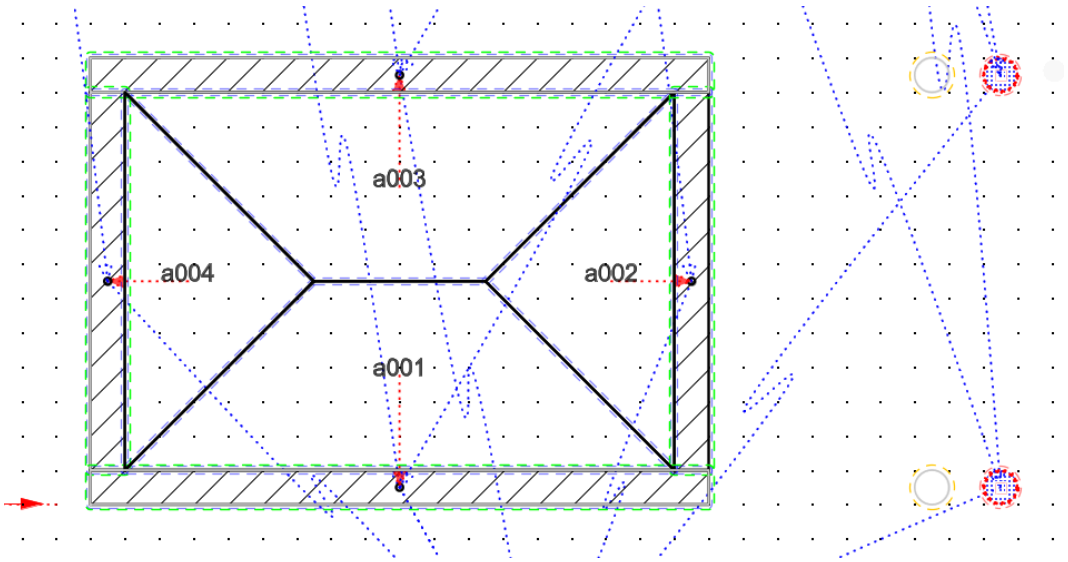


Abbildung 5.21: Szenario: Ungültige Bindungen in der Lastabtragsanwendung für das Erdgeschoss

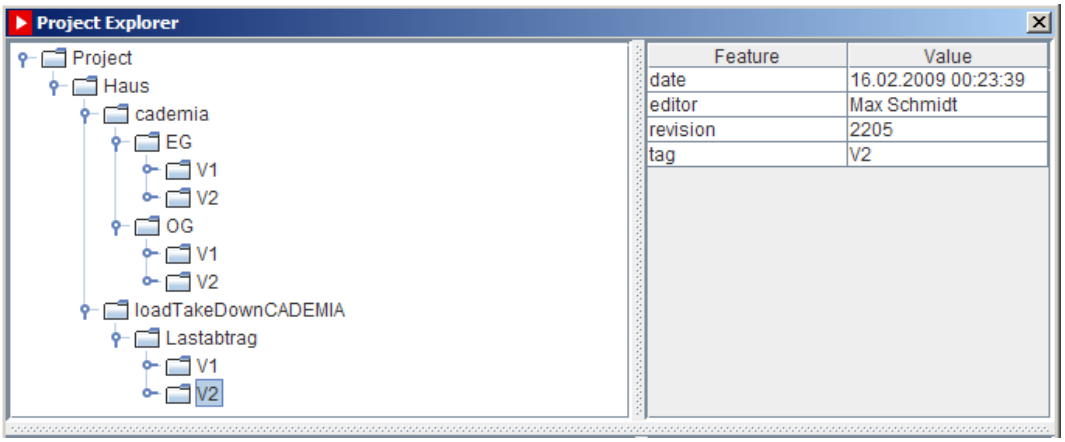


Abbildung 5.22: Szenario: Projektbaum des Projektexplorers nach dem Aktualisieren des Lastabtrags

Freigabestand: In Absprache mit dem anderen Planer kann der Architekt oder Statiker den aktuellen Projektstand freigeben. Durch Aufruf des Befehls *defineProjectState* erscheint der Dialog aus Abbildung 5.23 mit einer Liste der Dokumente in der Sandbox. Das Dokumente *Lastabtrag* ist gebunden und besitzt kein Objekt mit einer ungültigen Bindung, was an der grünen Farbe der Tabellenzeile zu erkennen ist. Nach Auswahl aller Dokumente und Vergabe des Namens *Vorentwurf* kann die Freigabe abgeschlossen werden.

Die korrekte Ausführung der vorangegangenen Operation lässt sich durch Aufrufen des Befehls *updateProjectState* überprüfen, der den Dialog aus Abbildung 5.24 anzeigt. Darin ist eine Liste mit allen jemals im Projekt definierten Freigabeständen aufgeführt.

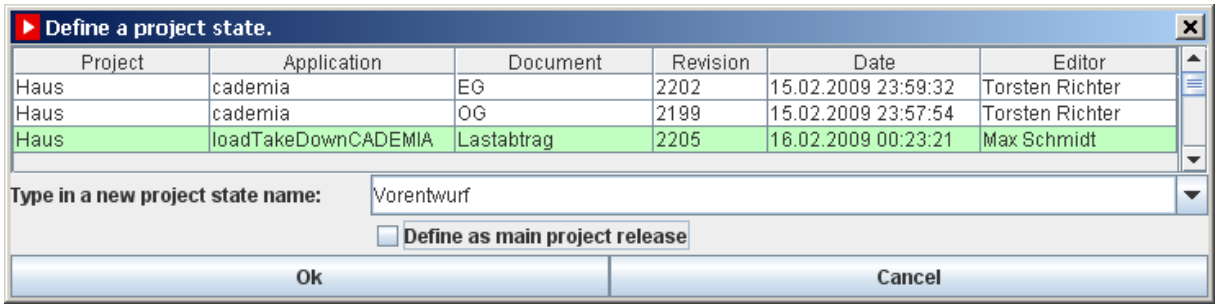


Abbildung 5.23: Szenario: Festlegen eines Freigabestands

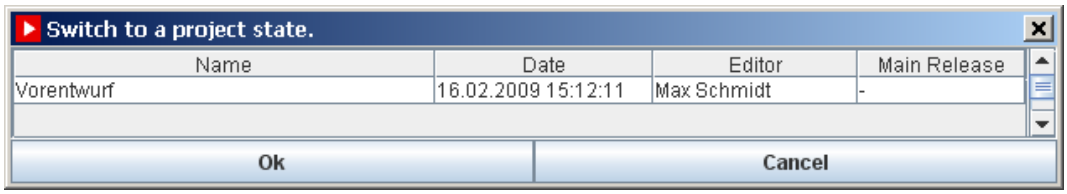


Abbildung 5.24: Szenario: Liste aller Freigabestände

6 Zusammenfassung und Ausblick

> Der beste Weg, die Zukunft vorherzusagen, ist, sie zu erfinden.
>
> *(Alan Kay, 1971)*

6.1 Zusammenfassung

Ausgangssituation: Bauwerke sind in der Regel Unikate, für die meist eine komplette und aufwändige Neuplanung durchzuführen ist. Der Umfang und die Verschiedenartigkeit der einzelnen Planungsaufgaben bedingt ein paralleles Arbeiten der beteiligten Fachplaner. Die Bauplanung ist ein kreativer und iterativer Prozess, der durch häufige Änderungen des Planungsmaterials und Abstimmungen zwischen den Fachplanern gekennzeichnet ist.

Im rechnergestützten Bauplanungsprozess erstellen die Planer mit speziellen Fachanwendungen verschiedene Datenmodelle, zwischen denen fachliche Abhängigkeiten bestehen. Die verteilte Kooperation über Rechnernetze zur Erstellung widerspruchsfreier Bauwerksmodelle wird durch bestehende Software nur unzureichend unterstützt. Das liegt unter anderem daran, dass kein paralleles Arbeiten am gleichen Planungsmaterial möglich ist und dass keine Abhängigkeiten zwischen den Fachmodellen definiert werden.

Die vorherrschende Art der Kooperation im Bauplanungsprozess besteht im Austausch von Datenmodellen über Dokumente in papiergebundener oder digitaler Form. Zum Datenaustausch digitaler Dokumente werden Standardformate verwendet, die sich im Laufe der Zeit etabliert haben. Lässt sich das Datenmodell einer Anwendung nicht komplett mit dem Dateiformat beschreiben, so entstehen Informationsverluste.

Ein Dokumenten-Management-System verwaltet digitale Dokumente an zentraler Stelle und koordiniert somit den Datenaustausch. Das zugrundeliegende pessimistische Zugriffsmodell erlaubt nur das sequentielle Bearbeiten eines Dokuments und verwendet dazu einen Sperrmechanismus. Jeder Zustand eines Dokuments entspricht einer Dokumentversion, wobei der Inhalt und die Struktur dem Dokumenten-Management-System nicht bekannt sind.

Aktuelle Anwendungen im Bauwesen sind objektorientiert programmiert und modellieren das Datenmodell mit Objekten, die zueinander in Beziehung stehen. Der Zustand eines Objekts wird durch seine Attribute beschrieben. Objektversionierte Ansätze versionieren

die Zustände relevanter Objekte eines Datenmodells, wodurch ein paralleles Bearbeiten von mehreren Planern am gleichen Material sowie die Definition von Abhängigkeiten auf Basis der Objektversionen ermöglicht werden soll.

Produktdaten- bzw. Bauwerksinformationsmodelle speichern alle Informationen zum gesamten Lebenszyklus eines Bauwerks, wodurch diese in der Regel sehr groß werden. Sie setzen sich aus semantischen Elementen zusammen und sind selbst objektorientiert modelliert. Es gibt Forschungsansätze, die untersuchen, wie ein solches Modell zentral verwaltet und zum Bearbeiten in kleinere Teilmodelle zerlegt werden kann. Jedoch ist dieser Vorgang aufgrund der Komplexität und der lückenhaften Objektidentifikation problembehaftet. Andere Forschungskonzepte versuchen mehrere unabhängige Modelle durch Bildung von Abhängigkeiten zwischen diesen zu koppeln, um so die Konsistenz sicherzustellen. Eine von der Software automatisch vorgenommene Beseitigung der Inkonsistenzen wird als nicht praktikabel angesehen.

Ziel der Arbeit: Ziel der Arbeit ist es, die Konsistenz der einzelnen Fachmodelle eines Bauwerks sicherzustellen, indem Abhängigkeiten auf Basis von Objektversionen definiert werden. Weiterhin soll das sequentielle und parallele Arbeiten mehrerer Fachplaner auf Basis eines optimistischen Zugriffsmodells unterstützt sowie die Historie des Planungsmaterials gespeichert werden. Der Ansatz soll formal beschrieben werden, so dass er allgemeingültig und technologieunabhängig ist. Auf Grundlage dieses Ansatzes sollen Konzepte für den Einsatz versionierter Objektmodelle im Bauwesen erarbeitet und mit einer Pilotimplementierung an einem praktischen Beispiel verifiziert werden.

Vorgehensweise: Zunächst werden im *Stand der Technik und Forschung* wesentliche Grundlagen und Begriffe der Dokumentation auf digitalen Rechnern sowie der objektorientierten Modellierung als Basis moderner Anwendungen aufgeführt. Danach werden Kooperationskonzepte im Bauwesen untersucht und thematisch eingeordnet. Diese reichen vom klassischen Austausch von Dateien in Standardformaten bis hin zu aktuellen Forschungsansätzen. Besondere Bedeutung für diese Arbeit hat das Objektversionierungskonzept, welches Parallelen zum sehr erfolgreichen Softwarekonfigurationsmanagement aufweist. Bei diesem Konzept werden statt Textdateien die Objekte von Datenmodellen versioniert. Ein Überblick zu bestehenden Werkzeugen zum Vergleichen und Zusammenführen von Datenmodellen sowie zu Anforderungen an die ergonomische Gestaltung von Benutzerschnittstellen schließen dieses Kapitel ab.

Die *formale Beschreibung* des verwendeten Ansatzes wird technologieunabhängig über die Mengenlehre und Relationenalgebra vorgenommen. Dabei wird die Rolle des Dokuments gegenüber dem Ansatz der Objektversionierung mit flexibler Teilmengenbildung wieder stärker betont, da es in der Praxis ein gewohnter Begriff ist und sich bewährt hat. Weiterhin werden von außen Abhängigkeiten zwischen Objektversionen modelliert, die dazu beitragen, die Konsistenz der Teilmodelle sicherzustellen. Zur Verteilung und Bearbeitung des Modells sind verteilte Operationen notwendig, die gleichfalls formal beschrieben werden.

Auf Grundlage der formalen Beschreibung wird eine anwendungsunabhängige *Systemarchitektur* nach dem Client-Server-Prinzip entworfen und implementiert, die lange Transak-

tionen unterstützt und so der Charakteristik des Bauplanungsprozesses entgegenkommt. Ein textbasiertes Versionsverwaltungssystem ist als Teil der Systemarchitektur für die Versionierung der Dokumente verantwortlich. Für die Speicherung der Datenmodelle im lokalen Arbeitsbereich des Clients (Sandbox) ist ein performantes Serialisierungsverfahren zu entwickeln, das die Dokumente in einer für die Versionierung geeigneten Form ablegt. Die Modellierung der Beziehungen im Modell sowie der Versions- und Bindungsgraphen erfolgt über die Mengenalgebra *Feature-Logic*.

Der Entwurf ergonomischer Benutzerschnittstellen ist ein wesentlicher Punkt für die Effizienz und Akzeptanz von Software. Die Bedienung der verteilten Operationen wird deshalb durch geeignete grafische Dialoge umgesetzt. Zur Leistungsverbesserung der Systemarchitektur wurden Möglichkeiten untersucht, wie die Speicherung der Dokumente und der Feature-Logic-Daten sowohl vom Zeit- als auch vom Speicherbedarf optimiert werden kann.

Die entworfenen Konzepte werden durch eine *Pilotimplementierung* für eine Open-Source-Ingenieurplattform an einem praxisnahen Szenario verifiziert, bei dem Fachmodelle von zwei unterschiedlichen Anwendungen aneinander gekoppelt werden. Für die Integration von bestehenden Fachanwendungen in die verteilte Systemarchitektur sind nur anwendungsspezifische Operationen sowie Vergleichs- und Zusammenführungswerkzeuge zu entwickeln.

Erkenntnisse und Bewertung: Die wesentlichen Erkenntnisse dieser Arbeit werden im Folgenden aufgeführt und bewertet.

- Aktuelle Anwendungen im Bauwesen sind objektorientiert programmiert und modellieren das Datenmodell mit Objekten, die zueinander in Beziehung stehen. Der Zustand eines Objekts wird durch seine Attribute beschrieben. Zur Unterstützung des gleichzeitigen Bearbeitens von Datenmodellen durch mehrere Planer als auch zur Definition feingranularer Abhängigkeiten ist es notwendig, die Objekte zu versionieren. Im Gegensatz zu einem Dokumenten-Management-System können damit Änderungen an einem Dokument exakt festgestellt werden. Für den objektversionierten Ansatz ist die eindeutige Identifizierbarkeit der Objekte und Objektversionen von grundlegender Bedeutung. Objekten wird eine POID und Objektversionen eine POVID als dauerhafter Identifikator zugewiesen.

- Die hierarchische Strukturierung des unversionierten Planungsmaterials in Projekte, Anwendungen, Dokumente und Objekte gewährt eine den Planern gewohnte Orientierung in großen Datenbeständen. Das Dokument, das einem in sich geschlossenen Teilmodell entspricht, enthält eine strukturierte Menge von Objekten. Durch die Versionierung von Dokumenten und Objekten ändert sich die Hierarchie im Repository zu Projekte, Anwendungen, Dokumente, Dokument- und Objektversionen. Dokumentversionen setzen sich dann aus einer strukturierten Menge von Objektversionen zusammen. Dokumente erhalten ebenso einen persistenten und eindeutigen Identifikator wie die Dokumentversionen. Die zu jeder Dokumentversion gespeicherten Metadaten Bearbeiter, Datum, Versionsnummer und Versionsname lassen sich effektiv zur Eingrenzung der Auswahl bei verteilten Operationen verwenden.

- Das Speichern eines jeden Objekts in eine Datei verschlechtert die Leistung der Systemarchitektur aufgrund der internen Organisation üblicher Dateisysteme erheblich. Die Datenmodelle werden deshalb auf der Client-Seite durch ein manuelles Serialisierungsverfahren in unkomprimierte ZIP-Archive gespeichert, die sich zuverlässig und speichereffizient mit einem textbasierten Versionsverwaltungssystem verwalten lassen. Die Operationen Speichern und Laden erreichen für große Modelle praxistaugliche Zeiten und weisen eine lineare Komplexität auf.
- Bindungen müssen zur Abbildung der Historie versioniert werden, da sie nicht nur erzeugt, sondern später auch wieder gelöscht werden können. Für eine eindeutige Zuordnung ist die Dokumentversion der bindenden und gebundenen Objektversion zu speichern. Durch die externe Modellierung der Objektabhängigkeiten müssen die Klassenschemas der Datenmodelle nicht erweitert werden, was dann von Vorteil ist, wenn der Quellcode der Anwendung nicht verfügbar ist oder nicht verändert werden kann. Die Modellierung von Freigaben wird ebenfalls in der Feature-Logic vorgenommen, was im Zusammenspiel mit dem Versionsverwaltungssystem später das Laden eines Freigabestands in die Sandbox erlaubt.
- Das Erzeugen von Bindungen durch die direkte Angabe der Objektidentifikatoren ist fehleranfällig und nicht nutzergerecht. Stattdessen wurde dies intuitiv in der Anwendung auf Basis der visualisierten Datenmodelle umgesetzt.
- Ergonomisch gestaltete Dialoge einer grafischen Benutzeroberfläche erleichtern die Durchführung der verteilten Operationen. Für die Anzeige des Projektstatus und für die Operationen zum Übertragen von Dokumentversionen in die Sandbox eignet sich ein Dialog mit einer Baumdarstellung der Projektstruktur und einer flexiblen Filterung der Dokumentversionen über die Metadaten. Eine Tabellendarstellung mit farblicher Hervorhebung von Tabellenzeilen zur Verdeutlichung ausgewählter Aspekte ist für die Operationen *Sandboxstatus, Dokumenthistorie, Freigabestand definieren* und *Wechseln zu einem Freigabestand* ausreichend.
- Vergleichs- und Zusammenführungswerkzeuge ermöglichen erst das parallele Arbeiten. Sie sind anwendungs- bzw. modellabhängig entwickelt und präsentieren dem Nutzer das Modell in der gewohnten Form. POIDs sind eine Voraussetzung für diese Operationen. Da jedes Objekt im serialisierten ZIP-Archiv einen ZIP-Eintrag belegt, lässt sich zur Prüfung der Zustandsänderung der Prüfsummenwert des ZIP-Eintrags heranziehen, auch wenn das Modell nicht instanziiert ist.
- Die Übertragung großer Modelle in das Versionsverwaltungssystem benötigt bei großen Modellen geringe Zeiten. Das Übertragen von Daten in die Feature-Logic auf dem Server dauert trotz Optimierung der Datenbankoperationen und einem Zwischenspeichern einzelner Relationen wesentlich länger. Das Zwischenspeichern ist nur bis zu einer bestimmten Größe des Datenbestands in Abhängigkeit vom Arbeitsspeicher des Servers anwendbar.
- Bestehende objektorientierte Fachanwendungen lassen sich durch geringen Aufwand in die Systemarchitektur einbinden, da ein Großteil der Client-Funktionalität anwendungsunabhängig implementiert ist. Die Pilotimplementierung zeigt die Praxistaug-

lichkeit des Objektversionierungsansatzes anhand eines durchgeführten Szenarios am Beispiel der Kopplung des vertikalen Lastabtrags an die Bauwerksgeometrie zu Beginn der Tragwerksplanung.

6.2 Ausblick

Mit der vorgestellten Systemarchitektur und den implementierten Erweiterungen für die Fachanwendungen konnte die prinzipielle Anwendbarkeit des objektversionierten Ansatzes gezeigt werden. Der Anspruch dieser Arbeit ist es jedoch nicht, eine marktreife Software zu entwickeln. In der Fortführung dieser Arbeit sind folgende Punkte untersuchenswert.

Variantenmanagement: Die Bildung von Dokumentvarianten wurde zwar theoretisch behandelt, aber noch nicht in der Systemarchitektur umgesetzt. Die Modellierung des Versionsgraphen mit parallelen Objekt- und Dokumentversionen sollte wie bisher in der Feature-Logic erfolgen. Weiterhin werden zusätzliche Operationen für das Bilden und Zusammenführen von Varianten benötigt. Die Erweiterung der Systemarchitektur um ein Variantenmanagement ermöglicht das „Durchspielen“ von Alternativen. Besondere Anforderung stellt die Visualisierung der Projektstruktur dar, da sich der gerichtete, azyklische Graph nicht in einen Baum überführen lässt. Stattdessen müsste er mit einer Graph-Komponente in der grafischen Benutzeroberfläche dargestellt werden.

Bindungsexplorer: Die Handhabung der Bindungen sollte in der Anwendung verbessert werden. Denkbar ist ein eigenes Fenster, das alle Bindungen mit den bindenden und gebundenen Objekten in einer Baumdarstellung enthält. Die Gültigkeit der Bindung sollte entsprechend im Bindungsknoten angezeigt werden und eine Selektion von Objektknoten die zugehörigen Objekte in der Anwendung hervorheben.

Vergleichen und Zusammenführen: Der im Prototyp verwendete Diff- und Merge-Dialog basiert auf dem Vergleich und Zusammenführen von ZIP-Einträgen. Eine Weiterentwicklung sollte das Zusammenführen mit einer Änderung von Objektzuständen unterstützen und muss dazu stärker in die Anwendung integriert werden.

Nutzerverwaltung und Rollenkonzept: In der vorliegenden Implementierung sind alle Nutzer gleichwertig und haben Schreib- und Lesezugriff auf alle Projektressourcen. Die erfolgreiche Durchführung eines Projekts setzt aber ein Rollenkonzept und eine Nutzerverwaltung voraus. Jedem Nutzer wird eine Rolle mit bestimmten Berechtigungen je nach Verantwortungsbereich zugewiesen. Zum Beispiel sollten die Fachplaner nur in ihrer Domäne ein Schreibrecht sowie Freigaben nur vom Projektleiter erteilt werden.

Akzeptanz in der Praxis: Die Akzeptanz neuer Kooperationsansätze hängt wesentlich von der Aus- bzw. Weiterbildung von Planern sowie der Unterstützung und dem Einsatzwillen maßgeblicher Entscheidungsträger ab. Durch das vorgestellte Konzept der Objektversionierung könnte sich der Bauplanungsprozess grundlegend ändern, falls es durchgängig angewendet werden würde. Vorbild ist und bleibt das erfolgreiche Softwarekonfigurationsmanagement.

Literaturverzeichnis

[Aho u. a. 2002] Aho, A. V. ; Sethi, R. ; Ullman, J. D.: *Compilers: principles, techniques and tools*. Reading, Mass. [u. a.] : Addison-Wesley, 2002

[Autodesk 2008] Autodesk: *TrustedDWG*. 2008. – http://www.autodesk.de/adsk/servlet/item?siteID=403786&id=10687492 – Online-Ressource, Abruf: 04.10.2008

[Bair 1989] Bair, J. H.: Supporting cooperative work with computers: Addressing meeting mania. In: *Proceedings of 34th IEEE Computer Society International Conference–CompCon Spring*. San Francisco, CA : IEEE, Februar 1989, S. 208–217

[Bartholmai u. Gornig 2008] Bartholmai, B. ; Gornig, M.: *Strukturdaten zur Produktion und Beschäftigung im Baugewerbe - Berechnungen für das Jahr 2006*. Januar 2008. – Gutachten im Auftrag des Bundesamtes für Bauwesen und Raumordnung – http://www.bmvbs.de/Bauwesen/Bauwirtschaft-,1520/Daten-zur-Bauwirtschaft.htm Online-Ressource, Abruf: 18.06.2008

[Beer 2005] Beer, D. G.: *Systementwurf für verteilte Applikationen und Modelle im Bauplanungsprozess*, Bauhaus-Universität Weimar, Dissertation, 2005. – http://e-pub.uni-weimar.de/frontdoor.php?source_opus=789&la=de ; http://www.shaker.de/Online-Gesamtkatalog/details.asp?ID=8370947&CC=59435&ISBN=3-8322-5060-3

[Bentley 2008] Bentley Systems (Hrsg.): *ProjectWise Explorer V8i Help*. Exton, PA: Bentley Systems, 2008. – Deutsch (18.12.2006): http://docs.bentley.com/de/PWExplorer/ Englisch (31.10.2008): http://docs.bentley.com/en/PWExplorer/– Online-Ressource, Abruf: 30.01.2009

[Berkhahn u. a. 2007] Berkhahn, V. ; Klinger, A. ; Hofmann, F. ; König, M.: Relationale Prozessmodellierung in kooperativer Gebäudeplanung. In: Rüppel, Uwe (Hrsg.): *Abschlussbericht zum DFG-Schwerpunktprogramm „Vernetzt-kooperative Planungsprozesse im Konstruktiven Ingenieurbau“*. Heidelberg : Springer, 2007, S. 31–51

[Berliner 1990] Berliner, B.: CVS II, Parallelizing Software Development. In: *Proceedings of USENIX Conference Winter*. Washington, DC : USENIX Association, Januar 1990, S. 341–352

[Beucke 2002] Beucke, K.: *CAE im Planungsprozess*, Bauhaus-Universität Weimar, Vorlesungsskript, März 2002. – http://www.uni-weimar.de/cms/index.php?id=9729

[Beucke u. a. 2007] BEUCKE, K. ; FIRMENICH, B. ; BEER, D. G. ; RICHTER, T.: Entwurf und Verifizierung einer CAD-Systemarchitektur zur Unterstützung der verteilten technischen Bearbeitung im Konstruktiven Ingenieurbau. In: RÜPPEL, Uwe (Hrsg.): *Abschlussbericht zum DFG-Schwerpunktprogramm „Vernetzt-kooperative Planungsprozesse im Konstruktiven Ingenieurbau“*. Heidelberg : Springer, 2007, S. 133–154

[Booch 1995] BOOCH, G.: *Objektorientierte Analyse und Design mit praktischen Anwendungsbeispielen*. Bonn [u. a.] : Addison-Wesley, 1995

[Bourne 1988] BOURNE, S. R.: *Das UNIX System V*. Bonn [u. a.] : Addison-Wesley, 1988

[Bretschneider 1998] BRETSCHNEIDER, D.: *Modellierung rechnerunterstützter, kooperativer Arbeit in der Tragwerksplanung*. Düsseldorf : VDI-Verlag, 1998 (Fortschrittsberichte VDI Reihe 4 Nr. 151). – ISBN 3–18–315104–9. – Dissertation

[Chou u. Kim 1986] CHOU, H.-T. ; KIM, W.: A Unifying Framework for Version Control in a CAD Environment. In: *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1986, S. 336–344

[Claus u. Schwill 1993] CLAUS, V. ; SCHWILL, A. ; ENGESSER, H. (Hrsg.): *Duden Informatik, Ein Sachlexikon für Studium und Praxis*. 2. Auflage. Mannheim, Leipzig, Wien, Zürich : Dudenverlag, 1993

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

[Collins-Sussman u. a. 2008] COLLINS-SUSSMAN, B. ; FITZPATRICK, B. W. ; PILATO, C. M.: *Version Control with Subversion: For Subversion 1.5*. Version: r3415, 2008. http://svnbook.red-bean.com/. – Online-Ressource, Abruf: 25.01.2009

[Conradi u. Westfechtel 1998] CONRADI, R. ; WESTFECHTEL, B.: Version Models for Software Configuration Management. In: *ACM Computing Surveys* 30 (1998), S. 232–282

[Dahm 2006] DAHM, M.: *Grundlagen der Mensch-Computer-Interaktion*. München [u. a.] : Pearson Studium, 2006

[DIN 6789-1 1990] Norm DIN 6789 Teil 1 September 1990. *Dokumentationssystematik; Aufbau Technischer Produktdokumentationen*

[DIN 6789-2 1990] Norm DIN 6789 Teil 2 September 1990. *Dokumentationssystematik; Dokumentensätze Technischer Produktdokumentationen*

[DIN 6789-5 1995] Norm DIN 6789 Teil 5 Oktober 1995. *Dokumentationssystematik; Freigabe in der Technischen Produktdokumentation*

[DIN ISO 10007 2004] Norm DIN ISO 10007 Dezember 2004. *Qualitätsmanagement - Leitfaden für Konfigurationsmanagement*

[DIN ISO 10303] Norm DIN ISO 10303 . *Industrielle Automatisierungssysteme und Integration - Produktdatendarstellung und -austausch*

[DIN ISO 11442 2006] Norm DIN ISO 11442 Juni 2006. *Technische Produktdokumentation – Dokumentenmanagement*

[Eddon u. Eddon 1998] EDDON, G. ; EDDON, H.: *Inside Distributed COM: Entdecken Sie die Programmierung von verteilten Applikationen*. Unterschleißheim : Microsoft Press, 1998

[Emmerich 2003] EMMERICH, W.: *Konstruktion von verteilten Objekten*. Heidelberg : dpunkt, 2003

[Estublier u. a. 2005] ESTUBLIER, J. ; LEBLANG, D. B. ; HOEK, A. van d. ; CONRADI, R. ; CLEMM, G. ; TICHY, W. F. ; WEBER, D. W.: Impact of software engineering research on the practice of software configuration management. In: *Transactions on Software Engineering and Methodology (TOSEM)* 14 (2005), Nr. 4, S. 383–430

[Fahrig 2007] FAHRIG, T.: *Kooperative Optimierung von Raumluftströmungen mittels agentengestützter Regelungstechnik in einer Computational Steering Umgebung*, Technische Universität Braunschweig, Dissertation, 2007. – http://www.digibib.tu-bs.de/?docid=00021869

[Firmenich 2000] FIRMENICH, B.: Eine CAD-Systemarchitektur zur Unterstützung der verteilten Bearbeitung im Internet. In: HANFF, J. (Hrsg.) ; KASPAREK, E. (Hrsg.) ; RUESS, M. (Hrsg.) ; SHUTTE, G. (Hrsg.): *12. Forum Bauinformatik*. Düsseldorf : VDI, August 2000. – ISBN 3–18–316304–3. – Beitrag nicht im Tagungsband erschienen.

[Firmenich 2002] FIRMENICH, B.: *CAD im Bauplanungsprozess: Verteilte Bearbeitung einer strukturierten Menge von Objektversionen*. Aachen : Shaker, 2002 (Berichte aus dem Bauwesen). – Dissertation

[Firmenich 2006] FIRMENICH, B.: *Grundlagenorientierter Systementwurf*, Bauhaus-Universität Weimar, Vorlesungsskript, Mai 2006. – http://gonzo.uni-weimar.de/~bauinf/cad/lehre/cib1/downloads/Skript/CIB1.pdf

[Firmenich u. a. 2005] FIRMENICH, B. ; KOCH, Ch. ; RICHTER, T. ; BEER, D. G.: Versioning structured object sets using text based Version Control Systems. In: SCHERER, R. J. (Hrsg.) ; KATRANUSCHKOV, P. (Hrsg.) ; SCHAPKE, S.-E. (Hrsg.): *Proceedings of 22nd CIB-W78 Conference on Information Technology in Construction*. Dresden : Institute of Construction Informatics, Juli 2005, S. 105–112

[Firmenich u. a. 2008] FIRMENICH, B. ; KOCH, Ch. ; RICHTER, T. ; OLIVIER, A. H. ; BEER, D. G.: CADEMIA: A Platform for the Development of Civil Engineering Applications. In: *Proceedings of the Twelfth International Conference on Computing in Civil and Building Engineering (ICCCBE-XII)*. Beijing : Tsinghua University, Oktober 2008

[Firmenich u. Rank 2007] FIRMENICH, B. ; RANK, E.: Überblick zum Themenbereich Verteilte Produktmodelle. In: RÜPPEL, Uwe (Hrsg.): *Abschlussbericht zum DFG-Schwerpunktprogramm „Vernetzt-kooperative Planungsprozesse im Konstruktiven Ingenieurbau“*. Heidelberg : Springer, 2007, S. 121–132

[Fischer u. Hofer 2008] FISCHER, P. ; HOFER, P.: *Lexikon der Informatik*. 14. Auflage. Berlin, Heidelberg : Springer, 2008

[Fogel u. Bar 2002] FOGEL, K. ; BAR, M.: *Open Source-Projekte mit CVS*. Bonn : mitp, 2002. – http://cvsbook.red-bean.com/

[Gamma u. a. 2004] GAMMA, E. ; HELM, R. ; JOHNSON, R. ; VLISSIDES, J.: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. München : Addison-Wesley, 2004

[Gray 1981] GRAY, J.: The Transaction Concept: Virtues and Limitations. In: *Proceedings of the Seventh International Conference on Very Large Data Bases*, VLDB Endowment, 1981, S. 144–154

[Grune 1986] GRUNE, D.: *Concurrent Versions System, a method for independent cooperation*. IR 113, Vrije Universiteit, 1986

[Gumm u. Sommer 2006] GUMM, H.-P. ; SOMMER, M.: *Einführung in die Informatik*. 7. Auflage. München [u. a.] : R. Oldenbourg, 2006

[Hacker 1986] HACKER, W.: *Arbeitspsychologie : psychische Regulation von Arbeitstätigkeiten*. Berlin : Dt. Verl. der Wissenschaften, 1986

[Hanff 2003] HANFF, J.: *Abhängigkeiten zwischen Objekten in ingenieurwissenschaftlichen Anwendungen*. Aachen : Shaker, 2003 (Berichte aus dem Bauwesen). – Dissertation

[Hansen u. Binar 2005] HANSEN, H. ; BINAR, K.: *Durchführung eines Lastabtrags*. Hochtief Construction AG, IKS, 2005. – Interner Technischer Bericht

[Hardy 2000] HARDY, V. J.: *Java 2D API graphics*. Palo Alto : Prentice Hall, 2000 (The Sun Microsystems Press Java Series)

[Harris 2005] HARRIS, J.: *Novell Open Enterprise Server Administrator's Handbook, NetWare Edition*. Indianapolis, Ind. : Pearson, 2005

[Hartmann 2007] HARTMANN, D.: Überblick zum Themenbereich Agentensysteme. In: RÜPPEL, Uwe (Hrsg.): *Abschlussbericht zum DFG-Schwerpunktprogramm „Vernetzt-kooperative Planungsprozesse im Konstruktiven Ingenieurbau“*. Heidelberg : Springer, 2007, S. 323–334

[Haskin u. a. 1982] HASKIN, R. L. ; ROGER, L. ; LORIE, R. A.: On extending the functions of a relational database system. In: *SIGMOD '82: Proceedings of the 1982 ACM SIGMOD international conference on Management of data*. New York, NY, USA : ACM, 1982, S. 207–212

[Hauschild 2003] HAUSCHILD, Th.: *Computer Supported Cooperative Work-Applikationen in der Bauwerksplanung auf Basis einer integrierten Bauwerksmodellverwaltung*, Bauhaus-Universität Weimar, Dissertation, 2003. – http://e-pub.uni-weimar.de/frontdoor.php?source_opus=67&la=de

[Herczeg 2005] HERCZEG, M.: *Software-Ergonomie: Grundlagen der Mensch-Computer-Kommunikation*. 2. Auflage. München [u. a.] : R. Oldenbourg, 2005

[Honda u. Miller 1989] HONDA, M. ; MILLER, T.: Software Management Using a Case Environment. In: *Proceedings of USENIX Workshop on Software Management.* New Orleans, LA : USENIX Association, April 1989

[Härder u. Reuter 1983] HÄRDER, T. ; REUTER, A.: Principles of transaction-oriented database recovery. In: *ACM Computing Survey* 15 (1983), Nr. 4, S. 287–317

[Hunt u. McIllroy 1976] HUNT, J. W. ; MCILLROY, M. D.: *An Algorithm for Differential File Comparison.* Bell Laboratories, Juni 1976. – Computing Science Technical Report

[ISO 10303-21 2002] Norm ISO 10303-21 2002. *Industrial automation systems and integration – Product data representation and exchange – Part 21: Implementation methods: Clear text encoding of the exchange structure*

[ISO 10303-214 2005] Norm ISO 10303-214 November 2005. *Industrial automation systems and integration – Product data representation and exchange – Part 214: Core data for automotive mechanical design processes*

[ISO 19005-1 2008] Norm ISO 19005-1 Dezember 2008. *Document management – Electronic document file format for long-term preservation – Part 1: Use of PDF 1.4 (PDF/A-1)*

[ISO 32000-1 2008] Norm ISO 32000-1 Juli 2008. *Document management – Portable document format – Part 1: PDF 1.7*

[ISO 6385 2004] Norm DIN EN ISO 6385 Mai 2004. *Grundsätze der Ergonomie für die Gestaltung von Arbeitssystemen*

[ISO 9241-11 1999] Norm DIN EN ISO 9241 Teil 11 Januar 1999. *Anforderungen an die Gebrauchstauglichkeit - Leitsätze*

[ISO 9241-110 2008] Norm DIN EN ISO 9241 Teil 110 September 2008. *Grundsätze der Dialoggestaltung*

[ISO 9241-12 2000] Norm DIN EN ISO 9241 Teil 12 August 2000. *Informationsdarstellung*

[ISO/IEC 10646-4 2008] Norm ISO/IEC 10646-4 Februar 2008. *Information technology – Universal Multiple-Octet Coded Character Set (UCS)*

[ISO/IEC 11578 1996] Norm ISO/IEC 11578 1996. *Information technology – Open Systems Interconnection – Remote Procedure Call (RPC)*

[ISO/IEC 14977 1996] Norm ISO/IEC 14977 Dezember 1996. *Information technology – Syntactic metalanguage – Extended BNF*

[ISO/IEC 26300 2006] Norm ISO/IEC 26300 November 2006. *Information technology – Open Document Format for Office Applications (OpenDocument) v1.0*

[ISO/IEC 29500 2008] Norm ISO/IEC 29500 November 2008. *Information technology – Document description and processing languages – Office Open XML File Formats*

[ISO/IEC 7498-1 1994] Norm ISO/IEC 7498-1 1994. *Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*

[ISO/IEC 8859-1 1998] Norm ISO/IEC 8859-1 1998. *Information technology – 8-bit single-byte coded graphic character sets – Part 1: Latin alphabet No. 1*

[ISO/IEC 9834-8 2005] Norm ISO/IEC 9834-8 August 2005. *Information technology – Open Systems Interconnection – Procedures for the operation of OSI Registration Authorities: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components*

[ISO/PAS 16739 2005] Norm ISO/PAS 16739 2005. *Industry Foundation Classes, Release 2x, Platform Specification (IFC2x Platform)*

[Jennings 1994] JENNINGS, N. R.: *Cooperation in industrial multi-agent systems*. River Edge, NJ, USA : World Scientific Publishing Co., Inc., 1994

[JKomG 2005] *Gesetz über die Verwendung elektronischer Kommunikationsformen in der Justiz (Justizkommunikationsgesetz – JKomG)*. Bonn, März 2005. – Bundesgesetzblatt Jahrgang 2005 Teil I Nr. 18

[Jungwirth u. a. 1996] JUNGWIRTH, D. ; FUHR, H. ; GÖPEL, R.-A. ; OTTO, Th. ; JUNGWIRTH, D. (Hrsg.): *Qualitätsmanagement im Bauwesen*. 2. Auflage. Düsseldorf : VDI, 1996

[Kampffmeyer u. Rogalla 1997] KAMPFFMEYER, U. ; ROGALLA, J.: *Grundsätze der elektronischen Archivierung. „Code of Practice“ zum Einsatz von Dokumenten-Management- und elektronischen Archivsystemen*. 2. Auflage. Darmstadt : Verband Optischer Informationssysteme e.V., 1997

[Katz 1990] KATZ, R. H.: Toward a unified framework for version modeling in engineering databases. In: *ACM Computing Surveys* 22 (1990), Nr. 4, S. 375–409

[Katz u. a. 1986] KATZ, R. H. ; CHANG, E. ; BHATEJA, R.: Version modeling concepts for computer-aided design databases. In: *SIGMOD '86: Proceedings of the 1986 ACM SIGMOD International Conference on Management of data*. New York, NY, USA : ACM, 1986, S. 379–386

[Kemper u. Eickler 2006] KEMPER, A. ; EICKLER, A.: *Datenbanksysteme*. 6. Auflage. München : Oldenbourg, 2006

[Kiviniemi u. a. 2005] KIVINIEMI, A. ; FISHER, M. ; BAZJANAC, V.: Integration of Multiple Product Models: IFC Model Servers as a Potential Solution. In: SCHERER, R. J. (Hrsg.) ; KATRANUSCHKOV, P. (Hrsg.) ; SCHAPKE, S.-E. (Hrsg.): *Proceedings of 22nd CIB-W78 Conference on Information Technology in Construction*. Dresden : Institute of Construction Informatics, Juli 2005, S. 37–10

[Kiviniemi u. a. 2008] KIVINIEMI, A. ; TARANDI, V. ; KARLSHØJ, J. ; KARUD, O. J.: *Review of the Development and Implementation of IFC compatible BIM*. Erabuild, 2008. – Forschungsbericht – http://www.deaca.dk/publicationsconstruction/76491/1/0 – Online-Ressource, Abruf: 30.01.2009

[Koch 2008] KOCH, Ch.: *Bauwerksmodellierung im kooperativen Planungsprozess: Mit der Objektorientierung zur Verarbeitungsorientierung*, Bauhaus-Universität Weimar, Dissertation, 2008

[Kochendörfer u. a. 2004] KOCHENDÖRFER, B. ; VIERING, M. G. ; LIEBCHEN, J.: *Bau-Projekt-Management: Grundlagen und Vorgehensweisen*. 2. Auflage. Stuttgart : Teubner, 2004 (Leitfaden der Bauwirtschaft und des Baubetriebs)

[Krüger 2007] KRÜGER, G.: *Handbuch der Java-Programmierung*. 5. Auflage. Bonn [u. a.] : Addison-Wesley, 2007

[Laabs 1998] LAABS, A.: *Methoden für die Modellierung mit Objekten im Bauingenieurwesen*. Aachen : Shaker, 1998. – Dissertation

[Mackall 2006] MACKALL, M.: Towards a Better SCM: Revlog and Mercurial. In: HUTTON, A. J. (Hrsg.) ; ROSS, C. C. (Hrsg.): *Proceedings of the Linux symposium* Bd. 2. Ottawa : Linux Symposium Inc, Juli 2006, S. 93–90. – http://www.linuxsymposium.org/2006/proceedings.php

[Maymounkov u. Mazières 2002] MAYMOUNKOV, B. ; MAZIÈRES, D.: Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In: DRUSCHEL, P. (Hrsg.) ; ROWSTRON, A. (Hrsg.): *Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*. Cambridge, MA : International workshop on Peer-To-Peer Systems, März 2002

[Myers 1986] MYERS, E.: An O(ND) Difference Algorithm and Its Variations. In: *Algorithmica* 1 (1986), S. 251–266

[Neuman 1994] NEUMAN, B. C.: Scale in Distributed Systems. In: CASAVANT, T. (Hrsg.) ; SINGH, M. (Hrsg.): *Readings in Distributed Computing Systems*. Las Alamitos, Kanada : IEEE Computer Society Press, 1994, S. 463–489

[Norman 2001] NORMAN, D. A.: *The design of everyday things*. London [u. a.] : MIT Press, 2001

[Nour u. a. 2006] NOUR, M. M. ; FIRMENICH, B. ; RICHTER, T. ; KOCH, Ch.: A versioned IFC database for multi-disciplinary synchronous cooperation. In: *Proceedings of the Eleventh International Conference on Computing in Civil and Building Engineering (ICCCBE-XI)*. Montreal/Canada : Université du Québec, Juni 2006

[Olivier 2007] OLIVIER, A. H.: *Supporting consistency in linked specialized engineering modells through bindings and updating*, Stellenbosch University, Dissertation, 2007. – http://ir.sun.ac.za/dspace/bitstream/10019/697/1/Olivier,+A.H.pdf.pdf

[OMG 2002] Object Management Group: *CORBA 3.0 - IDL Syntax and Semantics chapter*. 2002. – http://www.omg.org/cgi-bin/doc?formal/02-06-07 – Online-Ressource, Abruf: 11.07.2008

[OMG 2004] Object Management Group: *CORBA 3.0.3 and IIOP Specification*. 2004. – http://www.omg.org/technology/documents/formal/corba_iiop.htm – Online-Ressource, Abruf: 11.07.2008

[Pahl u. Beucke 2000] PAHL, P. J. ; BEUCKE, K.: Neuere Konzepte des CAD im Bauwesen; Stand und Entwicklungen. In: *Digital Proceedings des Internationalen Kolloquiums über Anwendungen der Informatik und Mathematik in Architektur und Bauwesen (IKM)*, Bauhaus-Universität Weimar, 2000

[Pahl u. Damrath 2000] PAHL, P. J. ; DAMRATH, R.: *Mathematische Grundlagen der Ingenieurinformatik*. Berlin [u. a.] : Springer, 2000

[Ramunno 2005] RAMUNNO, E.: *Vergleich und Zusammenführung unterschiedlicher Planungsstände*, Bauhaus-Universität Weimar, Bachelorarbeit, 2005. – http://www.uni-weimar.de/cms/fileadmin/bauing/files/professuren/bauinf/Publikationen/BA/BARamunno.pdf

[Reenskaug 1979] REENSKAUG, T.: *THING-MODEL-VIEW-EDITOR an Example from a planningsystem*. Xerox PARC, Mai 1979. – Technischer Bericht – http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html – Online-Ressource, Abruf: 14.01.2009

[Revit 2005] Autodesk: *White Paper: Multi-user Collaboration with Autodesk Revit Worksharing*. November 2005. – http://images.autodesk.com/adsk/files/Multi_User_Collaboration_Revit_8-10.pdf – Online-Ressource, Abruf: 30.01.2009

[Richter 2003] RICHTER, T.: Ein Java-Paket zur Verarbeitung von Datenstrukturen in beliebigen Datenquellen. In: KAAPKE, K. (Hrsg.) ; WULF, A. (Hrsg.): *15. Forum Bauinformatik*. Aachen : Shaker, Oktober 2003 (Reihe Bauinformatik). – ISSN 1612–6262, S. 136–146

[Richter 2005] RICHTER, T.: Diff und Merge von Objekten im verteilten Planungsprozess. In: SCHLEY, F. (Hrsg.) ; WEBER, L. (Hrsg.): *Forum Bauinformatik 2005*. Cottbus : Verlag der BTU Cottbus, September 2005, S. 217–225

[Rochkind 1975] ROCHKIND, M. J.: The Source Code Control System. In: *In IEEE Transactions on Software Engineering Volume SE-1, Number 4*, 1975, S. 364–370

[Rüppel 2007] RÜPPEL, U.: Grundlegende Betrachtungen zur vernetzten Kooperation. In: RÜPPEL, Uwe (Hrsg.): *Abschlussbericht zum DFG-Schwerpunktprogramm „Vernetzt-kooperative Planungsprozesse im Konstruktiven Ingenieurbau“*. Heidelberg : Springer, 2007, S. 3–17

[RRZN 2001] Regionales Rechenzentrum für Niedersachsen (RRZN) an der Universität Hannover: *Netzwerke: Grundlagen*. Hannover, 2001

[SAGA 4.0 2008] Bundesministerium des Innern: *SAGA – Standards und Architekturen für E-Government-Anwendungen, Version 4.0*. März 2008. – http://gsb.download.bva.bund.de/KBSt/SAGA/SAGA_v4.0.pdf – Online-Ressource, Abruf: 30.01.2009

[Scheid 1994] SCHEID, H. ; ENGESSER, H. (Hrsg.): *Duden Rechnen und Mathematik*. 5. Auflage. Mannheim, Leipzig, Wien, Zürich : Dudenverlag, 1994

[Schäfer 2006] SCHÄFER, S.: *Vergleichen und Zusammenführen von objektorientierten Ingenieurmodellen*, Bauhaus-Universität Weimar, Diplomarbeit, 2006

[Schindelhauer 2006] SCHINDELHAUER, Ch.: *Skript „Peer-to-Peer-Netzwerke“*. Freiburg : Universität Freiburg, 2006

[Smolka 1992] SMOLKA, Gert: Feature-Constraint Logics for Unification Grammars. In: *Journal of Logic Programming* 12 (1992), Nr. 1&2, S. 51–87. – http://citeseer.ist.psu.edu/smolka92feature.html

[SVNKit 2008] TMate Software: *SVNKit – The pure Java Subversion Library*. 2008. – http://svnkit.com/

[Tanenbaum u. van Steen 2003] TANENBAUM, Andrew ; STEEN, Marten van: *Verteilte Systeme: Grundlagen und Paradigmen*. München : Pearson Studium, 2003

[Tichy 1982] TICHY, W.: Design, Implementation, and Evaluation of a Revision Control System. In: *Proceedings of the 6th International Conference on Software Engineering*. Los Alamitos, CA : IEEE Computer Society Press, September 1982, S. 58–67

[Tulke u. a. 2008] TULKE, Jan ; NOUR, M. M. ; BEUCKE, K.: A Dynamic Framework for Construction Scheduling based on BIM using IFC. In: *Digital Proceedings of the 17th Congress of IABSE*. Chicago : ETH Zürich, September 2008

[Ullenboom 2008] ULLENBOOM, Ch.: *Java ist auch eine Insel, Programmieren mit der Java Standard Edition Version 6*. 7. Auflage. Bonn : Galileo Computing, 2008

[Unicode Consortium 2008] Unicode Consortium: *Unicode 5.1.0*. 2008. – http://www.unicode.org/versions/Unicode5.1.0/ – Online-Ressource, Abruf: 04.10.2008

[VDA 2006] Verband der Automobilindustrie (VDA): *Austausch von Daten in der Fabrikplanung – Austausch von Daten mit STEP-CDS*. Mai 2006. – http://www.vda.de/de/publikationen/publikationen_downloads/detail.php?id=340 – Online-Ressource, Abruf: 30.01.2009

[W3C 2003] World Wide Web Consortium (W3C): *Scalable Vector Graphics (SVG) 1.1 Specification*. Januar 2003. – http://www.w3.org/TR/2003/REC-SVG11-20030114/ – Online-Ressource, Abruf: 04.10.2008

[W3C 2004] World Wide Web Consortium (W3C): *XML Schema (Second Edition). Part 0: Primer, Part 1: Structures, Part 2: Datatypes*. 2004. – http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/, http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/, http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/ – Online-Ressource, Abruf: 04.10.2008

[W3C 2006] World Wide Web Consortium (W3C): *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. 2006. – http://www.w3.org/TR/2006/REC-xml-20060816/ – Online-Ressource, Abruf: 04.10.2008

[Weise 2006] WEISE, M.: *Ein Ansatz zur Abbildung von Änderungen in der modellbasierten Objektplanung*. Technische Universität Dresden, 2006 (Schriftenreihe des Instituts für Bauinformatik). – Dissertation

[Wikipedia 2008a] Wikipedia: *Abwärtskompatibilität*. 2008. – http://de.wikipedia.org/w/index.php?title=Abw%C3%A4rtskompatibilit%C3%A4t&stableid=44881907 – Online-Ressource, Artikelversion vom 14.04.2008, 10:24 Uhr

[Wikipedia 2008b] Wikipedia: *CRC-32*. 2008. – http://de.wikipedia.org/w/index.php?title=Zyklische_Redundanzpr%C3%BCfung&oldid=51887787 – Online-Ressource, Artikelversion vom 16.10.2008, 11:33 Uhr

[Wikipedia 2008c] Wikipedia: *.dwg*. 2008. – http://en.wikipedia.org/w/index.php?title=.dwg&oldid=241511524 – Online-Ressource, Artikelversion vom 28.09.2008, 11:11 Uhr

[Willenbacher 2002] WILLENBACHER, H.: *Interaktive verknüpfungsbasierte Bauwerksmodellierung als Integrationsplattform für den Bauwerkslebenszyklus*, Bauhaus-Universität Weimar, Dissertation, 2002. – http://e-pub.uni-weimar.de/frontdoor.php?source_opus=32&la=de

[Zeller 1997] ZELLER, A.: *Configuration Management with Version Sets - A Unified Software Versioning Model and its Applications*, Technische Universität Braunschweig, Dissertation, 1997. http://www.infosun.fim.uni-passau.de/st/papers/zeller-phd/

[Zeller u. Snelting 1997] ZELLER, A. ; SNELTING, G.: Unified Versioning through Feature Logic. In: *Transactions on Software Engineering and Methodology (TOSEM)* 6 (1997), Oktober, Nr. 4, S. 398–441

[ZIP 2007] *.ZIP File Format Specification*. 2007. – Version 6.3.2 vom 28. September 2007 http://www.pkware.com/documents/casestudies/APPNOTE.TXT – Online-Ressource, Abruf: 30.10.2008

[ZPO 2008] Zivilprozessordnung in der Fassung der Bekanntmachung vom 5. Dezember 2005 (BGBl. I S. 3202; 2006 I S. 431; 2007 I S. 1781), zuletzt geändert durch Artikel 2 G. vom 26. März 2008 (BGBl. I S. 441) – http://www.gesetze-im-internet.de/bundesrecht/zpo/gesamt.pdf – Online-Ressource, Abruf: 14.08.2008

Verzeichnis der Beispiele

Verzeichnis der Listings

A Abkürzungen

ACID Atomicity, Consistency, Isolation, Durability ⟨engl., „Atomarität, Konsistenz, Isoliertheit, Dauerhaftigkeit“⟩

ADT Architectural Desktop. Ein Produkt des Herstellers Autodesk.

API Application Programming Interface ⟨engl., „Schnittstelle zur Anwendungsprogrammierung“⟩

ASCII American Standard Code for Information Interchange

AWT Abstract Window Toolkit

BIM Building Information Modeling

BNF Backus-Naur-Form

B-Rep Boundary Representation

CAD Computer Aided Design ⟨engl., „rechnerunterstützte Konstruktion“⟩

CBR Call-by-Reference ⟨engl., dt. Entsprechung: „Referenzparameter“⟩

CBV Call-by-Value ⟨engl., dt. Entsprechung: „Wertparameter“⟩

CDS Construction Drawing Subset. Eigentlich STEP-CDS.

CERN Conseil Européen pour la Recherche Nucléaire ⟨franz., „Europäische Organisation für Kernforschung“⟩

CLI Command Line Interface ⟨engl., „Kommandozeile, Befehlszeile“⟩

COM Component Object Model ⟨engl., „Komponentenobjektmodell“⟩

CORBA Common Object Request Broker Architecture

CPU Central Processing Unit ⟨engl., „Hauptprozessor“, *auch:* „zentrale Verarbeitungseinheit“⟩

CR carriage return ⟨engl., „Wagenrücklauf“⟩

CRC Cyclic Redundancy Check ⟨engl., „Zyklische Redundanzprüfung“⟩

CSG Constructive Solid Geometry ⟨engl., „Konstruktive Festkörpergeometrie“⟩

CVS Concurrent Versions System

DCOM Distributed Component Object Model ⟨engl., „Verteiltes Komponentenobjektmodell“⟩

DSL Digital Subscriber Line ⟨engl., „Digitaler Teilnehmeranschluss“⟩

DFG Deutsche Forschungsgemeinschaft

DIN DIN Deutsches Institut für Normung e.V.

DMS Dokumenten-Management-System

DTD Document Type Definition ⟨engl., „Dokumenttypdefinition“⟩

EBNF Erweiterte Backus-Naur-Form

ECM Enterprise-Content-Management

FEM Finite-Elemente-Methode

FLOPS Floating Point Operations Per Second ⟨engl., „Gleitkommaoperationen pro Sekunde“⟩

GAEB Gemeinsamer Ausschuss Elektronik im Bauwesen

GPL GNU General Public License

GUI Graphical User Interface ⟨engl., „grafische Benutzeroberfläche“⟩

GUID Globally Unique Identifier ⟨engl., „global eindeutiger Identifikator“⟩

HTTP Hypertext Transfer Protocol ⟨engl., „Hypertext-Übertragungsprotokoll“⟩

HCI Human-Computer Interaction

HKLS Heizung-Klima-Lüftung-Sanitär

HPGL Hewlett Packard Graphic Language

IAI Industrieallianz für Interoperabilität e.V.

IANA Internet Assigned Numbers Authority

ID identifier ⟨engl., „Kennung, Identifikator“⟩

IDE Integrated Development Environment

IDL Interface Description Language ⟨engl., „Schnittstellenbeschreibungssprache“, *auch:* „Schnittstellendefinitionssprache“⟩

IEEE Institute of Electrical and Electronics Engineers

IFC Industry Foundation Classes

IP Internet Protocol

ISO International Organization for Standardization ⟨engl., „Internationale Organisation für Normung“⟩

ITU International Telecommunication Union ⟨engl., „Internationale Fernmeldeunion“⟩

JAR Java Archive

JAXP Java API for XML Processing

JDBC Java Database Connectivity

JDK Java Development Kit

JRMP Java Remote Method Protocol

JVM Java Virtual Machine

KI Künstliche Intelligenz

KM Konfigurationsmanagement

LZW Lempel-Ziv-Welch-Algorithmus

MMS Mensch-Maschine-Schnittstelle

MAC Media Access Control ⟨engl., „Medienzugriffskontrolle“⟩

MAS Multiagentensystem

MDI Multiple Document Interface ⟨engl.⟩ (eine Form der GUI für Programme)

MD5 Message-Digest Algorithm 5 ⟨engl.⟩ (ein Hash-Algorithmus)

MVC Model-View-Controller ⟨engl., „Modell/Präsentation/Steuerung“⟩

ODA Open Design Alliance

ODF OASIS Open Document Format for Office Applications, Kurzform: OpenDocument

OMF Object Modeling Framework

OMG Object Management Group

ORB Object Request Broker ⟨engl., „Vermittler für Objektanfragen“⟩

OUI Organizationally Unique Identifier ⟨engl., „organisatorisch eindeutiger Identifikator“⟩

PAS Publicly Available Specification ⟨engl., „öffentlich verfügbare Spezifikation“⟩

PDA Personal Digital Assistant ⟨engl., „persönlicher digitaler Assistent“⟩

PDF Portable Document Format ⟨engl., „(trans)portables Dokumentenformat“⟩

PID Persistenter Identifikator

PKI Public-Key-Infrastruktur

POID Persistenter Objektidentifikator

POVID Persistenter Objektversionsidentifikator

PVID Persistenter Versionsidentifikator

RAID Redundant Array of Independent Disks ⟨engl., „Redundante Anordnung unabhängiger Festplatten“⟩

RAM Random Access Memory ⟨engl., „Speicher mit wahlfreiem Zugriff“⟩

RCS Revision Control System

RDB Relationale Datenbank

RDBMS Relationales Datenbankmanagementsystem

RMI Remote Method Invocation ⟨engl., „entfernter Methodenaufruf“⟩

RPC Remote Procedure Call ⟨engl., „entfernter Prozeduraufruf“⟩

SCCS Source Code Control System

SCM Software Configuration Management ⟨engl., „Softwarekonfigurationsmanagement“⟩

SDI Single Document Interface ⟨engl.⟩ (eine Form der GUI für Programme)

SEM Structural Engineering Model

SHA Secure Hash Algorithm ⟨engl., „sicherer Hash-Algorithmus“⟩

SOAP Simple Object Access Protocol

SQL Structured Query Language ⟨engl., „strukturierte Anfragesprache“⟩

SSH Secure Shell

SSL Secure Sockets Layer

STEP Standard for the Exchange of Product model data

SVG Scalable Vector Graphics ⟨engl., „Skalierbare Vektorgrafiken“⟩

TCP Transmission Control Protocol ⟨engl., „Übertragungssteuerungsprotokoll“⟩

TGA Technische Gebäudeausrüstung

TUI Text User Interface

URL Uniform Resource Locator ⟨engl., „einheitlicher Ressourcen-Positionsanzeiger“⟩

UCS Universal Character Set ⟨engl., „universeller Zeichensatz“⟩

UI User Interface ⟨engl., „Benutzerschnittstelle“, *auch:* „Benutzungsschnittstelle“⟩

UDP User Datagram Protocol

USV Unterbrechungsfreie Stromversorgung

UTF Unicode Transformation Format

UUID Universally Unique Identifier ⟨engl., „allgemein eindeutiger Identifikator“⟩

VCS Version Control System ⟨engl., „Versionsverwaltungssystem“, *auch:* „Versionskontrollsystem“⟩

VDA Verband der Automobilindustrie

VM Virtuelle Maschine

VOI Verband Organisations- und Informationssysteme e. V.

VPN Virtual Private Network

VUI Voice User Interface

W3C World Wide Web Consortium

WebDAV Web-based Distributed Authoring and Versioning

WSDL Web Services Description Language

WWW World Wide Web

XML Extensible Markup Language ⟨engl., „erweiterbare Auszeichnungssprache“⟩

XMP Extensible Metadata Platform

B Verteilte Systeme

B.1 Allgemein

Netzwerk: In (RRZN, 2001) ist ein Netzwerk definiert als eine *„Gruppe miteinander verbundener Systeme, die in der Lage sind, untereinander zu kommunizieren.“*. Zwei Rechner, verbunden durch ein Kabel, ist die kleinste Variante eines Netzwerks. Die zur Zeit größte Variante ist das weltumspannende Internet. Bezüglich der Größe lassen sich Netzwerke in die Kategorien der Tabelle B.1 einteilen.

Abkürzung	Bezeichung	Größe
PAN	Personal Area Network	Netzwerk, das von Kleingeräten, wie Mobiltelefone oder PDAs mittels kabelgebundener oder kabelloser Technik auf- und abgebaut wird.
LAN	Local Area Network	Netzwerk, das von einem Nutzer oder einer Organisation verwaltet wird und geografisch auf das zugehörige Gelände begrenzt ist.
WLAN	Wireless LAN	Unterscheidet sich vom LAN nur dadurch, dass für die Verbindung anstatt Kabel Funktechnologie eingesetzt wird.
MAN	Metropolitan Area Network	Netzwerk, das sich auf das Gebiet einer Stadt oder einer Region bis zu einem Umkreis von 100 km ausdehnt.
WAN	Wide Area Network	Weitverkehrsnetzwerk, das sich geografisch unbegrenzt ausdehnen kann.
GAN	Global Area Network	Weiterer Begriff für ein WAN, um die weltweite Ausdehnung zu betonen.

Tabelle B.1: Einteilung von Netzwerken nach der Größe

Verteiltes System (Tanenbaum u. van Steen, 2003)**:** *„Ein verteiltes System ist eine Menge voneinander unabhängiger Computer, die dem Benutzer wie ein einzelnes kohärentes System erscheinen.“*

Ziele eines verteilten Systems:

- **Verbindung von Benutzern und Ressourcen:**
  - Der Zugriff auf entfernte Ressourcen wird erlaubt, um diese Ressourcen gemeinsam mit anderen Nutzern, z. B. aus Gründen der Kosteneinsparung, zu nutzen.
  - Förderung von Zusammenarbeit und Informationsaustausch.
  - Die Sicherheit und der Schutz der Privatsphäre muss gewährleistet sein.
- **Transparenz:** Für den Nutzer sollen sich die physisch auf mehrere Rechner verteilten Prozesse und Ressourcen als ein System darstellen. Es ist z. B. unwesentlich, wie und wo auf eine Ressource zugegriffen wird und wie viele andere Nutzer eine Ressource gleichzeitig nutzen.
- **Offenheit:** Ein verteiltes System sollte Standardprotokolle und -schnittstellen für Dienste zum Nachrichtenaustausch sowie austauschbare Komponenten verwenden, um Flexibilität zu schaffen.
- **Skalierbarkeit:** Nach (Neuman, 1994) ist ein System skalierbar, wenn es die Erweiterung um Nutzer und Ressourcen ohne wesentliche Performanceeinbußen oder erhöhte Verwaltungskomplexität handhaben kann. Er unterscheidet drei Dimensionen der Skalierbarkeit:
  - Numerisch:
    Die numerische Dimension umfasst die Anzahl der Nutzer und Ressourcen, wie Objekte und Dienste.
  - Geografisch:
    Die geografische Dimension zeigt, inwieweit das System geografisch verteilt ist.
  - Administrativ:
    Die administrative Dimension ist abhängig von der Anzahl der Organisationen, die Einfluss auf Teile des Systems haben.

B.2 Hardwarekonzepte

(Tanenbaum u. van Steen, 2003) unterscheiden die Hardwarekonzepte verteilter Systeme hinsichtlich der Hardwareanordnung und -verbindung. Es wird vorausgesetzt, dass die Systeme aus unabhängigen Computern aufgebaut sind.

- **Multiprozessor:** Die CPUs[1] teilen sich einen gemeinsamen Speicher. Änderungen im Speicher haben Einfluss auf alle Prozessoren.
- **Multicomputer:** Im Gegensatz zum Multiprozessor besitzt jeder Rechner einen privaten Arbeitsspeicher.

[1] CPU = Central Processing Unit ⟨engl., „Hauptprozessor“, *auch:* „zentrale Verarbeitungseinheit“⟩

– **Homogene Multicomputersysteme:** Sie bestehen im Allgemeinen aus Rechnern gleicher Bauart und Leistungsklasse. Zwischen ihnen existiert ein Netzwerk, das überall die gleiche Technologie aufweist.

– **Heterogene Multicomputersysteme:** Sie können sich aus verschiedenen Rechnertypen und Netzwerken zusammensetzen.

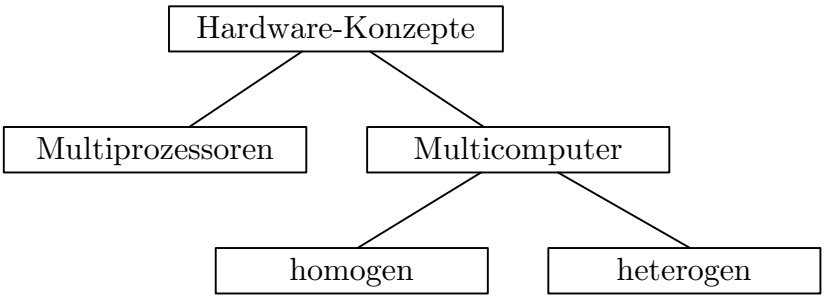


Abbildung B.1: Hardwarekonzepte von verteilten Systemen

B.3 Softwarekonzepte

Die Hardware allein ergibt noch kein funktionierendes verteiltes System, erst die Software ist entscheidend für die Funktionsweise. (Tanenbaum u. van Steen, 2003) teilen die Softwarekonzepte in drei Arten ein:

System	Beschreibung	Wichtigstes Ziel
Verteiltes Betriebssystem	Streng gekoppeltes Betriebssystem für Multiprozessoren und homogene Multicomputer	Hardware-Ressourcen verwalten
Netzwerkbetriebssystem	Locker gekoppeltes Betriebssystem für heterogene Multicomputer	Anbieten lokaler Dienste für entfernte Clients
Middleware	Zusätzliche Schicht über dem Netzwerkbetriebssystem, die allgemeine Dienste implementiert.	Verteilungstransparenz erzielen

Tabelle B.2: Softwarekonzepte verteilter Systeme

Haupzeinsatzzweck verteilter Betriebssysteme sind Computercluster, die nach außen wie ein Rechner erscheinen und die Rechenkapazität wesentlich erhöhen. Netzwerkbetriebssysteme dienen vorrangig der Bereitstellung von Speicherplatz und Druckdienstleistungen in einem Netzwerk unter Beachtung von Zugriffsberechtigungen der einzelnen Nutzer. Ein bekannter Vertreter ist Netware der Firma Novell (Harris, 2005). Middleware soll

die Vorteile der beiden anderen Konzepte vereinigen: Offenheit und Skalierbarkeit der Netzwerkbetriebssysteme sowie die Transparenz der verteilte Betriebssysteme. Verteilte Anwendungen verwenden Netzwerkdienste des Betriebssystems für ihre Bestimmung. Middleware ist eine zusätzliche Softwareschicht zwischen beiden, um die oben genannten Vorteile zu verwirklichen. Die verteilten Anwendungen sind nicht mehr auf die möglicherweise unterschiedlich implementierten Dienste des Betriebssystems angewiesen, sondern greifen über standardisierte Schnittstellen auf die Middleware zu. Abbildung B.2 zeigt die allgemeine Struktur einer Middleware nach (Tanenbaum u. van Steen, 2003).

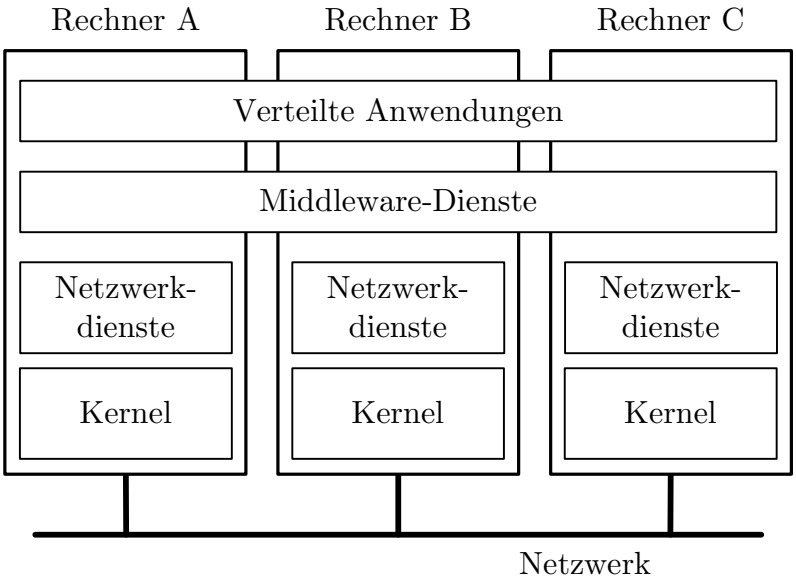


Abbildung B.2: Allgemeine Struktur eines verteilten Systems als Middleware

Beispiele für Middleware-Modelle sind die entfernten Prozeduraufrufe (RPC[2]) und die mit steigender Akzeptanz der Objektorientierung eingeführten entfernten Objekte.

Client-Server-Modell (Tanenbaum u. van Steen, 2003)**:** Das Client-Server-Modell ist ein Modell zur Organisation von verteilten Systemen, dass auch die Prozesse betrachtet.

- **Prozess:** Ein Prozess ist in der Informatik ein Programm, das gestartet wurde und noch ausgeführt wird. Im Betriebssystem Windows der Firma Microsoft heißen die Prozesse Tasks.
- **Server:** *„Ein Server ist ein Prozess, der einen bestimmten Dienst implementiert und zur Verfügung stellt.“*. Typische Beispiel für Server sind Authentifizierungs-, Datenbank-, Druck-, Datei- und Webserver. Im Sprachgebrauch wird mit Server oft die Einheit von Server-Hardware und Server-Software verstanden.
- **Client:** *„Ein Client ist ein Prozess, der einen Dienst von einem Server anfordert, indem er eine Anforderung sendet und dann auf die Antwort des Servers wartet.“*.

[2] RPC = Remote Procedure Call

Ein Server läuft normalerweise im Leerlauf und wartet auf Anforderungen von Clients. Falls eine Anforderung eintrifft, wird diese vom Server bearbeitet und das Ergebnis an den Client zurückgeschickt. Danach kehrt er in den Leerlauf zurück.

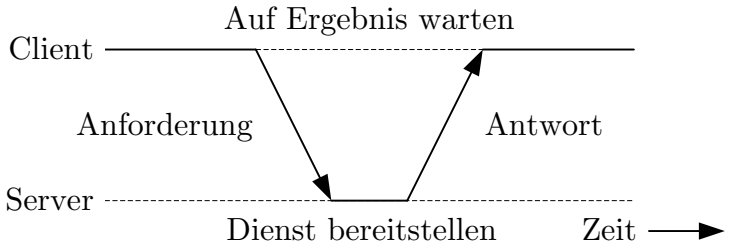


Abbildung B.3: Allgemeine Zusammenarbeit zwischen Client und Server

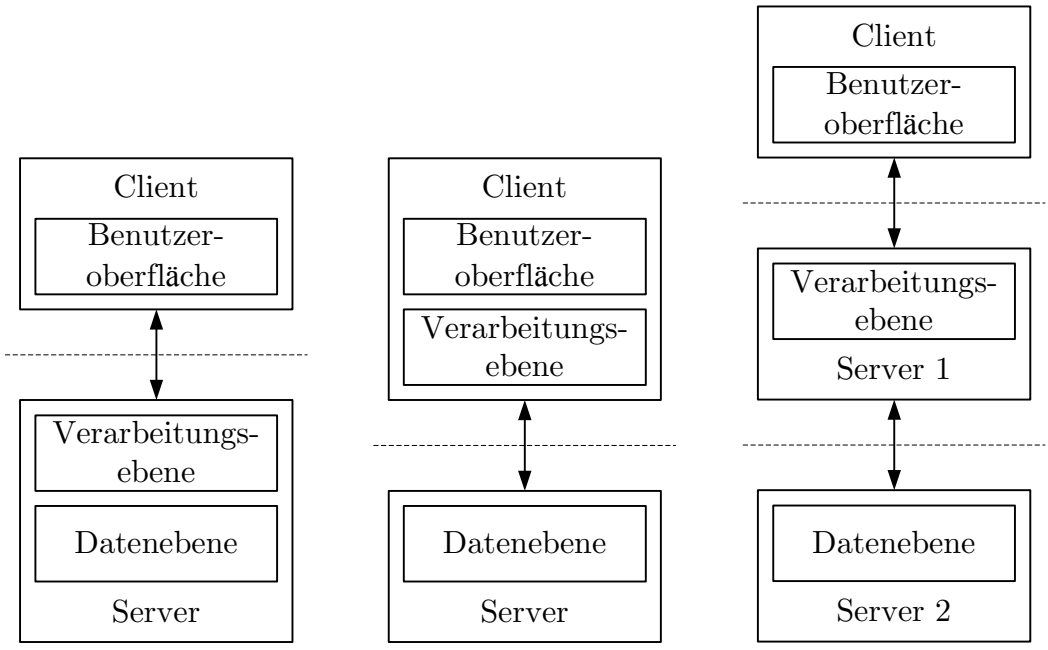
Anwendungsschichten: Die Einteilung eines Softwaresystems oder verteilten Systems in Schichten führt zu einer Verringerung der Komplexität und einem besseren Verständnis und erleichterter Wartbarkeit. Abhängigkeiten bestehen nur zwischen zwei benachbarten Schichten, wobei Aufrufe immer von einer Schicht zur nächsttieferen weitergeleitet werden.

- **Benutzeroberfläche:** Diese Schicht ist in der Regel auf Seiten des Clients implementiert und ist für die Ein- und Ausgabe von Daten verantwortlich. Die Benutzeroberfläche könnte eine textbasierte Konsole oder eine grafische Benutzeroberfläche mit einer Fensterverwaltung sein.
- **Datenebene:** Die Datenebene dient zur persistenten und anwendungsunabhängigen Speicherung der Daten entweder in einem Dateisystem oder in einer Datenbank. Ziel ist die Sicherstellung von Konsistenz und Datenunabhängigkeit.
- **Verarbeitungsebene:** Die Verarbeitungsebene ist zwischen den beiden anderen Schichten angeordnet und bildet den Kern der Anwendung. Die Daten werden hier verarbeitet.

Enthält der Client im Programm nur die Benutzeroberfläche, entspricht er nur einem Terminal und wird als Thin-Client bezeichnet. Die gesamte Programmlogik ist auf die Serverseite verschoben. In diesem Fall, kann man kaum von einem verteilten System sprechen. Ein Fat-Client vereint sowohl die Benutzeroberfläche als auch die Verarbeitungsebene. Diese Form ist heutzutage in der Regel anzutreffen, da die Workstations für geringe Anschaffungskosten eine sehr gute Performance bereitstellen. Beide Client-Arten bilden mit dem Server, wie in den Abbildungen B.4a und B.4b dargestellt, eine zweischichtige[3] Architektur. Die Verarbeitungsebene kann zusätzlich noch auf einen weiteren Rechner ausgelagert werden, so dass drei Schichten (three-tier) zur Verfügung stehen. Server 1 in Abbildung B.4c agiert gleichzeitig als Client und Server.

Eine mehrschichtige Architektur entspricht einer vertikalen Verteilung innerhalb des Client-Server-Modells. Genauso können bei steigenden Anforderungen an Leistung, Verfügbarkeit und Skalierbarkeit die Serverschichten horizontal verteilt werden. Nach außen tritt

[3] zweischichtig, auf engl. *two-tier*



(a) Zweischichtig, Thin-Client

(b) Zweischichtig, Fat-Client

(c) Dreischichtig

Abbildung B.4: Schichtenarchitekturen

die Schicht als ein logisches System auf, besteht aber intern aus mehreren Maschinen und einer Software zur Lastverteilung.

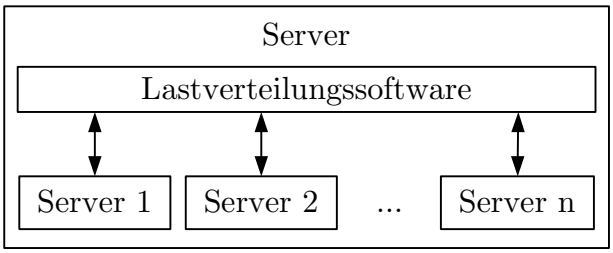


Abbildung B.5: Horizontale Verteilung im Client-Server-Modell

Peer-To-Peer-Modell: Beim Peer-To-Peer-Modell (P2P) existieren keine Server, dafür übernimmt jeder Client Serverfunktionen und ist autonom. Da jeder Client dem anderen gegenüber gleichgestellt ist, werden die Clients als *Peer*[4] bezeichnet. Das Peer-To-Peer-Netzwerk nutzt als Overlay-Netzwerk das vorhandene Netzwerk als Infrastruktur und bildet eine eigene Topologie, insbesondere zum Auffinden anderer Peers, ab. Peers können sich stark hinsichtlich Bandbreite, Rechenleistung und Verfügbarkeit voneinander unterscheiden.

(Schindelhauer, 2006) teilt die Peer-To-Peer-Systeme in vier Typen ein:

[4]peer ⟨engl., „Gleichgestellter“⟩

- **Zentralisierte P2P-Systeme:** Sie benötigen einen zentralen Server, der eine Liste mit angemeldeten Clients und eine Liste mit verfügbaren Ressourcen der teilnehmenden Clients führt (s. Abbildung B.6a). Der Peer muss zuerst den Server nach der Adresse einer gewünschten Ressource fragen und kann sich dann mit dem anderen Peer direkt verbinden. Vorteile sind eine effiziente Suche nach Ressourcen sowie die einfache Struktur; nachteilig wirkt sich die schlechte Skalierbarkeit und die Abhängigkeit von der Serververfügbarkeit aus.

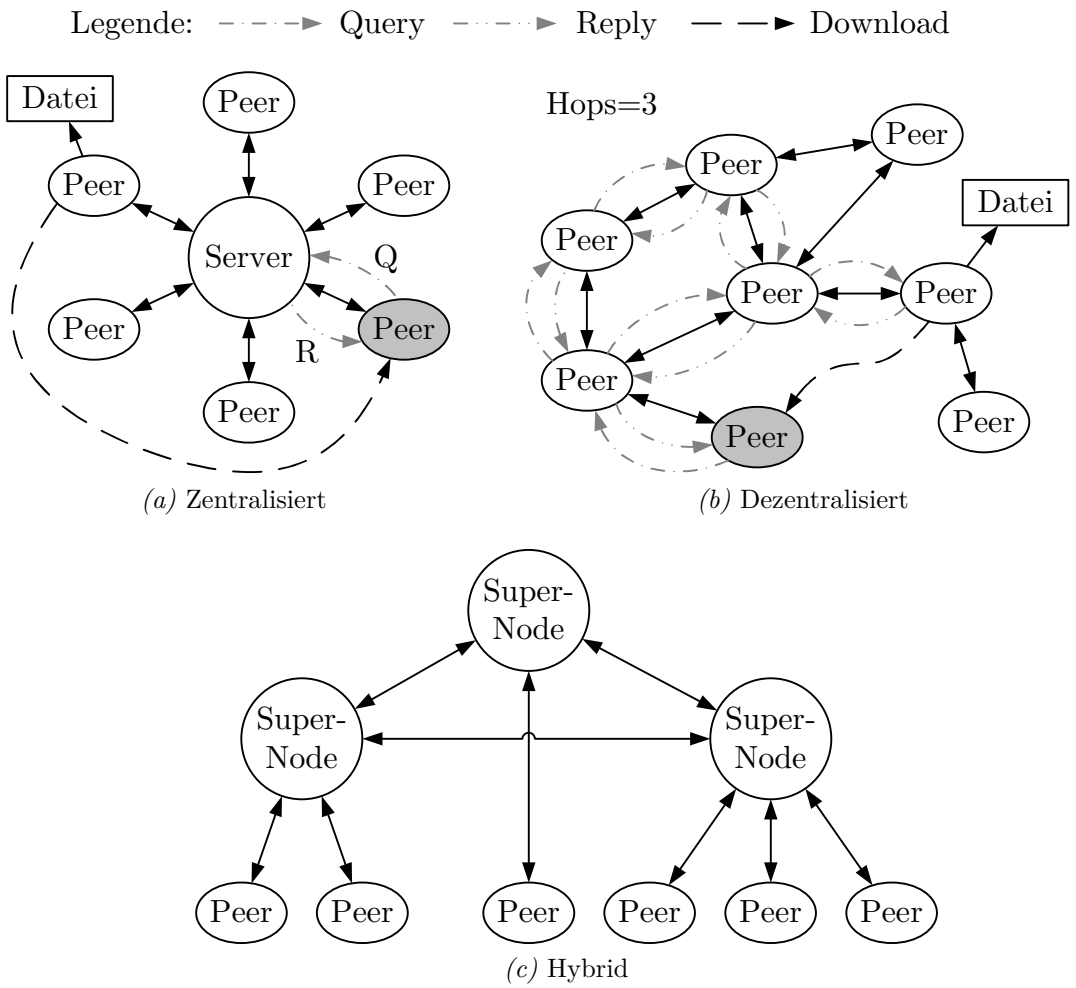
- **Dezentralisierte P2P-Systeme (Erste Generation):** Bei dezentralisierten P2P-Systemen der ersten Generation schickt ein Peer eine Ressourcenanfrage an seine Nachbarn und diese wiederum an ihre Nachbarn. Die maximale Anzahl von Sprüngen (Hops) wird vorher festgelegt. Falls die Ressource gefunden wurde, sendet sie der besitzende Peer direkt zum nachfragenden Peer. In Abbildung B.6b ist dieser Vorgang mit einer maximalen Sprunganzahl von 3 aufgezeigt. Vorteilhaft ist, dass kein zentraler Server mehr benötigt wird. Als Nachteile bestehen die Nichterreichbarkeit aller Knoten, der hohe Aufwand für das Finden von Ressourcen sowie die langsame Suchgeschwindigkeit.

- **Hybride P2P-Systeme:** Diese System kombinieren den zentralisierten und den dezentralisierten Ansatz, indem Peers mit großer Bandbreite zu P2P-Servern (Super-Node) ausgewählt werden und Informationen zu anderen Peers und Ressourcen speichern. Die Reaktionszeiten und die Skalierbarkeit verbessern sich zwar, jedoch sind diese Systeme trotzdem unzuverlässig und noch zu langsam.

- **Dezentralisierte P2P-Systeme (Zweite Generation):** Aktuelle P2P-Systeme verwenden verteilte Hashtabellen, um das Problem des Auffindens von Ressourcen zu lösen. Jedem Knoten und jeder Ressource wird mit einer Hash-Funktion eine ID[5] zugewiesen und danach in eine Hashtabelle eingetragen. Jeder Knoten speichert einen Teil der Hashtabelle, wobei auch gewünschte Redundanzen auftreten. Bei einer Anfrage findet ein entsprechender Algorithmus auf effiziente Art und Weise die betreffenden Peers (Maymounkov u. Mazières, 2002). Vorteilhafterweise ist die Skalierung logarithmisch – bei Verdoppelung der Knotenanzahl ist bei der Suchanfrage nur ein Knoten mehr zu befragen – und das System ist unabhängig von zentralen Servern.

Software-Agent: Software-Agenten wurden aus dem Forschungsgebiet der Künstlichen Intelligenz (KI) heraus entwickelt. Die Definition eines Software-Agenten ist je nach Fachgebiet und Sichtweise unterschiedlich (Hartmann, 2007). Allgemein gesagt ist ein Sofware-Agent ein Prozess bzw. Computerprogramm, das zu selbstständigem Handeln in der Lage ist. Ein Sofware-Agent besitzt mindestens folgende Eigenschaften: autonom, reaktiv, proaktiv, kommunikativ, lernfähig. Mobile Agenten können zusätzlich den Ort wechseln, was als Migration[6] bezeichnet wird. Je nach Aufgabe stehen Informations-, Schnittstellen- oder Kollaborationsagenten zur Verfügung. Arbeiten mehrere Agenten in einem Verbund zusammen, so spricht man von einem Multiagentensystem (MAS) (Jennings, 1994).

[5] ID = identifier ⟨engl., „Kennung, Identifikator“⟩
[6] migratio ⟨lat., „Wanderung“⟩



(a) Zentralisiert

(b) Dezentralisiert

(c) Hybrid

Abbildung B.6: Peer-To-Peer-Architekturen

B.4 Kommunikation in verteilten Systemen

Kommunikation (Tanenbaum u. van Steen, 2003)**:** Kommunikation ist die Voraussetzung für ein funktionierendes verteiltes System und *„basiert immer auf Nachrichtenübergabe auf unterster Ebene, die das zugrundeliegende Netzwerk bereitstellt“*. Für eine erfolgreiche Kommunikation ist es unabdingbar, vorher Regeln festzulegen.

OSI-Schichtenmodell (Tanenbaum u. van Steen, 2003; RRZN, 2001)**:** Die ISO hat zur Vereinfachung der komplexen Kommunikation zwischen offenen Systemen im Netzwerk ein Schichtenmodell entworfen und als OSI-Modell[7] bezeichnet (ISO/IEC 7498-1, 1994). Es dient als Vorbild für die Entwicklung von Kommunikationsprotokollen. Ein Protokoll ist eine Zusammenfassung von Regeln für die Form, den Inhalt und die Bedeutung der ausgetauschten Nachrichten. Ein offenes System ist darauf ausgerichtet, über diese Protokolle mit anderen offenen Systemen zu kommunizieren.

[7] Open Systems Interconnection Reference Model

Das OSI-Modell definiert sieben Schichten, wie in Tabelle B.3 ersichtlich ist. Mehrere Schichten bilden einen sogenannten Protokollstapel (Stack).

Schicht	Englisch	Deutsch
7	Application layer	Anwendungsschicht
6	Presentation layer	Darstellungs-, Präsentationsschicht
5	Session layer	Kommunikationssteuerungs-, Sitzungsschicht
4	Transport layer	Transportschicht
3	Network layer	Vermittlungs-, Netzwerksschicht
2	Data link layer	Sicherungs-, Datenverbindungsschicht
1	Physical layer	Bitübertragungsschicht
	Übertragungsmedium	

Tabelle B.3: Schichten des OSI-Modells

Die Schichten übernehmen bei der Kommunikation verschiedene Aufgaben:

- **Schicht 1** dient der Übertragung der Signale, also der einzelnen Bits. Festgelegt werden elektrische, mechanische und Signalschnittstellen.
- **Schicht 2** stellt sicher, dass die Bits korrekt übertragen werden. Der Bitstrom wird in Blöcke aufgeteilt und nach dem Empfang anhand der Prüfsummen kontrolliert. Fehlerhafte und verloren gegangene Blöcke können vom Empfänger erneut angefordert werden.
- **Schicht 3** ist für das Routing, also das Weiterleiten, von Datenpaketen zwischen den Netzknoten verantwortlich. Am weitesten verbreitet ist das verbindungslose *Internet Protocol* (IP).
- **Schicht 4** ist notwendig für eine zuverlässige Verbindung. Die Transportprotokolle sorgen dafür, dass der Datenstrom in Datenpakete aufgeteilt und verschickt wird. Die Datenpakete können verschiedene Wege zum Empfänger nehmen und dadurch in einer anderen Reihenfolge dort eintreffen. Das Transportprotokoll hat dafür Sorge zu tragen, dass die Datenpakete wieder korrekt zusammengefügt werden und nach außen einen kontinuierlichen Datenstrom vorzuspiegeln. Wichtige Vertreter sind das verbindungsorientierte *Transmission Control Protocol* (TCP) und das nicht-zuverlässige verbindungslose *User Datagram Protocol* (UDP), welches das IP nur um einige Eigenschaften erweitert. Die Schichten 1 bis 4 bilden zusammen den Netzwerkprotokollstapel und bilden die Grundlage für die Programmierung von Anwendungen, die über ein Netzwerk kommunizieren.
- **Schicht 5** steuert Sitzungen (sessions) und kümmert sich um den Aufbau, die Verwendung und den Abbau von Verbindungen. Bei Verbindungsabbrüchen wird der Datenaustausch synchronisiert und fehlende Daten seit einem Fixpunkt noch einmal gesendet.

- **Schicht 6:** Jedes System verwendet eine interne Darstellung der Daten, die zu Fremdsystemen inkompatibel sein kann. Die Darstellungsschicht wandelt die Daten vor dem Senden in ein Standardformat um. Die Datenkompression und -verschlüsselung gehören zu den weiteren Aufgaben dieser Schicht.
- **Schicht 7** ist die Schnittstelle zwischen den Anwendungen und den Netzwerkdiensten. Hier werden verschiedene Funktionalitäten bereitgestellt, wie z. B. Datenübertragung und E-Mail.

Die Schichten 5 bis 7 sind oftmals nicht getrennt anzutreffen, sondern werden meist in einem Protokoll zusammengefasst. Als Beispiel dient das *Hypertext Transfer Protocol* (HTTP) zur Übertragung von Webseiten, welches direkt auf dem TCP/IP-Stack aufsetzt und meist in den Webbrowsern und Webservern implementiert ist.

	Schicht	Protokoll
5-7	Anwendung	HTTP
4	Transport	TCP
3	Internet	IP
1-2	Netzzugang	z. B. Ethernet

Tabelle B.4: HTTP im OSI-Schichtenmodell

Jede Schicht stellt der nächsthöheren in einer Schnittstelle Dienste über Methoden bereit, die den Zugriff auf die Datenstrukturen erlauben. Die Kommunikation zwischen gleichwertigen Schichten zweier Netzknoten scheint horizontal abzulaufen, in Wirklichkeit werden alle darunterliegenden Schichten erst vertikal bis zur Übertragung durch das Netzwerk durchlaufen. Schichten können dabei nicht übersprungen werden. Jede Schicht fügt beim Senden dem zu sendenden Paket einen eigenen Header mit für sie wichtigen Informationen vorn an, der dann auf Empfängerseite wieder schrittweise entfernt wird (s. Abbildung B.7).

Port (Fischer u. Hofer, 2008)**:** TCP- oder UDP-Dienste werden an einen Port gebunden, dessen Kennung einer 16-Bit großen Zahl entspricht. Theoretisch wären damit 65536 gleichzeitige Verbindungen pro Protokoll möglich. Die sogenannten *well known ports* für bekannte Anwendungen im Bereich 0 bis 1023 werden von der Internet Assigned Numbers Authority (IANA) fest vergeben, die u. a. auch für die Zuweisung von Top Level Domains zuständig ist. Ports im Bereich von 1024 bis 49151 können von Anwendungsherstellern bei der IANA registriert werden lassen, um einen Dienst anhand der Portnummer zu erkennen. Der Name *registered ports* deutet darauf hin. Die restlichen Ports (*private ports*) können von jedermann frei verwendet werden.

Remote Procedure Call (RPC): RPC ist eine Technik zum entfernten Prozeduraufruf außerhalb des eigenen Adressraums in verteilten Systemen. Sie basiert auf einer Client-Server-Architektur und ist fast immer synchron ausgelegt, d. h. der Prozess auf der Clientseite wartet, bis der Server das Ergebnis zurückliefert. Hauptziel bei der Entwicklung war, dass der Aufruf einer entfernten Prozedur wie der Aufruf auf der eigenen Maschine

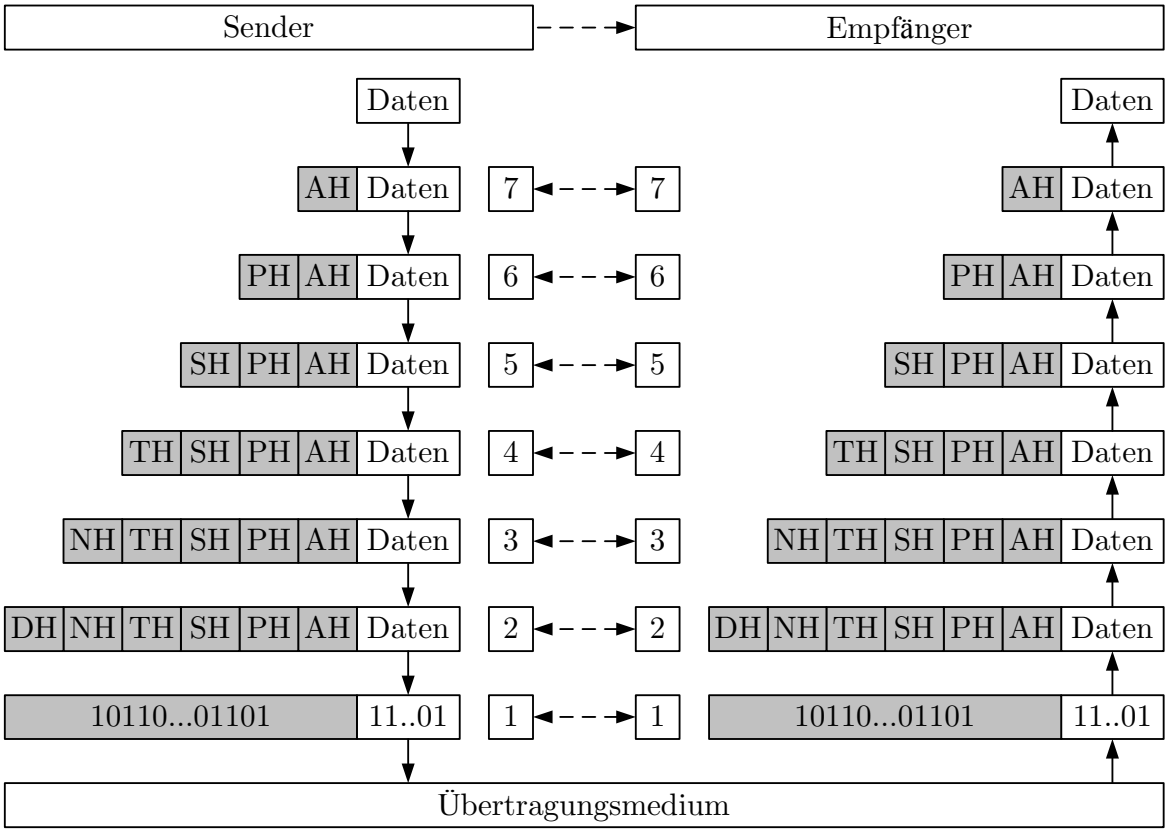


Abbildung B.7: Datenübertragung im OSI-Schichtenmodell

aussieht, um somit eine Zugriffstransparenz zu erreichen. Im OSI-Modell ist RPC auf der fünften, teilweise auch sechsten, Schicht angesiedelt.

Der Austausch von Daten zwischen Systemen im Netz erfolgt über Nachrichten mit den dazugehörigen Kommandos *send* und *receive*, die aber vor dem Prozess verborgen bleiben sollen. Als Lösung werden sogenannte Stubs[8] auf Client- und Serverseite eingeführt, die dem Stellvertreter- bzw. Proxy-Entwurfsmuster folgen (Gamma u. a., 2004). Der Client-Stub enthält keine Implementierung der Prozedur, sondern wandelt die Anfrage in eine Nachricht um, die dann an den Server weitergeleitet wird. Der Umwandlungsvorgang heißt *Marshalling*[9], was nach (Fischer u. Hofer, 2008) eine *„Umformung von Daten aus einer bestimmten Struktur in einen Nachrichten-Datenstrom oder eine andere Struktur“* bedeutet. Der Server-Stub nimmt die Nachricht entgegen, packt sie aus (*Unmarshalling*) und führt die auf dem Server befindliche Implementierung der Prozedur aus. Die Ergebnisse werden auf umgekehrtem Weg zum Client zurückgeschickt. Das Konzept und der schematische Ablauf sind in Abbildung B.8 dargestellt.

Für eine erfolgreiche Nachrichtenübertragung müssen vorher das Nachrichtenformat, die Darstellung der einfachen Datenstrukturen und die verwendete Byte-Reihenfolge – Little-

[8] stub ⟨engl., „Stumpf, Stummel“⟩
[9] marshal ⟨engl., „anordnen, arrangieren“⟩

oder Big-Endian[10] – festgelegt werden. Die Schnittstellen zwischen den Stubs und den Anwendungen werden häufig mit der Schnittstellenbeschreibungssprache IDL[11] der Object Management Group (OMG) unabhängig von einer Programmiersprache beschrieben. Ein IDL-Compiler kann daraus die Schnittstellen und Stubs für eine spezielle Programmiersprache und Rechnerarchitektur erzeugen.

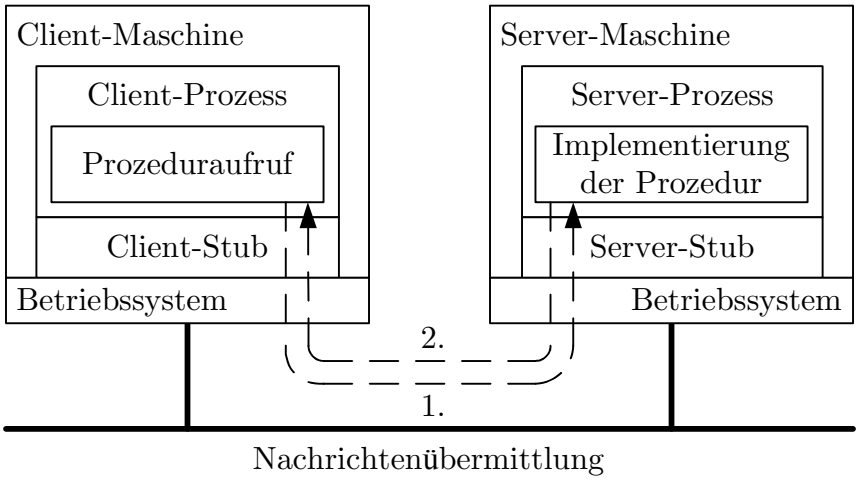


Abbildung B.8: Konzept des entfernten Methodenaufrufs (RPC)

Entfernte Objekte (Tanenbaum u. van Steen, 2003), (Emmerich, 2003), (Eddon u. Eddon, 1998)**:** Das Konzept der entfernten Objekte basiert auf RPC und ist diesem recht ähnlich, mit dem Unterschied, dass keine Prozedur, sondern Methoden eines Objekts auf dem Server aufgerufen werden. Eine vorher definierte Schnittstelle beinhaltet alle die Methoden, die das Objekt zu implementieren hat, und ist auf Client- sowie Server-Seite bekannt. Das Schnittstellenkonzept der objektorientierten Programmierung ist eine notwendige Voraussetzung für verteilte Objekte. Verbessert gegenüber RPC wurde auch die Fehlerbehandlung: Während RPC bei einem Fehler einfach einen Null-Wert zurückliefert, werden nun Ausnahmen übergeben, auf die je nach Typ reagiert werden kann. Die Stubs können von einem Compiler automatisch erzeugt werden, wobei der Server-Stub vom Programmierer noch erweitert werden kann und deshalb als Skeleton[12] bezeichnet wird. Das Konzept und ein beispielhafter Methodenaufruf zeigt Abbildung B.9.

Der Client fragt den Server nach einem entfernten Objekt, welches dann als Proxy in den Adressraum des Clients geladen wird. Ein Methodenaufruf wird vom Stub als Nachricht verpackt und zum Server geschickt. Der Skeleton entpackt diese, startet die Methode des Objekts und liefert das Ergebnis zurück. Für den Client erfolgt der Methodenaufruf vollkommen transparent.

Ein Server, der entfernte Objekte vorhält, muss über einen Namensdienst (engl. *naming service*) verfügen. Die Namen innerhalb des Namensraums müssen eindeutig sein und die

[10] Bei Big-Endian wird im Gegensatz zu Little-Endian das Byte mit den höchstwertigsten Bits zuerst gespeichert.
[11] IDL = Interface Description Language (OMG, 2002)
[12] skeleton ⟨engl., „Skelett, Gerüst“⟩

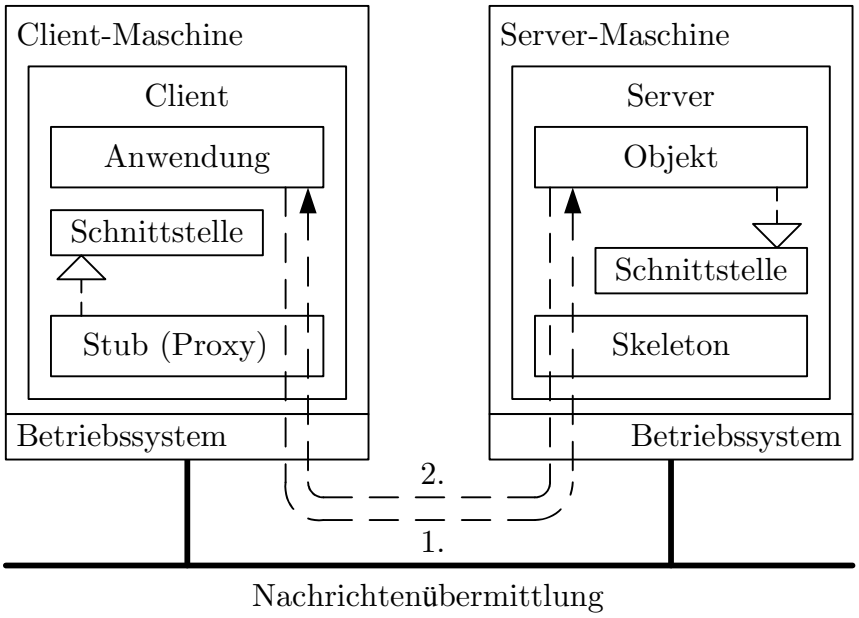


Abbildung B.9: Konzept von entfernten Objekten

Zuordnung eines Namens an ein Objekt wird als Namensbindung bezeichnet. Die Menge aller Namensbindungen ist eine Abbildung, denn allen Namen ist ein Objekt zugeordnet, aber nicht alle Objekte besitzen einen Namen. Jeder Namensserver muss mindestens die Operation *bind* und *resolve* unterstützen. Die erste weist einem Namen eine Objektreferenz zu, die zweite liefert für einen Namen eine Objektreferenz zurück.

Für den entfernten Methodenaufruf findet auch der Begriff *Remote Method Invocation* (RMI) Verwendung. Bekannte Implementierungen von RMI sind:

- **Common Object Request Broker Architecture (CORBA):** CORBA wurde von der OMG entwickelt, um verteilte Anwendungen auf heterogenen Systemen zu unterstützen, d. h. unabhängig von Rechnerarchitektur, Betriebssystem und Programmiersprache. Deshalb ist CORBA nicht an eine spezielle Programmiersprache gebunden, sondern es wird die Interface Description Language (IDL) zur Definition der Schnittstellen und Methoden benutzt. Kernstück der Architektur ist der Object Request Broker (ORB), der die Kommunikation zwischen den Objekten verwaltet.
- **Distributed Component Object Model (DCOM):** Die Firma Microsoft führte 1995 das Component Object Model (COM) ein, um die Kommunikation zwischen Programmen, Komponenten und Modulen zu ermöglichen und um für eine bessere Wiederverwendbarkeit zu sorgen. Die Funktionalität solcher COM-Komponenten werden über Schnittstellen (COM-Interface) festgelegt. Wenig später (1996) erweiterte Microsoft COM um die Kommunikationsmöglichkeit über ein Netzwerk und nannte es *Distributed Component Object Model* (DCOM).
- **Java/RMI:** Mit dem JDK[13] 1.1 wurde eine Programmierschnittstelle (API)[14] für

[13] JDK = Java Development Kit
[14] API = Application Programming Interface

die Unterstützung von entfernten Objekten in Java integriert, die als Java/RMI bezeichnet wurde. Mittlerweile wird der Begriff RMI synonym für das API verwendet. Im Gegensatz zu CORBA und DCOM unterstützt es nur die Sprache Java auf Client- und Serverseite, was zwar die Einsatzmöglichkeiten einschränkt, aber Vorteile bei der Einbindung in Java-Programme und eine höhere Performance bietet.

- **Schnittstelle**: Eine gesonderte Schnittstellenbeschreibungssprache ist nicht nötig, da die Schnittstellen direkt mit der Sprache Java formuliert werden. Die Schnittstelle wird als *Remote-Interface* bezeichnet und auf dem Server sowie dem Client gespeichert.
- **Stellvertreterobjekte**: Der Stub und der Skeleton mussten bis zum JDK 1.4 gesondert mit dem RMI-Compiler *rmic* erstellt werden, seit dem JDK 1.5 übernimmt das die Java Virtual Machine (JVM).
- **Namensdienst**: Dieser Dienst wird von einem eigenen Programm, der *RMI Registry*, übernommen, die früher extern gestartet werden musste. Inzwischen kann eine Registry-Instanz auch von der Server-Anwendung erzeugt und exportiert werden. Der Namensdienst wird standardmäßig an den reservierten Port 1099 gebunden.
- **Protokolle**: Java/RMI unterstützt vier verschiedene Protokolle:
  - Das Java Remote Method Protocol (JRMP) ist das Standardprotokoll von RMI.
  - RMI kann über HTTP getunnelt werden, um durch Firewalls, die den Port 80 freigeschaltet haben, zu kommunizieren.
  - Für die sichere Kommunikation lässt sich der Datenstrom von RMI mit dem SSL-Protokoll verschlüsseln. Der Secure Sockets Layer (SSL) befindet sich im OSI-Modell zwischen der Transportschicht und den darauf aufbauenden Schichten.
  - RMI-IIOP ermöglicht die Kommunikation mit CORBA. Das *Internet Inter-ORB Protocol* (IIOP) wurde von der OMG zu diesem Zweck verfasst (OMG, 2004).

Zu beachten ist bei Java/RMI die Parameterübergabe. Standardmäßig werden einfache Datentypen immer per Wertkopie[15] und Objekte durch die Speicheradresse[16] übergeben. Im Fall von RMI ändert sich für einfache Datentypen nichts, bei Objekten ist jedoch die Übergaberichtung zu unterscheiden. Per Definition haben Remote-Objekte keine Kenntnis von lokalen Objekten, deshalb werden lokale Objekte serialisiert und als Kopie an den Server übergeben. Die lokale Anwendung besitzt dahingegen über den Proxy immer Direktzugriff auf das Remote-Objekt im aktuellen Zustand.

[15] CBV = Call-by-Value
[16] CBR = Call-by-Reference

C UML-Klassendiagramme

C.1 Klasse WorkspaceSettings

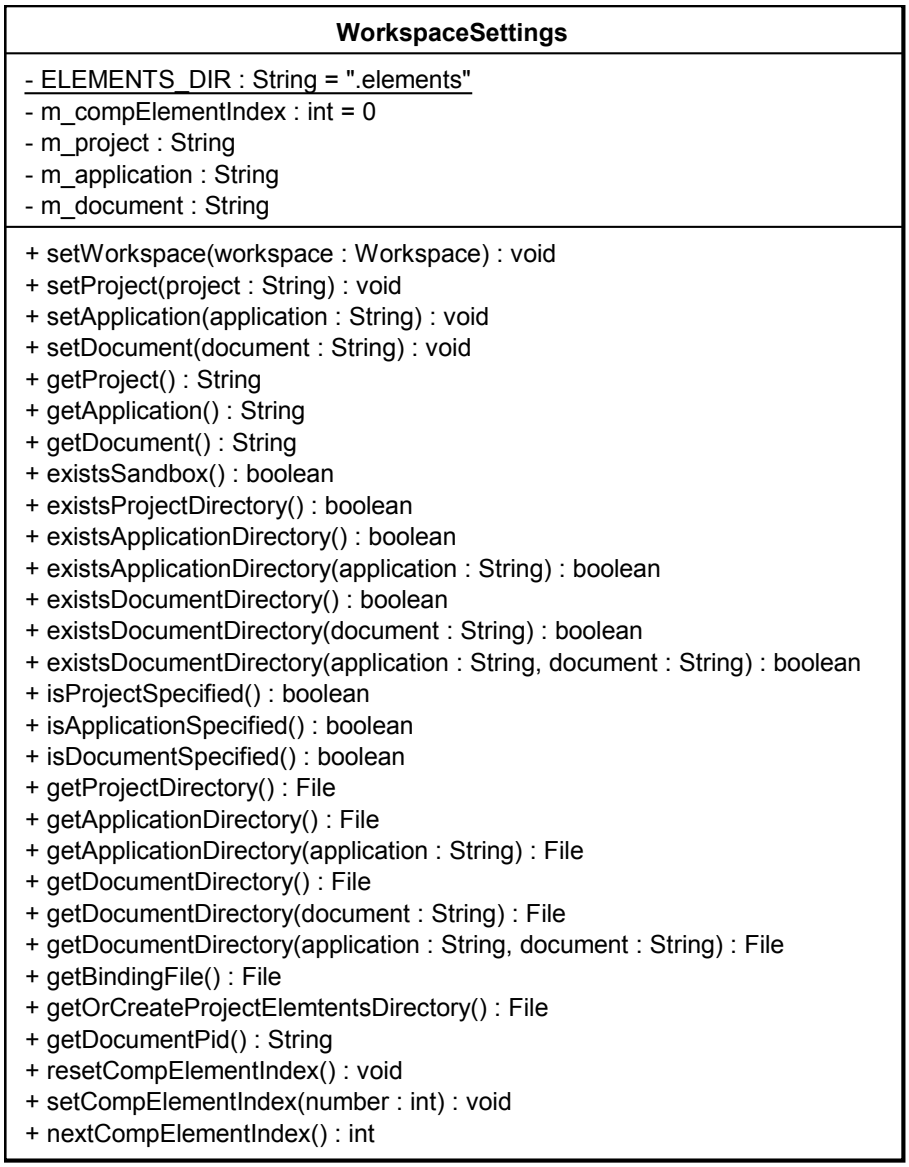


Abbildung C.1: UML-Klassendiagramm: Klasse *WorkspaceSettings*

C.2 IOObjectHandler-Hierarchie für CADEMIA-Objekte

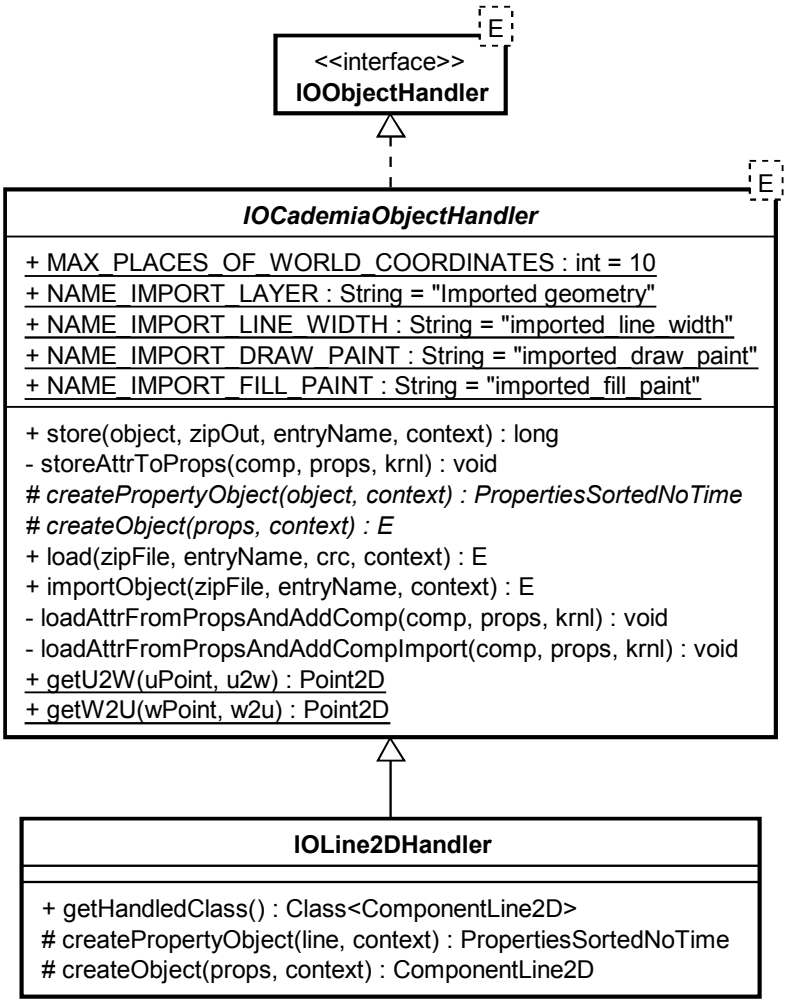


Abbildung C.2: UML-Klassendiagramm: *IOObjectHandler*-Hierarchie

D Listings

D.1 Feature-Logic-Server (RMI)

```
1  public class FLServer {
2
3      public FLServer(FeatureLogic fl) throws ProjectException {
4          if (fl == null)
5              throw new ProjectException(
6                  "FeatureLogic instance is null.");
7
8          try {
9              String hostName =
10                 InetAddress.getLocalHost().getHostAddress();
11             /* serverPackage     = objectVCS.project.server
12              * serverPackagePath = objectVCS/project/server
13              */
14             String serverPackage =
15                 FLServer.class.getPackage().getName();
16             String serverPackagePath =
17                 serverPackage.replace(".", "/");
18             URL urlServer = FLServer.class.getResource("");
19             URL urlPolicy = FLServer.class.getResource("/" +
20                 serverPackagePath + "/rmi_server.policy");
21             if (urlPolicy == null)
22                 throw new FileNotFoundException(
23                     "Servers's policy file not found.");
24             if (urlServer == null)
25                 throw new FileNotFoundException(
26                     "Servers's codebase not found.");
27
28             /* Hostname = Defines hostname of RMI-Registry
29              * Codebase = Tells where the classes are located
30              *            java.rmi.server.codebase =
31              *                  file://d:/cvs/rmi/bin
32              *            or   = http://uni-weimar.de:1111/
33              * Policy   = Limitation of rights (text file)
34              *            java.security.policy = "D:\dir\rmi.policy"
35              */
36
37             System.setProperty("java.rmi.server.hostname", hostName);
38             System.setProperty("java.rmi.server.codebase",
39                 urlServer.toExternalForm());
40             System.setProperty("java.security.policy",
```

```
41                  urlPolicy.toExternalForm());
42              System.setProperty("java.rmi.dgc.leaseValue", "900000");
43
44              String urlKeyStore = System.getProperty("user.dir") +
45                  FileOperations.getFileSeparator() +"Server_Keystore";
46              System.setProperty("javax.net.ssl.keyStore",urlKeyStore);
47              System.setProperty("javax.net.ssl.keyStorePassword",
48                  "***");
49
50              // Check output
51              System.out.println("Hostname        - " +
52                  System.getProperty("java.rmi.server.hostname"));
53              System.out.println("Code base       - " +
54                  System.getProperty("java.rmi.server.codebase"));
55              System.out.println("RMI policy file - " +
56                  System.getProperty("java.security.policy"));
57              System.out.println("javax.net.ssl.keyStore - " +
58                  System.getProperty("javax.net.ssl.keyStore"));
59
60
61              if (System.getSecurityManager() == null)
62                  System.setSecurityManager(new RMISecurityManager());
63
64              RMIClientSocketFactory rmiClientSocketFactory =
65                  new SslRMIClientSocketFactory();
66              RMIServerSocketFactory rmiServerSocketFactory =
67                  new SslRMIServerSocketFactory();
68
69              // Create Registry, Port 1099
70              Registry registry = LocateRegistry.createRegistry(
71                  FLRmiInterface.REGISTRY_PORT);
72              System.out.println(LocateRegistry.getRegistry());
73              RemoteServer.setLog(System.out);
74
75              FLRmiImpl flRmiImpl = new FLRmiImpl(fl);
76              FLRmiInterface stub =
77                  (FLRmiInterface) UnicastRemoteObject.exportObject(
78                      flRmiImpl, 5005, rmiClientSocketFactory,
79                      rmiServerSocketFactory);
80
81              registry.rebind(FLRmiInterface.SERVICE_NAME, stub);
82
83              // Redirect output and error stream to files
84              String userDir = null;
85              // In case of Eclipse project root
86              if (new File("bin").isDirectory())
87                  userDir = (System.getProperty("user.dir") +
88                                  "/bin/").replace("\\", "/");
89              else          // Inside bin directory
90                  userDir = (System.getProperty("user.dir") + "/").
91                                  replace("\\", "/");
92              File propertiesFile = new File(userDir +
93                  serverPackagePath + "/FL_server.ini");
94              if (! propertiesFile.exists()){
```

```
95                  propertiesFile =
96                      new File(userDir + "/FL_server.ini");
97                  if (! propertiesFile.exists()){
98                      System.err.println(
99                          "Could not find the FL_server.ini file.");
100                     System.exit(0);
101                 }
102             }
103             Properties properties = new Properties();
104             properties.load(new FileInputStream(propertiesFile));
105             File out = new File(properties.getProperty("out_log"));
106             File err = new File(properties.getProperty("err_log"));
107
108             System.setOut(new PrintStream(
109                 new FileOutputStream(out, true)));
110             System.setErr(new PrintStream(
111                 new FileOutputStream(err, true)));
112             DateFormat df = DateFormat.getDateTimeInstance();
113             System.out.println("\nServer with service '" +
114                 FLRmiInterface.SERVICE_NAME + "' started at "
115                     + df.format(System.currentTimeMillis()));
116             System.err.println("\nServer with service '" +
117                 FLRmiInterface.SERVICE_NAME + "' started at "
118                     + df.format(System.currentTimeMillis()));
119         } catch (IOException e) {
120             e.printStackTrace();
121         }
122     }
123 }
```

Listing D.1: Feature-Logic-Server

D.2 Implementierung der RMI-Schnittstelle

```
1  public class FLRmiImpl implements FLRmiInterface, Unreferenced{
2      FeatureLogic m_featureLogic;
3      Interpreter m_interpreter;
4
5      public FLRmiImpl(FeatureLogic fl) {
6          m_featureLogic = fl;
7      }
8
9      public void unreferenced() {
10         System.out.println("\n" + new FormattedDate()
11                 + ": FLRmiImpl is now unreferenced.");
12     }
13
14     public boolean checkConnection() throws RemoteException {
15         if (m_interpreter == null) {
16             m_interpreter = new Interpreter(new StringReader("{}"));
17             m_interpreter.setFeatureLogic(m_featureLogic);
18         }
19         else
20             m_interpreter.ReInit(new StringReader("{}"));
21         try {
22             SetDsc setDsc = interpreter.start();
23             if (setDsc == null)
24                 return false;
25         } catch (Exception e) {
26             return false;
27         }
28         return true;
29     }
30
31     public boolean clear() throws RemoteException {
32         if (m_featureLogic != null) {
33             try {
34                 m_featureLogic.clear(false);
35                 System.out.println("FL Server: " +
36                     new FormattedDate() +
37                     " -> FeatureLogic cleared.");
38                 return true;
39             } catch (SQLException e) {
40                 e.printStackTrace();
41             }
42         }
43         return false;
44     }
45
46     public boolean commit(FLData flData) throws RemoteException {
47         try {
48             System.out.println("\nFL Server: " +
49                 new FormattedDate() + " -> Commit starts.");
50             System.out.println("FL Data: " +
51                 Util.serializedSize(flData)/1024 + " kB.");
```

```
52
53              flData.store(m_featureLogic);
54              System.out.println("FL Server: " + new FormattedDate() +
55                  " -> Commit ends.");
56              return true;
57          } catch (FeatureLogicException e) {
58              System.err.println("Error: " + new FormattedDate());
59              e.printStackTrace();
60              return false;
61          }
62      }
63
64      public Set<String> askQuery(String query) throws RemoteException,
65              ParseException, SQLException, FeatureLogicException {
66          if (m_interpreter == null) {
67              m_interpreter = new Interpreter(new StringReader(query));
68              m_interpreter.setFeatureLogic(m_featureLogic);
69          }
70          else
71              m_interpreter.ReInit(new StringReader(query));
72          SetDsc setDsc = interpreter.start();
73          ResultSetIterator elmIt =
74              m_featureLogic.resultSetIterator(setDsc);
75
76          Set<String> resultSet = new HashSet<String>();
77          if (elmIt == null) {
78              return resultSet;
79          }
80          while (elmIt.hasNext())
81              resultSet.add(elmIt.next());
82          elmIt.close();
83          return resultSet;
84      }
85
86      public String getFeatureValue(String element, String feature)
87              throws RemoteException, SQLException,
88                     FeatureLogicException {
89          return m_featureLogic.getFeatureValue(element, feature);
90      }
91
92      public Map<String, String> getFeatureValues(String element)
93              throws RemoteException, SQLException,
94                     FeatureLogicException {
95          Map<String,String> features = new HashMap<String, String>();
96          m_featureLogic.getFeatureValues(element, features);
97          return features;
98      }
99
100     public Map<String,Map<String,String>> getFeatureValues(
101             Set<String> elements) throws RemoteException,
102             SQLException, FeatureLogicException {
103         return m_featureLogic.getFeatureValues(elements);
104     }
105
```

```
106     public Map<String, Object> getAtomValues(Set<String> elements)
107             throws RemoteException {
108
109         Map<String, Object> atomsMap = new HashMap<String, Object>();
110         for (String elm : elements) {
111             try {
112                 if (m_featureLogic.isUsedAsPrimitiveElement(elm))
113                     atomsMap.put(elm, m_featureLogic.getAtom(elm));
114             } catch (SQLException e) {
115                 e.printStackTrace();
116                 atomsMap.put(elm, null);
117             } catch (FeatureLogicException e) {
118                 e.printStackTrace();
119                 atomsMap.put(elm, null);
120             }
121         }
122         return atomsMap;
123     }
124
125     public Object getAtomValue(String element)
126             throws RemoteException {
127         Set<String> inputSet = new HashSet<String>();
128         inputSet.add(element);
129         Map<String, Object> atomMap = getAtomValues(inputSet);
130         if (atomMap.containsKey(element))
131             return atomMap.get(element);
132         return null;
133     }
134
135     public Map<String,Object> getAllAtoms() {
136         try {
137             return m_featureLogic.getAllAtoms();
138         } catch (SQLException e) {
139             e.printStackTrace();
140         } catch (FeatureLogicException e) {
141             e.printStackTrace();
142         }
143         return null;
144     }
145 }
```

Listing D.2: Klasse *FLRmiImpl*: Implementierung der RMI-Schnittstelle

D.3 Workspace-Schnittstelle

```
1  public interface Workspace {
2      public static final String COMP_FILE_EXTENSION = "obj";
3      public static final boolean STORE_COMPONENT_FEATURES = false;
4
5      // Getters
6      public WorkspaceSettings getWorkspaceSettings();
7      public FLClient getFLClient();
8      public VCClient getVCClient();
9      public Class<?> getComponentSuperClass();
10
11     // FeatureLogic
12     public void reinitFeatureLogic() throws WorkspaceException;
13     public FeatureLogic getFeatureLogic();
14     public void lockLocalFeatureLogic() throws WorkspaceException;
15     public void unlockLocalFeatureLogic();
16     public boolean isLocalFeatureLogicLocked();
17     public String getApplicationName();
18
19     // Local operations
20     public void createNewDocument(Object context)
21         throws WorkspaceException;
22     public void store(File documentDir, Object context)
23         throws WorkspaceException;
24     public void load(File documentDir, Object context)
25         throws WorkspaceException;
26     public void switchProject(Object context)
27         throws WorkspaceException;
28     public boolean cleanUpSandbox() throws WorkspaceException;
29
30     // Version control
31     public Result checkServersWithResult();
32     public Result checkoutProject(String project)
33         throws WorkspaceException;
34     public void commit(String msg, String tag)
35         throws WorkspaceException;
36     public Result update(String documentVersionPvid)
37         throws WorkspaceException;
38     public Result updateNewDocuments(Set<String>
39         documentVersionPvids) throws WorkspaceException;
40
41     public long getTimeSinceLastServerAccess();
42     public Set<String> getApplicationsFromRepository()
43         throws WorkspaceException;
44     public Set<String> getNotCheckedOutDocuments(String application)
45         throws WorkspaceException;
46     public Long getLocalRevisionOfCurrentDocument()
47         throws WorkspaceException;
48     public Long getRevisionOfLastDocumentVersion()
49         throws WorkspaceException;
50     public String getPvidOfLastDocumentVersion(String documentPid)
51         throws WorkspaceException;
```

```
52      public Long getRevisionOfLocalDocument(String docPid)
53          throws WorkspaceException;
54      public Long getRevisionOfObject(String objectPoid)
55          throws WorkspaceException;
56      public String getTagOfDocument() throws WorkspaceException;
57      public Date getLastChangeDateOfDocument()
58          throws WorkspaceException;
59      public Set<String> getLocalDocuments()
60          throws WorkspaceException;
61      public Set<String> getLocalDocuments(String application)
62          throws WorkspaceException;
63
64      // POID
65      public String getOrGenerateNewPoid(Object object)
66          throws WorkspaceException;
67      public String getOrGenerateNewPoid(Object object, String prefix)
68          throws WorkspaceException;
69      public String getPoid(Object object);
70      public boolean isPoid(String poid);
71      public Object getObjectFromPoid(String poid)
72          throws WorkspaceException;
73      public boolean objectHasPoid(Object object);
74      public void clearPoidMap();
75      public Set<String> objectsInLoadedDocument(Object context);
76
77      // POID: Imported documents
78      public boolean isImportedPoid(String poid);
79      public boolean isImportedDocument(String docPid);
80      public int numberOfImportedDocuments();
81      public Set<String> getImportedDocuments();
82
83      // Operations for imported documents
84      public int[] importDocument(String docPid, Object context)
85          throws WorkspaceException;
86      public int[] importBindingDocuments(String application,
87          Object context) throws WorkspaceException;
88      public int refreshImportedDocuments(Object context)
89          throws WorkspaceException;
90      public int dropImportedDocument(String docPid, Object context)
91          throws WorkspaceException;
92      public int dropAllImportedDocuments(Object context);
93
94      // Binding functionality
95      public boolean addBinder(PoidBinder b, boolean newBinding);
96      public boolean removeBinder(PoidBinder b);
97      public void clearAllBinders();
98      public Iterator<PoidBinder> iterateBinders();
99      public Set<PoidBinder> getBinders();
100     public Set<PoidBinder> getBinders(String destPoid)
101         throws WorkspaceException;
102     public Set<String> getBindingDocumentVersions();
103     public Set<String> getBindingDocumentVersionsFromServer(
104         String docVersPvid) throws WorkspaceException;
105     public Set<String> getBindingDocumentsWithLocalChanges()
```

```
106         throws WorkspaceException;
107
108     public int checkBindingsInSandbox(String documentPid)
109         throws WorkspaceException;
110     public int checkBindingsSimple(Set<String> objectsInDocument,
111         Set<PoidBinder> binders) throws WorkspaceException;
112     public Set<String>[] checkBindings(Set<String> objectsInDocument,
113         Set<PoidBinder> binders) throws WorkspaceException;
114     public int objectHasChangedSinceLastUpdateOrCommit(String poid)
115         throws WorkspaceException;
116     public void markBinderAsValid(PoidBinder poidBinder)
117         throws WorkspaceException;
118     public int markBindersAsValid() throws WorkspaceException;
119     public void deleteBindings(Set<PoidBinder> binders,
120         Object context) throws WorkspaceException;
121
122     // Project state
123     public boolean isProjectState();
124     public String getCurrentProjectStatePoid();
125     public String getCurrentProjectStateName();
126     public Result defineProjectState(String stateName,
127         boolean mainRelease, Set<String> docVersPvids)
128         throws WorkspaceException;
129     public TreeMap<String,String> getProjectStates()
130         throws WorkspaceException;
131     public Set<ProjectState> getProjectStatesComplete()
132         throws WorkspaceException;
133     public void switchToProjectState(ProjectState projectState,
134         Object context) throws WorkspaceException;
135     public void switchProjectStateToHead(Object context)
136         throws WorkspaceException;
137 }
```

Listing D.3: Schnittstelle *Workspace*

D.4 VCClient-Schnittstelle

```
1  public interface VCClient {
2      public static final String REVISION_FILENAME = "revisions.txt";
3      public static final String CRC_FILENAME = "crc.txt";
4      public static final String CRC_COMMIT_FILENAME = "crcCommit.txt";
5      public static final String BINDING_FILENAME = "bindings.txt";
6
7      // Setters
8      public boolean setInitialParameters(String URLString,
9          File sandboxDir, String sshUsername, String longUserName,
10         String sshPassphrase, String userName, String password,
11         File sshPrivateKeyFile);
12
13     public void setProject(String project) throws VCClientException;
14     public void setApplication(String application)
15         throws VCClientException;
16     public void setDocument(String document)
17         throws VCClientException;
18
19     // Getters
20     public String getURLString();
21     public File   getSandboxDirectory();
22     public String getUserName();
23     public String getLongUserName();
24     public String getPassword();
25     public String getSSHUserName();
26     public String getSSHPassphrase();
27     public File   getPrivateKeyFile();
28
29     // Check methods
30     public boolean checkServerSettings();
31     public boolean checkConnection();
32     public Result checkConnectionWithResult();
33
34     public boolean existsRepository();
35     public boolean existsProjectInRepository()
36         throws VCClientException;
37     public boolean isProjectVersioned() throws VCClientException;
38     public boolean existsApplicationInRepository()
39         throws VCClientException;
40     public boolean existsApplicationInRepository(String application)
41         throws VCClientException;
42     public boolean isApplicationVersioned() throws VCClientException;
43     public boolean isApplicationVersioned(String application)
44         throws VCClientException;
45     public boolean existsDocumentInSandbox(String application,
46         String document) throws VCClientException;
47     public boolean isDocumentVersioned() throws VCClientException;
48     public boolean isDocumentVersioned(String document)
49         throws VCClientException;
50     public boolean isDocumentVersioned(String application,
51         String document) throws VCClientException;
```

```
52
53      // Main version control methods
54      public void cleanUpProject() throws VCClientException;
55      public boolean checkoutProject() throws VCClientException;
56      public boolean checkoutApplication() throws VCClientException;
57      public boolean checkoutApplication(String application)
58          throws VCClientException;
59
60      public void commitApplication(String application)
61          throws VCClientException;
62      public Long commitDocument(String msg,
63          Map<File,VCFileStatus> commitResult)
64          throws VCClientException;
65      public Long commitZipDocument(String msg,
66          Map<String,VCFileStatus> commitResult)
67          throws VCClientException;
68      public long updateDocument(String application, String document,
69          String tag) throws VCClientException;
70      public Set<String> updateNewDocuments(Set<String> documents)
71          throws VCClientException;
72      public Set<String> updateNewDocuments(String application,
73          Set<String> documents) throws VCClientException;
74
75      /**
76       * @return The name of the special version control directory,
77       *         e.g. '.svn'.
78       */
79      public String getVCDirectoryName();
80      /**
81       * @return A {@link FilenameFilter} that returns all directories
82       *         except the special version control system folder.
83       */
84      public FilenameFilter getAllDirsExceptVCDirFilter();
85      public String getBindingFileBackupPath();
86
87      public Set<String>[] getDocumentStates()
88          throws VCClientException;
89      public Set<String>[] getDocumentStates(String application)
90          throws VCClientException;
91
92      public Set<String> getProjects() throws VCClientException;
93      public Set<String> getLocalProjects() throws VCClientException;
94      public Set<String> getApplications() throws VCClientException;
95      public Set<String> getLocalApplications()
96          throws VCClientException;
97      public Set<String> getDocuments() throws VCClientException;
98      public Set<String> getDocuments(String application)
99          throws VCClientException;
100     public TreeSet<String> getLocalDocuments()
101         throws VCClientException;
102     public TreeSet<String> getLocalDocuments(String application)
103         throws VCClientException;
104     public Set<String> getNotCheckedOutDocuments(String application)
105             throws VCClientException;
```

```
106     public Set<String> getLocalVersionedDocuments(String application)
107             throws VCClientException;
108
109     public TreeSet<String> getTags() throws VCClientException;
110     public long createTag(String tagName, String commitMessage)
111             throws VCClientException;
112
113     public boolean documentHasRemotlyChanged()
114         throws VCClientException;
115     public boolean documentHasLocallyChanged()
116         throws VCClientException;
117     public boolean documentHasLocallyChanged(String application,
118         String document) throws VCClientException;
119     public boolean cleanDocumentContent() throws VCClientException;
120
121     public long getCurrentCRCOfZipEntry(String application,
122         String document, String entryName) throws VCClientException;
123     public long getCRCOfZipEntryOfLastCommit(String application,
124         String document, String entryName) throws VCClientException;
125 }
```

Listing D.4: Schnittstelle *VCClient*

D.5 FLData-Schnittstelle

```
/**
 * Interface for sending FeatureLogic data.
 * Atom objects are stored as Java object not as
 * String representation.
 */
public interface FLData {
    // Adding
    public void addElementFeatureAtom(String elm, String ftr,
        Object atomValue);

    public void addElementToSet(String elm, String set);
    public void addElementFeatureElement(String elm1, String ftr,
        String elm2);
    public String addRelation(String relationType, String src,
        String dst);
    public void addBindingGraphRelation(String bindingVerPoid,
        Set<String> setOfSrcObjVersPoids, String dstObjVersPoid,
        String srcDocVersion, String dstDocVersion);
    public void addVersionGraphRelation(String src, String dst,
        String object);
    public void addProjectState(String project,
        String projectStatePoid, String name, String editor,
        Date date, boolean mainRelease, Set<String> docVersPvids);

    // Information
    public Set<String> getElements();
    public Map<String,String> getFeatures(String elm);
    public Map<String,Object> getAtoms();

    // Synchronisation with FeatureLogic data store
    public void store(FeatureLogic fl) throws
        FeatureLogicException;
    public void load(FeatureLogic fl) throws
        FeatureLogicException;
    public void load(FeatureLogic fl, SetDsc setdsc)
        throws FeatureLogicException;

    // Reset
    public void clear();
}
```

Listing D.5: Schnittstelle *FLData*

E Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Bei der Auswahl und Auswertung folgenden Materials haben mir andere Personen weder entgeltlich noch unentgeltlich geholfen.

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Ich versichere ehrenwörtlich, dass ich nach bestem Wissen die reine Wahrheit gesagt und nichts verschwiegen habe.

Weimar, den 21. März 2009 Torsten Richter

F Über den Autor

F.1 Lebenslauf

Allgemeine Angaben

Name:	Torsten Richter
Geburtsjahr und -ort:	1975 in Gera

Schulbildung

09/1982 – 08/1987	Polytechnische Oberschule „Magnus Poser“, Gera
09/1987 – 08/1990	Polytechnische Oberschule „Werner Lamberz“, St. Gangloff
09/1990 – 06/1994	Zabel-Gymnasium, Gera, Abitur

Wehrdienst

10/1994 – 09/1995	Grundwehrdienst als Vermesser in Hemau/Bayern

Studium

10/1995 – 10/2002	Studium des Bauingenieurwesens, Bauhaus-Universität Weimar, Diplom (Dipl.-Ing.): „Interaktive Bemessung und Nachweisführung von Schalentragwerken aus Stahlbeton“
03/2002 – 10/2002	Studentischer Mitarbeiter bei Prof. Dr.-Ing. habil. E. Raue, Lehrstuhl Massivbau I, Bauhaus-Universität Weimar

Beruflicher Werdegang

11/2002 – 12/2002	Wissenschaftlicher Mitarbeiter bei Prof. Dr.-Ing. habil. E. Raue, Lehrstuhl Massivbau I, Bauhaus-Universität Weimar
01/2003 – 10/2008	Wissenschaftlicher Mitarbeiter bei Prof. Dr.-Ing. K. Beucke, Lehrstuhl Informatik im Bauwesen, Bauhaus-Universität Weimar
01/2003 – 04/2006:	DFG-Projekt „interCAD“ im DFG-Schwerpunktprogramm 1103 „Vernetzt-kooperative Planungsprozesse im Konstruktiven Ingenieurbau"
11/2006 – 10/2008:	DFG-Transferprojekt „interCAD“ in Zusammenarbeit mit der HOCHTIEF Construction AG, IKS

F.2 Publikationen

[Richter 2003] Richter, T.: Ein Java-Paket zur Verarbeitung von Datenstrukturen in beliebigen Datenquellen. In: Kaapke, K. (Hrsg.) ; Wulf, A. (Hrsg.): *Forum Bauinformatik 2003*. Aachen : Shaker-Verlag, Oktober 2003 (Reihe Bauinformatik 2003), S. 136-146

[Beer u. a. 2004a] Beer, D. G. ; Firmenich, B. ; Richter, T. ; Beucke, K.: A Concept for CAD Systems with Persistent Versioned Data Models. In: *Digital Proceedings of the Tenth International Conference on Computing in Civil and Building Engineering (ICCCBE-X)*. Weimar, Juni 2004

[Beer u. a. 2004b] Beer, D. G. ; Richter, T. ; Firmenich, B. ; Beucke, K.: A Persistence Interface for Versioned Object Models. In: Dikbas, A. (Hrsg.) ; Scherer, R. (Hrsg.): *Proceedings of the Fifth European Conference on Product and Process Modelling in the Building and related Industries : eWork and eBusiness in Architecture, Engineering and Construction.* Leiden, London, New York u.a. : A. A. Balkema Publishers, September 2004

[Richter 2004] Richter, T.: Eine grafische Nutzerschnittstelle für versionierte Objektmodelle. In: Zimmermann, J. (Hrsg.) ; Geller, S. (Hrsg.): *Forum Bauinformatik 2004*. Aachen : Shaker-Verlag, September 2004 (Reihe Bauinformatik 2004), S. 264–271

[Richter 2005] Richter, T.: Diff und Merge von Objekten im verteilten Planungsprozess. In: Schley, F. (Hrsg.) ; Weber, L. (Hrsg.): Forum Bauinformatik 2005. Cottbus : Verlag der BTU Cottbus, September 2005, S. 217-225

[Richter u. Beucke 2006] Richter, T. ; Beucke, K.: Diff and merge for net-distributed applications in civil engineering. In: *Proceedings of the Eleventh International Conference on Computing in Civil and Building Engineering (ICCCBE-XI)*. Montreal/-Canada : Université du Québec, Juni 2006

[Nour u. a. 2006] Nour, M. M. ; Firmenich, B. ; Richter, T. ; Koch, Ch.: A versioned IFC database for multi-disciplinary synchronous cooperation. In: *Proceedings of the Eleventh International Conference on Computing in Civil and Building Engineering (ICCCBE-XI)*. Montreal/Canada : Université du Québec, Juni 2006

[Koch u. a. 2006] Koch, Ch. (Hrsg.) ; Richter, T. (Hrsg.) ; Tauscher, E. (Hrsg.): *Forum Bauinformatik 2006*. Verlag der Bauhaus-Universität Weimar, September 2006

[Beucke u. a. 2007] Beucke, K. ; Firmenich, B. ; Beer, D. G. ; Richter, T.: Entwurf und Verifizierung einer CAD-Systemarchitektur zur Unterstützung der verteilten technischen Bearbeitung im Konstruktiven Ingenieurbau. In: Rüppel, U.: *Abschlussbericht zum DFG-Schwerpunktprogramm „Vernetzt-kooperative Planungsprozesse im Konstruktiven Ingenieurbau“*. Heidelberg : Springer, 2007, S. 133-154

[Richter u. Beucke 2008] Richter, T. ; Beucke, K.: A Concept for Utilizing Versioned Object Models in Engineering Applications. In: *Proceedings of the Twelfth International Conference on Computing in Civil and Building Engineering (ICCCBE-XII)*. Beijing : Tsinghua University, Oktober 2008

[Firmenich u. a. 2008] Firmenich, B. ; Koch, Ch. ; Richter, T. ; Olivier, A. H. ; Beer, D. G.: CADEMIA: A Platform for the Development of Civil Engineering Applications. In: *Proceedings of the Twelfth International Conference on Computing in Civil and Building Engineering (ICCCBE-XII)*. Beijing : Tsinghua University, Oktober 2008