Diplomarbeit

Konzeption eines semantischen Datenmodells und Implementierung eines graphbasierten Datenspeichers

Conception of a semantic data model and implementation of a graph based data storage

Matthias Sedlmeier

25. April 2013

Zur Erlangung des akademischen Grades Dipl.-Ing.
des Fachbereiches Mathematik/Informatik
der Universität Bremen (Studiengang Informatik)

Erstgutachter: Prof. Dr. Martin Gogolla
Arbeitsgruppe Datenbanken

Zweitgutachter: Prof. Dr. Hans-Jörg Kreowski
Arbeitsgruppe Theoretische Informatik

Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit selbstständig verfasst und keine weiteren als die angegebenen Hilfsmittel und Quellen verwendet zu haben.

Ich erkläre mich damit einverstanden, dass meine Diplomarbeit in eine Bibliothek eingestellt oder kopiert wird.

Bremen, den 25. April 2013

Matthias Sedlmeier

Danksagung

An dieser Stelle möchte ich mich bei meinen Freunden, Bekannten und bei meiner Familie bedanken, die mich bei der Anfertigung dieser Arbeit unterstützt haben.

Weiterhin bedanke ich mich bei allen Personen, die mir das Diplomstudium an der Universität Bremen ermöglicht haben.

Mein besonderer Dank gilt Prof. Dr. Martin Gogolla, der mir stets als Ansprechpartner zur Seite stand und mir die Freiheit gelassen hat, diese Diplomarbeit nach eigenen Vorstellungen zu entwickeln.

Außerdem danke ich Prof. Dr. Hans-Jörg Kreowski für die Unterstützung im theoretischen Teil dieser Arbeit.

Ich möchte zudem allen Entwicklern von freier Software danken. Ihre Mühe und Opferbereitschaft ermöglichte die Realisierung der in dieser Arbeit entwickelten Referenzimplementierung.

Großer Dank gilt auch Gabriele Erradi und Inge Schabbehard für ihre kompetente Unterstützung bei allen Verwaltungsangelegenheiten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Eine grundlegende Komponente moderner Organisationsstrukturen, zum Beispiel in den Bereichen Verwaltung, Dienstleistung und Produktion, stellt die *Informationstechnik* (IT) dar. Eine gut funktionierende IT kann die Produktivität steigern [GJLS08, S. 40f.], indem sie Prozesse unterstützt und effizienter gestaltet. Schlecht integrierte, d. h. nicht auf die Anforderungen der Anwendung abgestimmte *elektronische Datenverarbeitung* (EDV) hingegen wird Abläufe komplizierter gestalten, unter Umständen stören und sich kontraproduktiv auswirken.

Informationssysteme (IS), die Arbeiter und Angestellte bei der Bewältigung ihrer Aufgaben unterstützen, wirken sich positiv auf Motivation, Leistungsbereitschaft und Arbeitsklima aus. Rechnergestützte Datenverarbeitung hingegen, die Nutzer gängelt, den Arbeitsfluss verkompliziert und die Informationserfassung erschwert, löst Frust und Resignation aus [NR12, S. 101f.] und ist damit nicht im Sinne rationaler und humaner Gestaltung von Produktivumgebungen.

In dem Maße, wie relevant ein gut funktionierendes Informationssystem für ein Unternehmen ist, so schwer ist der Entwurf eines effizienten und benutzerfreundlichen Systems, das Daten erfassen, speichern, verarbeiten und ausgeben soll. Die Qualität eines solchen Systems hängt im Wesentlichen von der korrekten Erfassung der Anforderungen ab [Wal01, S. 14f.], die in der Regel im Dialog mit allen, die mit dem System nach dessen Fertigstellung arbeiten, erhoben und spezifiziert werden müssen.

Je nach Organisationsstruktur muss hierfür zum Beispiel Abteilungsdenken überwunden und Rücksichtnahme auf die verschiedenen Interessen geübt werden. Zuletzt bleibt der Vorstand, der unabhängig von der persönlichen Verwendung des Systems die Gruppe der sogenannten *Stakeholder*[1] komplettiert und sich vornehmlich um die Entwicklungskosten Gedanken macht, die eine wesentliche Rolle bei der Einführung eines Informationssystems spielen.

[1] „Person, für die es aufgrund ihrer Interessenlage von Belang ist, wie ein bestimmtes Unternehmen sich verhält (z. B. Aktionär, Mitarbeiter, Kunde, Lieferant)“, Duden. Online im Internet: URL `http://www.duden.de/rechtschreibung/Stakeholder` (Stand 27.03.2013)

Ein Informationssystem hat u. a. die Aufgabe den in Unternehmen vorliegenden Informationsbedarf zu decken. Um diese Aufgabe zu erfüllen, müssen entsprechende Systeme in der Lage sein, Daten strukturiert abzulegen, zu berechnen und ausgeben zu können und zwar in einer Form, die für den Menschen verständlich ist. Informationssysteme werden in der Wirtschaftsinformatik häufig auch als MAT-Systeme bezeichnet, wobei die Abkürzung MAT für Mensch, Aufgabe und Technik steht und verdeutlicht, dass der Bedarf an Informationen in der Regel an eine Aufgabe geknüpft ist, die von einem Menschen mit technischer Hilfe bearbeitet wird [LWS08, S. 12f.].

Da Informationssysteme Unterstützung bei Aufgaben bieten sollen und diese Aufgaben in der Regel unternehmensspezifisch sind, sind auch die Anforderungen an Informationssysteme spezifisch über die Anwendungsdomäne festgelegt und werden in der Regel durch ein sogenanntes Datenmodell erfasst. Dieses Datenmodell enthält ein Abbild der Unternehmenswirklichkeit in der Form, die für die Bewältigung der angedachten Aufgaben notwendig ist und sollte daher adäquat sein, d. h. eine präzise Darstellung bieten.

Die Fähigkeit des menschlichen Verstandes zur Abstraktion von Ausschnitten der Wirklichkeit bildet die Grundlage für den Entwurf von Datenmodellen. Das Wort Abstraktion stammt aus dem Lateinischen und bedeutet *wegziehen* oder im übertragenen Sinn *entfernen* bzw. *trennen*. Der Vorgang der Abstraktion impliziert die Konzentration auf wesentliche Merkmale eines natürlichen Objektes und fordert damit im Umkehrschluss den Verzicht auf die detaillierte Betrachtung aller erkennbaren Eigenschaften. Abstraktion ist daher ein Vorgang der Vereinfachung bzw. Verallgemeinerung. Ein alltägliches Beispiel für Abstraktion ist die Bildung von Begriffen, mit denen wir Objekte sprachlich bezeichnen.

Der Entwurf eines guten Modells ist wesentlich für die Qualität eines Informationssystems und verlangt eine genaue Beobachtung und Analyse der Wirklichkeit und ihrer Bestandteile. Vereinfacht betrachtet, setzt sich die Realität aus Objekten zusammen, die Eigenschaften besitzen und Beziehungen untereinander eingehen. Für die informatische Modellierung im Kontext eines definierten Anwendungsfalls ist es wichtig zu erkennen, welche Teilmenge der beobachteten Aspekte abgebildet werden muss, um eine hinreichend genaue Repräsentation der Domäne zu erhalten.

Ein Modell besitzt dabei ebenso wie die Realität, die es abbildet, Elemente, die unterschiedlichster Natur sein können. Die Wirklichkeit kann theoretisch auf beliebige Weise verkörpert werden, sei es durch textuelle Beschreibungen, durch akustische Repräsentation oder durch visuelle Darstellung. Letztere spielt insbesondere für die

Datenmodellierung eine wichtige Rolle. Um Sachverhalte in einem Modell ausdrücken zu können, d. h. Objekte, deren Eigenschaften und Beziehungen untereinander beschreiben zu können, bedarf es einer Modellierungssprache, deren Elemente einen entsprechenden semantischen Gehalt besitzen. Die Regeln für die Verknüpfungen dieser Sprachelemente werden von einer entsprechenden Syntax festgelegt.

Diese Art der Herausbildung und Standardisierung einer spezifischen Sprache ermöglicht die Formulierung von Modellen, deren Beschreibung von allen verstanden werden kann, die die jeweilige Modellierungssprache interpretieren können. Gleichzeitig bietet die Festlegung einer Syntax die Möglichkeit zur formalen Transformation oder Modifikation von Modellen auf der Basis von bestimmten Regeln, die aufgrund erkennbarer Muster kontrollierte Änderungen bzw. Umformulierungen am Modell vornehmen.

Die Möglichkeit zur Übertragung von einem Modell in andere Modelle durch Übersetzung bzw. Transformation ist ausschlaggebend für die Realisierung des Entwurfs eines informationsverarbeitenden Systems, in dessen Phasen verschiedene Modelle auf unterschiedlichen Abstraktionsebenen betrachtet werden. Dabei werden Modelle, die ihrerseits von anderen Modellen abstrahieren, Metamodelle genannt. Der Entwurf eines Informationssystems ist im Wesentlichen davon geprägt, welche Modelle mit entsprechenden Abstraktionsgraden für welchen Zweck und in welcher Phase eingesetzt werden und wie die Übersetzung bzw. Transformation der jeweiligen Modelle in Modelle späterer Entwurfsphasen ausgeprägt ist.

Informatische Datenmodelle sind virtuell, d. h. Elemente eines solchen Modells sind nicht physisch, bestehen also nicht aus Materie und nehmen keinen Platz im Raum ein. Diese Art der Modelle unterscheidet sich daher im Wesentlichen von anderen Modellen wie sie zum Beispiel im Modellbau existieren. Datenmodelle besitzen selbst keine physische Existenz, sondern stellen diese lediglich dar. Neben den als real betrachteten Objekten können Modelle ebenso ideelle Konstrukte repräsentieren, die lediglich als Produkt des menschlichen Verstandes existieren. Ein gängiges Beispiel für die informatische Modellierung ideeller Konstrukte wird zum Beispiel mit dem Begriff *Mitgliedschaft* bezeichnet.

Die Fähigkeit zur Erstellung eines Modells ist ein wesentlicher Schritt zum Entwurf eines Informationssystems, dessen Bestandteile sich sowohl aus der benötigten Hardware als auch der benötigten Software und der Daten selbst zusammensetzen. Die folgenden Ausführungen beziehen sich auf die Softwarekomponente eines Informationssystems, die für die Datenhaltung verantwortlich ist. Weder der Entwurf von spezieller Geschäftslogik

zur Verarbeitung noch der Aufbau von Benutzungsschnittstellen zur Ein- und Ausgabe der gespeicherten Informationen sind Gegenstand dieser Arbeit.

Für die kontrollierte Speicherung von Informationen wird in der Regel eine Software eingesetzt, die als Datenbanksystem bezeichnet wird. Ein Datenbanksystem besteht aus einer Komponente, die die Daten hält und physisch speichert und einer Komponente, die diesen Datenspeicher verwaltet und den Zugriff auf Daten kontrolliert. Die Datenhaltungskomponente wird dabei allgemein als Datenbank und die Verwaltungskomponente als Datenbankmanagementsystem bezeichnet.

Der Entwurf einer Datenbank als Teil eines Informationssystems geschieht in mehreren Phasen, deren Ergebnisse jeweils in einem entsprechenden Dokument münden. Je nach Entwurfsschritt wird der zu modellierende Ausschnitt der Domäne in unterschiedlichen Modellen repräsentiert, mit denen je nach Phase entsprechend gearbeitet werden kann.

Im Zuge der Anforderungsspezifikation erfolgt zuerst die Ausarbeitung eines konzeptuellen Modells, das im Rahmen des Entwurfs von Datenbanken auch konzeptuelles Datenbankschema genannt wird. Dieses Modell wird klassischerweise in Form eines sogenannten ER- bzw. UML-Diagramms erarbeitet und dient im Datenbankbereich dem anschließenden Entwurf eines logischen Datenbankschemas, das konkret auf die Arbeitsweise des ausgesuchten Datenbankmodells eingeht. Im letzten Schritt erfolgt die Übersetzung des logischen Modells in ein sogenanntes physisches Modell, das genaue Angaben über die definierten Datenstrukturen beinhaltet.

Das am häufigsten eingesetzte Datenbankmodell ist das sogenannte relationale Datenbankmodell, bei dem das konzeptuelle Schema auf Tabellen abgebildet wird. Seit über 30 Jahren lenkt dieses Modell die Geschicke der Datenbankwelt, nachdem es sich in den 70er Jahren mit Hilfe von IBM gegen das hierarchische Modell sowie das Netzwerkmodell durchgesetzt hat.

Die Modellierung realer Sachverhalte via Tabellen erweist sich jedoch teilweise als umständlich, wie später noch genauer aufgezeigt wird. Die Anfertigung von Domänenmodellen sollte idealerweise durch eine Sprache geschehen, die in der Lage ist, auf das Wesen realer Zusammenhänge angemessen einzugehen, um damit den gesamten Entwicklungsprozess für alle Beteiligten zu vereinfachen.

Die in dieser Arbeit vorgestellten Konzepte stellen einen Versuch dar, diese Anforderung zu erfüllen. Die Ausführungen gliedern sich dabei in insgesamt sechs weitere Kapitel, über die an dieser Stelle eine kurze Übersicht gegeben werden soll.

Das zweite Kapitel *Problemstellung und Motivation* handelt im ersten Abschnitt von den Herausforderungen der Anforderungserhebungen im Softwareentwicklungsprozess.

Dabei wird u. a. auf die Bedeutung eines guten konzeptionellen Modells eingegangen. Im zweiten Abschnitt wird exemplarisch auf die Probleme der relationalen Modellierung eingegangen, bevor im letzten Abschnitt das Ziel dieser Arbeit formuliert wird.

Das dritte Kapitel *Stand der Technik* geht detaillierter auf den Prozess der Datenmodellierung ein und stellt zwei gängige Sprachen sowohl für die konzeptionelle als auch für die logische Modellierung vor. Diese Betrachtungen können als Grundlage für das Entwurfskapitel gesehen werden.

Das vierte Kapitel *Entwurf* beinhaltet im ersten Teil den Entwurf eines Typgraphen, der als integriertes semantisches Datenbankmodell verstanden werden kann. Im zweiten Teil folgt der Entwurf eines Graphspeichers, der diesen Typgraph als Schema nutzt, um konkrete Datenbankzustände auszuprägen.

Im fünften Kapitel *Implementierung* werden ausgesuchte Aspekte der Referenzimplementierung vorgestellt, die im Rahmen dieser Arbeit entwickelt wurde.

Das sechste Kapitel *Evaluation* beinhaltet im ersten Teil ein theoretisches Experiment zur Komprimierung von Datenbankzuständen mittels Unifikation. Im zweiten Teil wird ein praktisches Experiment mit Hilfe der bestehenden Referenzimplementierung durchgeführt und es werden Messungen zu Zeit- und Speicheraufwand diskutiert.

Das letzte Kapitel *Zusammenfassung, Fazit und Ausblick* beinhaltet eine Zusammenfassung, eine Bewertung der Ergebnisse dieser Arbeit und gewährt einen Ausblick auf weitere Entwicklungsmöglichkeiten.

2 Problemstellung und Motivation

2.1 Anforderungserhebung im Softwareentwicklungsprozess

Ein wichtiges Dokument im klassischen Softwareentwicklungsprozess ist die *Anforderungsdefinition*, in der festgelegt wird, welche Eigenschaften und Funktionen das zu entwickelnde System zur Bewältigung der übertragenen Aufgaben besitzen muss. Beim Entwurf eines Informationssystems bezieht sich eine Teilmenge dieser Anforderungen auf Aussagen darüber, welche Daten in welcher Form gespeichert werden müssen, um einen vorherrschenden Informationsbedarf decken zu können.

In Organisationen wie Unternehmen muss daher zunächst ermittelt werden, welche Domänenobjekte existieren und welche Beziehungen zwischen Datenelementen bestehen, um beispielsweise Geschäftsprozesse elektronisch abbilden zu können. Neben der textuellen Darstellung von Anforderungen können insbesondere grafische Datenmodelle eingesetzt werden, die die Anforderungen auf eine übersichtliche und einfache Weise wiedergeben.

Um jeden wichtigen Aspekt des betrachteten Anwendungsbereichs berücksichtigen zu können, muss mit allen an der späteren Verwendung der Software beteiligten Gruppen kommuniziert werden, welcher Informationsbedarf konkret besteht. Dies ist insbesondere in größeren Organisationen wichtig, in denen unterschiedliche Ebenen bzw. Abteilungen mit dem in Auftrag gegebenen System arbeiten sollen.

Benutzer von Softwaresystemen sind in der Regel keine Softwareentwickler, noch haben sie zwingend eine technische Ausbildung. Weiterhin muss davon ausgegangen werden, dass sich die unterschiedlichsten Berufsgruppen mit variierenden Anforderungen und Bedürfnissen mit dem späteren System auseinandersetzen müssen.

Soll mit diesem Personenkreis oder deren Stellvertretern ein konzeptuelles Datenmodell entwickelt werden, das die Anforderung an die Datenhaltung beschreibt, so ist es von essentieller Bedeutung, dass alle Beteiligten das verwendete Modell und seine Elemente sowie Syntax verstehen. Erst wenn dieser Anspruch erfüllt ist, kann das konzeptuelle Datenmodell als Kommunikationsbasis zwischen Entwicklern

und Benutzern eingesetzt werden [BF02, S. 27ff.], die technisch weniger versiert sind. Außerdem verringert sich das Risiko, dass Anforderungen falsch oder gar nicht erfasst werden.

Aus der Praxis ist bekannt, dass Benutzern maximal zu 30 Prozent bewusst ist, was sie genau möchten und wo ihre Vorstellungen liegen. Bis zu 40 Prozent der Anforderungen sind ihnen zwar theoretisch ebenfalls bekannt, jedoch unbewusst und können so in der Regel nicht direkt formuliert werden. Weitere 30 Prozent der eigentlichen Anforderungen sind unterbewusst und müssen erst von außen herangetragen werden[1].

Auf diesen Aspekt muss bei der Anforderungserhebung durch die Wahl eines geeigneten konzeptuellen Datenmodells eingegangen werden, das Änderungen unkompliziert zulässt und zügig von Dritten interpretiert werden kann, die nicht an vorherigen Fassungen beteiligt waren. Ein gutes Datenmodell muss also Änderungen einfach zulassen können, da diese selbst bei einer sehr sauberen Erfassung der Anforderungen nicht vermeidbar sind.

Neben den zuvor aufgeführten Gründen für Anforderungsänderungen können auch andere Faktoren eine Rolle spielen. So kann ein potentieller Kunde während der Entwicklung zur Einsicht gelangen, dass bestimmte Anforderungen geändert, verworfen, erweitert oder neu formuliert werden müssen. Dies kann zum Beispiel passieren, wenn sich der Kunde a priori seiner Wünsche nicht sicher ist, oder wenn es zu internen organisatorischen oder strukturellen Änderungen beim Auftraggeber kommt.

Anforderungen können auch durch die Wahl des Entwicklungsprozesses systematisch einer gewissen Fluktuation ausgesetzt sein. Wird anstelle des klassischen Wasserfallmodells z. B. ein agiler Ansatz gewählt, so sind häufige Anreicherungen der Anforderungsdefinition Teil des Entwicklungsprinzips. Ein gutes Datenmodell kann hier das Abarbeiten agiler Entwicklungszyklen vereinfachen und den gesamten Prozess effizienter gestalten.

Anforderungsänderungen sind nicht vermeidbar. In der Praxis sind sie an der Tagesordnung und schließlich sogar Teil des agilen Entwicklungsprinzips. Es muss also dafür gesorgt werden, dass die Kosten für Änderungen minimiert werden. Grafik 2.1 zeigt, wie schnell Kosten für Änderungen wachsen können. Sobald sich das System im Entwurf befindet, steigen die Kosten für eine Änderung bereits um einen Faktor von 3, während der Entwicklung um einen Faktor von 7. Während der Testphase um einen Faktor von 50 und nach der Auslieferung um einen Faktor von 100 [Pre10, S. 408f.].

[1] Koschke, Rainer: Script zur Vorlesung Softwaretechnik (Entwicklungsprozesse), S. 16, 2006. Online im Internet: URL `http://www.informatik.uni-bremen.de/st/Lehre/swt/prozesse-1x2.pdf` (Stand 27.03.2013)

Ziel eines guten konzeptuellen Datenmodells muss es also auch sein, nicht nur die Definition von Anforderungen besser zu unterstützen, sondern auch den Entwickler.

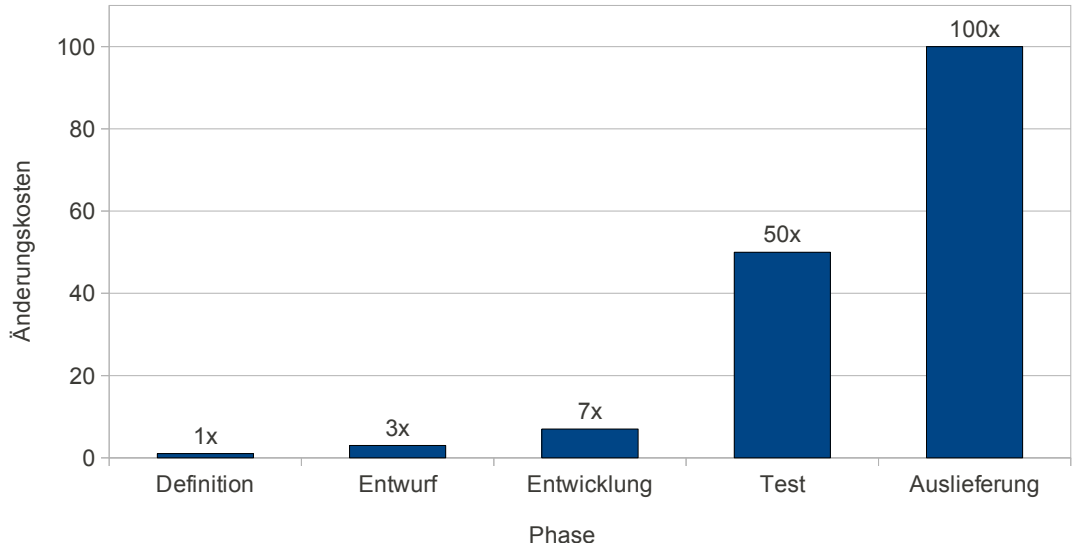

Abbildung 2.1: Übersicht Änderungskosten

Die Senkung der Kosten für Änderungen an bestehenden Systemen bietet noch eine weitere interessante Perspektive. Klein- und mittelständische Unternehmen verwenden aus Budgetgründen häufig keine für ihre Belange angepasste Individualsoftware, sondern Standardsoftware, die wenig auf die konkret zu erfüllenden Aufgaben zugeschnitten ist und damit in der Regel nicht zu der erhofften Effizienzsteigerung führt. Im schlimmsten Fall verursacht die Einführung von Software, die nicht in den Unternehmenskontext passt, eine Behinderung der Arbeitsabläufe und weitere Kosten. Für viele kleine und mittelständische Betriebe wäre es von großem Interesse, maßgeschneiderte Software zu günstigen Konditionen zu erhalten.

2.2 Probleme relationaler Modellierung

Die Datenhaltungskomponente eines Informationssystems besteht heute in der Regel aus einer relationalen Datenbank, die in drei Phasen entworfen wird. Die erste Phase sieht die Erfassung der Domäne in Form eines konzeptuellen Modells vor. Dieses konzeptuelle Modell wird anschließend in ein logisches Modell umgewandelt, aus dem dann das physische Modell abgeleitet wird.

Das konzeptuelle Modell kann zum Beispiel durch ER- bzw. UML-Diagramme dargestellt werden, die eine grafische Repräsentation der Domäne bieten und wichtige Sachverhalte wie Beziehungen direkt abbilden können.

Das logische Modell wird im Bereich der relationalen Datenbanken unter Verwendung der relationalen Entwurfstheorie erstellt. Hierzu werden die im konzeptuellen Modell formulierten Aspekte auf Elemente des relationalen Modells abgebildet, also auf Relationen (Tabellen) und deren Attribute (Spalten). Anschließende Schritte können u. a. aus Normalisierungen bzw. Synthese von Relationen bestehen.

Im letzten Schritt erfolgt die Übersetzung in ein physisches Modell, das in der relationalen Welt in der Regel in SQL formuliert wird.

Am Ende des Entwurfs einer relationalen Datenbank existieren folglich drei Modelle, die in verschiedenen Sprachen vorliegen. Bei Änderungen eines dieser Modelle müssen die beiden übrigen ebenfalls aktualisiert werden, um inkonsistente Abbildungen auszuschließen. Einerseits ist die Korrektheit des konzeptionellen Modells wichtig, da es als Basis für die Kommunikation mit dem Kunden dient. Andererseits sollten die abgeleiteten Modelle mit dem konzeptuellen Modell im Einklang stehen, damit die Domäne korrekt repräsentiert wird.

An dieser Stelle soll erwähnt werden, dass es technisch möglich ist, jedes einzelne Modell an sich zu manipulieren, ohne dass die Veränderungen automatisch in die anderen Ebenen einfließen. Es muss in der Regel ausdrücklich darauf geachtet werden, dass der Zusammenhang zwischen den Modellen erhalten bleibt, da dies nicht als gegeben angesehen werden kann.

Diese Synchronisation beinhaltet kontinuierliche Modelltransformationen in verschiedene Sprachen und ist u. U. verlustbehaftet, wenn es keine direkte Abbildung zwischen den verwendeten Sprachmitteln gibt. Auch aus diesem Grund ist eine Transformation ggf. nicht direkt umkehrbar, was die Synchronisation erschweren kann.

Wie bereits erwähnt wurde, spielt das konzeptuelle Modell insbesondere bei der Kommunikation mit dem Kunden eine wichtige Rolle. Es wird u. a. in Form von ER- bzw. UML-Diagrammen erstellt, da diese für technische Laien verständlicher sind als beispielsweise Relationenschemata. Dies hängt einerseits mit der unterschiedlichen visuellen Darstellung zusammen und andererseits mit der Art, wie Domänenzusammenhänge in diesen Modellen repräsentiert werden.

Um ein konkretes Beispiel zu nennen, sei auf die Darstellung von Beziehungen verwiesen. Bei der Darstellung mittels ER- bzw. UML-Diagramm ist eine Beziehung

zwischen zwei Domänenobjekten direkt ersichtlich. Zwischen den beiden grafisch dargestellten Objekten wird eine Kante gezogen, die so den Zusammenhang ausdrückt.

Die Darstellungen von Beziehungen mittels Relationen bleibt hinter dieser Einfachheit zurück, da sie auf Fremdschlüssel und Verbundtabellen zurückgreifen muss, um auszudrücken, dass zwei oder mehr Objekte in Verbindung stehen. Diese indirekte Darstellung reflektiert nicht, dass die Zusammenhänge von Domänenobjekten eher Graphstrukturen entsprechen, worauf im konzeptuellen Modell noch direkt eingegangen werden kann.

Dieses Darstellungsproblem ist der Tatsache geschuldet, dass Relationen Zusammenhänge und Strukturen einer Domäne nur flach ausdrücken können und betrifft weitere Modellierungsaspekte, auf die an dieser Stelle eingegangen werden soll.

Domänenobjekte haben in der Regel bestimmte Attribute, die der detaillierteren Beschreibung dienen. Sind diese Attribute flach, so lassen sie sich direkt über Tabellenspalten abbilden. Strukturierte Attribute jedoch bilden Hierarchien aus, die nur indirekt dargestellt werden können, indem sie zum Beispiel über den relationalen Verbund zusammengesetzt werden. Dieser Weg muss auch dann genommen werden, falls es sich um multiple Attribute handelt, die eine beliebige maximale Kardinalität aufweisen.

Ein anderes Problem ist der sogenannte objektrelationale Bruch (engl., Sg.: *Object Relational Impedance Mismatch*), der in Zusammenhang mit Fachlogik auftritt, die auf Basis des objektorientierten Paradigmas konstruiert wurde. Datenstrukturen, die zuvor für die relationale Speicherung in eine flache Repräsentation überführt wurden, müssen zur objektorientierten Verarbeitung zunächst in eine strukturierte Darstellung übersetzt werden. Nach der Verarbeitung erfolgt ein erneuter Wechsel in die flache Darstellung. Dieser kontinuierliche Darstellungswechsel erzeugt einen zusätzlichen Aufwand, der durch direkte strukturierte Speicherung vermieden werden kann.

Die Abbildung von relationalen Datenbeständen auf Objektstrukturen erfolgt häufig durch einen sogenannten *objektrelationalen Mapper*, der als weitere Schicht in die Architektur des Informationssystems aufgenommen werden kann. Hierbei gilt es zu beachten, dass die Repräsentation von Relationen als Objektstrukturen Synchronisationsmaßnahmen erfordern, da die Fachlogik in diesem Fall auf einer Kopie von Originaldatenbeständen arbeitet.

Ein weiterer Punkt bezieht sich auf das Konzept der Vererbung. Sie kann zwar mit Hilfe von Mustern wie *Class Table Inheritance*, *Concrete Tabel Inheritance* oder *Single*

Inheritance [Fow03, S. 285, S. 293, S. 278] simuliert werden, ist jedoch kein Bestandteil der relationalen Theorie und bietet so auch keine direkte Darstellung.

In den letzten Absätzen wurden Probleme erläutert, die im Zuge eines relationalen 3-Phasen-Entwurfs auftreten können. Einerseits stellt die Synchronisation von drei Modellen hohe Anforderungen an die Entwicklungsdisziplin. Andererseits führt die Transformation des konzeptuellen Modells in das relationale zu einem Verlust der direkten Darstellung von Zusammenhängen und Strukturen.

Ein Lösungsansatz für dieses Problem ist der Entwurf eines integrierten Datenmodells, das als Synthese zwischen konzeptuellem, logischem und pyhsischem Modell angesehen werden kann. Als Basis können dabei das ER- bzw. UML-Klassenmodell sowie ein graphbasierter Ansatz zur Datenspeicherung direkt im Hauptspeicher dienen. Einen ähnlichen Ansatz verfolgen Serge Abiteboul und Richard Hull in ihrer Arbeit über den Entwurf eines semantischen Datenbankmodells [AH84].

2.3 Ziel dieser Arbeit

Ziel dieser Arbeit ist es, die aufgeführten Probleme zu lösen, indem ein semantisches Modell eingeführt wird, das grafische Schemadefinitionen erlaubt und sich in seiner Ausdrucksstärke auf der konzeptuellen Ebene bewegt. Dieses Modell wird durch einen Typgraphen repräsentiert, über den die Domäne modelliert werden kann und dient gleichzeitig als logisches und phyisches Datenbankmodell für die Ableitung von Datenbankzuständen.

Im Entwurfskapitel dieser Arbeit wird wie vorgeschlagen eine Sprache für Typgraphen entwickelt und ebenso darauf eingegangen, wie Instanzgraphen daraus abgeleitet und Datenbankzustände aufgebaut werden können. Dies sind zwei der drei notwendigen Elemente für ein Datenbankmodell, das nach Edgar F. Codd aus folgenden Komponenten bestehen sollte [Cod80, S. 122]:

- Einer Sammlung von generischen Integritätsregeln, die Konsistenz gewährleisten.
- Einer Datenstruktur zum Aufbau von Zuständen.
- Operatoren, mit denen Zustandsdaten abgefragt bzw. geändert werden können.

3 Grundlagen und Stand der Technik

3.1 Datenmodelle als Grundlage für Informationssysteme

Durch den Einsatz von Informationssystemen in Unternehmen und Organisationen können Produktions- bzw. Geschäftsprozesse unterstützt und effizienter gestaltet werden.

Grundlage für die Entwicklung eines Informationssystems ist ein ausgereiftes Datenmodell, das als Produkt aus dem Prozess der Datenmodellierung hervorgeht und alle Informationsanforderungen im Kontext des konkreten Anwendungsbereichs erfüllt. Ein Datenmodell reflektiert also möglichst exakt die Informationsanforderungen eines Unternehmens, indem es auf alle relevanten Aspekte eingeht.

Wie eingangs bereits erwähnt wurde, spielt die adäquate Abbildung aller relevanten Objekte, Eigenschaften und Beziehungen eine maßgebliche Rolle für die Entwurfsgüte und die Qualität der Implementierung einer entsprechenden Softwarekomponente. Dabei muss festgestellt werden, dass alle Datenmodelle, die im Laufe einer Entwicklung entstehen, lediglich die *fachlich-inhaltliche* Sicht beschreiben und keine Informationen über etwaige Geschäftslogik beinhalten. Die Spezifikation von Funktionen, die die gespeicherten Daten als Eingabe für eine Weiterverarbeitung nutzen, ist also nicht Bestandteil eines Datenmodells, wird in der Regel jedoch von diesem beeinflusst. Schließlich werden Funktionen den Daten angepasst, die sie verarbeiten sollen, und nicht umgekehrt.

Sowohl die Spezifikation von Geschäftslogik, der Entwurf von Benutzungs-Schnittstellen als auch die Auswahl und Installation der richtigen Hardware liegen außerhalb des Rahmens dieser Arbeit. Diese Punkte finden hier lediglich Erwähnung, um zu verdeutlichen, dass ein Informationssystem ein komplexes Gebilde ist, dessen erwartungsgemäßes Funktionieren von vielen Faktoren abhängt. Der Fokus liegt auf jener Softwarekomponente, die die strukturierte Speicherung von Daten auf Basis definierter Regeln übernimmt und den Zugriff auf diese Daten kontrolliert.

Inhaltlich nimmt dieses Kapitel u. a. Bezug zu Themen, die in [SSH08], [Pon07], [GP03], [HM06], [Stö05], [Som07] und [Gei11] behandelt werden und dient damit als Grundlage für die Ausarbeitung des Konzept- bzw. Implementierungskapitels.

3.2 Datenmodellierung in drei Phasen

Bei der Datenmodellierung werden in der Regel drei Phasen unterschieden, deren Aktivitäten jeweils Produkte in Form von grafischen oder textuellen Modellen hervorbringen. Bei diesen Produkten handelt es sich stets um abgeleitete Modelle der jeweils vorangegangenen Phase, beginnend mit einem sehr abstrakten Modell, das zu Entwurfsbeginn angefertigt wird.

Das Ausgangsmodell in dieser Modellhierarchie stellt das sogenannte konzeptionelle Modell dar, das häufig auch den Namen konzeptuelles Datenbankschema trägt und zu der Familie der semantischen Datenmodelle gehört. Dieses konzeptionelle Modell ist als Basismodell Dreh- und Angelpunkt bei der Erfassung der Anforderungen des Informationsbedarfs und muss in der Regel sowohl von Technikern als auch von Laien verstanden werden können, um Missverständnissen bei der Anforderungsdefinition vorzubeugen. Die Modellsprache muss daher möglichst einfach gehalten sein, dabei jedoch gleichzeitig alle notwendigen Konstrukte für eine adäquate Abbildung der Realität mitbringen. Diese Anforderung ist in Teilen ambivalent und unsicher, da weder exakt definiert werden kann was einfach noch was adäquat ist. Das am häufigsten eingesetzte Modell für die Anfertigung von konzeptionellen Modellen ist das sogenannte Entity-Relationship-Modell (ER-Modell), dessen Sprachelemente grafisch dargestellt werden.

In der zweiten Phase wird das sogenannte logische Modell konstruiert. Dieses Modell ist weniger abstrakt als das konzeptionelle Modell und bereits an die verwendete Datenbank angepasst, indem es auf die spezifischen Datenstrukturen eingeht. An dieser Stelle ist es nur noch Technikern möglich, dieses Modell zu verstehen. Falls an dieser Stelle Anforderungsänderungen auftreten, so müssen diese zunächst in das konzeptionelle Modell eingearbeitet und dann durch Neuerstellung des logischen Modells übernommen werden. Der Standard für das logische Datenmodell ist derzeit das relationale Datenmodell, das zu speichernde Informationen in Relationen (Tabellen) organisiert und dadurch beispielsweise Hierarchien nicht natürlich abbilden kann. Es gibt weitere Modelle, die im Zuge dieses Kapitels betrachtet werden sollen.

Das physische Modell bzw. Datenbankschema ist am konkretesten und dient direkt als Eingabe für das ausgewählte Datenbankmanagementsystem, das gemäß der festgelegten

Syntax der Datendefinitionssprache das Schema übernehmen kann und anschließend für eine Verwendung bereit steht. In der relationalen Welt werden physische Modelle in der Regel in SQL formuliert.

3.3 Grundlegende Modellierungselemente

Um eine vollständige adäquate Abstraktion realer Umstände zu erhalten, ist es notwendig, jeden relevanten Aspekt im Einzelnen hinreichend gut repräsentieren zu können.

Soll ein Abbild in Form eines Modells geschaffen werden, so muss die zugrundeliegende Modellsprache also entsprechende Elemente beinhalten, die Konstrukte der Wirklichkeit darstellen. So gibt es eine Reihe klassischer Sachverhalte, die direkt abbildbar sein sollten, wie zum Beispiel Objekte, deren Attribute und Beziehungen untereinander.

Mit Objekt ist in diesem Fall jedes mögliche *Ding* gemeint, das als wahrnehmbar gilt. Im Allgemeinen hat sich dafür der Begriff Entität (engl., Sg.: *entity*) etabliert.

Zwischen Objekten bestehen in der Regel Beziehungen, die zum Beispiel Zugehörigkeit ausdrücken können. In der Datenmodellierung finden für Beziehung die Begriffe Assoziation oder *Relationship* häufig Verwendung. Weitere gebräuchliche Arten von Beziehungen stellen die sogenannte Aggregation und Komposition dar, die beide als Spezialfälle der Assoziation gesehen werden können.

Objekte lassen sich derweil durch ihre charakteristischen Merkmale bzw. Attribute differenzieren und bilden so notwendige Informationen ab. Auch hierfür existieren entsprechende Sprachelemente.

Nun lassen sich auf Basis der zuvor erwähnten Sprachmittel schon brauchbare Modelle konstruieren. Jedoch gibt es weitere Konzepte, die einen höheren Detailgrad zulassen, wie zum Beispiel Vererbung, mit der Generalisierung bzw. Spezialisierung ausgedrückt werden kann oder sogenannte Kardinalitäten, mit denen eine quantitative Aussage über Attribute bzw. Teilnahmen an Beziehungen ausgedrückt werden können und in den Bereich der Integritätsbedingungen fallen.

Die wichtigsten Modellierungselemente und ihre Semantik werden in den nächsten Absätzen detaillierter beschrieben.

3.3.1 Entitätstypen

3.3.1.1 Starke Entitätstypen

Im Zentrum der Modellierungsarbeit steht die Entität bzw. der Entitätstyp. Diese beiden Begriffe werden häufig synonym verwendet, jedoch besteht ein essentieller Unterschied zwischen ihnen. Eine Entität steht in der Regel für genau ein Objekt der Realität, für eine sogenannte Instanz.

Für eine allgemeine abstrakte Beschreibung der Realität ist die Verwendung von einzelnen spezifischen Objekten jedoch ungeeignet. Deshalb wird von dem zu beschreibenden Ding an sich auf seine Klasse bzw. seinen Typ abstrahiert und dieser stellvertretend für alle potentiell existierenden Ausprägungen in das Modell aufgenommen.

In einem Modell finden wir so Entitätstypen vor, die klassenspezifische Eigenschaften besitzen. So wird es in einem Informationssystem für Flughäfen zum Beispiel den Entitätstyp *Flugzeug* oder *Terminal* geben. Entitätstypen beziehen sich also auf unterscheidbare dingliche Objekte im Kontext der zu modellierenden Domäne.

Sie beschränken sich dabei nicht auf die Repräsentation von physikalischen Objekten, sondern können auch für ideelle Konstrukte wie beispielsweise *Mitgliedschaft* oder *Buchung* stehen.

Entitätstypen sind in der Regel durch Substantive bezeichnet und können so im Rahmen der Software-Entwicklung unter Annahme einer schriftlichen Anforderungsdefinition z. B. über die objektorientierte Textanalyse bzw. -dekomposition extrahiert werden.

3.3.1.2 Schwache Entitätstypen

Schwache Entitätstypen stehen grundsätzlich in Beziehung zu starken Entitätstypen, da sie aufgrund der Modellierung nicht unabhängig ausgeprägt werden dürfen. Die Existenz von Instanzen eines schwachen Entitätstyps impliziert in einem gültigen Datenbankzustand immer auch das Vorhandensein von starken Entitäten. Als Beispiel kann hier das ideelle Konzept der *Buchung* herangezogen werden, dessen Ausprägung ohne die Verbindung zu einer starken Entität beispielsweise vom Typ *Person* oder *Kunde* nicht gültig ist.

3.3.2 Attribute

Für den Begriff Attribut gibt es zahlreiche Synonyme wie beispielsweise Eigenschaft, Merkmal oder Charakteristikum. Attribute gehören zu einem Entitätstyp, dessen Instanziierung diese durch Belegung mit konkreten Werten ausprägt. Sie repräsentieren dabei jene Eigenschaften eines Objekts, die im Kontext des Informationsbedarfs als relevant angesehen werden und deshalb erfasst werden müssen. Neben natürlichen Eigenschaften von Objekten werden dabei auch künstliche Eigenschaften wie beispielsweise *Steuernummer* erfasst. Eigenschaften können auch in Form von selbstständigen Entitäten auftreten, die über Aggregation oder Komposition verknüpft werden. Auf die verschiedenen Arten von Attributen gehen die folgenden Absätze ein.

3.3.2.1 Flache Attribute

Flache Attribute erfassen Eigenschaften, die als atomar angesehen werden und strukturell nicht weiter zerlegt werden, beispielsweise *Alter* oder *Geschlecht*. Es sei angemerkt, dass die Atomarität von Eigenschaften vom Modellierungskontext abhängt und flexibel definiert werden kann. So können Datumsangaben entweder als flach oder strukturiert modelliert werden, je nachdem, ob eine Zerlegung bis auf die einzelnen Datumsbestandteile gefordert ist.

3.3.2.2 Strukturierte Attribute

Strukturierte Attribute setzen sich aus mehreren Bestandteilen zusammen und bilden so schwache Konzepte aus. Als Beispiel sei hier die Eigenschaft *Adresse* angeführt, die sich aus *Straße*, *Hausnummer*, *Postleitzahl* und *Ort* zusammensetzen kann. Jeder Bestandteil kann dabei seinerseits strukturiert sein. Bei dieser Form von Attributen muss im Zuge der Modellierung entschieden werden, ob es sinnvoll ist, sie u. U. als eigenständigen Entitätstyp zu repräsentieren.

3.3.2.3 Optionale Attribute

Die Qualifizierung von Attributen als optional sagt aus, dass sie beim Aufbau einer Entität nicht zwingend erfasst werden müssen, wovon in der Regel ausgegangen wird. Bei der Verarbeitung von Datensätzen muss auf diese Attributform speziell eingegangen werden, da die Existenz von optionalen Attributen nicht als Vorbedingung für eine algorithmische Behandlung angesehen werden kann.

3.3.2.4 Mehrfachattribute

Die Mehrfachheit von Attributen bedeutet, dass mehrere Werte erfasst werden können, ohne für jeden einzelnen dieser Werte ein eigenes Attribut zu definieren. Über Mehrfachattribute lassen sich so Mengen oder Listen dynamischer Mächtigkeit modellieren.

3.3.2.5 Abgeleitete Attribute

Abgeleitete oder dynamische Attribute werden aus bestehenden Attributen berechnet und in der Regel nicht manuell ausgeprägt. Für die Berechnung ist entsprechende Fachlogik notwendig, die ebenfalls in einem Datenbanksystem hinterlegt werden muss und entweder über sogenannte gespeicherte Prozeduren (engl., Sg.: *Stored Procedures*) oder über Standardfunktionen der verwendeten Datenbanksprache angesprungen werden kann. Die Trennung zwischen Daten und Verarbeitungslogik wird dabei aufgehoben.

3.3.2.6 Beziehungsattribute

Zwischen Entitäten können Beziehungen aufgebaut werden, um anzuzeigen, dass ein semantischer Zusammenhang zwischen ihnen besteht. Dieser Zusammenhang wird mittels Beziehungstypen ausgedrückt, die ebenfalls Attribute besitzen können, die den Zusammenhang detaillierter beschreiben. Diese Daten hängen dann von den in Beziehung stehenden Entitäten ab.

3.3.2.7 Schlüsselattribute

Es kann gefordert sein, dass ein Teil der Attribute funktional von einem oder mehreren Attributen abhängt. Ist dies der Fall, so können Schlüsselattribute verwendet werden, deren Werte im Kontext aller bestehender Entitäten eindeutig sein müssen. Für die Speicherung von Personendaten kann z. B. die Personalausweisnummer als Schlüsselattribut definiert werden.

3.3.3 Super- und Subtypen

Entitätstypen können aufgrund ihrer Beschaffenheit ähnliche Attributmengen aufweisen, wenn z. B. die repräsentierten Objekte ähnlich oder verwandt sind. Es entsteht so eine Kategorisierung, die ggf. die Einführung von Typhierarchien zulässt.

Hierzu werden alle infrage kommenden Entitätstypen untersucht und gleiche Attribute in einen sogenannten Supertyp extrahiert, von dem die betrachteten Entitätstypen erben können und so die Rolle von Subtypen übernehmen. Diese eingeführte Vererbungshierarchie kann über beliebig viele Ebenen verlaufen und führt entlang der Vertikalen zu einer Generalisierung bzw. Spezialisierung von Typen. In der Regel sind Supertypen, die auf einer höheren Ebene angesiedelt sind, abstrakter als erbende Subtypen.

Bei der Bildung von Subtypen findet eine Partitionierung statt [PJ00, S. 110ff.], bei der vier Fälle unterschieden werden. Diese Fälle werden in der Abbildung[1] 3.1 dargestellt.

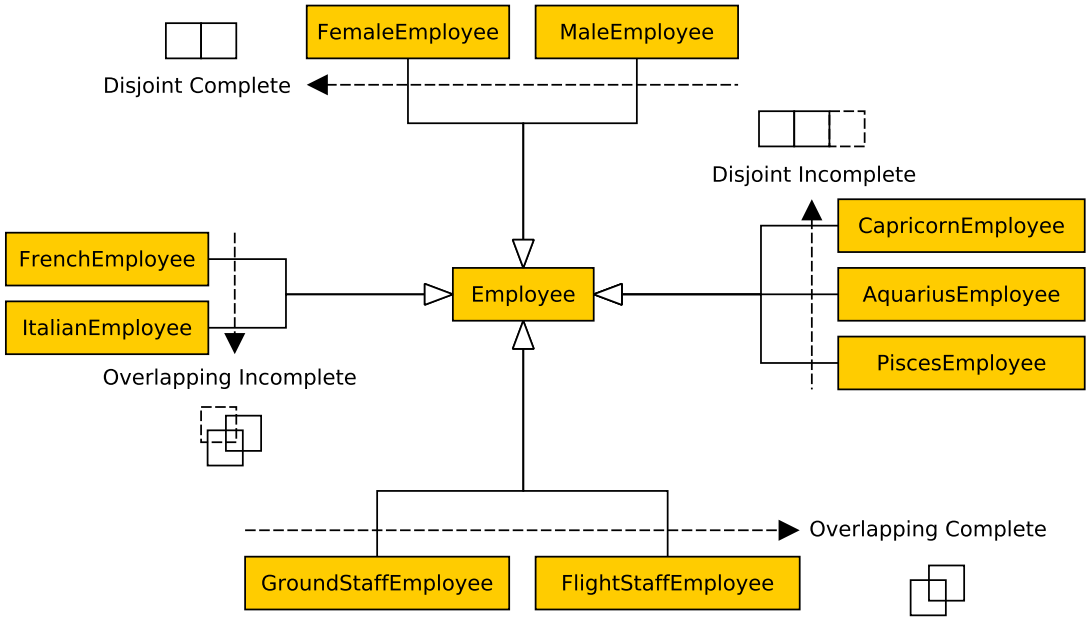

Abbildung 3.1: Partitionierung von Subtypen

3.3.4 Beziehungen

Entitätstypen können untereinander in Beziehung stehen, d. h. eine semantische Verbindung besitzen. Die *Assoziation* stellt die geringsten Anforderungen an eine solche Beziehung, da sie lediglich ausdrückt, dass ein loser Zusammenhang besteht. Allerdings macht sie keine Einschränkungen bezüglich der aufgrund dieser Verbindung potentiell bestehenden *Existenzabhängigkeiten*.

[1] Frei nach einer Abbildung von Martin Gogolla aus dem Script zur Vorlesung *Entwurf von Informationssystemen*.

So hat ein Dokument, das von einem Betrachter gelesen wurde, zu diesem eine Beziehung, die genau diesen Sachverhalt „gelesen von“ ausdrückt. Sowohl das Dokument als auch der Betrachter sind jedoch nicht existenzabhängig voneinander, d. h. die Assoziation „gelesen von“ spielt für deren Dasein keine Rolle.

Genau diesen Punkt betrifft die Aggregation bzw. Komposition, die jeweils eine Assoziation mit spezieller Konnotation darstellt. Dabei fasst die Komposition die Existenzbedingung zwischen den Teilnehmern der Beziehung noch enger als die Aggregation. Die Anforderung an die Aggregation bzw. Komposition könnte als „gehört zu“ bzw. „besteht aus“ beschrieben werden.

3.3.4.1 Komposition

Ein Beispiel soll helfen, zunächst die Komposition zu verdeutlichen. Angenommen, es gibt einen Entitätstyp *Nachrichtensendung* und einen Entitätstyp *Beitrag*. Nehmen wir weiterhin an, die der Ausprägung des Entitätstyps Nachrichtensendung konkret zugeordneten Ausprägungen des Entitätstyps Beitrag können ausschließlich im Rahmen genau dieser Sendung wiedergegeben werden. Dann ist ihre Existenz in der Welt der Datenmodellierung von der Existenz der Nachrichtensendung abhängig. Existiert die Nachrichtensendung nicht mehr, so wird es auch die Beiträge nicht mehr geben.

3.3.4.2 Aggregation

Das Szenario soll für die Erklärung der Aggregation ein wenig verändert werden. Wir gehen nun davon aus, dass die Beiträge auch von anderen Nachrichtensendungen eingespielt werden können. Ihre Existenz ist nun nicht mehr allein von einer Nachrichtensendung abhängig, sondern erst dann beendet, wenn alle aggregierenden Nachrichtensendungen hinfällig werden. Je nach Auslegung der Aggregationsbeziehung kann es auch erlaubt sein, dass Beiträge ohne Bezug zu Nachrichtensendungen existieren.

Abstrakt formuliert bedeutet die Aggregation eine Teil-Ganzes-Beziehung zu mehreren Ganzheiten und die Komposition eine Teil-Ganzes-Beziehung zu exakt einer Ganzheit.

3.3.4.3 Beziehungstypen

Entitäten können über Beziehungen in einen semantischen Zusammenhang gebracht werden. Dabei kann die Frage auftreten, wie diese Beziehung quantitativ beschaffen ist. Angenommen zwei Entitätstypen werden miteinander verknüpft, so gilt es

zu klären, wie viele Entitäten des einen Typs mit wie vielen Entitäten des anderen Typs in Verbindung stehen bzw. an wie vielen Beziehungsausprägungen die jeweiligen Teilnehmer partizipieren dürfen. Grundsätzlich werden an dieser Stelle die Beziehungstypen 1:1, 1:n sowie n:m unterschieden.

Der 1:1-Fall sagt aus, dass eine Entität von Typ A genau mit einer Entität von Typ B verbunden werden darf und stellt damit einen funktionalen Zusammenhang her. Für beide Teilnehmer bedeutet diese Konstellation, dass sie jeweils exakt einmal an einer Beziehungsausprägung beteiligt sein dürfen.

Der 1:n Fall erlaubt die Verbindung einer Entität von Typ A mit n Entitäten von Typ B. Anders ausgedrückt dürfen Instanzen des Typs A n-mal an entsprechenden Beziehungsausprägungen teilnehmen und Instanzen des Typs B genau einmal.

Eine n:m-Beziehung hebt diese Einschränkung auf und erlaubt, dass beliebig viele Instanzen von Typ A mit beliebig vielen Ausprägungen vom Typ B in Verbindung treten dürfen. In diesem Fall gilt, dass seitens der Typ-A-Instanzen n Teilnahmen und seitens der Typ-B-Instanzen m Teilnahmen an Beziehungstypinstanzen erlaubt sind.

Die Variablen n bzw. m können dabei durch konkrete Zahlenwerte bzw. Bereichsangaben ersetzt werden und erlauben so genauere Einschränkungen, die auch als Kardinalitäten bezeichnet werden.

3.3.4.4 Unäre Beziehungen

Unäre Beziehungen bezeichnen Beziehungen zu Entitäten desgleichen Typs. Diese Art der Beziehung wird auch als rekursiv bzw. reflexiv bezeichnet.

3.3.4.5 Binäre Beziehungen

In den meisten Modellierungsszenarien treten binäre Beziehungen auf. Hier nehmen exakt zwei Entitätstypen an einer Beziehung teil und prägen in einem Datenbankzustand Paare von Entitäten aus, die ggf. eindeutig auf zusätzliche Beziehungsattribute zeigen.

3.3.4.6 Beziehungen höheren Grades

Bei Beziehungen höheren Grades werden in einem Datenbankzustand n-Tupel ausgebildet, deren Komponenten aus Instanzen der beteiligten Entitätstypen bestehen. Auch n-äre Beziehungen können Beziehungsattribute besitzen.

3.3.5 Kardinalitäten

Der Begriff Kardinalität bezeichnet mathematisch die Mächtigkeit einer Menge. Kardinalitäten machen Aussagen darüber, wie Entitäten quantitativ zueinander in Beziehung stehen dürfen bzw. an wie vielen Beziehungen einzelne Entitäten teilnehmen dürfen. Auf der Ebene des konzeptionellen Modells werden keine spezifischen Objekte, sondern Typen notiert, wie bereits erwähnt wurde. Diese Typen können als Mengen betrachtet werden, deren Elemente alle potentiellen Instanzen beinhalten, für die mengenmäßige Integritätsbedingungen aufgestellt werden können.

3.3.6 Rollen

Entitäten, die miteinander in Beziehung stehen, können über Aliase angesprochen werden, um sie z. B. bei Mehrfachbeziehungen unterscheiden zu können. Diese Aliase werden im Rahmen der Datenmodellierung auch als Rollen bezeichnet.

3.4 Konzeptuelle Modelle

Der Prozess der Datenmodellierung beginnt mit der Anfertigung von konzeptuellen Modellen, die die betrachtete Wirklichkeit abbilden. Es gibt eine ganze Reihe von grafischen Sprachen, die für diese Tätigkeit eingesetzt werden können und sich in ihrer Syntax und Ausdrucksstärke unterscheiden. Neben den Standardsprachen zur konzeptuellen Modellierung wie ER und UML seien in folgender Tabelle 3.1 zwei weitere Vertreter genannt.

Die folgenden Abschnitte betrachten die ER- bzw. UML-Modellierung ausführlicher, da diese für das Entwurfskapitel von grundlegender Bedeutung sind.

Tabelle 3.1: Übersicht Sprachen für konzeptuelle Modellierung

Kurzname	Name	Entwickler	Veröffentlichung
ER	Entity-Relationship-Modell	Peter Chen	1976
UML	Unified Modeling Language	Object Management Group (OMG)	1997
IDEF1X	Integration Definition for Information Modeling	US Air Force	1981–1985
IE	Information Engineering	Clive Finkelstein, James Martin	1976-1980

3.4.1 Das Entity-Relationship-Modell

Das ER-Modell wurde 1976 von Peter Chen veröffentlicht [sC76] und steht für eine der bekanntesten Modellierungssprachen. Neben der ursprünglichen Fassung gibt es einige Erweiterungen wie die *Chen-Notation*, die *Strukturierte Entity-Relationship-Modellierung* (SERM) oder das *erweiterte ER-Modell* (EER).

3.4.1.1 Grundlegende Sprachmittel

Zu den grundlegenden Sprachmitteln gehören jene Syntaxelemente, die Entitätstypen, Beziehungen und Attribute repräsentieren. Diese Modellierungsartefakte wurden eingangs bereits erläutert.

Abbildung 3.2 zeigt ein einfaches ER-Diagramm mit den Entitätstypen *Kunde* und *Hotelzimmer* sowie einem Beziehungstyp *Buchung*. Die Konzepte Kunde und Hotelzimmer besitzen die Attribute *Name* und *Nummer*. Das Beziehungsattribut *Preis* wird dem Konzept der Buchung zugeordnet. Dieses Diagramm drückt aus, dass ein Kunde ein Hotelzimmer zu einem bestimmten Preis buchen kann.

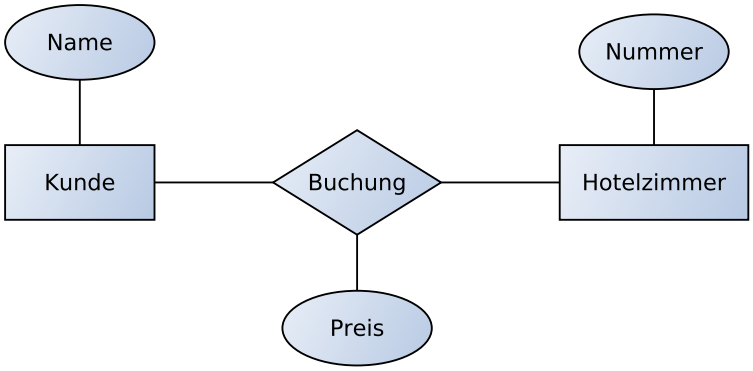

Abbildung 3.2: Einfaches ER-Diagramm

3.4.1.2 Erweiterte Sprachmittel

Das aufgeführte Minimalbeispiel drückt zwar die wesentlichen Aspekte aus, aber es fehlt an wichtigen Angaben beispielsweise zu Beziehungskardinalitäten. Abbildung 3.3 führt deshalb weitere Sprachmittel ein, wie sie ebenfalls eingangs erwähnt wurden.

Das Diagramm erweitert das einführende Beispiel zunächst um ein strukturiertes Attribut *Name*, das in die Unterattribute *Vorname* und *Nachname* aufgespalten wird. Weiterhin wird das Attribut *Telefonnummer* in ein multiples Attribut überführt, da ein Kunde mehrere Telefonnummern besitzen kann. Diesem wird außerdem eine eindeutige Kundennummer (*KNr*) als Schlüsselattribut zugewiesen.

Die Beziehung *Buchung* unterliegt jetzt quantitativen Einschränkungen. Die Kardinalitätsangaben sagen aus, dass ein existierender *Kunde* höchstens eine Buchung vornehmen kann. Bucht er zu einem bestimmten Zeitpunkt kein Zimmer, existiert er weiterhin. Ein *Hotelzimmer* kann ebenfalls maximal von einem Kunden belegt sein oder aber leer stehen.

Jedes Hotelzimmer erhält über eine spezielle *hat*-Beziehung eine *Einrichtung*, die nur existiert, falls das Zimmer existiert. Aus diesem Grund wurde das Konzept der Einrichtung als schwacher Entitätstyp modelliert. Weiterhin gilt, dass ein Hotelzimmer auch keine Einrichtung haben kann, falls es noch nicht entsprechend ausgestattet wurde.

Das Konzept des Kunden unterliegt außerdem einer Generalisierungs- bzw. Spezialisierungsbeziehung zum Supertyp *Person*, der in diesem Beispiel das Namensattribut vererbt. Diese Beziehung wird über eine spezielle Beziehung realisiert, die mit „ist“ benannt wird.

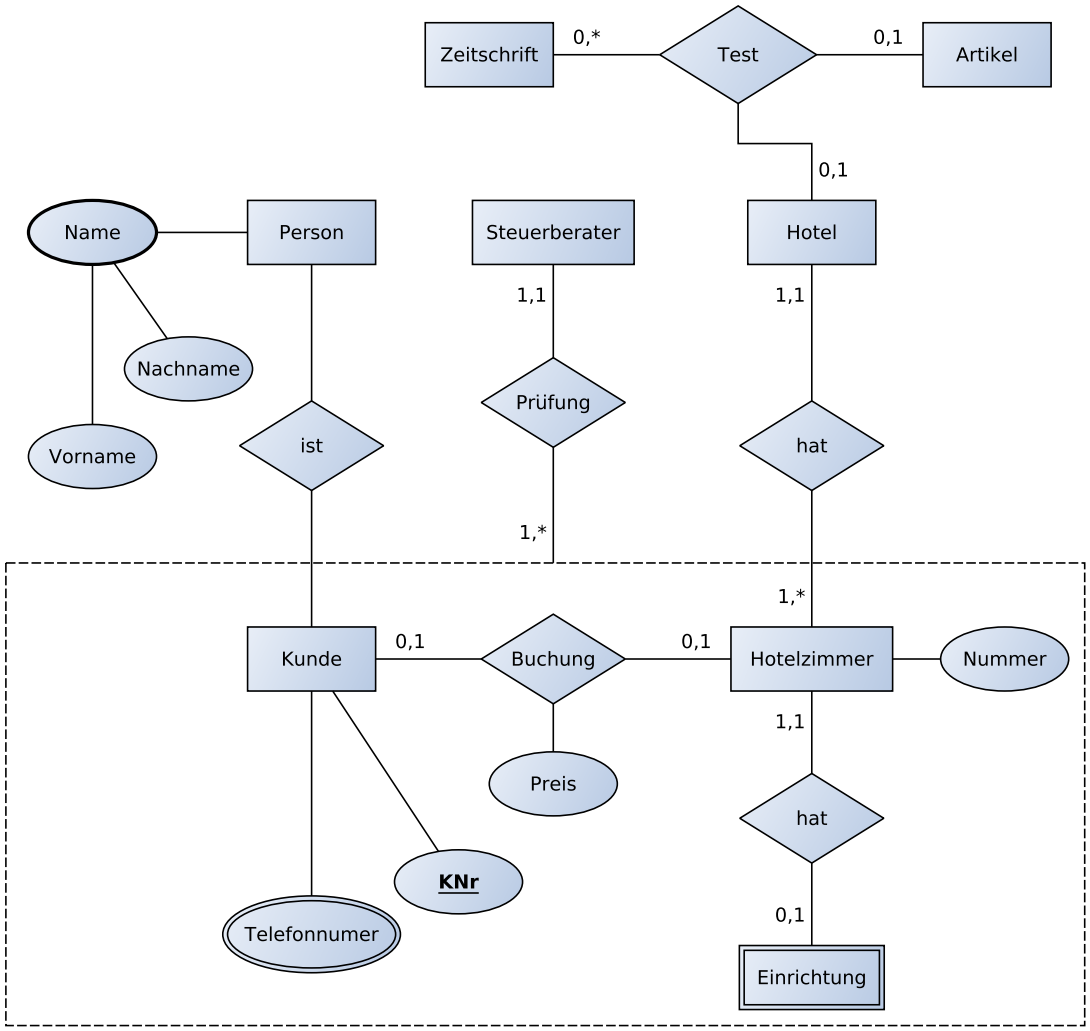

Abbildung 3.3: Erweitertes ER-Diagramm

Als weiteres Konzept wird der *Steuerberater* eingeführt, der die Buchungsvorgänge prüft. Hierzu hält dieser eine Verbindung über einen entsprechenden Beziehungstyp *Prüfung*. Die Notation verweist auf die Besonderheit, dass hier ein Beziehungsteilnehmer ebenfalls eine Beziehung darstellt und sich so aus den Beziehungsmengen der Entitätstypen *Kunde* und *Hotelzimmer* zusammensetzt. Als Integritätsbedingung wird festgelegt, dass ein Steuerberater in der Regel mehrere Buchungen prüft, jede Buchung jedoch nur von einem Steuerberater geprüft wird.

Ein Hotel aggregiert mehrere Hotelzimmer, was über die „hat“-Relation ausgedrückt wird. Ein Hotelzimmer muss stets einem Hotel angehören, während ein Hotel mindestens mit einem Zimmer ausgestattet sein muss.

Bisher wurden nur binäre Beziehungen betrachtet. Als Beispiel für eine höherwertige Beziehung wurde der Beziehungstyp *Test* eingeführt, der die Entitätstypen *Hotel*, *Zeitschrift* und *Artikel* miteinander verbindet. Die Kardinalitätsangaben sagen aus, dass (a) ein Artikel in einer Zeitschrift über den Test eines Hotels handelt, (b) eine Zeitschrift über ein getestetes Hotel in Form eines Artikels berichtet, und dass (c) ein Artikel über den Test eines Hotels von mehreren Zeitschriften gedruckt werden kann.

3.4.2 Das UML-Klassendiagramm

Die UML unterstützt eine Vielzahl von Diagrammtypen, die sich in Struktur- und Verhaltensdiagramme einteilen lassen. Für die Datenmodellierung wird typischerweise das UML-Klassendiagramm eingesetzt, das zu der Gruppe der Strukturdiagramme gehört. Die Übersetzung der ER-Modelle im letzten Abschnitt soll eine Übersicht über die Ausdrucksfähigkeit dieses Diagrammtyps geben.

3.4.2.1 Grundlegende Sprachmittel

Das in Abbildung 3.4 abgebildete UML-Klassendiagramm kann als eine mögliche Übersetzung des Modells in Grafik 3.2 angesehen werden. Sowohl für Entitätstypen als auch für Beziehungstypen wird das Sprachmittel der UML-Klasse verwendet. Über die Angabe entsprechender Stereotypen wird der Kontext zur ER-Modellierung hergestellt.

Die Klassen *Kunde* sowie *Hotelzimmer* werden über eine UML-Assoziationskante miteinander verbunden, um die Beziehung kenntlich zu machen. An dieser Kante hängt eine sogenannte Assoziations- oder auch Schnittklasse, die nach dem Beziehungstyp benannt ist. Die benötigten Attribute werden direkt als Bestandteile der Klassen notiert.

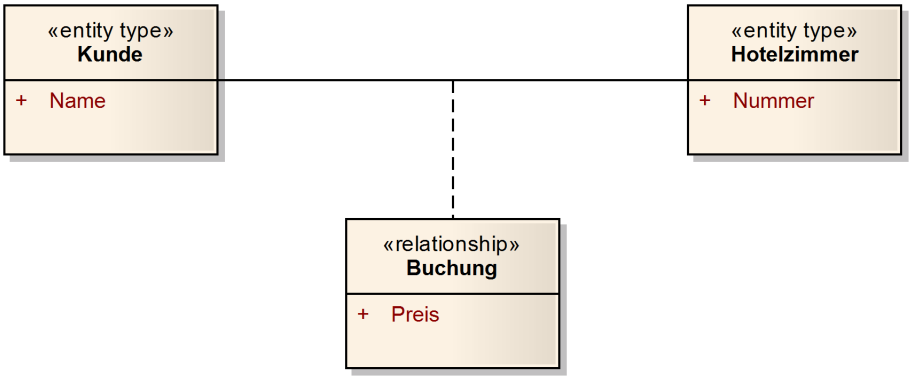

Abbildung 3.4: Einfaches UML-Diagramm

3.4.2.2 Erweiterte Sprachmittel

Die weiteren Notationsmöglichkeiten sollen im Zuge einer möglichen Übersetzung des erweiterten ER-Diagramms 3.3 vorgestellt werden, die in Grafik 3.5 abgebildet ist.

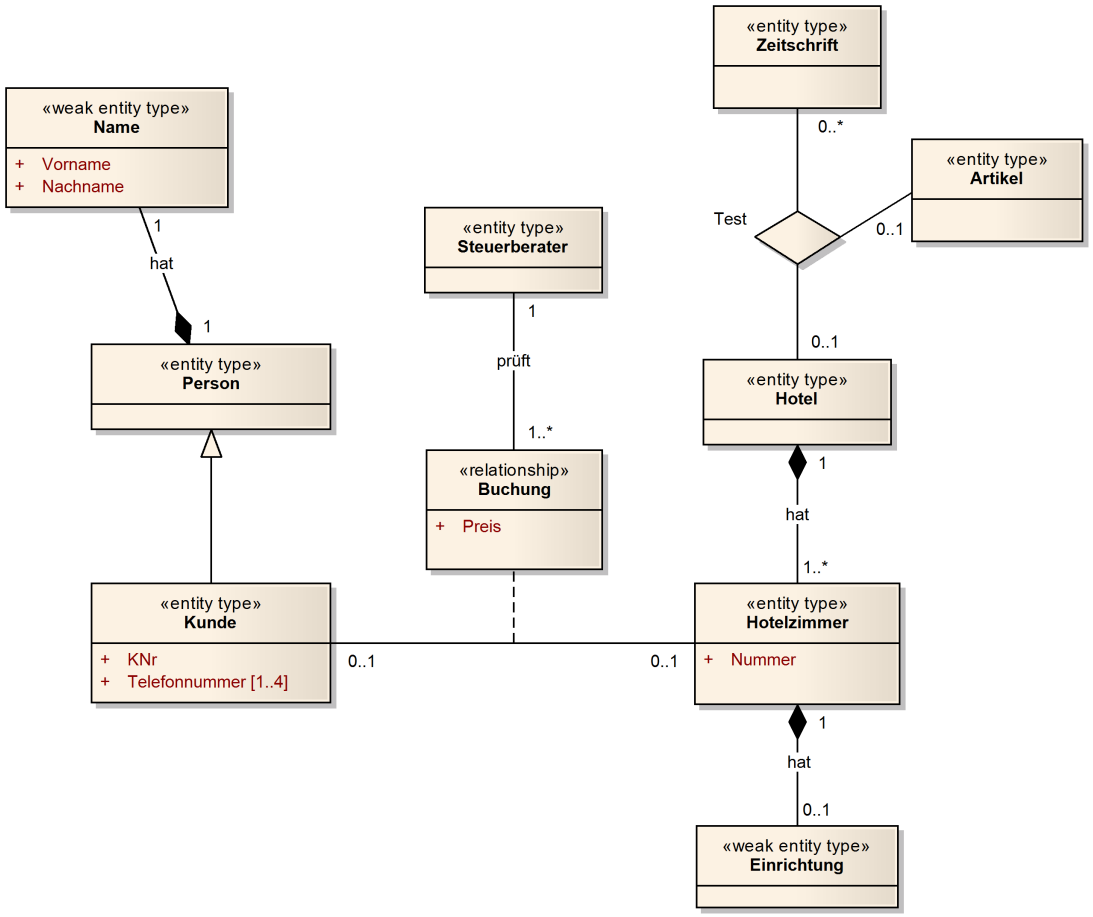

Abbildung 3.5: Erweitertes UML-Diagramm

Das Attribut Name wird in diesem Beispiel als eigene Klasse modelliert, die mit dem Stereotyp *weak entity type* versehen wird. Diese Klasse steht in einer 1:1-Kompositionsbeziehung mit der Klasse Person. Die Komposition ist in der UML ein Spezialfall der Assoziation und drückt eine existenzielle Bindung zwischen den beteiligten Konzepten aus. Ein Kunde hat in diesem Fall exakt einen Namen, der nur solange existiert, wie auch der Kunde. Auf der anderen Seite ist jeder Name exakt einem Kunden zugeordnet.

In diesem Zusammenhang soll auch die Aggregation als weiterer Spezialfall der Assoziation betrachtet werden. Angenommen die Modellierung berücksichtigt, dass Kunden mit gleicher Namensgebung existieren, so kann die Kompositionsbeziehung in eine Aggregationsbeziehung umgewandelt werden, indem das Konzept *Name* als starker Entitätstyp repräsentiert und ein 1:n-Beziehungstyp in Richtung Kunde eingeführt wird.

Die Klasse *Kunde* besitzt neben der eindeutigen Kundennummer (*KNr*) das multiple Attribut *Telefonnummer*, das als Kollektionsattribut unter Angabe einer oberen und unteren Schranke modelliert wird. Die Multiplizitätsangabe sagt aus, dass mindestens eine und maximal beliebig viele Telefonnummern erfasst werden können, wobei keine Duplikate existieren dürfen.

Die Beziehung *Buchung* wird wie gehabt über eine Assoziations- bzw. Schnittklasse modelliert und nimmt direkt an der Beziehung *prüft* teil, die als Assoziation mit der Klasse *Steuerberater* modelliert wird.

Die *hat*-Beziehungen zwischen *Hotelzimmer* und *Einrichtung* bzw. zwischen *Hotel* und *Hotelzimmer* werden in der UML-Darstellung als Kompositionsbeziehung repräsentiert. Die Kardinalitätsangaben bleiben wie auch bei den zuvor betrachteten Assoziationen erhalten.

Für die Darstellung von ternären bzw. höhergradigen Beziehungen bietet die UML ein eigenes Darstellungsmittel in Form einer Raute, an der alle Teilnehmer per Assoziationskante angebunden sind. Dieses Darstellungsmittel wird für die Abbildung der Beziehung *Test* bei gleichbleibenden Kardinalitätsangaben verwendet.

3.5 Logische Modelle

Konzeptuelle Modelle werden in der Regel in logische Modelle übersetzt, die auf die interne Darstellung der gewählten Datenbank eingehen. So werden bei der relationalen Modellierung beispielsweise ER- bzw. UML-Diagramme in Relationenschemata übertragen, die die modellierten Sachverhalte in Form von Tabellen wiedergeben. Logische Modelle werden auch als Datenbankmodelle bezeichnet. Neben dem relationalen Modell gibt es weitere Modelle, die für die logische Darstellung verwendet werden.

Als erstes wichtiges Datenbankmodell ist das sogenannte *hierarchische Datenbankmodell* zu nennen, das zur Darstellung der Domäne eine Baumstruktur nutzt. Es wurde in den 1960er Jahren bei der Firma *North American Rockwell* aufgrund der

Feststellung entwickelt, dass die Speicherung von Daten über ein Dateisystem ineffizient ist, da sie zu einer hohen Redundanz führt.

Eine wesentliche Schwäche des hierarchischen Modells stellt die Tatsache dar, dass es keine Möglichkeit zur direkten Abbildung von n:m-Beziehungen besitzt. Diese Einschränkung hebt das sogenannte *Netzwerkdatenbankmodell* auf, indem es zulässt, dass Knoten *mehrere* Vorgänger besitzen können. Das Netzwerkmodell wurde 1971 im Zuge der *Conference on Data Systems Languages* (CODASYL) definiert.

Ein Jahr zuvor veröffentlichte Edgar F. Codd seine Idee für eine *relationale Datenbank*, die aufgrund von Hardware-Beschränkungen erst 1978 von IBM realisiert werden konnte. Dieses Modell abstrahiert Sachverhalte einer Anwendungsdomäne auf Tabellen und ermöglicht deren Verknüpfung durch Verbundoperationen.

Im Gegensatz zum hierarchischen Modell bzw. zum Netzwerkmodell liegen diese Verknüpfungen lediglich zur Abfragezeit vor, d. h. sie werden nicht mithilfe der verwendeten Datenstruktur explizit dargestellt. Dies führt zu einer strukturellen Unabhängigkeit der Daten und kann einerseits als ein Vorteil des relationalen Systems angesehen werden. Andererseits kann die strukturelle Unabhängigkeit auch zum Nachteil werden, wenn sich beispielsweise Beziehungssemantik in Abfragen verlagert und der Bezug zum konzeptuellen Modell nicht direkt durch die Datenstruktur hergestellt wird.

Relationale Datenbanken speichern Informationen in einer flachen Darstellung. Aus diesem Grund gilt die Speicherung von Objektstrukturen als problematisch. Dieser objektrelationale Bruch wird insbesondere dann deutlich, wenn Informationssysteme nach dem objektorientierten Paradigma implementiert sind. Aus diesem Grund wurde das Konzept der *objektorientierten Datenbank* eingeführt, das die Speicherung von komplexen Objekten inklusive ihrer Attribute und Methoden erlaubt. Außerdem gehen objektorientierte Datenbanken in der Regel auf weitere objektorientierte Konzepte wie Vererbung, Polymorphismus, etc. ein.

Ein wesentlicher Nachteil von objektorientierten Datenbanken ist der größere Verwaltungsaufwand beispielsweise beim Navigieren zwischen Objekten. Der erhöhte Verwaltungsaufwand führt insbesondere im Vergleich zu relationalen Datenbanken zu erheblichen Geschwindigkeitsproblemen.

Eine Lösung für dieses Problem stellen sogenannte *objektrelationale Datenbanken* dar, die als eine Mischung von relationalen und objektorientierten Datenbanken angesehen werden können. Bei dieser Darstellung wird die Trennung zwischen Daten und Fachlogik wieder hergestellt, indem die Daten separat in relationaler Form gespeichert werden.

Eine weitere Möglichkeit zur Speicherung von Daten bieten sogenannte *graphbasierte Datenbanken*. Diese Form der Datenbanken nutzen Graphen als logische Datenstruktur und weisen damit Ähnlichkeiten zum Netzwerkmodell auf. Wie Daten konkret dargestellt werden, hängt davon ab, welche mathematische Definition eines Graphen verwendet wird und welche Aspekte Knoten respektive Kanten repräsentieren. Graphbasierte Datenbanken zählen ebenso wie Objektdatenbanken zu der Klasse der sogenannten *NoSQL-Datenbanken*, die nicht relational arbeiten.

Einige wichtige Fragen bei der Definition eines graphbasierten Modells sollen an dieser Stelle aufgelistet werden.

- Handelt es sich um einen gerichteten bzw. ungerichteten Graph oder eine Kombination?
- Besitzt der Graph Mehrfachkanten, Schleifen bzw. Zyklen?
- Besitzen alle Knoten bzw. Kanten den gleichen Typ?
- Handelt es sich um einen attributierten Graphen, d. h. besitzen Knoten bzw. Kanten Metainformationen?
- Wie werden die zu speichernden Daten auf Knoten bzw. Kanten abgebildet?

Je nach Beantwortung dieser Fragen entstehen sehr unterschiedliche graphbasierte Modelle. Es existiert kein Standard, der festlegt, auf welche Art eine Graphdatenbank Daten speichern muss. Eine sehr ausführliche Übersicht bietet die Arbeit von Renzo Angles, die existierende graphbasierte Datenbanksysteme anhand von über 40 Kriterien vergleicht [Ang12].

4 Entwurf

4.1 Einbettung in Schichtenarchitektur

Grafik 4.1 zeigt das Schema einer klassischen 3-Schichten-Architektur [Bal99, S. 372], wie sie auch für Informationssysteme eingesetzt wird. Die erste sogenannte Repräsentationsschicht (engl., Sg.: *representation layer*) ermöglicht die Ein- und Ausgabe von Daten und wird in der Regel als grafische Benutzungsschnittstelle (engl., Sg.: *graphical user interface*) realisiert. Die zweite Schicht beinhaltet die Steuerungs- sowie Geschäftslogik und wird als Applikationsschicht (engl., Sg.: *application layer*) bezeichnet. Die Datenhaltungsschicht (engl., Sg.: *data access layer*) ist für die persistente Ablage der zu verarbeitenden Informationen verantwortlich und deshalb für dieses Kapitel von besonderem Interesse.

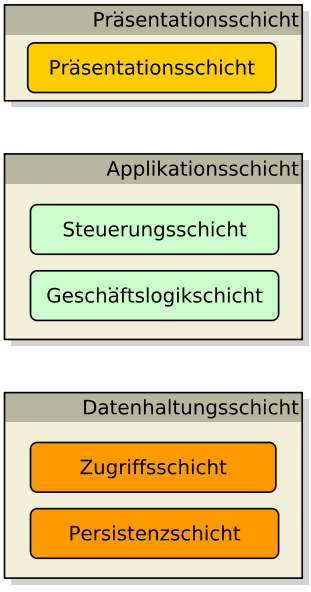

Abbildung 4.1: Klassische 3-Schichten-Architektur

4.2 Strategien zur Datenspeicherung

Zunächst soll die Frage geklärt werden, warum Informationen überhaupt *gespeichert* werden müssen und welche Strategien hierfür grundsätzlich zur Verfügung stehen.

Zur Laufzeit eines Programmes befinden sich sowohl dessen Code als auch die zu verarbeitenden Daten im Arbeitsspeicher und sind damit temporär. Eine Unterbrechung der Stromversorgung führt sofort zu einem vollständigen Verlust aller Daten im Arbeitsspeicher. Während Programme, die in der Regel auf Festspeichern hinterlegt sind, einfach erneut geladen werden können, gilt dies für die verarbeiteten Daten nur dann, wenn diese ebenfalls explizit auf einem entsprechenden Speicher abgelegt worden sind. Daten dürfen also nicht verloren gehen, wenn das entsprechende EDV-System ausgeschaltet wird.

Für die *dauerhafte* Ablage von Daten gibt es verschiedene Strategien. Die erste Strategie ist die *unstrukturierte* Speicherung in Form von Binär- bzw. Textdateien. Diese Strategie gilt als unflexibel, da die hinterlegten Daten in einem proprietären Format vorliegen und in der Regel nur durch genau jene Programme interpretiert werden können, die sie auch abgelegt haben.

Eine weitere Möglichkeit bietet die *semi-strukturierte* Speicherung von Daten, wie sie zum Beispiel XML (engl., Sg.: *Extensible Markup Language*) zulässt. In diesem Fall werden die Informationen zerlegt und in Textdateien als Baumstruktur abgelegt, sodass auch Hierarchien ausgebildet werden können. Die einzelnen Bestandteile können über semantisch benannte Abgrenzer (engl., Sg.: *tag*) differenziert und interpretiert werden. Aufgrund dieser einfachen Lesbarkeit bieten XML-Dateien einen wesentlich flexibleren Umgang. Am Rande sei bemerkt, dass XML mit einer entsprechenden Dokumenttypdefinition (engl., Sg.: *DTD, Document Type Definition*) ebenfalls eine strukturierte Speicherung zulässt [NH00, S. 114ff.].

Weder die unstrukturierte noch die semi-strukturierte Speicherung stellt Anforderungen an die Datenstruktur oder erhebt Ansprüche an die Integrität der abzulegenden Daten. Auch die Art, wie Daten angelegt, ausgelesen oder verändert werden, bleibt unreguliert. Aufgrund dieser Unsicherheiten ist es kaum möglich, ein Standardmodell darüber zu formulieren, wie Daten persistent gemacht werden können. Ein solches Standardmodell wird allerdings für den Aufbau einer Datenbank benötigt, weshalb auf die *strukturierte* Speicherung von Daten zurückgegriffen wird. Notwendige Bedingung für diese Strategie ist ein sogenanntes Datenbankmodell, das die angeführten Aspekte aufgreift und Lösungen bereitstellt.

Ein Informationssystem, mit dem zum Beispiel Unternehmensdaten in einer Datenbank abgelegt werden sollen, benötigt also ein entsprechendes Datenbankmodell, das sowohl die zu verwendende Datenstruktur als auch notwendige Integritätsbedingungen und die verfügbaren Operationen zur Manipulation des Datenbankzustandes definiert. So sieht das relationale Datenbankmodell zum Beispiel die Speicherung der Informationen in Form von Tabellen vor und bietet sowohl für die Formulierung von Integritätsbedingungen als auch für die Durchführung von Datenmanipulation die Verwendung von SQL (engl., Sg.: *Structured Query Language*) an.

SQL als sogenannte *Datenbanksprache* zerfällt dabei in verschiedene Sprachbereiche, die jeweils für einen bestimmten Aufgabenbereich konzipiert sind [MS05, S. 9]. Grundsätzlich lässt sich unabhängig vom verwendeten Datenbankmodell ein Bedarf von mindestens 2 Sprachbereichen formulieren. Die Datendefinitionssprache (engl., Sg.: *DDL, Data Definition Language*) ermöglicht die Beschreibung der Datenstruktur und deren Elemente. Die Datenmanipulationssprache (engl., Sg.: *DML, Data Manipulation Language*) wird eingesetzt, wenn Daten abgefragt, eingefügt, verändert oder gelöscht werden sollen. Einen weiteren Sprachbereich stellt die Datenüberwachungssprache (engl., Sg.: *DCL, Data Control Language*) dar, die eingesetzt wird, um Datenzugriffsrechte zu verwalten. Die Unterstützung einer Datenüberwachungssprache ergibt allerdings nur Sinn, wenn die Datenbank grundsätzlich über ein System für die Vergabe von Zugriffsrechten verfügt.

Der nun folgende Entwurf eines integrierten Datenbankmodells muss also sowohl die Aspekte eines konzeptuellen Modells als auch die in diesem Abschnitt benannten Sachverhalte für ein logisches Modell berücksichtigen. Er gliedert sich dabei in zwei Abschnitte.

Im ersten Abschnitt wird der Typgraph eingeführt, der als Datenbankschema einerseits die Datendefinition und andererseits die Formulierung von Integritätsbedingungen ermöglicht. Im zweiten Abschnitt wird darauf eingegangen, wie anhand des Schemas ein entsprechender Datenbankzustand aufgebaut werden kann.

4.3 Modellsichten

Im Vorfeld wurde die Idee beschrieben, das 3-Phasen-Modell zu ersetzen und an dessen Stelle ein *integriertes* Datenmodell einzuführen, das sowohl als konzeptuelles als auch als logisches und in seiner textuellen Form zuletzt auch als physisches Modell eingesetzt werden kann.

Um den unterschiedlichen Detailansprüchen bei der Modellierung Rechnung zu tragen, ist es sinnvoll, mindestens zwei Sichten (engl., Sg.: *view*) auf den Typgraph zu definieren, die dem Verständnisgrad der Betrachter Rechnung tragen. So kann es eine einfache Sicht für die Anforderungsdefinition beim Kunden geben und eine komplexe Sicht, in der Entwickler zum Beispiel detaillierte Integritätsbedingungen darstellen können.

Das Konzept der Sichten ist dabei flexibel, sodass außerdem die Möglichkeit besteht, weitere Sichten variabler Informationsdichte in Abhängigkeit von der Erfahrung des Kunden einzuführen. So könnten je nach Domäne, Unternehmenskontext und Erfahrungshorizont der Akteure bestimmte Modellierungsaspektgruppen ein- oder ausgeblendet werden.

Wichtig hierbei ist zu erkennen, dass es sich bei dem Wechsel zwischen den Sichten nicht um eine Transformation des Modells handelt, da die Unterschiede zwischen den Sichten im Informationsgehalt bestehen. Zum Aufbau vereinfachter Darstellungen sind lediglich Veränderungen am Modell notwendig, die sich allerdings im Rahmen derselben Modellsprache abspielen und daher weit weniger Aufwand erfordern als die Transformation in eine gänzlich andere Repräsentation.

Um es in der Terminologie der modellgetriebenen Softwareentwicklung zu formulieren, handelt es sich bei den Übergängen zwischen den Sichten um sogenannte Modellmodifikationen (engl., Sg.: *Inplace Transformation*), bei der lediglich Änderungen am Modell vorgenommen werden, ohne dass ein neues Modell entsteht und explizit keine *Modell-zu-Modell-Transformationen* (M2M) stattfinden.

In der Regel kann davon ausgegangen werden, dass Modellmodifikationen weniger aufwendig sind, da keine Übersetzung von einer Modellsprache in eine andere erfolgt. Damit bricht die Einführung von Sichten nicht mit der in dieser Arbeit formulierten Maxime, den Modellierungsprozess insgesamt zu vereinfachen.

4.4 Darstellung von Graphen im Hauptspeicher

Für die Definition des Typgraphen wurde eine Datenbeschreibungssprache entwickelt, die die Graphinformation auf textueller Ebene beschreibt. Wir erhalten so eine textuelle Abstraktion des Typgraphen, die in einem dauerhaften Speicher abgelegt werden kann. Der tatsächliche Betrieb einer Datenbank bedarf der Repräsentation des Typgraphen im Hauptspeicher, daher muss es eine Abbildung der textuellen Abstraktion auf die Ebene der verwendeten Datenstrukturen der eingesetzten Programmiersprache geben.

Auch für die Ausprägung von Instanzgraphen wird eine entsprechende Repräsentation der Zustandsdaten im Hauptspeicher benötigt.

Da es sich bei der verwendeten Programmiersprache Ruby um eine Hochsprache handelt, stellt die Repräsentation der Graphstruktur wiederum nur ein Zwischenmodell dar, das wesentlich abstrakter ist als die tatsächliche Repräsentation von Graphen zum Beispiel auf Ebene der Maschinensprache.

An dieser Stelle soll die Repräsentation von Graphen auf Ebene der Datenstruktur einer Hochsprache betrachtet werden. Ohne auf die speziellen Eigenschaften von Typ- bzw. Instanzgraphen einzugehen, gibt es grundsätzlich die Möglichkeit der *impliziten* und *expliziten* Darstellung von Graphstrukturen.

Die implizite Darstellung kann zum Beispiel über den Einsatz von Adjazenz- bzw. Inzidenzmatrizen geschehen. Bei dieser Art der Darstellung kann über die Angabe von Knoten- und Kantenkoordinaten ermittelt werden, ob eine Kante existiert. Eine weitere Möglichkeit ist die Darstellung über sogenannte Adjazenzlisten. Matrizen können dabei als zweidimensionale Arrays und Adjazenzlisten als einfach oder mehrfach verkettete Liste realisiert werden.

Die explizite Darstellung erfolgt durch die Modellierung einer entsprechenden Datenstruktur, die an sich einen Graph darstellt und Knoten sowie Kanten direkt als Konzepte dekomponiert. Je nach Konstruktion halten die Knoten- bzw. Kanteninstanzen Referenzen auf die benachbarten Elemente sowie weitere Metainformationen.

Der Typgraph, der im Rahmen dieser Arbeit konstruiert wird, wird zur Laufzeit explizit dargestellt, d. h. sowohl Knoten und Kanten werden als eigenständige Konzepte behandelt. Der Instanzgraph wird nur zum Teil explizit dargestellt, da seine Kanten über einfache Speicherreferenzen modelliert werden.

4.5 Allgemeine Betrachtung zur Generalisierung

Eine Grundgesamtheit von Entitäten lässt sich mittels einer Klassifikation zerlegen, indem für die beteiligten Objekte bestimmte Typen definiert werden. Diese Typen werden in der Regel anhand ausgesuchter Merkmale abgeleitet und als abstrakte Konzepte für die Instanziierung von konkreten Ausprägungen verwendet. Ziel einer Klassifikation ist in der Regel der Aufbau einer Ordnung und dadurch die Vereinfachung von Zugriffen auf ganze Objektgruppen.

Typen bzw. Klassen können auch als abstrakte Konzepte wiederum gleiche Merkmale tragen, sodass Supertypen bzw. Oberklassen ausgebildet und Bestandteil einer

hierarchischen Klassifikationsstruktur werden können. Die Übertragung von Merkmalen einer höheren Klassenebene auf eine niedrigere nennt man dabei im Allgemeinen *Vererbung*. Hierbei gilt es zwischen Mono- und Polyhierarchie zu unterscheiden. Eine Monohierarchie gestattet lediglich die Vererbung von *einer* Oberklasse, während eine Polyhierarchie Mehrfachvererbung zulässt.

Das Herabsteigen einer Klassifikations- bzw. Vererbungshierarchie über die subordinierten Abstraktionsebenen kann allgemein als *Spezialisierung* bezeichnet werden, das Aufsteigen über ranghöhere Ebenen als *Generalisierung*.

Nun gibt es bei der Bildung von Oberklassen, die in der Informatik auch als Superklassen, Supertypen oder Basisklassen bezeichnet werden, ein Problem bezüglich der Semantik von Klassifikationsstrukturen, da die objektorientierte Modellierung im Allgemeinen nicht zwischen *genetischer* und *phänotypischer* Vererbung unterscheidet.

Bei der genetischen Klassifikation erfolgt die Einteilung über die tatsächliche Verwandtschaft, während die phänotypische Zuordnung das äußere Erscheinungsbild als Klassifikationskriterium berücksichtigt. Wie wir am folgenden Beispiel sehen werden, kann dies zu semantisch inkonsistenten Typhierarchien führen und die Verständlichkeit eines Modells mindern. Abbildung 4.2 zeigt ein UML-Klassendiagramm, das das Problem verdeutlicht.

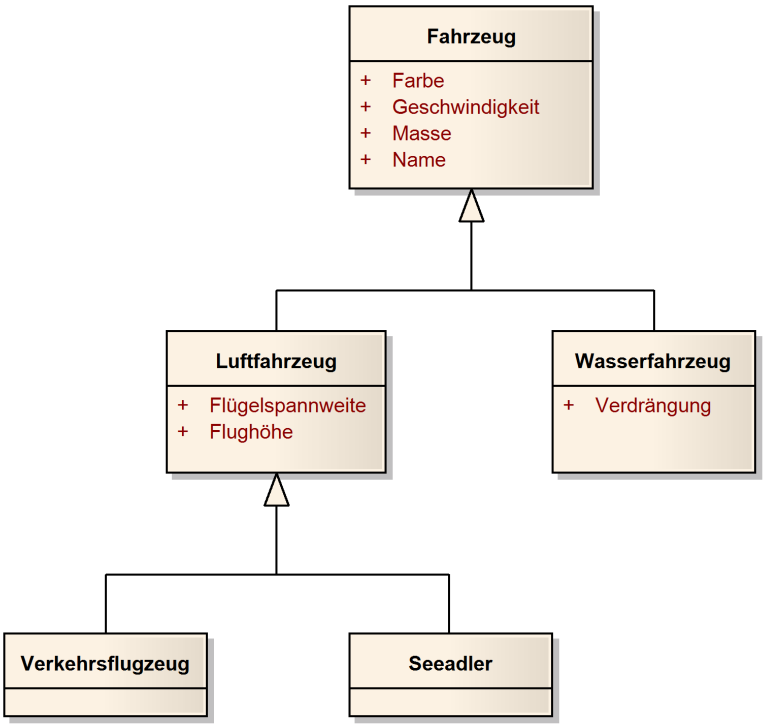

Abbildung 4.2: UML-Diagramm phänotypische Vererbung

Für das Beispiel wurde zunächst eine Oberklasse mit der Bezeichnung *Fahrzeug* angelegt. Von dieser Klasse erben jeweils die Konzepte *Luftfahrzeug* und *Wasserfahrzeug* die aufgeführten Merkmale. Die Klasse *Luftfahrzeug* dient wiederum als Oberklasse für die beiden Typen *Verkehrsflugzeug* und *Seeadler*.

An dieser Stelle sei auf das zuvor beschriebene Problem hingewiesen. Aus rein technischer bzw. phänotypischer Sicht ist an der Kategorisierung von *Seeadler* als *Luftfahrzeug* nichts auszusetzen, denn schließlich trägt ein Seeadler die angegebenen Merkmale. Allerdings macht diese Strukturierung aus semantischer bzw. genetischer Perspektive keinen Sinn, da ein Seeadler kein Luftfahrzeug darstellt. An dieser Stelle ist ein Bruch mit dem Anspruch an eine adäquate Abbildung der Domäne zu erkennen.

Um den Typ *Seeadler* als Ebene einer genetischen Klassifikationsstruktur zu modellieren, müssen Oberklassen eingeführt werden, deren Namen semantisch korrekt sind und die die entsprechenden Begriffe verwenden. So könnten zum Beispiel weitere Oberklassen *Vogel* und *Wirbeltier* eingeführt werden. Dieses Vorgehen hat jedoch zur Folge, dass zwei Vererbungshierarchien ausgebildet werden, die exakt dieselben Merkmale anhäufen. Mag es für dieses Beispiel akzeptabel sein, eine

Parallelhierarchie aufzubauen, so lassen sich andere Beispiele finden, bei denen die einzelnen Klassengrenzen weniger scharf abgesteckt werden können und das beschriebene Vorgehen daher ausscheidet.

Das in dieser Arbeit eingeführte Datenmodell kann dieses Problem nicht lösen, es stellt aber Möglichkeiten zur Verfügung, es zu entschärfen, indem unterschiedliche Klassifikationsinstrumente parallel verwendet werden können, um eine Verbesserung der Semantik zu ermöglichen.

So gibt es zunächst die Möglichkeit der Ausbildung einer Typhierarchie mit Mehrfachvererbung. Es können also Entitätstypen ausgebildet werden, die von Supertypen abgeleitet werden, die wiederum Supertypen spezialisieren. Dabei können von jeder Ebene aus zusätzliche Merkmale eingeführt werden. Alle Supertypen gelten dabei als abstrakt, d. h. es können ausschließlich konkret abgeleitete Typen, die Blätter der Hierarchie, instanziiert werden.

Die nächste Möglichkeit stellt die Definition von *Schnittstellen* dar. Typen können beliebig viele Schnittstellen erfüllen, indem sie deren Merkmale erben. Im Gegensatz zu den eingeführten Supertypen können Schnittstellen jedoch ihrerseits nicht erben und bilden keine Hierarchie aus.

Um bei dem Problem der genetischen und phänotypischen Klassifikation eine gewisse Unterstützung zu bieten, ist es möglich, jeden Typ bzw. Supertyp frei einer bzw. mehreren Klassen zuzuordnen, die ebenfalls wieder mittels Oberklassen kategorisiert werden können und so eine Polyhierarchie beliebiger Tiefe ausbilden. Im Gegensatz zu dem Konzept des Supertyps erfolgt dabei keine Vererbung von Merkmalen. Die Einteilung in Klassen dient ausschließlich dem Aufbau einer zusätzlichen Ordnung und der semantischen Abgrenzung.

Als letztes Instrument wird das Clustering eingeführt, das beliebige Typen, Supertypen, Schnittstellen, Klassen und Oberklassen unter einem Namen zusammenfassen kann und besonders beim Einsatz der Aggregation eine interessante Rolle spielt.

Es sei an dieser Stelle angemerkt, dass keine der genannten Klassifikationsmechanismen obligatorisch verwendet werden muss, um ein gültiges Modell zu erhalten.

4.6 Aufbau von Typgraphen

4.6.1 Vorbemerkung

Voraussetzung für die kontrollierte Speicherung strukturierter Daten ist in der Regel ein formales Modell, aus dem hervorgeht, welche Daten auf welche Weise gespeichert werden, welche *Beziehungen* (engl., Sg.: *relationship*) zwischen ihnen bestehen und welche *Einschränkungen* (engl., Sg.: *constraint*) bei der Instantiierung berücksichtigt werden müssen.

Der in diesem Kapitel eingeführte Typgraph (engl., Sg.: *type graph*) stellt ein solches Modell dar, das im Bereich der Informatik und insbesondere im Zusammenhang mit Datenbanken oft als *Schema* (engl., Sg.: *schema*) bezeichnet wird.

Die Repräsentation der schematischen Informationen durch einen Graphen ermöglicht es, Definitionen aus der diskreten Mathematik als Grundlage für die formale Definition des Typgraphen zu verwenden. Um die nachstehenden Erläuterungen verständlicher zu gestalten, werden nachvollziehbare Beispielkonzepte aus der Realität modelliert. Dabei werden die Begriff Schema, Datenbankschema und Typgraph synonym verwendet.

4.6.2 Modellierungsansatz

Im Fokus der Definition eines Typgraphen steht der sogenannte *Entitätstyp* (engl., Sg.: *entity type*) als objektorientierte Abstraktion eines realen Sachverhalts. Auf Basis eines Entitätstyps lassen sich später entsprechende Instanzen ausprägen, sogenannte *Entitäten* (engl., Sg.: *entity*). Entitätstypen können in sogenannten *Entitätstypklassen* (engl., Sg.: *entity type class*) kategorisiert werden. Dieser Zusammenhang wird über eine korrespondierende *Klassifikationsrelation* (engl., Sg.: *classification relation*) $R_{classification}$ definiert.

Für die Modellierung von Entitätstypen werden *Annotationstypen* (engl., Sg.: *annotation types*) benötigt, die Attribute repräsentieren und über eine *Mitgliedschaftsrelation* (engl., Sg.: *membership relation*) $R_{memberhsip}$ verbunden werden. Annotationstypen können untereinander ebenfalls Mitgliedschaften eingehen, um u. a. hierarchisch strukturierte Sachverhalte abzubilden. Hierfür muss die Definition der Mitgliedschaftsrelation geändert werden. Instantiierte Entitätstypen, also Entitäten, besitzen entsprechend instantiierte Annotationstypen, die *Entitätsannotationen* (engl., Sg.: *entity annotation*) genannt werden. Die Begriffe Annotation, Attribut, Eigenschaft

sowie Merkmal werden synonym verwendet und beziehen sich auf das Konzept des Annotationstyps.

Neben der Wiederverwendung von Annotationstypen durch mehrere Entitätstypen wird außerdem eine Möglichkeit zur Vererbung von Attributen gegeben, indem das Konzept des *Superentitätstyps* (engl., Sg.: *super entity type*) eingeführt wird. Ein Superentitätstyp gleicht einem Entitätstyp bis auf die Tatsache, dass keine Instantiierung möglich ist. Dieser Umstand wird kenntlich gemacht, indem Superentitätstypen als *virtuell* (engl., Sg.: *virtual*) ausgezeichnet werden. Superentitätstypen können ihrerseits Attribute erben, wodurch eine Mehrfachvererbung beliebiger Tiefe ermöglicht wird. Mathematisch wird diese Funktionalität über eine *Vererbungsrelation* (engl., Sg.: *inheritance relation*) $R_{inheritance}$ ausgedrückt.

Neben der klassischen Vererbung mit Hilfe von Superentitätstypen können außerdem *Entitätstyp-Schnittstellen* definiert werden, die ebenfalls zusätzliche Attribute liefern und als virtuell gekennzeichnet sind. Hierfür wird die sogenannte *Implementierungsrelation* (engl., Sg.: *implementation relation*) $R_{implementation}$ eingesetzt. Entitätstyp-Schnittstellen können dabei weder von Supertypen erben noch andere Schnittstellen erfüllen.

Mit Hilfe der *Clustering-Relation* (engl., Sg.: *clustering relation*) $R_{clustering}$ können außerdem Superentitätstypen, Entitätstypen, Entitätstypklassen sowie Entitätstyp-Schnittstellen generisch in benannten *Clustern* (engl., Sg.: *cluster*) zusammengefasst werden.

Der Typgraph beinhaltet neben Informationen über die Typisierung potentieller Instanzen auch Regeln für Beziehungen und Einschränkungen, wie eingangs bereits erwähnt wurde. Diese Regeln werden als Integritätsbedingungen bezeichnet (engl., Sg.: *integrity constraint*) und zerfallen in Bedingungen für Werte (engl., Sg.: *value constraint*) sowie in Regeln für strukturelle Anforderungen (engl., Sg.: *structural constraint*).

Wie sich der Aufbau eines Typgraphen genau gestaltet, soll anhand von Beispielen erläutert werden. Beginnend mit einem minimalen Beispiel werden dabei schrittweise neue konzeptionelle Sachverhalte eingeführt und visualisiert. Eine Übersicht über die Syntaxelemente sei in A gegeben. Diese Auflistung führt nicht alle Kombinationsmöglichkeiten explizit auf.

4.6.3 Übersicht über die Elemente des Typgraphen

Für die Definition eines Schemas können zunächst Knoten der Art Entitätstypklasse verwendet werden. Knoten dieser Art werden entsprechend *Entitätstypklassen-Knoten*

(engl., Sg.: *entity type class node*) genannt. Diese Knoten wiederum fassen Knoten der Art Entitätstyp, im Folgenden als *Entitätstyp-Knoten* (engl., Sg.: *entity type node*) bezeichnet, über eine *Klassifikationskante* (engl., Sg.: *classification edge*) zusammen. Der Entitätstyp-Knoten stellt die Basis für die objektorientierte Modellierung dar. Er bildet dabei gedankliche bzw. konkrete Sachverhalte der Domäne ab, zum Beispiel das Konzept *Vereinsmitgliedschaft* oder *Kunstgegenstand*.

Ein Entitätstyp-Knoten reicht für die Modellierung eines Gegenstandes und seiner Merkmale nicht aus. Hierfür werden Knoten der Art Annotationstyp benötigt, die im Folgenden als *Annotationstyp-Knoten* (engl., Sg.: *annotation type node*) bezeichnet werden. Annotationstyp-Knoten können an durch Entitätstyp-Knoten repräsentierten Konzepten teilnehmen und ermöglichen so deren Modellierung. Diese Teilnahme wird über eine entsprechend dekorierte *Mitgliedschaftskante* (engl., Sg.: *membership edge*) zwischen den jeweiligen Knoten angezeigt. Ein Annotationstyp-Knoten kann so zum Beispiel das Attribut *Tonerzeugung* eines potentiellen Konzeptes *Musikinstrument* repräsentieren. Auch zwischen Annotationstyp-Knoten können Mitgliedschaftskanten existieren, um strukturierte Typen darstellen zu können. Mit der *Dekoration* (engl., Sg.: *decoration*) von Mitgliedschaftskanten ist deren Attributierung mit Informationen gemeint, die entsprechende Teilnahmen qualifizieren.

Die *Wiederverwendung* von Teildefinitionen mittels Vererbung wird über Knoten der Art Superentitätstyp realisiert. Die sogenannten *Superentitätstyp-Knoten* (engl., Sg.: *super entity type node*) sind hierfür über entsprechende *Vererbungskanten* (engl., Sg.: *inheritance edge*) mit den erbenden Entitätstyp-Knoten verbunden oder, bei einer tieferen Vererbungshierarchie, wiederum mit Superentitätstyp-Knoten.

Entsprechend der vorgestellten Modellierung stehen außerdem *Schnittstellen-Knoten* (engl., Sg.: *entity type interface node*) sowie *Cluster-Knoten* (engl., Sg.: *cluster node*) zur Verfügung, die über *Implementierungskanten* (engl., Sg.: *implementation edge*) bzw. über *Clustering-Kanten* (engl., Sg.: *clustering edge*) referenziert werden.

4.6.4 Mathematische Einführung des Typgraphen

An dieser Stelle soll der zu entwerfende Typgraph mathematisch definiert werden. Hierzu wird zunächst die allgemeine Definition eines gerichteten Graphen ohne Mehrfachkanten gegeben.

Definition 4.6.1 (Gerichteter Graph) *Ein gerichteter Graph G ohne Mehrfachkanten ist ein System $G = (V(G), E(G) \subseteq V(G) \times V(G))$ derart, dass $V(G)$*

die Menge der Knoten und $E(G)$ die Menge der Kanten darstellt, die als Teilmenge des kartesischen Produktes auf der Menge $V(G)$ definiert ist.

Diese Definition erlaubt Schleifen sowie Zyklen und dient als Basis für die Einführung eines getypten Graphen bzw. Typgraphen G_T, wie er in dieser Arbeit bezeichnet wird.

Definition 4.6.2 (Typgraph) *Sei $T = (V(T), E(T))$ ein Typbelegungsgraph. Dann ist ein Typgraph ein System $G_T = (G, type)$ derart, dass $G = (V(G), E(G))$ ein gerichteter Graph nach 4.6.1 ist und $type : G \to T$ ein Graphmorphismus, der mit $type_V : V(G) \to V(T)$ und $type_E : E(G) \to E(T)$ die Knotenmenge $V(G)$ auf die Menge der Knotentypen $V(T)$ und die Kantenmenge $E(G)$ auf die Menge der Kantentypen $E(T)$ abbildet, sodass gilt:*

$$type_E((v, v')) = (type_V(v), type_V(v')) \tag{4.1}$$

Nach 4.6.2 kann eine Typrelation R_t eingeführt werden, die als Teilmenge des kartesischen Produktes auf $V(G)$ genau jene Paare beinhaltet, für die eine Abbildung in die Menge der Kantentypen $E(T)$ existiert:

$$R_t = \{(v, v') \in E(G) \mid t \in E(T) \wedge type_E((v, v')) = t\} \tag{4.2}$$

Für die weiteren Abschnitte gilt also, dass bestimmte Kanten nur zwischen bestimmten Knoten existieren dürfen, so wie es die Typbelegung zulässt. Welche Knoten- und Kantentypen existieren und in welcher Beziehung diese stehen dürfen, wird in den nächsten Abschnitten erläutert.

Dabei werden bezüglich des Graphmorphismus alle Knoten mit gleichem Typ und alle Kanten mit gleichem Type betrachtet, woraus sich je eine disjunkte Zerlegung der Knotenmenge und der Kantenmenge ergibt. Umgekehrt definieren solche disjunkten Zerlegungen immer auch einen Graphmorphismen, wenn bei den Kantentypmengen jeweils noch festlegt wird, zwischen welchen beiden Knotentypmengen sie eine Beziehung herstellen.

Für die folgenden Beispiele von Typgraphen wird meist diese Darstellung gewählt. In der graphischen Darstellung von Typgraphen werden die Knotentypen durch gleiche Form repräsentiert und die Kanten erhalten je nach Typ unterschiedliche Linien- bzw. Pfeilspitzen.

4.6.5 Klassifikation

Zur Erläuterung soll ein Schema für die Speicherung von Informationen über Kletterhallen definiert werden. Hierzu legen wir zunächst einen passenden Entitätstypklassen-Knoten *Sportstätte* fest. Über eine entsprechende Klassifikationskante wird diesem Knoten ein Entitätstyp-Knoten *Kletterhalle* zugeordnet.

Die Klassifikationskante wird über die binäre Klassifikationsrelation $R_{classification}$ berechnet, die als Teilmenge des kartesischen Produktes zwischen der Menge der Entitätstyp-Knoten E und der Menge der Entitätstypklassen-Knoten K definiert wird:

$$R_{classification} \subseteq \{(x, y) \mid x \in E \wedge y \in K\} \tag{4.3}$$

Für das aktuelle Beispiel erhalten wir folgende Ausprägung der Klassifikationsrelation $R_{classification}$, wobei die beteiligten Knoten entsprechende Bezeichnungen erhalten, wie sie in Tabelle 4.1 aufgeführt sind. Generell gilt, dass konkrete Beispielmengen indiziert werden, um sie in späteren Berechnungen referenzieren zu können.

Tabelle 4.1: Knotenübersicht Klassifikationsbeispiel

Knotenart	**engl. Bezeichnung**	**Erläuterung**
Entitätstypklasse	`SportsFacility`	repräsentiert Sportstätte
Entitätstyp	`IndoorClimbing`	repräsentiert Kletterhalle

Ausprägung 4.6.1 (Klassifikationsrelation $R_{classification_1}$)

$$E_1 = \{\texttt{IndoorClimbing}\}$$
$$K_1 = \{\texttt{SportsFacility}\}$$
$$R_{classification_1} = \{(\texttt{IndoorClimbing}, \texttt{SportsFacility})\}$$

Die Visualisierung des Schemas beinhaltet die Darstellung des korrespondierenden Typgraphen G_{T_1} und basiert auf der in Abschnitt 4.6.4 gegebenen Definition eines gerichteten Graphen ohne Mehrfachkanten. Ein Typgraph G_T lässt sich dabei wie folgt konstruieren:

Konstruktionsschema 4.6.1 (Typgraph G_T, 1. Definition)

$$\begin{aligned} G_T &= (V(G_T), E(G_T) \subseteq V(G_T) \times V(G_T)) \\ V(G_T) &= E \cup K \\ E(G_T) &= R_{classification} \end{aligned}$$

Auf der Basis dieses Konstruktionsschemas lässt sich nun ein Typgraph erstellen, dessen Knotenmenge sich aus den Entitätstyp-Knoten sowie den Entitätstypklassen-Knoten zusammensetzt und dessen Kantenmenge aus $R_{classification}$ hervorgeht. Die entsprechende Visualisierung ist in Abbildung 4.3 zu betrachten.

Ausprägung 4.6.2 (Typgraph G_{T_1})

$$\begin{aligned} G_{T_1} &= (V(G_{T_1}), E(G_{T_1}) \subseteq V(G_{T_1}) \times V(G_{T_1})) \\ V(G_{T_1}) &= E_1 \cup K_1 \\ &= \{\texttt{IndoorClimbing}, \texttt{SportsFacility}\} \\ E(G_{T_1}) &= R_{classification_1} \\ &= \{(\texttt{IndoorClimbing}, \texttt{SportsFacility})\} \end{aligned}$$

Entitätstypklassen-Knoten werden in der folgenden Grafik als Hexagone dargestellt, Entitätstyp-Knoten als Rechtecke. Die Klassifikationskante wird durchgehend dargestellt und zeigt mit einem runden offenen Pfeil in die entsprechende Richtung.

Es gilt, dass Entitätstyp-Knoten von mehreren Klassen erfasst werden können. Außerdem besteht die Möglichkeit zur Ausbildung von Klassenhierarchien, wenn die Klassifikationsrelation $R_{classification}$ als Teilmenge des kartesischen Produktes zwischen der Vereinigungsmenge $E \cup K$ und der Menge der Entitätstypklassen-Knoten K definiert wird.

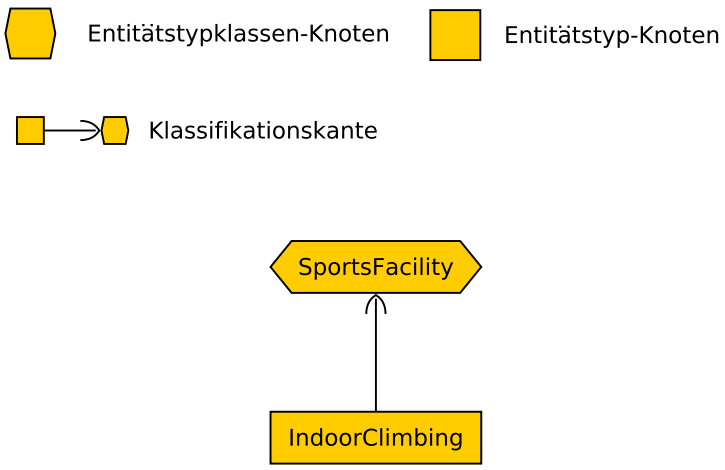

Abbildung 4.3: Typgraph Klassifikationsbeispiel (G_{T_1})

4.6.6 Mitgliedschaft von Annotationstypen

4.6.6.1 Reguläre Mitgliedschaft von Annotationstypen

Der Entitätstyp-Knoten *Kletterhalle* wurde im letzten Beispiel der Klasse *Sportstätte* zugeordnet und wird für das aktuelle Beispiel nun mit Attributen versehen, die ihn detaillierter modellieren. Wie eingangs erwähnt wurde, dient hierfür die Mitgliedschaftsrelation $R_{membership}$, die als Teilmenge des kartesischen Produktes zwischen der Menge der Entitätstyp-Knoten E und der Menge der Annotationstyp-Knoten A definiert wird:

$$R_{membership} \subseteq \{(x, y) \mid x \in E \wedge y \in A\} \tag{4.4}$$

Der Entitätstyp-Knoten *Kletterhalle* aggregiert nun im Sinne seiner Repräsentation entsprechende Annotationstyp-Knoten, die in Tabelle 4.2 aufgeführt sind.

Tabelle 4.2: Annotationstyp-Knoten Mitgliedschaftsbeispiel

Knotenart	**engl. Bezeichnung**	**Erläuterung**
Annotationstyp	Id	Eindeutige Kennnummer
Annotationstyp	Name	Name
Annotationstyp	Location	Ort
Annotationstyp	Height	Höhe (der Halle)
Annotationstyp	Walls	Anzahl der Kletterwände
Annotationstyp	Routes	Anzahl der Kletterrouten
Annotationstyp	Admission	Eintrittspreis

Die *Teilnahmebeziehung* zwischen dem Entitätstyp-Knoten *Kletterhalle* und den aufgeführten Annotationstyp-Knoten wird über die Mitgliedschaftsrelation $R_{membership}$ etabliert:

Ausprägung 4.6.3 (Mitgliedschaftsrelation $R_{membership_1}$)

$$
\begin{aligned}
E_1 &= \{\texttt{IndoorClimbing}\} \\
A_1 &= \{\texttt{Id},\ \texttt{Name},\ \texttt{Location},\ \texttt{Height},\ \texttt{Walls},\ \texttt{Routes},\ \texttt{Admission}\} \\
R_{membership_1} &= \{(\texttt{IndoorClimbing},\ \texttt{Id}),\ (\texttt{IndoorClimbing},\ \texttt{Name}), \\
&\quad (\texttt{IndoorClimbing},\ \texttt{Location}), (\texttt{IndoorClimbing},\ \texttt{Height}), \\
&\quad (\texttt{IndoorClimbing},\ \texttt{Walls}), (\texttt{IndoorClimbing},\ \texttt{Routes}), \\
&\quad (\texttt{IndoorClimbing},\ \texttt{Admission})\}
\end{aligned}
$$

Um den Informationen der Mitgliedschaftsrelation $R_{membership}$ Rechnung zu tragen, wird das Konstruktionsschema des Typgraphen erweitert, indem die Menge der Annotationstyp-Knoten in das Knotenuniversum aufgenommen und die Menge der Kanten um das Ergebnis der Mitgliedschaftsrelation ergänzt werden:

Konstruktionsschema 4.6.2 (Typgraph G_T, 2. Definition)

$$
\begin{aligned}
G_T &= (V(G_T), E(G_T) \subseteq V(G_T) \times V(G_T)) \\
V(G_T) &= E \cup K \cup A \\
E(G_T) &= R_{classification} \cup R_{membership}
\end{aligned}
$$

Die aktuellen Beispieldaten erlauben die Berechnung des Typgraphen G_{T_2}, der in Abbildung 4.4 visualisiert wird. Die durchgehend gezeichneten Kanten mit den spitzen Pfeilen repräsentieren dabei die eingeführten Mitgliedschaftskanten.

Ausprägung 4.6.4 (Typgraph G_{T_2})

$$
\begin{aligned}
G_{T_2} &= (V(G_{T_2}), E(G_{T_2}) \subseteq V(G_{T_2}) \times V(G_{T_2})) \\
V(G_{T_2}) &= E_1 \cup K_1 \cup A_1 \\
&= \{\texttt{SportsFacility},\ \texttt{IndoorClimbing}, \\
&\quad\ \texttt{Id},\ \texttt{Name},\ \texttt{Location},\ \texttt{Height},\ \texttt{Walls},\ \texttt{Routes},\ \texttt{Admission}\} \\
E(G_{T_2}) &= R_{classification_1} \cup R_{membership_1} \\
&= \{(\texttt{SportsFacility},\ \texttt{IndoorClimbing}), \\
&\quad\ (\texttt{IndoorClimbing},\ \texttt{Id}),\ (\texttt{IndoorClimbing},\ \texttt{Name}), \\
&\quad\ (\texttt{IndoorClimbing},\ \texttt{Location}), (\texttt{IndoorClimbing},\ \texttt{Height}), \\
&\quad\ (\texttt{IndoorClimbing},\ \texttt{Walls}), (\texttt{IndoorClimbing},\ \texttt{Routes}), \\
&\quad\ (\texttt{IndoorClimbing},\ \texttt{Admission})\}
\end{aligned}
$$

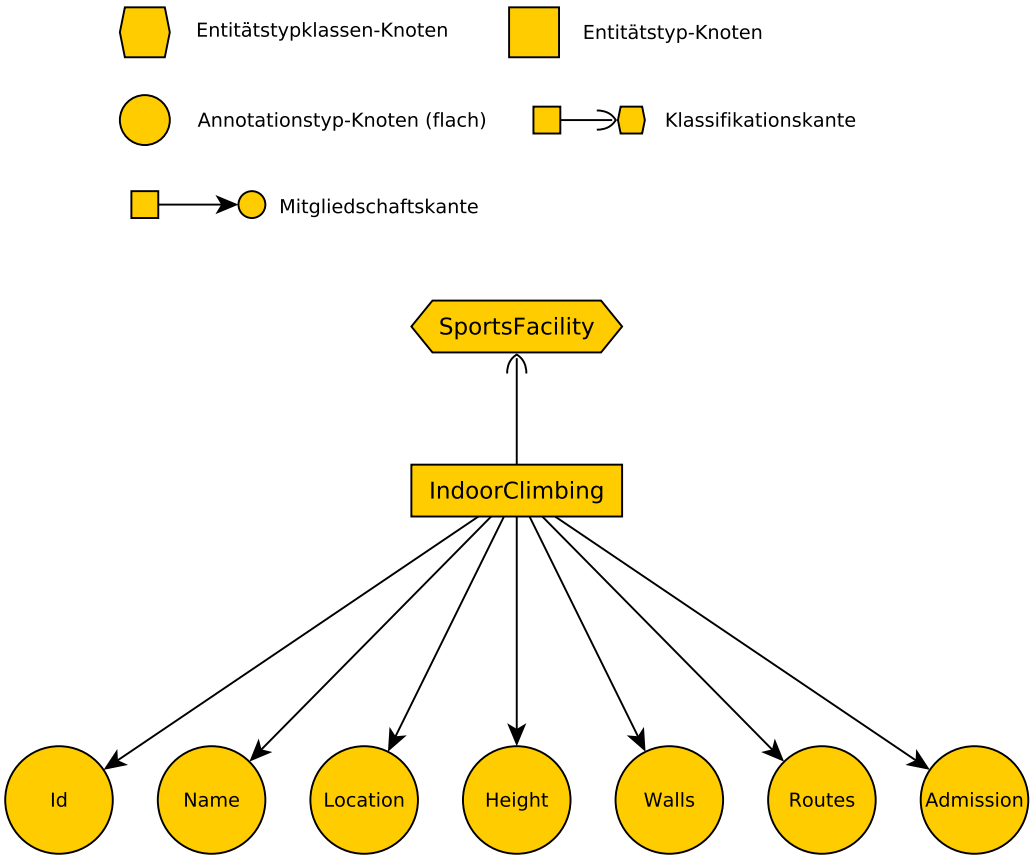

Abbildung 4.4: Typgraph Mitgliedschaftsbeispiel (G_{T_2})

4.6.6.2 Erweiterte Mitgliedschaft von Annotationstypen

Die reguläre Mitgliedschaft von Annotationstypen ist stark einschränkend, da im Falle einer Instanziierung gefordert wird, dass genau *eine* Attributausprägung angelegt wird. Mit der *erweiterten Mitgliedschaft* ist es möglich anzugeben, ob die Mitgliedschaft *erzwungen* (engl., Sg.: *required*) oder *optional* (engl., Sg.: *optional*) ist. Außerdem kann durch die Angabe von Kardinalitäten festgelegt werden, wie viele Instanzen desselben Annotationstyps erlaubt sind. So können Integritätsbedingungen bezüglich Existenz und Quantität wesentlich feiner modelliert werden. *Funktionale Abhängigkeit* zwischen Attributen als weitere wichtige Integritätsbedingung kann über die Markierung des Annotationstyps als einzigartig (engl., Sg.: *unique*) aufgebaut werden. Hierzu folgen nun einige Beispiele und weitere Erläuterungen.

4.6.6.2.1 Optionale Annotationstypen Ein Annotationstyp kann im Zuge seiner Mitgliedschaft als optional gekennzeichnet sein. Dies bedeutet, dass beim Aufbau eines Zustands keine Instanz dieses Attributs vorhanden sein muss. Erfolgt während der Ausprägung einer Entität keine Belegung des Annotationstyps mit einem gültigen Wert, so wird auch explizit kein Knoten hierfür instanziiert. Es gibt also keine *null*-Belegung, so wie es zum Beispiel im relationalen Modell gehandhabt wird. Handelt es sich um eine optionale Mitgliedschaft, so wird dies im Modell durch eine gestrichelte Mitgliedskante dargestellt. Siehe hierzu Abbildung 4.5, die eine abgewandelte Version der vorherigen Grafik darstellt und in der das Attribut Höhe (*Height*) als optional angegeben wird.

4.6.6.2.2 Multiple Annotationstypen Durch Angabe von *Mitgliedschafts-Kardinalitäten* (engl., Sg.: *membership cardinality*) kann festgelegt werden, wie viele Ausprägungen des gegebenen Annotationstyps angelegt werden dürfen. Es ist hierdurch möglich, Kollektionen zu modellieren. Dabei wird durch Angabe einer *unteren Schranke* (engl., Sg.: *lower bound*) angegeben, wie viele Elemente mindestens ausgeprägt werden müssen (engl., Sg.: *minimum membership cardinality*) und durch Angabe einer *oberen Schranke* (engl., Sg.: *upper bound*) angezeigt, wie viele Instanzen maximal ausgeprägt werden dürfen (engl., Sg.: *maximum membership cardinality*). Dabei gilt:

$$0 < minmc \leq maxmc \leq \infty \tag{4.5}$$

Die untere Schranke (*minmc*) muss mindestens 1 betragen, die obere Schranke (*maxmc*) kann theoretisch unendlich betragen. Die Angabe von 0 als untere Schranke, die intuitiv auf eine optionale Angabe schließen lässt, ist ungültig. Optionalität wird, wie oben angegeben, explizit angegeben, und ist mit der Angabe von beliebigen Kardinalitäten verträglich. So kann zum Beispiel gefordert sein, dass eine optionale Angabe, falls sie gemacht wird, in ihrer Anzahl innerhalb eines bestimmten Intervalls liegt. Falls eine konstante Anzahl von Ausprägungen gefordert ist, so reicht die Angabe einer Kardinalität.

Im Rahmen des gegebenen Beispiels kann so gefordert sein, dass es zu dem regulären Eintrittspreis mindestens noch einen ermäßigten gibt, und dass aus Gründen der Übersichtlichkeit die maximale Anzahl von Eintrittspreisen auf acht beschränkt wird. Die Notation sieht vor, dass die untere sowie die obere Schranke textuell von links nach rechts getrennt durch zwei Punkte neben die Pfeilspitze der Mitgliedschaftskante aufgetragen werden. Siehe hierzu das Attribut *Admission* in der Beispielgrafik 4.5.

Handelt es sich um ein Intervall, dessen obere Grenze als beliebig angegeben werden soll, so wird ein Stern (*) angegeben, siehe hierzu das umbenannte Attribut *Route*, das nun nicht mehr die Anzahl der verfügbaren Kletterrouten angibt, sondern eine Liste von Routennamen. Hier wird davon ausgegangen, dass es in einer Kletterhalle mindestens vier unterschiedliche Routen geben muss, um einen gewissen Besucheranspruch erfüllen zu können. In der folgenden Tabelle 4.3 sind alle möglichen Kardinalitätsangaben inklusive möglicher Kurzangaben aufgelistet sowie ihre Übersetzungen in die mathematische Schreibweise für abgeschlossene Intervalle natürlicher Zahlen $\mathbb{N}$.

Tabelle 4.3: Kardinalitätsangaben

Angabe	**Kurzschreibw.**	**Intervallnotation**	**Bedingung**
1..n	k. A.	$[1, n]$	$n \in \mathbb{N} \wedge n < \infty$
1..*	*	$[1, \infty]$	
m..n	k. A.	$[m, n]$	$m, n \in \mathbb{N} \wedge m < n < \infty$
n..n	n	$[n, n]$	$n \in \mathbb{N} \wedge n < \infty$ (konstant)
n..*	k. A.	$[n, \infty]$	$n \in \mathbb{N} \wedge n < \infty$

4.6.6.2.3 Einzigartige Annotationstypen Es kann die Anforderung bestehen, dass in einem Datenbankzustand σ die Wertebelegungen aller Instanzen von Annotationstypen unterschiedlich sein müssen. Im aktuellen Beispiel muss der Annotationstyp *Name* eindeutig sein und wird entsprechend markiert. Aufgrund der nach diesem Abschnitt eingeführten Mehrfachverwendung von Annotationstypen in 4.6.7 muss dabei festgelegt werden, ob diese Auszeichnung lediglich im Kontext der Verwendung eines bestimmten Entitätstyps oder aber global gültig sein soll. Im ersten Fall wird die Markierung an die Mitgliedschaftskante notiert, im zweiten Fall in den Knoten durch Unterstreichung des Attributnamens.

In diesem Beispiel soll diese Integritätsbedingung nur für Attributknoten greifen, die über den Entitätstyp `IndoorClimbing` ausgeprägt werden, was durch die Beschriftung der Mitgliedschaftskante mit *unique* kenntlich gemacht wird. Angenommen E sei eine Menge von Entitäten des Typs *IndoorClimbing* und A die Menge aller Ausprägungen von Annotationstypen, die als einzigartig gekennzeichnet sind, so gilt:

$$\forall x, y \in E \; \forall a \in A : value(x, a) \neq value(y, a) \tag{4.6}$$

In dem Beispiel findet sich ein Attribut *Id*, um eindeutige Kennungen für jede Entität speichern zu können, wie es zum Beispiel aus dem relationalen Entwurf bekannt ist. Hier werden häufig synthetische Schlüssel, sogenannte *Surrogatschlüssel* (engl., Sg.: *surrogate key*), in einer extra Spalte als fortlaufende Nummern mitgeführt, um Tupel eindeutig identifizieren zu können und den Zugriff auf Datensätze zu vereinfachen.

Es sei an dieser Stelle ausdrücklich darauf hingewiesen, dass dies im Rahmen des vorliegenden Konzepts nicht notwendig ist, da jeder später instanziierte Knoten eine global eindeutige Identifikation trägt, so auch jede Entität. Sofern nicht explizit eine weitere Kennung erwünscht ist, kann dieses Attribut aus diesem Beispiel entfernt werden, was in der Grafik 4.5 durch Ausgrauen symbolisiert wird.

Falls es sich um ein multiples Attribut handelt, so bezieht sich die Integritätsprüfung auf Ebene der einzelnen Elemente, d. h. insbesondere, dass nicht die Kollektionen als Ganzes verglichen werden, sondern ihre einzelnen Objekte. Die Integritätsbedingung der Einzigartigkeit zweier Ausprägungen multipler Annotationstypen ist also genau dann gegeben, wenn weder innerhalb noch zwischen den Kollektionen Identische Elemente existieren. Gegeben seien die Multimengen A, B von Annotationstyp-Instanzen zweier Entitäten, so gilt:

$$\forall a_1, a_2 \in A : a_1 \neq a_2 \tag{4.7}$$

$$\forall b_1, b_2 \in B : b_1 \neq b_2 \tag{4.8}$$

$$\forall a \in A \; \neg\exists b \in B : a = b \tag{4.9}$$

4.6.6.2.4 Datentypen Jeder Annotationstyp besitzt einen *Datentyp*, um die potentiellen Wertebelegungen von Attributen einzuschränken. Eine wesentliche Anforderung bei der Wahl von Datentypen ist die Relevanz für den praktischen Einsatz. Aus diesem Grund wurde eine Teilmenge der atomaren Datentypen von *XML Schema* übernommen, wie sie in der Tabelle 4.4 aufgelistet sind.

Tabelle 4.4: Atomare Datentypen

Engl. Datentypname	**Beschreibung**	**Anmerkung**
`string`	Zeichenkette	
`integer`	pos. bzw. neg. Ganzzahl	
`decimal`	pos. bzw. neg. Dezimalzahl	exakte Darstellung
`float`	pos. bzw. neg. Fließkommazahl	annähernde Darstellung
`boolean`	Wahrheitswert	
`date`	Datum	
`time`	Zeit	

4.6.6.2.5 Aufzählungen Innerhalb eines Datentypuniversums können weitere Einschränkungen gefordert sein, die nur eine bestimmte Teilmenge aller möglichen Werte zulassen. Aus diesem Grund gestattet das vorliegende Konzept die Angabe von *Aufzählungen* (engl., Sg.: *enumeration*), die in der Mengendarstellung die erlaubten Werte spezifizieren. Ein Attribut namens *Season*, das die Angabe der Jahreszeit erlaubt, kann zum Beispiel unter Angabe der folgenden Menge eingeschränkt werden:

$$\{spring,\ summer,\ autumn,\ winter\} \tag{4.10}$$

Anstelle einer Mengenaufzählung kann für numerische Werte auch eine *Mengenbeschreibung* unter Zuhilfenahme eines Prädikats definiert werden. Das folgende Beispiel definiert eine Menge V von gültigen Werten als Teilmenge der natürlichen Zahlen $\mathbb{N}$ für einen Annotationstyp *Date*, der eine Jahreszahl repräsentiert:

$$V = \{n \in \mathbb{N} \mid n \geq 1984 \ \wedge \ n < 2012\} \tag{4.11}$$

Auch Zeichenketten können über Angabe von regulären Ausdrücken weiter beschränkt werden. In diesem Beispiel muss das Attribut *Name* folgenden Kriterien genügen:

- Es sind lediglich Buchstaben sowie Bindestriche erlaubt.
- Der Name muss mit einem Großbuchstaben beginnen.
- Die Länge des Namens muss zwischen 4 und 24 liegen.

Hieraus können exemplarisch folgende nicht optimierte Perl-kompatible reguläre Ausdrücke (PCRE) abgeleitet werden, die je nach Situation auch zusammengefasst werden können:

- ^[A-Za-z0-9-]+$
- ^.{4,24}$

Die Beschränkung von Zeichenketten über die Definition von regulären Ausdrücken bietet ein mächtiges Mittel, um entsprechende Anforderungen spezifizieren zu können. Sie können zum Beispiel in einer Sicht verwendet werden, die für Entwickler gedacht ist.

In der Beispielgrafik 4.5 wurden alle Annotationstypen mit einem Datentyp versehen und die potentiellen Werte für Höhe und Name weiter eingeschränkt.

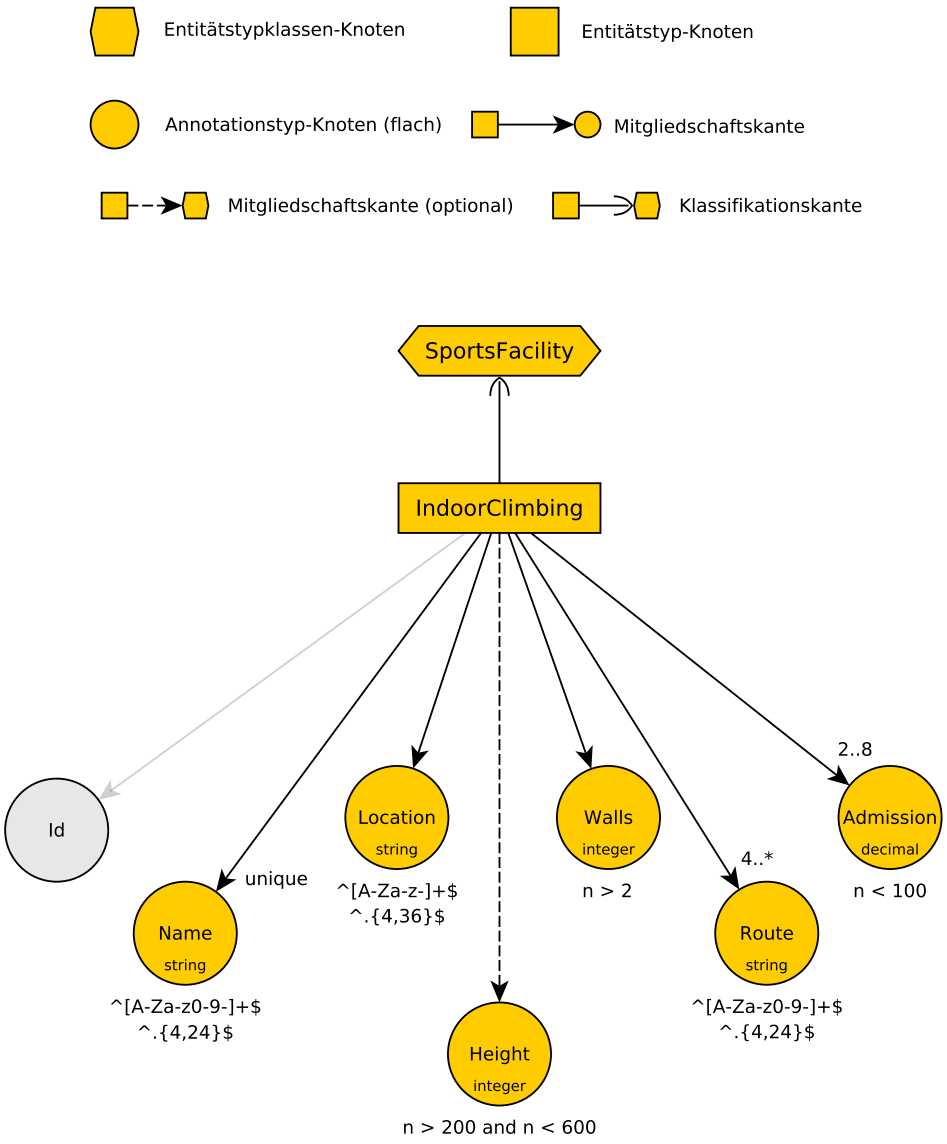

Abbildung 4.5: Typgraph Mitgliedschaftsbeispiel (mit Erweiterungen) (G_{T_3})

4.6.7 Wiederverwendung von Annotationstypen

Die Repräsentation der Attribute als eigenständige Annotationstyp-Knoten ermöglicht auf einfache Weise deren *Wiederverwendung* (engl., Sg.: *reuse*), indem der Quellmenge E der Mitgliedschaftsrelation $R_{membership}$ neue Entitätstyp-Knoten hinzugefügt werden.

An diesem Punkt wird festgestellt, dass die Eigenständigkeit der Annotationstyp-Knoten eingeschränkt ist, da sie nicht unabhängig von einer Mitgliedschaft existieren

können. Sie müssen stets mit mindestens einem Entitätstyp-Knoten verbunden sein, um später als Teil eines Zustandes gelten zu können. Lediglich die vorläufige Instanziierung im Zuge des Aufbaus einer Entität ist möglich, allerdings nicht die isolierte Speicherung.

Die Wiederverwendung bricht einerseits mit dem Konzept der Kapselung, doch sie bietet andererseits einen Vorteil zugunsten der Integrität, da nun davon ausgegangen werden kann, dass ein Attribut mit gleicher Benennung unabhängig vom Objektkontext denselben Sachverhalt beschreibt. Die redundante Definition von Attributen und ihren Integritätsbedingungen wird dadurch vermieden.

Für dieses Beispiel soll das Konzept *Skatepark* eingeführt werden, das ebenfalls der Klasse *SportsFacility* angehört:

Ausprägung 4.6.5 (Klassifikationsrelation $R_{classification_2}$)

$$
\begin{aligned}
K_1 &= \{\texttt{SportsFacility}\} \\
E_2 &= \{\texttt{IndoorClimbing},\ \texttt{Skatepark}\} \\
R_{classification_2} &= \{(\texttt{SportsFacility},\ \texttt{IndoorClimbing}) \\
&\qquad (\texttt{SportsFacility},\ \texttt{Skatepark})\}
\end{aligned}
$$

Für den Aufbau des neuen Typgraphen ist eine Erweiterung der bisherigen Knotenmengen um einen Entitätstyp-Knoten und mehrere Annotationstyp-Knoten notwendig, wie sie in Tabelle 4.5 aufgeführt sind.

Tabelle 4.5: Erweiterung der Knotenmengen

Knotenart	**engl. Bezeichnung**	**Erläuterung**
Entitätstyp	Skatepark	repräsentiert Skatepark
Annotationstyp	Length	Länge (der Parkfläche)
Annotationstyp	Width	Breite (der Parkfläche)
Annotationstyp	Obstacles	Anzahl der Hindernisse
Annotationstyp	Roofed	Angabe, ob Park überdacht ist

Die bisherige Definition der Mitgliedschaftsrelation $R_{membership}$ bleibt für dieses Beispiel gültig. Die Ergebnismenge ihrer Ausprägung wird allerdings erheblich größer als im vorherigen Beispiel:

Ausprägung 4.6.6 (Mitgliedschaftsrelation $R_{membership_2}$)

$$
\begin{aligned}
E_2 &= \{\texttt{IndoorClimbing}, \texttt{Skatepark}\} \\
A_2 &= \{\texttt{Name}, \texttt{Location}, \texttt{Height}, \texttt{Walls}, \texttt{Routes}, \texttt{Admission} \\
&\quad\ \texttt{Length}, \texttt{Width}, \texttt{Obstacles}, \texttt{Roofed}\} \\
R_{membership_2} &= \{(\texttt{IndoorClimbing}, \texttt{Name}), \\
&\quad\ (\texttt{IndoorClimbing}, \texttt{Location}), (\texttt{IndoorClimbing}, \texttt{Height}), \\
&\quad\ (\texttt{IndoorClimbing}, \texttt{Walls}), (\texttt{IndoorClimbing}, \texttt{Routes}), \\
&\quad\ (\texttt{IndoorClimbing}, \texttt{Admission})\} \\
&\quad\ (\texttt{Skatepark}, \texttt{Name}), (\texttt{Skatepark}, \texttt{Location}), \\
&\quad\ (\texttt{Skatepark}, \texttt{Length}), (\texttt{Skatepark}, \texttt{Width}), \\
&\quad\ (\texttt{Skatepark}, \texttt{Obstacles}), (\texttt{Skatepark}, \texttt{Roofed}), \\
&\quad\ (\texttt{Skatepark}, \texttt{Admission})\}
\end{aligned}
$$

Der neue Typgraph G_{T_4} in Abbildung 4.6 wird wie im letzten Beispiel auf Basis der Knotenmengen und der Ergebnismengen der Klassifikationsrelation bzw. der Mitgliedschaftsrelation berechnet, wobei diesmal auf eine explizite Mengenaufzählung verzichtet wird.

Konstruktionsschema 4.6.3 (Typgraph G_{T_4})

$$
\begin{aligned}
G_{T_4} &= (V(G_{T_4}), E(G_{T_4}) \subseteq V(G_{T_4}) \times V(G_{T_4})) \\
V(G_{T_4}) &= E_2 \cup K_1 \cup A_2 \\
E(G_{T_4}) &= R_{classification_2} \cup R_{membership_2}
\end{aligned}
$$

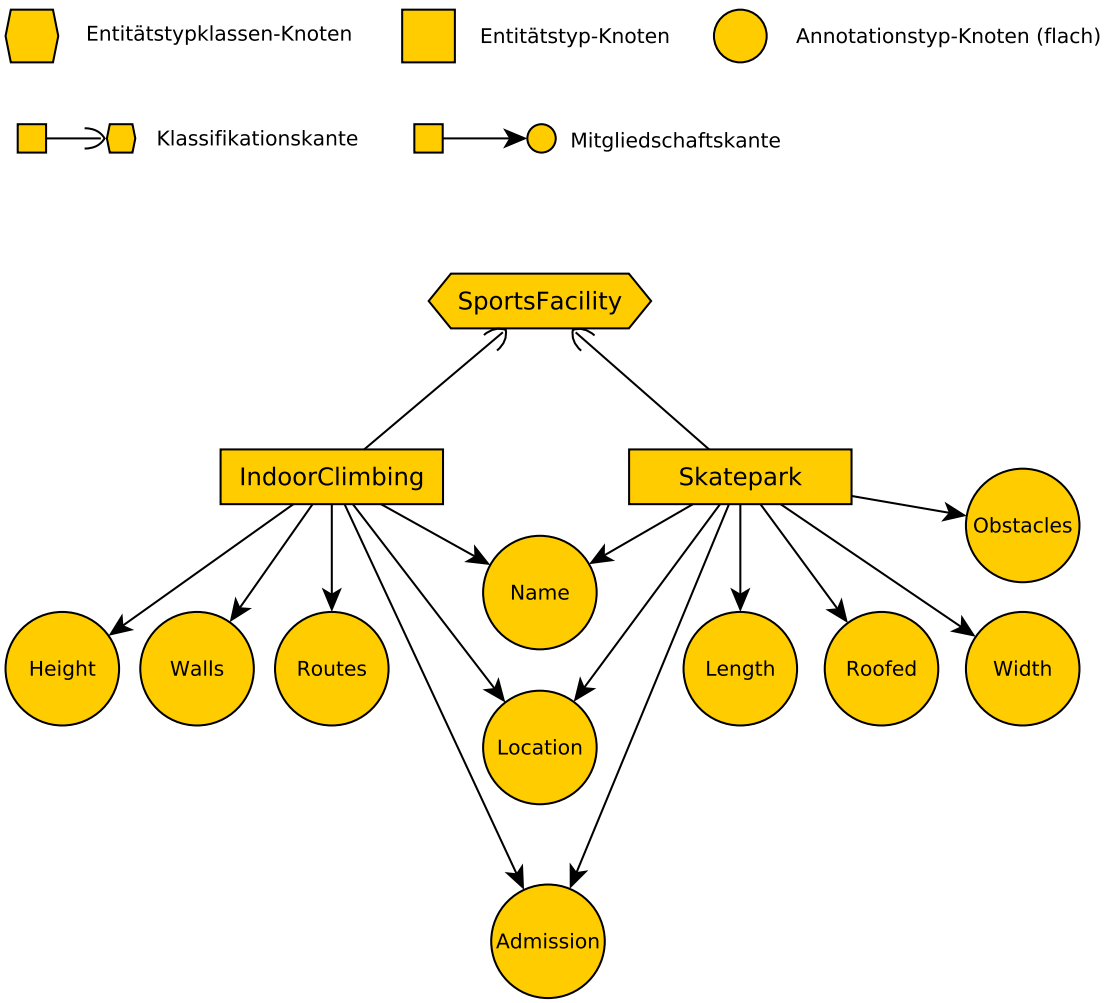

Abbildung 4.6: Typgraph Wiederverwendungsbeispiel (G_{T_4})

4.6.8 Vererbung von Annotationstypen (Generalisierung)

Um die Wiederverwendbarkeit von Konzepten zu ermöglichen, kann ein Entitätstyp-Knoten von einem oder mehreren Superentitätstyp-Knoten erben, die wiederum untereinander erben können. Die Menge aller Annotationstyp-Knoten des Superentitätstyp-Knotens und des erbenden Entitätstyp-Knotens bzw. Superentitätstyp-Knotens werden dann vereinigt. Die Vererbung wird in ER-Diagrammen z. B. mittels einer „is-a“-Beziehung realisiert.

Ausgehend von einem Szenario der Wiederverwendung von Annotationstyp-Knoten wird nun ein Superentitätstyp-Knoten abgeleitet, der als abstraktes Superkonzept

dient und seine Attribute vererbt. Dieses Beispiel behandelt die Einfachvererbung der Tiefe 1. Es ist jedoch möglich, von beliebig vielen Supertypen gleichzeitig zu erben bzw. Vererbungshierarchien beliebiger Tiefe aufzubauen. An dieser Stelle erweist es sich als praktisch, dass jeder Annotationstyp durch seine Benennung eindeutig referenziert werden kann, denn so können bei der Vereinigung der Attributmengen keine Namenskollisionen auftreten.

Im vorangegangenen Beispiel ist zu erkennen, dass bestimmte Annotationstyp-Knoten einer zweimaligen Verwendung zugeführt werden. Die beiden Entitätstyp-Knoten *Kletterhalle* und *Skatepark* unterhalten jeweils Mitgliedschaften zu folgenden Attributen (siehe Tabelle 4.6):

Tabelle 4.6: Zweimalig verwendete Annotationstypen

Knotenart	**engl. Bezeichnung**	**Erläuterung**
Annotationstyp	Name	Name
Annotationstyp	Location	Ort
Annotationstyp	Admission	Eintrittspreis

Das Konzept des Superentitätstyps erlaubt es nun, diese Attribute in einem Supertyp zusammenzufassen und eine Vererbungshierarchie auszubilden. Hierzu wird eine binäre Vererbungsrelation $R_{inheritance}$ als Teilmenge des kartesischen Produktes zwischen der Menge der Entitätstyp-Knoten E und der Menge der Superentitätstyp-Knoten S definiert:

$$R_{inheritance} \subseteq \{(x, y) \mid x \in E \wedge y \in S\} \tag{4.12}$$

Um eine entsprechende Ausprägung der Vererbungsrelation $R_{inheritance}$ zu erhalten, muss mindestens ein Superentitätstyp-Knoten konstruiert werden. Hierzu betrachten wir die Mitgliedschaftsrelation.

Sei A' die zu berechnende Menge der Attribute eines potentiellen Supertyps. Seien weiterhin die Menge E der Entitätstyp-Knoten sowie die Menge A der Annotationstyp-Knoten gegeben, dann gilt:

Konstruktionsschema 4.6.4 (Attributmenge eines Supertypen)

$$
\begin{aligned}
A' &= \{a \mid a \in A \wedge \forall e \in E : (a, e) \in M\} \\
&\quad mit \\
E &: \; Menge\ der\ betrachteten\ Entitätstyp\text{-}Knoten, \\
A &: \; Menge\ der\ betrachteten\ Annotationstyp\text{-}Knoten, \\
M &: \; Menge\ der\ betrachteten\ Mitgliedschafts\text{-}Tupel
\end{aligned}
$$

Unter Verwendung der konkreten Beispielmengen E_2, A_2 und $R_{membership_2}$ enthält A' nun genau alle Annotationstypen die von beiden Entitätstypen *Kletterhalle* und *Skatepark* assoziiert werden. Es sei angemerkt, dass die aufgeführte Berechnungsvorschrift in dieser Form aufgrund der Allquantifizierung keine Ermittlung von partiellen Supertypen zulässt.

Ausprägung 4.6.7 (Attributmenge eines Supertypen)

$$
\begin{aligned}
E &= E_2 \\
A &= A_2 \\
M &= R_{membership_2} \\
A' &= \{\texttt{Name}, \texttt{Location}, \texttt{Admission}\}
\end{aligned}
$$

Die ermittelten Annotationstyp-Knoten aus A' werden nun durch einen entsprechenden Superentitätstyp-Knoten aggregiert:

Tabelle 4.7: Superentitätstyp-Knoten für Vererbungsbeispiel

Knotenart	**engl. Bezeichnung**	**Erläuterung**
Superentitätstyp	SuperFacility	Superkonzept Sportstätte

Ausprägung 4.6.8 (Vererbungsrelation $R_{inheritance_1}$)

$$
\begin{aligned}
S_1 &= \{\texttt{SuperFacility}\} \\
E_2 &= \{\texttt{IndoorClimbing}, \texttt{Skatepark}\} \\
R_{inheritance_1} &= \{(\texttt{IndoorClimbing}, \texttt{SuperFacility}) \\
&\qquad (\texttt{Skatepark}, \texttt{SuperFacility})\}
\end{aligned}
$$

Die Einführung der Vererbungsrelation $R_{inheritance}$ bedingt eine Änderung der Definition für die Mitgliedschaftsrelation $R_{membership}$, deren Quellmenge um die Superentitätstyp-Knoten erweitert wird. Sie wird nun als Teilmenge des kartesischen Produkts zwischen der Vereinigungsmenge $E \cup S$ und der Menge der Annotationstyp-Knoten A definiert:

$$R_{membership} \subseteq \{(x, y) \mid x \in (E \cup S) \wedge y \in A\} \tag{4.13}$$

Ausprägung 4.6.9 (Mitgliedschaftsrelation $R_{membership_3}$)

$$\begin{aligned}
E_2 \cup S_1 &= \{\texttt{SuperFacility},\ \texttt{IndoorClimbing},\ \texttt{Skatepark}\} \\
A_2 &= A_2 \\
R_{membership_3} &= \{(\texttt{SuperFacility},\ \texttt{Name}), \\
&\quad (\texttt{SuperFacility},\ \texttt{Location}),\ (\texttt{SuperFacility},\ \texttt{Admission}), \\
&\quad (\texttt{IndoorClimbing},\ \texttt{Height}),\ (\texttt{IndoorClimbing},\ \texttt{Walls}), \\
&\quad (\texttt{IndoorClimbing},\ \texttt{Routes}) \\
&\quad (\texttt{Skatepark},\ \texttt{Length}),\ (\texttt{Skatepark},\ \texttt{Width}), \\
&\quad (\texttt{Skatepark},\ \texttt{Obstacles}), (\texttt{Skatepark},\ \texttt{Roofed})\}
\end{aligned}$$

Der neue Typgraph G_{T_5} in Abbildung 4.7 wird auf Basis der Ergebnismengen der Klassifikationsrelation $R_{classification}$ und Mitgliedschaftsrelation $R_{membership}$ berechnet, wobei diesmal die Menge der Superentitätstyp-Knoten berücksichtigt wird.

Wie bereits erwähnt wurde, besteht die Möglichkeit zur Ausbildung von Klassenhierarchien. Hierzu kann die Vererbungsrelation $R_{inheritance}$ als Teilmenge des kartesischen Produktes zwischen der Vereinigungsmenge $E \cup S$ und der Menge der Superentitätstyp-Knoten S definiert werden.

Konstruktionsschema 4.6.5 (Typgraph G_{T_5})

$$\begin{aligned}
G_{T_5} &= (V(G_{T_5}), E(G_{T_5}) \subseteq V(G_{T_5}) \times V(G_{T_5})) \\
V(G_{T_5}) &= E_2 \cup K_1 \cup A_2 \cup S_2 \\
E(G_{T_5}) &= R_{classification_2} \cup R_{membership_3} \cup R_{inheritance_1}
\end{aligned}$$

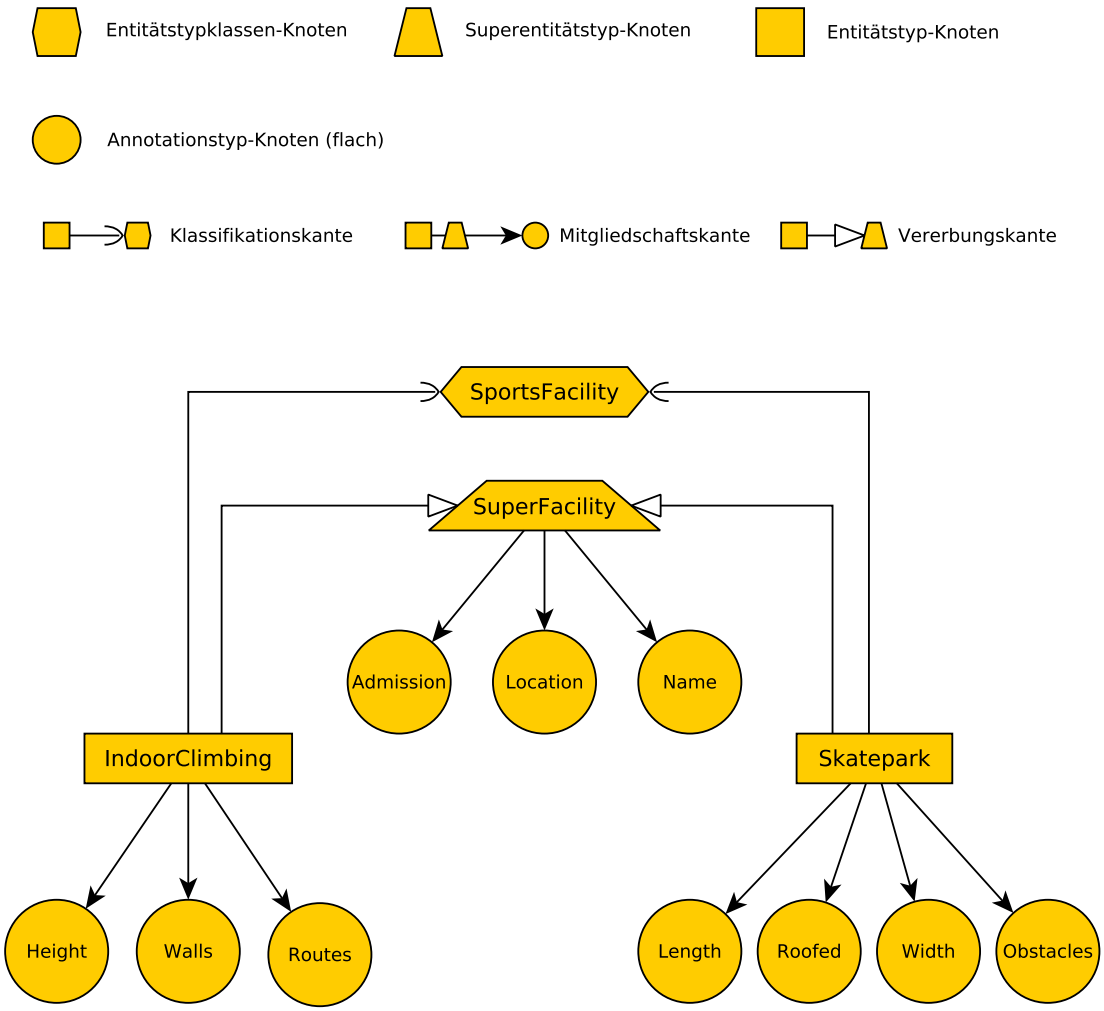

Abbildung 4.7: Typgraph Vererbungsbeispiel (G_{T_5})

4.6.9 Entitätstyp-Schnittstellen

Das Konzept der Entitätstyp-Schnittstelle kommt dann zum Einsatz, wenn sichergestellt sein muss, dass Entitätstypen bestimmte Attribute besitzen, die für die Verarbeitung abgeleiteter Instanzen notwendig sind, und bildet so eine weitere Art von Integritätsbedingung aus.

Entitätstyp-Schnittstellen können theoretisch auch durch Vererbung realisiert werden, indem ein Superentitätstyp modelliert wird, der die benötigten Attribute aggregiert. Hierauf wurde aufgrund der irreführenden Semantik verzichtet, denn ein Supertyp

repräsentiert in der Regel das gleiche Konzept wie alle von ihm abgeleiteten Klassen nur auf einer höheren Abstraktionsebene.

Entitätstyp-Schnittstellen sind deshalb so konzipiert, dass sie weder von andern Schnittstellen noch von Superentitätstypen erben können und sich so nicht in einer Typhierarchie einordnen. Umgekehrt können Supertypen allerdings Schnittstellen erfüllen.

Das folgende Beispiel realisiert eine Erweiterung des Schemas aus dem letzten Abschnitt 4.6.8, indem davon ausgegangen wird, dass beide Entitätstypen *IndoorClimbing* sowie *Skatepark* die Schnittstelle *Visitable* mit folgenden Attributen erfüllen müssen:

Tabelle 4.8: Knotenübersicht für Schnittstellen-Beispiel

Knotenart	**engl. Bezeichnung**	**Erläuterung**
Entitätstyp-Schnittstelle	Visitable	übersetzt als *besuchbar*
Annotationstyp	OpeningTime	Öffnungszeit
Annotationstyp	ClosingTime	Geschäftsschluss
Annotationstyp	Drive	Anfahrt

Eine sogenannte Implementierungsrelation $R_{implementation}$ verknüpft die besagten Entitätstyp-Knoten und den entsprechenden Schnittstellen-Knoten über Implementierungskanten und wird als Teilmenge des kartesischen Produktes zwischen der Menge der Entitätstyp-Knoten E und der Menge der Entitätstyp-Schnittstellen-Knoten F definiert:

$$R_{implementation} \subseteq \{(x, y) \mid x \in E \wedge y \in F\} \tag{4.14}$$

Ausprägung 4.6.10 (Implementierungsrelation $R_{implementation_1}$)

$$\begin{aligned} F_1 &= \{\texttt{Visitable}\} \\ E_2 &= \{\texttt{IndoorClimbing}, \texttt{Skatepark}\} \\ R_{implementation_1} &= \{(\texttt{IndoorClimbing}, \texttt{Visitable}) \\ &\qquad (\texttt{Skatepark}, \texttt{Visitable})\} \end{aligned}$$

Die Einführung der Implementierungsrelation $R_{implementation}$ bedingt eine Änderung der Definition für die Mitgliedschaftsrelation $R_{membership}$, deren Quellmenge um die Entitätstyp-Schnittstellen-Knoten erweitert wird. Sie wird nun als Teilmenge des kartesischen Produkts zwischen der Vereinigungsmenge $E \cup S \cup F$ und der Menge der Annotationstyp-Knoten A definiert:

$$R_{membership} \subseteq \{(x, y) \mid x \in (E \cup S \cup F) \wedge y \in A\} \tag{4.15}$$

Die konkrete Tupelmenge für das Ergebnis der Mitgliedschaftsrelation $R_{membership}$ für das aktuelle Beispiel sieht wie folgt aus:

Ausprägung 4.6.11 (Mitgliedschaftsrelation $R_{membership_4}$)

$$\begin{aligned}
E_2 \cup S_1 \cup F_1 &= \{\texttt{Visitable}, \texttt{SuperFacility}, \texttt{IndoorClimbing}, \texttt{Skatepark}\} \\
A_2 &= A_2 \\
R_{membership_4} &= \{(\texttt{Visitable},\ \texttt{OpeningTime}),\ (\texttt{Visitable},\ \texttt{ClosingTime}), \\
&\quad\ (\texttt{Visitable},\ \texttt{Drive}),\ (\texttt{SuperFacility},\ \texttt{Name}), \\
&\quad\ (\texttt{SuperFacility},\ \texttt{Location}),\ (\texttt{SuperFacility},\ \texttt{Admission}), \\
&\quad\ (\texttt{IndoorClimbing},\ \texttt{Height}),\ (\texttt{IndoorClimbing},\ \texttt{Walls}), \\
&\quad\ (\texttt{IndoorClimbing},\ \texttt{Routes}), \\
&\quad\ (\texttt{Skatepark},\ \texttt{Length}),\ (\texttt{Skatepark},\ \texttt{Width}), \\
&\quad\ (\texttt{Skatepark},\ \texttt{Obstacles}),\ (\texttt{Skatepark},\ \texttt{Roofed})\}
\end{aligned}$$

Der neue Typgraph G_{T_6} in Abbildung 4.8 enthält nun auch die Schnittstellen-Knoten und setzt sich wie folgt zusammen:

Konstruktionsschema 4.6.6 (Typgraph G_{T_6})

$$\begin{aligned}
G_{T_6} &= (V(G_{T_6}), E(G_{T_6}) \subseteq V(G_{T_6}) \times V(G_{T_6})) \\
V(G_{T_6}) &= E_2 \cup K_1 \cup A_2 \cup S_1 \cup F_1 \\
E(G_{T_6}) &= R_{classification_2} \cup R_{membership_4} \cup R_{inheritance_1} \cup R_{implementation_1}
\end{aligned}$$

Die Implementierungskanten enden mit einem schwarzen ausgefüllten Kreis in Richtung der Entitätstyp-Schnittstellen-Knoten.

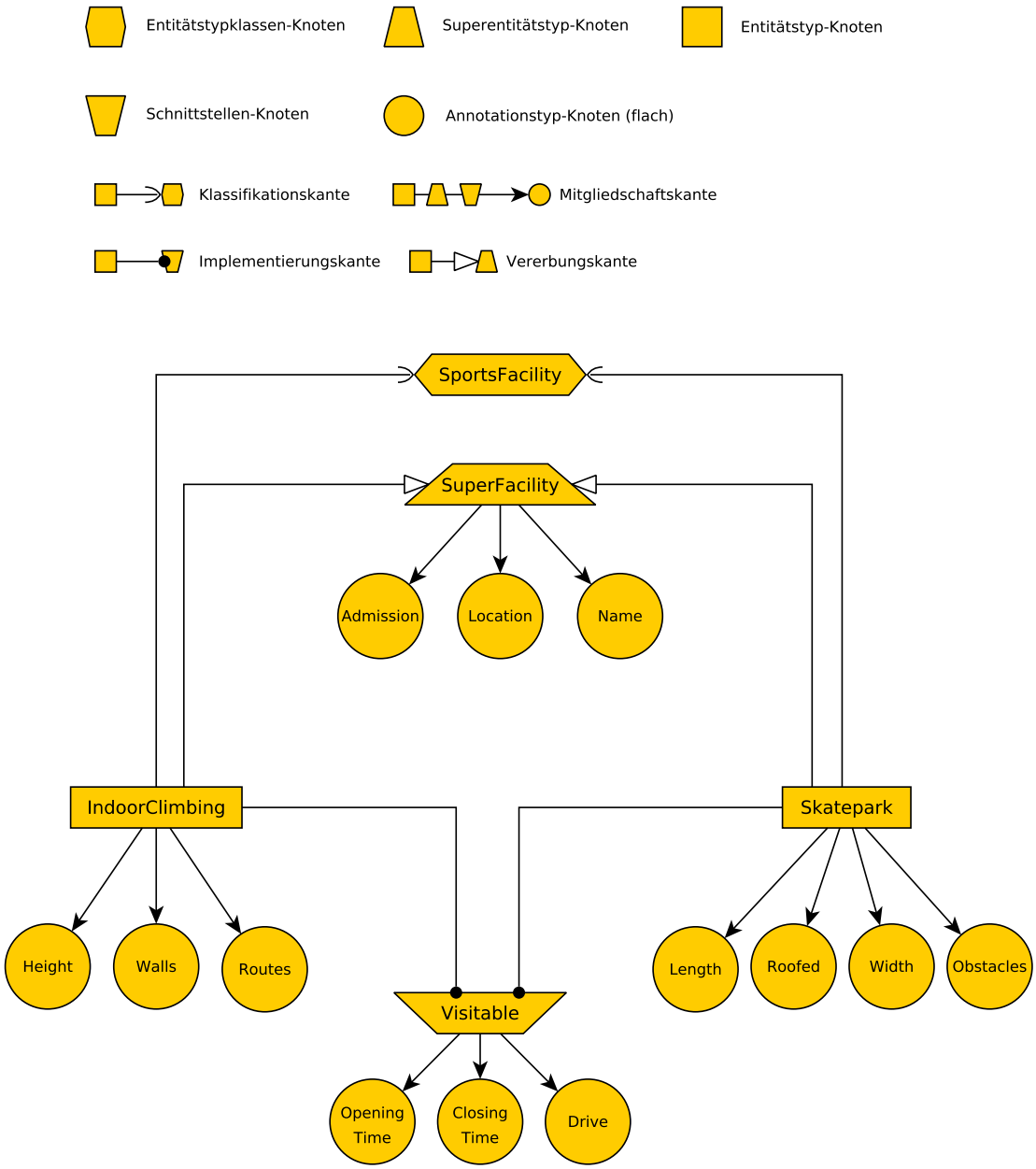

Abbildung 4.8: Typgraph Schnittstellen-Beispiel (G_{T_6})

4.6.10 Strukturierte Typen

Wie bereits erläutert wurde, repräsentieren Entitätstyp-Knoten eigenständige ideelle oder sachliche Konzepte der Anwendungsdomäne. Es wurde festgestellt, dass für die

Modellierung von Sachverhalten Annotationstyp-Knoten verwendet werden, um die benötigten Merkmale abzubilden. Bisher waren diese Attribute *flach* (engl., Sg.: *plain*). Das Attribut *Titel* kann nach einer potentiellen Instantiierung in einem bestimmten Zustand σ zum Beispiel den literalen Wert „Berlin-Marathon 2013“ annehmen. Es gibt jedoch Merkmale, die *strukturiert* (engl., Sg.: *structured*) sein können und für die eine flache Abbildung nicht ausreicht.

Nehmen wir an, wir wollen sowohl dem Konzept *Kletterhalle* als auch dem Konzept *Skatepark* ein gemeinsames Attribut *Dimension* hinzufügen. Dieses Attribut ist strukturiert, denn es fasst die Merkmale *Länge*, *Breite* und *Höhe* zusammen, ordnet diese unter und bildet so ein hierarchisches Gefüge aus.

Um anzuzeigen, ob ein Annotationstyp-Knoten flache oder strukturierte Daten repräsentiert, wird an dieser Stelle der sogenannte Annotationstyp-Modus (engl., Sg.: *annotation type mode*) eingeführt. Der Annotationstyp-Modus definiert die Semantik des Annotationstyp-Knotens, indem er angibt, wie der entsprechende Knoten nebst seiner Nachfahren behandelt werden muss.

Auch jeder Entitätstyp-Knoten bildet eine hierarchische Struktur aus und es ist eine Entwurfsentscheidung, ob Attribute als strukturierte Annotationstyp-Knoten abstrahiert werden oder als eigenständige Entitätstyp-Knoten in das Schema eingehen.

Hierzu betrachte man die Stärke des Typs im Kontext des Modellierungsszenarios. Die Stärke eines Typs lässt sich u. a. durch die Relevanz für das gesamte Schema ermitteln. Wird zum Beispiel das Merkmal *Dimension* nur im Zusammenhang mit bestimmten Entitätstyp-Knoten bzw. ausschließlich im Rahmen anderer strukturierter Attribute gebraucht und findet es ansonsten keine eigenständige bzw. unabhängige Verwendung, so ist die Modellierung als strukturierter Annotationstyp-Knoten angebracht.

Soll das Attribut *Dimension* hingegen eine eigenständige Existenz ausbilden, um zum Beispiel in einem bestimmten Zustand σ eine Beziehung zu anderen Entitäten eingehen zu können, dann ist die Abbildung als eigenständiger Entitätstyp-Knoten angebracht.

Unabhängig davon kann es auch sinnvoll sein, strukturierte Attribute zunächst als eigenständige Konzepte zu modellieren, wenn dies der subjektiven Abstraktion realer Verhältnisse näher kommt.

Um also komplexe Datenstrukturen aufbauen zu können, ist die Mitgliedschaftsrelation $R_{membership}$ nun auch zwischen Annotationstyp-Knoten zulässig. Sie wird neu definiert als Teilmenge des kartesischen Produkts zwischen der Vereinigungsmenge $S \cup E \cup F \cup A$ und der Menge der Annotationstyp-Knoten A:

$$R_{membership} \subseteq \{(x, y) \mid x \in (S \cup E \cup F \cup A) \wedge y \in A\} \tag{4.16}$$

Ausprägung 4.6.12 (Mitgliedschaftsrelation $R_{membership_5}$)

$$\begin{aligned}
E_2 \cup S_1 \cup F_1 \cup A_2 = \{&\texttt{Visitable}, \texttt{SuperFacility}, \texttt{IndoorClimbing}, \texttt{Skatepark},\\
&\texttt{Name}, \texttt{Location}, \texttt{Height}, \texttt{Walls}, \texttt{Routes}, \texttt{Admission},\\
&\texttt{Length}, \texttt{Width}, \texttt{Obstacles}, \texttt{Roofed}, \texttt{Dimension}\}\\
R_{membership_5} = \{&(\texttt{Visitable}, \texttt{OpeningTime}), (\texttt{Visitable}, \texttt{ClosingTime}),\\
&(\texttt{Visitable}, \texttt{Drive}), (\texttt{SuperFacility}, \texttt{Name}),\\
&(\texttt{SuperFacility}, \texttt{Location}), (\texttt{SuperFacility}, \texttt{Admission}),\\
&(\texttt{SuperFacility}, \texttt{Dimension}),\\
&(\texttt{IndoorClimbing}, \texttt{Walls}), (\texttt{IndoorClimbing}, \texttt{Routes}),\\
&(\texttt{Skatepark}, \texttt{Obstacles}), (\texttt{Skatepark}, \texttt{Roofed})\\
&(\texttt{Dimension}, \texttt{Width}), (\texttt{Dimension}, \texttt{Length})\\
&(\texttt{Dimension}, \texttt{Height})\}
\end{aligned}$$

Für den Beispieltypgraph G_{T_7} in Abbildung 4.9 ergibt sich auf Basis der aktuellen Definition für Mitgliedschaften folgende Berechnung:

Konstruktionsschema 4.6.7 (Typgraph G_{T_7})

$$\begin{aligned}
G_{T_7} &= (V(G_{T_7}), E(G_{T_7}) \subseteq V(G_{T_7}) \times V(G_{T_7}))\\
V(G_{T_7}) &= E_2 \cup K_1 \cup S_1 \cup F_1 \cup A_2\\
E(G_{T_7}) &= R_{classification_1} \cup R_{membership_5} \cup R_{inheritance_1} \cup R_{implementation_1}
\end{aligned}$$

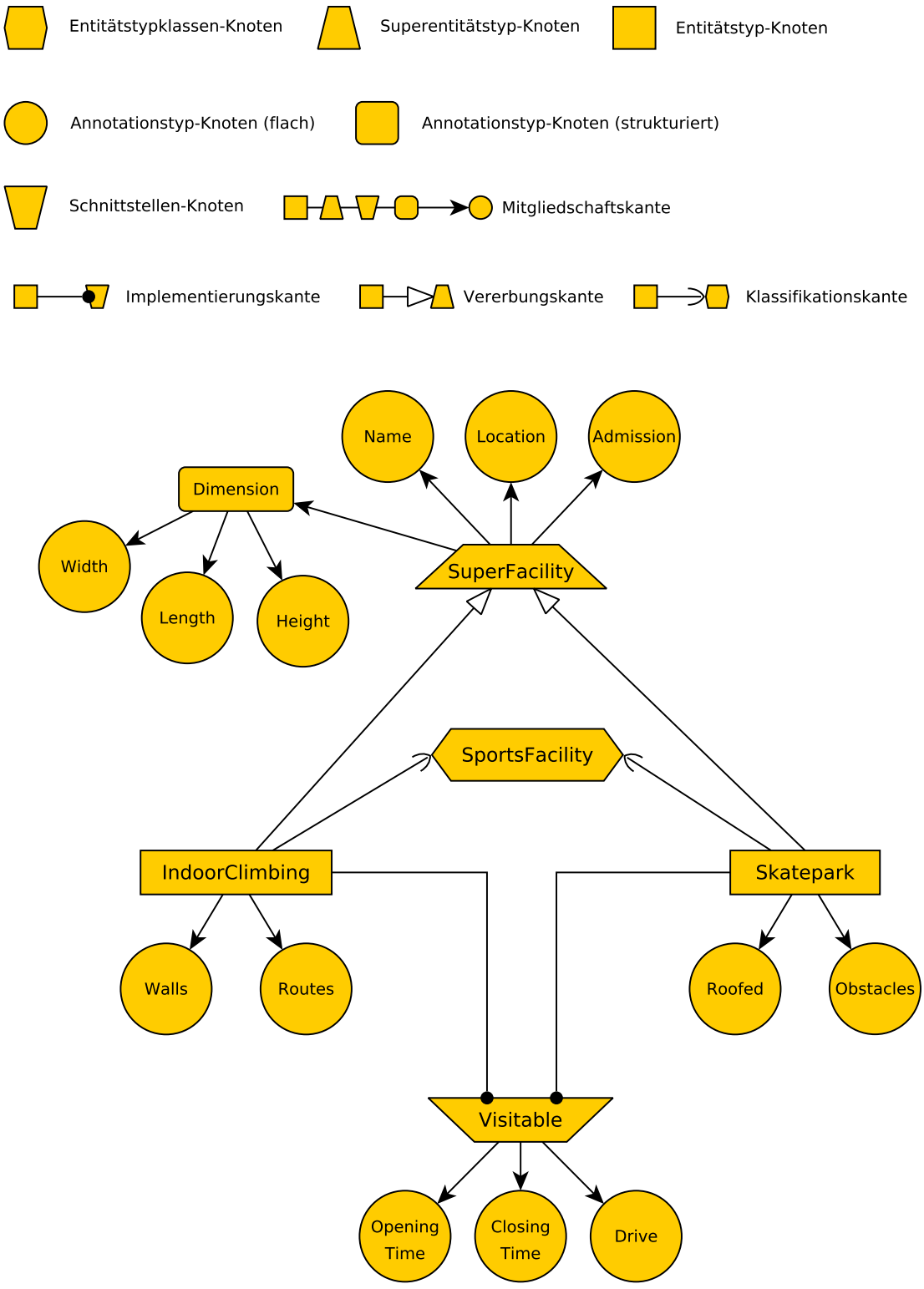

Abbildung 4.9: Typgraph Strukturierter Typ (G_{T_7})

4.6.11 Verschachtelung strukturierter Typen

Angenommen, es soll das Konzept *Freibad* modelliert werden, das ebenfalls als *Sportstätte* klassifiziert wird. Ausgehend von den vorangegangenen Beispielen wird der Typgraph in mehreren Schritten definiert, wobei die Konzepte *Kletterhalle* und *Skatepark* nicht berücksichtigt, in der Abbildung jedoch angedeutet werden.

Zunächst erfolgt eine Übersicht über alle Knoten, die verwendet werden in Tabelle 4.9:

Tabelle 4.9: Knotenübersicht für das Konzept Freibad

Knotenart	**engl. Bezeichnung**	**Erläuterung**
Entitätstypklasse	SportsFacility	Klasse Sportstätte
Basisentitätstyp	SuperFacility	Superkonzept Sportstätte
Entitätstyp	Lido	Konzept Freibad
Annotationstyp	Admission	Strukturiertes Attribut Eintritt
Annotationstyp	AdmissionType	Eintrittsart (Erwachsene, Kinder, etc.)
Annotationstyp	Amount	Geldbetrag (für den Eintritt)
Annotationstyp	Pool	Strukturiertes Attribut Schwimmbecken
Annotationstyp	Dimension	Strukturiertes Attribut Dimension
Annotationstyp	Name	Name (des Freibads)
Annotationstyp	Location	Ort
Annotationstyp	Length	Länge (eines Schwimmbeckens)
Annotationstyp	Width	Breite (eines Schwimmbeckens)
Annotationstyp	Height	Höhe (eines Schwimmbeckens)

Im ersten Schritt wird die Klassifikationsrelation $R_{classifcation_3}$ berechnet:

Ausprägung 4.6.13 (Klassifikationsrelation $R_{classification_3}$)

$$K_1 = \{\texttt{SportsFacility}\}$$
$$E_3 = \{\texttt{Lido}\}$$
$$R_{classification_3} = \{(\texttt{SportsFacility}, \texttt{Lido})\}$$

Das Konzept *Schwimmbecken* erbt in diesem Beispiel alle Attribute vom Superkonzept *Sportstätte*, was im zweiten Schritt zur Berechnung der Vererbungsrelation $R_{inheritance_2}$ führt:

Ausprägung 4.6.14 (Vererbungsrelation $R_{inheritance_2}$)

$$\begin{aligned} S_1 &= \{\texttt{SuperFacility}\} \\ E_3 &= \{\texttt{Lido}\} \\ R_{inheritance_2} &= \{(\texttt{SuperFacility}, \texttt{Lido})\} \end{aligned}$$

Im dritten Schritt wird die Mitgliedschaftsrelation $R_{membership_6}$ berechnet. Die Menge A_3 beinhaltet alles Annotationstyp-Knoten, die in Tabelle 4.9 aufgeführt sind.

Ausprägung 4.6.15 (Mitgliedschaftsrelation $R_{membership_6}$)

$$\begin{aligned} E_3 \cup A_3 \cup S_1 = \{&\texttt{SuperFacility}, \texttt{Lido}, \texttt{Pool}, \texttt{Dimension}, \texttt{Name}, \texttt{Location}, \texttt{Admission}, \\ &\texttt{AdmissionType}, \texttt{Amount}, \texttt{Temperature}, \texttt{Width}, \texttt{Length}, \texttt{Height}\} \\ R_{membership_6} = \{&(\texttt{SuperFacility}, \texttt{Name}), \\ &(\texttt{SuperFacility}, \texttt{Location}), (\texttt{SuperFacility}, \texttt{Admission}), \\ &(\texttt{Admission}, \texttt{AdmissionType}), (\texttt{Admission}, \texttt{Amount}), \\ &(\texttt{Lido}, \texttt{Pool}), (\texttt{Pool}, \texttt{Dimension}), (\texttt{Pool}, \texttt{Temperature}) \\ &(\texttt{Dimension}, \texttt{Width}), (\texttt{Dimension}, \texttt{Length}), (\texttt{Dimension}, \texttt{Height})\} \end{aligned}$$

Der vierte Schritt behandelt die Konstruktion des Typgraphen G_{T_8}:

Konstruktionsschema 4.6.8 (Typgraph G_{T_8})

$$\begin{aligned} G_{T_8} &= (V(G_{T_8}), E(G_{T_8}) \subseteq V(G_{T_8}) \times V(G_{T_8})) \\ V(G_{T_8}) &= E_3 \cup K_1 \cup A_3 \cup S_1 \\ E(G_{T_8}) &= R_{classification_1} \cup R_{membership_6} \cup R_{inheritance_2} \end{aligned}$$

Am Ende steht die Visualisierung des ermittelten Typgraphen G_{T_8} in Abbildung 4.10:

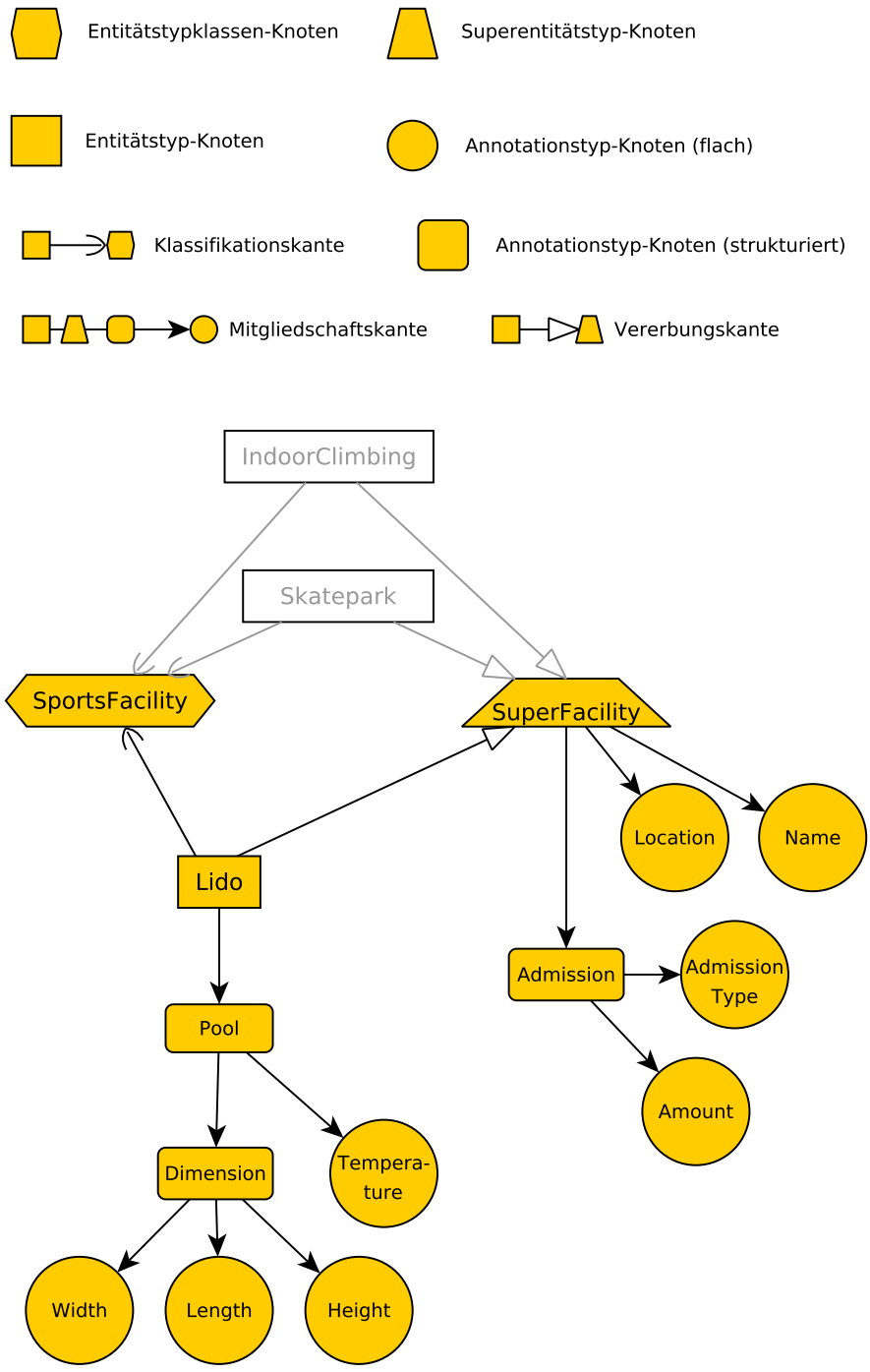

Abbildung 4.10: Typgraph Verschachtelung strukturierter Typen (G_{T_8})

4.6.12 Aggregation

Das vorliegende Konzept erlaubt bisher nicht, Entitäten miteinander in Beziehung zu setzen. Diese erhebliche Einschränkung soll mit der Einführung von Aggregations-Knoten behoben werden.

Beziehungen werden im Bereich der Datenmodellierung häufig als Assoziationen bezeichnet und erlauben es in der Regel mindestens zwei Objekte miteinander zu verbinden. Die Semantik der Assoziation ist dabei unbestimmt, d. h. die Existenz einer Assoziationsbeziehung trifft keine weitere Aussage über potentielle Rollen, die die beteiligten Entitäten spielen.

In der Praxis muss häufig ausgedrückt werden, dass Objekte in einer Teil-Ganzes-Beziehung stehen, um eine hierarchische Strukturierung organisieren oder Existenzbedingungen formulieren zu können.

Für den ersten Fall kann die sogenannte Aggregation verwendet werden, die ein Spezialfall der Assoziation darstellt, da sie zusätzlich zu der Beziehungssemantik die Gleichberechtigung der teilnehmenden Entitäten aufhebt und sie einander unterordnet. Diese Beziehung ist in der Regel gerichtet, d. h. es gibt eine Ganzheit, die Aggregat (engl., Sg.: *aggregate*) genannt wird und die untergeordneten Teile, die an dieser Stelle als Komponenten (engl., Sg.: *component*) bezeichnet werden. Bei der Aggregation findet keine existenzielle Bindung zwischen dem Aggregat und seiner Komponenten statt. Endet der Lebenszyklus des Aggregats, so werden die beteiligten Komponenten aus der Beziehung entlassen und bestehen fort.

Für den zweiten Fall kommt das Konzept der Komposition infrage, mit dem sich Entitäten existenziell aneinander binden lassen. Im Gegensatz zur Aggregation erlischt mit der Existenz des Aggregats auch die Existenz der Komponenten. Die Komposition kann als Spezialfall der Aggregation angesehen werden, da sie ebenfalls hierarchisch strukturiert und außerdem strenge Existenzbedingungen an die untergeordneten Teile formuliert.

Sowohl das Konzept der Aggregation als auch der Komposition ist aufgrund seiner maßgeblichen Bedeutung auch beim Entity-Relationship-Modell umgesetzt worden. Hier kann die Aggregation über eine sogenannte „has-a“ und die Komposition über eine „is-part-of“-Beziehung dargestellt werden.

In Anlehnung an die Benennung der dargestellten Beziehungsmodi in der UML besitzt ein Aggregations-Knoten entweder den Typ *shared* für die Repräsentation der Aggregation oder den Typ *composite* für die Darstellung von Komposition. Es gibt in

der UML noch eine weitere Art der Aggregation, die als *none* bezeichnet wird und die Teil-Ganzes-Beziehung zu einer klassischen Assoziation degradiert. Dieser Typ ist jedoch ungültig für den Aggregations-Knoten, da reine Assoziationen über später eingeführte Beziehungstypen (engl., Sg.: *relationships*) realisiert werden.

4.6.12.1 Monomorphe Aggregation

In den vorherigen Beispielen wurde das Konzept Schwimmbecken als strukturierter Datentyp *Pool* modelliert und ist damit Teil einer gegebenen Instanz des Entitätstyps Freibad (*Lido*). Angenommen, *Pool* würde ebenfalls als eigenständiger Entitätstyp modelliert, so benötigen wir ein Konzept, das die Verknüpfung zwischen der Freibad-Instanz und der Schwimmbecken-Instanz unter Erhaltung der Semantik abbilden kann. Gemäß der oben zuvor beschriebenen Technik kann also die Komposition verwendet werden, wenn wir davon ausgehen, dass ein Schwimmbecken an die Existenz des Schwimmbads gebunden werden soll.

Das Modell in Grafik 4.11 zeigt eine entsprechende Umformulierung des vorangegangenen Beispiels 4.9 aus dem letzten Abschnitt. Das Konzept *Pool* wird hier als eigenständiger Entitätstyp modelliert und über einen Aggregations-Knoten *Pool* als Komponente des Typs *Lido* subordiniert. Der Aggregations-Knoten ist über eine spezialisierte Mitgliedschaftskante mit dem Freibad-Typ verbunden, die anstelle eines Pfeils einen schwarzen Diamanten aufweist und so die Komposition kennzeichnet. Diese spezielle Mitgliedschaftskante (engl., Sg.: *composition membership edge*) erbt dabei alle Möglichkeiten zur Ausgestaltung von Integritätsbedingungen, wie sie zuvor für normale Mitgliedschaftskanten eingeführt wurden. In diesem Beispiel können maximal vier unterschiedliche Schwimmbecken aggregiert werden, was durch die Kardinalitätsangabe `1..4` angezeigt wird. Außerdem müssen sich diese Schwimmbecken in ihrer Identität unterscheiden, damit die Integritätsbedingung bezüglich der Einzigartigkeit erfüllt werden kann.

Die von dem Aggregations-Knoten ausgehende Kante (engl., Sg.: *aggregation edge*) gibt Aufschluss über den für diese Komposition erlaubten Typ und endet mit einem sogenannten Krähenfuß in diesem Beispiel am Entitätstyp-Knoten *Pool*.

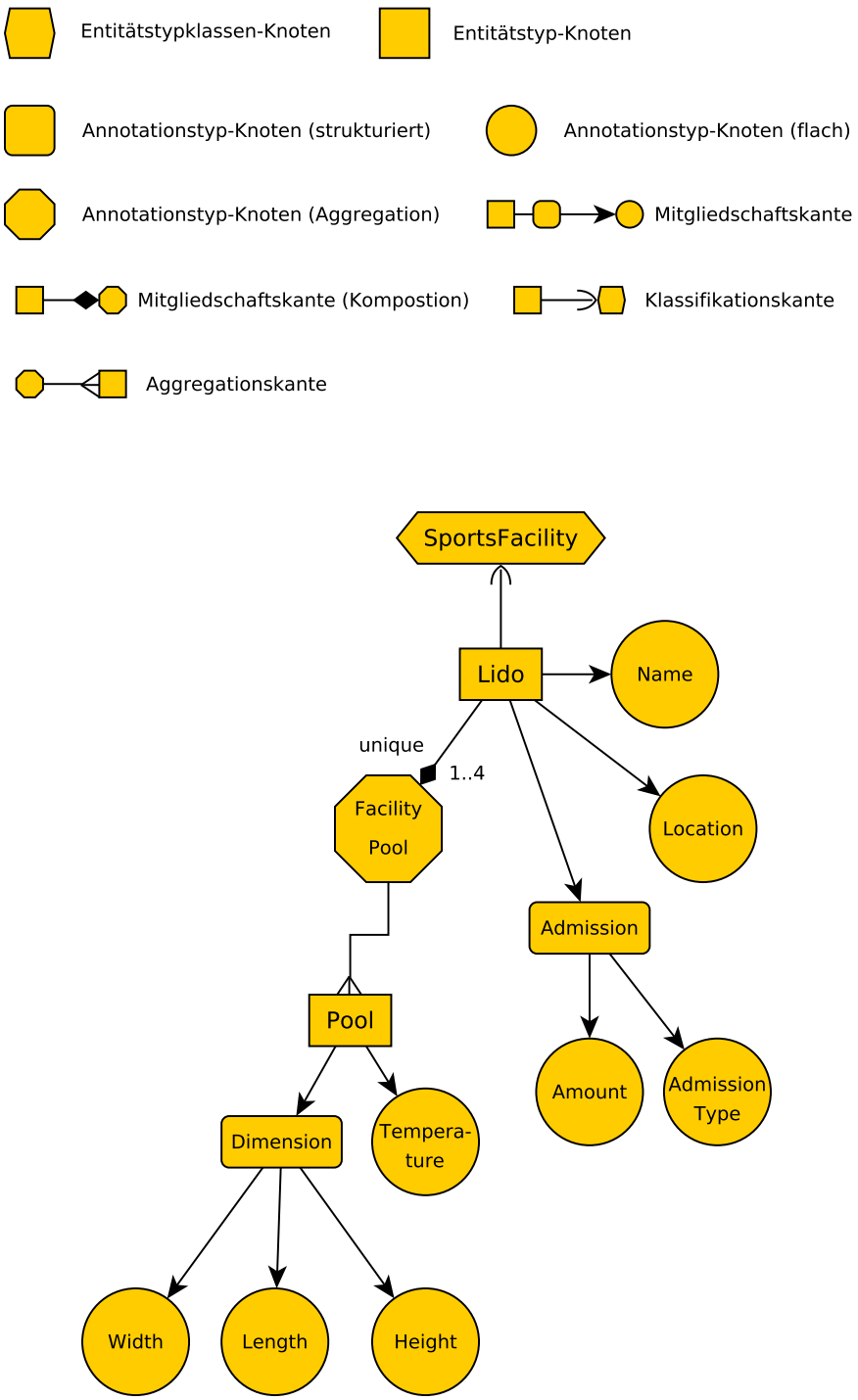

Abbildung 4.11: Typgraph Aggregationsbeispiel (G_{T_9})

4.6.12.2 Polymorphe Aggregation

Monomorphe Aggregation bzw. Komposition ist relativ unflexibel, da lediglich Instanzen erlaubt sind, die genau den angegebenen Typ besitzen. Es kann die Anforderung bestehen, eine bestimmte Gruppe von Typen zuzulassen, um Integritätsbedingungen großzügiger gestalten zu können. Aus diesem Grund erlaubt das vorliegende Konzept, Aggregationskanten nicht nur auf konkrete Entitätstypen zu richten, sondern auch auf alle Arten von Knoten, die eine Kategorisierung realisieren. So können Aggregationskanten auch Superentitätstyp-Knoten, Schnittstellen-Knoten, Klassen-Knoten und Cluster-Knoten referenzieren und erlauben so die Beziehung zu Entitätstypen, die über die angegebenen Konstrukte erreichbar sind. Grafik 4.12 zeigt ein entsprechendes Beispiel, bei dem ein Superentitätstyp namens *SuperPool* eingeführt wurde, der mittels der neu angelegten Typen *Swimmer* und *Nonswimmer* spezialisiert wird. Der Aggregations-Knoten *Pool* verweist nun auf *SuperPool*, das bedeutet für die Ausprägung einer Beziehung sind nun sowohl Instanzen des Typs Schwimmer- als auch Nichtschwimmerbecken erlaubt.

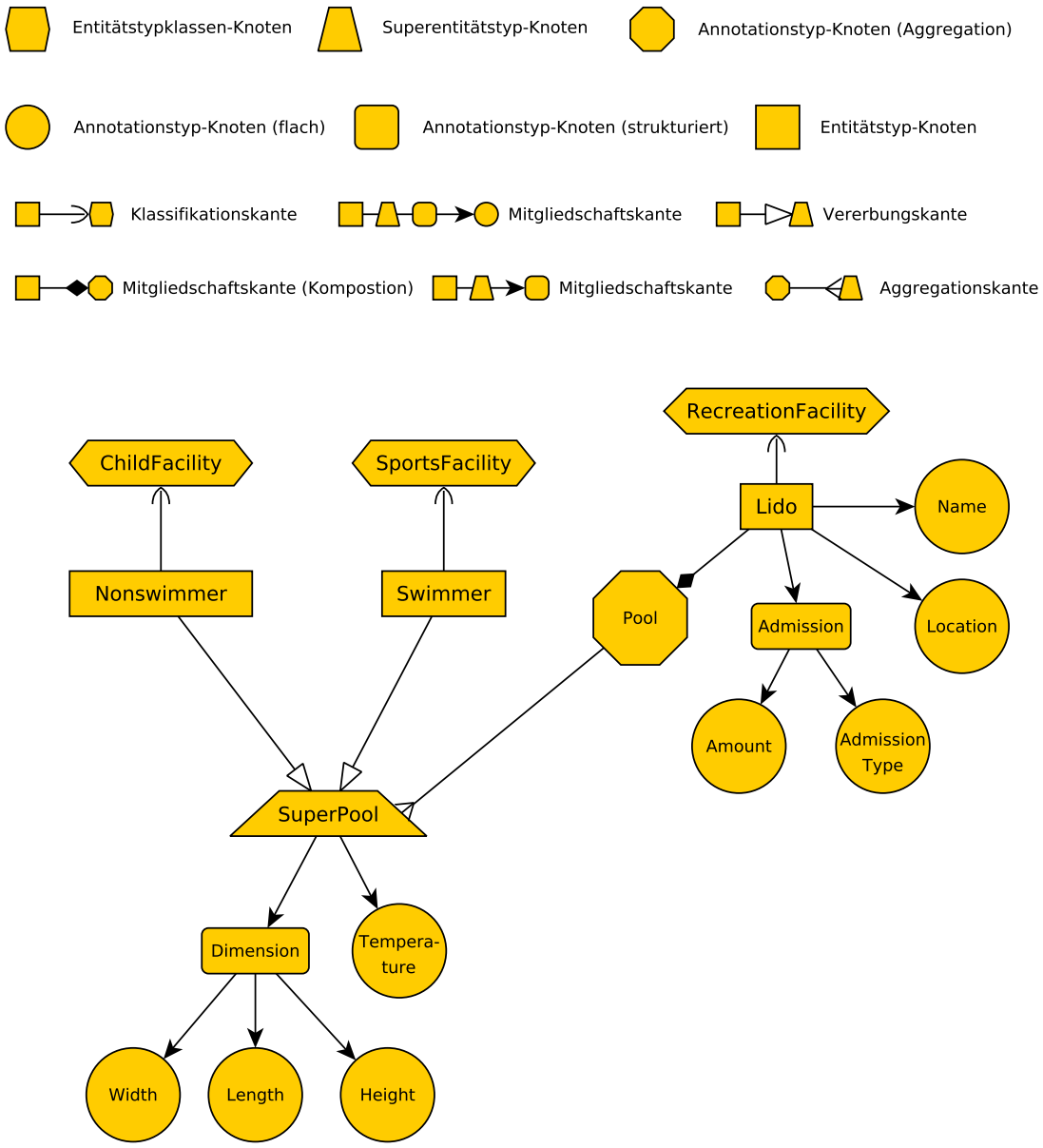

Abbildung 4.12: Typgraph Aggregationsbeispiel (polymorph) ($G_{T_{10}}$)

4.6.12.3 Rollen

Das Konzept der Rollen ermöglicht bei einer Aggregation den unterscheidbaren Zugriff auf denselben Komponententyp durch Zuweisung eines semantischen Ersatznamens. Dieser Zusammenhang soll durch ein Beispiel verdeutlicht werden.

Der vorangehende Abschnitt zeigt eine Modellierung, bei der die Entitätstypen Schwimmer- und Nichtschwimmerbecken explizit von einem Supertyp *SuperPool* abgeleitet werden, obgleich sie keine eigenen Merkmale besitzen. Die Einführung des Supertyps ist in diesem Fall einzig damit begründet, unterschiedliche Arten von Schwimmbecken auch namentlich differenzieren zu können. Diese Einführung von trivialen Subtypen zur Verbesserung der Semantik stellt allerdings eine suboptimale Lösung dar. Eine Alternative zu trivialen Subtypen ist die Verwendung von Rollen, wie sie in Grafik 4.13 zu erkennen ist.

Anstelle der Einführung von zusätzlichen Entitätstypen wurden zwei Aggregations-Knoten erstellt, die beide auf den Entitätstyp-Knoten *Pool* gerichtet sind und die gewünschten Rollennamen *Swimmer* und *Nonswimmer* tragen. Mithilfe dieser Bezeichnungen kann von einer Freibad-Instanz unter verschiedenen Namen zu den aggregierten Schwimmbecken-Instanzen navigiert werden.

Da, wie bereits angemerkt wurde, der Aggregations-Knoten einen spezialisierten Annotationstyp-Knoten darstellt, der über eine Mitgliedschaftskante angebunden wird, können auch die hierfür entsprechend spezifizierten Möglichkeiten genutzt werden. So ist in der Beispielgrafik aufgeführt, dass die vorherige Klasse *RecreationaleFacility* in einen Supertyp umgewandelt wurde, von dem nun die Typen *Lido* für Freibad und *SportsBath* für Sportbad erben, die beiderseits unter definierten Integritätsanforderungen Beziehungen zum Typ *Pool* herstellen. Für ein Freibad gilt hier exemplarisch, dass mindestens ein Nichtschwimmerbecken vorhanden ist, während die Existenz eines Schwimmerbeckens optional ist. Für ein Sportbad hingegen soll hier gelten, dass mindestens zwei Schwimmerbecken nutzbar sind.

Dieses Beispiel beinhaltet lediglich Rollennamen für das Ziel der Aggregation. Es ist jedoch auch möglich, Rollennamen für Aggregate zu vergeben, wie in Typgraph 4.11 zu sehen ist. In diesem Fall kann von Quellrollen (engl., Sg.: *source role*) im Gegensatz zu Zielrollen (engl., Sg.: *target role*) gesprochen werden. Die Angabe der Quellrolle erfolgt ebenfalls im Aggregations-Knoten über der Bezeichnung für die Zielrolle.

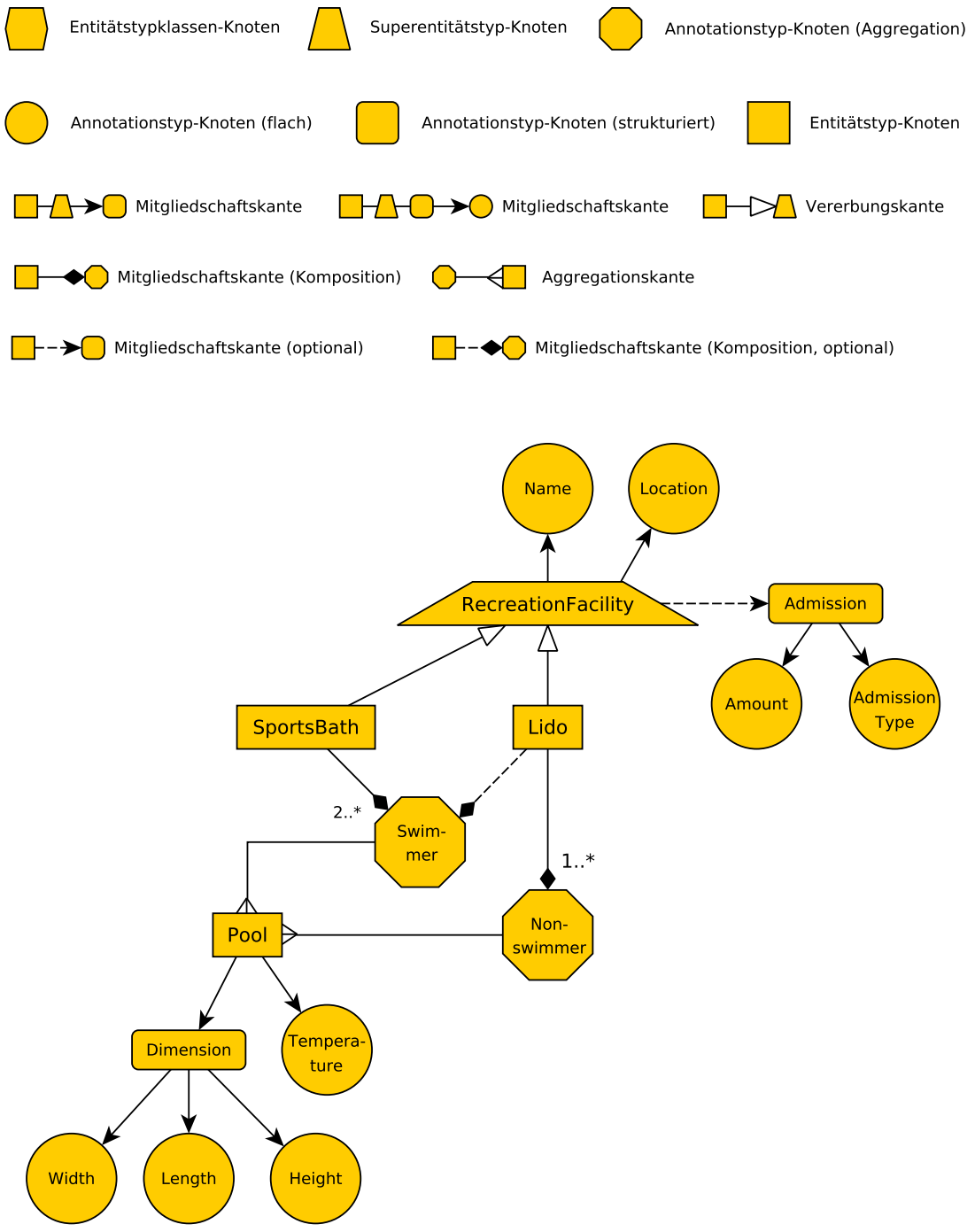

Abbildung 4.13: Typgraph Aggregationsbeispiel (polymorph, mit Rollen) ($G_{T_{11}}$)

4.6.12.4 Reflexivität

Entitätstypen können auch mit sich selbst in Beziehung stehen. Ein häufiges Beispiel für eine solche reflexive Beziehung ist die Modellierung eines Entitätstyps, der Personen repräsentiert und zum Beispiel Eltern- bzw. Kind-Instanzen aggregiert, die ebenfalls vom Typ *Person* sind.

4.6.13 Einschränkungsgruppen

Bisher konnten lediglich Aussagen über das optionale Vorhandensein von Mitgliedern getroffen werden. Die Einführung der Einschränkungsgruppe erlaubt die Definition von logischen Aussagen über die Existenzbeziehung ihrer Mitglieder bzw. von Bedingungen für deren Werteverhältnisse.

Angenommen in einer Datenbank sollen Informationen über Sportereignisse (*SportsEvent*) gespeichert werden. Zu jedem Ereignis wird der Titel (*Title*), das Datum (*Date*) und der Veranstaltungsort (*Venue*) hinterlegt, für den spezielle Anforderungen existieren, da er sich per Aggregation entweder auf eine Sporthallen- (*SportsHall*) oder eine Stadion-Instanz (*Stadium*) bezieht.

Das Beispielschema 4.14 zeigt einen möglichen Typgraph, der diese Anforderungen umsetzt. Die Einschränkungsgruppe wird dabei als Trapez gezeichnet. Die Mitgliedschaftskante ist außerdem mit einem spitzen, offenen sowie geschwungenen Pfeil versehen und trägt die englische Bezeichnung der verwendeten Verknüpfung, in diesem Fall `XOR`.

Für die Definition von Existenzbedingungen stehen die logischen Verknüpfungen `und (and)`, `oder (or)` bzw. `exklusives oder (xor)` zur Verfügung, die sich auf beliebig viele Stellen anwenden lassen, und die aufgrund der Kommutativität keine spezielle Reihenfolge benötigen.

Für die Werte von Merkmalen können Bedingungen mit Hilfe der Vergleichsoperatoren `>`, `<`, `=`, `>=`, `<=` und `!=` formuliert werden. In diesen Fällen hängt die Auswertung von der Position der Werte innerhalb der Terme ab. Außerdem sind die Operatoren binär, sie lassen also maximal zwei Operanden zu. Für die Umsetzung von Wertevergleichen bedarf es also weiterer Überlegungen zum Beispiel über Einführung einer Ordnung für Kindknoten.

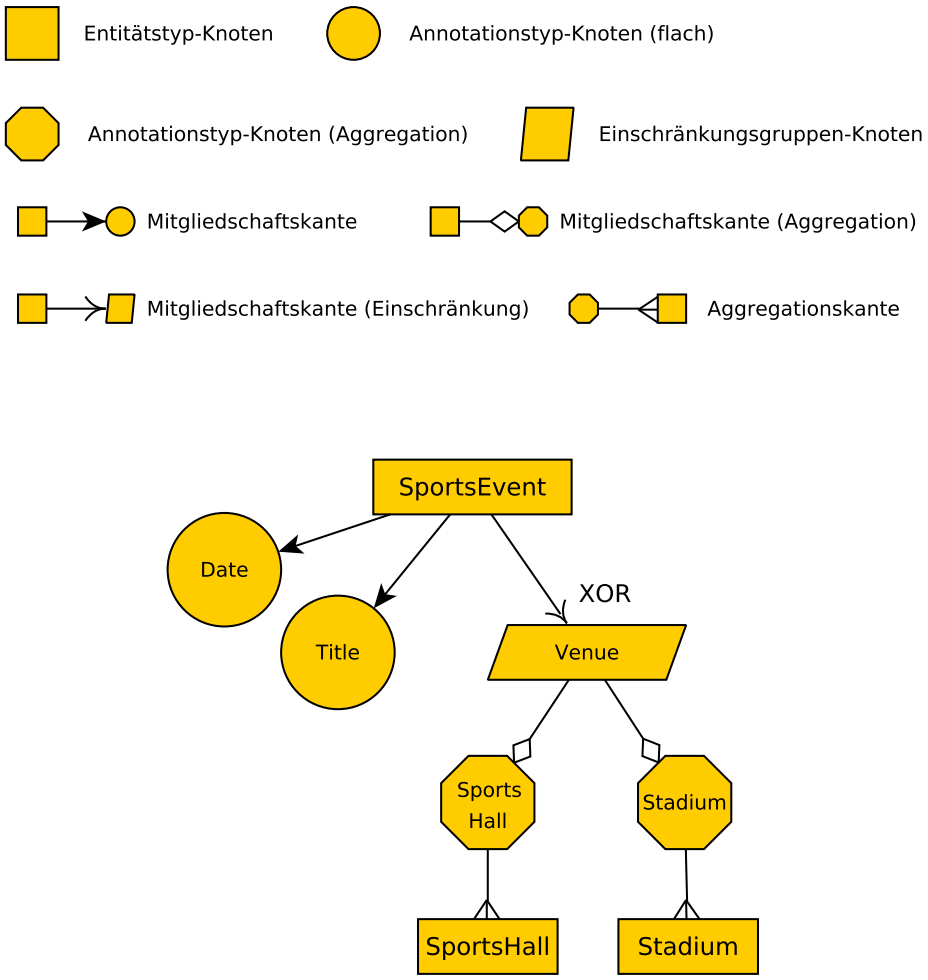

Abbildung 4.14: Typgraph Einschränkungsgruppe ($G_{T_{12}}$)

4.6.14 Beziehungstypen

Entitätstypen können mit Hilfe von Beziehungstypen assoziiert werden. Beziehungstypen stellen dabei eine Spezialisierung von Entitätstypen dar und werden im Rahmen dieses Konzepts daher genauer als Beziehungsentitätstypen (engl., Sg.: *relationship entity type*) bezeichnet. Im Rahmen einer Schemadefinition kann festgelegt werden, welche Entitätstypen in Beziehung treten und welchen quantitativen Partizipationsrestriktionen sie dabei unterliegen. Letzteres wird durch die Angabe von Kardinalitäten ermöglicht.

Bei der Angabe von Kardinalitäten ist die Unterscheidung wichtig, ob Beziehungen oder Entitäten gezählt werden. Im ersten Fall wird zum Beispiel mittels der (min,max)-Notation angegeben, an wie vielen Beziehungstypausprägungen eine Entität teilnehmen darf. Im zweiten Fall werden nicht die Beziehungen gezählt, sondern zum Beispiel mittels Angabe von Multiplizitäten festgelegt, wie viele Entitäten des einen Teilnehmers mit wie vielen Entitäten des anderen Teilnehmers verbunden sein dürfen.

Grundsätzlich können also (a) die Beteiligungen von Entitäten an Beziehungen oder (b) die an einer Beziehung beteiligten Entitäten gezählt werden.

Die Definition der Kardinalitäten erfolgt entweder als fixe Zahl oder in Form von Bereichsangaben, um Integritätsbedingungen flexibler gestalten zu können.

Bei der Definition eines Beziehungstyps spielt zunächst die Anzahl der beteiligten Entitätstypen eine Rolle. Für die folgenden Beispiele werden binäre Beziehungen mit maximal zwei Teilnehmern modelliert. Das Konzept erlaubt ebenso ternäre, quaternäre bzw. höhergradige Beziehungskonstellationen.

Grundsätzlich werden drei Beziehungsarten unterschieden, für die grundlegende Kardinalitäten angegeben werden können:

1:1 Einem Element der Entitätsmenge des Entitätstyps A ist genau ein Element der Entitätsmenge des Entitätstyps B zugeordnet und umgekehrt.

1:n Einem Element der Entitätsmenge A sind $n \in \mathbb{N}$ Elemente der Entitätsmenge B zugeordnet. Ein Element aus B wiederum darf nur zu einem Element aus A in Beziehung stehen.

n:m Aus der Entitätsmenge A dürfen $n \in \mathbb{N}$ Elemente in Beziehung zu $m \in \mathbb{N}$ Elementen der Entitätsmenge B stehen und umgekehrt.

In den folgenden Abschnitten wird für jeden angegeben Fall ein Beispiel konstruiert und darauf eingegangen, auf welche Weise Kardinalitäten als Integritätsbedingungen behandelt werden und wie weiterhin Entitätsbeziehungstypen über die bisher vorgestellten Typgraphelemente modelliert werden können.

4.6.14.1 Beziehungstyp 1:1

In diesem Beispielszenario soll ausgedrückt werden, dass ein Staatsbürger genau einen Personalausweis besitzt, und dass umgekehrt ein Personalausweis eindeutig

einem Staatsbürger zugeordnet werden kann. Ein informelles ER-Diagramm zu diesem Sachverhalt sei mit Abbildung 4.15 gegeben.

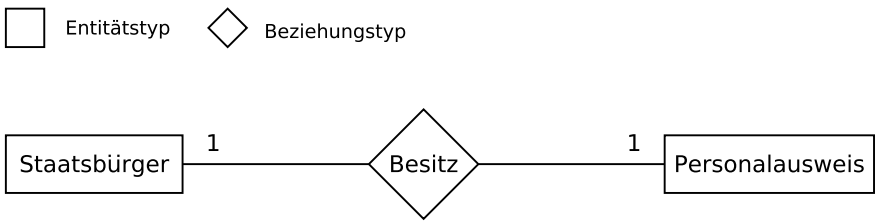

Abbildung 4.15: ER-Diagramm 1:1-Beziehung

Die Kardinalitätsangaben an den jeweiligen Kantenenden drücken die 1:1-Beziehung (engl., Sg.: *one-to-one relationship*) der beiden beteiligten Entitätstypen *Staatsbürger* sowie *Personalausweis* im Kontext des Beziehungstyps *Besitz* aus.

Formal betrachtet bildet die verwendete Beziehung eine *funktionale Abbildung* zwischen den Teilnehmern aus. Dies kann mathematisch über eine *partielle Funktion* ausgedrückt werden, die eine spezielle Relation darstellt und fordert, dass jedes Element in A genau einem Element in B zugeordnet wird. Diese Funktion ist nicht zwingend total, da ihr Definitionsbereich nicht alle Elemente in A beinhalten muss.

Für die folgende Definition wird ein potentieller Zustand σ betrachtet, in der S die Menge alle Staatsbürger-Instanzen und P die Menge aller Personalausweis-Instanzen darstellt. Die Menge B der Besitz-Beziehungstypinstanzen bildet dabei eine Teilmenge des kartesischen Produktes von S und P und wird in diesem Fall als partielle Funktion beschrieben.

$$\sigma(Besitz) : \sigma(S) \rightarrow \sigma(P) \; mit \; dom(Besitz) \subseteq \sigma(S) \tag{4.17}$$

Abbildung 4.16 zeigt einen Beispielzustand mit fünf Staatsbürger-, fünf Personalausweis- und drei Beziehungsausprägungen vom Typ *Besitz*.

Die Angabe von Kardinalitäten bilden Integritätsbedingungen, die beim Übergang in einen neuen Datenbankzustand erfüllt sein müssen. Für das aktuelle Beispiel bedeutet dies, dass für alle Entitäten die Teilnahme an Beziehungen gezählt werden. Das Ergebnis dieser Zählung darf für jede Entität höchstens 1 betragen, da die Partizipation an mehreren Beziehungen die Anforderung an eine funktionale 1:1-Abbildung nicht erfüllen würde.

Das gegebene informelle Beispielschema soll nun in einen äquivalenten Typgraph übersetzt werden. Hierfür wurde eingangs ein neues Syntaxelement eingeführt, das Beziehungsentitätstypen repräsentiert. Es wird als Diamant mit innenliegendem Rechteck gezeichnet, das mit dem Namen der Beziehung versehen ist. Die Teilnahme von Entitätstypen wird über das bereits vorgestellte Konzept der Aggregations-Knoten realisiert.

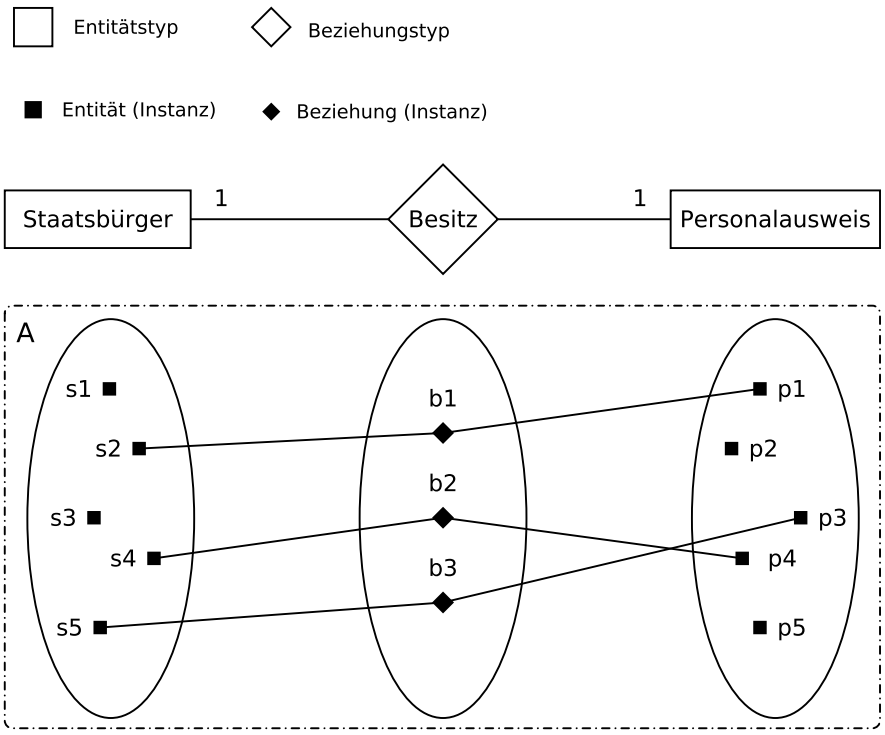

Abbildung 4.16: Ausprägungsbeispiel 1:1-Beziehung

Abbildung 4.17 zeigt eine entsprechende Schemaübersetzung mit englischen Typbezeichnungen, auf die kurz eingegangen werden soll.

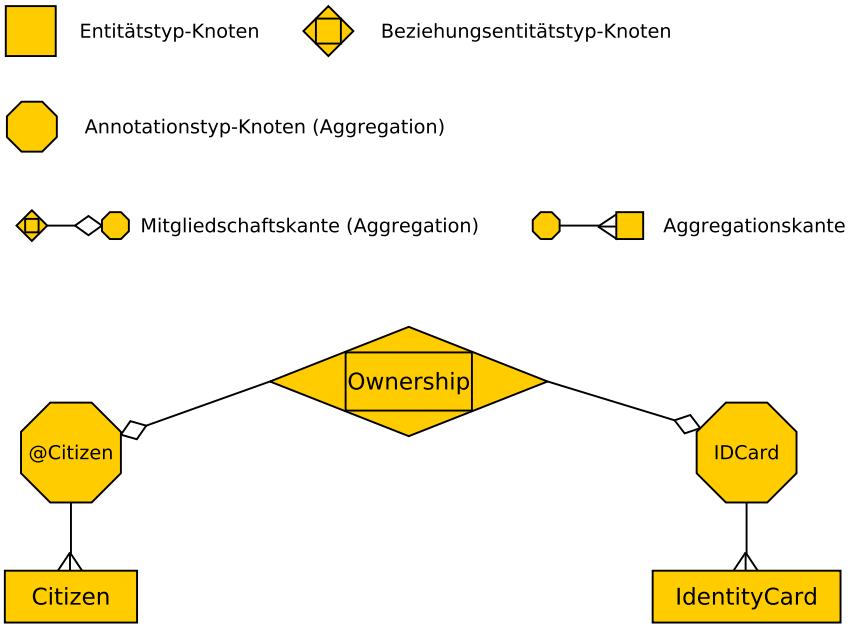

Abbildung 4.17: Typgraph 1:1-Beziehung ($G_{T_{13}}$)

Entsprechend der Vorlage wurden zwei Entitätstypen *Citizen* und *IdentityCard* eingeführt, die ihre Partizipation an dem Beziehungstyp *Ownership* über die Aggregations-Knotentypen *Citizen* und *IDCard* verwirklichen. Auf die Angabe der Kardinalitäten kann laut Typgraphsyntax im Fall des Wertes 1 verzichtet werden (siehe 4.3).

4.6.14.2 Beziehungstyp 1:n

Ein weiteres Beispielszenario behandelt den 1:n-Beziehungstyp (engl., Sg.: *one-to-many relationship*), in dem der Einsatz von Spielern für Mannschaften modelliert wird. Hierzu sei zunächst ein informelles Diagramm in Abbildung 4.18 gegeben, das die (min,max)-Notation für die Angabe der Kardinalitäten nutzt.

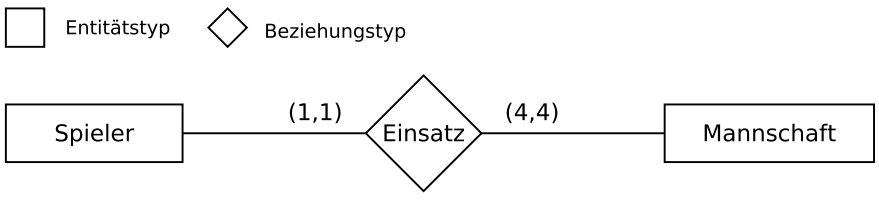

Abbildung 4.18: ER-Diagramm 1:n-Beziehung, Kardinalitäten in (min,max)-Notation

Diese Kardinalitätsangabe definiert die Anzahl der möglichen oder notwendigen Beteiligungen an Beziehungen und formuliert so entsprechende Integritätsbedingungen. Für das aktuelle Beispiel bedeutet dieses Vorgehen, dass eine *Mannschaft* genau 4-mal an der Beziehung *Einsatz* teilnehmen muss. Ein Spieler hingegen muss genau einmal teilnehmen. Die Kardinalität eines Beziehungstyps kann auch anders ausgedrückt werden, indem eine UML-Notation verwendet wird. Dies zeigt Abbildung 4.19.

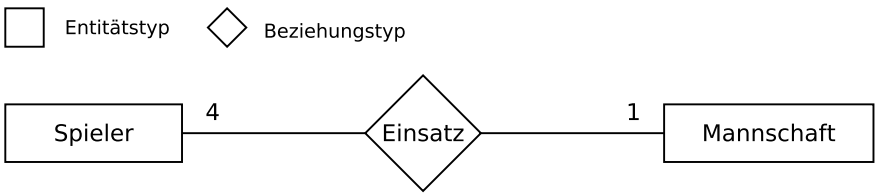

Abbildung 4.19: ER-Diagramm 1:n-Beziehung, Kardinalitäten in UML-Notation

Hier wird die Anzahl der möglichen oder notwendigen Beziehungen zwischen den Entitätstypen selbst dargestellt. Es ist direkt ablesbar, dass eine Mannschaft den Einsatz von 4 Spielern benötigt und 1 Spieler in höchstens einer Mannschaft eingesetzt werden kann.

Diese Art der Notation soll auch für das vorliegende Konzept verwendet werden, da sie einerseits einfacher zu lesen ist und sich andererseits mit der Syntax für die Angabe von Kardinalitäten bei multipler Mitgliedschaft von Aggregations-Knoten deckt, mit denen die Teilnahme schließlich umgesetzt wird. Gleichzeitig kann sie Informationen über die Anzahl der Beteiligungen an Beziehungstypen geben, wie im Folgenden aufgezeigt wird.

Grafik 4.20 zeigt einen Typgraph, der die vorgestellten Integritätsbedingungen berücksichtigt.

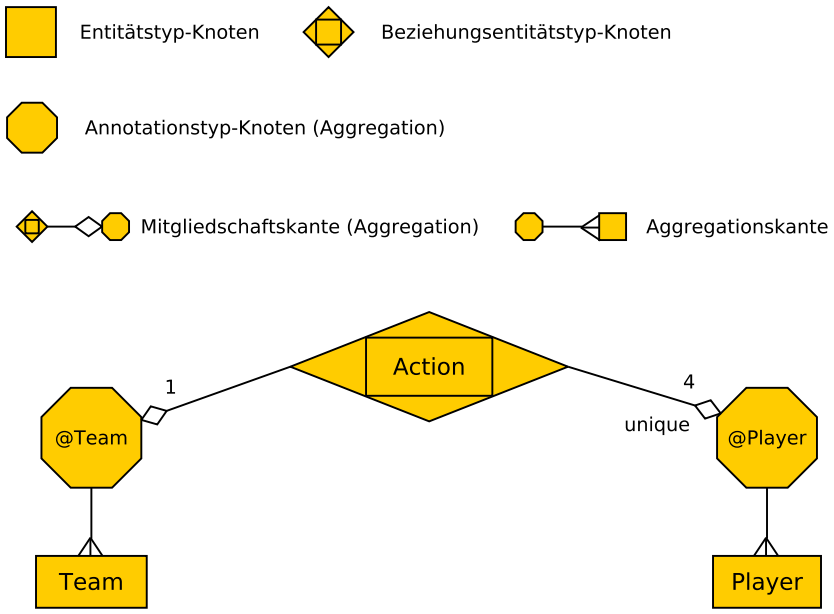

Abbildung 4.20: Typgraph 1:n-Beziehung ($G_{T_{14}}$)

Die Modellierung über multiple Mitgliedschaften bedeutet insbesondere, dass in einem abgeleiteten Datenbankzustand pro Abbildungsinstanz zwischen einer Mannschaft und 4 Spielern nur *eine* Ausprägung des Beziehungstyps *Action* existiert. Diese Ausprägung hält über Aggregations-Knoten 4 Instanzen vom Typ *Player* und eine Instanz vom Typ *Team*.

Grafik 4.21 verdeutlicht diesen Sachverhalt durch die vereinfachte Darstellung der Abbildung zwischen den in Beziehung stehenden Entitätsmengen. Abschnitt A zeigt dabei einen theoretisch korrekten, aber im Kontext des definierten Typgraphen illegalen Zustand im Vergleich zu Abschnitt B, der einen schemagetreuen Zustand beinhaltet.

Das Schema gibt weiterhin Aufschluss über die erlaubte Anzahl von Beziehungen, an denen die einzelnen Entitäten beteiligt sein dürfen, indem die Mitgliedschaftskardinalitäten kreuzgelesen werden. So ist auch bei dieser Notation ablesbar, dass eine *Team*-Instanz 4- und eine *Player*-Instanz 1-mal an einer *Action*-Beziehungstypinstanz partizipieren können.

Bei der Überprüfung der Beziehungskardinalitäten schlägt eine einfache Zählung der Beziehungstypinstanzen aufgrund der vorliegenden Modellierung fehl. Die Anzahl der Beteiligungen von Instanzen des Typs *Player* kann über die referenzierenden Aggregations-Knoten abgeleitet werden, indem deren Verwendermenge auf Instanzen des zur überprüfenden Beziehungstyps untersucht wird. Bei der Beteiligungsmessung

von Instanzen des Typs *Team* muss ebenfalls umgedacht werden, da diese aufgrund der Zusammenfassung auf eine einzige Beziehungsinstanz grundsätzlich falsche Ergebnisse liefert. An dieser Stelle muss die andere Teilnehmerseite der *Player*-Instanzen betrachtet werden, um festzustellen, ob hier tatsächlich 4 Instanzen aggregiert werden.

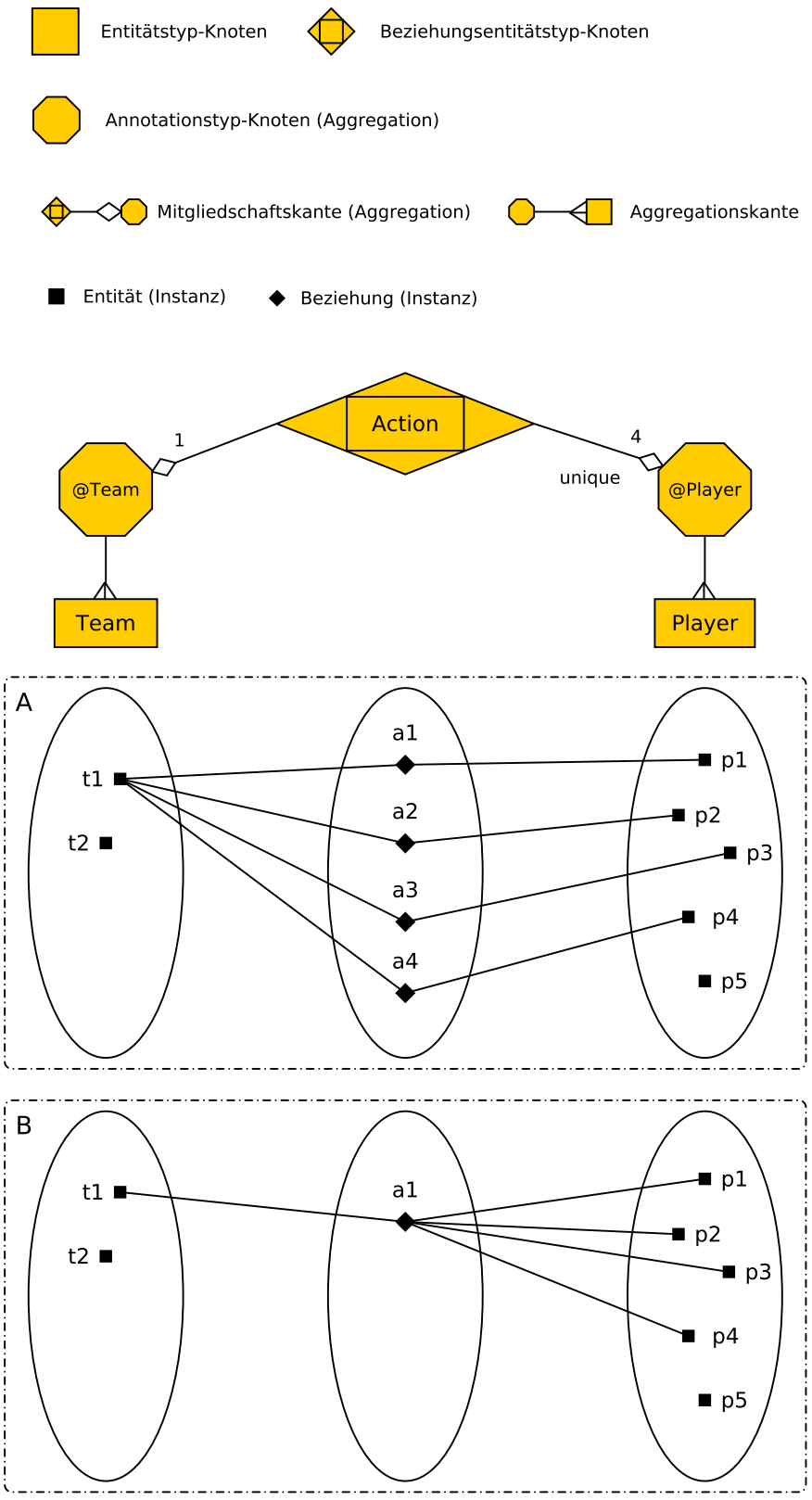

Abbildung 4.21: Ausprägungsbeispiel 1:n-Beziehung

4.6.14.3 Beziehungstyp n:m

Der n:m-Beziehungstyp (engl., Sg.: *many-to-many relationship*) erlaubt die beliebige Zuordnung zwischen den Elementen zweier in einer Beziehung stehenden Entitätsmengen. Er stellt damit den allgemeinen Fall einer 1:n-Beziehung dar, kann allerdings nicht naiv nach dem gleichen Prinzip modelliert werden.

Zur Begründung dieser Aussage soll ein Beispielzustand auf Grundlage des informellen ER-Diagramms in Abbildung 4.22 betrachtet werden, das aus einem Beispielschema für eine BibTeX-Datenbank von Martin Gogolla [Gog12] abgeleitet wurde.

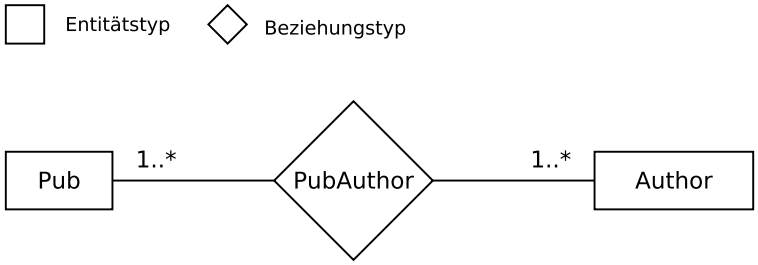

Abbildung 4.22: ER-Diagramm n:m-Beziehung, Kardinalitäten in UML-Notation

Der Beispielzustand sieht für zwei Instanzmengen der Typen *Pub* und *Author* folgende Beziehungsausprägungen vom Typ *PubAuthor* vor:

$$\sigma(PubAuthor) \subseteq \sigma(Publication) \times \sigma(Author) \tag{4.18}$$

$$\sigma(Publication) = \{p1,\ p2,\ p3\} \tag{4.19}$$

$$\sigma(Author) = \{a1,\ a2,\ a3\} \tag{4.20}$$

$$\sigma(PubAuthor) = \{(p1,\ a1),\ (p1,\ a4),\ (p2,\ a1),\ (p2,\ a2)\} \tag{4.21}$$

Bei Übertragung des Modellierungsprinzips aus dem vorangegangenen Abschnitt werden die an der Beziehung beteiligten Entitätstypen unter Zuhilfenahme von zwei multiplen Mitgliedschaften aggregiert, was in Abbildung 4.23 dargestellt wird.

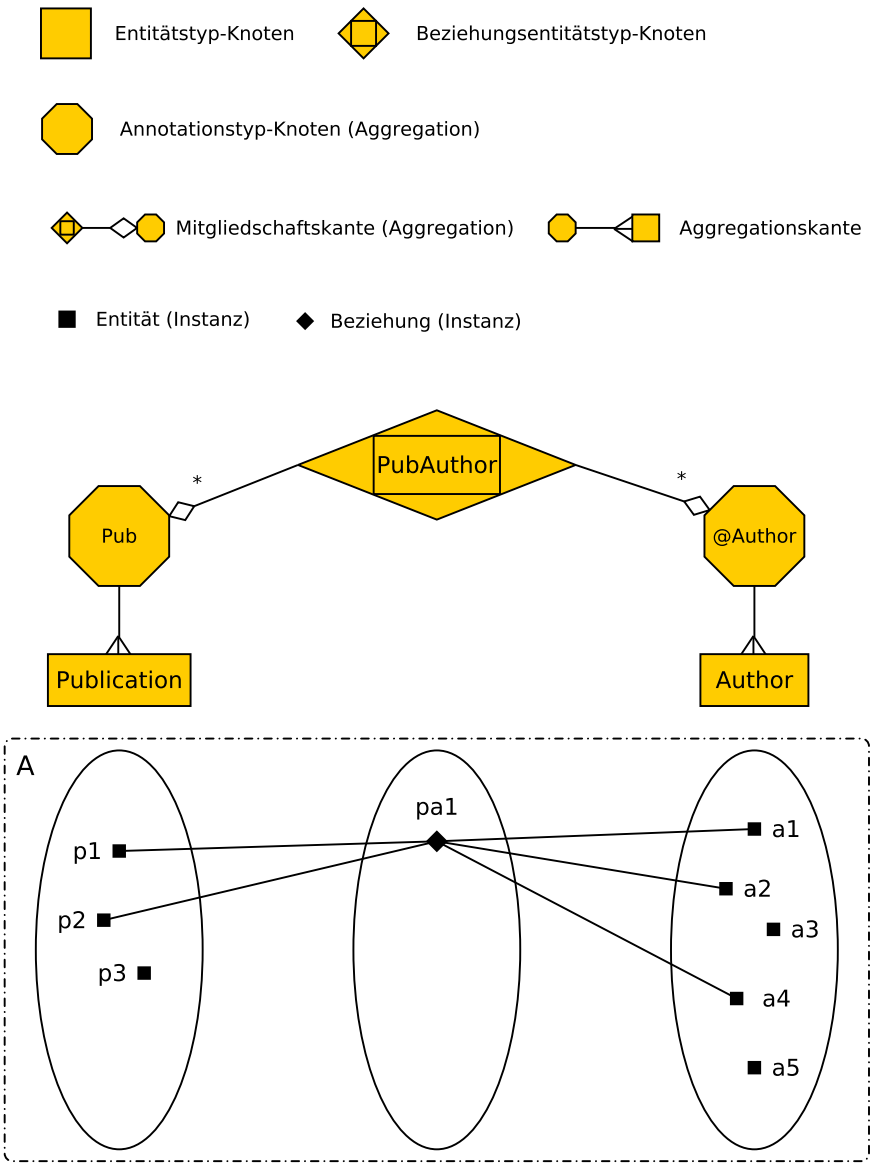

Abbildung 4.23: Ausprägungsbeispiel n:m-Beziehung, inkonsistenter Zustand

Bei näherer Betrachtung wird deutlich, dass dieses Vorgehen inadäquat ist, da es zu einem Verlust der korrekten Abbildung führt. Zwar beinhalten die in der Beispielgrafik ausgeprägten Relationen die Elemente von `pa1` und `pa2`, allerdings auch alle Elemente des Kreuzproduktes.

Dieses Problem soll behoben werden, indem eine Teilnehmerseite fixiert und die Modellierung so auf den 1:n-Fall reduziert wird. Dies bedeutet für die Ausprägungen vom Typ *Pub*, dass diese wie im Fall einer 1:n-Beziehung maximal einmal an einer

Beziehungstypinstanz teilnehmen dürfen. Für die nicht fixierte Seite der Ausprägungen vom Typ *Author* gilt diese Einschränkung nicht. Auf dieser Seite darf jedes Element n-mal an einer Beziehung vom Typ *PubAuthor* partizipieren.

Die Beziehungskardinalität ist für die Instanzen einer Teilnehmerseite sowohl im 1:n- als auch im n:m-Fall stets 1. Im 1:n-Fall gilt dies auch für die andere Teilnehmerseite, im n:m-Fall ist die Kardinalität gerade n.

Für die Beziehungsausprägungen selbst gilt, dass sie auf der multiplen Seite im 1:n-Fall n und im n:m-Fall m Mitgliedschaftsinstanzen besitzen.

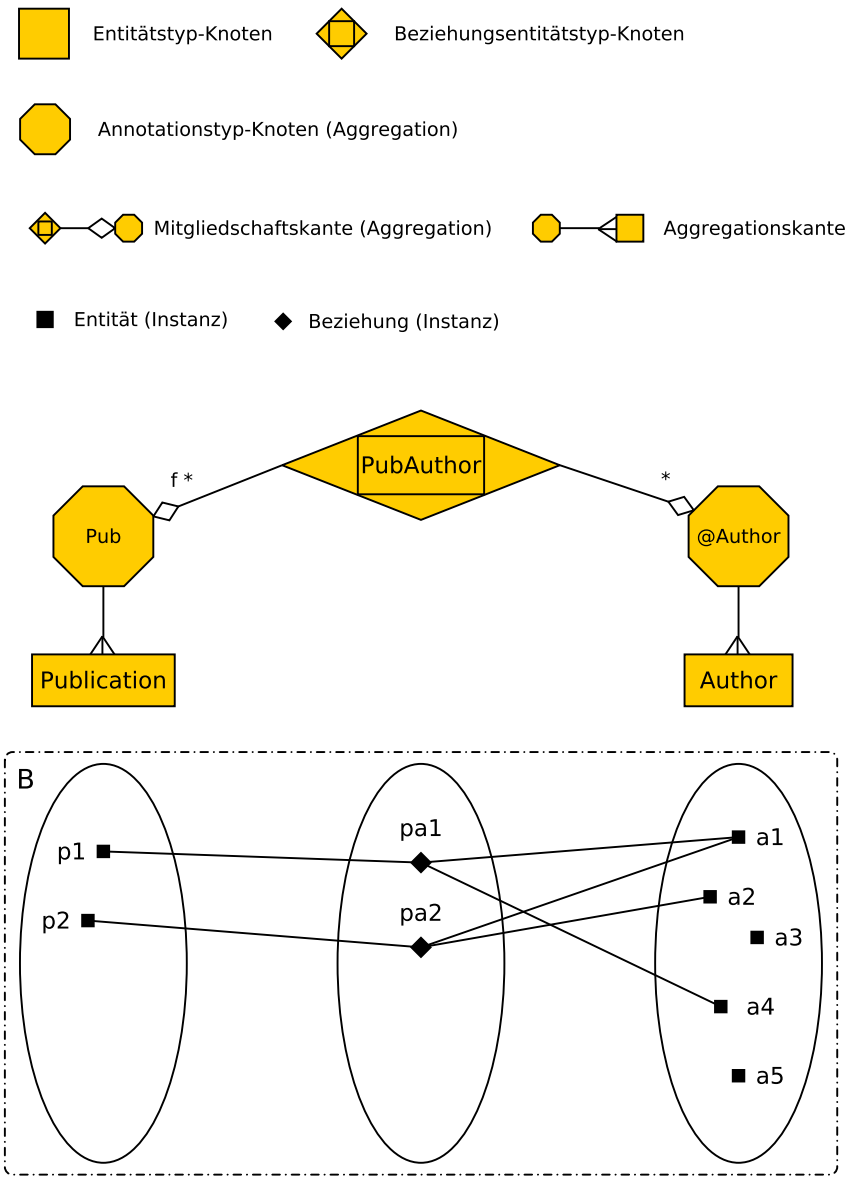

Abbildung 4.24: Ausprägungsbeispiel n:m-Beziehung, konsistenter Zustand

4.6.15 Clustering

Es wurden bereits verschiedene Möglichkeiten der Kategorisierung von Entitätstypen vorgestellt. Zum einen kann ein Entitätstyp einer Klasse zugeordnet werden, zum anderen kann er von Superentitätstypen erben. Auch das Erfüllen von Schnittstellen stellt eine Art Kategorisierung dar. Insbesondere für die flexible

Realisierung von Aggregationen zwischen Entitäten ist es jedoch sinnvoll, ein weiteres Klassifikationsinstrument einzuführen, das sich über Klassen, Supertypen, Schnittstellen und Typen erstreckt.

Hierfür werden der sogenannte Cluster-Knoten (engl., Sg.: *cluster node*) und die entsprechende Clustering-Relation $R_{clustering}$ verwendet, die entsprechende Beziehungen zu anderen Knoten über Clustering-Kanten herstellt. Die Clustering-Relation $R_{clustering}$ wird definiert als Teilmenge des kartesischen Produkts zwischen der Vereinigungsmenge $E \cup K \cup S \cup F$ und der Menge der Cluster-Knoten C:

$$R_{clustering} \subseteq \{(x, y) \mid x \in (E \cup K \cup S \cup F) \wedge y \in C\} \tag{4.22}$$

Cluster-Knoten können in einem Schema als Zielknoten für Aggregationen verwendet werden. Für die Ausbildung einer konkreten Beziehung wird bei Zuweisung einer Entität geprüft, ob deren Typ transitiv über ein mit dem Cluster-Knoten verbundenes Element erreichbar ist.

Beispielabbildung 4.25 zeigt ein Schema, in dem ein Kluster-Knoten namens *Location* den Entitätstyp *IndoorClimbing*, die Klasse *SportsFacility*, die Schnittstelle *Visitable* und den Supertyp *SuperFacility* generisch als Veranstaltungsorte kategorisiert. Auf diesen Cluster-Knoten nimmt der Aggregations-Knoten *Venue* Bezug und erlaubt so die generische Assoziation mit allen entsprechenden Entitätstypen.

Es besteht außerdem die Möglichkeit Cluster hierarchisch zu verknüpfen. Hierzu kann die Clustering-Relation $R_{clustering}$ als Teilmenge des kartesischen Produktes zwischen der Vereinigungsmenge $E \cup K \cup S \cup F \cup C$ und der Menge der Cluster-Knoten C definiert werden.

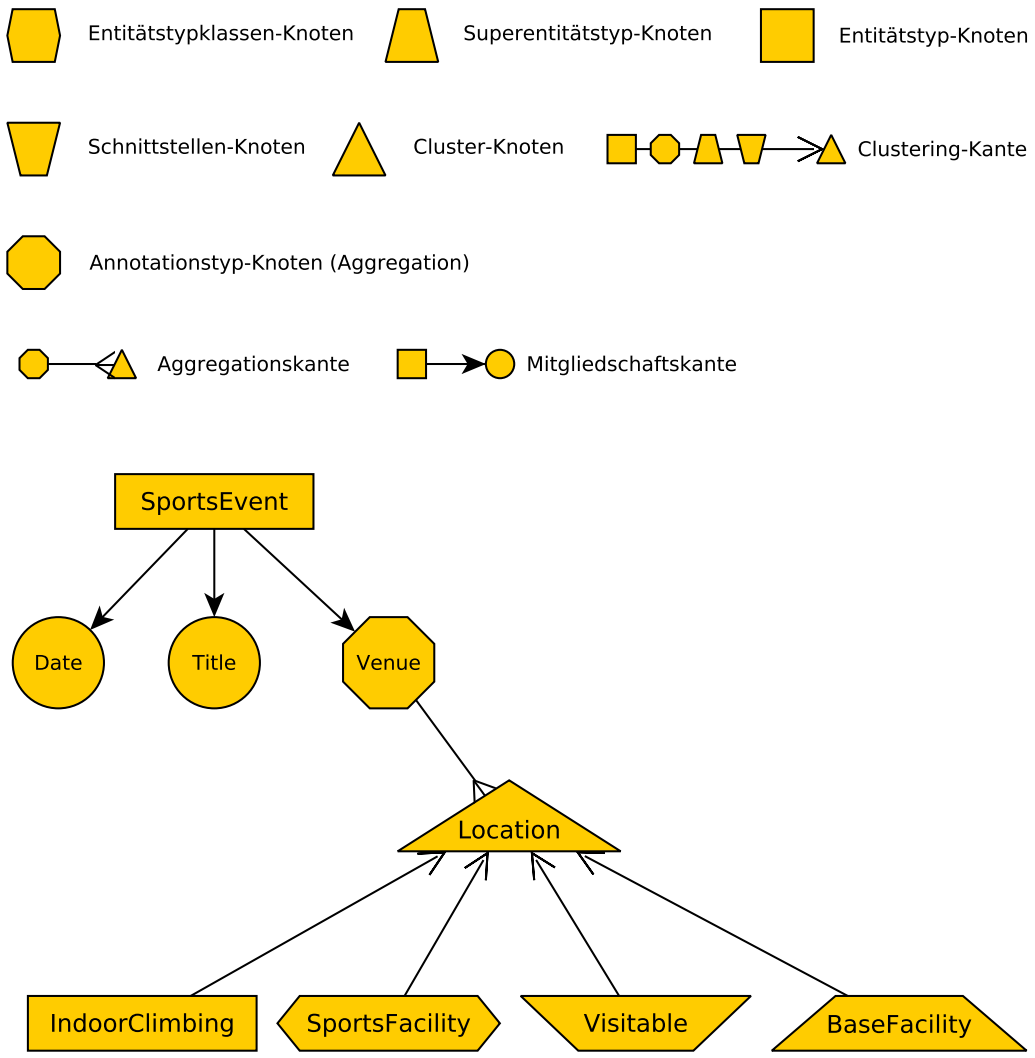

Abbildung 4.25: Typgraph Clustering-Beispiel ($G_{T_{15}}$)

4.6.16 Aliastypen

Mit Hilfe von Aliastypen ist es möglich, existierende Annotations-Knoten unter Umbenennung wieder zu verwenden.

Das Beispielschema in Abbildung 4.26 zeigt einen möglichen Einsatz für dieses Konzept. Anstelle separate Annotationstypen für Beginn- (*BeginDate*) und Enddatum (*EndDate*) einzuführen, findet eine Umbenennung mit Verweis auf das Original (*Date*) statt. Über Aliastypen wird so eine semantische Spezialisierung auf Ebene der Attributnamen realisiert.

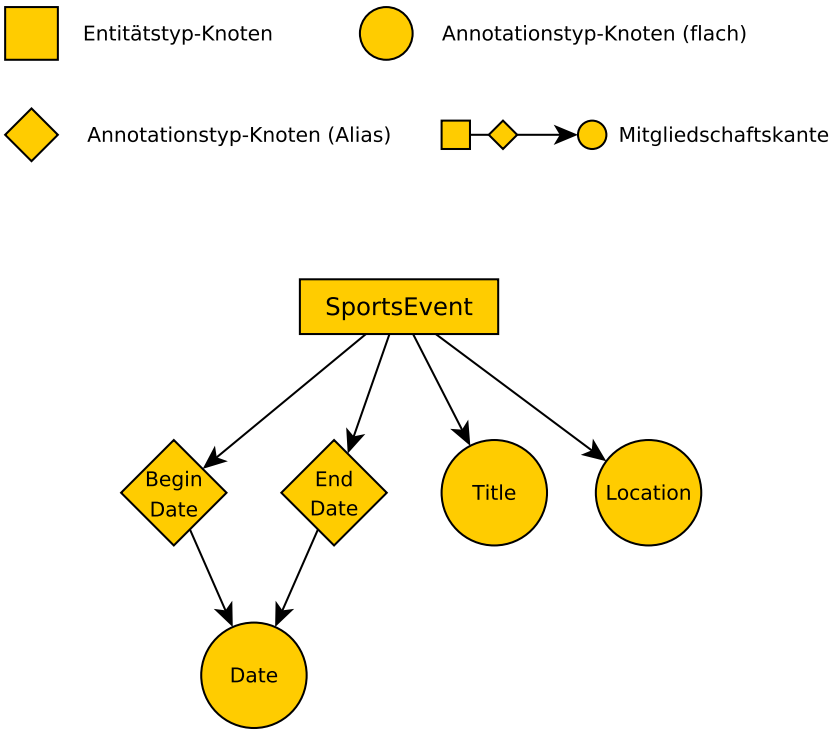

Abbildung 4.26: Typgraph Aliasbeispiel ($G_{T_{16}}$)

4.6.17 Knoten- und Kantendekorationen

Anhand des knapp eingeführten Modus für Annotationstyp-Knoten wird deutlich, dass Knoten neben ihrer eindeutigen Bezeichnung noch weitere Eigenschaften haben müssen, um sinnvolle Definitionen von Schemata zuzulassen. Tatsächlich gibt es eine ganze Reihe von zusätzlichen Knotendekorationen, mit deren Hilfe Regeln bzw. Einschränkungen bei der Verarbeitung des Schemas bzw. der Zustandsbildung festgelegt werden können. Auch die Kanten können mit Metainformationen versehen werden, um einen gewissen Spielraum bei der Gestaltung des Schemas zu ermöglichen.

Knoten respektive Kanten werden in der Regel mit zwei verschiedenen Arten von Dekorationen versehen. Es gibt zum einen Dekorationen, deren Existenz für die Definition eines gültigen Schemas notwendig sind und als notwendige Dekorationen (engl., Sg.: *required decoration*) bezeichnet werden sollen. Zum anderen gibt es sogenannte optionale Dekorationen (engl., Sg.: *optional decoration*), die für beliebige Metainformationen frei gesetzt werden können.

4.7 Aufbau von Instanzgraphen

4.7.1 Vorbemerkung

Der vorgestellte Typgraph bildet die Basis für die Ausprägung von Datenbankzuständen, in denen konkrete Daten gespeichert werden. Der Aufbau eines gültigen Datenbankzustandes geschieht durch Auswahl und Instanziierung von verfügbaren Entitätstypen unter Einhaltung der Integritätsbedingungen. Ein Entitätstyp wird ausgeprägt, indem seine Merkmale mit erlaubten Werten belegt und seiner Instanzrepräsentation eine eigene Identität zugewiesen wird. Aus einem Entitätstyp werden so Entitäten abgeleitet, die eigenständig existieren, und auf die mittels eindeutiger Identifikationsnummer (*id*) zugegriffen werden kann.

Der verwendete Datentyp für den Aufbau eines Datenbankzustands ist ebenfalls ein Graph, sodass von einer Transformation des Typgraphen in einen Instanzgraphen gesprochen werden kann, die an dieser Stelle nicht formal spezifiziert wird. Stattdessen werden die Eigenschaften dieser Übersetzung anhand von Beispielen erläutert.

4.7.2 Elemente des Instanzgraphen

Grafik 4.27 zeigt eine Kombination von Entitätstyp- und Entitäts-Diagramm und beschreibt im unteren Abschnitt eine mögliche Ausprägung des Entitätstyps *Kletterhalle* (*IndoorClimbing*) unter Einhaltung der Integritätsbedingungen.

4.7.2.1 Entitäts- und Annotations-Knoten

Für die Instanziierung stehen zwei generische Knotentypen bereit, die als Entitäts-Knoten (engl., Sg.: *entity node*) bzw. Annotations-Knoten (engl., Sg.: *annotation node*) bezeichnet werden. Ein Entitäts-Knoten bildet dabei das Pendant zu einem Entitätstyp-Knoten und ein Annotations-Knoten das Gegenstück zu einem Annotationstyp-Knoten. Dabei nutzen die Instanzknoten die entsprechenden Typknoten als Strategie, was in der Abbildung durch einfache Kanten ohne Pfeile dargestellt ist.

Entitäts-Knoten werden als Rechtecke gezeichnet und mit einer Kombination aus Ausprägungs- und Typnamen beschriftet, die unterstrichen dargestellt wird. Der Ausprägungsname ist frei wählbar, muss eindeutig sein und wird mit einem Doppelpunkt vom Typnamen getrennt.

Annotations-Knoten können in zwei Weisen dargestellt werden. Die triviale Notation sieht die Verwendung von Ellipsen für alle Arten von Annotationstyp-Knoten sowie die Verwendung genau einer Kanten- bzw. Pfeilart vor. Dies entspricht der tatsächlichen Repräsentation von Instanzen eher als die erweiterte Notation, die Knoten- und Kantenarten teilweise aus der Schemasyntax übernimmt und damit dem Informationsgehalt eines Typdiagramms weniger nachsteht. Die Unterschiede werden in den Folgeabschnitten deutlich, in denen beide Formen verwendet werden. In beiden Fällen folgt die Beschriftung der Knoten dem Beispiel der Entitäts-Knoten wobei außerdem die Wertebelegung in Hochkommata als weitere Knotenmarkierung hinzukommt.

Beim Aufbau einer Entität wird zunächst ein Entitäts-Knoten ausgebildet und mit einer entsprechenden Typstrategie versehen. Anschließend werden gültige Annotations-Knoten abgeleitet und als Kindknoten verankert. Im Gegensatz zur schematischen Repräsentation werden für den Instanzbereich Mitgliedschaftskanten nicht explizit ausgebildet, sondern über bidirektionale Referenzierung modelliert. Dies wird in der Abbildung durch Verbindungslinien mit kurzen spitzen Pfeilen kenntlich gemacht, die als generische Referenzkanten bezeichnet werden und der vereinfachten Darstellungsform entsprechen.

4.7.2.2 Multi-Knoten

Ein weiterer Unterschied betrifft den Umgang mit multiplen Wertebelegungen für Annotationstypen. Das Konzept sieht vor, dass es innerhalb einer Menge von Kindknoten keine zwei Knoten desselben Typs geben darf, wobei Kindknoten jene Knoten darstellen, die als direkte Nachfolger erreichbar sind und somit auf einer Ebene liegen. Diese Anforderung kann im Schemabereich einfach durch die Angabe von Kardinalitäten erfüllt werden. Im Instanzbereich muss diese Metainformation allerdings konkret umgesetzt werden. Dies hat eine mehrfache Ausprägung von Instanzknoten des gleichen Typs zur Folge, die aufgrund der zuvor gestellten Anforderung nicht direkt als Kindknoten eingehängt werden können.

Die Lösung für dieses Problem besteht in der Einführung sogenannter Multi-Knoten (engl., Sg.: *multi node*), die als Behälter für Mehrfachausprägungen dienen und im vorliegenden Diagramm als μ-Knoten mit entsprechender Typreferenz integriert werden. Der Multi-Knoten wird als Repräsentant der Mehrfachausprägung als direkter Nachfolger in die Entitäts-Struktur eingehängt und referenziert seinerseits alle entsprechend ausgeprägten Merkmale. Er stellt damit das Gegenstück zur

Kardinalitätsangabe im Schema dar. Seine Beschriftung unterscheidet sich darin, dass anstelle einer konkreten Wertebelegung ein fett gedrucktes μ gesetzt wird.

Einen sehr wichtigen Faktor stellt die Tatsache dar, dass ausschließlich Entitätstyp- und Annotationstyp-Knoten instanziiert werden. Entitätstypklassen-, Superentitätstyp- und Entitätstyp-Schnittstellen- sowie Cluster-Knoten werden nicht ausgeprägt. Sie sind jedoch über die Typreferenzierung erreichbar, was für den Aufbau von konsistenten Datenbankzuständen unablässig ist. In der Beispielgrafik wird der Bezug des Entitäts-Knotens `ic` zur Klasse seines Typs über eine gestrichelte Klassifikationskante angezeigt. Diese Darstellungsart gehört zum Repertoire der erweiterten Notation.

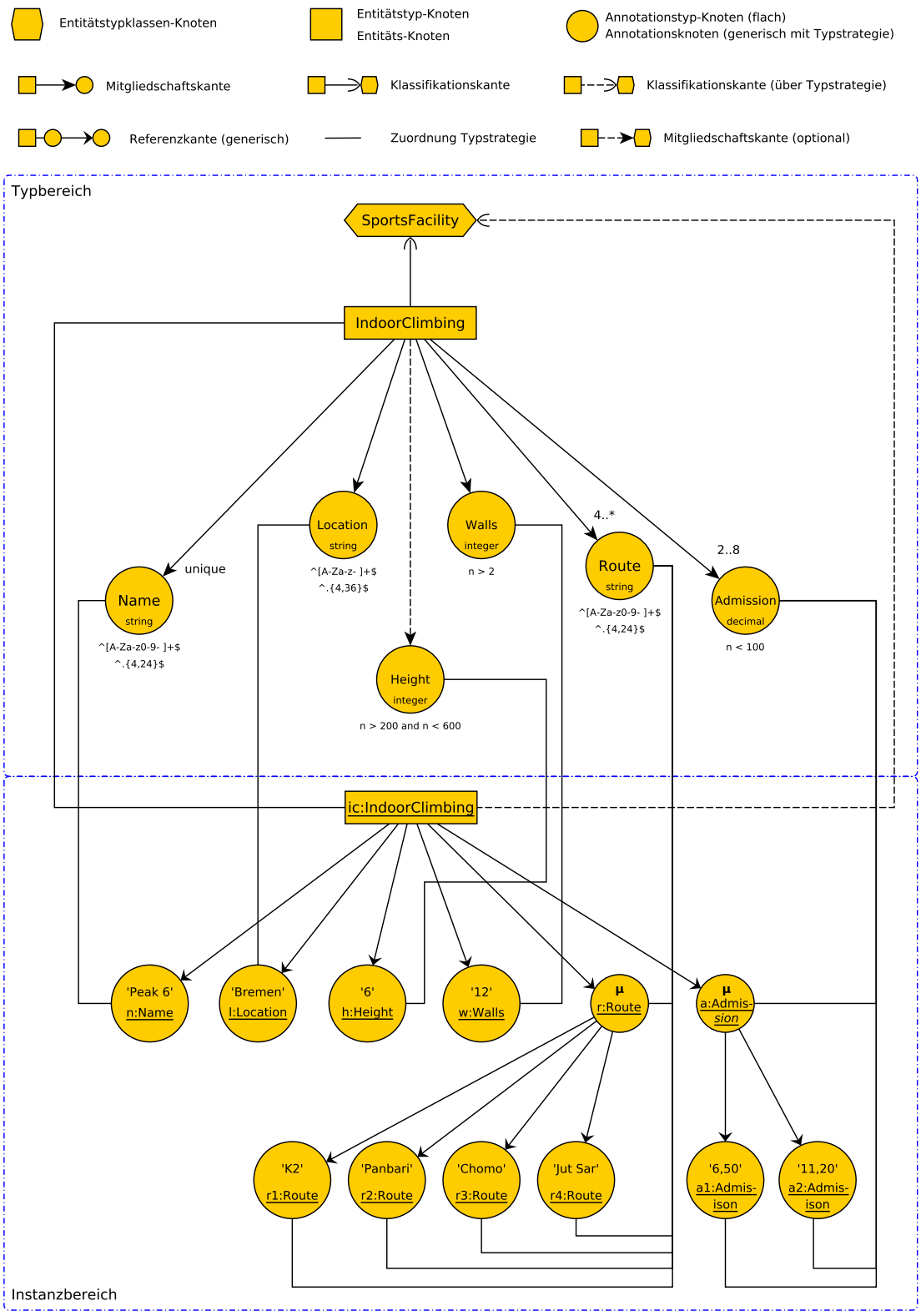

Abbildung 4.27: Übersicht Zustandsableitung

4.7.3 Ausführliches Beispiel

Grafik 4.28 zeigt eine angepasste Version des Schemas 4.13, für das nach einer kurzen Erläuterung ein Zustand aufgebaut wird.

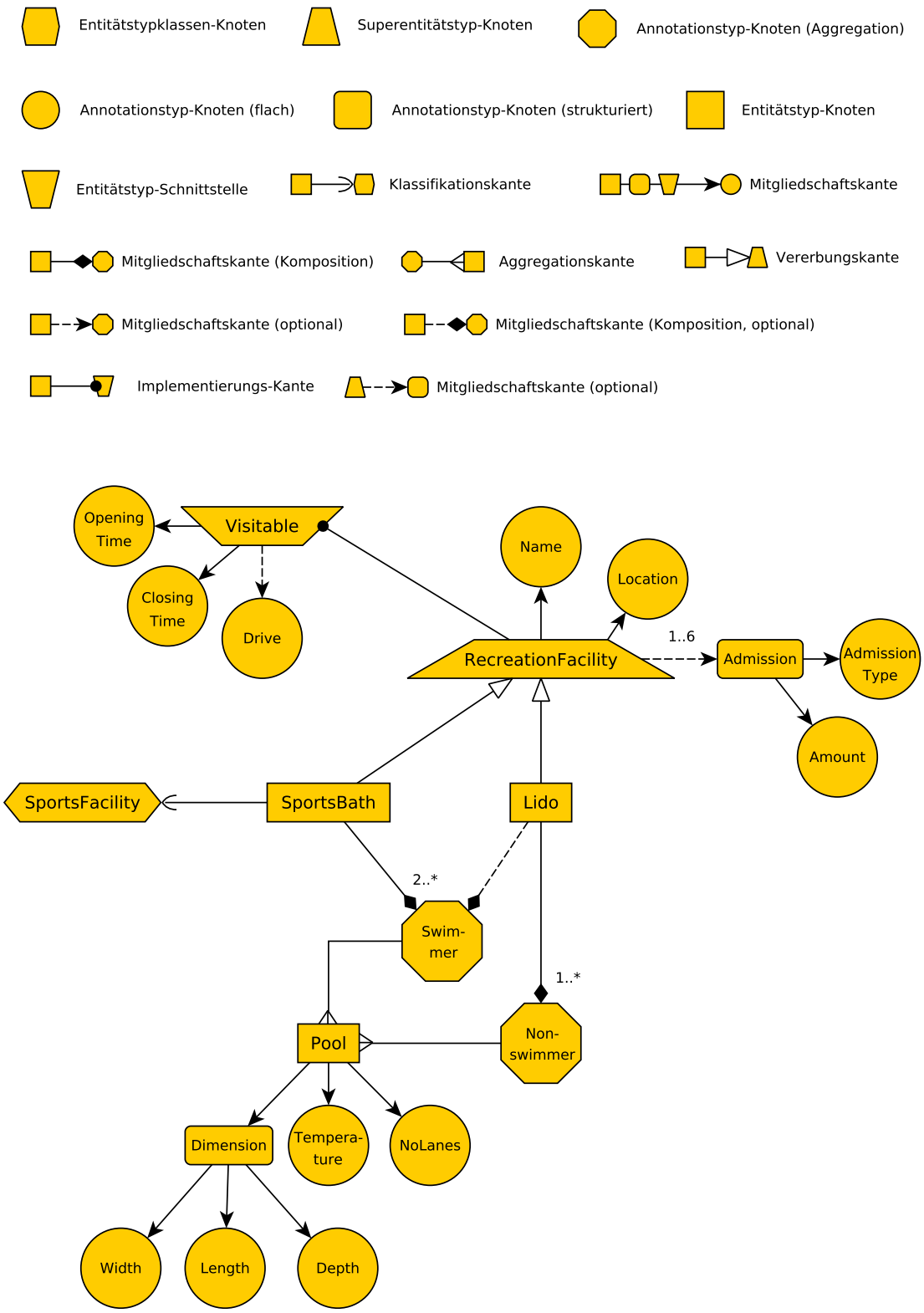

Abbildung 4.28: Typgraph ausführliches Beispiel

Das Schema beinhaltet die Entitätstypen *Lido* für Freibad, *SportsBath* für Sportbad und *Pool* für Schwimmbecken. Freibad und Sportbad erben vom Supertyp *RecreationFacility*, der Attribute für ein generelles Konzept einer Freizeit- bzw. Erholungsstätte vererbt. Dieser Supertyp wiederum implementiert die Schnittstelle *Visitable* und bietet so Merkmale an, die wichtige Informationen für Besucher liefern.

Der Entitätstyp *SportsBath* wird außerdem als Sportstätte (*SportsFacility*) klassifiziert. Sowohl Sportbad und Freibad stellen Aggregatoren für den Typ Schwimmbecken dar, dessen Ausprägungen über entsprechende Aggregations-Knoten komponiert werden. Die verwendeten Annotationstypen sind aus den vorherigen Beispielen bekannt. Auf Angabe von Datentypen und weiteren Wertebeschränkungen wurde der Einfachheit halber verzichtet.

Ein gültiger Zustand für das angezeigte Schema lässt sich der Abbildung 4.29 entnehmen. Die Darstellung zeigt die erweiterte Notation mit Elementen aus dem Typgraphen und beinhaltet Kanten zu Klassen- bzw. Supertypen, die gestrichelt dargestellt werden, um zu verdeutlichen, dass es sich um Verweise zu Elementen aus dem Typbereich handelt. Aggregations-Knoten tragen anstelle einer Wertebelegung ein fett gedrucktes α. Annotations-Knoten, die für einen strukturierten Typ stehen, werden mit einem ebenfalls fett gedruckten σ markiert.

Für den vorliegenden Zustand wurden insgesamt 64 Knoten ausgeprägt, die mit 62 Kanten verbunden sind. Unter Ausblendung der Kanten zu den Typelementen beinhaltet dieser Zustand zwei unzusammenhängende Graphen, genauer zwei Wurzelbäume. In den folgenden Abschnitten wird darauf eingegangen, wie daraus ein vollständiger Instanzgraph entsteht, der alle Daten speichert.

An dieser Stelle soll zunächst auf die Instanzen des Typs *Pool* eingegangen werden. Diese werden im Rahmen von Kompositionsbeziehungen als Teile existentiell an entsprechende Schwimmbad-Instanzen gebunden. Das bedeutet, dass eine Schwimmbecken-Instanz entweder zu *einer* Frei- oder Sportbad-Instanz gehört und nicht von zwei Ganzheiten gleichzeitig als Komponente verwendet werden kann. Diese Einschränkung gibt es bei einer Aggregationsbeziehung, die keine Komposition fordert, nicht. Existenziell ungebundene Entitäten können beliebig oft von Aggregations-Knoten referenziert werden, um beispielsweise an Assoziationsbeziehungen teilzunehmen.

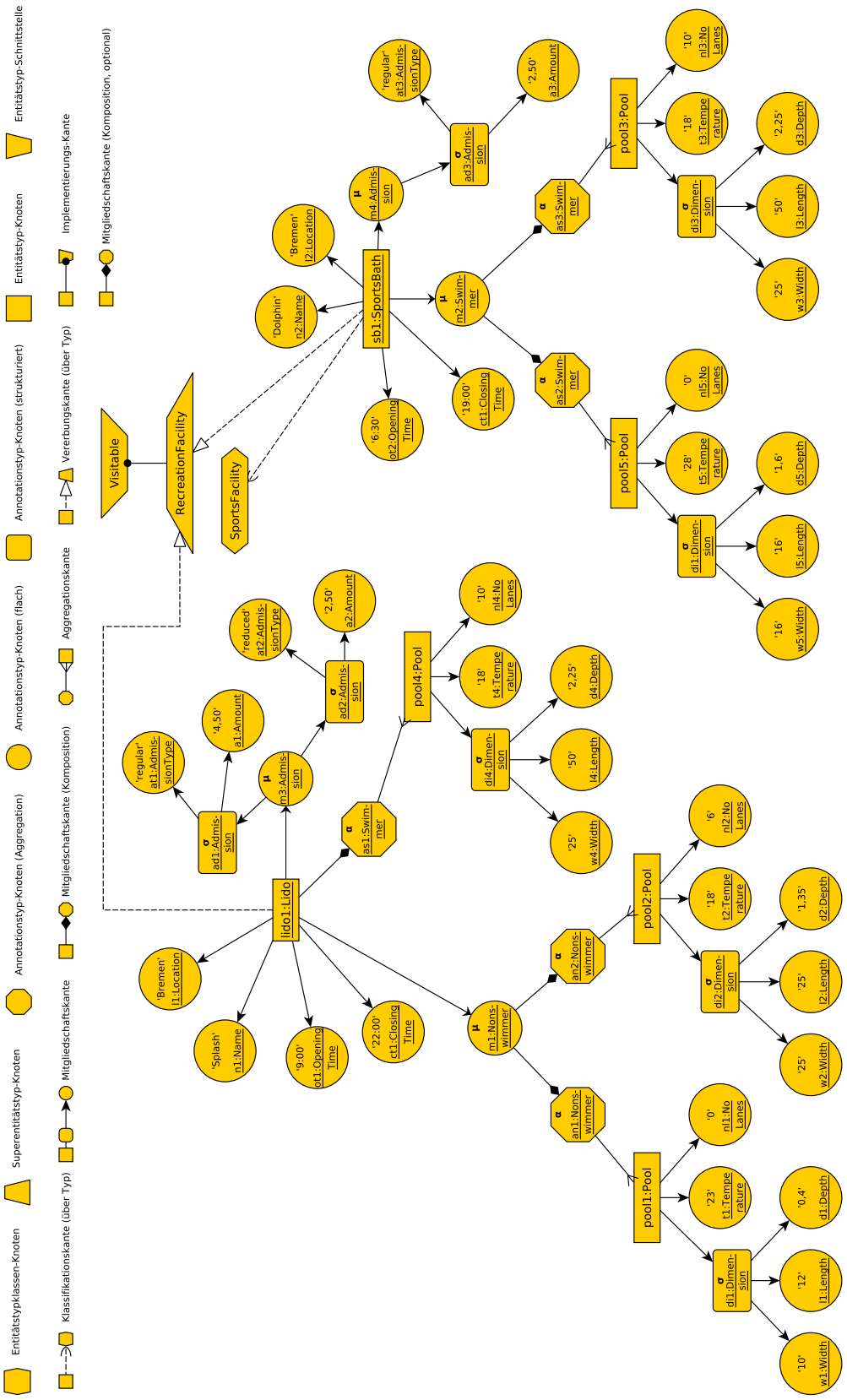

Abbildung 4.29: Zustand ausführliches Beispiel

Es sei angemerkt, dass die Mehrfachreferenzierung im Kontext einer Komposition theoretisch ebenfalls möglich ist, allerdings muss hier gefragt werden, ob es semantisch sinnvoll sein kann, dass ein Teil zu mehr als einer Ganzheit eine existenzielle Verbindung eingeht. Hier muss dann auch überlegt werden, welche Lösch-Anomalien auftreten können, und ob man diese zum Beispiel durch eine kaskadierte Löschung vermeidet.

Durch semantische Benennung der Aggregations-Knoten können die Schwimmbecken-Instanzen über Rollennamen angesprochen werden. Der Einsatz von Multi-Knoten erlaubt außerdem eine Mehrfachaggregation von Entitäten.

4.7.4 Wiederverwendung von Instanzknoten

4.7.4.1 Unifikation 1. Stufe

Der im vorangegangenen Beispiel erzeugte Zustand ist redundant, da eine bestimmte Anzahl von Attribut-Knoten desselben Typs bei gleicher Wertebelegung existiert. Diese Redundanz ist kein Zufall, da Entitäten grundsätzlich auf Basis eines Typs abgeleitet werden und damit dieselbe Menge von Merkmalen besitzen, die aufgrund diskreter Wertebereiche mit einer gewissen Wahrscheinlichkeit identisch ausgeprägt werden [Mit11, S. 13ff.]

Dieses Phänomen bildet die Grundlage für ein Konzept, das an dieser Stelle als Unifikation (engl., Sg.: *unification*) bezeichnet wird. Unter der Annahme, dass die Ausprägung von Knoten grundsätzlich teurer ist als das Setzen von Referenzen, wird bei jeder Zustandsänderung ermittelt, ob bereits ein entsprechender Annotations-Knoten mit gleicher Wertebelegung existiert. Ist dies der Fall, so wird auf diesen referenziert und auf die Neuausprägung eines Knotens verzichtet. Die Unifikation führt damit eine neue Repräsentation der Instanzdaten ein, die sowohl den Speicher- als auch den Zeitaufwand beim Zugriff auf Entitäts-Strukturen verringern soll.

Dieses Prinzip soll anhand eines Beispiels verdeutlicht werden, in dem der Zustand aus dem letzten Abschnitt analysiert wird. Dabei werden zunächst die Ausprägungen des Typs *Pool* betrachtet, die in einer Menge $POOL_I$ zusammengefasst werden:

$$POOL_I = \{pool1,\ pool2,\ pool3,\ pool4,\ pool5\} \tag{4.23}$$

Zwischen den Schwimmbecken-Instanzen $POOL_I$ und den zugehörigen Merkmalsausprägungen existieren Beziehungen, die mittels Referenzkanten hergestellt

werden. Für die formale Beschreibung werden die Merkmalsausprägungen ebenfalls in Mengen zusammengeführt:

$$DIMENSION_I = \{di1,\ di2,\ di3,\ di4,\ di5\} \tag{4.24}$$

$$TEMPERATURE_I = \{t1,\ t2,\ t3,\ t4,\ t5\} \tag{4.25}$$

$$NOLANES_I = \{nl1,\ nl2,\ nl3,\ nl4,\ nl5\} \tag{4.26}$$

Mittels einer Referenzrelation $R_{reference}$ erfolgt nun eine entsprechende Abbildung. Die Referenzrelation ist für dieses Beispiel als Teilmenge des kartesischen Produktes zwischen der Menge der Instanzen des Typs *Pool* und der Vereinigungsmenge $DIMENSION_I \cup TEMPERATURE_I \cup NOLANES_I$ definiert:

$$R_{reference} = \{(x, y) \mid x \in POOL_I \wedge y \in A_I\} \tag{4.27}$$

$$A_I = DIMENSION_I \cup TEMPERATURE_I \cup NOLANES_I \tag{4.28}$$

$$R_{reference_1} = \{(pool1,\ di1),\ (pool1,\ t1),\ (pool1,\ nl1),\ ..,\ (pool5,\ nl5)\} \tag{4.29}$$

Bei dem strukturierten Attribut *Dimension* kann entsprechend verfahren werden, wobei als Urmenge $DIMENSION_I$ und als Bildmenge die Vereinigung von $WIDTH_I$, $LENGTH_I$ und $DEPTH_I$ herangezogen werden.

Um von dieser Darstellung nun in die unifizierte Darstellung zu wechseln, bedarf es zweier Schritte. Zunächst wird für die Mengen der Merkmalsausprägungen eine Bedingung formuliert, wobei A_I stellvertretend für die konkreten Annotations-Mengen steht:

$$\forall x, y \in A_I : type(x) \neq type(y) \wedge value(x) \neq value(y) \tag{4.30}$$

Es darf in einer Attributmenge also keine zwei Objekte geben, die den gleichen Typ und den gleichen Wert besitzen. Die Identität wird also nicht mehr über den Instanznamen definiert, sondern über Typ und Wert. Für die zuvor aufgeführten Mengen bedeutet diese Bedingung eine Reduzierung der Elemente:

$$DIMENSION_I = \{di1,\ di2,\ di3,\ di5\} \tag{4.31}$$
$$TEMPERATURE_I = \{t1,\ t2,\ t4\} \tag{4.32}$$
$$NOLANES_I = \{nl1,\ nl2,\ nl3\}. \tag{4.33}$$

Auch die Instanzmenge des strukturierten Annotationstyps *Dimension* wird reduziert, da der Vergleich auf Basis einer Kombination subordinierter Werte durchgeführt werden kann. Unabhängig davon können diese subordinierten Knotenmengen ebenfalls unifiziert werden:

$$WIDTH_I = \{w1,\ w2,\ w5\} \tag{4.34}$$
$$LENGTH_I = \{l1,\ l2,\ l3,\ l5\} \tag{4.35}$$
$$DEPTH_I = \{d1,\ d2,\ d3,\ d5\} \tag{4.36}$$

Im zweiten Schritt müssen die Referenzen auf jene Knoten, die aus den Mengen entfernt werden, auf die verbleibenden identischen Knoten gerichtet werden. An dieser Stelle wird auf weitere Mengenaufzählungen bzw. -beschreibungen verzichtet und auf Grafik 4.30 verwiesen, die einen partiell unifizierten Zustand abbildet.

Dieses Vorgehen wird für alle ausgeprägten Attribute des aktuellen Beispielzustands wiederholt und formal mit der Einführung einer Äquivalenzrelation beschrieben, die alle als gleichwertig angesehenen Elemente der Attributmenge A_I in disjunkte Äquivalenzklassen übernimmt. Diese Äquivalenzrelation $R_{unification}$ kann auf Basis der zuvor formulierten Bedingung definiert werden als:

$$x \sim_{R_{unification}} y :\Leftrightarrow x, y \in A_I : type(x) = type(y) \wedge value(x) = value(y) \tag{4.37}$$

Es seien an dieser Stelle exemplarisch einige Äquivalenzklassen für die Annotationstypen *Temperature*, *NoLanes* und *Width* gegeben:

$$t2/_{\sim_{R_{unification}}} = \{t3,\ t4\} \tag{4.38}$$

$$nl1/_{\sim_{R_{unification}}} = \{nl5\} \tag{4.39}$$

$$w2/_{\sim_{R_{unification}}} = \{w3,\ w4\} \tag{4.40}$$

Die Äquivalenzrelation $R_{unification}$ wird auf dem Universum der Attribut-Instanzen definiert. Diese Menge ist heterogen, da ihre Elemente alle vorgestellten Annotationstypen als Strategie verwenden können. Dieser Sachverhalt führt zu Problemen, die an dieser Stelle erläutert werden.

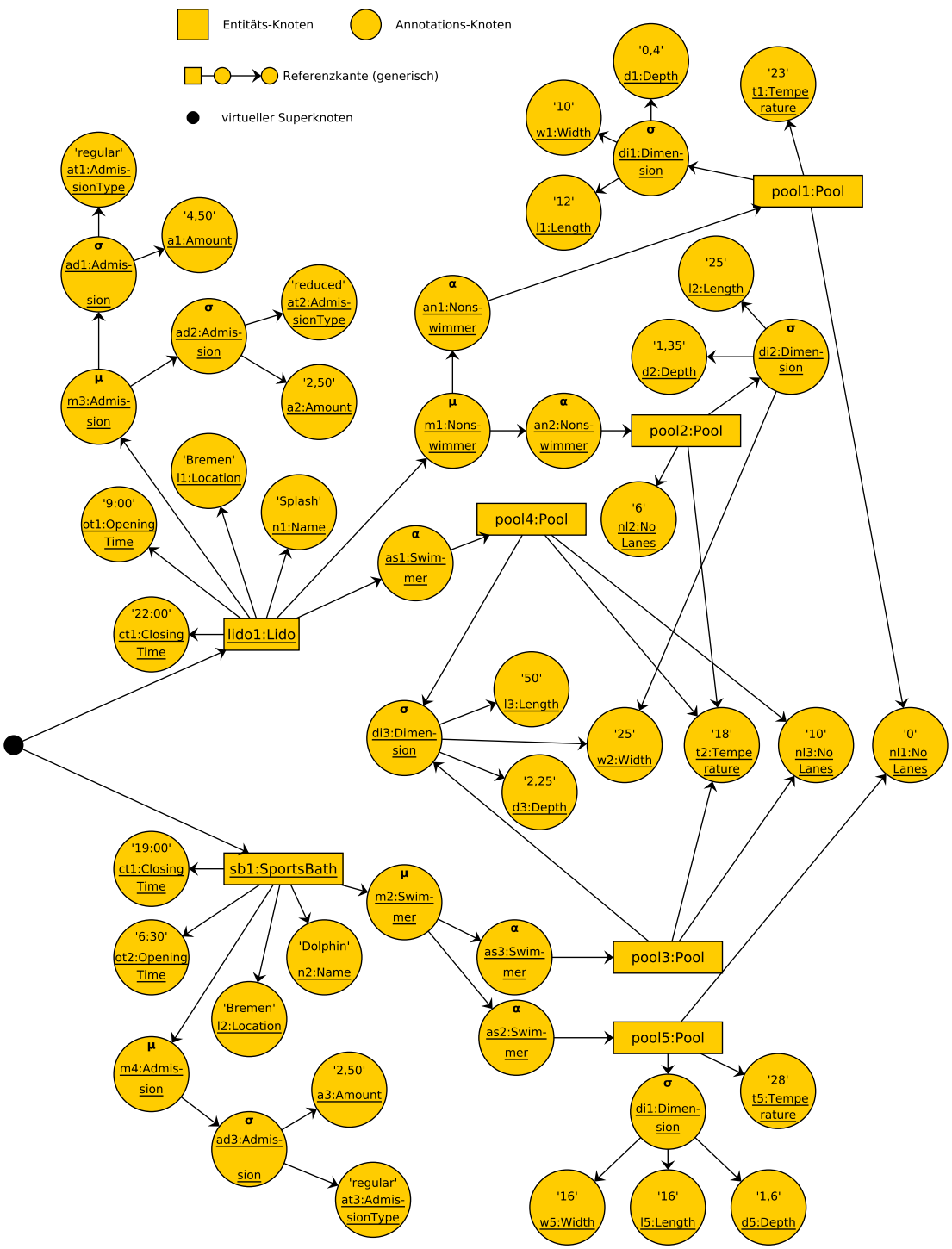

Abbildung 4.30: Partiell unifizierter Zustand (1. Stufe)

Die Formulierung der Bedingung für $R_{unification}$ verwendet zwei Abbildungen, die bisher nicht definiert wurden:

$$type : A_I \rightarrow A \tag{4.41}$$

$$value : A_I \rightarrow S^* \tag{4.42}$$

Die Abbildung *type* liefert für einen Instanzknoten den verwendeten Typ und *value* die entsprechende Wertebelegung. S^* stellt dabei vereinfacht eine Menge dar, die alle möglichen Werte beinhaltet. Während der Bildbereich von *type* durch das verwendete Schema vollständig gegeben ist, bleibt dieser für *value* bisher undefiniert, wenn es sich um Annotationstypen handelt, die keine flachen Merkmale repräsentieren.

Das Problem wird gelöst, indem für diese Knoten Pseudo-Wertebelegungen generiert werden, deren Ausprägungen durch vorhandene Kindknoten, falls nötig, rekursiv festgelegt werden. Folgende Annotationstypen benötigen eine entsprechende Generierung:

- strukturierte Typen
- Aggregations-Typen
- Beschränkungsgruppen-Typen
- Alias-Typen

Außerdem wird zur Vereinfachung eine Abbildung *hash* eingeführt, die alle Instanzknoten auf einen eindeutigen numerischen Wert abbildet:

$$hash : A_I \rightarrow H \tag{4.43}$$

Die Bildmenge H setzt sich aus allen potentiellen Hash-Werten zusammen, die für die zu betrachtenden Annotationen berechnet werden.

Die Abbildung *hash* kann anstelle der zuvor eingeführten Abbildungen *type* und *value* verwendet werden, sodass sich die Definition der Äquivalenzrelation bezüglich der Unifikation wie folgt ändert:

$$x \sim_{R_{unification}} y :\Leftrightarrow x, y \in A_I : hash(x) = hash(y) \tag{4.44}$$

Die Ermittlung der Hash-Werte erfolgt auf Basis der Bedingung, wie sie für die Gleichwertigkeit von Instanzknoten eingeführt wurde und berücksichtigt sowohl den Typ eines Instanzknotens als auch dessen tatsächlichen oder ersatzweisen Wert. Pseudo-Werte werden dabei abhängig vom Annotationstyp unterschiedlich generiert.

Strukturierter Typ:
Der Ersatzwert für Instanzknoten mit einem strukturiertem Typ wird mittels Konkatenation aller Werte subordinierter Instanzknoten ermittelt. Diese Kindknoten können dabei flach sein oder wiederum strukturiert. Im ersten Fall wird dann der tatsächliche Wert berücksichtigt, im zweiten Fall die rekursiv berechneten Pseudo-Werte.

Aggregations-Typ:
Als Ersatzwert für Instanzknoten vom Typ Aggregation wird die eindeutige Identifikationsnummer der referenzierten Entität verwendet.

Alias-Typ:
Der Pseudo-Wert für Alias-Knoten ist genau der Wert des untergeordneten umbenannten Knotens.

Beschränkungsgruppen-Typ:
Hier wird der Pseudo-Wert durch Konkatenation aller Kindknoten-Werte erzeugt.

In Abbildung 4.31 ist der vollständig unifizierte Zustand abgebildet.

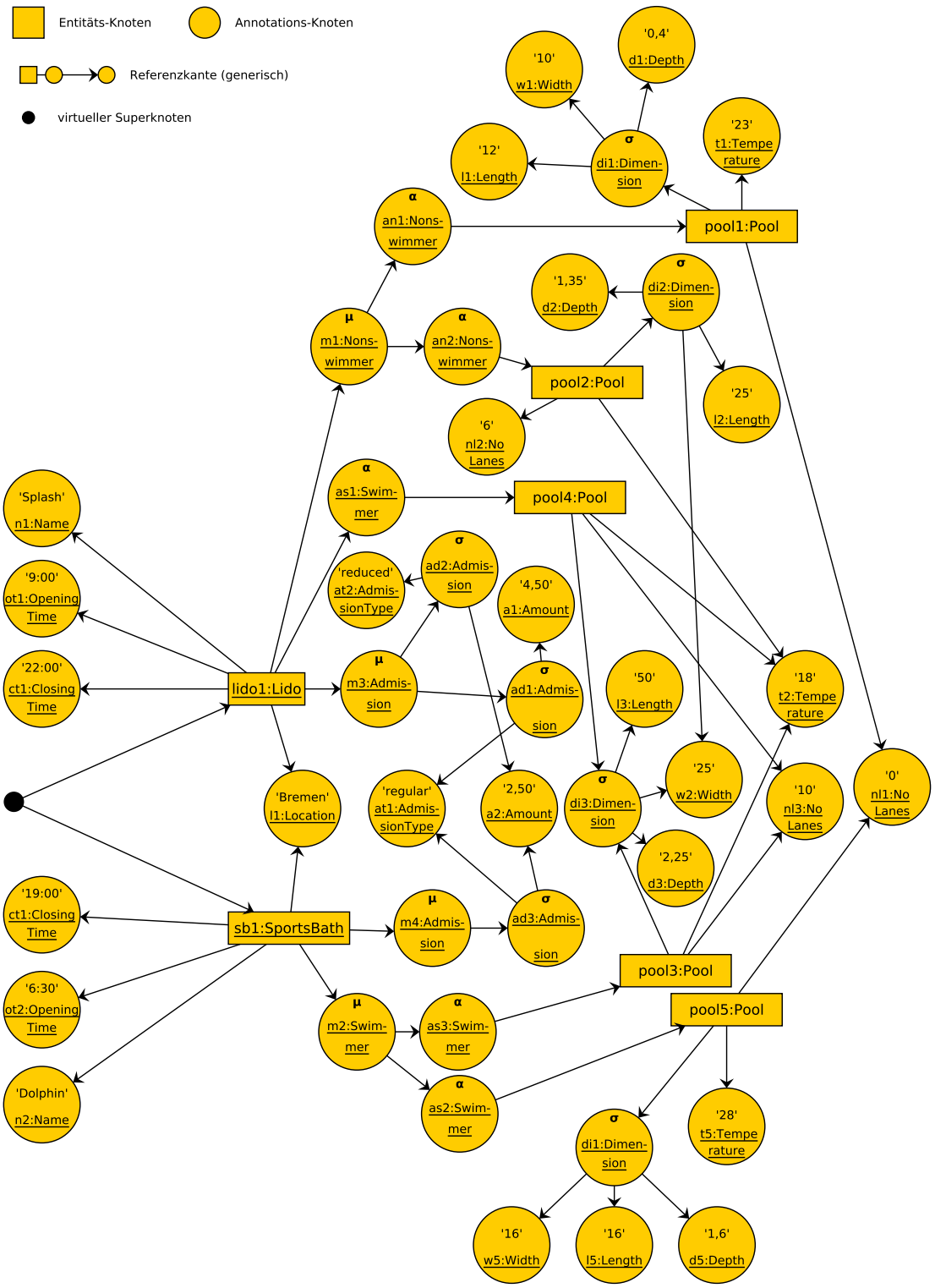

Abbildung 4.31: Vollständig unifizierter Zustand (1. Stufe)

4.7.4.2 Unifikation 2. Stufe

Im letzten Beispiel sind lediglich flache und strukturierte Annotationen von der Unifikation betroffen. Das nächste Beispiel zeigt, dass auch Aggregations-Knoten mehrfach genutzt werden können, wenn sie nicht ausschließlich im Rahmen von Kompositionen eingesetzt werden, sondern auch für einfache Aggregationen, wie sie u. a. für den Aufbau von Beziehungen benötigt werden.

In Grafik 4.32 ist ein Schema zu sehen, das diesen Aspekt aufgreift, indem es den Beziehungstyp *Renting* einführt.

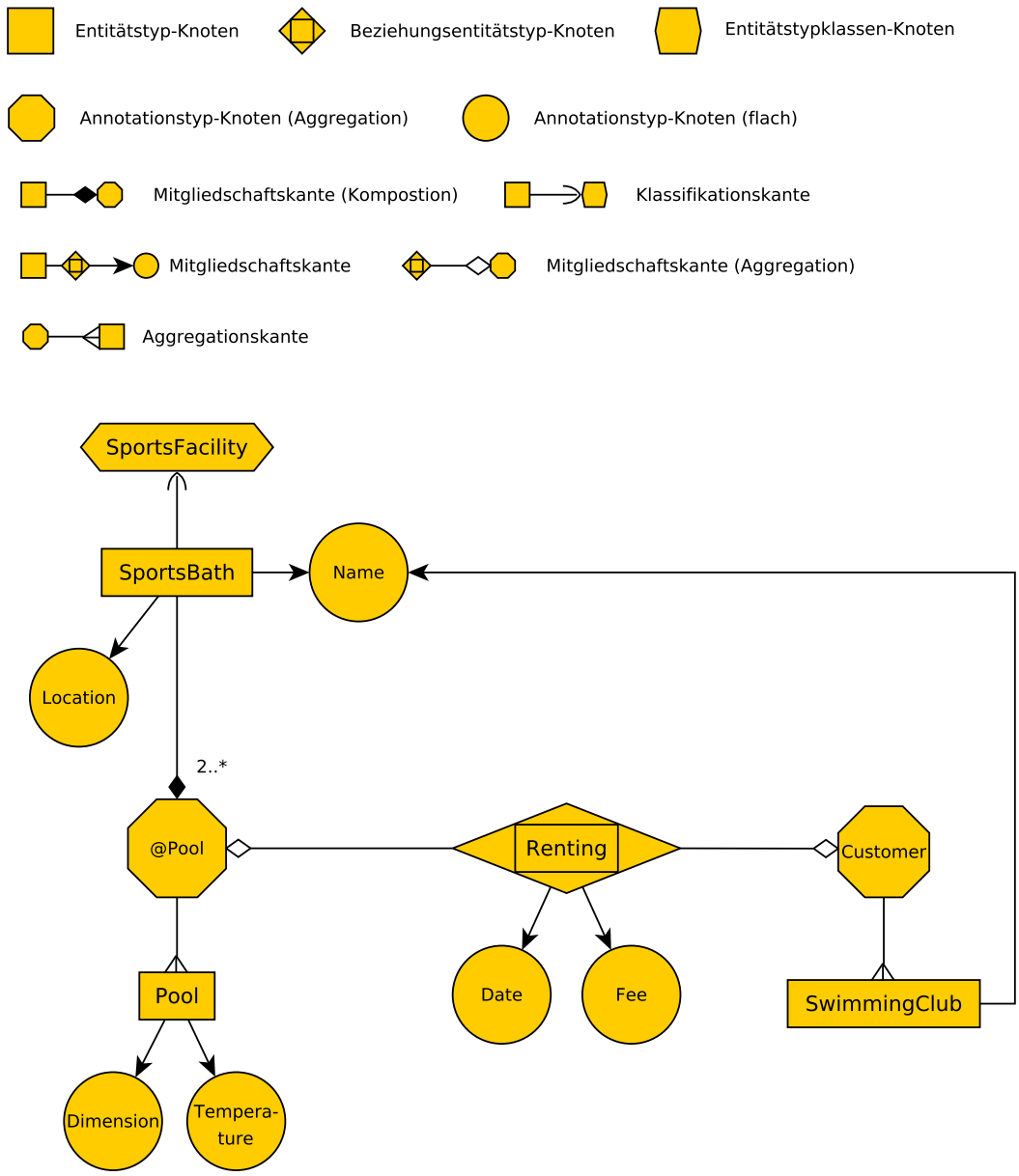

Abbildung 4.32: Typgraph Unifikationsbeispiel (2. Stufe)

Der Typgraph modelliert ein Szenario, in der Schwimmvereine (*SwimmingClub*) einzelne Schwimmbecken (*Pool*) für ihre Aktivitäten mieten können, die zu entsprechenden Sportbädern (*SportsBath*) gehören. Der Beziehungstyp *Renting* stellt hierfür eine einfache Assoziation zwischen Schwimmbecken- und Schwimmverein-Typen her und ist selbst mit den Merkmalen Datum (*Date*) und Gebühr (*Fee*) versehen, die den

Mietvorgang näher beschreiben. Schwimmbecken sind über die Rolle *Pool* erreichbar und Schwimmvereine über die Rolle *Customer*.

Ein möglicher Datenbankzustand ist in Grafik 4.33 gegeben, die einen abgeleiteten Instanzgraphen abbildet. Sie verwendet die erweiterte Notation.

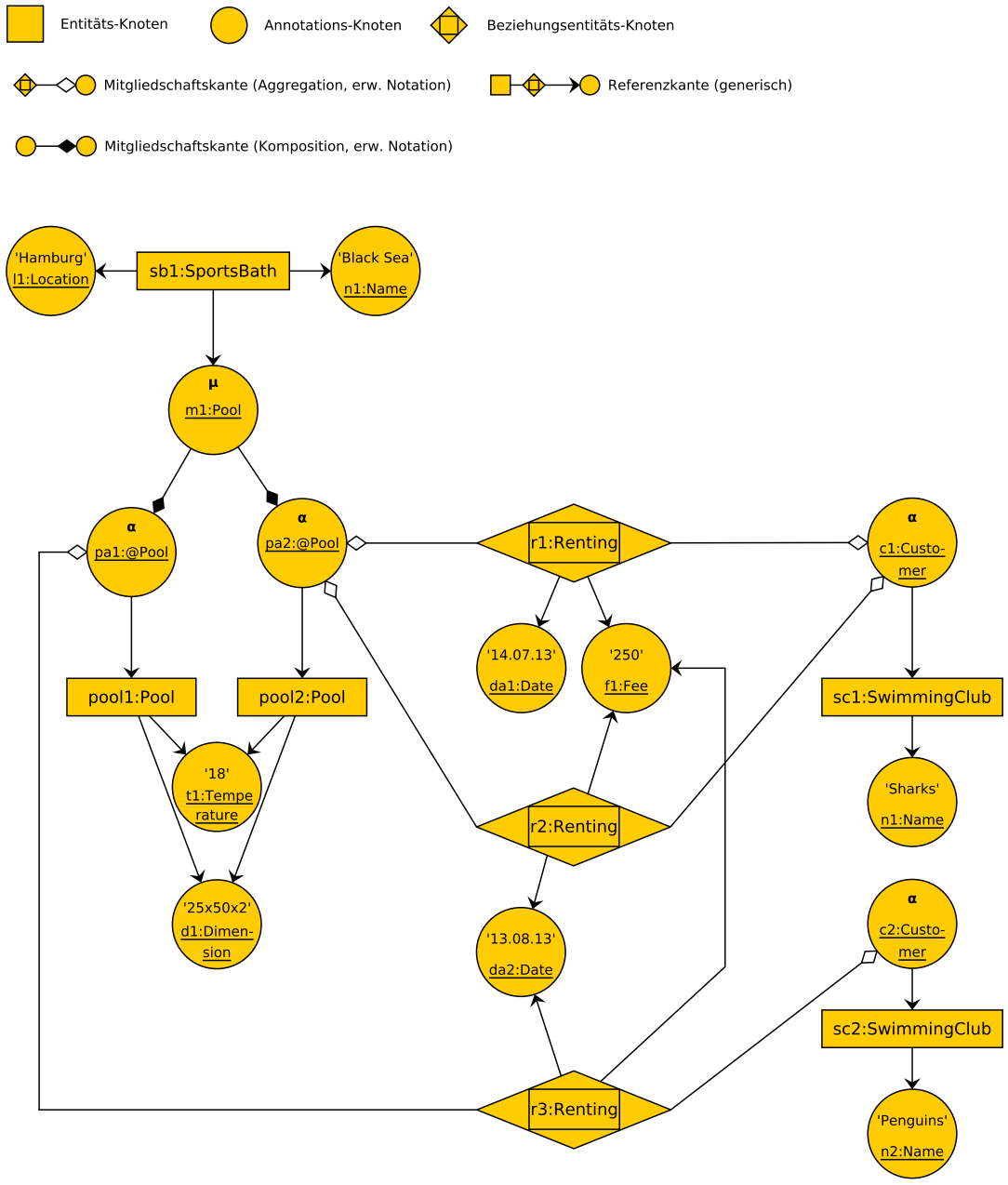

Abbildung 4.33: Zustand Unifikationsbeispiel (2. Stufe)

Dem Zustand ist zu entnehmen, dass der Schwimmverein der „Sharks“ das Schwimmbecken `pool2` zweimal zu unterschiedlichen Zeitpunkten bei gleicher Gebühr mietet. Die Aggregations-Knoten pa2 sowie `c1` können an dieser Stelle unifiziert verwendet werden.

4.7.5 Aufbau von Datenbankzuständen

Die vorherigen Abschnitte zeigen einige Beispiele für Datenbankzustände. An dieser Stelle soll genau beschrieben werden, welche Schritte notwendig sind, um Daten ablegen und damit einen neuen Zustand erzeugen zu können.

Die Datenbank wird durch den bereits erwähnten Instanzgraphen repräsentiert, der sich aus allen erzeugten Entitäten zusammensetzt. Wenn also von Instanzgraphen die Rede ist, dann ist damit der vollständige aktuelle Zustand der Datenbank gemeint. Wenn hingegen von Entitätsgraphen gesprochen wird, dann sind damit die einzelnen Ausprägungen von Entitätstypen gemeint, die als Teilgraphen in den Gesamtzustand eingehen.

Der Instanzgraph ist hauptspeicherresident und folgt damit dem Konzept einer *In-Memory-Datenbank* (engl., Sg.: *in-memory database*). Dies bedeutet insbesondere, dass Daten in einem flüchtigem Speicher abgelegt werden und nicht auf einem Festspeicher, wie es der gängigen Erwartungshaltung entspricht. In diesem Zusammenhang wird die Semantik des Verbs *speichern* verändert, dessen Verwendung eigentlich meint, dass Daten *persistent* gemacht werden, also über Arbeitssitzungen bzw. Systemneustarts hinweg erhalten bleiben, was der Hauptspeicher eines Computersystems nicht gewährleisten kann.

Um Klarheit zu schaffen, werden an dieser Stelle neue Definitionen von Begriffen und Zusammenhängen gegeben, wie sie für dieses Konzept nützlich erscheinen.

Grafik 4.34 zeigt auf der linken Seite das Schema einer Datenhaltungsschicht mit *Zugriffskomponente*, *Schema* und *Persistenzschicht*. Die rechte Seite nimmt als konkrete Umsetzung Bezug zu diesem konzeptionellen Aufbau und zeigt den typischen Aufbau eines Datenbanksystems (DBS) bestehend aus Datenbankmanagementsystem (DBMS) und der Datenbank (DB) selbst. Das Datenbankmanagementsystem beinhaltet die Komponenten für Zugriff, Schema und weitere Verwaltungsaufgaben wie zum Beispiel Rechte- und Transaktionsmanagement, die an dieser Stelle nicht weiter beschrieben werden sollen. Die Datenbank, dargestellt durch einen Zylinder, repräsentiert die Persistenzschicht, die klassischerweise unter Zuhilfenahme eines Festspeichers konstruiert wird.

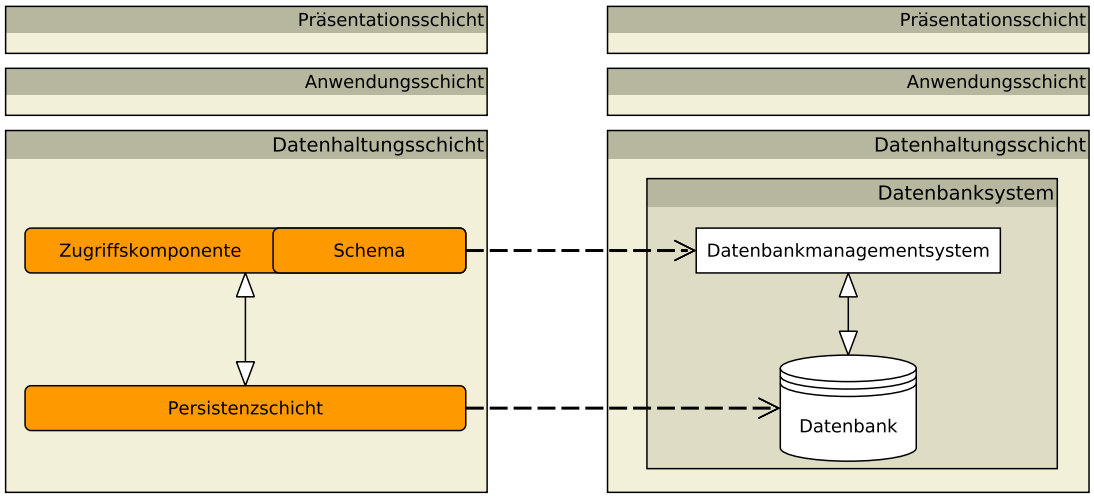

Abbildung 4.34: Klassische Realisierung einer Datenhaltungsschicht

Diese Organisationsstruktur wird für das vorliegende Konzept erweitert und in Abbildung 4.35 im rechten Abschnitt dargestellt. Zum Vergleich dient der linke Bereich der Grafik, der sich auf die vorangegangene Beschreibung bezieht. Erhalten bleibt eine allgemeine Zugriffskomponente, die in der Regel als Teil eines Managementsubsystems implementiert wird und vier grundsätzliche Operationen bereitstellt, die in der Datenbankdomäne unter dem Akronym *CRUD* bekannt ist. CRUD steht für Create, Read, Update und Delete also Erstellen, Lesen, Aktualisieren und Löschen von Datensätzen. Das Schema der Datenbank wird durch den Typgraph repräsentiert, der das logische Modell für eine korrekte Speicherung unter Berücksichtigung aller Integritätsbedingungen darstellt.

Wesentliche Änderungen gibt es bei der Definition der Persistenzschicht. Als persistent gilt hier, was sich innerhalb des Instanzgraphen befindet. Das Verb *speichern* kann so in seiner bisherigen Bedeutung weiterverwendet werden. Die tatsächliche Ablage auf Festspeichern wird über die Einführung einer neuen sogenannten *Permanenzschicht* realisiert, die transparent für den Datenbanknutzer im Hintergrund kontinuierlich beständige Zustandsabbilder erzeugt. Dieser Vorgang wird als *Materialisierung* bezeichnet.

Um also einen neuen Zustand zu erzeugen, wird über die Zugriffskomponente eine neue Entität angelegt, indem entsprechende Merkmale ausgeprägt werden. Anschließend wird mittels Typgraph geprüft, ob alle Integritätsbedingungen erfüllt sind. Ist dies der Fall, so wird der neue Entitätsgraph in den Instanzgraph integriert und damit *gespeichert*.

Der nächste Materialisierungszyklus schreibt diese Daten dann transparent auf einen Festspeicher und macht diese damit *permanent*.

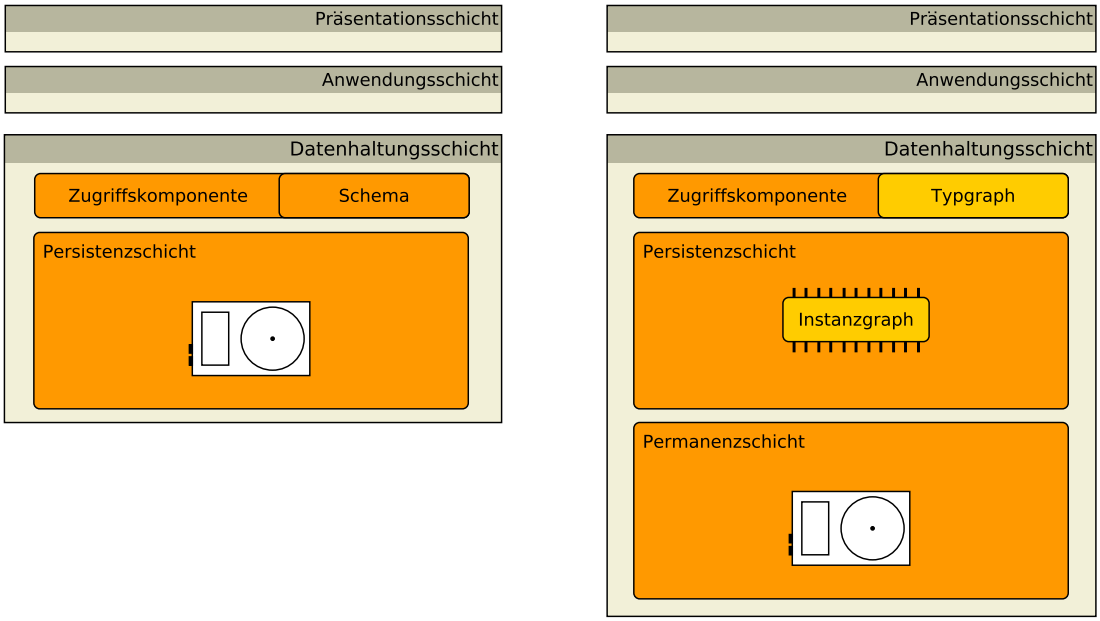

Abbildung 4.35: Adaption Datenhaltungsschicht

Der Mechanismus der Materialisierung kann zum einen als *Sicherungs-* und zum anderen als *Auslagerungsstrategie* verwendet werden. Als Sicherungsstrategie sorgt er dafür, dass Daten auch zwischen Systemstarts erhalten bleiben, wie bereits erläutert wurde. Hierfür wird der Instanzgraph zunächst neu aufgebaut, damit wieder auf ihm gearbeitet werden kann. Ein direkter Zugriff auf die Permanenzschicht durch den Datenbanknutzer ist nicht möglich.

Als Auslagerungsstrategie sorgt er dafür, dass Teile des Instanzgraphen auf Festspeicher verlagert werden, falls ein Überlauf des Hauptspeichers droht. So ist es dann auch möglich, Datenbestände aufzubauen, die die Größe des Arbeitsspeichers eines Computersystems übersteigen und eine entsprechende Speicherhierarchie aufzubauen.

Da der Arbeitsspeicher wesentlich schneller arbeitet als zum Beispiel SSD- oder Magnet- bzw. Bandlaufwerke, erfolgt die Materialisierung verzögert. Wenn sichergestellt sein muss, dass Daten nach der Speicherung permanent auf einem Festspeicher existieren, so kann das Anlegen eines Datensatzes im Durchschreibe-Modus (engl., Sg.: *write-through mode*) erfolgen, d. h. der Speichervorgang ist erst dann abgeschlossen, wenn die anzulegenden Informationen sowohl persistent als auch permanent gemacht

worden sind. Auf die Einzelheiten eines solchen Vorgehens wird nicht weiter eingegangen.

4.7.6 Outdoor- und Indoor-Speicherbereiche

Der Vorgang der Speicherung von Daten muss im Rahmen dieses Konzepts ein wenig näher betrachtet werden. Die Tatsache, dass Entitäten unabhängig von ihrem Persistenzstatus im Hauptspeicher residieren, verlangt eine virtuelle Differenzierung der entsprechenden Speicherinhalte.

Diese Differenzierung soll mittels eines sogenannten *Outdoor-Indoor-Prinzips* durchgeführt werden. Entitäten, die den Status *nicht gespeichert* besitzen, werden dabei einem virtuellen *Outdoor-Bereich* (engl., Sg.: *outdoor space*) zugeordnet. Entitäten, die den Status *gespeichert* besitzen, gehören dem *Indoor-Bereich* (engl., Sg.: *indoor space*) an. Der Zustandsübergang zwischen *nicht gespeichert* und *gespeichert* erfolgt mittels einer Aktion, die als *Speicherung* bezeichnet wird. Daten vor einer Speicherung sind nicht persistent, Daten nach einer Speicherung sind persistent und Daten, die vom Materialisierungsprozess erfasst wurden, sind zuletzt auch permanent.

Als Beispiel sei angenommen, dass eine Entität eines beliebigen Typs gespeichert werden soll. Nach Auswahl des Typs und Ausprägung der entsprechenden Merkmale befindet sich die Entität im Outdoor-Bereich. Der Aufruf einer entsprechenden Speicher-Operation überführt die Entität in den Indoor-Bereich, nachdem sichergestellt ist, dass alle Integritätsbedingungen erfüllt sind und der gesamte Datenbankzustand nach dem Speichervorgang in einen konsistenten Zustand fällt. Die Entität besitzt nun den Status gespeichert und wird entweder sofort (Durchschreibemodus) oder in einem der nächsten Zeitslots des Materialisierungs-Schedulers permanent auf einen Festspeicher geschrieben.

Grafik 4.36 zeigt die Speicherung zweier Entitäten mittels *Create*-Operationen. Das Verb *speichern* sei mit *store* ins Englische übersetzt. Die Repräsentations- sowie Anwendungsschicht wird zur Orientierung angedeutet und nicht weiter ausgeführt. Die Datenhaltungsschicht teilt sich in zwei Subschichten mit den Bezeichnungen Persistenz und Permanenz.

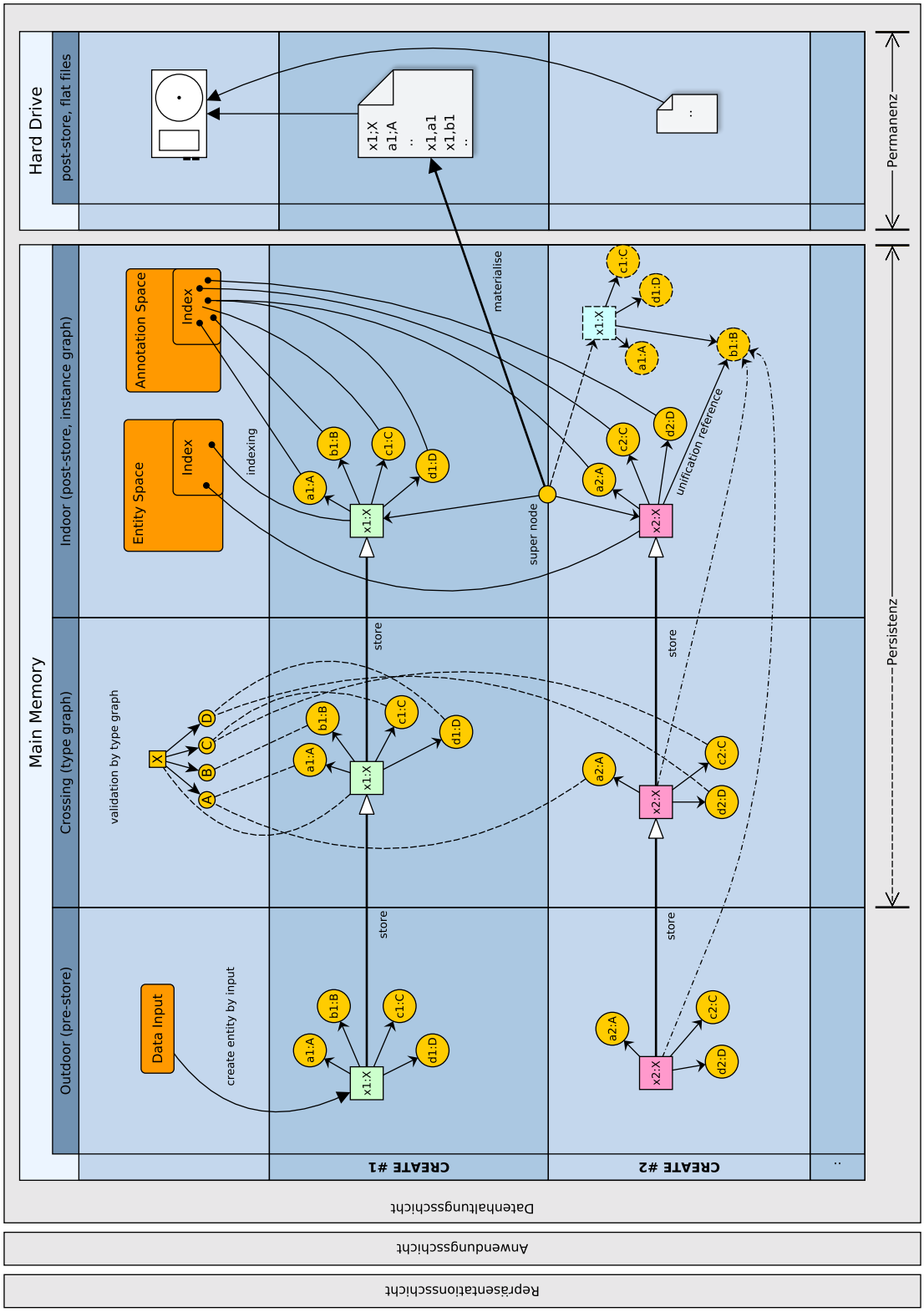

Abbildung 4.36: Übersicht Create-Operationen

Der erste *Create*-Vorgang erzeugt eine Entität und prägt entsprechende Merkmale aus, wie in der linken Spalte dargestellt wird. Wurde die Entität aufgebaut, trägt sie zunächst den Status *ungespeichert* und befindet sich im Outdoor-Bereich. In der zweiten Spalte erfolgt die Prüfung der Integritätsbedingungen mittels Typgraph.

Metaphorisch steht die Entität an der Schwelle zum Indoor-Bereich, die mit dieser Prüfung überschritten wird, was in diesem Fall als *Crossing* bezeichnet werden soll. Diese Überquerung geht in der dritten Spalte mit der Aufnahme aller ausgeprägten Knoten in entsprechende virtuelle Speicherräume (engl., Sg.: *space*) einher, die in dieser Darstellung mit Entitätsraum (engl., Sg.: *entity space*) und Annotationsraum (engl., Sg.: *annotation space*) benannt sind. Nachdem die Entität nun offiziell den Status *gespeichert* innehat, erfolgt die bereits erwähnte Materialisierung auf einen Festspeicher und der Übergang von der Persistenz zur Permanenz. Dabei müssen die Daten flachgeklopft werden, um in einfachen Dateien gespeichert werden zu können, was an dieser Stelle als *flatten* bezeichnet wird. Dieser Zusammenhang ist stark vereinfacht im rechten Bildbereich wiedergegeben.

Vor der Beschreibung des zweiten *Create*-Vorgangs soll darauf eingegangen werden, dass die Aufnahme von Instanzknoten in die einzelnen Speicherräume auch eine Indizierung beinhaltet, die mittels einer Hash-Tabelle realisiert wird. Es wurde bereits angemerkt, wie sich für einzelne Merkmalsknoten bzw. ganze Knotenstrukturen die benötigten Hash-Werte berechnen lassen. Auch ganze Entitäten lassen sich auf diese Weise indizieren, wobei hier sogenannte Entitätsgrenzen (engl., Sg.: *entity frontier*) beachtet werden müssen, um zu verhindern, dass mittels Aggregations-Knoten transitiv weite Teile des kompletten Instanzgraphen in die Ermittlung des Hash-Wertes einbezogen werden.

Beim zweiten *Create*-Vorgang spielt die Tatsache eine Rolle, dass bereits eine Merkmalsausprägung desselben Typs bei gleicher Belegung existiert. Vor dem Anlegen eines neuen Merkmals wird geprüft, ob bereits eine äquivalente Ausprägung im Annotationsraum vorhanden ist. Ist dies der Fall, so wird diese direkt referenziert, wie in der Abbildung dargestellt. Die Referenzierung ist dabei zunächst unidirektional, d. h. nur die Entität im Outdoor-Bereich hält eine Referenz auf den entsprechenden Knoten im Indoor-Bereich, aber nicht umgekehrt. Dies verhindert, dass nicht gespeicherte Entitäten Teil eines gültigen Datenbankzustandes werden und erfüllt so die Anforderung an eine strikte Trennung beider Speicherbereiche.

Der Übergang vom Outdoor- in den Indoor-Bereich bei aktiver Unifikation ist in Grafik 4.37 verdeutlicht. Vor der Speicherung (pre-store unifying) verweist `x2` lediglich unidirektional auf die bereits existierende Annotation `b`. Nach dem Speichervorgang

(post-store unifying) wird `x2` vollständig in den Zustand integriert und verweist nun auch bidirektional auf `b`.

Bei der Unifikation muss grundsätzlich zwischen zwei Fällen unterschieden werden. Die sogenannte *Sofort-Unifikation* (engl., Sg.: *instant unification*) versucht noch im Outdoor-Bereich, äquivalente Knoten zu ermitteln und diese anstelle eines potentiell neu zu erstellenden Knotens einzuhängen. Die Sofort-Unifikation ist dabei nicht für alle Annotationstypen möglich, da der auszuprägende Wert, der für die Äquivalenzermittlung notwendig ist, nicht immer während des Anlegens bekannt ist. So können beispielsweise sowohl flache Knoten als auch Aggregations-Knoten sofort unifiziert werden, da beim ersteren der Wert in der Regel im Zuge der Erstellung übergeben wird. Auch bei Aggregations-Knoten, deren Wert die eindeutige Kennnummer des referenzierten Knotens darstellt, ist diese Art der Unifikation möglich.

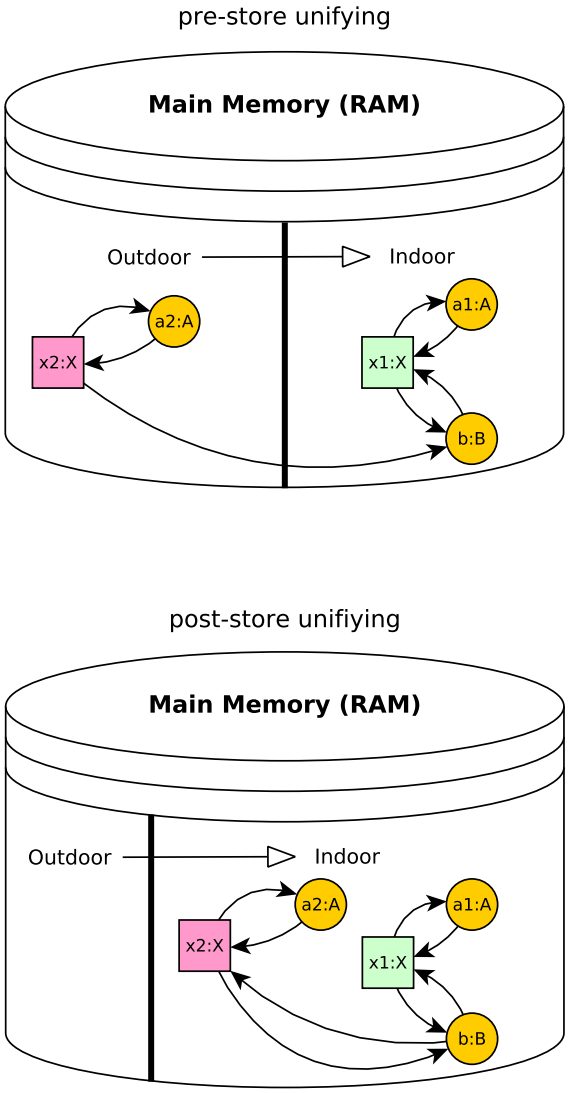

Abbildung 4.37: Übergang bei aktiver Sofort-Unifikation

Ein anderes Vorgehen ist die Unifikation nach dem Anlegen der Knotenstruktur im Outdoor-Bereich. Die sogenannte *verzögerte Unifikation* (engl., Sg.: *deferred unification*) findet während des Speicherns beim Übergang in den Indoor-Bereich statt und betrifft sowohl strukturierte Knoten als auch Multi-Knoten. Bei diesen Knoten ist die Unterstruktur, aus der sich der Wert ermittelt, beim sukzessiven Anlegen nicht sofort bekannt. Aus diesem Grund kann auch nicht unmittelbar nach bestehenden Instanzen gesucht werden. Das vorliegende Konzept unterstützt die verzögerte Unifikation auch für flache Knoten und Aggregations-Knoten.

4.8 Architekturansätze

Grafik 4.38 zeigt das Modell einer Schichtenarchitektur, wie es in diesem Kapitel bereits erläutert wurde. Es macht dabei keine Angaben darüber, in welcher Form die einzelnen Elemente technisch realisiert werden oder auf welche Weise sie Daten untereinander austauschen.

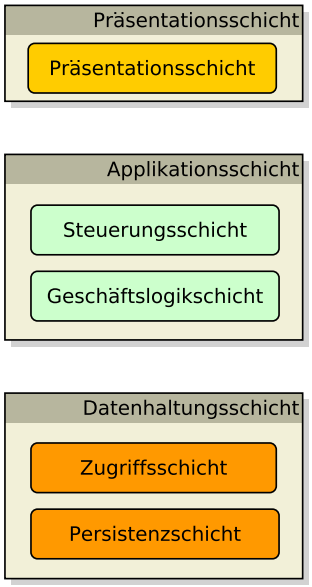

Abbildung 4.38: Klassische 3-Schichten-Architektur

Einen neuralgischen Punkt bei der Speicherung und Manipulation von Daten stellt die Frage dar, ob alle Komponenten Zugriff auf denselben Speicherbereich besitzen, oder ob sie zum Beispiel aufgrund eines *Client-Server-Ansatzes* in unterschiedlichen Prozessen residieren und Daten lediglich mittels *Interprozesskommunikation* austauschen können.

Für das vorliegende Konzept bedeutet der erste Fall, dass alle Schichten direkt mittels Speicherreferenz auf die Daten Bezug nehmen können. Die Anwendungslogik ist so zum Beispiel in der Lage, die gespeicherten Informationen ohne spezielle Abfrage- bzw. Manipulationssprache direkt zu lesen und zu ändern, indem sie direkt auf dem Instanzgraphen operiert. Die Präsentationsschicht wiederum greift die Daten aus der Anwendungsschicht ab und stellt sie in einem menschenlesbaren Format gemäß der zugrundeliegenden Benutzungsschnittstelle dar. Dieser Zusammenhang wird in 4.39 illustriert.

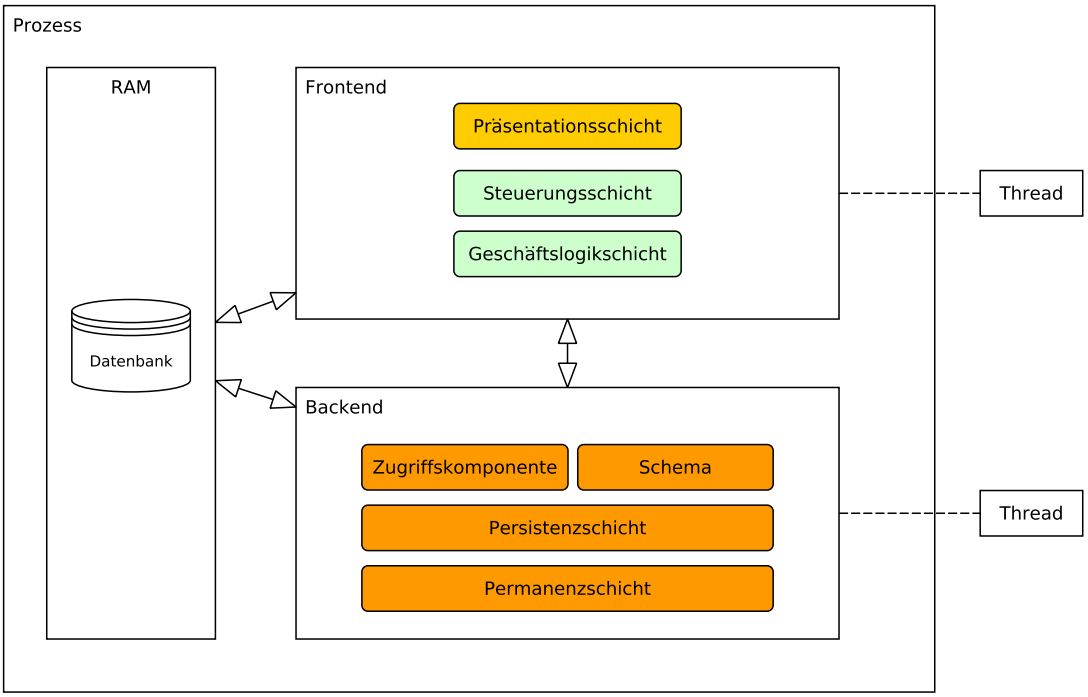

Abbildung 4.39: Multithreading-Ansatz (Stand-Alone)

Der dargestellte Aufbau segmentiert die Schichten in Front- und Backend, die in eigenen Threads laufen und dadurch gemeinsam auf den Datenbestand zugreifen können. Die Verwendung von *Multithreading* erlaubt den parallelen Zugriff auf den Datenbestand durch mehrere Benutzer. Je nach Operation muss hierbei auf *Synchronisation* geachtet werden, indem zum Beispiel einzelne Entitäten während einer Manipulation gesperrt und ein *exklusiver Zugriff* gewährleistet werden. Auf den Entwurf entsprechender Transaktionskonzepte soll hier nicht weiter eingegangen werden.

Im zweiten Fall müssen die Daten über Interprozesskommunikation zwischen den Schichten ausgetauscht werden. Die Realisierung der Datenhaltungsschicht als dedizierte Komponente, zum Beispiel in Form eines Datenbankservers, bedingt obligatorisch die Verwendung von Intermediärdarstellungen, um Daten über Prozessgrenzen hinweg transportieren zu können, da der direkte Zugriff mittels Speicherreferenzen nicht möglich ist.

Abbildung 4.40 zeigt einen Systemaufbau mit dedizierter Datenhaltung und Interprozesskommunikation.

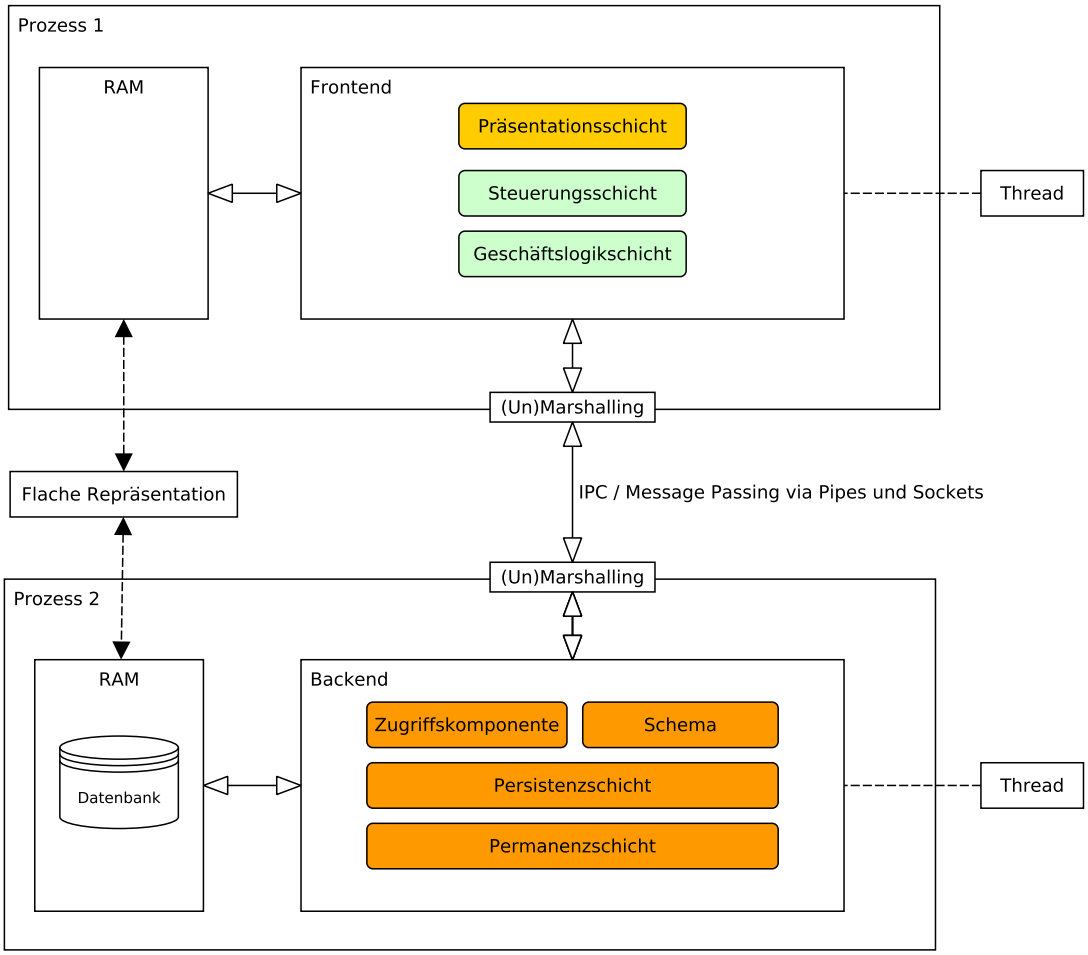

Abbildung 4.40: Ansatz Interprozesskommunikation (Client/Server)

Front- und Backend zerfallen bei diesem Aufbau in zwei Prozesse, die mittels Interprozesskommunikation Daten austauschen. Welche konkreten Protokolle für die Kommunikation genutzt werden, soll an dieser Stelle nicht weiter betrachtet werden. In der Abbildung wird beispielhaft das *Marshalling*-Konzept verwendet, das strukturierte Daten so darstellen kann, dass sie über Pipes bzw. Sockets ausgetauscht werden können. Denkbar wäre auch der Einsatz eines textbasierten XML-Protokolls, das zum Beispiel direkt auf dem IP-Protokoll arbeitet. Auch hier erfolgt die Umwandlung in eine flache Darstellung, aus der die ursprünglichen Daten nach dem Transport wiederhergestellt werden können. Auf dieser Basis können dann auch die bereits angesprochenen Datenbanksprachen arbeiten.

5 Implementierung

5.1 Entwicklung

5.1.1 Entwicklungsprozess

Auf der Basis des vorliegenden Konzepts wurde eine Referenzimplementierung angefertigt, die eine Teilmenge der formulierten Anforderungen erfüllt. Grundlegende Funktionalitäten wurden dabei im Rahmen eines agilen Entwicklungsprozesses sukzessive umgesetzt. Zwischen den einzelnen Implementierungszyklen wurden Tests durchgeführt und auftretende Probleme analysiert und korrigiert. Diese progressive Vorgehensweise erlaubte es, entlang eines roten Fadens zu einem funktionierenden System zu gelangen, mit dem bereits experimentiert werden kann.

Während dieser Iterationen wurde stets darauf geachtet, eine solide Gesamtstruktur aufzubauen, die spätere Erweiterungen zulässt. Darunter fällt auch die Verwendung von Entwurfsmustern und die Einführung einer grundlegenden Architektur.

5.1.2 Entwicklungsumgebung

Die Implementierung basiert auf der Programmiersprache *Ruby*. Ruby wurde 1995 von Japaner Yukihiro Matsumoto et al. entwickelt und gilt aufgrund der dynamischen Typisierung, der Fähigkeiten zur Metaprogrammierung und des flexiblen Einsatzes von Programmierparadigmen als äußerst mächtig. Die aktuelle Ruby-Implementierung in der Version 2.0.0 nutzt einen Bytecode-Interpreter, der *YARV* genannt wird. Dieser ist im Gegensatz zum vorher verwendeten MRI-Interpreter wesentlich schneller, da er nicht mehr direkt auf dem Syntaxbaum, sondern auf einer optimierten Zwischendarstellung arbeitet.

Ruby bringt in der Basisinstallation bereits eine Vielzahl nützlicher Bibliotheken mit. Für die Erweiterung dieser Palette kommt in der Regel ein eigenes Paketsystem zum Einsatz, das als *RubyGems* bzw. *Gems* bezeichnet wird.

Als integrierte Entwicklungsumgebung (IDE) wird *Netbeans* in der Version 6.9.1 eingesetzt, da sie frei verfügbar und als Java-Applikation auf allen gängigen Betriebssystemen ausführbar ist. Aus organisatorischen Gründen endet die Ruby-Unterstützung bei genau dieser Version, sodass zukünftig über einen Wechsel der Entwicklungsplattform nachgedacht werden muss.

Für die Entwicklung von UML-Diagrammen kommt das kostenpflichtige *Enterprise Architect* in der Version 7.1 von *Sparx Systems* zum Einsatz.

Das manuelle Zeichnen von Graphen und anderen Übersichtsabbildungen erfolgt mit dem *yEd Graph Editor* (Version 1.7.0_09), einem frei verfügbaren und überaus mächtigen Editor von *yWorks*. Die automatisierte Visualisierung von Typ- bzw. Instanzgraphen ermöglicht die Open-Source-Software *Graphviz* Version 2.31, die aus einem von *AT&T* und den *Bell-Labs* initiierten Projekt hervorgegangen ist.

Der Einsatz einer virtuellen Maschine erweist sich als sinnvoll, um die Entwicklungs- und Testumgebung gegen Störungen zu schützen und möglichst konstante Laufzeitbedingungen zu schaffen. Hierfür wird *VirtualBox* (Version 4.1.22) von *Oracle* verwendet und auf dem derzeitigen Host-System Ubuntu 10.04 LTS (64 Bit) installiert. Als Gast-System kommt Ubuntu 12.10 Server (64 Bit) zum Einsatz.

5.2 Systemstruktur

Abbildung 5.2 zeigt die Organisation der aktuellen Implementierung in Form eines Paketdiagramms. Das Hauptpaket trägt den Namen *Tibet*. Unter Berücksichtigung der Beschreibungen im Entwurfskapitel beinhaltet dieses Paket die Unterpakete *Management*, *Schema*, *Persistence* und *Permanence*, auf die an dieser Stelle eingegangen wird.

5.2.1 Management-Komponente

Das Paket *Management* stellt eine eigene Schicht dar, die aus dem Unterpaket *Processing* bzw. dem *Access*-Subsystem besteht. Das *Processing*-Paket beinhaltet Klassen, die von allen drei Hauptkomponenten *Schema*, *Persistence* und *Permanence* verwendet werden. Das *Access*-Subsystem beinhaltet die Fachlogik für das Outdoor-Indoor-Prinzip.

Die *Management*-Komponente wird in dem Diagramm horizontal über alle drei Hauptkomponenten gezeichnet, womit angedeutet werden soll, dass sie parallel Zugriff auf all diese Komponenten besitzt. Die Zugriffsorganisation ist vereinfacht in dem Komponentendiagramm 5.1 dargestellt.

5.2.2 Schema-Komponente

Die *Schema*-Komponente beinhaltet das Unterpaket *Type*, indem alle weiteren Pakete und Klassen für den Aufbau und die Verwendung des Typgraphen bereitgestellt werden. Auf Teile des *Graph*-Pakets wird im nächsten Abschnitt in Form von UML-Klassendiagrammen noch genauer eingegangen. Das *Processing*-Paket wiederum setzt sich aus einem *Traversal*- und *Activity*-Paket zusammen, in denen sich Fachlogik für Traversierung und Manipulation befinden.

5.2.3 Persistenz-Komponente

Die *Persistenz*-Komponente beinhaltet das Unterpaket *Instance*, in dem alle weiteren Pakete und Klassen für den Aufbau und die Verwendung des Instanzgraphen bereitgestellt werden. Auf Teile des *Instance*-Pakets wird noch genauer eingegangen. Das *Processing*-Paket beinhaltet Funktionalität für die Traversierung und Verarbeitung des Instanzgraphen.

5.2.4 Permanenz-Komponente

Die *Permanenz*-Komponente existiert bisher nur konzeptionell.

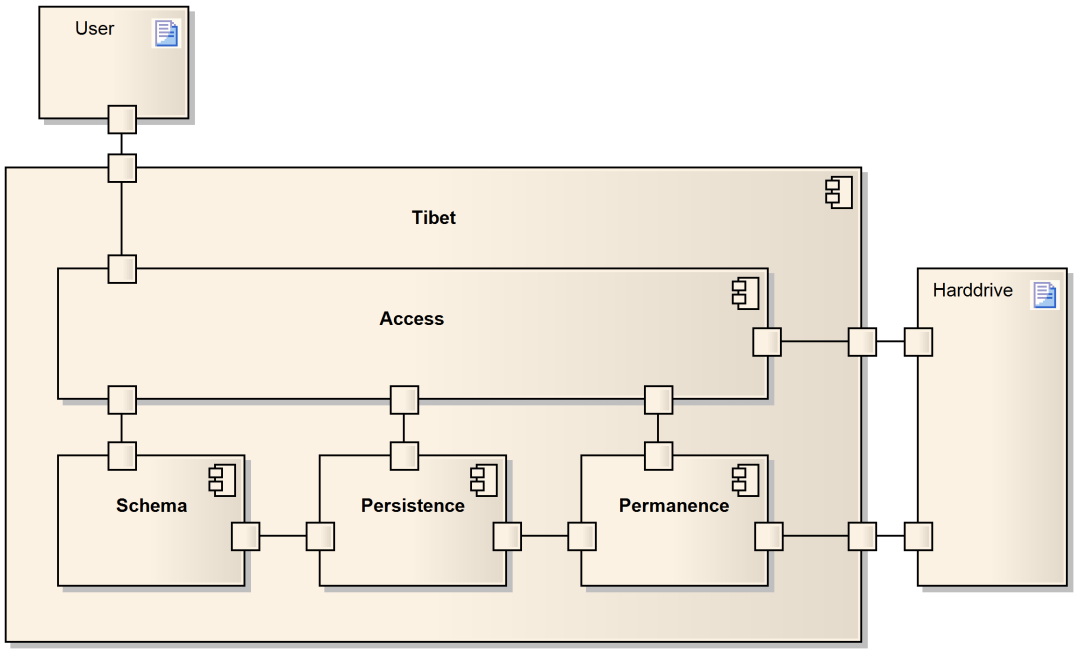

Abbildung 5.1: Tibet Komponentendiagramm

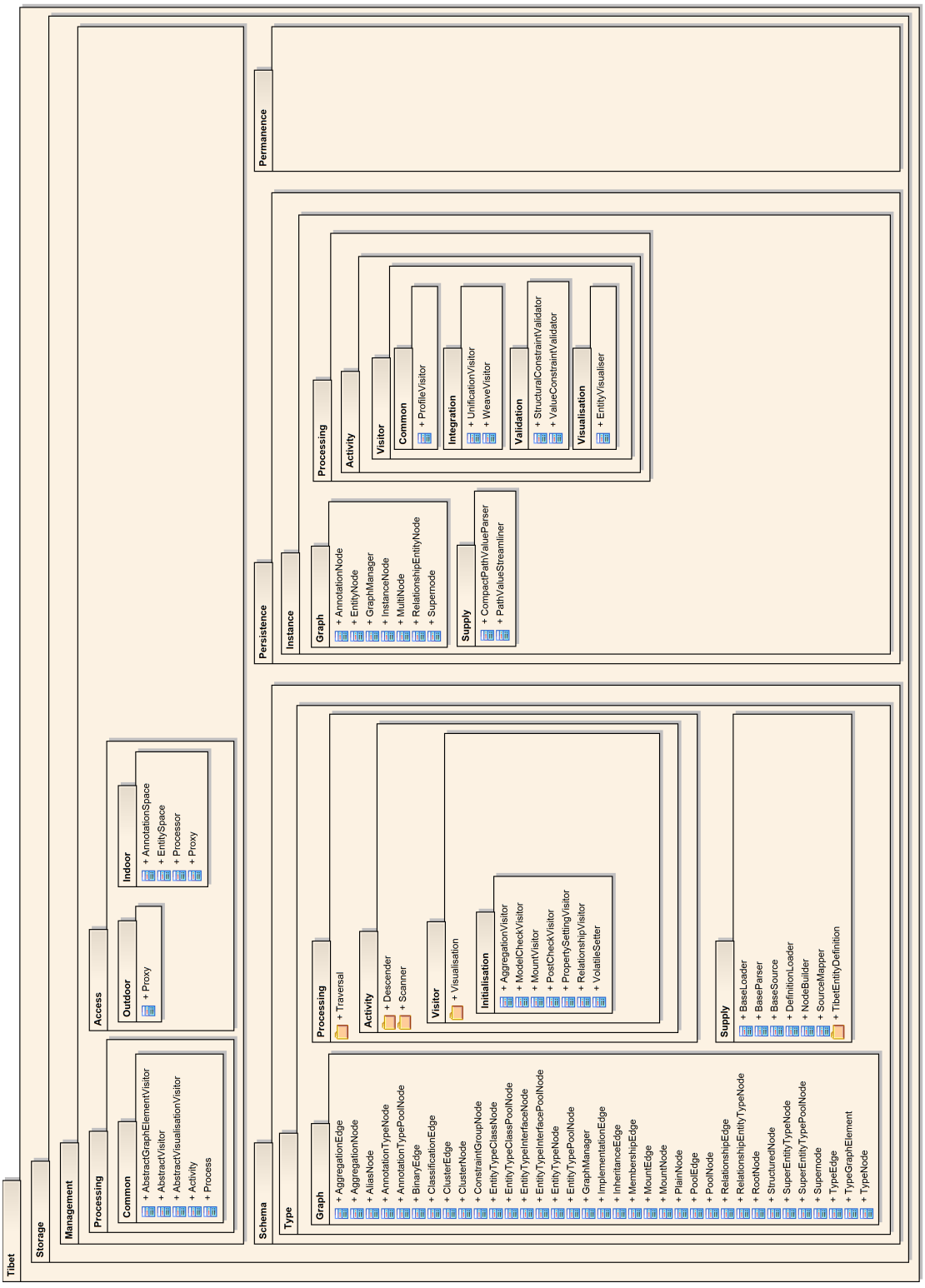

Abbildung 5.2: Tibet Paketdiagramm

5.3 Typgraph

Der Typgraph bildet die Basis für die Arbeit mit der Datenbank. Er wird, wie auch der Instanzgraph, im Arbeitsspeicher gehalten und beim Starten des Datenbankprozesses aufgebaut. Die benötigten Schemainformationen liegen im Textformat vor und werden in einer eigens entwickelten Definitionssprache namens *Tibet Definition Language* (TDL) verfasst. Diese Definitionssprache stellt dabei lediglich eine Interimslösung dar, bis Typdiagramme direkt eingelesen werden können. Für die graphische Definition von Schemata kann zum Beispiel der von yEd verwendete *GraphML*-Dialekt eingesetzt werden.

Die benötigten Elemente des Schemas werden durch entsprechende Ruby-Klassen repräsentiert, die beim Aufbau des Typgraphen zur Laufzeit instanziiert werden. Diese Typobjekte bilden nun den instanziierten Schemagraph und können ab diesem Zeitpunkt als *Typstrategien* [GHJV04, S. 373ff.] für Instanzknoten verwendet werden, um einen Datenbankzustand aufzubauen.

Für den Aufbau eines gültigen Typgraphen werden in der vorliegenden Implementierung entsprechende Ruby-Klassen verwendet. Eine teilweise Übersicht über diese Klassen und deren Vererbungsbeziehungen geben die folgenden UML-Klassendiagramme sowie die Diagramme A.2 und A.3.

5.3.1 UML-Klassendiagramm 1

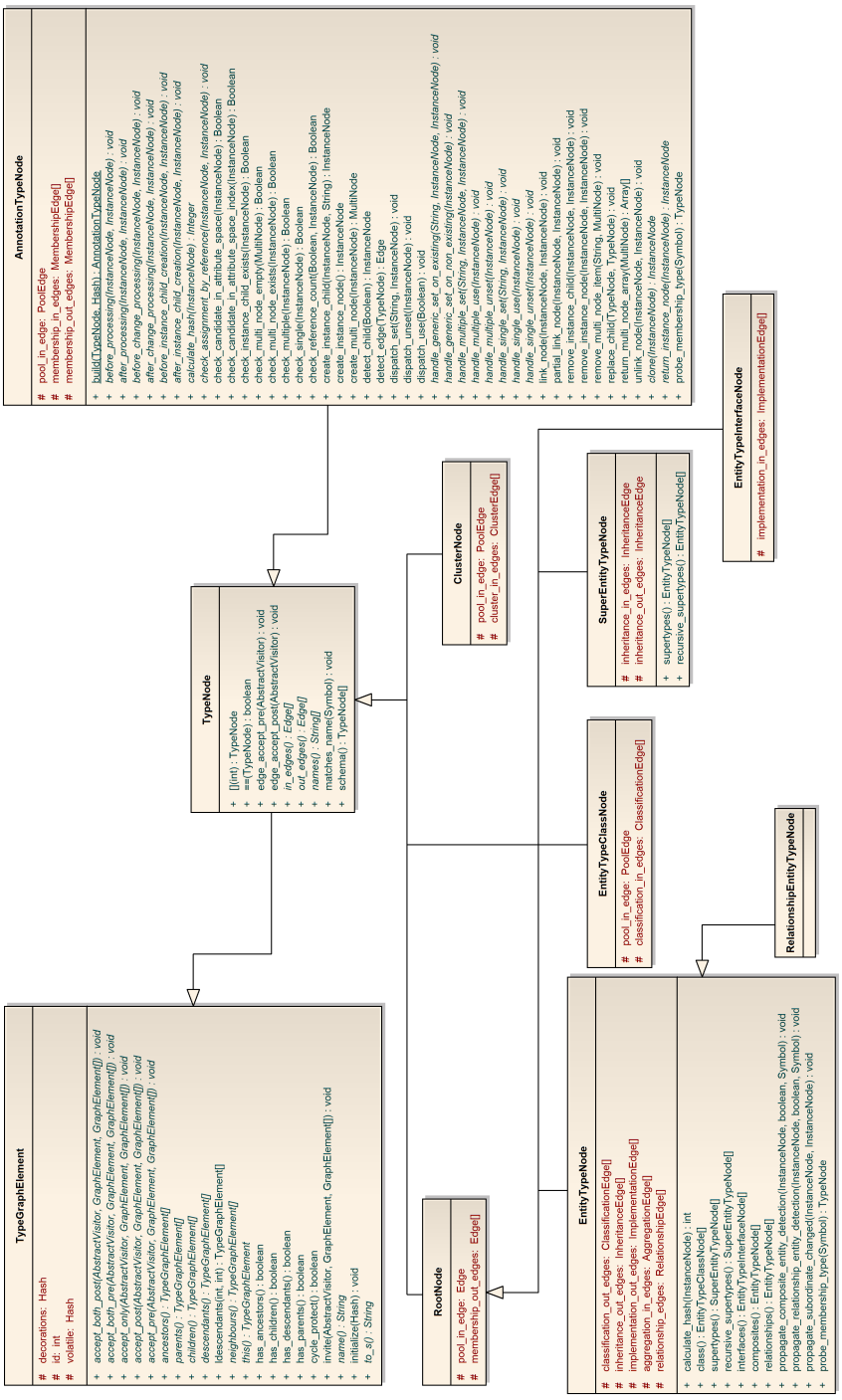

Abbildung 5.3: 1. UML-Klassendiagramm Typgraph

5.3.1.1 Die Klasse *TypeGraphElement*

In Abbildung 5.3 bildet die Klasse *TypeGraphElement* die abstrakteste Ebene der Vererbungshierarchie. Sie enthält alle für ein Typgraphelement grundlegend notwendigen Attribute und Methoden. So besitzt jedes Element eine eindeutige Identifikationsnummer, die an dieser Stelle mit `id` benannt wird. Außerdem können sowohl Typknoten als auch Typkanten mit Metainformationen versehen werden, die in einem Ruby Hash-Objekt gespeichert werden, das sich wie ein einfacher *Key/Value-Speicher* verhält. Dieser Speicher wird mit `decorations` benannt. Für Daten, die unter Umständen während der Verarbeitung des Typgraphen temporär gesichert werden müssen, kommt das Attribut `volatile` zum Einsatz.

Die Traversierung des Typgraphen erfolgt in der Regel auf Basis des Entwurfsmusters *Visitor* [GHJV04, S. 301ff.], auf dessen zweckdienliche Erweiterung später mit einem Code-Beispiel eingegangen werden soll. Die Standardmethoden für ein zu besuchendes Element der Graphstruktur sind im Kopf der Methodensignaturabschnitts der Klasse *TypeGraphElement* zu sehen. Die aufgeführten *accept*-Methoden unterscheiden sich einerseits in der Besuchsreihenfolge und andererseits in der Frage, ob lediglich Knoten oder Knoten und Kanten besucht werden.

Nach den Besuchsmethoden folgen die Navigationsmethoden, die ebenfalls abstrakt gehalten sind und je nach Knotentyp unterschiedlich definiert werden. Die Methoden `invite` und `cycle_protect` sind für die Erkennung von Zyklen verantwortlich und sind hier bereits konkret implementiert.

5.3.1.2 Exkurs Visitor-Pattern

An dieser Stelle soll nun ein kurzes Code-Beispiel für eine `accept_pre`-Methode gegeben werden, wie sie zum Beispiel für die Klasse *EntityTypeNode* implementiert wurde:

Listing 5.1: `accept_pre`-Methode der Klasse *EntityTypeNode*

```
1  # @override
2  def accept_pre( visitor, target = self, path = [] ) #;d
3    return unless super
4
5    visitor.visit_abstract( target, path )
6
7    return if visitor.prevent_propagation_entity_type( target, path )
8
9    visitor.before_entity_type( target, path )
10   visitor.before_node( target, path )
```

```
11
12    visitor.visit_entity_type( target, path )
13    visitor.visit_node( target, path )
14
15    target.children.each { | child |
16      child.accept_pre( visitor, child, path.clone )
17    }
18
19    visitor.after_entity_type( target, path )
20    visitor.after_node( target, path )
21  end
```

Der Methodenaufruf erfolgt unter Angabe von 3 Argumenten, von denen 2 optional sind. Obligatorisch ist die Übergabe einer Visitor-Instanz, die entsprechende *Callbacks* bereitstellt, um auf die Verarbeitung des Entitätstyp-Knotens einzugehen.

Das Argument `target` enthält die Referenz auf ein Objekt, über dessen Kinder im Zuge der Traversierung iteriert wird. In der Regel ist dieses Objekt entweder eine Instanz des Typknotens, also `self`, oder aber ein Instanzknoten, der den Besuchsaufruf an seine Strategie delegiert. Das dritte Argument beinhaltet die Elemente des aktuellen Pfades und ist wichtig für die Erkennung von Zyklen, die aufgrund von Aggregationsbeziehungen auftreten können. Wird an dieser Stelle nichts übergeben, so wird ein leerer Pfad für die folgenden Besuchsinstanzen eröffnet.

Die Erkennung von Zyklen erfolgt durch den Aufruf in Zeile 3. Liefert der Aufruf der gleichen Methode in der Oberklasse *TypeGraphElement* `false` zurück, so wird dort unter Betrachtung der bisherigen Pfadkonfiguration ein Zyklus ausgemacht und die Methode vorzeitig verlassen. Bei `super`-Aufrufen werden, sofern nicht anderes angegeben, alle Originalargumente übergeben.

Der Aufruf des `visit_abstract`-Callbacks in Zeile 5 ermöglicht dem Visitor die Vorbereitung auf den bevorstehenden Besuch. Hier können zum Beispiel allgemeine Verwaltungsaufgaben erledigt werden, die nicht an den Typ des Knotens geknüpft sind.

In Zeile 7 kann der Visitor einen Iterations-Abbruch bewirken, falls dies aus einem bestimmten Grund notwendig ist. Nach dieser Zeile erfolgt der eigentliche Besuch, der in 3 zeitliche Abschnitte unterteilt ist und den Aufruf von typspezifischen und generischen Callbacks beinhaltet. Nach den `before`- und `visit`-Callbacks erfolgt die Iteration über die weitere Knotenstruktur. Jedem direkten Nachfolgeknoten wird eine entsprechende Nachricht gesendet und erhält eine neue Pfadkopie. Der Besuch schließt mit dem Aufruf der `after`-Callbacks.

5.3.1.3 Die Klasse *TypeNode*

Die Klasse *TypeNode* bildet die Basisklasse für alle verwendeten Typknoten und spezifiziert unter anderem abstrakte Methoden, die in der späteren konkreten Implementierung Listen aller ein- bzw. ausgehender Kanten zurückliefern. Die Methoden `in_edges` und `out_edges` werden so insbesondere benötigt, um das Besuchen von Kanten zu initiieren.

5.3.1.4 Die Klasse *AnnotationTypeNode*

Die Klasse *AnnotationTypeNode* bildet die Basis für alle vorgestellten Annotationstypen, wie sie in dem UML-Diagramm A.2 zu sehen sind. Sie beinhaltet die Factory-Methoden [GHJV04, S. 131ff.] `build` zur Erschaffung von Typobjekten sowie `create_instance_node` und `create_instance_child` zum Ausprägen von Instanzknoten. Weiterhin definiert sie abstrakte Operationen, mit denen diese Instanzknoten verwaltet werden können. Auf diesen Aspekt soll später mit einem konkreten Beispiel eingegangen werden.

Insbesondere die *handle*-Methoden spielen eine wesentliche Rolle spielen, da sie das Setzen, Verwenden und Löschen von Knoten bzw. deren Werten ermöglichen. Diese Operationen sind relativ aufwendig, da sie viele Bedingungen berücksichtigen müssen und wurden deshalb gemäß dem Muster der *Schablonenmethode* [GHJV04, S. 366ff.] dekomponiert. In diesem Zusammenhang kommen auch sogenannte *Hooks* zum Einsatz, die zu bestimmten Zeitpunkten des Verarbeitungsprozesses aufgerufen werden und individuell auf die entsprechenden Ereignisse reagieren können. Eine unvollständige Auflistung aller Hook-Operationen ist im Kopf des Methodenabschnitts zu sehen.

Weiterhin werden die Eigenschaften `pool_in_edge` sowie `membership_in_edges` und `membership_out_edges` vererbt. Die letzten beiden Attribute sind aus der Konzeptvorstellung bekannt und beinhalten Listen von Mitgliedschaftskanten, die in den entsprechenden Typknoten hinein- bzw. herausführen. Zu dem erstgenannten Attribut soll eine kurze Erläuterung folgen.

Alle vorgestellten Typknoten werden unabhängig von ihren Beziehungen untereinander in der Implementierung zusätzlich mittels *Pool-Kanten* an sogenannte *Pool-Knoten* gehängt. Alle Annotationstypen werden zum Beispiel direkt von einem sogenannten Annotationstyppool-Knoten subordiniert, alle Entitätstyp-Knoten von einem sogenannten Entitätstyppool-Knoten, etc. Dies dient unter anderem der Vereinfachung des Zugriffs auf einzelne Typelemente und ist ein Spezifikum der Implementierung, weshalb auf eine Erwähnung im Konzeptkapitel verzichtet wurde.

5.3.1.5 Die Klasse *RootNode*

Die Klasse *RootNode* bildet die Basis für alle Typknoten, die keine Annotationstypen darstellen. Sie bringt sowohl das Attribut `pool_in_edge` als auch `membership_out_edges` mit. Letzteres stellt einen Container für alle ausgehenden Mitgliedschaftskanten dar. Eingehende Mitgliedschaftskanten sind für alle ableitenden Knotenklassen nicht vorgesehen.

5.3.1.6 Unterklassen von *RootNode*

Eine wichtige Unterklasse von *RootNode* stellt *EntityTypeNode* dar. Diese Klasse repräsentiert Entitätstyp-Knoten und bringt eine ganze Reihe von Attributen und Methoden mit, um die Konnektivität zu Klassen-, Super-, Schnittstellen- sowie Cluster-Knoten zu realisieren und abzufragen. Außerdem bietet sie *propagate*-Operationen, mit denen nach aggregierenden Entitäten gesucht werden kann. Von ihr erbt die Klasse *RelationshipEntityTypeNode*, die Beziehungstypen realisiert.

Weitere Unterklassen von *RootNode* sind *EntityTypeClassNode*, *SuperEntityTypeNode*, *EntityTypeInterfaceNode* sowie *ClusterNode*, die an dieser Stelle nicht weiter erläutert werden, da sie in der aktuellen Form nur rudimentär ausgeprägt sind.

5.3.2 UML-Klassendiagramm 2

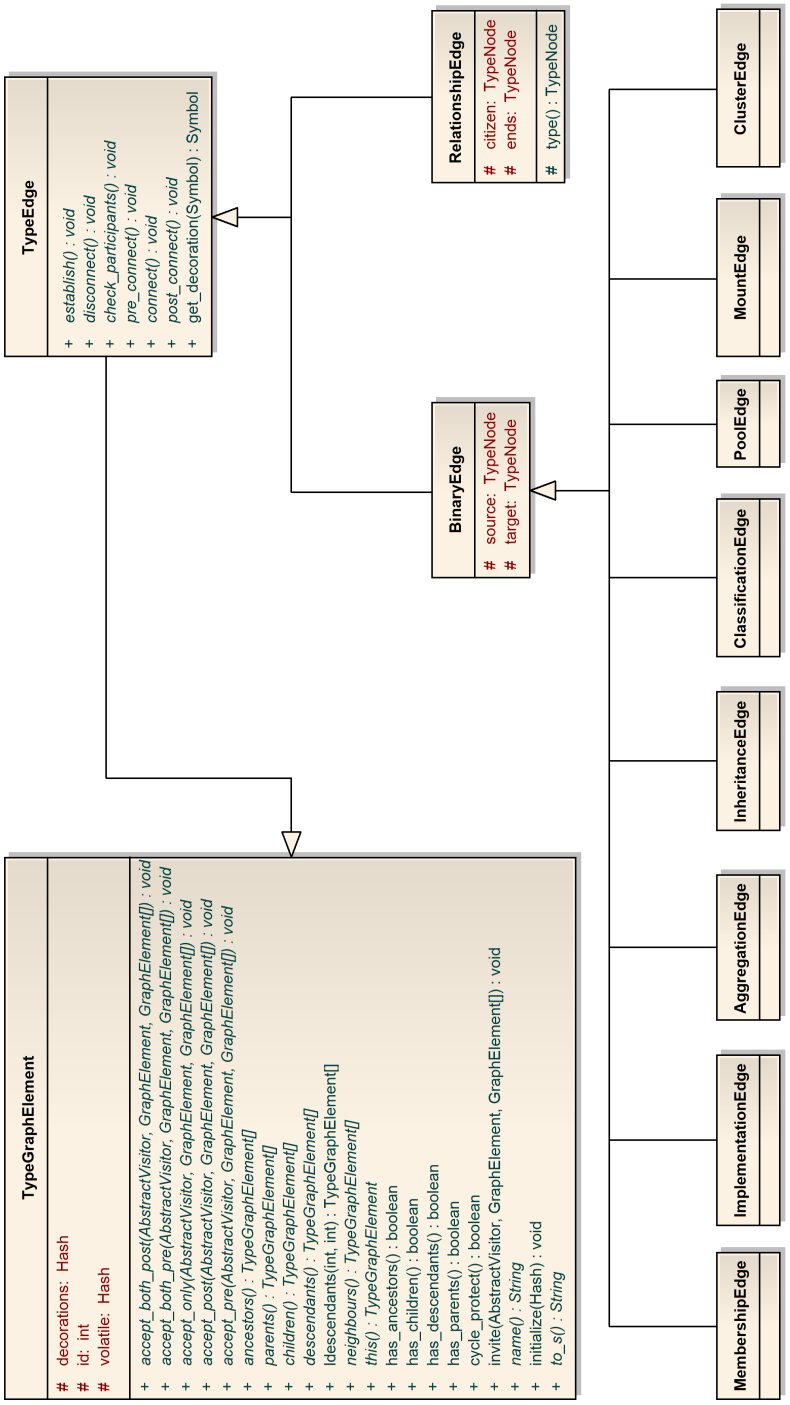

Abbildung 5.4: 2. UML-Klassendiagramm Typgraph

5.3.2.1 Die Klasse *TypeEdge* und ihre Unterklassen

Die Klasse *TypeEdge* erbt von der bereits erläuterten Klasse *TypeGraphElement* und bildet die abstrakte Vorlage für alle semantischen Kanten, die für die Schemamodellierung benötigt werden. Von ihr leitet *BinaryEdge* ab, die wiederum die Vorlage für alle binären Kanten darstellt und die Attribute `source` sowie `target` vererbt. Ein Sonderfall stellt die Klasse *RelationshipEdge* dar, die nicht auf den binären Fall beschränkt ist und *Hyperkanten* repräsentieren kann. Damit ist sie in der Lage, n-äre Beziehungstypen darzustellen, die sie mittels des Attributs `citizen` referenziert. Die Enden werden in dem Attribut `ends` gespeichert.

Die Methoden in der Oberklasse *TypeEdge* dienen der Prüfung und Etablierung der Kanten. Sie stellen unter anderem sicher, dass die aufgehängten Knoten vom richtigen Typ sind und diese nicht bereits referenziert werden.

5.4 Instanzgraph

Der Instanzgraph besteht aus einer Menge von Instanzknoten, die miteinander verbunden sind. Im Gegensatz zum Typgraph werden die Kanten dabei nicht explizit modelliert, sondern über bidirektionale Referenzen realisiert. Instanzknoten werden durch entsprechende Ruby-Klassen repräsentiert, die das folgende Klassendiagramm vorstellt.

5.4.1 Die Klasse *InstanceNode* und ihre Unterklassen

Die Klasse *InstanceNode* bildet die Basis für alle verfügbaren Instanzknotentypen. Sie bringt zum einen die Attribute `in_references` und `out_references` mit, in denen Referenzen auf andere Knoten gespeichert werden, um Kanten auszubilden. Zum anderen besitzt sie das Attribut `type`, das eine Referenz auf die gewählte Typstrategie hält, sowie ein Attribut namens `literal` zum Speichern des Typliterals. Letzteres ermöglicht den Aufbau von Roh-Instanzgraphen, die erst nach ihrem Aufbau typisiert werden. In dem Attribut *volatile* können flüchtige Informationen gespeichert werden, die während der Arbeit mit dem Knoten anfallen.

Neben den aufgeführten Attributen bringt die Klasse *InstanceNode* auch eine Reihe von Methoden mit. Mit Hilfe der *Factory*-Methode `build_native` können entsprechende Knoten über eine statische Klassenmethode ausgebildet werden. Des Weiteren liefert *InstanceNode* eine ganze Reihe von Methoden, um Referenzkanten

zu verwalten und zu prüfen, ob ein Instanzknoten bereits an einer Kante hängt. Die *interconnection*-Methoden etablieren dabei eine bidirektionale Verbindung, während die *reference*-Methoden die Menge der ein- bzw. ausgehende Referenzen gezielt bearbeiten.

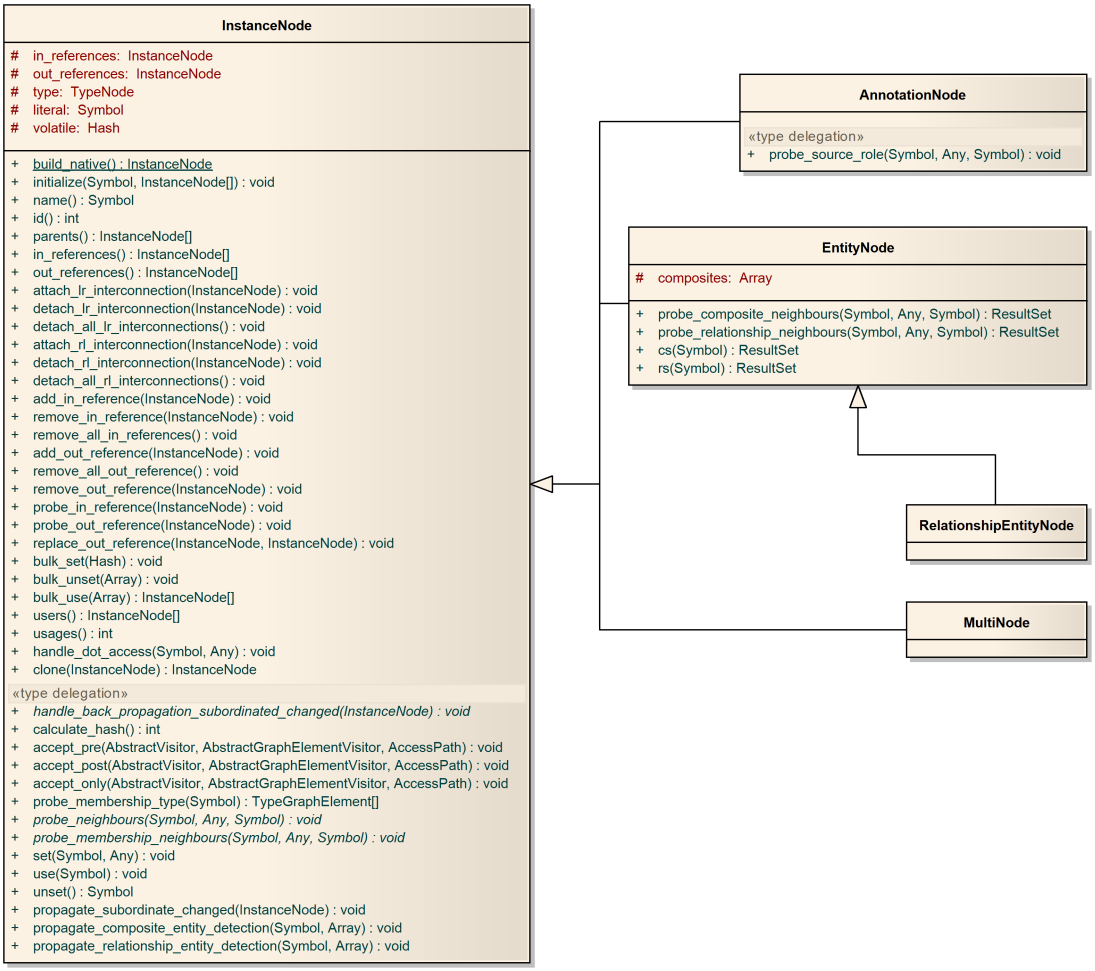

Abbildung 5.5: UML-Klassendiagramm Instanzgraph

Die *bulk*-Methoden werden verwendet, um das Verwenden, Setzen und Entfernen von Knoten bzw. Werten auf einer Menge von Zielknoten zu veranlassen. Es folgen die verbleibenden Methoden mit einer kurzen Beschreibung. Auf die Elemente im Stereotyp `«type delegation»` wird gesondert eingegangen.

users

Liefert alle referenzierenden Knoten.

usages

Liefert die Anzahl der referenzierenden Knoten.

handle_dot_access

Diese Methode realisiert den Zugriff auf Merkmale mittels Punktnotation, wie sie im objektorientierten Bereich gebräuchlich ist.

clone

Veranlasst den Knoten, sich selbst zu klonen und die Kopie zurückzugeben.

Die Arbeit mit dem Instanzgraph wird durch weitere Methoden unterstützt, deren Aufrufe an die jeweilige Typstrategie delegiert wird. Neben den bereits bekannten *accept*-Methoden spielen hier insbesondere die *probe*-Methoden eine wichtige Rolle, da sie das *Probing*-Verfahren etablieren, auf das in Abschnitt 5.7 genauer eingegangen wird. Mit Hilfe der Methoden `set`, `use` und `unset` können Merkmalsstrukturen bzw. Merkmalsbelegungen angelegt und geändert werden. Auf die Methoden `propagate_composite_entity_detection` sowie `propagate_relationship_entity_detection` wird im Zuge des Probing-Verfahrens noch genauer eingegangen.

Auf die Methode `propagate_subordinate_changed` wird im folgenden Abschnitt ausführlicher eingegangen.

5.5 Änderungen an unifizierten Substrukturen

5.5.1 Aufbau eines korrekten Initialzustandes

Bedingt durch die Unifikation können an sich korrekte Änderungen an Knotenstrukturen zu inkonsistenten Datenbankzuständen führen, falls die geänderten Merkmalsausprägungen mehrfach referenziert werden. Dies soll anhand eines Beispiels verdeutlicht werden.

Grafik 5.6 führt ein einfaches Schema für die Speicherung von personenbezogenen Daten ein. Der Entitätstyp *Person* unterhält eine Mitgliedschaft mit dem strukturierten Annotationstyp *Address*, der sich einerseits aus den flachen Merkmalen *Street*, *Number*, *City* sowie *Postcode* und andererseits aus einem ebenfalls strukturierten Typen *Name* zusammensetzt. Der Typ *Name* selbst besteht aus den einfachen Merkmalen *FirstName* sowie *LastName*.

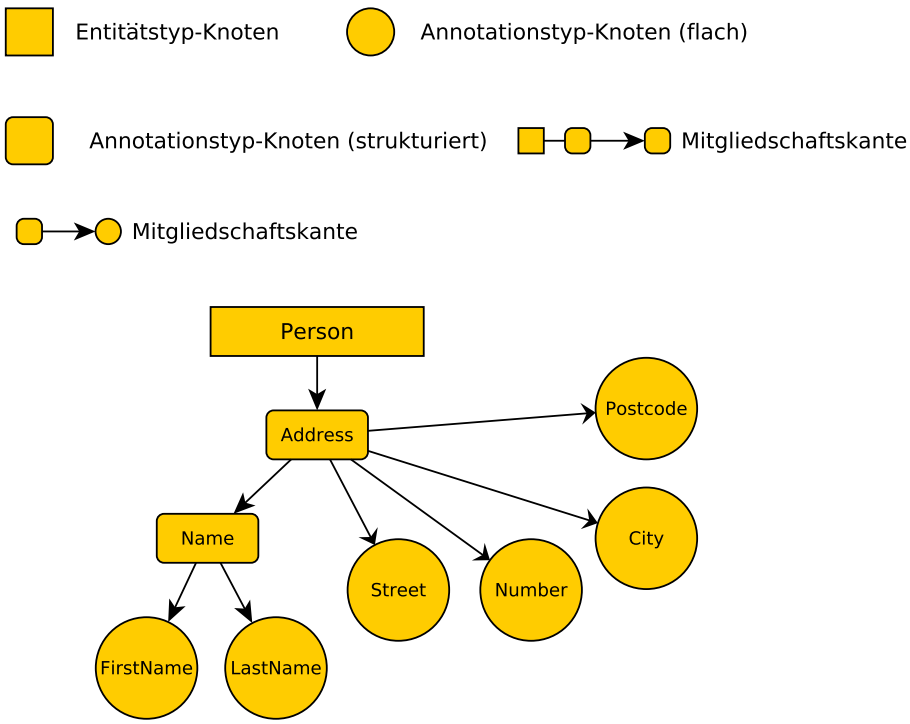

Abbildung 5.6: Beispielschema Personendatenbank

Unter Berücksichtigung dieser Schemainformationen zeigt Grafik 5.7 einen unifizierten Zustand, in dem 3 Personen ausgeprägt werden.

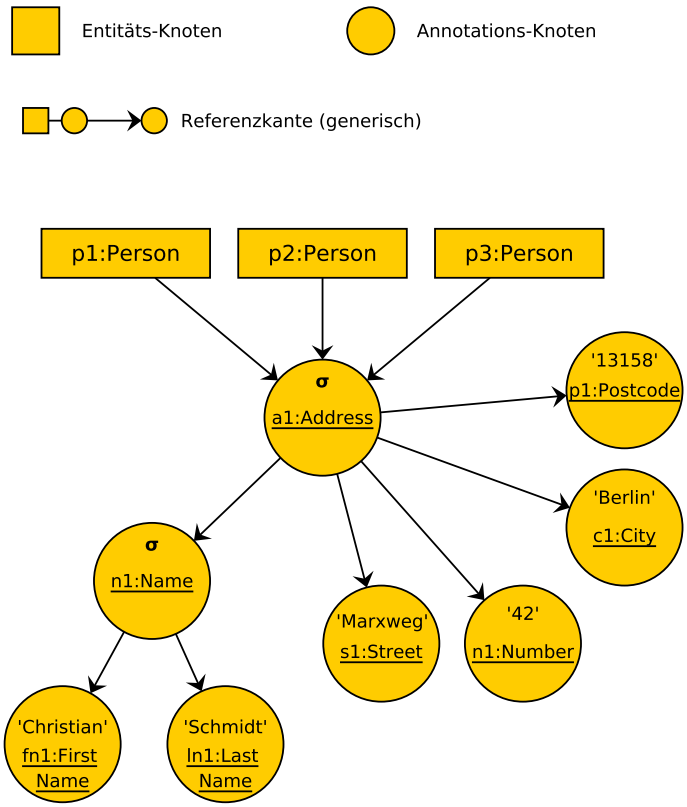

Abbildung 5.7: Konsistenter Ausgangszustand

In diesem Beispiel sei davon ausgegangen, dass alle 3 Personen denselben Namen tragen und gemeinsam in einer Wohngemeinschaft leben.

Der Aufbau des gezeigten Zustandes erfolgt mittels der folgenden Befehlssequenz, die kurz erläutert werden soll. In den Zeilen 1, 11 und 14 werden jeweils neue Entitäten vom Typ *Person* ausgeprägt. Die Merkmalsausprägungen und entsprechenden Wertebelegungen erfolgen für Person 1 (`p1`) in den Zeilen 3-4 und 6-9. Person 2 sowie 3 erhalten die Zuweisung der Adresse mittels Referenzübergabe in den Zeilen 12 sowie 15. An dieser Stelle sei angemerkt, dass Namen von Annotationstypen beim Zugriff grundsätzlich auf ein signifikantes Maß abgekürzt werden können.

Der Zugriff mittels Typname wird durch die Verwendung des bereits erwähnten *Probing*-Verfahrens ermöglicht, auf das in 5.7 eingegangen wird und zum Komplex der Navigationsmöglichkeiten innerhalb der Typ- bzw. Instanzstrukturen gehört. In den letzten drei Zeilen werden alle Entitäten gespeichert und dem *Indoor*-Bereich übergeben.

Listing 5.2: Befehlssequenz zum Anlegen der Personeninstanzen

```
1  p1 = E[:Person]
2
3  p1.Address.Name.FirstName = 'Christian'
4  p1.Address.Name.LastName  = 'Schmidt'
5
6  p1.A.Street   = 'Marxweg'
7  p1.A.Number   = '42'
8  p1.A.City     = 'Berlin'
9  p1.A.Postcode = '13158'
10
11 p2 = E[:Person]
12 p2.A = p1.A
13
14 p3 = E[:Person]
15 p3.A = p1.A
16
17 p1.store
18 p2.store
19 p3.store
```

5.5.2 Zustandskorruption

Angenommen Person 2 zieht um und erhält eine neue Adresse, wie es mittels der folgenden Befehlssequenz umgesetzt wird:

Listing 5.3: Befehlssequenz zum Ändern der Adresse von Person 2

```
1  p2.A.S  = 'Müllerweg'
2  p2.A.Nu = '23'
3
4  p2.store
```

Bei naiver Verarbeitung dieser Anweisungen und ohne Berücksichtigung der Unifikation ergibt sich der in Grafik 5.8 visualisierte Zustand, der den bisherigen Datenbestand weitreichend korrumpiert, da er nicht berücksichtigt, dass die Personenausprägungen `p1` sowie `p3` den Addressknoten referenzieren. Anders als bei der Verarbeitung von klassischen Objektstrukturen muss also bei der Manipulation des Instanzgraphen zwingend darauf geachtet werden, ob Attributstrukturen *geteilt* werden.

5.5.3 Korrekturmaßnahmen

Dies geschieht durch Prüfung der entsprechenden Referenzen und durch Duplikation der betroffenen Knotenstrukturen. Dieses Vorgehen soll anhand des gegebenen Beispiels

erläutert werden. Der Zustand nach Ausführung der ersten Befehlssequenz 5.2 sei mit σ_0 bezeichnet, der Zustand nach Ausführung der zweiten Sequenz 5.3 mit σ_{error}. Ausgehend vom ersten Zustand σ_0 erfolgt die erste Änderungsanweisung der zweiten Befehlssequenz 5.3 in Zeile 1.

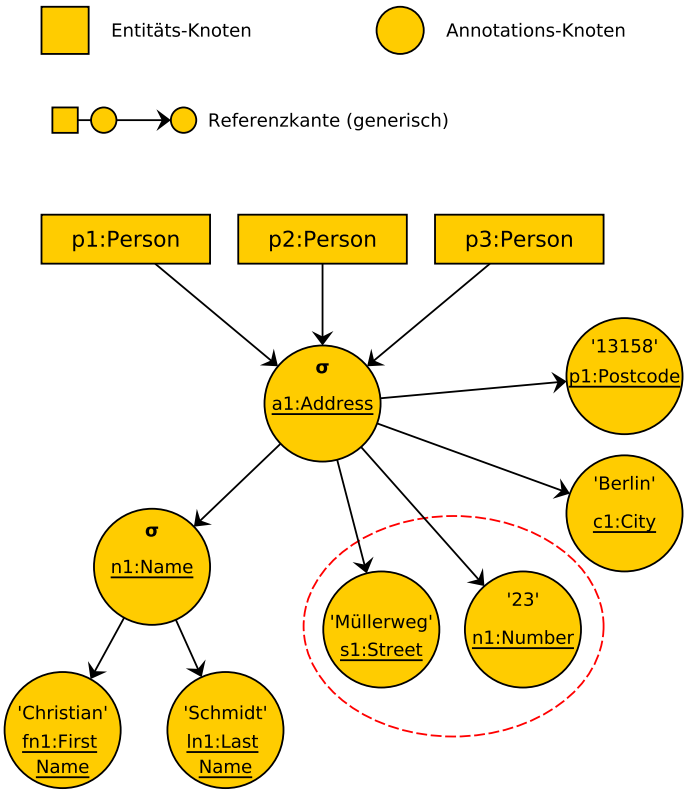

Abbildung 5.8: Inkonsistenter Zustand

Bei der Aktualisierung des Attributs `Street` muss zunächst geprüft werden, ob dieses mehrfach referenziert wird. Ist dies nicht der Fall, so kann es sicher geändert werden, falls es nicht Teil einer Unterstruktur darstellt, wie es für dieses Beispiel konstruiert wurde. Die Prüfung wird in diesem Fall über die Vorfahren des Knotens propagiert, sodass der strukturierte Annotations-Knoten des Typs *Address* ein positives Testergebnis auslöst, da er zusätzlich von den Personeninstanzen `p1` sowie `p3` referenziert wird. Da der Adressknoten selbst nicht als Teil eines strukturierten Attributs ist, endet die Prüfung an dieser Stelle.

Um nun zu verhindern, dass die Änderung von Teilattributen des strukturierten Adressknotens gleichzeitig illegal Änderungen aller in einer Referenzbeziehung stehenden

Entitäten verursacht, wird zunächst ein neuer Straßenknoten `s2` erstellt und der Adressknoten bis auf `s1` geklont. Der duplizierte Adressknoten trägt nun die Kennung `a2`. Anschließend werden alle eingehenden Kanten bis auf jene von `p2` gelöscht. Zuletzt wird eine Kante zum neuen Straßenknoten `s2` eingefügt und die verbleibende Kante von `p2` auf `a1` entfernt. Der hierdurch erzeugte Zustand wird mit σ_1 bezeichnet und ist in Abbildung 5.9 dargestellt.

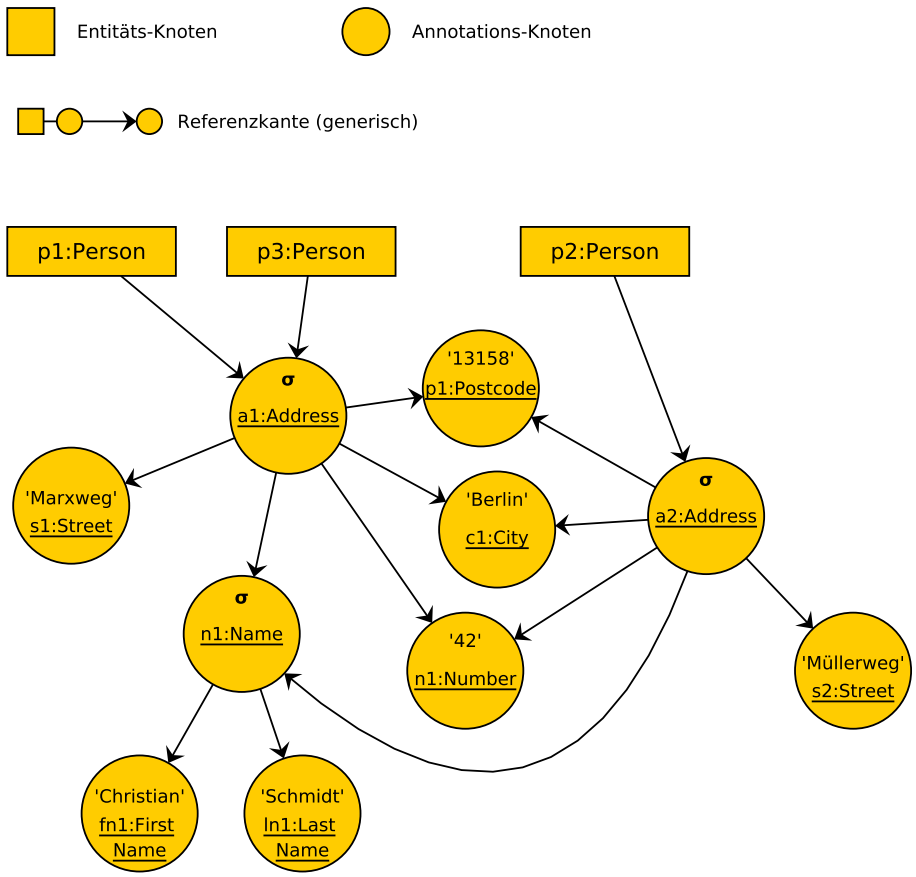

Abbildung 5.9: Zustandskorrektur Schritt 1

Zuletzt bleibt die Verarbeitung der Zuweisung in Zeile 2, die einen neuen Wert für die Hausnummer setzt. Im Zustand σ_1 wird der Hausnummer-Knoten mit dem Wert 42 nicht nur von `a2`, sondern auch von `a1` verwendet. Aus diesem Grund kann der Wert nicht einfach geändert werden. Stattdessen wird ein neuer Knoten vom Typ *Number* angelegt (`n2`) und mit dem Wert 23 ausgeprägt. Anschließend wird die Kante zwischen dem zweiten Adressknoten `a2` und dem ersten Hausnummer-Knoten `n1` gelöscht und eine

neue Kante zwischen `a2` und dem zuvor erstellten Knoten `n2` eingefügt. Das Ergebnis dieser Operation überführt den Zustand σ_1 in σ_2 und ist in Grafik 5.10 abgebildet.

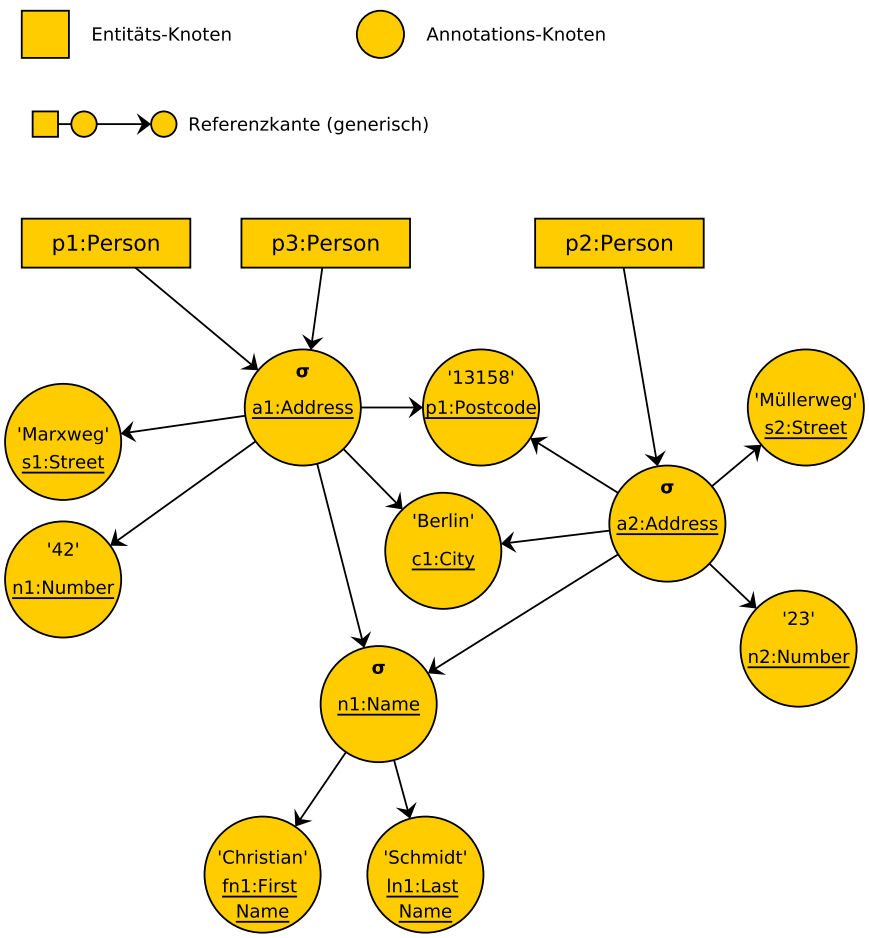

Abbildung 5.10: Zustandskorrektur Schritt 2

Für die Prüfung auf Mehrfachverwendung von Attributstrukturen soll ein weiteres Beispiel gegeben werden, wobei dieses Mal auf die detaillierte Darstellung der Modifikationen am Instanzgraphen verzichtet wird. Auslöser für die Zustandsüberführung sei folgende Anweisung:

Listing 5.4: Befehlssequenz zum Ändern des Nachnamens von Person 1

```
1 p1.A.Name.LastName = 'Schmidt-Meier'
2
3 p1.store
```

Sie drückt den Sachverhalt aus, dass Person 1, Christian Schmidt, heiratet und einen kombinierten Nachnamen erhält. Diese Änderung wird von den oben beschriebenen Prüf- und Umstrukturierungsmaßnahmen begleitet. In diesem Fall muss sowohl der strukturierte Namensknoten geklont werden als auch der superordinierte Adressknoten. Das Ergebnis der Änderungen endet in Zustand σ_3 und ist in Abbildung 5.11 dargestellt.

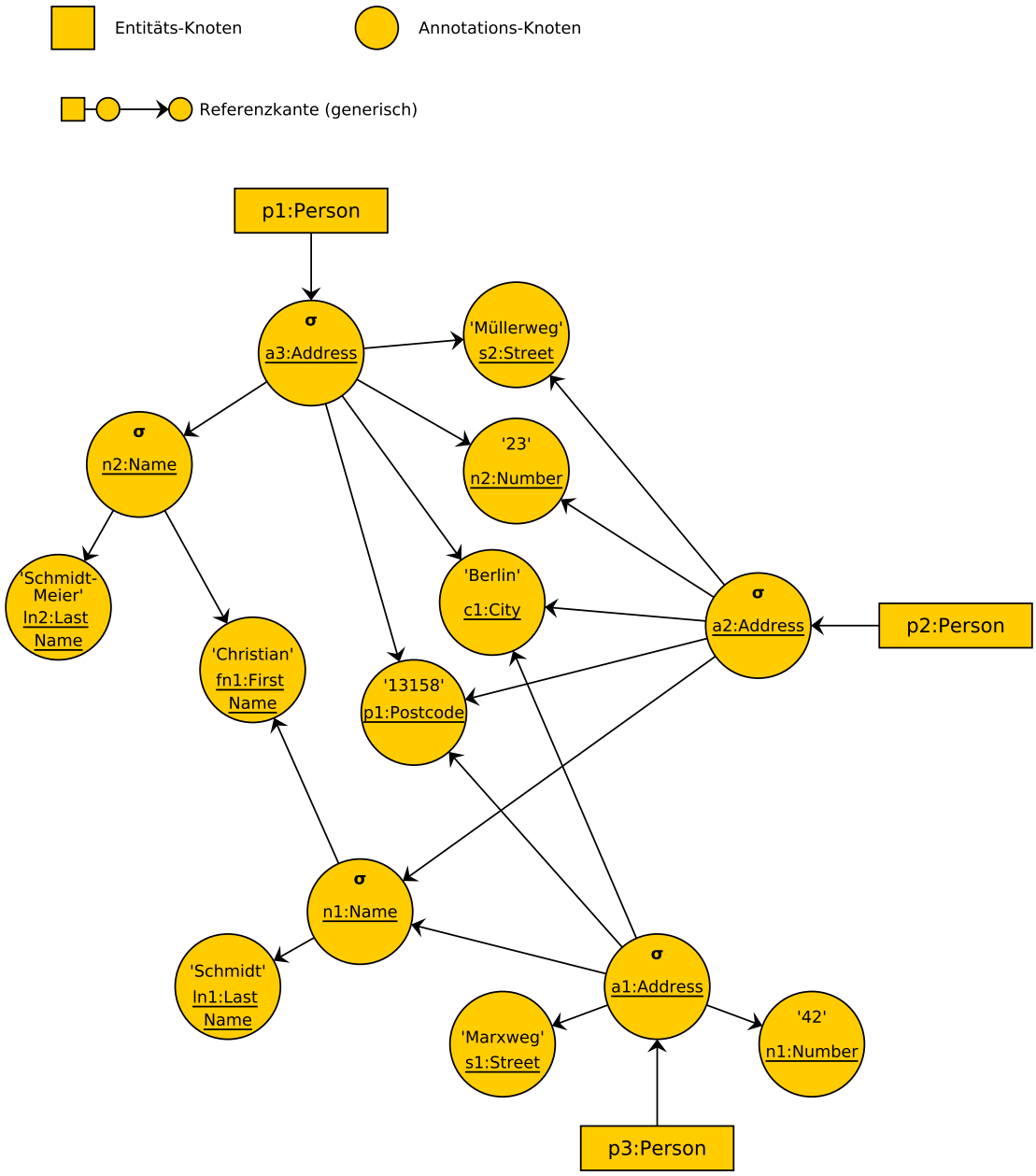

Abbildung 5.11: Zustandskorrektur Schritt 3

5.6 Die *Set*-Operation

5.6.1 Zuweisung und Ausprägung von Attributwerten

Betrachten wir zunächst die Variable `p1` aus dem letzten Abschnitt, die eine Referenz auf ein Objekt der Klasse *EntityNode* hält. Dieses Objekt repräsentiert im Kontext des Graphspeichers eine Entität vom Typ *Person*. Hierfür hält es eine entsprechende Instanz der Klasse *EntityTypeNode*, die Entitätstypen repräsentiert. Diese Typinstanz wird als Strategie verwendet, indem an das Entitätsobjekt gerichtete Methodenaufrufe ggf. an das Entitätstypobjekt delegiert werden. Es gilt an dieser Stelle die Ebene der Implementierung und die Meta-Ebene der Datenmodellierung bzw. -speicherung zu unterscheiden.

Um das Objekt `p1` zu befähigen, Nachrichten in Form von Methodenaufrufen entgegenzunehmen, erfolgt in der Regel die Definition von entsprechenden Methoden im Zuge des Klassenentwurfs. Damit also `p1` beispielsweise auf die Nachricht `Address` reagieren kann, muss es eine Methode `Address` in der Klassendefinition von *EntityTypeNode* bzw. *EntityNode* geben, die den Adressknoten zurückliefert.

Da der Attributzugriff über Punktnotation realisiert wird, muss also für *jedes* Attribut eine entsprechende Methode definiert werden. Und da jeder Entitätstyp unterschiedliche Attribute besitzt, bedarf es demnach auf Ebene der Implementierung entitätstyp-spezifischen Klassen, deren Methodendefinitionen sich nach den verwendeten Annotationstypen richtet. Es müssen also für jeden modellierten Entitätstyp und für alle strukturierten Annotationstypen eigene Ruby-Klassen eingeführt werden.

Dieses Vorgehen ist aufwendig und kann durch Verwendung des Ruby `method_missing`-Hooks vermieden werden. Der Aufruf dieser Methode erfolgt, wenn eine Nachricht von einem Objekt nicht verarbeitet werden kann, da sie in keiner Klasse entlang der Vererbungshierarchie definiert wurde. Als Argumente werden zum einen der Methodenname als auch alle für den vergeblichen Aufruf eingesetzten Argumente übergeben. Dieser Mechanismus ermöglicht es, jene Ruby-Klassen, die für die Modellierung des Typ- bzw. Instanzgraphen verwendet werden, generisch zu halten, indem Attributabfragen dynamisch mittels der `method_missing`-Methode verarbeitet werden.

Die Nachricht `Address` an `p1` verursacht also den Aufruf der `method_missing`-Methode mit den entsprechenden Argumenten, aus denen hervorgeht, dass das Merkmal *Address* angesprungen werden soll. Die Entität `p1` prüft jetzt, ob dieses Attribut Teil seines Schemas ist, indem es eine Anfrage an seinen Typ stellt, der untersucht, ob er als

Entitätstyp-Knoten einen Attributknoten subordiniert, der den angefragten Namen trägt (siehe 5.7). Ist dies der Fall, so wird eine Referenz auf diesen Typknoten zurückgegeben. Auf dieser Referenz wird die `use`-Operation aufgerufen, die untersucht, ob die Entität ein Instanzkind des besagten Typs besitzt. Bei erfolgreicher Prüfung wird dieses Instanzkind zurückgegeben.

Durch Hintereinanderschaltung dieses Vorgehens kann über die vorliegenden Graphstrukturen navigiert werden. So erzeugt die Anweisung `p1.Addr.Name.FirstName` einen Aufruf der `use`-Operation des Annotationstyps *FirstName* und liefert, sofern die Attributstruktur bereits ausgeprägt wurde, den Wert „Christian“.

Mit den vorgestellten Mitteln werden auch Zuweisungen realisiert, die in Ruby überschrieben werden können. Zuweisungen sind in klassischen Programmiersprachen in der Regel Anweisungen, die aus zwei Seiten bestehen, die durch einen Zuweisungsoperator getrennt werden. Auf der linken Seite steht entweder ein atomarer Bezeichner oder der Zugriff auf ein Objektattribut. Auf der rechten Seite steht ein beliebiger Ausdruck, der zunächst ausgewertet werden muss, um das Ergebnis in den von der linken Seite referenzierten Speicherbereich zu schreiben. Dies gilt zunächst auch für Ruby.

Gleichzeitig kann der Zuweisungsoperator, in diesem Fall das Gleichheitszeichen, einfach als Bestandteil eines Methodennamens verwendet werden, sodass eine vermeintliche Zuweisung nichts weiter als ein Methodenaufruf bedeutet. Die Anweisung `p1.A.Name.LastName = 'Schmidt-Meier'` resultiert also in einem Aufruf der Methode `LastName=` auf jenem Objekt, das durch den Aufruf von `p1.Address.Name` zurückgegeben wird. Existiert diese Methode nicht, wird der `method_missing`-Hook angesprungen, der den Methodennamen parst, ein Gleichheitszeichen erkennt und daraufhin die `set`-Operation auf dem Annotationstypobjekt von *LastName* aufruft. Die rechte Seite der Zuweisung wird dabei als Argument übergeben.

5.6.2 Kontrollflussdiagramm *Set*-Operation

Das in Abbildung 5.12 dargestellte Flussdiagramm beschreibt die Arbeit der *set*-Operation für flache singuläre Annotationstypen, wie sie zum Beispiel für das Setzen des Nachnamens im letzten Szenario aufgerufen wird. Das Diagramm beginnt dabei direkt mit dem Aufruf der *set*-Operation und beschäftigt sich nicht mit dem zuvor erläuterten Auflösen der Punktnotation.

Die Knoten sind nach den Namen der aufgerufenen Methoden benannt, zeigen allerdings nicht die Parameter an. Zur besseren Orientierung wurde jeder Knoten

mit einer Zahl versehen. Die Beschreibung erfolgt entlang der sich auffächernden Kontrollflusspfade. Bei der Wahl der Methodennamen wurde darauf geachtet, dass sie sprechend sind und Assoziationen mit ihrer Aufgabe zulassen. Die *set*-Operation selbst ist als Schablonenmethode abstrakt gehalten und konkretisiert ihre Arbeit durch den Aufruf jener Methoden, die den jeweiligen Teilschritt abarbeiten. Die orange eingefärbten Blöcke stellen Prädikate dar.

Der Kontrollfluss startet beim formalen `begin`-Knoten (1) und verweist auf die `set`-Operation (2). Bei diesem Aufruf handelt es sich um eine Delegation des schematisch übergeordneten Instanzknotens an das Objekt des Typs, für das eine Instanz ausgeprägt bzw. eine bereits vorhandene Instanz manipuliert werden soll. Der übergeordnete Knoten wird dabei als Vaterknoten bezeichnet und übergibt seine eigene Speicheradresse als Argument.

Anschließend wird die Methode `dispatch_set` (3) aufgerufen, die mittels des Prädikats `check_entity_outdoor` (4) prüft, ob die Entität, dessen Attribut manipuliert wird, als neu angelegt oder als gespeichert gilt.

Im zweiten Fall befindet sich die Entität im *Indoor*-Bereich und wird folglich aktualisiert. Der hierfür notwendige Kontrollflussstrang wird mit dem Aufruf der `handle_indoor_set` (5) sowie folgenden Blöcken (6) angedeutet und schließt mit dem formalen Endknoten (7).

Ist das Ergebnis der Prüfung positiv und tritt der erste Fall ein, so wird der Kontrollfluss mit dem Aufruf der Methode `handle_outdoor_set` (8) fortgeführt, die mittels des Prädikats `check_single` (9) prüft, ob die maximale Kardinalität des zu setzenden Merkmals kleiner oder gleich 1 ist.

Bei einem negativen Ergebnis wird der Alternativstrang ab `handle_outdoor_multiple_set` (10) abgedeckt, der wiederum über nachfolgende Blöcke (11) bei einem formalen Endknoten (12) endet.

Bei einem positiven Test zunächst `handle_outdoor_single_set` (13) und anschließend `check_instance_child_exists` (14) aufgerufen. Dieses Prädikat ermittelt, ob bereits eine Instanz des Merkmals für den aktuellen Vaterknoten angelegt wurde. Ist dies nicht der Fall, so wird die Bearbeitung mit `handle_outdoor_single_set_on_non_existing` (15) fortgesetzt und endet über nachfolgende Blöcke (16) mit Knoten (17).

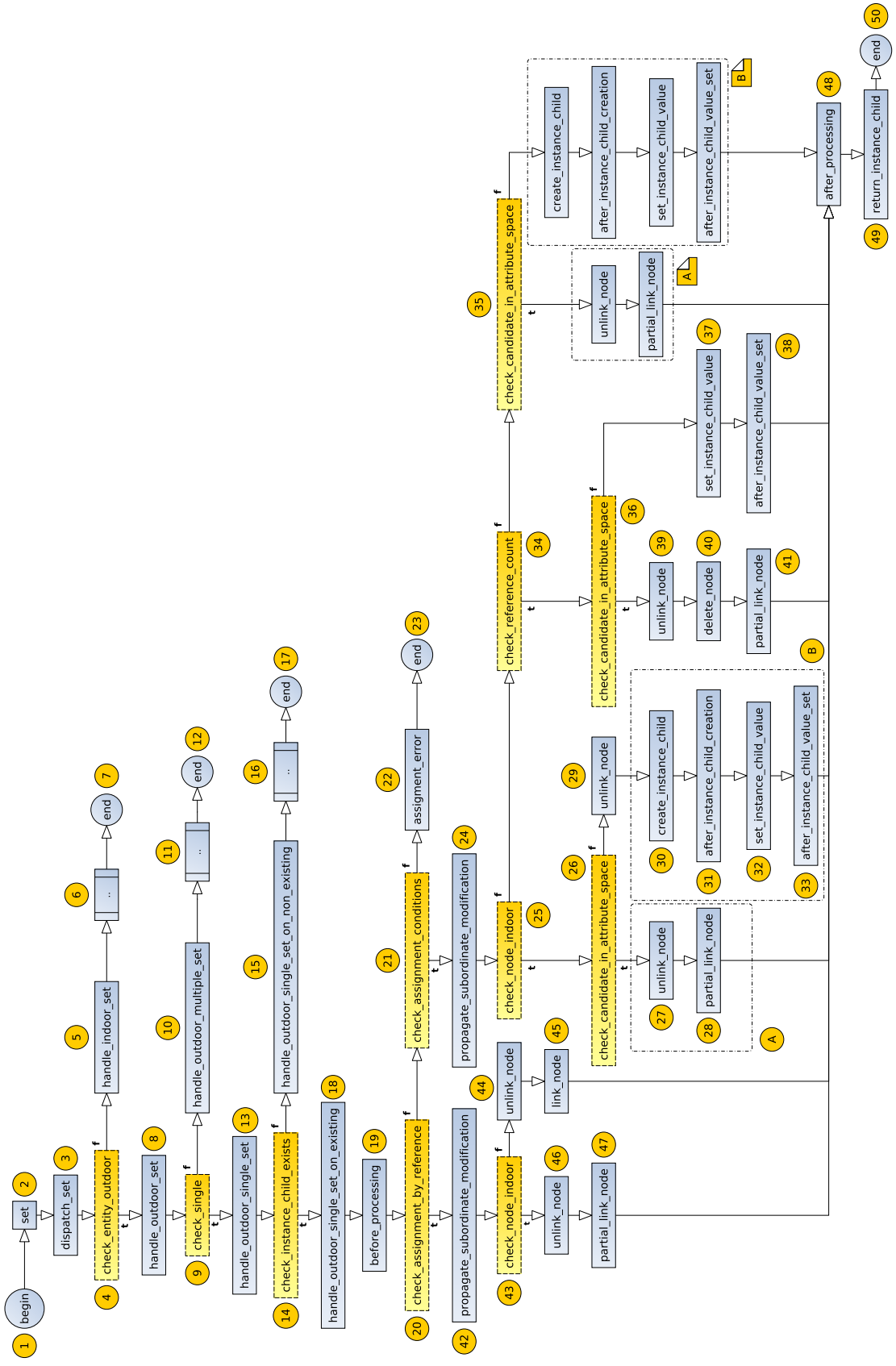

Abbildung 5.12: Kontrollflussdiagramm *Set*-Operation

Existiert bereits eine Instanz, so muss von einer Aktualisierung des Attributwertes ausgegangen werden. Der Aufruf der Methode `handle_outdoor_single_set_on_non_existing` (18) initiiert den hierfür notwendigen Kontrollflussstrang unter den durch die vorherigen Prüfungen ermittelten Annahmen, dass erstens die merkmalstragende Entität zuvor noch nicht gespeichert wurde, und dass zweitens kein Multi-Knoten angelegt werden muss.

Die Hook-Methode `before_processing` (19) bietet die Möglichkeit, den Verarbeitungszyklus individuell zu erweitern. Auf diesen Punkt wird nicht weiter eingegangen. Das folgende Prädikat `check_assignment_by_reference` (20) prüft, ob anstelle eines konkreten Wertes die Referenz auf einen bestehenden Instanzknoten zugewiesen wird und verzweigt den Kontrollfluss in zwei weitere Pfade, die näher erläutert werden.

Im negativen Fall erfolgt die Zuweisung eines konkreten Wertes, der mittels `check_assignment_conditions` (21) auf seine Konsistenz geprüft wird, indem zum Beispiel die Zulässigkeit des Datentyps kontrolliert wird. Schlägt diese Prüfung fehl, so endet der Kontrollfluss mit dem Aufruf der Fehlerroutine `assignment_error` (22) und einem formalen Endknoten (23).

Nach positiver Überprüfung der Zuweisungsbedingungen erfolgt ein Aufruf der Methode `propagate_subordinate_modification`, die entsprechende Maßnahmen zur Konsistenzerhaltung unifizierter Instanzstrukturen vornimmt, wie sie in Abschnitt 5.5 beschrieben werden.

Hiernach muss per `check_node_indoor` (25) ermittelt werden, ob sich der zu aktualisierende Instanzknoten im *Indoor*-Bereich befindet. Dieser Fall tritt auf, wenn bei vorangegangener Wertebelegung eine Unifikation des Instanzknotens möglich war, das heißt ein Knoten mit entsprechender Wertebelegung ausgeprägt werden sollte, dessen Äquivalent bereits im Indoor-Bereich existierte und aus diesem Grund unidirektional referenziert wurde. Diese Prüfung ist außerordentlich wichtig, da der Indoor-Bereich nicht von außen manipuliert werden darf. Sein Zustand darf nur im Zuge einer regulären Speicheroperation geändert werden, die grundsätzlich mit einer Integritätsprüfung einhergeht. Etwaige Aktualisierungsmaßnahmen können folglich nicht direkt auf dem Instanzknoten durchgeführt werden, da dies obligatorisch zu inkonsistenten Datenbankzuständen führt.

Nach positiver Prüfung durch `check_node_indoor` (25) erfolgt eine letzte Fallunterscheidung per `check_candidate_in_attribute_space` (26). Dieser Aufruf prüft, ob ein Instanzkandidat im Indoor-Bereich existiert, der denselben Typ bei einer

Wertebelegung besitzt, die durch die Zuweisung angestrebt wird. Ist dies der Fall, so greift der Unifikationsmechanismus, indem der alte Instanzknoten mit `unlink_node` (27) gelöst wird und der ermittelte Kandidat per `partial_link_node` (28) referenziert wird. Existiert kein Kandidat, so wird die Kante zum bisherigen Knoten via `unlink_node` (29) gelöscht und ein neuer Instanzknoten ausgeprägt, wodurch der im letzten Absatz formulierte Anspruch auf logische Trennung der Outdoor- und Indoor-Bereiche erfüllt wird.

Die Ausprägung von neuen Instanzknoten ist derzeit in 4 Schritte unterteilt und beginnt mit dem Anlegen des neuen Kindknotens durch Aufruf von `create_instance_child` (30), setzt sich fort durch das Anspringen einer Hook-Methode namens `after_instance_child_creation` (31) sowie einer Methode zum Setzen des Wertes namens `set_instance_child_value` (32) und endet mit einer Hook-Methode `after_instance_child_value_set` (33), die es wie auch der zuvor erwähnte *after*-Hook ermöglicht, weitere individuelle Anweisungen ereignisgerecht zu positionieren.

Befindet sich das zu manipulierende Merkmal nicht im Indoor-Bereich, so muss zunächst mittels `check_reference_count` (34) untersucht werden, ob der aktuelle Kindknoten ein- oder mehrfach referenziert wird. Der zweite Fall tritt zum Beispiel auf, wenn er mehreren neu erstellten Entitäten mittels Referenz zugewiesen wird und bedeutet ein negatives Prüfungsergebnis. Als Konsequenz hieraus darf die aktuelle Kind-Instanz nicht einfach geändert werden, da dieses Vorgehen die Konsistenz der referenzierenden Vaterknoten gefährdet. Aus diesem Grund wird ein frischer Instanzknoten erstellt, falls mittels `check_candidate_in_attribute_space` (35) kein Äquivalent ermittelt werden kann und die bereits beschriebene Anweisungssequenz (B) ausgeführt. Andernfalls erfolgt die Ausführung von Sequenz (A).

Erfolgt die einfache Verwendung durch den aktuellen Vaterknoten selbst, so kann der Merkmalsknoten ohne Risiko aktualisiert werden. Dies geschieht nach negativer Prüfung mittels `check_candidate_in_attribute_space` (36) entweder durch Setzen eines neuen Wertes in (37) bzw. (38) oder nach positivem Test durch Rereferenzierung eines Indoor-Merkmals in (41), nachdem die aktuelle Kind-Instanz in (39) ausgehängt und dann per `delete_node` (40) endgültig entfernt wird, da es von keiner Instanzstruktur mehr verwendet wird.

Es bleibt die Beschreibung des Alternativstranges, sofern das Prädikat `check_assignment_by_reference` (20) ein positives Ergebnis liefert, das heisst ein Knoten mittels Referenz zugewiesen wird. An dieser Stelle erfolgt ebenfalls ein Aufruf der Methode `propagate_subordinate_modification` (42), die entsprechende Maßnahmen zur Konsistenzerhaltung unifizierter Instanzstrukturen vornimmt.

Anschließend wird mittels `check_node_indoor` (43) geprüft, ob sich der Ersatzknoten bereits im Indoor-Bereich befindet. Ist dies nicht der Fall, so wird die Kante zur bisherigen Kind-Instanz in (44) gelöscht und der zugewiesene Knoten in (45) *bidirektional* angebunden. Im anderen Fall erfolgt ebenfalls eine Löschung (46) und eine *unidirektionale* Verknüpfung (47) mit dem Indoor-Knoten.

Alle beschriebenen Pfade münden, sofern nicht anders angegeben, im `after_processing`-Hook (48) und anschließend in der Methode `return _instance_child` (49), die den zugewiesenen Wert zurückliefert, bevor der formale `end`-Knoten (50) angesprungen wird.

5.7 Das *Probing*-Verfahren

5.7.1 Navigation in Graphstrukturen

Das *Probing*-Verfahren wurde bereits in den vorangegangen Abschnitten erwähnt. Es dient in erster Linie dazu, herauszufinden, über welche Nachbarknoten navigiert und welche Instanzknotenmenge dabei erfasst werden soll.

Grundsätzlich kann von jedem Instanzknoten aus über abstrahierte Achsen navigiert werden, indem die umliegenden Knoten mittels Standardmethoden adressiert werden. Tabelle 5.1 gibt eine kurze Übersicht über diese Methoden.

Tabelle 5.1: Adressierung von Knoten im Typgraph

Achse	**adressierte Knoten**
`parents`	direkte Vorgängerknoten (Elternknoten)
`ancestors`	Vorgängerknoten (Vorfahren)
`self (this)`	aktueller Knoten selbst
`children`	direkte Nachfolgerknoten (Kindknoten)
`descendants`	Nachfolgerknoten (Nachfahren)
`siblings`	Nachbarknoten (bei gleichem Elternknoten)
`neighbours`	direkte Nachbarknoten

Um über die angegeben Achsen navigieren zu können, ist es wichtig, innerhalb der Graphstruktur unterschiedliche Richtungen zu definieren. Aus diesem Grund erfolgt die

Einführung einer virtuellen Baumansicht für Entitäten, die aufgrund der Unifikation nicht gegeben ist. Entitäts-Knoten bilden bei dieser Abstraktion die Wurzelknoten, denen entsprechende Attributknoten untergeordnet sind. Diese Wurzelknoten können aufgrund von Aggregationsbeziehungen wiederum Vorfahren haben, was grundsätzlich zu Zyklen führen kann. Diese müssen für die Baumabbildung entsprechend behandelt werden. Beispielgrafik 5.13 deutet die Entitäten als Bäume durch Umrandung der zugehörigen Knotenstrukturen an. Die genauen Achsendefinitionen hängen dabei vom verwendeten Knotentyp ab.

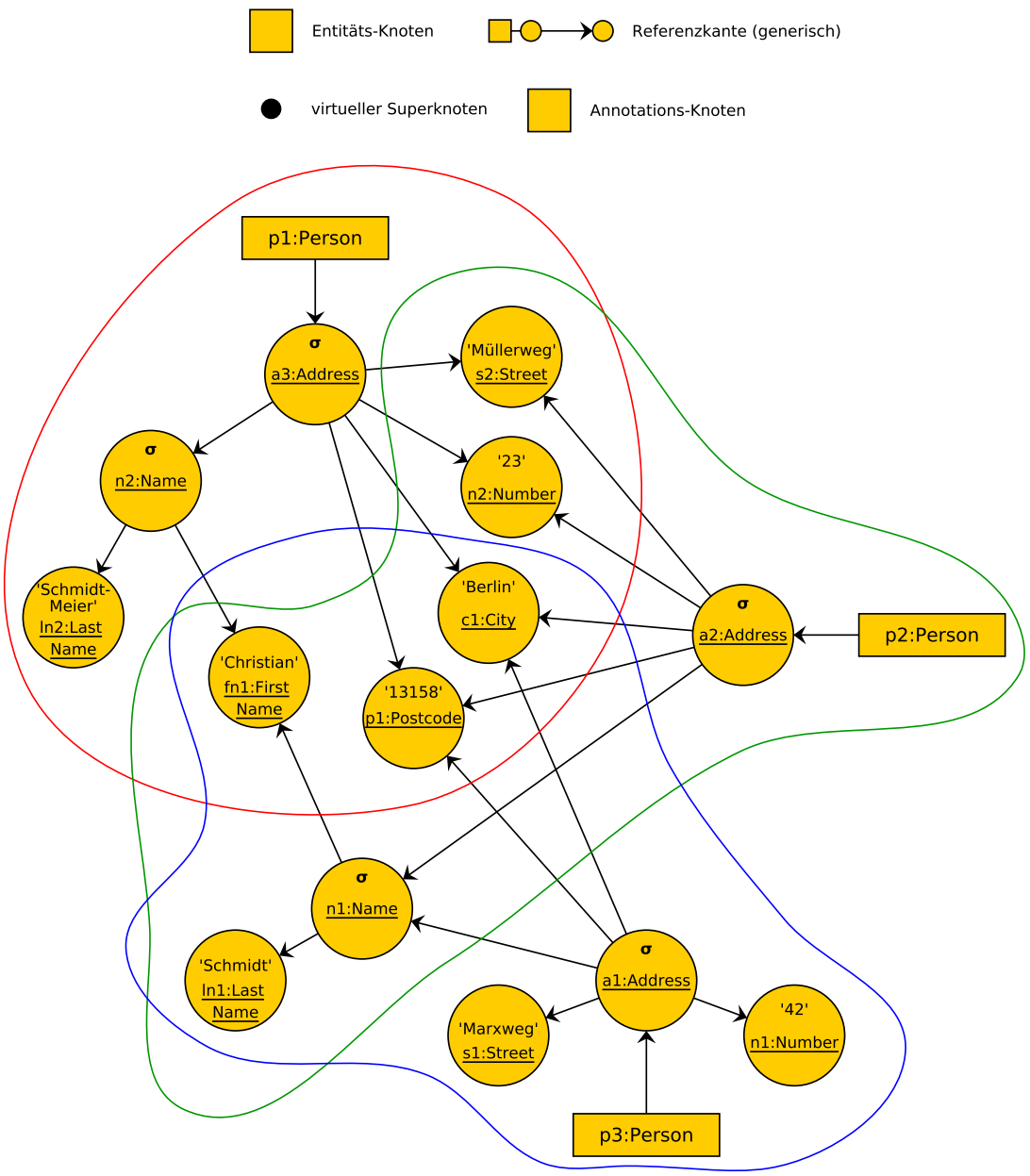

Abbildung 5.13: Baumsicht

5.7.2 Grafisches BibTeX-Schema

Eine weitere Möglichkeit ist die Navigation über semi- bzw. voll qualifizierte Adressierung unter Bezugnahme auf bestimmten Knotentypen bzw. -strukturen oder

aber unter Angabe der schematischen Bezeichnungen. Dies soll anhand eines Zustandes näher erläutert werden, der vom in Grafik 5.14 angegeben Schema abgeleitet wird. Als Vorlage für dieses Schema dient das *BibTeX*-Beispiel von Martin Gogolla [Gog12]. Diese Materialien beinhalten ebenfalls konkrete Daten, mit denen ein Instanzgraph aufgebaut wird.

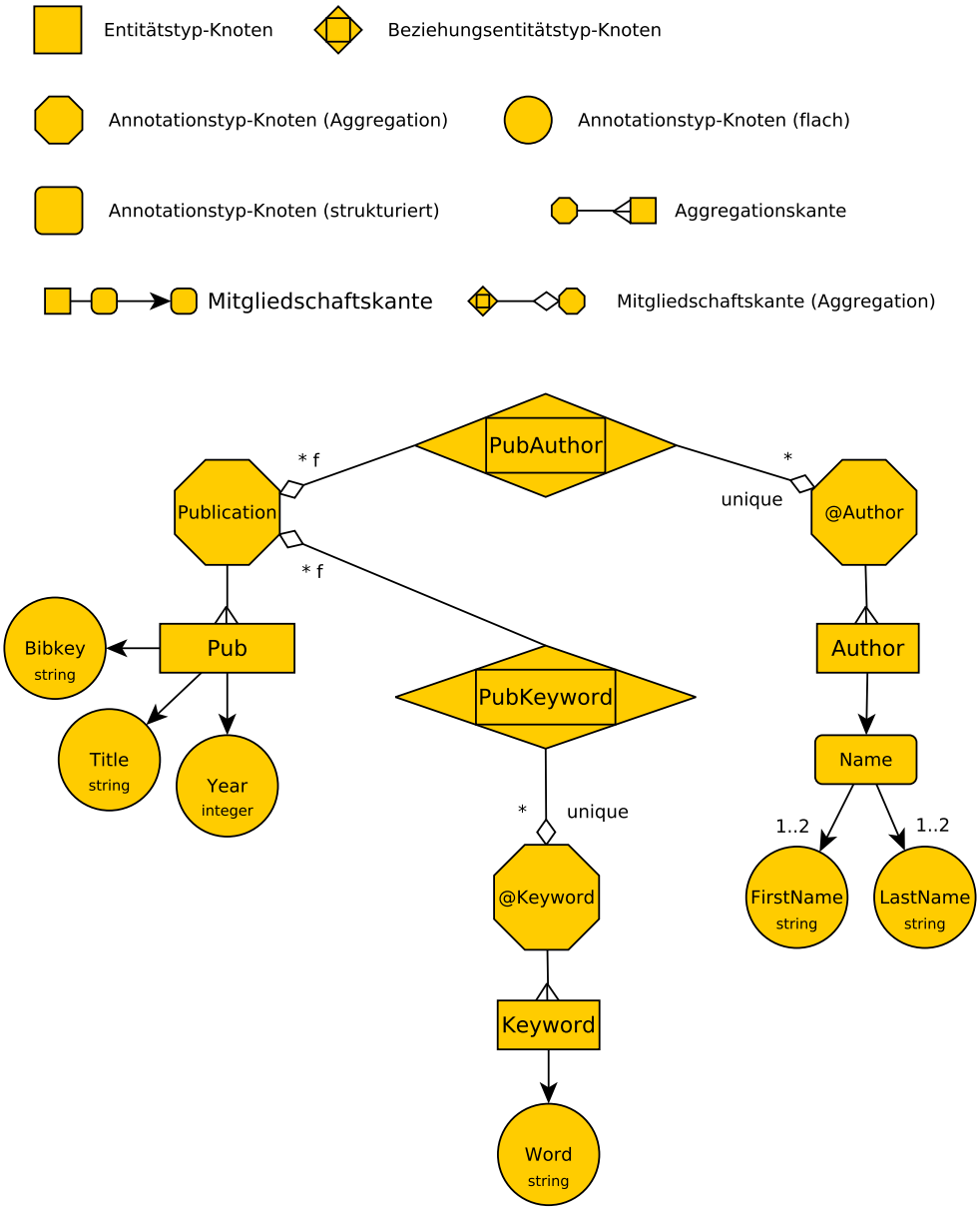

Abbildung 5.14: Schema BibTeX-Beispiel

5.7.3 Textuelle Repräsentation des BibTeX-Schemas

Die direkte Interpretation von Typgraphen ist derzeit noch nicht möglich. Aus diesem Grund wurde eine einfache textuelle Datendefinitionssprache als Ruby-DSL (engl., Sg.: *domain specific lanuage*) entwickelt, die ohne formale Syntax und expliziten Aufbau eines Parse-Baums auskommt. Entsprechend benannte Methoden definieren dabei die Syntaxelemente dieser Sprache. Beim Einlesen werden diese Methoden aufgerufen, die als Argumente sowohl Werte als auch Anweisungsblöcke akzeptieren, die wiederum Methodenaufrufe beinhalten und nach dem Closure-Konzept unabhängig vom Ausführungszeitpunkt auf ihren Erstellungskontext zurückgreifen können. Auf diese Weise wird das Absteigen in verschachtelte Strukturen realisiert.

Die textuelle Darstellung des Schemas aus 5.14 wird in den folgenden Abschnitten dargestellt. Die Definitionen sind dabei nicht erschöpfend, da sie nicht alle verfügbaren Direktiven für die Festlegung von Integritätsbedingungen nutzen.

Die folgende Anweisungssequenz zeigt die textuelle Definition aller benötigten Annotationstypen, die von allen Entitätstypen bzw. den Beziehungsentitätstypen verwendet werden können, indem sie diese per Mitgliedschaftskanten an sich binden.

Listing 5.5: Tibet-Definitionsdirektiven für die verwendeten Annotationstypen

```
1   # explicit definition of 'bibkey'
2   at :Bibkey do
3     required do
4       mode    :plain
5       data    :string
6       status  :active
7     end
8   end
9
10  # using defaults for 'title'
11  at :Title
12
13  # using defaults for 'year' except for data type setting
14  at :Year do
15    data :integer
16  end
17
18  # using defaults
19  at :FirstName
20  at :LastName
21
22  # definition of structured type 'name'
```

```
23      at :Name do
24        mode :structured
25
26        use :FirstName do
27          membership do
28            maximum_cardinality 2
29          end
30        end
31
32        use :LastName do
33          membership do
34            maximum_cardinality 2
35          end
36        end
37
38      end
39
40      # using defaults for 'word'
41      at :Word
42
43      # defining aggregation types
44      # using target directive to specify role name
45      at :Publication do
46        target :Pub
47        mode :aggregation, :shared
48      end
49
50      at :@Author do
51        target :Author
52        mode :aggregation, :shared
53      end
54
55      at :@Keyword do
56        target :Keyword
57        mode :aggregation, :shared
58      end
```

Das Schema verwendet die Entitätstypen *Pub*, *Author* sowie *Keyword* und die Beziehungsentitätstypen *PubAuthor* sowie *PubKeyword*, deren textuelle Repräsentation folgen.

Listing 5.6: Tibet-Definitionsdirektiven für die verwendeten Entitätstypen

```
1       entity_type :Pub do
2
3         # explicit definition
```

```
4         use :Bibkey do
5           membership do
6             minimum_cardinality 1
7             maximum_cardinality 1
8           end
9         end
10
11        # using membership defaults implies cardinality of (1,1)
12        use :Title
13        use :Year
14
15      end
16
17      entity_type :Author do
18
19        # using membership defaults here, too
20        use :Name
21
22      end
23
24      entity_type :Keyword do
25
26        # using membership defaults here, too
27        use :Word
28
29      end
```

Listing 5.7: Tibet-Definitionsdirektiven für die verwendeten Beziehungstypen

```
1       entity_type :PubAuthor do
2
3         classified :Relationship
4
5         use :@Pub do
6           membership do
7             maximum_cardinality :*, :fixed
8           end
9         end
10
11        use :@Author do
12          membership do
13            maximum_cardinality :*
14          end
15        end
16
17      end
```

```
18
19      entity_type :PubKeyword do
20
21        classified :Relationship
22
23        use :@Pub do
24          membership do
25            maximum_cardinality :*, :fixed
26          end
27        end
28
29        use :@Keyword do
30          membership do
31            maximum_cardinality :*
32          end
33        end
34
35      end
```

5.7.4 Beispielabfragen

Es folgt eine Reihe von Beispielen für Navigationsanweisungen, wie sie mit der derzeitigen Implementierung verwendet werden können. Die Variable `pub1` referenziert einen Entitäts-Knoten vom Typ *Pub*, also eine konkrete Ausprägung mit entsprechenden Attributen und Beziehungen, wie sie gemäß dem Schema erlaubt sind. Die Rückgabe erfolgt entweder in Form einer Kollektion, die eine geordnete Menge (engl., Sg.: *Ordered Set*) repräsentiert.

Navigation zu Annotations-Knoten

Anweisung:	`pub1.Title`
Alternative:	`pub1.T`
Ergebnis:	`{title3:Title}`

Unter Angabe des Typnamens kann zum entsprechenden Instanzknoten navigiert werden. Der Typname kann soweit abgekürzt werden, wie er eindeutig bleibt.

Abfrage von Knoteninhalten

Anweisung: `pub1.Title.content`

Alternative: `pub1.T.content`

Ergebnis: `'OCL4ALL - Modelling Systems with OCL'`

Die Methode `content` liefert den Wert von flachen Knoten.

Navigation zu Beziehungs-Knoten

(1) Anweisung: `pub1.rs`

(1) Alternative: `pub1.rs(:Pub)`

(1) Ergebnis: `{pubkeyword2728:PubKeyword, pubauthor4862:PubAuthor}`

(2) Anweisung: `pub1.rs(:PubKeyword)`

(2) Ergebnis: `{pubkeyword2728:PubKeyword}`

Die Methode `rs` liefert alle Beziehungsinstanzen vom Typ *PubKeyword* bzw. *PubAuthor* zurück, an denen `pub1` partizipiert. Dieser Methode kann ein Filterargument übergeben werden, sodass nur jene Beziehungs-Knoten zurückgeliefert werden, deren Typnamen partiell bzw. vollständig übereinstimmen. Die Navigation über die Aggregations-Knoten, mit denen Beziehungen realisiert werden, erfolgt transparent.

Navigation über Beziehungs-Knoten

Anweisung: `pub1.Keyword`

Ergebnis: `{keyword873:Keyword, keyword1345:Keyword,`
`keyword1353:Keyword, keyword1385:Keyword, ..}`

Liefert alle Entitäten vom Typ *Keyword*, die über sämtliche Beziehungen, an denen `pub1` teilnimmt, erreichbar sind.

Navigation über Beziehungs-Knoten zu Annotations-Knoten

Anweisung: `pub1.Keyword.Word`

Ergebnis: `{word838:Word, word1346:Word,`
`word1354:Word, word1386:Word, ..}`

Liefert alle Annotations-Knoten vom Typ *Word* aller Entitäten vom Typ *Keyword*, die über sämtliche Beziehungen erreichbar sind.

Navigation über Beziehungs-Knoten und Rollennamen

Anweisung: `pub1.Keyword.Pub`

Alternative: `pub1.Keyword.Publication`

Ergebnis: `{pub9:Pub, pub13:Pub, pub57:Pub, pub61:Pub, ..}`

Liefert transitiv sämtliche Entitäten vom Typ *Pub*, die mit allen von `pub1` aus erreichbaren Instanzen vom Typ *Keyword* in Beziehung stehen. Die alternative Anweisung verwendet einen Rollennamen, mit dem schematisch der Aggregationstyp-Knoten bezeichnet wird.

Indizierter Zugriff auf Instanzknoten

Anweisung: `pub1.Keyword[0].Word`

Ergebnis: `{word838:Word}`

Liefert das Attribut *Word* der ersten Entität vom Typ *Keyword*, die direkt von `pub1` aus erreichbar ist.

Zugriff auf alle aggregierenden Entitäten

Anweisung: `pub1.cs`

Ergebnis: `{pubkeyword2728:PubKeyword, pubauthor4862:PubAuthor}`

Die Methode `cs` liefert alle Entitäten zurück, die `pub1` aggregieren. Dies können sowohl Beziehungstypinstanzen sein als auch Entitäten. Die Ergebnismenge enthält stets auch alle Beziehungstypinstanzen.

Zugriff auf alle Verwender eines Knotens

Anweisung: `pub1.Year.users`

Ergebnis: `{pub1:Pub}`

Die Methode `users` liefert alle Instanzknoten zurück, die aufgrund der Unifikation ebenfalls auf diesen Knoten zeigen und damit dieselbe Ausprägung des angegebenen Typs referenzieren.

6 Evaluation

6.1 Experiment 82 Millionen Deutsche

6.1.1 Kompression durch Unifikation

Das Konzept unterstützt die Reduzierung des Speicheraufwands durch *Deduplizierung* von einzelnen Knoten bzw. Knotenstrukturen. Durch diese Unifikation der Knotenmenge kann eine effektive verlustfreie *Kompression* des vorliegenden Datenmaterials erreicht werden, indem bereits vorhandene Knoten bzw. Knotenstrukturen durch Referenzierung wiederverwendet werden.

Je nach Gesamtdatenmaterial können dabei hohe Kompressionsraten erzielt werden, wie das folgende theoretische Experiment zeigen soll. Wenn in den folgenden Abschnitten von Kompressionsrate die Rede ist, dann ist damit das Größenverhältnis zwischen der komprimierten und der unkomprimierten Darstellung bezüglich des benötigten Speicherplatzes gemeint. Die Anzeige der Kompressionsrate bzw. der Kompressionsfaktor kann dabei über eine Verhältnisangabe oder prozentual erfolgen. Der Kompressionsfaktor 1:4 sagt zum Beispiel aus, dass das Originalmaterial aufgrund der günstig gewählten Repräsentation in 25 Prozent des ursprünglich benötigten Speicherplatzes untergebracht werden kann.

Bei dem folgenden Experiment geht es um die Speicherung von Datensätzen, die personenbezogene Informationen für alle deutschen Staatsbürger speichern. Ein für dieses Vorhaben einfach gehaltenes Schema ist in Grafik 6.1 angegeben.

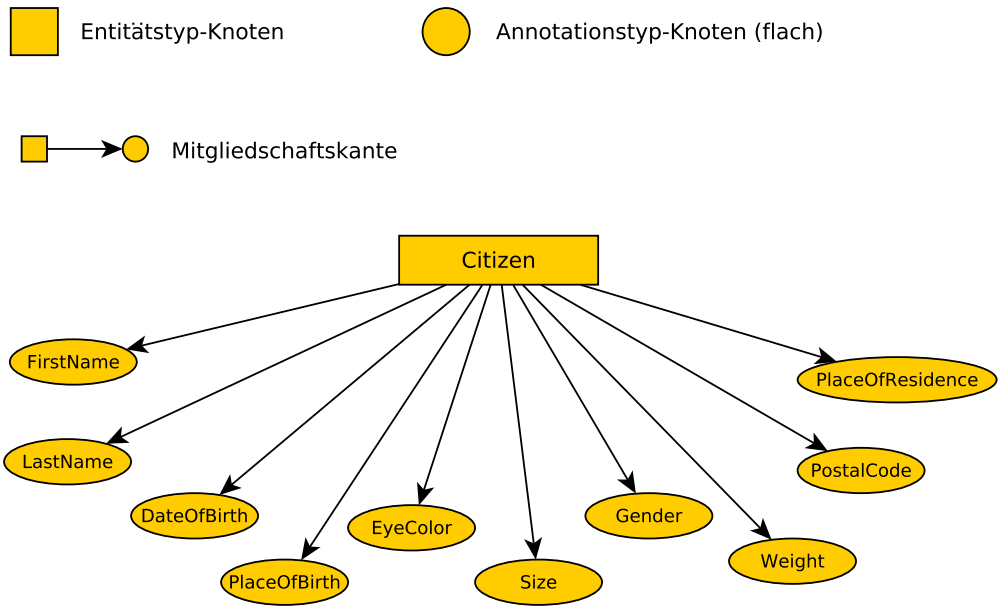

Abbildung 6.1: Beispielschema Personendatenbank

Um den Kompressionsfaktor ermitteln zu können, muss es zwei Messungen geben. Eine Messung bezieht sich auf den theoretischen Speicheraufwand ohne Unifikation, die andere Messung auf den Speicheraufwand mit aktivierter Deduplizierung. Grundlage dieser Messungen ist der Aufbau eines entsprechenden Datenbankzustandes und die Annahme, dass die Ausprägung eines Knotens grundsätzlich aufwendiger ist als die bloße Referenzierung auf bereits bestehende Knoten. Die Messungen beziehen sich an dieser Stelle vereinfacht auf die Anzahl der instanziierten Knoten und berücksichtigen damit nicht den zusätzlichen Aufwand, der in einer konkreten Implementierung durch Speicherung von Referenzen verursacht wird.

6.1.2 Datenerhebung

Vor der Durchführung des Experiments wird zunächst die Art der zu speichernden Informationen betrachtet, um eine Abschätzung über die möglichen unterschiedlichen Ausprägungen der einzelnen Merkmale geben zu können. Dabei wird davon ausgegangen, dass jedes Attribut mit einem Element einer diskreten Wertemenge belegt wird.

Für den definierten Entitätstyp *Citizen* in Grafik 6.1 sind 10 Attribute vorgesehen, die nun insbesondere in Hinblick auf ihre möglichen Wertebereiche betrachtet werden sollen. Die Abschätzung der Mächtigkeit der einzelnen Wertebereichsmengen in

einem möglichen Datenbankzustand dient der Berechnung der theoretisch möglichen Kompressionsrate. Bei der Analyse der einzelnen Attribute wird der Einfachheit halber festgelegt, dass sich alle möglichen Attributsausprägungen auf Deutschland beziehen, was zum Beispiel für die Belegung des Merkmals *Wohnort* eine wichtige Rolle spielt, da die Zahl der möglichen Wohnorte in diesem Fall relativ genau ermittelt werden kann.

6.1.2.1 Attribut Vorname

Die Abschätzung des Wertebereichs für Vornamen kann an dieser Stelle nur grob vorgenommen werden. Durchgeführte Recherchen zur Anzahl deutscher Vornamen ergaben, dass 25 Prozent der Bundesbürger einen der am häufigsten 2000 Vornamen trägt. Dies bedeutet, dass 60 Millionen Menschen einen Vornamen besitzen, der weniger häufig vorkommt. Auf Platz 2000 der genannten Häufigkeitsliste steht der Name „Fatih“ mit 237 Trägern. Für die angegebenen 60 Millionen Menschen gilt also, dass sie einen Namen besitzen, der höchstens 236 Mal vorkommt, wobei die tatsächliche Verteilung unbekannt ist. Unter der Annahme, dass alle weiteren Namen mit einer konstanten Häufigkeit von 236 auftreten, kann die Wertemenge für Vornamen auf eine Größe von insgesamt ca. 250.000 Elemente geschätzt werden.

6.1.2.2 Attribut Nachname

Die Menge der möglichen Ausprägungen für das Merkmal Nachname wird ebenfalls grob abgeschätzt. Nach vorliegenden Informationen tragen ca. 16 Prozent der Bundesbürger einen der am häufigsten 3400 Nachnamen. Die letzte Position dieser Häufigkeitsliste hält der Name „Dose“ mit 1000 Vorkommen. Dies bedeutet, dass 67 Millionen Deutsche einen Nachnamen mit einer geringeren Häufigkeit tragen. Unter der Annahme, dass alle weiteren Nachnamen mit einer konstanten Häufigkeit von 999 auftreten, kann die Wertemenge auf eine Größe von insgesamt ca. 70.000 Elemente geschätzt werden.

6.1.2.3 Attribut Geburtsdatum

Unter der Annahme, dass die Lebensspanne eines Menschen höchstens 120 Jahre beträgt und das Geburtsdatum bis auf den Tag genau aufgelöst wird, kann die Anzahl der möglichen Elemente auf ca. 42.000 geschätzt werden. Hierbei wird davon ausgegangen, dass das Geburtsdatum als Einheit gespeichert wird, zum Beispiel als „16.01.1948“, und dass nur Informationen über lebende Personen erfasst werden.

6.1.2.4 Attribut Geburtsort

Gesetzt den Fall, dass alle Bürger in Deutschland geboren wurden, ergibt sich für diesen Wertebereich eine Größe von ca. 12.000.

6.1.2.5 Attribut Augenfarbe

Die Anzahl der reinen Augenfarben blau, braun, grün und grau beträgt 4. Aus diesen Grundfarben ergeben sich durch Kombination insgesamt 15 Mischfarben und damit die Anzahl der möglichen Wertebelegungen.

6.1.2.6 Attribut Körpergröße

Bei vorliegendem Hochwuchs kann ein Mensch eine Körpergröße von ca. 260 Zentimetern erreichen. Bei dieser Auflösung enthält die Menge der möglichen Werte also 260 Elemente.

6.1.2.7 Attribut Geschlecht

Unter Berücksichtigung potentieller Intersexualität beinhaltet die Menge in diesem Fall 3 Elemente.

6.1.2.8 Attribut Körpergewicht

Der schwerste Mensch der Welt hat ein Körpergewicht von ca. 600 Kilogramm. Bleiben wir bei dieser Maßeinheit ohne Dezimalangabe, so setzt sich die Menge der möglichen Merkmalsausprägungen aus 600 Elementen zusammen.

6.1.2.9 Attribut Postleitzahl

Ausgehend von der Anzahl der vergebenen Postleitzahlen 1992 kann die Größe des Wertebereichs auf ca. 27.000 geschätzt werden.

6.1.2.10 Attribut Wohnort

Unter Annahme, dass alle deutschen Staatsbürger auch in Deutschland leben, entspricht die Menge möglicher Wohnorte genau der Menge der zuvor angegebenen Geburtsorte.

6.1.3 Durchführung und Ergebnis

Bei den nun folgenden Messungen wird davon ausgegangen, dass in Deutschland derzeit ca. 82 Millionen Menschen leben.

Für die Messung ohne Kompression ist die Berechnung der Knotenanzahl trivial, da für jedes Attribut genau 82 Millionen Knoten angelegt werden müssen. Zu diesen Attributknoten kommen noch einmal 82 Millionen Knoten für die ausgeprägten Entitätstyp-Knoten. Der Speicheraufwand beträgt demnach insgesamt 902 Millionen Knoten.

Für die Messung mit Kompression muss die Anzahl der möglichen Wertebelegungen pro Merkmal berücksichtigt werden, wie in Tabelle 6.1 dargestellt. Die Berechnung der Kompressionsrate erfolgt dabei über die Anzahl der maximalen Ausprägungen von 82 Millionen pro Merkmal.

Tabelle 6.1: Kompressionsraten der einzelnen Attribute

Knotenart	**Name**	**Anzahl Instanzen**	**Kompression**
Entitätstyp	`Citizen`	82.000.000	1:1
Annotationstyp	`FirstName`	250.000	1:328
Annotationstyp	`LastName`	70.000	1:1171
Annotationstyp	`DateOfBirth`	42.000	1:1952
Annotationstyp	`PlaceOfBirth`	12.000	1:6833
Annotationstyp	`EyeColor`	15	1:5466667
Annotationstyp	`Size`	260	1:31538
Annotationstyp	`Gender`	3	1:27333333
Annotationstyp	`Weight`	600	1:136667
Annotationstyp	`Postalcode`	27000	1:3037
Annotationstyp	`PlaceOfResidence`	0	k. A., siehe Geburtsort

Wie der Tabelle 6.1 zu entnehmen ist, wird die Anzahl instantiierter Knoten unter Einsatz der Kompression auf 82,4 Millionen erheblich reduziert. Unter Einbezug der ersten Messung kann so eine Kompressionsrate von 1:11 ermittelt werden. Die Deduplizierung führt in diesem Beispiel also zu einer Reduzierung des Speicherbedarfs um rund 91 Prozent.

6.2 BibTeX-Experiment

Die folgenden empirischen Beobachtungen sollen konkrete Daten zum Zeit- und Speicheraufwand beim Anlegen eines Datenbankzustandes liefern. Hierzu wird auf Basis des BibTeX-Beispiels [Gog12] von Martin Gogolla ein entsprechendes Schema abgeleitet und mit den ebenfalls vorliegenden Beispieldaten instanziiert. Das Schema ist aus vorherigen Erläuterungen zur Navigation bereits bekannt (s. 5.14). Die Beispieldaten bestehen aus einer Reihe von Datensätzen zu jedem Entitäts- bzw. Beziehungsentitätstyp. Eine Übersicht über die Mächtigkeit der einzelnen Ausprägungsmengen liefert Tabelle 6.2.

Tabelle 6.2: Ausprägungsmengen und ihre Mächtigkeit 1

Knotenart	**Typname**	**Anzahl Datensätze**
Entität	Pub	205
Entität	Keyword	491
Entität	Author	152
Beziehungs-Entität	PubKeyword	1519
Beziehungs-Entität	PubAuthor	594

6.2.1 Theoretische Betrachtung der Beispieldaten

Diese Informationen dienen als Grundlage für die Berechnung der zu erwartenden Instanzknotenmengen und liefert so eine Prognose zur quantitativen Beschaffenheit des Beispielzustandes ohne Einsatz von Unifikation und ohne spezielle Darstellung von n:m-Beziehungen mittels multipler Mitgliedschaften. Hierzu muss neben der reinen Anzahl der Entitäten bzw. Beziehungen auch die Anzahl der Annotations-Knoten berücksichtigt werden. Eine Übersicht über die theoretisch zu erwartende Knotenzusammensetzung und -anzahl zeigt die folgende Tabelle 6.3.

Im ersten und zweiten Tabellenabschnitt werden alle zu erwartenden Instanzknoten in Verbindung mit dem Typ *Pub* sowie *Keyword* berechnet. Hier gibt es keine Besonderheiten.

Im dritten Abschnitt erfolgt die Ermittlung der Knotenmengen für den Typ *Author* und dessen Merkmal *Name*, das einen strukturierten Typ darstellt. An dieser Stelle beginnt die Rechnung mit 152 Knoten. Da jeder Autor mindestens einen Vor- und

Nachnamen besitzt, erhöht sich der Knotenbedarf mindestens um weitere 304 Knoten. Und da es sich bei den Typen *FirstName* und *LastName* um multiple Attribute handelt, fließen weitere Multi-Knoten in die Endsumme von 925 ein.

Tabelle 6.3: Ausprägungsmengen und ihre Mächtigkeit 2

Knotenart	**Typname**	**Anzahl Datensätze**
Entität	Pub	205
Annotation	Bibkey	205
Annotation	Title	205
Annotation	Year	205
Summe		820
Entität	Keyword	491
Annotation	Word	491
Summe		982
Entität	Author	152
Annotation	Name	152
Multi-Knoten	FirstName	152
Annotation	FirstName	163
Multi-Knoten	LastName	152
Annotation	LastName	154
Summe		925
Beziehungs-Entität	PubKeyword	1519
Annotation	@Pub	1519
Annotation	@Keyword	1519
Summe		4557
Beziehungs-Entität	PubAuthor	594
Annotation	@Pub	594
Annotation	@Author	594
Summe		1782
Summe insgesamt		9066

Der folgende Abschnitt berechnet den Knotenbedarf rund um den Beziehungstyp *PubKeyword*. Hier fallen zusätzlich zu den 1519 Beziehungs-Knoten selbst weitere 3038 Aggregations-Knoten an, die in der Summe von 4557 berücksichtigt werden müssen. Nach dem gleichen Prinzip erfolgt die Berechnung im letzten Abschnitt für den Beziehungstyp *PubAuthor*.

Ein durchgeführter Test mit einem alternativen Schema, das die Eigenschaften für die theoretische Berechnung erfüllt, ergab mit einer Gesamtknotenanzahl von 9053 eine Abweichung von 13 Knoten, die mit einer etwas erhöhten Anzahl für Vor- und Nachnamen erklärt werden kann.

6.2.2 Praktische Betrachtung der Beispieldaten

Nach der theoretischen Betrachtung folgt nun eine tabellarische Zusammenfassung (s. 6.4) der Messergebnisse für die Anzahl der Knoten bei spezieller Darstellung der n:m-Beziehung, bei der die Abbildung über multiple Mitgliedschaften der nicht fixierten Teilnehmer realisiert wird (s. 4.6.14). Hierzu wird das zu Beginn angesprochene Schema 5.14 verwendet.

Tabelle 6.4: Ausprägungsmengen und ihre Mächtigkeit 3

Knotenart	**Typname**	**Anzahl Datensätze**
Entität	Pub	205
Annotation	Bibkey	205
Annotation	Title	205
Annotation	Year	205
Summe		820
Entität	Keyword	491
Annotation	Word	491
Summe		982
Entität	Author	152
Annotation	Name	152
Multi-Knoten	FirstName	152
Annotation	FirstName	163
Multi-Knoten	LastName	152
Annotation	LastName	154
Summe		925
Beziehungs-Entität	PubKeyword	205
Annotation	@Pub	205
Multi-Knoten	@Keyword	205
Annotation	@Keyword	1519
Summe		2134
Beziehungs-Entität	PubAuthor	205
Annotation	@Pub	205
Multi-Knoten	@Author	205
Annotation	@Author	594
Summe		1209
Summe insgesamt		6070

Bei der Betrachtung der Ergebnisse wird deutlich, dass insbesondere die Knotenanzahl im Kontext der Beziehungsausprägungen erheblich gesunken ist, da nicht mehr für jede einzelne Abbildung zwischen zwei Elementen der in Beziehung stehenden Mengen tatsächlich ein Beziehungs-Knoten sowie ein entsprechender Aggregations-Knoten ausgeprägt werden muss. Für die Zuordnung zwischen Instanzen vom Typ *Pub* und Instanzen der Typen *Keyword* und *Author* verwenden die Beziehungstypen jeweils multiple Aggregations-Knoten, die an zusätzlichen Multi-Knoten hängen.

6.2.3 Kompression des Beispielzustandes mittels Unifikation

Bevor nun eine detaillierte Erläuterung über Zeit- und Speicheraufwand gegeben wird, soll die folgende Tabelle darüber Aufschluss geben, wie die Knotenanzahl unter Einsatz der Unifikation minimiert werden kann, ohne dass die Integrität der Daten gestört wird. Bei der folgenden Messung werden alle Knotentypen unifiziert.

Tabelle 6.5: Ausprägungsmengen und ihre Mächtigkeit 4

Knotenart	**Typname**	**Anzahl Datensätze**
Entität	Pub	205
Annotation	Bibkey	205
Annotation	Title	192
Annotation	Year	30
Summe		632
Entität	Keyword	491
Annotation	Word	491
Summe		982
Entität	Author	152
Annotation	Name	152
Multi-Knoten	FirstName	135
Annotation	FirstName	144
Multi-Knoten	LastName	150
Annotation	LastName	151
Summe		884
Beziehungs-Entität	PubKeyword	205
Annotation	@Pub	**205
Multi-Knoten	@Keyword	192
Annotation	@Keyword	491
Summe		1093
Beziehungs-Entität	PubAuthor	205
Annotation	@Pub	**geteilt
Multi-Knoten	@Author	92
Annotation	@Author	152
Summe		449
Summe insgesamt		4040

6.2.4 Messungen zu Zeit- und Speicheraufwand

Für den Einsatz einer Datenbank ist die Betrachtung des Ressourcenbedarfs wichtig. An dieser Stelle sollen deshalb detaillierte Messungen der Ladezeit und des Arbeitsspeicherbedarfs angeführt werden.

6.2.4.1 Messaufgabe und Maßeinheiten

Die Messaufgabe ist der vollständige Aufbau des Instanzgraphen mit den Beispieldaten im Arbeitsspeicher. Hierbei wird der Speicherbedarf in Kilobyte (KiB) sowie die Zeit in Sekunden (s) gemessen. Außerdem die dimensionslosen Größen wie Knoten- und Kantenanzahl bzw. die Summe dieser beiden Größen als Anzahl der Graphelemente.

6.2.4.2 Messszenarien

Zum Vergleich werden insgesamt 9 verschiedene Szenarien durchgespielt. Beim ersten Szenario ist die Unifikation vollständig deaktiviert. Beim zweiten Szenario ist sie exklusiv für flache Knoten aktiviert, beim dritten exklusiv für strukturierte Knoten, usw. Das letzte Szenario beinhaltet die gleichzeitige Aktivierung der Unifikation für alle Annotationstypen, d. h. die vollständige Unifikation. Jedem Szenario ist eine eindeutige Kennung zugeordnet. Eine Übersicht aller Szenarien bietet Tabelle 6.6.

Tabelle 6.6: Übersicht Szenarien

Akronym	**Bedeutung**
NU	keine Unifikation
FUDO	vollständige Unifikation (alle Knoten, nur verzögert)
FUDI	vollständige Unifikation (alle Knoten, sofort und verzögert)
POUDO	exklusive verzögerte Unifikation flacher Knoten
POUDI	exklusive sofortige und verzögerte Unifikation flacher Knoten
SOUDO	exklusive verzögerte Unifikation strukturierter Knoten
AOUDO	exklusive verzögerte Unifikation von Aggregations-Knoten
AOUDI	exklusive sofortige und verzögerte Unifikation von Aggregations-Knoten
MOUDI	exklusive sofortige und verzögerte Unifikation von Multi-Knoten

6.2.4.3 Vorgehen und Randbedingungen

Die Messungen erfassen den Aufbau des Instanzgraphen. Das Einlesen der Daten aus der vorliegenden Textdatei und die Aufbereitung der Daten geschieht vorher. Um die Messung der Laufzeit und des Speicherbedarfs nicht zu verfälschen, kommt kein *Profiler* zum Einsatz. Stattdessen wird die Laufzeit anhand manuell erfasster Start- und Stoppzeiten ermittelt und der Speicherbedarf anhand der Prozessinformationen ermittelt.

Jede Messreihe enthält für jede Messgröße 100 Werte, für die sowohl der Durchschnittswert als auch die Standardabweichung berechnet wird, um die Qualität der Stichprobe bezüglich deren Streuung zu beurteilen und Messungen bei zu starken Ausreißern zu wiederholen. Eine Messung gilt als verwertbar, wenn die Standardabweichung der Stichprobe unter 5 Prozent liegt.

Die Messungen werden auf einer virtuellen Maschine durchgeführt, auf der Ubuntu 12.10 (GNU/Linux 3.5.0-17-generic x86_64) installiert ist. Diese ist mit insgesamt 5120 Mb Hauptspeicher sowie einem 64 Bit Intel Core 2 Duo Prozessor T9550 (6 Mb Cache, 2.66 GHz, 1066 MHz FSB) ausgestattet. Es sei angemerkt, dass aktuelle Prozessoren wesentlich schneller arbeiten. Die Messergebnisse haben also einen tendenziell anachronistischen Charakter.

Als virtuelle Maschine für die Ausführung des Ruby-Codes kommt YARV zum Einsatz, der zur Beschleunigung eine Bytecode-Zwischenrepräsentation nutzt. YARV bildet die Referenzimplementierung für die aktuelle Ruby-Version 2.0.0p0, die am 24. Februar 2013 veröffentlicht wurde. Sie ist derzeit wesentlich schneller als die Java-Implementierung *JRuby*, die eigentlich zum Einsatz kommen sollte.

Zwischen jeder Messreihe wird die virtuelle Maschine neu gestartet, um potentielle Störeffekte durch Cache-Inhalte zu vermeiden. Weiterhin ist die Systemauslastung sowohl auf dem Wirts- als auch auf dem Gastsystem minimal gehalten, um die Ausführung des Ruby-Codes nicht künstlich zu verlangsamen.

6.2.4.4 Einfluss der Unifikation

Ein sehr wichtiger Unterschied der beiden Unifikationsmodi im Kontext dieses Experiments ist die Feststellung, dass, sofern die sofortige Unifikation deaktiviert ist, immer die maximale Knotenanzahl von 6070 angelegt wird und erst im Zuge des verzögerten Unifikationsprozesses auf maximal 4040 reduziert wird. Dies ist insbesondere für die Interpretation der Ladezeiten wichtig. Außerdem gilt es zu bedenken, dass die

sofortige Unifikation im Gegensatz zur verzögerten Unifikation unvollständig sein kann, falls es äquivalente Ausprägungen gibt, die lediglich im Outdoor-Bereich existieren und noch nicht gespeichert wurden, also auch nicht über den Index des Attributraumes erfasst werden können. Aus diesem Grund gibt es auch keine Einzelmessungen, bei denen ausschließlich die sofortige Unifikation aktiv ist.

Um sicher zu gehen, dass redundante Knoten während bzw. nach der Unifikation auch tatsächlich aus dem *Heap-Speicher* entfernt werden, wird der *Ruby Garbage Collector* vor der Messung des Arbeitsspeicherbedarfs manuell gestartet. Anschließend wird genau geprüft, ob sich noch Knoteninstanzen im Heap-Bereich des Ruby-Prozesses befinden, die nicht Teil des Datenbankzustandes sind. Erst wenn die Anzahl der Instanzknoten im aktuellen Datenbankzustand und im Heap-Speicher übereinstimmten, gelten der Gesamtzustand und die ermittelten Messwerte als valide. Dies ist für alle hier aufgeführten Messergebnisse der Fall.

6.2.4.5 Durchführung und Diskussion

Eine ausführliche Übersicht über die Messergebnisse für die einzelnen Szenarien ist in C gegeben. In den folgenden Abschnitten werden diese Messergebnisse diskutiert.

Bevor mit der Analyse begonnen wird, soll an dieser Stelle ausdrücklich darauf hingewiesen werden, dass die Messergebnisse in direktem Zusammenhang mit der Beschaffenheit der Beispieldatensätze stehen. Dies bedeutet insbesondere, dass die getroffenen Aussagen keine Allgemeingültigkeit besitzen müssen. Insbesondere die Möglichkeiten zur Kompression hängen speziell von der Redundanz der zu speichernden Daten ab.

Weiterhin sei gesagt, dass der Zustandsaufbau *ohne* Integritätsprüfungen stattfindet, da diese zum Testzeitpunkt noch nicht vollständig implementiert sind. Dies hat einen bisher unbekannten Einfluss auf das Laufzeitverhalten.

6.2.4.6 Analyse Knoten- und Kantenanzahl

Grafik 6.2 zeigt die Messergebnisse für die Knoten- und Kantenanzahl der einzelnen Szenarien nach Aufbau eines Datenbankzustandes unter Verwendung der vorliegenden Beispieldaten, deren Beschaffenheit u. a. in 6.2 bzw. 6.3 dargestellt ist.

Für die Diskussion wird davon ausgegangen, dass eine geringe Anzahl von Elementen als positiv bewertet werden kann, da sie unmittelbar zu geringerem Speicheraufwand führt.

Die Anzahl der Knoten bewegt sich insgesamt zwischen 4040 und 6070; die Anzahl der Kanten zwischen 5299 und 7335, je nachdem, welcher Kompressionsfaktor durch die Unifikation erreicht wird. Die Szenarien sind nach abnehmender Elementeanzahl sortiert.

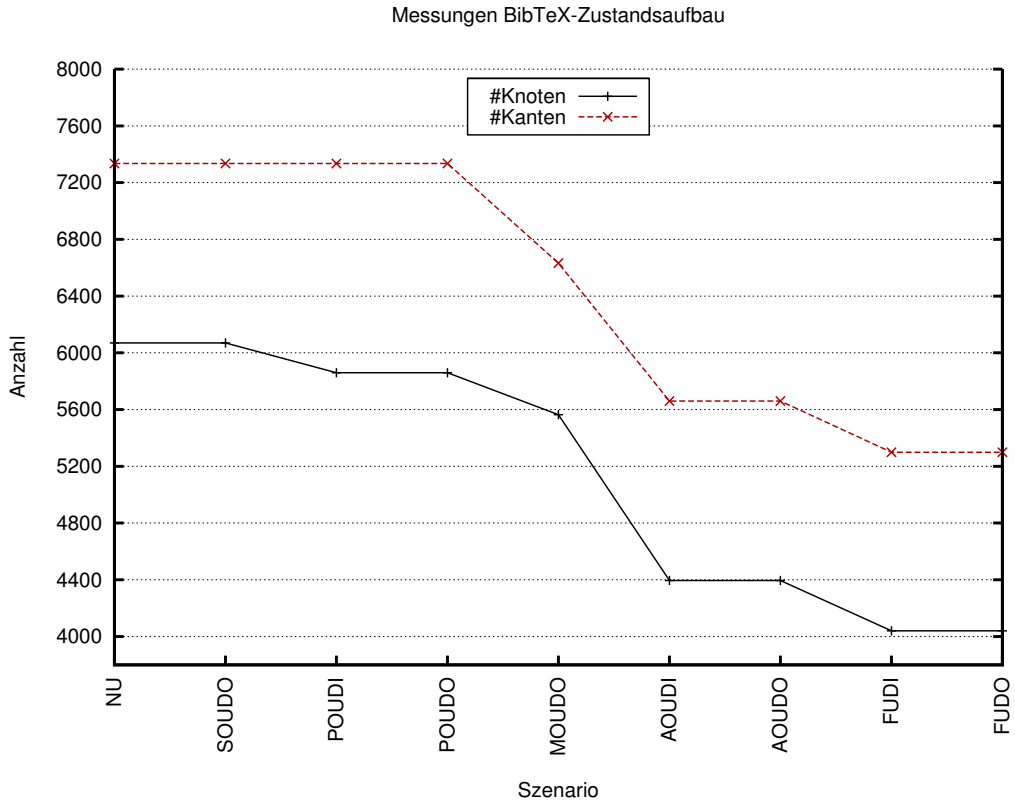

Abbildung 6.2: Messergebnisse Knoten- und Kantenanzahl der einzelnen Szenarien

Das erste Szenario (NU) verzichtet vollständig auf Unifikationsmaßnahmen, weshalb der Datenbankzustand das Maximum an Knoten- und Kantenelementen beinhaltet.

Von links nach rechts der x-Achse folgend kann für das zweite Szenario (SOUDO) das gleiche Ergebnis beobachtet werden. Dies hängt damit zusammen, dass die Unifikation des strukturierten Typs *Name* aufgrund fehlender Gleichnamigkeit von Autoren keinen Effekt besitzt. Weitere strukturierte Typen sind nicht definiert.

Das dritte Szenario (POUDI) beinhaltet eine Reduzierung der Knotenanzahl auf 5860. Dies kann damit erklärt werden, dass einige Autoren gleiche Vor- bzw.

Nachnamen besitzen, die als flache Attribute gespeichert werden. Weiterhin existieren Ausprägungen des Annotationstyps *Year*, die mehrfach verwendet werden. Aufgrund der Tatsache, dass bei dieser Art der Unifikation keine größeren Substrukturen mehrfach referenziert werden, bleibt die Kantenanzahl weiterhin bei 7335. Das Szenario (POUDO) unterscheidet sich lediglich in dem Zeitpunkt der Unifikation und prägt einen äquivalenten Zustand aus.

Bei dem folgenden Szenario (MOUDO) ist eine deutliche Reduzierung der Knoten- sowie Kantenanzahl zu erkennen. Die Unifikation von Multi-Knoten betrifft zunächst einmal die multiplen flachen Unterattribute *FirstName* und *LastName*. Den größten Effekt allerdings haben die multiplen Aggregations-Knoten der Typen *@Keyword* und *@Author*, da sie die n:m-Beziehung für die Beziehungstypen *PubKeyword* und *PubAuthor* realisieren. Innerhalb dieser Beziehungen stehen unterschiedliche Instanzen des Typs *Pub* mit derselben Menge von Teilnehmern der jeweils anderen Seite in Beziehung. Die Unifikation von multiplen Knoten reduziert die Knotenmenge auf 5564 und die Anzahl der Kanten auf 6632, da weniger Kollektionskanten für die Mitglieder von Multi-Knoten verwendet werden müssen.

Dieser Effekt macht sich auch in den Folgeszenarien (AOUDI) sowie (AOUDO) bemerkbar, denn bisher wurden für jede Beziehung pro Entität mehrere Aggregations-Knoten ausgeprägt. Dies wird durch die Unifikation verhindert und ermöglicht so eine Reduzierung auf 4395 Knoten bzw. 5560 Kanten, da Aggregationskanten gespart werden können.

Die letzten beiden Szenarien (FUDI) und (FUDO) verwenden die vollständige Unifikation zu unterschiedlichen Zeitpunkten mit demselben Ergebnis. Die Anzahl der Knoten kann in diesem Fall auf 4040 und die Anzahl der Kanten auf 5299 reduziert werden. Der Kompressionsfaktor liegt damit für die Knoten bei ca. 33 Prozent und für die Kanten bei ca. 28 Prozent.

6.2.4.7 Analyse Zeit- und Platzaufwand

Grafik 6.3 zeigt den Zeit- sowie Platzaufwand für die einzelnen Szenarien. Der Zeitaufwand bezieht sich dabei auf die Ladezeit des Beispielzustandes und der Platzaufwand auf den Arbeitsspeicherbedarf. Die Szenarien sind anhand ihrer Messwerte für die Ladezeit aufsteigend sortiert.

Der Speicherbedarf wird auf der ersten y-Achse (links) aufgetragen und als schwarze Kurve gezeichnet. Die Ladezeit wird auf der zweiten y-Achse (rechts) aufgetragen und als rote gestrichelte Kurve dargestellt.

Die Werte für den Arbeitsspeicherbedarf bewegen sich in einem Intervall zwischen 9397,12 und 7901,52 Mb. Die Ladezeiten unterscheiden sich in einem Bereich von ca. zwei Zehntelsekunden und liegen in einem Intervall zwischen 1,1589 und 1,2910 Sekunden. Aufgrund des geringen Abstandes zwischen diesen Werten müssen Vergleiche vorsichtig angestellt werden.

Anders als bei der vorangegangenen Diskussion spielt die Unterscheidung zwischen sofortiger und verzögerter Unifikation eine signifikante Rolle in Bezug auf die Ladezeiten.

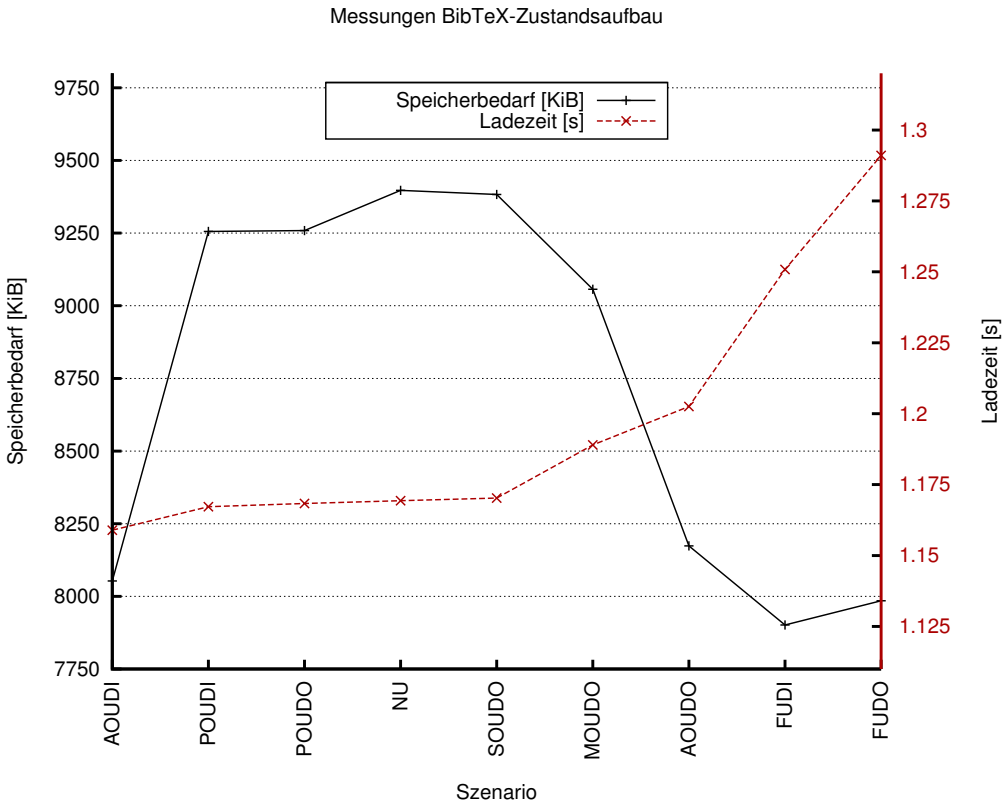

Abbildung 6.3: Messergebnisse Zeit- und Platzaufwand der einzelnen Szenarien

Die geringste Ladezeit mit 1,1589 s weist das erste Szenario (AOUDI) auf. In diesem Szenario werden Aggregations-Knoten sofort und verzögert unifiziert. Hier ist ein deutlicher Unterschied zu dem verwandten Szenario (AOUDO) mit 1,2025 s erkennbar, bei dem Aggregations-Knoten lediglich verzögert unifiziert werden, d. h. zunächst das

Maximum an Knoten ausgebildet wird. Um diesen Zusammenhang zu erklären, muss auf die Speicherreihenfolge beim Laden der Testdaten eingegangen werden.

Bei dem vorliegenden Experiment werden Entitäten angelegt und *unmittelbar* gespeichert. Dies begünstigt die sofortige Unifikation, da potentiell äquivalente Instanzen direkt in den Attributindex aufgenommen werden. Bei verzögerter Überführung der Daten in den Indoor-Bereich findet eine Indizierung zu einem späteren Zeitpunkt statt, sodass äquivalente Knoten u. U. mehrfach angelegt werden. Dieser Umstand wird als Erklärung angesehen, warum sich die Ladezeit reduziert, wenn bei unmittelbarer Speicherung der Daten die sofortige Unifikation aktiv ist. In diesem Fall werden keine äquivalente Knoten redundant ausgeprägt und es entfällt die Instanziierung entsprechender Ruby-Objekte.

Im zweiten Szenario (POUDI) werden ausschließlich flache Knoten sofort und verzögert unifiziert, was zu einem erheblichen Anstieg des Speicherbedarfs auf 9255,76 Mb führt, da nur wenige Kandidaten für eine Mehrfachreferenzierung existieren, wie bereits erläutert wurde. Der erhöhte Zeitaufwand von 1,1672 s erklärt sich damit, dass für alle übrigen Knotentypen die maximale Anzahl ausgeprägt werden muss, d. h. der Rechenaufwand für die Instanziierung entsprechender Ruby-Objekte steigt. Selbiges kann über das verwandte Folgeszenario (PUODO) ausgesagt werden.

Das vierte Szenario (NU) unterlässt die Unifikation vollständig und prägt die maximale Anzahl der Knoten aus. Aus diesem Grund erhöht sich in diesem Fall die Ladezeit auf 1,1693 s und erreicht den maximalen Speicherbedarf von 9397,12 Mb. Da beim folgenden Szenario (SOUDO) nahezu keine Kandidaten für eine Unifikation existieren, ähneln sich die Messwerte stark mit 1,1703 s und 9383,00 Mb.

Das Szenario (MOUDO) ist mit 1,1890 s der Beginn signifikanter Änderungen beim Zeitaufwand. In diesem Szenario werden Multi-Knoten lediglich verzögert unifiziert, da eine sofortiger Abgleich nicht möglich ist. In diesem Fall wird also das Maximum an redundanten Multi-Knoten ausgeprägt und erst beim Aufrufen der Speicherroutine unifiziert. Dies bedeutet einerseits einen Mehraufwand bei der Erzeugung von Ruby-Objekten und andererseits mehr Rechenarbeit bei der Überführung in den unifizierten Zustand, da eine wesentliche Anzahl von Knoten umgehängt bzw. gelöscht werden muss. Dieses Phänomen kann ebenfalls als Ursache für die Erhöhung der Ladezeit auf 1,2025 beim Folgeszenario (AOUDO) betrachtet werden.

Das vorletzte Szenario (FUDI) unifiziert alle Knotentypen sofort, falls möglich, und verzögert. Entgegen der Erwartung erhöht sich die Ladezeit mit 1,2508 s deutlich. Eine mögliche Erklärung ist, dass in diesem Fall sehr viele Indexanfragen aufgrund

der erhöhten Unifikationsversuche stattfinden (siehe C) und diese die Vorteile der verringerten Anzahl von Instanziierung aufheben.

Die nochmals signifikant erhöhte Ladezeit im letzten Szenario (FUDO) kann mit dem Restrukturierungsaufwand begründet werden, der bei verzögerter Unifikation auftritt. Mit 1,2910 s dauert der Zustandsaufbau in diesem Fall am längsten.

Bisher wurde hauptsächlich der Zeitaufwand betrachtet. Wie im letzten Abschnitt bereits erläutert, führt die Unifikation u. U. zu einer wesentlichen Verringerung des Speicherbedarfs. Diesbezüglich soll an dieser Stelle noch einmal auf die Szenario-Paare (POUDI) und (POUDO) bzw. (FUDI) und (FUDO) eingegangen werden.

In beiden Fällen müssen die Werte für den Speicherbedarf übereinstimmen, da der endgültig gemessene Platzaufwand unabhängig von der Frage ist, wann die Unifikation stattfindet. Dies ist jedoch nicht der Fall. Da sichergestellt ist, dass sich nach der Unifikation keine redundanten Knoten mehr im Heap-Speicher des Ruby-Prozesses befinden, kann die Vermutung angestellt werden, dass die Ausbildung redundanter Knoten im Outdoor-Bereich bei ausschließlicher Verwendung der verzögerten Unifikation zusätzlichen Speicherbedarf verursacht. Die Ursache dafür ist nicht bekannt.

Als wichtiges Ergebnis dieser Betrachtung kann die Hypothese aufgestellt werden, dass die Unifikation und damit die Komprimierung von Datenbeständen negative Auswirkungen auf das Laufzeitverhalten hat. Dieser Effekt wird noch verstärkt, wenn die Unifikation verzögert stattfindet. Eine Bestätigung bzw. Widerlegung dieser Vermutung müssen weitere Experimente zeigen.

7 Zusammenfassung, Fazit und Ausblick

7.1 Zusammenfassung

Das Motivationskapitel (2) geht zu Beginn auf die Bedeutung der Anforderungserhebung für die Entwicklung von Informationssystemen ein (2.1). Es wird verdeutlicht, dass das konzeptuelle Modell eine wichtige Rolle spielt, da es aufgrund der direkten semantischen Abbildung der Anwendungsdomäne leicht zu verstehen ist und dadurch z. B. als Basis für die Kommunikation zwischen Entwicklern und Kunden eingesetzt werden kann.

Anschließend wird auf die Probleme der relationalen Modellierung eingegangen (2.2), die insbesondere durch die Übersetzung des konzeptuellen Modells in Relationenschemata verursacht werden. Zudem wird auf die Schwierigkeiten der Behandlung von relationalen Datenbeständen unter Verwendung objektorientierter Fachlogik eingegangen.

Am Ende des Kapitels (2.3) wird das Ziel formuliert, ein integriertes semantisches Datenbankmodell zu entwerfen, das die Entwicklung von Informationssystemen auf Ebene der Datenhaltungsschicht einfacher gestalten und beschleunigen soll.

Im Kapitel 3 wird zunächst auf die grundlegende Bedeutung von Datenmodellen für die Entwicklung von Informationssystemen eingegangen (3.1). Anschließend wird der Prozess der Datenmodellierung beschrieben, wie er derzeit üblich ist (3.2). Es folgt eine Übersicht über grundlegende Modellierungselemente, mit denen die Sachverhalte einer Anwendungsdomäne beschrieben werden können (3.3).

Als Grundlage für den Entwurf dienen einerseits das ER-Modell sowie das UML-Klassendiagramm, die in Abschnitt 3.4.1 bzw. 3.4.2 betrachtet werden und andererseits der objektorientierte bzw. graphbasierte Ansatz für logische Modelle (3.5).

Das zentrale Entwurfskapitel (4) erläutert eingangs den Einsatz des zu konzipierenden Modells innerhalb der Datenhaltungsebene einer klassischen Schichtenarchitektur (4.1). Anschließend geht es auf Strategien zur Datenspeicherung ein und hebt an dieser Stelle die Vorteile einer strukturierten Speicherung hervor (4.2). Es folgen Erläuterungen zum

Thema Modellsichten (4.3), bevor allgemein die Möglichkeiten zur Darstellung von Graphen im Hauptspeicher eines Computers diskutiert werden (4.4).

Nach Betrachtungen zur genotypischen bzw. phänotypischen Generalisierung (4.5) wird der Aufbau von Typgraphen besprochen, die als semantische Datenbankmodelle eingesetzt werden können (4.6). Die einzelnen Beispiele erläutern, wie die in Abschnitt 3.3 aufgeführten Modellierungselemente repräsentiert werden können.

Der Typgraph kann als Schema für die Ausprägung von Datenbankzuständen verwendet werden. Der Aufbau von Instanzgraphen wird in Abschnitt 4.7 detailliert beschrieben, indem die Abbildung zwischen Typ- und Instanzknoten erläutert (4.7.2) und ein ausführliches Beispiel vorgestellt wird (4.7.3). Die Komprimierung von Datenbeständen mittels Wiederverwendung von Instanzknoten wird als Unifikation bezeichnet und in Abschnitt (4.7.4) formal erläutert.

Abschnitt 4.7.5 handelt von der Ausprägung eines Datenbankzustandes und geht der Vorstellung eines Konzeptes zur virtuellen Trennung des Hauptspeichers in Outdoor- und Indoor-Bereiche voraus (4.7.6). Das Kapitel schließt mit der Erläuterung zweier fundamentaler Architekturansätze für Datenbanksysteme als Stand-Alone- bzw. Client-Server-Software (4.8).

Das Implementierungskapitel (5) beschreibt zunächst allgemein den Entwicklungs-Prozess (5.1.1) und die Entwicklungsumgebung (5.1.2), bevor es auf die Struktur der Referenzimplementierung eingeht (5.2). In Abschnitt 5.3 sowie 5.4 werden Implementierungsdetails über Typ- und Instanzgraph in Form von UML-Klassendiagrammen gegeben. Abschnitt 5.5 erläutert einen Lösungsansatz zur Verhinderung inkonsistenter Zustände bei Manipulation von strukturierten Daten, die mehrfach referenziert werden.

Es folgt Abschnitt 5.6 mit Erläuterungen über Zuweisung und Ausprägung von Attributwerten anhand eines Kontrollflussdiagramms, das die Arbeit der *Set*-Operation darstellt. Anschließend wird beschrieben, wie mittels des sogenannten *Probing*-Verfahrens innerhalb von Instanzgraphen navigiert werden kann (5.7).

Kapitel 6 beinhaltet ein theoretisches Experiment, das sich mit möglichen Kompressionsraten unifizierter Zustände auseinandersetzt (6.1) und ein praktisches Experiment, das durchgeführte Messungen zu Zeit- und Platzaufwand beim Aufbau eines größeren Beispielzustandes diskutiert (6.2).

Die Arbeit schließt mit Kapitel 7, das neben dieser Zusammenfassung ein Fazit (7.2) sowie einen Ausblick (7.3) beinhaltet.

7.2 Fazit

Der Autor zieht insgesamt ein positives Fazit aus der Diplomarbeit. Der vorgestellte Entwurf eines Typgraphen kann einerseits als konzeptuelles und andererseits als logisches bzw. physisches Datenbankmodell eingesetzt werden. Dies vereinfacht den Prozess der Datenmodellierung, da keine zusätzlichen Modelle erstellt und Aktualisierungen nicht synchronisiert werden müssen.

Der Typgraph basiert auf den Sprachmitteln des ER-Modells bzw. des UML-Klassendiagramms und führt weitere Konzepte aus dem Bereich der Objektorientierung ein. Sein Entwurf begegnet dem Problem der genotypischen bzw. phänotypischen Vererbung mittels Klassifikation und bietet außerdem die Möglichkeit zur logischen Formulierung von Existenzbedingungen über Einschränkungsgruppen. Die Einführung polymorpher Aggregation erlaubt sehr flexible Definitionen von Beziehungen zu anderen Entitäten. Entitäten können dabei direkt Teil anderer Entitäten sein oder als Bestandteil von strukturierten Daten aggregiert werden.

Jedes Element eines Typgraphen ist eindeutig benannt. Dies gilt auch für die Annotationstypen, die mehrfach verwendet werden können. Der Autor denkt, dass durch die eindeutige Benennung von Attributen die semantische Konsistenz verbessert werden kann. Im Allgemeinen fördert die direkte Darstellung von Generalisierung, strukturierten Typen, Schnittstellen, Aggregationsbeziehungen, etc. die Semantik des Modells und vereinfacht so die Abbildung der Anwendungsdomäne.

Die übersichtliche Darstellung von Diagrammen stellt allerdings eine Herausforderung dar. Dies ist insbesondere der Fall, wenn eine hohe Beziehungsdichte der modellierten Artefakte vorliegt oder ausführliche Integritätsbedingungen direkt an die Knoten notiert werden. Hier kann das Konzept der Sichten durchaus helfen. Sichten können in einem Diagrammeditor beispielsweise mittels Ebenen bzw. Attributfilter realisiert werden. An dieser Stelle sei ausdrücklich auf den Grapheditor *yEd* hingewiesen, der entsprechend konfiguriert werden kann.

Für die konkrete Speicherung von Ausprägungen wurde in dieser Arbeit ein Graphspeicher beschrieben und implementiert. Dieser nutzt den Typgraph als logisches Schema, weshalb dieser genauer als integriertes semantisches Datenbankmodell bezeichnet werden kann. Der Aufbau von Instanzgraphen erfolgt direkt am Schema durch Belegung der Annotationstypen mit konkreten Werten. Dieses Vorgehen wurde in dieser Arbeit eingehend beschrieben.

Außerdem wurde auf die Möglichkeit zur Komprimierung von Datenbankzuständen eingegangen, die als Unifikation bezeichnet wird. Der Autor erlaubt sich aufgrund der Ergebnisse von weiteren Experimenten, die in dieser Arbeit nicht aufgeführt wurden, die Vermutung, dass die unifizierte Darstellung generell zu wesentlich geringerem Speicherbedarf führt. Ein weiterer Vorteil der Unifikation ist der direkte Zugriff auf alle Entitäten, die ein entsprechendes Attribut mit dem gleichen Wert ausprägen.

Als sehr wichtig sind die Ergebnisse des BibTeX-Experiments zu beurteilen. Sie geben neben der theoretischen Betrachtung Anhaltspunkte über das tatsächliche Laufzeitverhalten der Referenzimplementierung und gestatten einen Blick auf Zeit- und Speicheraufwand. Insbesondere der Speicherbedarf ist ein kritisches Moment, da der Graphspeicher primär als In-Memory-Datenbank konzipiert wird. Das BibTeX-Experiment zeigt, dass der Beispielzustand bei vollständiger Unifikation (Szenario FUDI) 7902 Kb Hauptspeicher belegt.

Zu diesem Beispielzustand gehören sowohl die Knoten als auch die Kanten. Letztere werden über Speicherreferenzen dargestellt und belegen pro Ausprägung auf einem 64 Bit System 8 Byte. Bei insgesamt 5299 Kanten ergibt sich ein Speicherbedarf von 41 Kb. Es verbleibt demnach 7861 Kb Speicherbedarf für die insgesamt 4040 Knoten. Ein Knoten belegt damit im Schnitt 1,95 Kb Speicher.

Dieser Wert kann als Grundlage für die Prognose von potentiell benötigten Hauptspeichergrößen verwendet werden. Hierbei ist es wichtig zu berücksichtigen, dass dieser Wert je nach Beschaffenheit der zu speichernden Daten stark schwanken kann. Eine Prognose ist also nur belastbar, wenn der Durchschnittswert für die Speicherbelegung eines Knotens anhand eines realistischen Beispielzustandes ermittelt wurde. Es gilt nun die interessante Frage zu klären, wie viele Knoten man bei einer bestimmten Speicherausstattung ausprägen könnte.

Die aktuelle Computertechnik erlaubt derzeit den Einsatz von Systemen mit 512 Gigabyte RAM und mehr[1]. Bei dieser Ausstattung und unter Verwendung eines Durchschnittswertes von 1,95 Kb pro Knoten wäre theoretisch die Speicherung von mehr als 275 Millionen Knoten möglich. Die Auslagerung von Datenbeständen auf Festspeicher bietet außerdem eine Möglichkeit zur zusätzlichen Erweiterung des Speicherplatzes.

Aufgrund der genannten Punkte und dem in dieser Arbeit insgesamt erlangtem Wissen ist der Autor der Meinung, dass der vorgestellte Entwurf eines semantischen Datenbankmodells dabei helfen kann, den Prozess der Datenmodellierung effizienter

[1] Am Beispiel IBM BladeCenter HX5. Online im Internet: URL `http://www-03.ibm.com/systems/bladecenter/hardware/servers/hx5/index.html` (Stand 22.04.2013)

zu gestalten. In Kombination mit einem leistungsfähigen Graphspeicher können Informationssysteme so schneller entwickelt werden.

Der Autor ist außerdem der Überzeugung, dass der Einsatz von relationalen System in vielen Bereichen überdacht werden muss. Er ist sich sicher, dass graphbasierte Ansätze zukünftige Entwicklungen im Datenbankbereich noch stärker prägen werden, als es bisher der Fall war.

7.3 Ausblick

Die nächsten Absätze gehen darauf ein, welche Möglichkeiten der vorgestellte Entwurf und die Referenzimplementierung für zukünftige Entwicklungen bieten. Die Ausführungen sind nicht erschöpfend, sondern beinhalten die Themen, die seitens des Autors als besonders wichtig erachtet werden.

Die Ausarbeitung von Datenmodellen erfolgt mittels Typgraphen, die entsprechende Sprachmittel zur Modellierung der Anwendungsdomäne zur Verfügung stellen. Eines dieser Sprachmittel stellt die Einschränkungsgruppe dar, die bisher lediglich die Formulierung einfacher logischer Bedingungen für die Existenz von nachfolgenden Knoten zulässt. Dieses Konzept sollte dahingehend erweitert werden, dass auch Bedingungen unter Verwendung von Vergleichsoperatoren ausgedrückt werden können. Hierbei muss sowohl auf die Reihenfolge der zu vergleichenden Knoten als auch auf die Sicherstellung der Typkompatibilität eingegangen werden.

Die übersichtliche Darstellung von Modellen erhöht deren Verständlichkeit. Dies ist insbesondere für die Anforderungserhebung wichtig. Die Einführung von Sichten auf Typgraphen wurde im Entwurfskapitel angesprochen, allerdings nicht erschöpfend behandelt. Eine weitere Bearbeitung dieses Themas kann einerseits die genaue Spezifikation von Sichten beinhalten und sich andererseits mit der Qualität von verschiedenen Layouts auseinandersetzen. Ein weiterer Punkt betrifft die praktische Anfertigung von Modellen mit Hilfe von Diagrammeditoren. In diesem Zusammenhang kann beispielsweise die Einsatzfähigkeit von *yEd* untersucht werden, der bereits Möglichkeiten zum Einführen von Sichten bzw. automatische Layout-Hilfen anbietet.

Die eingeführte Sprache zur Beschreibung von Domänen beinhaltet bisher keine Möglichkeit zur Festlegung von Zugriffsrechten. Eine interessante Perspektive stellt die Einführung einer rollenbasierten Zugriffskontrolle dar (engl., Sg.: *role base access control, RBAC*), die direkt mit Hilfe von zusätzlichen Knoten- bzw. Kantendekorationen modelliert werden kann. Informationen über Rollen und deren erlaubte Operationen

können beispielsweise auf den Mitgliedschaftskanten hinterlegt werden, was eine sehr feine Modellierung von Zugriffsrechten bis auf Ebene einzelner Attribute zulässt.

Während die Einführung eines Rechtesystems dem Anspruch auf Datensicherheit gerecht wird, bedarf es außerdem Überlegungen zur Einführung eines Mehrbenutzerbetriebs. An dieser Stelle muss überlegt werden, wie Mechanismen zur Synchronisation etabliert werden können. In diesen Bereich fällt auch das Thema Transaktionen, die die Konsistenz bei nicht atomaren Operationen gewährleisten sollen. Für die technische Umsetzung sollte außerdem die Fähigkeit der eingesetzten Programmiersprache Ruby in Bezug auf Multithreading untersucht werden.

Die gezielte Abfrage von Daten unter Angabe von Suchkriterien stellt einen weiteren wichtigen Punkt dar. Derzeit bietet die Implementierung die Möglichkeit zur direkten Traversierung von Datenbankzuständen, bei der Instanzknoten auf entsprechende Wertebelegung geprüft werden können. Es existiert jedoch noch keine definierte Abfragesprache, mit der beispielsweise Bedingungen für Wertebelegungen formuliert werden können.

In diesen Bereich fällt die Konstruktion von Suchgraphen, die auf Basis entsprechender Kriterien definiert werden. Für diese Suchgraphen können dann Hash-Werte ermittelt werden [LWfC11], nach denen beispielsweise in einem Index gesucht werden kann. Weiterhin kann die Suche nach speziellen Knotenstrukturen auf das Problem der Suche nach Teil- bzw. Untergraphen abgebildet werden. In diesem Zusammenhang kann die Auseinandersetzung mit den Arbeiten [SS95] und [Ull76] in Erwägung gezogen werden.

Der Entwurf in dieser Arbeit beschreibt keine Ansätze für die Konstruktion der Permanenz-Schicht, mit der Datenbankzustände auf Festspeicher geschrieben werden sollen. Die konkrete Umsetzung einer entsprechenden Komponente ist allerdings wichtig, damit Datenbankzustände zwischen Systemstarts erhalten bleiben und die Möglichkeit zur Auslagerung von Datenbeständen gegeben wird.

In diesem Zusammenhang kann außerdem untersucht werden, inwiefern die bisher erreichten Kompressionsraten erhöht werden können. Hierbei gibt es zwei Ansätze, die erläutert werden sollen. Zum einen können Knotenwerte an sich kompakter dargestellt werden, in dem sie einer Datenkompression beispielsweise mittels *gzip* unterworfen werden. Zum anderen kann der Ansatz der Unifikation verallgemeinert und zu einer Konstantenunifikation ausgebaut werden. Diese Erweiterung würde Knoten unabhängig von ihrem Typ unifizieren, d. h. nur die Äquivalenz von Werten betrachten.

Eine weitere Untersuchungsmöglichkeit resultiert aus dem gewählten Ansatz einer In-Memory-Datenbank. Da Datenbestände beträchtliche Größen annehmen

können, wäre eine Untersuchung interessant, die feststellt, wie die verwendete Implementierungssprache Ruby auf Allokation von mehreren Giga- bzw. Terabyte Hauptspeicher reagiert. In diesem Zusammenhang wäre die Konstruktion entsprechender Tests und die Durchführung von Messungen sinnvoll, wie sie z. B. im Evaluationskapitel zu finden sind. Außerdem sollte die Auslagerung von Datenbeständen auf schnelle SSD-Festspeicher untersucht werden.

Anhang

A Syntaxelemente des Typgraphen

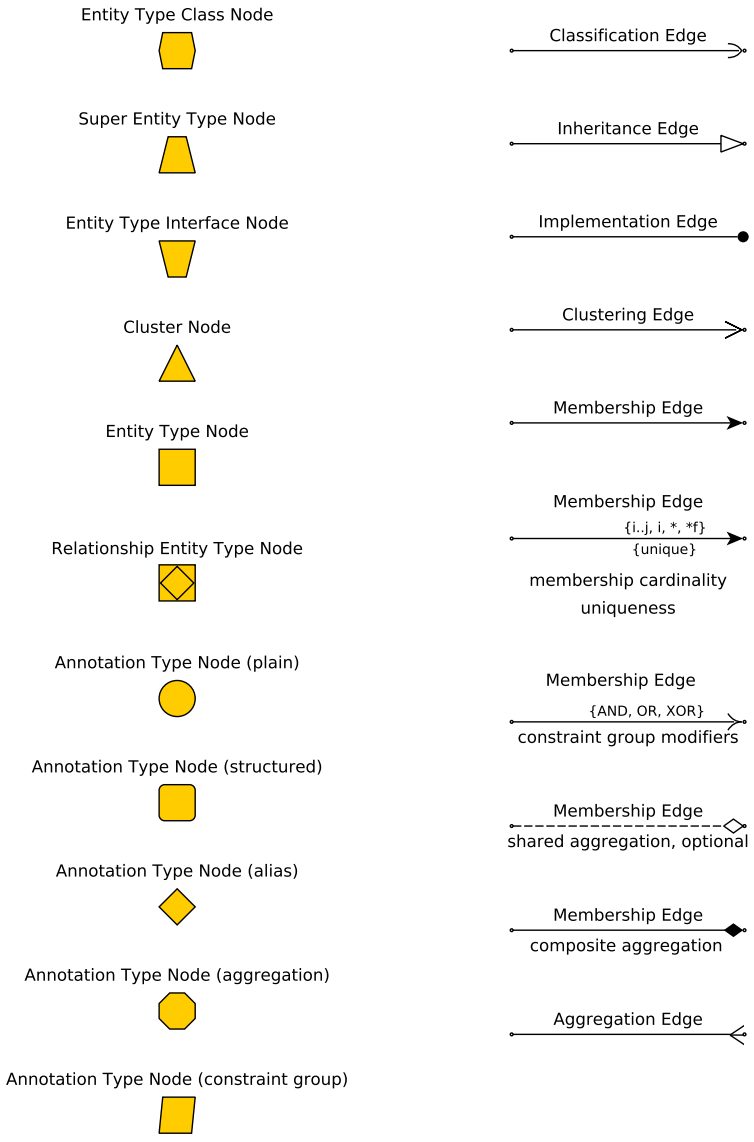

Abbildung A.1: Typgraph Syntaxelemente

B UML-Klassendiagramme Schema-Komponente

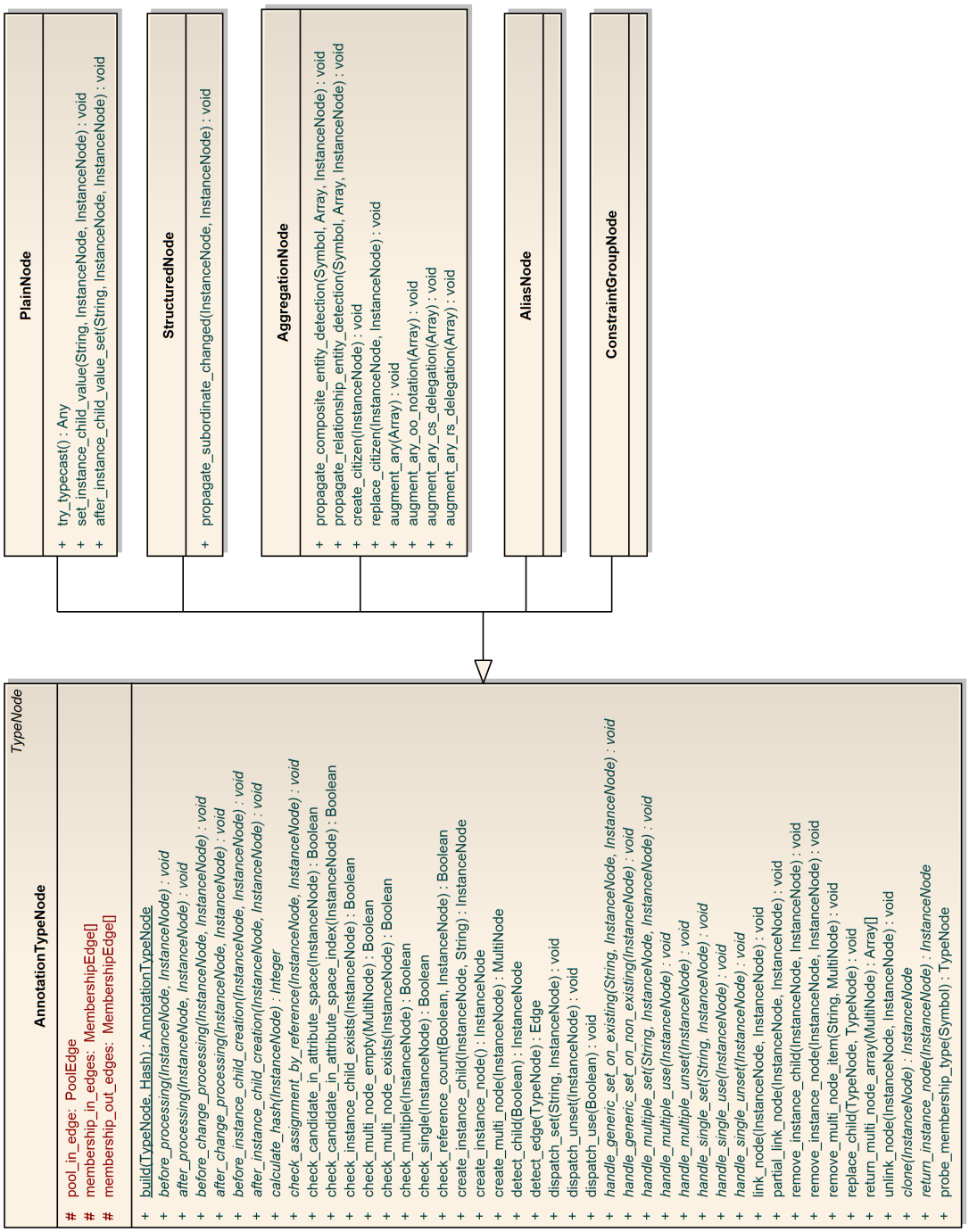

Abbildung A.2: 3. UML-Klassendiagramm

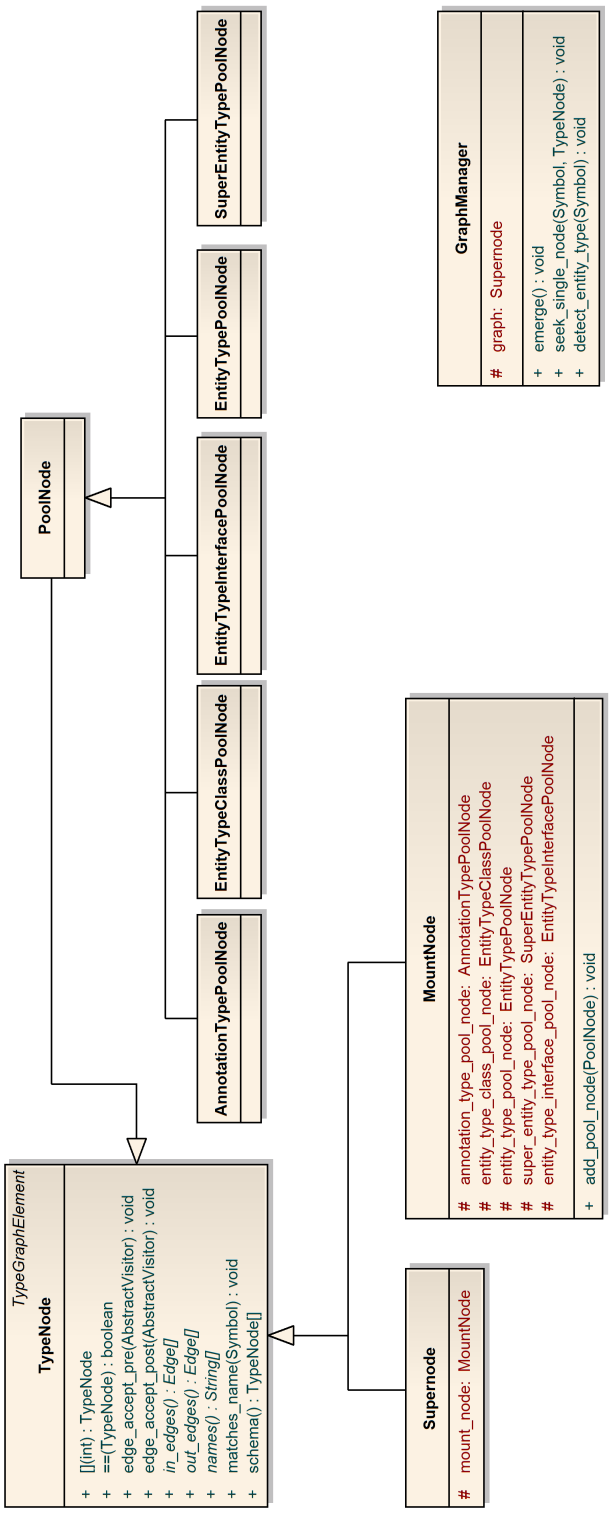

Abbildung A.3: 4. UML-Klassendiagramm

C Messergebnisse zu BibTeX-Experiment

No.	Scenario	# Nodes	# Edges	# Elements	Average Time [s]	StDev Time (abs.) [s]	StDev Time (%)	Average Space [kb]	StDev Space (abs.) [kb]	StDev Space (%)	# index-requests
1	AOUDI	4395	5660	10055	1,1589	0,0411	3,5492%	8052,8000	240,4286	2,9857%	5046
2	POUDI	5860	7335	13195	1,1672	0,0301	2,5825%	9255,7600	331,6587	3,5833%	2846
3	POUDO	5860	7335	13195	1,1684	0,0342	2,9307%	9258,6800	263,7943	2,8492%	1423
4	NU	6070	7335	13405	1,1693	0,0320	2,7364%	9397,1200	111,1222	1,1825%	0
5	SOUDO	6070	7335	13405	1,1703	0,0325	2,7805%	9383,0000	102,6208	1,0937%	152
6	MOUDO	5564	6632	12196	1,1890	0,0418	3,5186%	9056,6000	80,6126	0,8901%	714
7	AOUDO	4395	5660	10055	1,2025	0,0422	3,5108%	8173,8800	331,8447	4,0598%	2523
8	FUDI	4040	5299	9339	1,2508	0,0457	3,6556%	7901,5200	238,1435	3,0139%	8397
9	FUDO	4040	5299	9339	1,2910	0,0605	4,6862%	7984,7600	230,4909	2,8866%	4451

Abbildung A.4: Messergebnisse

Literaturverzeichnis

[AH84] Serge Abiteboul and Richard Hull. Ifo: a formal semantic database model. In *Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '84, pages 119–132, New York, NY, USA, 1984. ACM. 12

[Ang12] R. Angles. A Comparison of Current Graph Database Models. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 171–177. IEEE, April 2012. 30

[Bal99] Heide Balzert. *Lehrbuch der Objektmodellierung: Analyse und Entwurf.* Spektrum Akademischer Verlag GmbH, Heidelberg · Berlin, 1999. 31

[BF02] Rolf Böhm and Emmerich Fuchs. *System-Entwicklung in der Wirtschaftsinformatik.* vdf, Hochsch.-Verl. an der ETH, Zürich, 5 edition, 2002. 8

[Cod80] E. F. Codd. Data models in database management. *SIGMOD Rec.*, 11(2):112–114, June 1980. 12

[Fow03] Martin Fowler. *Patterns of Enterprise Application Architecture.* Pearson Education, Inc., Boston, 2003. 12

[Gei11] Frank Geisler. *Datenbanken: Grundlagen und Design.* mitp-Verlag, Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg, 4 edition, 2011. 14

[GHJV04] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Entwurfsmuster.* Addison-Wesley-Verlag, München, 2004. 130, 132, 134

[GJLS08] Dorina Gumm, Monique Janneck, Roman Langer, and Edouard J. Simon. *Mensch - Technik - Ärger? Zur Beherrschbarkeit soziotechnischer Dynamik aus transdisziplinärer Sicht.* Lit Verlag Dr. W. Hopf Berlin, Münster, 2008. 1

[Gog12] Martin Gogolla. USE BibTeX-Example. Technical report, 2012. 88, 156, 168

[GP03] Rainer Unland Günther Pernul. *Datenbanken im Unternehmen: Analyse, Modellbildung und Einsatz.* Oldenbourg Wissenschaftsverlag GmbH, München, 2 edition, 2003. 14

[HM06] Kim Hamilton and Russel Miles. *Learning UML 2.0.* O'Reilly, München, 2006. 14

[LWfC11] Wei Liu, Jun Wang, and Shih fu Chang. Hashing with graphs. In *ICML*, 2011. 188

[LWS08] Franz Lehner, Stephan Wildner, and Michael Scholz. *Wirtschaftsinformatik: Eine Einführung.* Carl Hanser Verlag München Wien, München · Wien, 2 edition, 2008. 2

[Mit11] Hans Joachim Mittag. *Statistik: Eine interaktive Einführung.* Springer-Verlag GmbH & Co. KG, Heidelberg, 2011. 103

[MS05] Neil Matthew and Richard Stones. *Beginning Databases with PostgreSQL: From Novice to Professional.* Springer-Verlag GmbH & Co. KG, Heidelberg, 2 edition, 2005. 33

[NH00] Simon North and Paul Hermans. *XML in 21 Tagen.* Markt+Technik Verlag, München, 2000. 32

[NR12] Werner Neubauer and Bernd Rudow. *Trends in der Automobilindustrie.* Oldenburg Wissenschaftsverlag GmbH, München, 2012. 1

[PJ00] Meilir Page-Jones. *Fundamentals Of Object-Oriented Design in UML.* Dorset House Publishing, New York, 2000. 19

[Pon07] Paulraj Ponniah. *Data modeling fundamentals: a practical guide for IT professionals.* John Wiley & Sons, Inc., New Jersey, 2007. 14

[Pre10] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, New York, 7 edition, 2010. 8

[sC76] Peter Pin shan Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1:9–36, 1976. 23

[Som07] Ian Sommerville. *Software Engineering.* Pearson Education Limited, Essex, 8 edition, 2007. 14

[SS95] Gopalakrishnan Sundaram and Steven S. Skiena. Recognizing small subgraphs. *Networks*, 25:183–191, 1995. 188

[SSH08] Gunter Saake, Kai-Uwe Sattler, and Andreas Heuer. *Datenbanken: Konzepte und Sprachen.* mitp-Verlag, Redline GmbH, Heidelberg, 3 edition, 2008. 14

[Stö05] Harald Störrle. *UML 2 erfolgreich einsetzen.* Addison-Wesley-Verlag, München, 2005. 14

[Ull76] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976. 188

[Wal01] Ernest Wallmüller. *Software-Qualitätsmanagement.* Carl Hanser Verlag München, München · Wien, 2 edition, 2001. 1