Relationale Algebra im dreidimensionalen Software-Entwurf - ein werkzeugbasierter Ansatz

Oliver Szymanski

Betreuer: Dr. Alexander Fronk
Lehrstuhl Software-Technologie
Fachbereich Informatik
Universität Dortmund

3. Dezember 2003

Inhaltsverzeichnis

Zusammenfassung

Die vorliegende Diplomarbeit ist im Kontext des dreidimensionalen Software-Entwurfs angesiedelt.

Schwerpunkt der Arbeit ist die Entwicklung einer Java-Bibliothek, welche den Zugriff auf eine vorhandene native Bibliothek des Institutes für Informatik und Praktische Mathematik der Technischen Fakultät der Christian-Albrechts-Universität zu Kiel zur Definition und Manipulation von Relationen und der Auswertung von relationalen Termen ermöglicht. Desweiteren wird innerhalb der Arbeit ein Prototyp entwickelt, der die Verbindung zwischen dem dreidimensionalen Software-Entwurf und der relationalen Algebra mit Hilfe der entwickelten Java-Bibliothek darstellt, und der die weitere Arbeit im Bereich des dreidimensionalen Software-Entwurfs am Lehrstuhl für Software-Technologie der Universität Dortmund unterstützt.

Die Arbeit ist wie folgt gegliedert:

- Kapitel 1 beschreibt das Thema der Arbeit und den Kontext in dem sie entwickelt wurde. Die Aufgabenstellung sowie der in der Arbeit entwickelte Lösungsansatz werden erläutert.
- Kapitel 2 befaßt sich mit der nativen Bibliothek sowie mit dem Programm RELVIEW, in welchem sie angewendet wird.
- Kapitel 3, 4, 5 und 6 diskutiert die Entwicklung der Java-Bibliothek anhand der Teillösungen, welche sich aus dem gewählten Lösungsansatz ergeben.
- Kapitel 7 beschäftigt sich mit der Entwicklung des Prototypen, der die Java-Bibliothek nutzt und demonstriert, wie die Java-Bibliothek im Umfeld des dreidimensionalen Software-Entwurfs eingesetzt werden kann.
- Kapitel 8 gibt einen Ausblick auf weitere Einsatzmöglichkeiten.

Kapitel 1

Einleitung

In der Softwareentwicklung werden oft zweidimensionale Ansätze zum graphischen Design von Software genutzt. So können mit Hilfe der UML zum Beispiel Vererbungshierarchien und Assoziationen in zweidimensionalen Klassendiagrammen dargestellt werden. Es existiert eine große Zahl an Programmen, die das Entwickeln von Software anhand von zweidimensionalen Zeichnungen am Computer erlauben. Mit diesen Hilfsprogrammen kann man Software sowohl visualisieren als auch die Darstellungen editieren. Am Lehrstuhl Software-Technologie wird an einem Ansatz gearbeitet dreidimensionale Darstellungen beim Software-Entwurf zu nutzen. Dabei soll ein Programm entwickelt werden, welches Software dreidimensional visualisiert und das erlaubt Änderungen an der Darstellung vorzunehmen. Inwiefern die vorliegende Arbeit in diese Thematik eingebunden ist und die Entwicklung eines solchen Programmes unterstützt, wird in diesem Kapitel erläutert.

1.1 Dreidimensionale Darstellungen im Software-Entwurf

In [1],[3] und [2] wird ein Ansatz beschrieben, für den Software-Entwurf dreidimensionale Darstellungen zu nutzen. Dieser Ansatz legt Java-Programme zugrunde. Die Programmierung im Kleinen, d.h. Methoden von Klassen zu entwickeln, wird vernachlässigt. Es werden hier nur die Darstellungen für Relationen zwischen Java-Paketen, -Schnittstellen und -Klassen, demnach die statische Struktur der Software betrachtet. Diese Struktur ergibt sich unter anderem durch Klassenhierarchien, Pakethierarchien, Schnittstellenhierarchien, `Implements`-Beziehungen, sowie Assoziationen und `Uses`-Beziehungen. Der Ansatz will nicht die zweidimensionalen Darstellungen der UML in dreidimensionale umsetzen, sondern beschreibt vielmehr, wie man die Vorteile von dreidimensionalen Darstellungen effektiv im Software-Entwurf nutzen kann.

Die besonderen Eigenschaften von dreidimensionalen Welten sind die Transparenz von graphischen Objekten, der Perspektiveneffekt und die Art und Weise, wie man Objekte im dreidimensionalen Raum anordnen kann. In [1], [3] und [2] werden unterschiedliche Anordnungsformen zur Darstellung verschiedener Beziehungen genutzt. Diese werden folgend kurz erläutert:

`Cone Trees` dienen der Darstellung von Vererbungshierarchien. Ein `Cone Tree` hat eine Spitze, welche die Superklasse der betrachteten Vererbungshierarchie darstellt. Die abgeleiteten Klassen werden auf einer Kreisbahn unterhalb der Spitze dargestellt und mit Pfeilen mit der Spitze verbunden.

`Cone Trees` können rekursiv zur Darstellung komplexer Vererbungshierarchien genutzt werden.

Für die Darstellung der Paketzugehörigkeit von Klassen und für Pakethierarchien werden sogenannte `Information Cubes` genutzt. `Information Cubes` gruppieren verwandte Informationen in „Kisten“, welche beliebig verschachtelt werden können. Dabei muss eine solche Kiste immer vollständig in einer übergeordneten liegen. Die Kisten besitzen semitransparente Wände, die es ermöglichen, den Inhalt der Kisten sehen zu können.

Schnittstellenhierarchien werden in sogenannten `Walls` dargestellt. Eine `Wall` ist dabei eine zweidimensionale Ebene. Innerhalb der Ebene wird die Schnittstellenhierarchie als Graph dargestellt. Durch die dreidimensionale Perspektive lassen sich Teile des Graphen im Vordergrund darstellen, während der Rest kleiner aber sichtbar im Hintergrund erscheint, beispielsweise erzeugbar durch eine schräge Sicht vom Beobachterstandpunkt auf die Ebene.

Natürlich kann man die unterschiedlichen Darstellungsformen auch kombinieren. `Cone Trees`, `Information Cubes`, `Walls` und die darin enthaltenen 3D-Objekte werden als graphische Entitäten bezeichnet. Die Positionen dieser Entitäten im dreidimensionalen Raum sind wichtig, da anhand ihrer Anordnung die Beziehungen der dargestellten Klassen, Schnittstellen und Pakete verdeutlicht werden. Insbesondere benötigt man Relationen der Art *enthalten sein*, *nahe bei* und *überschneiden sich*, um aus der graphischen Darstellung wieder Beziehungen in Java abzuleiten. Solche Relationen werden im Abschnitt 1.4 auf Seite 7 noch weiter erläutert.

1.2 Das Eclipse Plugin

Am Lehrstuhl Software-Technologie des Fachbereiches Informatik der Universität Dortmund werden zur Zeit Vorarbeiten für ein Plugin zur dreidimensionalen Darstellung von statischen Beziehungen in Java-Programmen für die Entwicklungsumgebung Eclipse (siehe [10]) durchgeführt. Das Plugin soll für die Darstellung die genannten Anordnungsformen nutzen. Außerdem soll es dem Benutzer erlauben, sich in der dargestellten Szene frei zu bewegen. Das Eclipse-Plugin wird nach dem Model/View/-Controller Entwurfsmuster entwickelt. Es nutzt als Modell den abstrakten Syntaxgraphen des in Eclipse erzeugten und verwalteten Quellcodes und liefert eine dreidimensionale Sicht darauf. Diese Sicht entspricht einem View auf den Syntaxgraphen, analog zur Darstellung durch UML-Diagramme, die ebenfalls als Sicht auf einen Syntaxgraphen verstanden werden können.

1.3 Manipulationen der Darstellung

Für die Nützlichkeit des Eclipse-Plugins ist es auch im Vergleich zu ähnlichen Programmen mit zweidimensionalen Darstellungsformen wichtig, ebenfalls die dreidimensionale Darstellung manipulieren zu können. Auf diese Weise kann man die statischen Beziehungen in einem betrachteten Java-Programm innerhalb der dreidimensionalen Darstellung ändern.

Nach einer Manipulation muss der Syntaxgraph bzw. der Java-Quellcode mit der erhaltenen Darstellung synchronisiert werden. Damit nach der Synchronisation korrekter Java-Quellcode vorliegt, muss das Eclipse-Plugin überprüfen, ob die neue Darstellung noch erlaubten statischen Strukturen in Java entspricht und somit die Manipulation gültig ist.

Um die Manipulationen in der dreidimensionalen Darstellung auf Gültigkeit überprüfen zu können, benötigt man einen geeigneten Mechanismus der in [2] beschrieben ist. Der Mechanismus sieht vor, Eigenschaften der Diagramme durch Prädikate zu erfassen und diese nach Manipulation der Darstellung zu prüfen. Dies entspricht dem deklarativen Ansatz zur Definition graphischer Sprachen (Vergleiche [12]). Es gibt zwei Arten von Eigenschaften, die eine Darstellung erfüllen muss. Die erste Art ergibt sich durch die geometrischen Darstellungsarten, zum Beispiel werden wie im vorherigen Abschnitt näher erläutert, Schnittstellenhierarchien in einer Ebene dargestellt. Daraus ergibt sich die Eigenschaft, dass alle Schnittstellen, welche zu einer Hierarchie gehören, in einer Ebene des Raumes liegen müssen. Die zweite Art beschreibt, welche Eigenschaften in der Darstellung erfüllt sein müssen, damit aus dieser gültiger Java-Quellcode generiert werden kann. Darunter fällt z.B. die Eigenschaft, dass eine Schnittstellenhierarchie keine Zyklen aufweisen darf oder eine Klasse in genau einem Paket liegen muss.

Eigenschaften von Java lassen sich in Eigenschaften der dreidimensionalen Diagramme übersetzen. [2] nennt Bedingungen, die in der dreidimensionalen Darstellung erfüllt sein müssen, damit die Darstellung konform zur Java-Syntax ist. Diese Bedingungen spiegeln sich in Eigenschaften der dreidimensionalen Darstellung wieder. So ist es zum Beispiel nicht erlaubt, dass Klassen in zwei Paketen liegen. Dies bedeutet, dass für die dreidimensionalen Darstellung die Eigenschaft „keine `Information Cubes` überschneiden sich“ gelten muss.

Alle Eigenschaften müssen nach einer Manipulation auf ihre Gültigkeit geprüft werden. Ist bereits eine Eigenschaft nicht gültig, kann kein Quellcode generiert werden. In diesem Fall ist das Diagramm inkonsistent und muss durch weitere Manipulationen konsistent gemacht werden.

In [8] wird beschrieben, wie diese Bedingungen bzw. Diagrammeigenschaften mit Hilfe der Relationalen Algebra beschrieben und mit dem Programm RELVIEW effizient auf ihre Gültigkeit geprüft werden können. Darauf wird nachfolgend näher eingegangen. RELVIEW wird in Abschnitt 1.5 auf Seite 8 diskutiert.

1.4 Relationen zur Prüfung der Darstellung und der Manipulationen

In [8] wird gefordert, dass jedes konsistente dreidimensionale Diagramm den syntaktischen Eigenschaften der Java-Syntax und den geometrischen Bedingungen, welche aus den möglichen Anordnungen hergeleitet werden, genügen muss. Grundlegende Relationen in der statischen Struktur eines Java-Programmes sind die baumartigen Hierarchien über Pakete und Klassen und der azyklische, gerichtete Graph der Schnittstellenhierarchie, sowie der gerichtete Graph der Assoziationen und der Uses-Beziehungen. Wir schreiben $R : A \leftrightarrow B$, falls R eine Relation zwischen den Mengen A und B ist und $R_{x,y}$, falls das Paar (x, y) in der Relation R enthalten ist.

In [8] werden drei Relationen für Klassenhierarchie, Pakethierarchie und Schnittstellenhierarchie angegeben, die visualisiert werden sollen. Dazu kommen sechs weitere Relationen für die Beziehungen zwischen der dreidimensionalen Darstellung und dem abstrakten Syntaxgraphen des betrachteten Java-Programms. Dazu gehört zum Beispiel die Zugehörigkeit von Klassen, Schnittstellen und Paketen zu `Information Cubes` $cM : CUBE \leftrightarrow CLASS \cup INTERFACE \cup PACK$.

Es handelt es sich bei $CUBE$ um die Menge der `Information Cubes`, sowie bei $CLASS$, $INTERFACE$ und $PACK$ um die Mengen aller Java-Klassen, -Schnittstellen und -Pakete. Eine weitere wichtige Relation ist $in : ENTITY \leftrightarrow ENTITY$, die angibt, ob sich eine Entität

in der Darstellung innerhalb einer anderen befindet. Dabei ist ENTITY die Vereinigungsmenge der `Information Cubes`, `Walls`, `Cone Trees`, Boxen und Kugeln. Boxen visualisieren Klassen und Kugeln Schnittstellen in der dreidimensionalen Darstellung. Die oben aufgezeigten Bedingungen für die Korrekheit der dreidimensionalen Darstellung werden zuerst in der Schreibweise der Prädikatenlogik aufgeschrieben, z.B.

$\forall b, c : in_{b,\,c} \leftrightarrow cM_{c,\,C(b)} \vee \exists c' : cM_{c,\,P(c')} \wedge in_{b,\,c'}$

Dies gibt an, dass sich eine Box b genau dann in einem `Information Cube` c befindet, wenn die der Box entsprechende Java-Klasse $C(b)$ direkt zu dem Paket $P(c)$ gehört oder sich in einem Subpaket $P(c')$, welches einem `Information Cube` c' in der Darstellung entspricht, innerhalb des `Information Cubes` c befindet. Weitere Bedingungen und ihre relationen-algebraische Formulierung sind in [8] ersichtlich. Die Bedingungen werden dazu genutzt, die syntaktische Korrektheit von Diagrammen zu überprüfen. Die Idee ist jetzt, die dreidimensionalen Diagramme anhand der gezeigten Relationen nach Manipulation auf die Einhaltung der Bedingungen hin zu überprüfen. Dies muss effizient erfolgen, da dem Benutzer des Programms bei der Arbeit an der Darstellung keine langen Wartezeiten zugemutet werden können. Dazu kann das im folgenden Abschnitt 1.5 erläuterte Programm RELVIEW genutzt werden.

1.5 RELVIEW zur Auswertung der Invarianten

Das in der Programmiersprache C entwickelte Programm RELVIEW, welches in [5] und [6] dokumentiert ist, kann relationen-algebraische Ausdrücke effizient auswerten. RELVIEW erlaubt die Eingabe von Relationen am Bildschirm oder durch in ASCII codierte Adjazenzlisten über das Dateisystem. RELVIEW bietet unterschiedliche Sichten auf diese Relationen. Man kann sich die Relationen als boolesche Matrizen oder Graphen anzeigen lassen. RELVIEW bietet Unterstützung bei der Manipulation von Relationen durch vordefinierte Funktionen und erlaubt die Deklaration von neuen Funktionen und die Angabe von relationalen Programmen, die ausgeführt werden können. Bei den relationalen Programmen handelt es sich um `While`-Programme, mit deren Hilfe man z.B. Graphalgorithmen implementieren und in der RELVIEW-Umgebung ausführen kann um komplexe Eigenschaften von Relationen zu betrachten.

Da man alle für dreidimensionale Diagramme relevanten Bedingungen durch relationale Ausdrücke formalisieren kann, lassen sich diese mit RELVIEW auswerten. Weil es sich bei RELVIEW um ein interaktives, fensterorientiertes Programm handelt, ist es nicht direkt möglich, relationale Ausdrücke aus dem Eclipse-Plugin mit Hilfe von RELVIEW auszuwerten. Die Lösung, die betrachteten Relationen und die auszuwertenden Ausdrücke als ASCII-Dateien abzuspeichern und diese dann vom Benutzer interaktiv mit RELVIEW auswerten zu lassen, ist nicht effizient und auch nicht benutzerfreundlich, da diese bei jeder Manipulation der dreidimensionalen Darstellung erzeugt und ausgewertet werden müssen.

Daher müssen die C-Funktionen zur Definition und Auswertung von Relationen von RELVIEW direkt zur Verfügung stehen, um von praktischem Nutzen für die Entwicklung des Eclipse-Plugins zu sein. Dies ermöglicht dem Eclipse-Plugin den Aufruf von Funktionen von RELVIEW für die Auswertung der Bedingungen. Erst dann können Manipulationen der Darstellung, die zugehörige Korrekheitsprüfung und die Synchronisation mit dem Quellcode implementiert werden. Auf diese Weise wird das Eclipse-Plugin nicht nur eine weitere Sicht des abstrakten Syntaxgraphen zur Verfügung stellen, son-

dern ermöglicht es dem Benutzer, interaktiv Änderungen am Java-Quellcode in der dreidimensionalen Darstellung vornehmen zu können.

Die Menge der benötigten Funktionen wird im folgenden als der funktionale Kern von RELVIEW bezeichnet, der in Kapitel 2 definiert wird. Der extrahierte funktionale Kern des Programms RELVIEW kann auch in andere Projekte integriert werden, die ähnliche Bedingungen auf Relationen auswerten müssen, oder aber aus anderen Gründen mit relationalen Ausdrücken arbeiten. Ebenso kann die Unterstützung von relationalen Programmen, die RELVIEW anbietet genutzt werden, um in Programmen Graphalgorithmen anzuwenden, da diese relational aufgeschrieben werden können. Die Einsatzmöglichkeiten des funktionalen Kerns sind somit für unterschiedlichste Aufgaben auch über die Diplomarbeit hinaus denkbar. Der bereits heute in der Literatur hinreichend dokumentierte breite Einsatz von RELVIEW belegt weiterhin die Sinnhaftigkeit dieses Tuns.

1.6 Ziel der Diplomarbeit

Ziel der Diplomarbeit ist, den funktionalen Kern des Programmes RELVIEW innerhalb von Java-Programmen ansprechbar zu machen und dadurch die Relationale Algebra im dreidimensionalen Software-Entwurf nutzbar zu machen. Dazu wird eine Java-Bibliothek entwickelt, welche auf den in C implementierten funktionalen Kern von RELVIEW zugreift und seine Funktionalität beliebigen Java-Programmen zur Verfügung stellt. Ein geeignetes Plugin soll prototypisch implementiert werden, um die Anbindung von RELVIEW testen zu können. Nach Fertigstellung der Diplomarbeit ist es möglich, die vorgenannten Diagrammeigenschaften in diesem Eclipse-Plugin über geeignete Methodenaufrufe direkt durch den funktionalen Kern von RELVIEW effizient auswerten zu lassen. Dadurch kann das Eclipse-Plugin Manipulationen auf ihre Gültigkeit prüfen.

Die Diplomarbeit setzt sich grob aus zwei Teilen zusammen:

- das Kapseln des funktionalen Kerns von RELVIEW in eine Java-Bibliothek,
- die prototypische Realisierung eines Plugins für Eclipse, welches auf die Java-Bibliothek zugreift.

Im Folgenden werden die in der Arbeit zu betrachtenden Aspekte näher beschrieben. Die Diplomarbeit wird inhaltlich strukturiert und ein Arbeitsplan erstellt.

1.7 Lösungsansatz

Um RELVIEW für Java-Programme zur Verfügung zu stellen, sind die im Folgenden beschriebenen Aspekte zu berücksichtigen. Da bislang keine Dokumentation der Funktionen von RELVIEW existiert, ist eine intensive Einarbeitungszeit in den Quellcode von RELVIEW notwendig. Wie bereits genannt ist es nicht effizient, RELVIEW über das Dateisystem mit eingehenden Daten zu versorgen, die RELVIEW dann verarbeitet. Deutlich effizienter ist es, direkt auf Funktionen von RELVIEW zuzugreifen. Dazu muss RELVIEW ohne die interaktive, graphische Oberfläche in Form einer betriebssystemunabhängigen C-Bibliothek zur Verfügung gestellt werden. Diese C-Bibliothek muss dann für Java verfügbar gemacht werden. Dabei ist zu klären, wie Datenaustausch zwischen dem Java-Programm und der

C-Bibliothek möglich gemacht wird. Auch muss eine geeignete Möglichkeit der Fehlerbehandlung berücksichtigt werden. Es können Fehler beim Datenaustausch oder innerhalb des funktionalen Kerns bei der Auswertung von Formeln auftreten. Hier bietet sich eine geeignete Ausnahmenbehandlung in Java an.

In Abbildung 1.1 auf Seite 11 ist ersichtlich, welche Schichten implementiert werden, um den funktionalen Kern zugänglich zu machen.

Um von Java aus auf Funktionen in C zuzugreifen, wurde das *Java Native Interface* (*JNI*) gewählt. Über *JNI* kann man von Java native Funktionen aufrufen. Die in C geschriebenen entsprechenden Funktionen, die nach Konvention von *JNI* den als `native` definierten Methoden zugehören, unterliegen einem Namensschema. Das Namensschema ermöglicht es Java, zu einer als `native` definierten Methode das entsprechende Gegenstück zu finden. Dieses besondere Namenschema verhindert den direkten Zugriff auf beliebige Funktion der C-Bibliothek. Daher bietet sich hier das Adapterentwurfsmuster an. Es müssen demnach Adapter für die C-Bibliothek entwickelt werden, deren Funktionsnamen dem *JNI* Namensschema unterliegen und die die Aufrufe der eigentlichen Funktionen des funktionalen Kerns kapseln (Vergleiche dazu Abbildung 1.1 auf Seite 11). Es ist eine weitere Adapterschicht nötig, welche die erforderlichen Datentypkonvertierungen zwischen C und Java übernimmt.

Auf Java-Seite ist bei *JNI* eine Java-Adapterschicht erforderlich, in der die C-Funktionen durch native Methodendeklarationen gekapselt werden. Zur Laufzeit werden dann beim Aufruf dieser nativen Methoden die C-Funktionen über *JNI* aufgerufen. Somit kann man die C-Funktionen aus RELVIEW von Java aus aufrufen. Um nicht nur Funktionen nutzen, sondern um auch Objektorientierung und Java-Ausnahmenbehandlung einsetzen zu können, ist eine weitere Schicht erforderlich, welche auf der Java-Adapterschicht aufsetzt und Java-Objekte und Java-Ausnahmen statt einfacher Datentypen und Fehlerübermittlung anhand von speziellen reservierten Rückgabewerten verwendet. Zuletzt ist eine Schicht notwendig, die Java-Client-Programmen den Zugriff auf die Bibliothek erlaubt.

Ist dieser Schritt getan und liegt die Bibliothek damit einsatzbereit für Java-Programme vor, ist ein Prototyp für das Eclipse-Plugin zu entwickeln, der aus einem definierten dreidimensionalen Diagramm, einer sogenannten Szene, Beziehungen herleitet und darüber Relationen beschreibt. Bei dem Prototypen handelt es sich nicht um ein vollständiges Plugin zur Manipulation, sondern um ein Demonstrationsprogramm. Der Prototyp wird also keine interaktive dreidimensionale Manipulation des Syntaxgraphen beinhalten. Der Prototyp erhält als Eingabe eine Szenenbeschreibung in Form einer XML-Datei, welche das dreidimensionale Diagramm in einer XML-Struktur beschreibt. Der Prototyp ermittelt Relationen aus der Darstellung und prüft diese Relationen auf Eigenschaften. Ergibt die Prüfung, dass die durch die Szene visualisierten Pakete, Schnittstellen und Klassen in Java gültige Beziehungen besitzen, generiert der Prototyp diese über Eclipse.

Es ist dabei vorab zu spezifizieren, welche Relationen benötigt werden. Diese Relationen kann man RELVIEW übergeben und ihre Eigenschaften prüfen. Diese Eigenschaften entsprechen dann den skizzierten Bedingungen. Dabei sind sowohl die Eigenschaften, die sich aus der Syntax von Java, als auch die Eigenschaften, die sich aus den erlaubten Anordnungen von dreidimensionalen Objekten ergeben, zu beachten. Diese Eigenschaften sind sie so zu formalisieren, dass sie mit RELVIEW auf den vorher ermittelten Relationen getestet werden können. Auf diese Weise kann das Eclipse-Plugin die ermittelten Relationen nach Manipulation der Diagramme prüfen. Sind alle Eigenschaften erfüllt, kann im Plugin aus der Darstellung Java-Quellcode generiert werden.

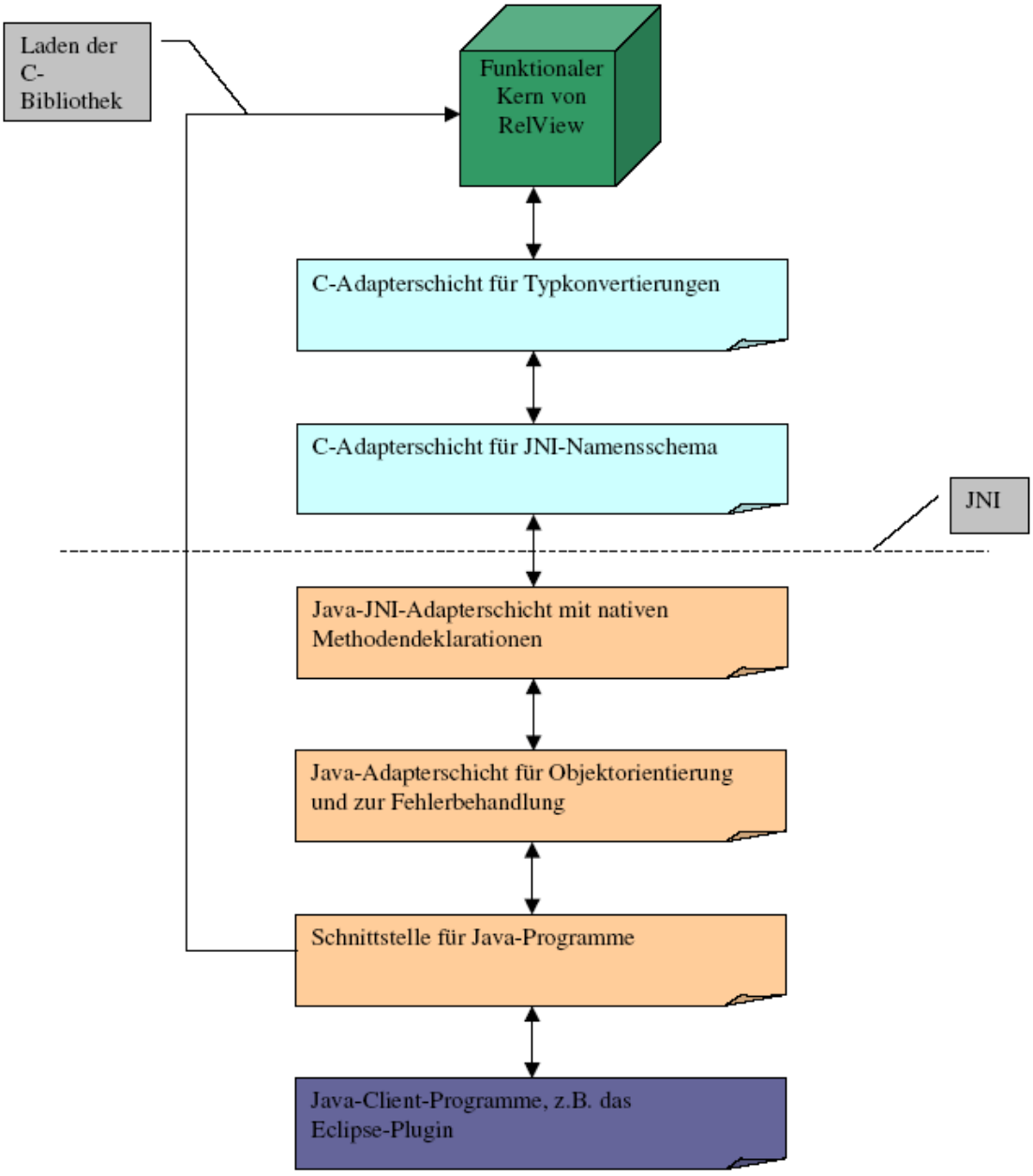

Abbildung 1.1: Grobentwurf der Schichten, um von Java-Programmen auf RELVIEW zuzugreifen. Unter dem Begriff „funktionaler Kern von RELVIEW“ sind die C-Funktionen aus RELVIEW zusammengefaßt, die benötigt werden. Eine genaue Definition des funktionalen Kerns wird in Kapitel 2 gegeben. Dabei sind die hellblauen und hellbraunen Schichten als Teil der Diplomarbeit zu implementieren.

1.8 Zusammenfassung der Arbeitsschritte

Um RELVIEW für Java-Programme nutzbar machen zu können, ist eine Analyse des Quellcodes von RELVIEW erforderlich. Es müssen notwendige Funktionen aus RELVIEW extrahiert vorliegen. Es ist unter anderem erforderlich das Adapterentwurfsmuster in C für den Zugriff auf den funktionalen Kern umzusetzen, um die Funktionen für das Namensschema von *JNI* verfügbar zu machen und Typumwandlungen zwischen den Programmiersprachen vorzunehmen. Auf Java-Seite ist das Adapterentwurfsmuster anzuwenden, um die nativen Funktionsdefinitionen zu kapseln, Objektorientierung zu unterstützen und den in Java bekannten Ausnahmebehandlungsstandard bei auftretenden Fehlern zu ermöglichen. Es ist eine Schicht erforderlich, welche den Zugriff auf die entwickelte Bibliothek ermöglicht. Ein Prototyp des Eclipse-Plugins ist zu entwickeln.

Kapitel 2

RELVIEW

Das Programm RELVIEW ermöglicht das Rechnen mit Relationen und das Entwickeln von relationalen Programmen (vgl. [5], [6]). Bei RELVIEW handelt es sich um ein C-Programm mit graphischer Benutzerschnittstelle. Eine frühe Version von RELVIEW wurde bereits 1988 bis 1992 an der Universität der Bundeswehr München entwickelt. Aufbauend auf den damaligen Erfahrungen wurde RELVIEW an der Kieler Universität neu entwickelt und ist unter [14] verfügbar.

Innerhalb der RELVIEW Oberfläche kann man Relationen definieren, Funktionen deklarieren, relationale Programme eingeben oder laden und relationale Terme auswerten sowie verschiedene Sichten auf Relationen betrachten. Innerhalb der Oberfläche werden die definierten Relationen, Funktionen und die geladenen Programme angezeigt und können verwaltet werden.

RELVIEW ermöglicht die Eingabe von Relationen interaktiv oder über das Dateisystem. Die Eingabe über die X-Window Oberfläche erfolgt durch zweidimensionale boolesche Matrizen oder in Form von gerichteten Graphen, welche sich in der graphischen Oberfläche zeichnen lassen. Die Eingabe über das Dateisystem erfolgt durch Adjazenzlisten, welche in Form von ASCII-codierten Dateien angegeben werden. Dazu muss die entsprechende Lade-Funktion innerhalb der Oberfläche vom Benutzer aufgerufen werden. Auch können definierte Relationen in diesem Dateiformat abgespeichert werden. RELVIEW bietet zwei Visualisierungen auf eingegebene Relationen: eine Matrizendarstellung und verschiedene Grapharten. In den Graphen lassen sich Graphkanten und Graphknoten durch Markierungen hervorheben. Über zusätzliche Dateien, die der Benutzer durch eine Laden-Funktion einlesen kann, ist es möglich, den Reihen und Spalten der Matrixdarstellung beziehungsweise den Graphknoten Beschriftungen hinzuzufügen. Diese Beschriftungen gelten allerdings ausschließlich für die unterschiedlichen Sichten der Relation. Sie werden nicht zu der Relation gespeichert.

2.1 Evaluierung von relationalen Termen

RELVIEW dient hauptsächlich der Evaluierung von relationalen Termen. Relationale Terme werden aus vordefinierten Operationen, vordefinierten Funktionen und selbstdefinierten Funktionen gebildet. Dabei gibt man die relationalen Terme als Zeichenketten ein. Man kann die Operationen und Funktionen dabei beliebig kombinieren und daraus relationale Terme bilden.

Die in RELVIEW vordefinierten Operationen werden durch die folgenden Sonderzeichen eingegeben:

- $-$ zur Komplementbildung
- $\wedge$ für Transpositionen
- $\&$ zur Durchschnittsbildung
- $|$ zur Vereinigung
- $*$ für Multiplikation

Diese Operationen können in den relationalen Termen genutzt werden. Sie bilden die grundlegenden Möglichkeiten mit Relationen zu arbeiten. Ebenso kann man in den Termen auf Funktionen zurückgreifen.

Funktionen werden in RELVIEW über ihren Namen angesprochen. Eventuelle Parameter, welche die Funktion erwartet, werden in Klammern hinter dem Namen angegeben. Es existieren an vorgefertigten Funktionen in RELVIEW unter anderem $empty(R)$, um zu testen, ob es sich bei einer Relation R um eine leere Relation handelt, $equal(Rel1, Rel2)$, um zu prüfen, ob zwei Relationen $Rel1$ und $Rel2$ gleich sind, $dom(R)$ und $ran(R)$ zur Bestimmung des Vor- und des Nachbereiches einer Relation R und $trans(R)$ zur Bestimmung des transitiven Abschlusses einer Relation R. Weitere vordefinierte Funktionen sind unter [6] ersichtlich. Neue Funktionen definiert man, indem man den Funktionsnamen, gefolgt von Parametern in Klammern, einem Gleichheitszeichen und dem relationalen Term, welcher die Funktion bildet, angibt (vergleiche [6]). Bei den Parametern handelt es sich um Relationen. Beispielsweise könnte man die Funktion $ungleich(Rel1, Rel2)$ wie folgt definieren: $ungleich(Rel1, Rel2) = -equals(Rel1, Rel2)$.

Jeder Term in RELVIEW liefert nach Auswertung als Ergebnis erneut eine danach verwendbare Relation. Die Ergebnisrelation wird dabei entweder in eine neue Relation mit dem Namen `result` gespeichert oder in eine bereits existierende Relation, die angegeben werden kann.

2.2 Relationale Programme

Zusätzlich zur Auswertung von relationalen Termen bietet RELVIEW die Möglichkeit, in einer `While`-Programmiersprache relationale Programme zu definieren und sie wie Funktionen auf Relationen in Termen anzuwenden. Diese Programme basieren auf dem Datentyp `Relation` und haben Ähnlichkeiten zu Prozeduren der Programmiersprache Pascal oder Modula-2. Ein relationales Programm ist dabei vergleichbar mit einer Prozedur. Relationale Programme werden in ASCII-codierten Dateien definiert. Der Programmcode ist für Menschen lesbar und kann über die Oberfläche durch administrative Kommandos eingelesen werden. Die Unterstützung für Relationale Programme wurde innerhalb der vorliegenden Arbeit berücksichtigt, damit die in der Arbeit entwickelte Java-Bibliothek diese Funktionalität bietet, somit ein größeres Einsatzgebiet abdeckt und für spätere Erweiterungen bei der Entwicklung von Programmen im dreidimensionalen Software-Entwurf, wie z.B. automatische Entwurfsmustererkennung in der dreidimensionalen Darstellung von Software, genutzt werden kann. Innerhalb des Prototypen werden keine Relationalen Programme genutzt.

Relationale Programme beginnen mit dem Programmnamen und einer Liste der formalen Parameter des Programms in Klammern. Nach dieser Kopfzeile beginnt der Deklarationsteil des Programmes, in dem lokale Relationen, Funktionen und Variablen definiert werden. Die Deklarationen werden mit dem Schlüsselwort `DECL` eingeleitet. Im anschließenden Hauptteil des Programmes, welcher mit `BEG` eingeleitet und mit `END.` abgeschlossen wird, erfolgt der eigentliche Programmcode. Dieser besteht aus einer Sequenz von durch Semikolons getrennten Anweisungen. Die letzte Anweisung in dieser

Abfolge ist die RETURN-Anweisung, welche angibt, welche Relation das Programm zurückliefert. Folgendes Beispiel ist unter [5] beschrieben. es handelt sich dabei um die Prim-Methode zur Berechnung der Relation eines Spannbaumes für einen nicht leeren, ungerichteten und verbundenen Graphen, gegeben durch die symmetrische und irreflexive Relation E.

Bsp.: Spannbaumberechnung

```
PRIM(E)
  DECL T,v
  BEG  T=atom(E);
       v=dom(T) | ran(T);
       WHILE -empty(-v) DO
         T=T | atom(v*-v^ & E);
         v=dom(T) | ran(T)
         OD
       RETURN T | T^
  END.
```

In dem Programm wird eine Relation E als Eingabeparameter angenommen. Es werden mit der DECL-Anweisung lokale Relationen T und E angelegt. Die erste Anweisung weist der Relation T einen Eintrag aus der Relation E zu. Danach wird die Relation v als Vereinigung des Vor- und Nachbereiches der Relation T gebildet. Solange dann das Komplement von v nicht leer ist, wird die WHILE-Schleife durchlaufen. In der WHILE-Schleife wird die Relation T bei jedem Durchlauf durch einen Eintrag erweitert, welcher aus v, multipliziert mit dem Komplement der transponierten v-Relation und dem Durchschnitt mit E stammt. Die Relation v wird innerhalb der Schleife vor dem Ende eines Durchlaufes als Vereinigung des Vor- und Nachbereiches der erweiterten Relation T gesetzt. Am Ende der WHILE-Schleife wird die Vereinigungsrelation aus der Relation T und ihrer Transposition zurückgeliefert.

2.3 Anwendungen von RELVIEW

Viele Konzepte der Mathematik und der Informatik lassen sich durch Relationen veranschaulichen. Dazu gehören unter anderem Graphen, Ordnungen und relationale Datenbanken. Weitere Beispiele sind in der Literatur dokumentiert, siehe [9] und [16]. In den vergangenen Jahren wurde RELVIEW an vielen Fallstudien erprobt und hat sich dabei bewährt. Unter anderem wurde RELVIEW im Bereich der Graphentheorie (vergleiche das Beispiel zur Spannbaumberechnung in Abschnitt 2.2 auf Seite 15), der Petri-Netze, bei Berechnungen von Ordnungen, bei Datenflussanalysen, bei der Analyse von endlichen Automaten und in der Lehre genutzt. Vergleiche dazu [5], [6], [7] und [4]. RELVIEW hat sich dabei als nützliches Werkzeug für den interaktiven Umgang mit Relationen bewiesen.

2.4 Der funktionale Kern von RELVIEW

Im Rahmen der Arbeit war eine intensive Einarbeitungsphase in RELVIEW und eine Analyse der C-Funktionen von RELVIEW erforderlich. Zusammen mit den Entwicklern von RELVIEW wurden

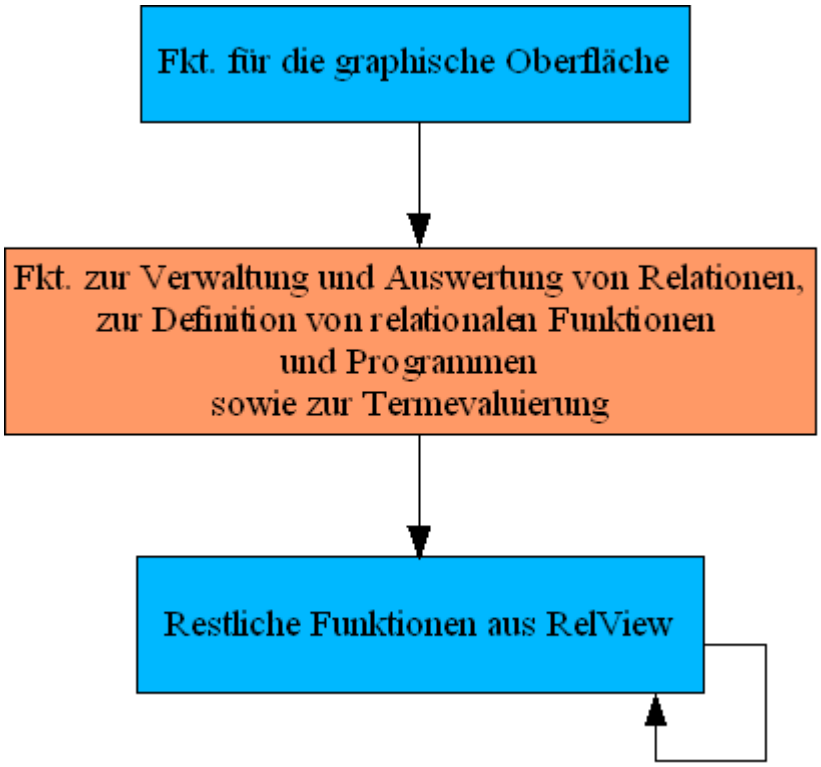

Abbildung 2.1: Die Einteilung der C-Funktionen aus RELVIEW in Schichten (Pfeile zeigen die Aufrufrichtung). Die unteren zwei Schichten sind vom Betriebssystem unabhängig.

zuerst die C-Funktionen, welche die graphische Oberfläche bilden, ermittelt. Aus den übrigen Funktionen wurden die ermittelt, welche von den GUI-Funktionen direkt aufgerufen werden. Somit ergab sich insgesamt eine Klassifizierung der Funktionen in drei Schichten, wobei die erste aus den GUI-Funktionen, die zweite aus den direkt von den GUI-Funktionen aufgerufenen und die dritte Schicht aus allen restlichen Funktionen besteht (siehe Abbildung 2.1 auf Seite 16).

Eine Analyse der Schichten ergab, dass jeweils Funktionen der dritten Schicht nur Funktionen aus der dritten Schicht nutzen, Funktionen der zweiten Schicht nur Funktionen der dritten Schicht und die Funktionen der ersten Schicht nur die der zweiten. Somit handelt sich bei der Aufteilung um eine klassische Schichtenarchitektur. Eine weitere Analyse der Funktionen ergab, dass die Funktionen der zweiten und dritten Schicht betriebssystemunabhängig sind und dass es sich bei den Funktionen der zweiten Schicht um Funktionen zur Verwaltung und Auswertung von Relationen, zur Definition von relationen Funktionen und Programmen und zur Termevaluierung handelt. Diese Funktionen der zweiten Schicht sind genau die Funktionen, welche zur Nutzung von RELVIEW für einen werkzeugbasierten Ansatz im dreidimensionalen Software-Entwurf benötigt werden (die Schicht ist in Abbildung 2.1 rötlich gefärbt). Diese werden im folgenden auch als die exportierten Funktionen des funktionalen Kerns bezeichnet. Da die Funktionen der zweiten Schicht die der dritten Schicht erfordern, bilden diese beiden Schichten die im folgenden als funktionalen Kern von RELVIEW bezeichnete Menge von Funktionen, die für einen werkzeugbasierten Ansatz im dreidimensionalen Software-Entwurf notwendig sind. Es handelt sich dabei um eine Liste von 16 Funktionen, die im nächsten Abschnitt aufgelistet und erläutert werden.

Definition funktionaler Kern: Eine Funktion `f` gehört zum funktionalen Kern von RELVIEW, wenn `f` keine Komponenten der graphischen Oberfläche bildet oder anspricht.

Definition exportierte Funktionen: Die Menge der exportierten Funktionen ist diejenige Teilmenge des funktionalen Kerns, für deren Funktionen f gilt, dass sie von den Funktionen, welche die graphische Oberfläche bilden, aufgerufen werden.

Da die Funktionsmenge von RELVIEW aus Funktionen besteht, welche die graphische Oberfläche bilden und aus Funktionen, die selbst nicht die graphische Oberfläche bilden, lässt sich für jede Funktion eindeutig entscheiden, ob sie zum funktionalen Kern von RELVIEW gehört. Für jede Funktion aus dem funktionalen Kern lässt sich eindeutig entscheiden, ob sie von den Funktionen der graphischen Oberfläche aus aufgerufen werden, so dass die Einteilung in den exportierten funktionalen Kern von RELVIEW ebenso eindeutig ist.

Der funktionale Kern von RELVIEW wurde in Form einer C-Bibliothek extrahiert. Die C-Bibliothek wird in [13] als „KURE package“ bezeichnet und erläutert. Die ausschließliche Nutzung des funktionalen Kerns über die exportierten Funktionen ist in [13] als Black-Box-Benutzung bezeichnet. Man kann über die C-Bibliothek auch auf weitere Funktionen von RELVIEW zugreifen; in [13] als Clear-Box-Benutzung bezeichnet. Dabei greift man auch auf die internen Funktionen der Bibliothek zu, um zum Beispiel Terme auszuwerten, statt Terme durch Zeichenketten anzugeben. Die Behandlung des funktionalen Kerns als Clear-Box wird hier nicht weiter beschrieben. Im Folgenden werden wir uns ausschließlich mit der funktionalen Bibliothek als Black-Box über die exportierten Funktionen beschäftigen.

2.5 Die exportierten Funktionen des funktionalen Kerns

Die exportierten Funktionen des funktionalen Kerns nutzen hauptsächlich zwei Datenstrukturen. Die Datenstruktur `Kure_Rel` kapselt den Zugriff auf Relationen und beinhaltet unter anderem den Namen, die Breite und Höhe der Relation. Die Datenstruktur `RelManager` beinhaltet Listen für Funktionen, Programme und Relation und ermöglicht, über ihren Namen auf diese zuzugreifen. Auf die Datenstrukturen `Kure_Rel` und `RelManager` muss bei der Benutzung der Bibliothek als Black Box nicht selbst zugegriffen werden. Ein `RelManager` muss zu Beginn über eine `Kure_Init` Methode initialisiert und den weiteren exportierten Funktionen beim Aufruf übergeben werden. Mit Hilfe der anderen Methoden lassen sich Relationen anlegen, Funktionen und Programme definieren und Terme auswerten.

- Kure_Init

  ```
  RelManager * Kure_Init (void);
  ```

  Diese Funktion muss bei Benutzung der Bibliothek zu Beginn aufgerufen werden. Es wird dabei ein Zeiger auf eine Datenstruktur vom oben beschriebenen Typen `RelManager` zurückgeliefert, den man für die weiteren Funktionen als Eingabeparameter benötigt.

- Kure_Quit

  ```
  void Kure_Quit (RelManager *);
  ```

Diese Funktion muss aufgerufen werden, wenn die Arbeit mit einem `RelManager` beendet ist. Die Bibliothek gibt den belegten Speicher frei. Auf diesen `RelManager` darf nicht mehr zugegriffen werden. Der Lebenszyklus eines `RelManagers` ist damit beendet. Da alle anderen Funktionen einen `RelManager` als Eingabeparameter benötigen, können die anderen Funktionen nur zwischen den Aufrufen von `Kure_Init`und `Kure_Quit` eines `RelManagers` genutzt werden. Die Funktion `Kure_Quit` muss zumindest beim Beenden eines Programms aufgerufen werden, damit der Speicher wieder freigegeben wird.

- Kure_RelationNew

```
Kure_Rel * Kure_RelationNew (RelManager *, char *,
                             int height, int width);
```

Mit dieser Funktion definiert man eine neue Relation. Man übergibt der Funktion als Eingabeparameter einen Zeiger auf einen existierenden `RelManager`, den gewünschten Relationsnamen als Zeichenkette und die Spalten- und Reihenanzahl der Relation. Der `RelManager` muss existieren und die Spalten- und Reihenanzahl dabei mindestens 1 betragen, sonst liefert die Funktion `NULL` zurück. Die Funktion liefert einen Zeiger auf eine Datenstruktur des oben genannten Typen `Kure_Rel` zurück, welcher eine Relation repräsentiert. Der übergebene Name wird in die Datenstruktur kopiert. Die neue Relation wird auch in der Datenstruktur des übergebenen `RelManagers` vermerkt. Eine Relation muss man sich dabei als eine binäre Matrix vorstellen, in der vermerkt ist, ob ein Paar zueinander in Relation steht oder nicht. Die Matrix kann als Einträge `0` und `1` beinhalten. Zu Beginn sind alle Einträge `0`, d.h. es steht nichts in Relation zueinander. Steht z.B. in Reihe 4, Spalte 3 eine `1` in der Matrix, beinhaltet die Relation das Paar `(4, 3)`. Das Setzen der Matrixeinträge geschieht mit Hilfe der später erläuterten Funktion `Kure_RelationSetBit`.

- Kure_RelationExists

```
int Kure_RelationExists (RelManager *, char *);
```

Die Funktion gibt genau dann 1 zurück, wenn eine Relation des übergebenen Namens in dem `RelManager` verwaltet wird.

- Kure_RelationGet

```
Kure_Rel *  Kure_RelationGet (RelManager *, char *);
```

Mit dieser Funktion kann man auf eine Relation aus einem `RelManager` zugreifen. Es wird ein Zeiger auf eine Datenstruktur vom Typen `Kure_Rel` zurückgegeben, welche die gewünschte Relation repräsentiert. Die Funktion gibt `NULL` zurück, wenn der übergebene `RelManager` keine Relation mit dem angegeben Namen verwaltet.

- Kure_RelationGetHeight

```
int Kure_RelationGetHeight (RelManager *, Kure_Rel *);
```

Diese Funktion liefert die Reihenanzahl der Relation zurück oder `0`, wenn einer der Eingabeparameter `NULL` ist.

- Kure_RelationGetWidth

```
int Kure_RelationGetWidth (RelManager *, Kure_Rel *);
```

  Diese Funktion liefert die Spaltenanzahl der Relation zurück oder `0`, wenn einer der Eingabeparameter `NULL` ist.

- Kure_RelationGetNumberOfEntries

```
int Kure_RelationGetNumberOfEntries (RelManager *,
                                     Kure_Rel *);
```

  Diese Funktion liefert die Anzahl der Einträge der Relation zurück oder `0`, wenn einer der Eingabeparameter `NULL` ist. Dabei handelt es sich um die Anzahl von zueinander in Relation stehenden Einträgen, also die Anzahl von `1`-Einträgen in der Relationsmatrix.

- Kure_RelationDelete

```
int Kure_RelationDelete (RelManager *, Kure_Rel *);
```

  Mit dieser Funktion kann man eine Relation löschen. Dabei wird sie aus dem übergebenen `RelManager` ausgetragen und der von ihr belegte Speicher freigegeben. Die Funktion gibt genau dann `1` zurück, wenn die Relation erfolgreich ausgetragen und der Speicher freigegeben werden konnte.

- Kure_RelationSetBit

```
int Kure_RelationSetBit (RelManager *, Kure_Rel *,
                         int row, int column);
```

  Mit dieser Funktion kann man einen Eintrag in der Relationsmatrix setzen. Dabei wird der entsprechende Wert in der Relationsmatrix auf `1` gesetzt. Die Position des Eintrages wird durch Reihen- und Spaltennummer angegeben. Die Funktion liefert genau dann `1`, wenn der Eintrag gesetzt werden konnte. Die Relation beinhaltet dann das Paar `(row, column)`.

- Kure_RelationClearBit

```
int Kure_RelationClearBit (RelManager *, Kure_Rel *,
                           int row, int  column);}
```

  Mit dieser Funktion kann man einen Eintrag in der Relationsmatrix löschen. Dabei wird der entsprechende Wert in der Relationsmatrix auf `0` gesetzt. Die Funktion liefert genau dann `1`, wenn der Eintrag gelöscht werden konnte. Die Relation beinhaltet dann das Paar `(row, column)` nicht mehr.

- Kure_RelationGetBit

```
int Kure_RelationGetBit (RelManager *, Kure_Rel *,
                         int row, int column);
```

Mit dieser Funktion kann man überprüfen, ob eine Relation ein Paar enthält. Die Funktion liefert genau dann `1`, wenn das Paar in der Relation enthalten ist, also der Eintrag an der Position gesetzt ist. Sie liefert genau dann `0`, wenn der Eintrag an der Position nicht gesetzt ist und `-1`, wenn ein Fehler aufgetreten ist.

- Kure_FunctionNew

```
int Kure_FunctionNew (RelManager *, char *);
```

Mit dem Aufruf von `Kure_FunctionNew` kann man eine neue Funktion definieren, die man in den relationalen Termen verwenden kann. Die Funktionsdefinition wird als Zeichenkette übergeben. Diese Definition wird dann dem übergebenen `RelManager` hinzugefügt. Es wird genau dann `1` zurückgeliefert, wenn die Funktion dem `RelManager` korrekt hinzugefügt werden konnte.

- Kure_ProgramNew

```
int Kure_ProgramNew (RelManager *, char *);
```

Mit dieser Funktion kann man ein Programm angeben, das man in den relationalen Termen verwenden kann. Das Programm wird als Zeichenkette übergeben. Es wird dem übergebenen `RelManager` hinzugefügt. Die Funktion liefert genau dann `1` zurück, wenn das Programm korrekt dem `RelManager` hinzugefügt werden konnte.

- Kure_ProgramFileRead

```
int Kure_ProgramFileRead (RelManager *, char * filename);
```

Mit dieser Funktion kann man ein Programm aus einer Datei einlesen und einem `RelManager` hinzufügen. In der Datei können auch mehrere Programme angegeben sein. Der Dateinamen wird als Zeichenkette angegeben. Die Funktion liefert genau dann `1` zurück, wenn die Datei eingelesen werden konnte.

- Kure_EvaluateTerm

```
int Kure_EvaluateTerm (RelManager *,
                       char * term,
                       char * resultname);
```

Mit dieser Funktion kann ein relationaler Term ausgewertet und das Ergebnis unter der mit dem Parameter `resultname` bezeichneten Relation abgelegt werden. Existiert der Name der Ergebnisrelation bereits, wird die Relation unter diesem Namen überschrieben. Die Funktion liefert genau dann `1` zurück, wenn der Term korrekt ausgewertet werden konnte.

2.6 Beispielprogramm

Im folgenden ist ein kurzes Beispielprogramm angegeben, welches die in 2.5 beschriebenen Funktionen benutzt. Die Beschreibung des Beispielprogrammes ist in den Kommentaren im Quelltext enthalten.

```
// Den RelManager anlegen und initialisieren
RelManager * relManager = Kure_Init();

// Relationen und Hilfsvariablen deklarieren
Kure_Rel * relResult;
Kure_Rel * rel1;
Kure_Rel * rel2;
int width;
int height;
int entries;
int result;
int i;
int j;

// Ein neues Programm anlegen, welches zwei Relationen als
// Eingabeparameter erwartet und die Vereinigung zurueckliefert
result =
  Kure_ProgramNew(relManager,
    "UNIT(R1,␣R2)␣DECL␣R3␣BEG␣R3=R1␣|␣R2␣RETURN␣R3␣END.";}

// Eine neue Funktion anlegen, welche zwei Relationen als
// Eingabeparameter erwartet und eine 1x1 Relation zurueckliefert,
// die als einzigen Eintrag eine 1 enthaelt, wenn die beiden
// uebergebenen Relationen ungleich sind, sonst 0
result =
  Kure_FunctionNew(relManager,
                   "ungleich(Rel1,␣Rel2)=-equals(Rel1,␣Rel2)");

// Eine neue Relation anlegen
rel1 = Kure_RelationNew(relManager, "relation1", 2, 2);
rel2 = Kure_RelationNew(relManager, "relation2", 2, 2);

// Pruefen ob die Relation korrekt angelegt wurde
result = Kure_RelationExists(relManager, "relation1");
if (result == 1) {

  // Alle Eintraege in der Relation 1 setzen
  width = Kure_RelationGetWidth (relManager, rel1);
  height = Kure_RelationGetHeight  (relManager, rel1);
  for (i = 0; i < height; i++) {
    for (j = 0; j < width; j++) {
      result = Kure_RelationSetBit(relManager,
                                   relation,
                                   i+1, j+1);
    }
```

```
  }

  // Term evaluieren der vorher angelegtes Programm benutzt
  // Ergebnis wird in einer neuen Relation mit Namen
  // "resultRelation" abgelegt
  result = Kure_EvaluateTerm(relManager,
                             "UNIT(rel1,rel2)",
                             "resultRelation");

  // Ergebnisrelation abfragen
  relResult = Kure_RelationGet(relManager, "resultRelation");

  // Testen ob Ergebnisrelation und Relation 1 ungleich sind
  // mit Hilfe der oben definierte Ungleichfunktion
  // Ergebnisrelation ist eine 1x1 Matrix, s.o. beim
  // Anlegen der Funktion
  result = Kure_EvaluateTerm(relManager,
                             "ungleich(relResult,rel2)",
                             "testRelation");

  // Ausgabe ob Test wahr oder falsch ergab
  result = Kure_RelationGetNumberOfEntries(relManager,
                                           testRelation);

  if (result == 1)
    // wahr
    printf("%s", "Relationen␣sind␣ungleich");
  else
    // falsch
    printf("%s", "Relationen␣sind␣gleich");

  // Ergebnisrelation loeschen
  result = Kure_RelationDelete(relManager, relResult);
  relResult = NULL;
}

// Den RelManager beenden
Kure_Quit(relManager);
```

2.7 Die Callback-Methoden des funktionalen Kerns

In der Bibliothek des funktionalen Kerns von `RelView` werden drei Funktionen deklariert, die von den internen Funktionen der Bibliothek aufgerufen werden, um eventuelle Fehlermeldungen auszugeben und Rückfragen vorzunehmen. Diese Funktionen werden Callback-Funktionen genannt. Dies bedeutet hier, dass ein Programm eine Funktion der Bibliothek aufruft und die Bibliothek das aufrufende Programm innerhalb dieses Aufrufes zurück aufrufen kann. Diese Funktionen sind in der Bibliothek nicht implementiert. Wenn man die Bibliothek nutzen möchte und sie mit einem entwickelten Programm verlinkt, muss man diese drei Funktionen implementieren.

- `int Kure_Request (char * message);`

  Diese Funktion dient der Bibliothek dazu Fragen an den Benutzer, bzw. das Programm, welches auf die Bibliothek zugreift, zu stellen. Sie wird unter anderem dazu verwendet, eine Sicherheitsabfrage vorzunehmen, bevor eine Relation überschrieben wird. Abhängig von dem Rückgabewert der Funktion, reagiert die Bibliothek unterschiedlich. Welche Frage die Bibliothek stellt, wird als Zeichenkette übergeben. Eine Implementierung könnte zum Beispiel vorsehen, dem Benutzer die Meldung anzuzeigen um eine Auswahl zu treffen, die dann zurückgegeben wird.

  Als mögliche Rückgabewerte definiert die Bibliothek fünf Konstanten:

  `KURE_NOTICE_YES 1`

  `KURE_NOTICE_NO 2`

  `KURE_NOTICE_OVERWRITE_ALL 3`

  `KURE_NOTICE_CANCEL 4`

  `KURE_NOTICE_INVALID -10`

- `void Kure_PrintMessage (char * message);`

  Diese Funktion dient der Bibliothek dazu, Meldungen auszugeben. Dabei handelt es sich nicht um Fehler, sondern um Statusmeldungen, die als Zeichenkette übergeben werden.

- `void Kure_PrintError (char * message);`

  Diese Funktion dient der Bibliothek dazu, Fehlermeldungen auszugeben. Die Fehlermeldung wird als Zeichenkette übergeben.

2.8 Einbinden des funktionalen Kerns in C-Programme

Um ein C-Programm zu entwickeln, welches den funktionalen Kern von RELVIEW benutzt, muss man die Datei „kure.h“ einbinden, die beschriebenen Callback-Funktionen wie gewünscht implementierten und die Bibliothek des funktionalen Kerns verlinken (die Dateien sind als „KURE package“ über CVS bei `cvs.informatik.uni-kiel.de` erhältlich). Die Bibliothek selbst benötigt Bibliotheken des CUDD Package, welches unter [17] erhältlich ist. Somit kann man auf die exportierten Funktionen des funktionalen Kern innerhalb von C-Programmen zugreifen.

Kapitel 3

C-*JNI*-Adapterschicht

Dieses Kapitel beschreibt das *Java Native Interface* (JNI) für den Zugriff auf in C implementierte Funktionen, die Adapterschicht für das *JNI*-Namensschema und die Adapterschicht für Typkonvertierungen zwischen C und Java, um über diese Schichten und *JNI* auf die Funktionen des funktionalen Kerns von RELVIEW von Java-Anwendungen aus zugreifen zu können (siehe Abbildung 3.1 auf Seite 25).

3.1 Der Zugriff auf den funktionalen Kern

Um die C-Bibliothek mit dem funktionalen Kern von RELVIEW für Java-Programme nutzbar zu machen, muss eine Möglichkeit gefunden werden, zwischen den Programmiersprachen Java und C Funktionen aufzurufen. Eine direkte Interoperabilität von Java und C existiert leider nicht. Aufgrund der wünschenswerten Plattformunabhängigkeit bietet sich etwa die JavaBeans Bridge for ActiveX von Sun nicht an, die in [19] näher beschrieben ist. Denn um die JavaBeans Bridge nutzen zu können, muss man die zu nutzenden Funktionen über Microsoft Windows spezifische ActiveX-, COM- oder OLE-Komponenten zugänglich machen. Dies ist zwar möglich, indem man über das Adapterentwurfsmuster (vergleiche [11] für Entwurfsmuster) beispielsweise eine ActiveX-Komponente entwickelt, die Zugang zu den zu benutzenden Funktionen ermöglicht; allerdings würde dies ein Microsoft Windows System voraussetzen. Gleiches gilt für das in [15] beschriebene Java2COM. Mit Hilfe der Common Object Request Broker Architecture (CORBA) kann man Methoden anderer Programmiersprachen aufrufen. CORBA ist aber für verteilte Anwendungen und somit für Methodenaufrufe in verteilten Systemen gedacht und würde hier eine zusätzliche Belastung durch Netzwerkprotokolle bedeuten.

Das *Java Native Interface* (kurz *JNI*) ist besser geeignet, plattformunabhängig auf C-Funktionen zuzugreifen. Man kann in Java Methoden mit dem Schlüsselwort `native` definieren. Diese Methoden dürfen dann nicht implementiert werden, sind demnach im Prinzip wie abstrakte Methoden anzusehen. Bevor der Klassenlader der Java-Laufzeitumgebung Klassen lädt, die native Methoden definieren, muss die entsprechende native Bibliothek, welche die Implementation der Methoden erhält, geladen werden. Diese Implementation liegt im Falle des funktionalen Kerns in einer in C entwickelten Bibliothek vor. Zur Laufzeit werden dann beim Aufrufen der nativen Methoden durch *JNI* die Implementierungen, welche in der Bibliothek vorhanden sind, aufgerufen. *JNI* wird im nächsten Abschnitt näher erläutert.

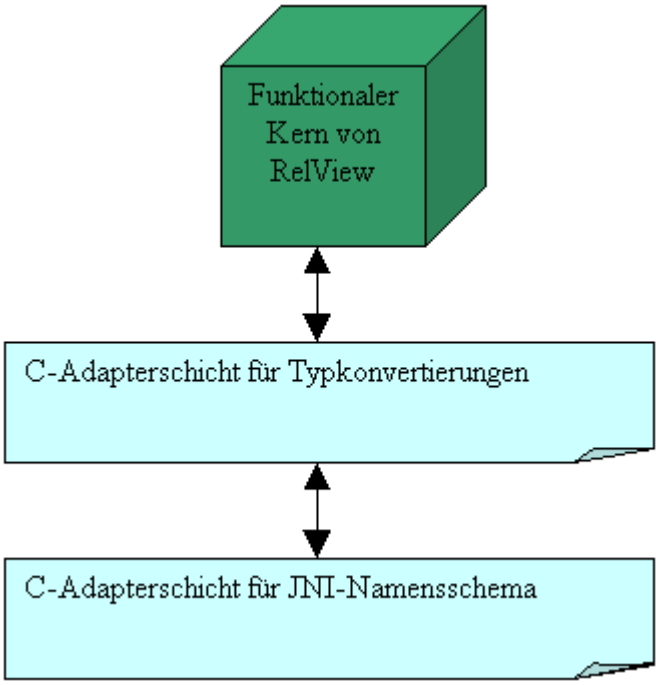

Abbildung 3.1: Die Abbildung zeigt die Adapterschichten vor dem funktionalen Kern von RELVIEW, um über das *Java Native Interface* auf den funktionalen Kern zugreifen zu können. Die Pfeile zeigen den Datenfluss.

3.2 Das *Java Native Interface*

Das *Java Native Interface* steht in Java für die Möglichkeit, Funktionen aus Bibliotheken fremder Programmiersprachen auszuführen. Bei diesen Funktionen handelt es sich um nativen Code, d.h. um betriebssystemabhängige Funktionen. *JNI* wird genutzt, um rechenintensive Algorithmen, die besser in C oder Assembler als in Java entwickelt werden sollten, in native Bibliotheken auszulagern. Man greift dann über *JNI* auf die Funktion dieser Bibliothek zu. Ebenso kann man *JNI* verwenden, wenn man bereits eine Bibliothek vorliegen hat, deren Funktionalität man benötigt. Statt die Bibliothek in Java neu zu implementieren, lässt sich *JNI* verwenden, um die Funktionalität der Bibliothek in einem Java-Programm zu benutzen. Auch kann man mit *JNI* Java-Anwendungen um Funktionen erweitern, welche man nicht mit Java implementieren kann, z.B. betriebssystemabhängige Befehle.

JNI ermöglicht von Java aus native Bibliotheksfunktionen aufzurufen und dient ebenso dazu, innerhalb von nativem Code Java-Funktionen nutzen zu können. Dafür besteht die Möglichkeit, in nativem Code Instanzen der *Java Virtual Machine* anzulegen und auf Java-Klassen zuzugreifen.

Native Funktionen, die man über *JNI* aus einem Java-Programm heraus aufrufen möchte, müssen einem besonderen Namensschema bei den Funktionsnamen genügen. Dieses Namensschema ist im folgenden Abschnitt näher beschrieben. Wenn eine fertige native Bibliothek vorliegt, unterliegen die Funktionsnamen nicht diesem Namenschema, es sei denn die Bibliothek wurde für den Einsatz unter *JNI* entwickelt. Wenn die Funktionen aufgrund fehlender Namensschemakonformität nicht über *JNI* direkt aufgerufen werden können, implementiert man eine native Adapterschicht um die Bibliothek, welche das Namensschema befolgt und Aufrufe an die entsprechenden Funktionen der Bibliothek

delegiert.

In *JNI* ist einer nativen Bibliothek üblicherweise in Java genau eine Klasse zugeordnet, in der die Funktionen der Bibliothek, auf die zugegriffen werden soll, als `native` Methoden deklariert werden. Auf die Funktionen kann dann über die Klasse zugegriffen werden. In einem Klassenkonstruktor wird die jeweilige Bibliothek über die `System.loadLibrary`-Methode geladen. Native Bibliotheken werden in Java nicht statisch gelinkt, sondern dynamisch geladen. *JNI* delegiert Aufrufe der nativen Methoden an die geladene Bibliothek. Dazu sucht *JNI* nach einer passenden Funktion in der geladenen Bibliothek, welche anhand des Namensschemas zu der aufgerufenen nativen Methode gehört. Existiert keine solche Funktion, wird beim Aufruf der nativen Methode eine Ausnahme geworfen.

Um zu verdeutlichen, wie man den Zugriff auf eine native Bibliothek mit *JNI* ermöglicht, wird im Abschnitt 3.2.2 auf Seite 27 das typische „HelloWorld“-Beispiel als *JNI*-Lösung beschrieben.

3.2.1 Namensschema für die Funktionen

Der Einsatz von *JNI* zwingt dazu, bei den in C aufzurufenden Funktionen ein von *JNI* vorgegebenes Namensschema einzuhalten, welches nachfolgend erläutert wird. Dieses Namensschema, welches die in RELVIEW enthaltenen Funktionen nicht befolgen, muss durch eine Adapterschicht in C realisiert werden. Um die Funktionen des funktionalen Kerns von RELVIEW für Java über *JNI* nutzbar zu machen, wurde für jede der Funktionen eine entsprechende Adapterfunktion entwickelt, welche dem *JNI*-Namensschema unterliegt und als Adapter zu der eigentlichen Funktion des Kerns dient. Diese Adapterfunktionen werden in der Datei `kurejava.h` deklariert und in der Datei `kurejava.c` implementiert. Diese Adapterfunktionen bilden die Adapterschicht für das Namensschema. Jeder der Funktionen des funktionalen Kerns muss dabei gemäß der *JNI*-Spezifikation ein bestimmter Namenspräfix vorangestellt werden. Der Name jeder nativen Funktion, welche über *JNI* aufgerufen wird, muss mit `Java_` beginnen. Danach folgt der Packetpfad der Javaklasse, in der die nativen Funktionen deklariert werden und anschließend der Klassenname. In dem Paketpfad samt Klassennamen werden Unterstriche zur Trennung der einzelnen Pfadbestandteile genutzt.

Den Adapterfunktionen wird daher als Namenspräfix

```
Java_de_osz_kurejava_Kurejava_
```

vorangestellt, um namenskonform zu *JNI* zu sein. Der Namenspräfixpart `de_osz_kurejava` entspricht dem Java-Paketpfad `de.osz.kurejava`. Dieser Paketpfad wurde konform zur in Java üblichen Angabe von Paketen gewählt, wobei das Paket `de` verdeutlicht, dass dieses Projekt in Deutschland entwickelt wird. Der nächste Paketteil steht in Java üblich für die verantwortliche Organisation, Firma oder Person, die ein Paket entwickelt. Bei `osz` handelt es sich dabei um das Kürzel des Entwicklers. Der letzte Paketbestandteil steht für den Projektnamen, welcher hier als `kurejava` gewählt wurde. Dieser Projektname hat sich ergeben, weil innerhalb des Projektes Zugriff auf die `KURE`-Bibliothek von Java aus ermöglicht wird. Der letzte Teil des Namenspräfixes (`_Kurejava_`)nach dem Paketpfad bedeutet, dass die Adapterfunktionen in Java in einer Klasse `Kurejava` im Paket `de.osz.kurejava` als `native` deklariert werden müssen.

Allgemein gilt für `native` Funktionen, die über *JNI* aufgerufen werden, folgendes Deklarationsschema:

```
JNIEXPORT void JNICALL
```

```
Java_de_osz_kurejava_Kurejava_FUNCTIONNAME
```

Statt des Schlüsselwortes `void`, das Funktionen ohne Rückgabewert kennzeichnet, sind auch *JNI*-Datentypen als Rückgabewerte erlaubt. Diese *JNI*-Datentypen werden nachfolgend beschrieben. Anschließend folgen die funktionsabhängigen Parameter. Die beiden Schlüsselworte `JNIEXPORT` und `JNICALL` sind Macros, die *JNI* definiert und welche die *JNI*-Spezifikation erfordert. Die Deklaration der einzelnen Adapterfunktionen wird in Abschnitt 3.6 auf Seite 31 beschrieben.

3.2.2 `HelloWorld`-Beispiel

Im Folgenden wird ein Beispiel für den Zugriff auf Methoden einer nativen Bibliothek gegeben. Für dieses Beispiel wird angenommen, dass eine native Bibliothek namens `HelloWorldNativeLib` existiert, welche die Funktion `printHelloWorld` zur Verfügung stellt. Diese Funktion gibt den Text `HelloWorld` aus und soll in einer Java-Anwendung genutzt werden.

Nun wird in Java eine Klasse entwickelt, welche die Funktionen der Bibliothek als native Klassenmethoden zur Verfügung stellt. Eine Klasse für das `HelloWorld`-Beispiel kann wie folgt implementiert werden:

```
class HelloWorld {

  // native Methode
  public static native void printHelloWorld();

  // Klassenkonstruktor
  static {
    System.loadLibrary("HelloWorldJNILib");
  }

  // Programmstart
  public static void main(String[] args) {
    printHelloWorld();
  }

}
```

Beim ersten Zugriff auf die Klasse innerhalb einer Java-Anwendung wird der Klassenkonstruktor ausgeführt und die Bibliothek namens `HelloWorldJNILib` geladen, die nachfolgend beschrieben wird. Die Methode `loadLibrary` sucht dabei je nach Betriebssystem nach Dateien mit der Endung `.dll` unter Windows, bzw. `.so` unter SunSolaris/Linux in je nach Betriebssystem konfigurierten Bibliothekspfaden. Beim Aufruf der `main`-Methode, wird die native Methode `printHelloWorld` aufgerufen und *JNI* versucht in den geladenen Bibliotheken eine Funktion zu finden, deren Namen nach dem *JNI*-Namensschema zu der Funktion passt. Hierbei handelt es sich um die Funktion `Java_HelloWorld_printHelloWorld`, welche in der Bibliothek `HelloWorldJNILib` implementiert sein muss.

Hat man eine Java-Klasse, welche die Funktion einer nativen Bibliothek kapselt, entwickelt, und muss man noch die Adapterschicht implementieren, kann man dies entweder direkt tun oder sich von dem Hilfsprogramm `javah` aus dem *Java Developement Kit* eine C-Headerdatei generieren lassen. Diese Headerdatei implementiert man in einer C-Quellcodedatei.

Die von dem Programm `javah` generierte C-Headerdatei `HelloWorld.h` für die oben angegebene Beispielklasse sieht wie folgt aus:

```
/* DO NOT EDIT THIS FILE - it is machine generated */

#include <jni.h> /* Header for class HelloWorld */

#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef

__cplusplus extern "C" { #endif /*
 * Class:     HelloWorld
 * Method:    printHelloWorld
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloWorld_printHelloWorld
  (JNIEnv *, jclass);
#ifdef __cplusplus }
#endif
#endif
```

Die Adapterschicht kann man in einer Datei `HelloWorld.c` wie folgt implementieren:

```
#include <jni.h>
#include "HelloWorld.h"

JNIEXPORT void JNICALL
  Java_HelloWorld_printHelloWorld
    (JNIEnv *env, jclass class) {
      printHelloWorld();
  }
```

Im Beispiel handelt es sich bei `JNIEXPORT` und `JNICALL` um Makros, die von *JNI* definiert werden, und Funktionen kennzeichnen, die über *JNI* aufgerufen werden können. In der Headerdatei `jni.h` definiert *JNI* spezifische Informationen, unter anderem Makros und Datentypen. Die zwei angegebenen Parameter muss jede über *JNI* verfügbare Funktion als erste Parameter besitzen, auch wenn diese zwei Parameter nicht im Java-Quellcode bei den als `native` deklarierten Methoden angegeben sind. Diese Parameter ermöglichen den Zugriff auf die Java-Umgebung und die Java-Klasse, in welcher die Funktionen als `native` deklariert wurden.

Die Datei `HelloWorld.c` muss als dynamische Bibliothek mit dem Namen `HelloWorldJNILib` kompiliert und die Bibliothek `HelloWorldNativeLib` mit der eigentlichen Implementierung der `printHelloWorld`-Funktion dazugelinkt werden. Legt man die Bibliothek `HelloWorld` dann unter einem Pfad ab, der als Bibliotheksverzeichnis des Betriebssystems konfiguriert ist, kann die Java-Anwendung über die `main`-Methode der Klasse `HelloWorld` ausgeführt werden.

3.2.3 Parameterübergabe und Rückgabewerte in *JNI*

Beliebige Parameter und Rückgabewerte, die mit dem Schlüsselwort `native` deklarierten Methoden in Java besitzen, werden auf spezifische *JNI*-Datentypen abgebildet und folgen in den Funktionsde-

klarationen im nativen Code den beiden bereits genannten *JNI*-Parametern, die eine Funktion immer erhält. Dabei werden die Datentypen in Java wie folgt abgebildet:

Tabelle 3.1: Java- und *JNI*-Datentypen

`boolean`	`jboolean`
`byte`	`jbyte`
`short`	`jshort`
`char`	`jchar`
`int`	`jint`
`long`	`jlong`
`float`	`jfloat`
`double`	`jdouble`
`String`	`jstring`
`Object`	`jobject`
`void`	`void`
`byte[]`	`jbyteArray`

Diese Datentypen lassen sich also direkt im nativen Quellcode benutzen und sind äquivalent zu den entsprechenden Datentypen in Java. Dabei ist zu beachten, dass alle Objekte aus Java im nativen Code als `jobject` angesprochen werden. Im Folgenden wird ein Beispiel angegeben, welches eine native Methode mit Parametern zeigt:

```
class Test {
  public static native int sum (int a, int b);

  static {
    // entsprechende Bibliothek laden
  }
}
```

Hier eine Beispielimplementierung der Methode `sum`, in der die Werte der beiden Parameter `a` und `b` addiert und das Ergebnis zurückgeliefert wird:

```
JNIEXPORT jint JNICALL
  Java_Test_sum
    (JNIEnv *env, jclass class, jint a, jint b) {
      return a + b;
  }
```

3.3 Methoden zur Konvertierung von Datentypen

Zusätzlich zu der Adapterschicht für das *JNI*-Namensschema ist eine Adapterschicht für notwendige Datentypkonvertierungen zwischen C und Java erforderlich (siehe Abbildung 3.1 auf Seite 25). Diese Adapterschicht ist zwingend erforderlich, da für den Aufruf von C-Funktionen aus Java Parameter übergeben werden, die dazu in einen in C gültigen Datentypen konvertiert werden müssen, ebenso Rückgabewerte in einen in Java gültigen Datentypen. Die Funktionalität der Adapterschicht für Typkonvertierungen befindet sich in der Implementation der *JNI*-Namensschema-Adapterfunktionen.

In diesen Funktionen werden Parameter- und Rückgabetypen zwischen Datentypen, die von *JNI* definiert sind und Datentypen aus C konvertiert sowie Funktionen des funktionalen Kerns von REL-VIEW aufgerufen. Da die Adapterschicht für das Namensschema nur aus der richtigen Deklaration der C-Funktionen besteht, bietet es sich an, die Konvertierungen in diesen Funktionen zu implementieren und somit beide Adapterschichten durch dieselben Funktionen abzudecken. Die zweite Adapterschicht in neuen Funktionen zu implementieren, hätte nur eine weitere Funktionsdelegation bedeutet. Für die Implementierung der Adapterschicht für Typkonvertierung wurden vier Hilfsfunktionen in C entwickelt, welche Typkonvertierungen vornehmen und in den anderen Funktionen genutzt werden.

- `jstring charToJString(const char *message)`

  Diese Funktion konvertiert eine C-Zeichenkette in den Datentyp `jstring`.

- `char* jstringToChar(const jstring message)`

  Diese Funktion konvertiert einen Parameter vom Typ `jstring`-Datentyp in eine C-Zeichenkette und gibt diese zurück.

- `jbyteArray convertPointerToByteArray(const void* pointer)`

  Diese Funktion konvertiert einen beliebigen Zeiger in den Datentyp `jbyteArray`.

- `void* convertByteArrayToPointer(const jbyteArray arr)`

  Diese Funktion konvertiert einen Parameter vom Typ `jbyteArray` in einen Zeiger und gibt diesen zurück.

3.4 Callback-Funktionen nach Java

Um innerhalb der nativen C-Bibliotheken Methoden der Java-Anwendung aufrufen zu können, unter anderem für Debug-Meldungen, wurden zwei Funktionen entwickelt, welche statische Methoden der zu entwickelnde Java-Klasse `Kurejava` des Paketes `de.osz.kurejava` aufrufen. Mit Hilfe der beiden Funktionen ist es möglich, der anzubindenden Java-Anwendung Ereignisse mitzuteilen. Die native Bibliothek kann beispielsweise beim Überschreiben von Relationen nachfragen, ob dies wirklich geschehen soll. Die beiden Funktionen werden zum Delegieren der Callback-Funktionen des funktionalen Kerns und für die Ausgabe von Debug-Meldungen in der Java-Anwendung benötigt. Wenn man aus einer Anwendung Funktionen einer Bibliothek aufruft, und in diesem Aufrufkontext von der Bibliothek aus Methoden der Anwendung aufgerufen werden, nennt man dies Rückruf, bzw. Callback. Daher werden die Funktionen als Callback-Funktionen bezeichnet.

- `void command(const jstring str, const jobject param)`

  Diese Funktion ruft die statische Methode `command` mit dem Rückgabetyp `void` der Klasse `Kurejava` im Paket `de.osz.kurejava` auf. Der Parameter `str` soll dabei die Art des auszuführenden Befehls angeben, der Parameter `param` ist der Parameter des Befehls. Somit lassen sich beliebige Kommandos an Java-Anwendungen übermitteln. Ein Kommando besteht dabei aus einem Kommandonamen und Kommandoparametern. Die möglichen Befehle hängen von der Implementierung der Java-Klasse `Kurejava` des Paketes `de.osz.kurejava` ab, indem in der Implementierung entschieden wird, was bei welchem übergebenen Kommandonamen und Kommandoparametern ausgeführt wird.

- `int commandAndAcknowledge(const jstring str,`
  `const jobject param)`

  Diese Funktion ruft die statische Methode `commandAndAcknowledge` mit dem Rückgabetyp `int` der Klasse `Kurejava` im Paket `de.osz.kurejava` auf. Diese Methode erweitert die oben angegebene Methode `command` um Rückgabewerte aus der Java-Anwendung zu ermöglichen. Falls die native Bibliothek aufgerufen wird um eine Relation zu löschen, kann über diese Methode nachgefragt werden, ob der Löschvorgang wirklich ausgeführt werden soll. Für diesen Fall kann man z.B. ein Kommando mit Namen `overwriteRelation` definieren, und den Namen der zu überschreibenden Relation als Parameter übermitteln. Die Implementierung der Java-Klasse `Kurejava` könnte auf dieses Kommando so reagieren, dass sie ein Fenster öffnet und darin dem Benutzer ermöglicht, das Löschen zu bestätigen oder den Vorgang abzubrechen. Diese Methode wurde vorgesehen, um die Funktionalität der Bibliothek bei Bedarf erweitern zu können.

3.5 Callback-Funktionen des funktionalen Kerns von RELVIEW

Der funktionale Kern von RELVIEW definiert drei Callback-Funktionen, welche implementiert werden müssen, wenn man die exportierten Funktionen des funktionalen Kerns benutzen möchte. Diese drei Funktionen wurden so implementiert, dass sie die Java-Callback-Funktionen, welche in Abschnitt 3.4 auf Seite 30 beschrieben wurden, nutzen. Damit wird dem funktionalen Kern ermöglicht, Rückfragen an den Benutzer zu stellen und Meldungen an die angebundene Java-Anwendung auszugeben.

3.6 Deklaration und Implementierung der Adapterfunktionen

Die zwei Adapterschichten für das *JNI*-Namensschema und die Datentypkonvertierungen wurden in der C-Headerdatei `kurejava.h` und der C-Quellcodedatei `kurejava.c` realisiert (siehe Abbildung 3.2 auf Seite 32). Die nach *JNI*-Standard genormten Funktionsnamen und Parametertypen bilden die C-Adapterschicht für *JNI*, während die C-Adapterschicht für Typkonvertierungen aus der Implementierung dieser Funktionen besteht. Die Deklaration der Adapterfunktionen ergibt sich also aus der *JNI*-Spezifikation. Ihre Namen bestehen aus dem *JNI*-Namenspräfix und dem Namen der C-Funktion aus RELVIEW, welche sie kapseln. Die Implementierung der Adapterfunktionen besteht aus Typkonvertierungen der Parameter- und Rückgabetypen durch die vier Hilfsfunktionen (siehe Abschnitt 3.3 auf Seite 29) und dem Delegieren von Aufrufen an den funktionalen Kern von RELVIEW. Daher sind die Adapterfunktionen hier im Einzelnen nicht aufgelistet, sondern werden im Anhang A beschrieben. Zusätzlich zu reinen Adapterfunktionen wurden drei weitere Funktionen implementiert, die zum Initialisieren und Beenden des Zugriffs auf den funktionalen Kern benötigt werden. Diese werden im Folgenden erläutert:

Alle über *JNI* aufrufbaren Funktionen erhalten nach der *JNI*-Spezifikation als erstes die Datentypen `JNIEnv` und `jclass`. Bei dem Datentyp `JNIEnv` handelt es sich um eine Sammlung von Zeigern auf Funktionen, die ermöglichen in C auf die Java-Umgebung, also die Java-Klassen, ihre Methoden sowie ihre Eigenschaften, zuzugreifen. Man bezeichnet den Datentypen auch als Funktionszeigertabelle. Hauptsächlich benötigt man diese Funktionszeigertabelle um in C Java-Methoden aufzurufen.

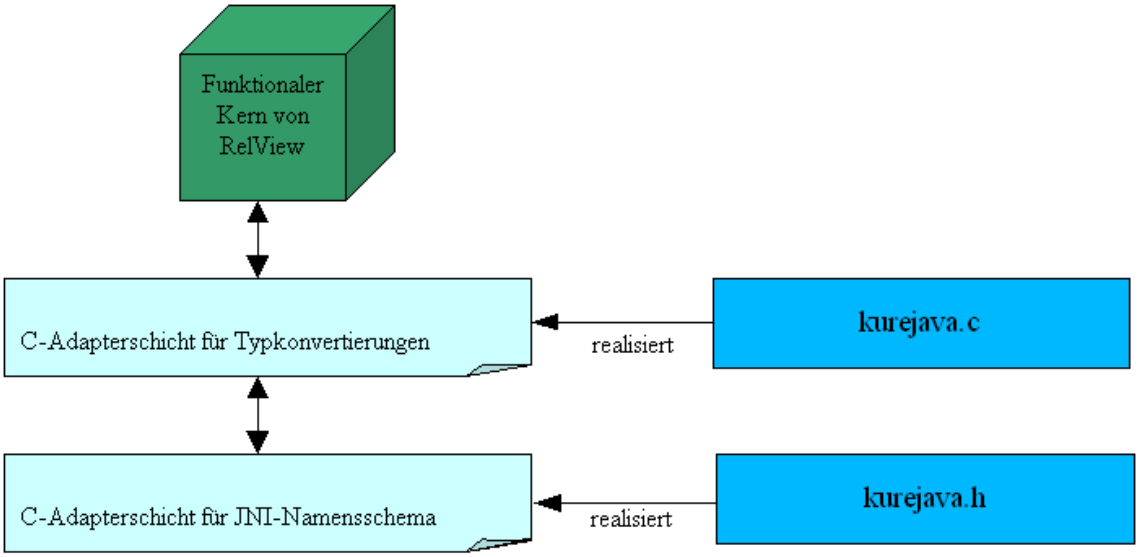

Abbildung 3.2: Die Abbildung zeigt die Realisierung der Adapterschichten, zum einen durch die Namenskonvention der in der Datei `kurejava.h` angegebenen Funktionen, zum anderen durch die Implementierung dieser Funktionen in der Datei `kurejava.c`.

Der Datentyp `jclass` kapselt Informationen zu der Java-Klasse, in welcher die native Methode deklariert ist.

Folgende aufgelistete Funktionen besitzen alle das Präfix (vgl. 3.2.1 auf Seite 26):

`Java_de_osz_kurejava_Kurejava_`

Dieser Präfix ist zur Erfüllung des *JNI*-Namensschemas unbedingt notwendig, wird aber zur besseren Lesbarkeit im folgenden nicht aufgeführt.

- `JNIEXPORT void JNICALL init(JNIEnv* env, jclass cl)`

  Diese Funktion speichert die in *JNI* spezifizierten Parameter in globalen Variablen, damit sie in den Konvertierungs- und den Java-Callback-Funktionen genutzt werden können, um Java-spezifische Befehle auszuführen. Diese Funktion muss vor allen anderen aufgerufen werden, wenn der funktionale Kern von RELVIEW über *JNI* genutzt werden soll.

- `JNIEXPORT void JNICALL quit(JNIEnv* env, jclass cl)`

  Diese Funktion gibt die in `init` gesicherten Variablen frei und muss aufgerufen werden, wenn die Arbeit mit dem funktionalen Kern von RELVIEW über *JNI* beendet ist.

- ```
  JNIEXPORT void JNICALL setDebug
    (JNIEnv* env, jclass cl, jboolean value)
  ```

  Diese Funktion ermöglicht, den Debugmodus der Schnittstelle zu aktivieren und zu deaktivieren. Dies geschieht über die Angabe des dritten Parameters vom Datentyp `jboolean`.

Mit Hilfe der Adapterfunktionen kann eine Java-Klasse entwickelt werden, die gleichbenannte Methoden (ohne das Namenspräfix) mit dem Schlüsselwort `native` deklarieren, so dass über *JNI* die
  ```

hier beschriebenen Funktionen genau dann aufgerufen werden, wenn in einer Java-Anwendung die nativen Methoden aufgerufen werden. Dies ist im folgenden Kapitel 4 mit den nativen Methodenaufrufen erläutert. Die C-Adapterschichten wurden so implementiert, dass ein Zugriff auf die exportierten Funktionen des funktionalen Kerns von Java aus möglich ist.

3.7 Erstellung einer nativen Bibliothek für *JNI*

Die beiden Adapterschichten und die zusätzlichen Methoden wurden in den Dateien `kurejava.h` und `kurejava.c` implementiert. Um die Dateien zu kompilieren, muss in der Datei `kurejava.c` die Datei `kurelib.h`, welche zu RELVIEW gehört, eingebunden werden, um Zugriff auf den funktionalen Kern von RELVIEW zu ermöglichen.

Um nun die Bibliothek, welche den funktionalen Kern von RELVIEW enthält, in Java über *JNI* nutzen zu können, muss man die Datei `kurejava.c` als dynamische Bibliothek kompilieren und die native Bibliothek des funktionalen Kerns dazu linken. Dies geht mit einem beliebigen Übersetzer für die Sprache C. Die durch diesen Vorgang erhaltene native Bibliothek kann dann für den Zugriff von Java über *JNI* genutzt werden.

Die für den Zugriff auf diese Bibliothek benötigte Klasse `Kurejava` enthält die nativen Methodendeklaration der Funktionen; ihre Implementierung liegt in der entwickelten nativen Bibliothek. Die Klasse stellt eine weitere Adapterschicht in Java zu den C-Adapterschichten dar (siehe Abbildung 3.3 auf Seite 34). Die Klasse ist in Abbildung 4.4 auf Seite 42 abgebildet.

Folgende statische Methoden sind direkt in der Klasse implementiert und dienen dazu Ausgaben von dem funktionalen Kern über die Callback-Funktionen an Java zu ermöglichen und die native Bibliothek zu laden:

- ```
  static void setLogPrintStream
    (PrintStream lognative)
  ```

  Mit einem Aufruf dieser Methode kann man den Ausgabestrom für Debug-Meldungen angeben. Die Java-Standardausgabe wird verwendet, wenn die Methode nicht aufgerufen wird.

- `static void loadLibrary(File library)`

  Diese Methode lädt die mit dem Parameter `library` angegebene native Bibliothek. Bei diesem Parameter handelt es sich um ein Objekt, das eine Datei repräsentiert. Die native Bibliothek enthält die Implementierung der nativen Methoden dieser Klasse. Es handelt sich dabei um eine wie oben beschrieben zu erstellende native Bibliothek.

- `static void loadLibrary(String libraryname)`

  Methode ist analog zur gleichnamigen oben beschriebenen Methode. Hier wird die native Bibliothek über den absoluten Dateinamen angegeben.

- ```
  private static void command
    (String type, Object cmd)
  ```

  Dies ist die Implementierung einer der zwei Methoden, welche die native Bibliothek benötigt, um Meldungen auszugeben. Eintreffende Meldungen werden auf den mit der Methode `setLogPrintStream` angegebenen Ausgabestrom ausgegeben.

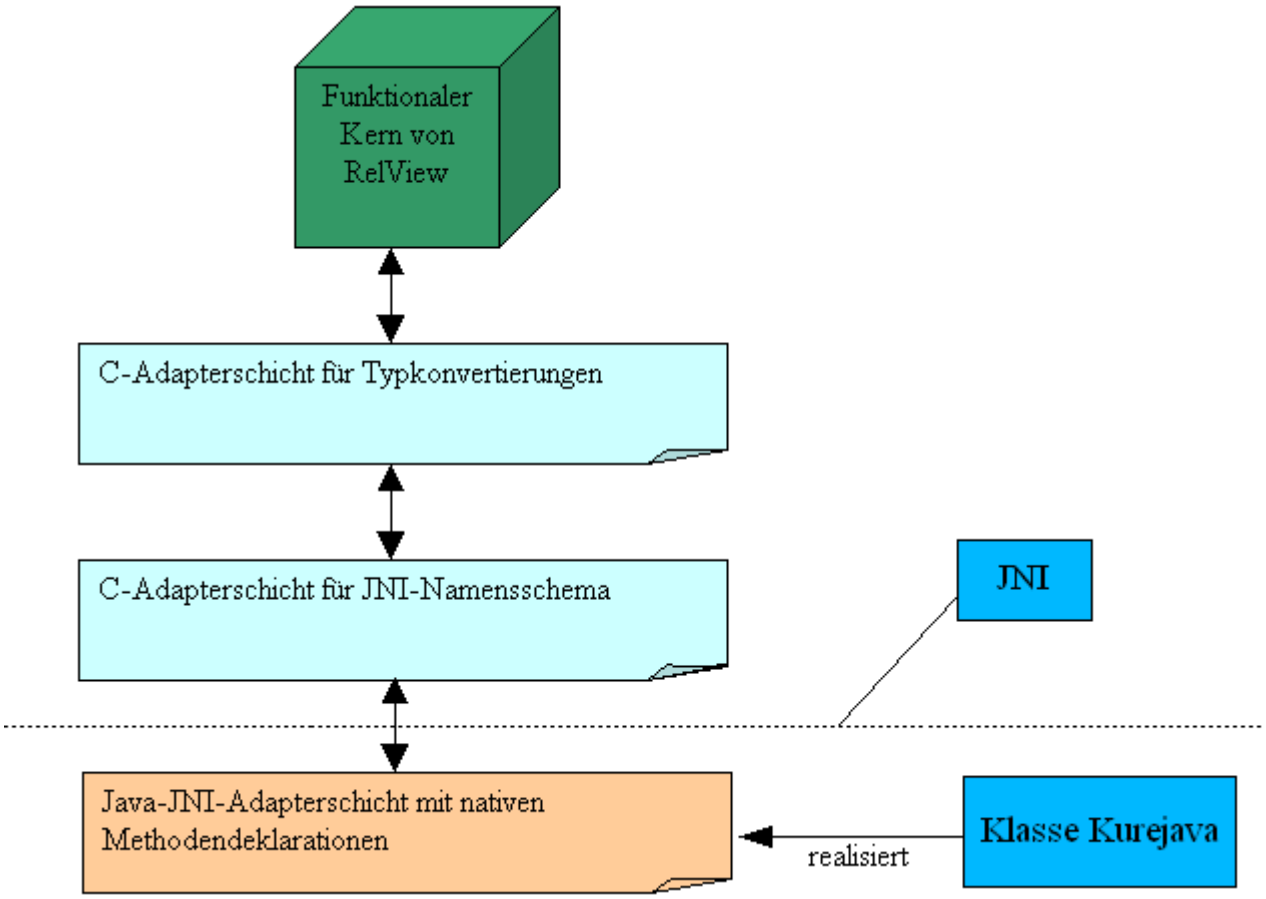

Abbildung 3.3: Die JNI-Adapterschicht in Java (unterste Schicht), welche auf die C-Adapterschichten zugreift und somit Zugriff auf die exportierten Funktionen des funktionalen Kerns ermöglicht. Diese Schicht wird durch die Klasse `Kurejava` realisiert.

- `private static int commandAndAcknowledge (String type, Object cmd)`

  Analog zur der Methode `command`. Diese Methode gibt immer `0` zurück. Die Rückgabe `0` bedeutet, dass die eintreffende Nachricht der nativen Bibliothek bestätigt wird. In späteren Versionen ist es denkbar, dass hier unterschiedliche Rückgaben möglich sind.

Die Klasse `Kurejava` ist somit direkt an die native Bibliothek gekoppelt und ermöglicht den Zugriff auf den funktionalen Kern von RELVIEW (siehe Abbildung 4.4 auf Seite 42).

Kapitel 4

Java-Adapterschicht

Dieses Kapitel beschreibt die Adapterschicht in Java, welche auf die nativen Funktionen aus dem Kapitel 3 zugreift, in Java den Zugriff auf diese Funktionen über Objekte erlaubt und statt einfacher Fehlercodes eine Fehlerbehandlung durch Ausnahmen über Java-Exceptions ermöglicht (siehe Abbildung 4.1 auf Seite 37). Wird im Folgenden die Bezeichnung „Bibliothek“ verwendet, ist damit die Java-API für Relationale Algebra gemeint, welche innerhalb dieser Arbeit entwickelt wurde. Wird dem gegenüber explizit von „nativer Bibliothek“ gesprochen, sind damit der funktionale Kern von RELVIEW und die C-*JNI*-Adapterfunktionen gemeint. Eine Java-Anwendung, welche die Bibliothek nutzt, wird im Folgenden als Client-Anwendung bezeichnet.

4.1 Die Java-Adapterschicht

Bislang ist es möglich über die nativen Methodendeklarationen der Klasse `Kurejava` und die Implementierung der Funktionen in Form einer nativen Bibliothek auf den funktionalen Kern von RELVIEW zuzugreifen. Diese Methoden sind alle statisch und benutzen einfach Datentypen als Parameter- und Rückgabetypen. Fehler werden durch die Rückgabewerte der Funktionen übermittelt. In Java ist es wünschenswert statt mit einfachen Datentypen mit Objekten, und statt mit Fehlermeldungen durch Rückgaben mit Java-Ausnahmen zu arbeiten. Die Schnittstelle für den Zugriff auf den funktionalen Kern sollte somit für eine Java-Anwendung Objekte als Parameter und Rückgaben, sowie Ausnahmen zur Fehlerbehandlung ermöglichen. Die bereits vorliegende Schnittstelle sieht dies jedoch nicht vor. Um die erwartete Schnittstelle anzubieten und die bereits vorliegende Schnittstelle zu benutzen, ist es folglich notwendig, nicht nur in der C-Bibliothek das Adapterentwurfsmuster zu verwenden, sondern in Java ein entsprechendes Gegenstück zu entwickeln, welches über die Klasse `Kurejava` auf die Adapter in der C-Bibliothek zugreift und die Rückgaben der Funktionen auswertet, um bei aufgetretenen Fehlern eventuell Ausnahmen auszulösen. Auch müssen die einfachen Parameter, wie z.B. `Byte-Arrays`, welche Pointer auf der Java-Seite darstellen, in Objekten gekapselt werden, um objektorientierten Umgang mit der Bibliothek zu ermöglichen. Diese Java-Adapterschicht wurde in der Klasse `KurejavaOO` realisiert. Das Adapterentwurfsmuster dient dazu, das Problem von inkompatiblen Schnittstellen (die erwartete Schnittstelle ist anders als die vorliegende) zu lösen. Bei der vorliegenden Lösung handelt es sich um eine Umsetzung des Adapterentwurfsmuster, da die Schicht aus einer Menge von Methoden besteht, die zwar Objektorientierung und Java-Ausnahmebehandlung

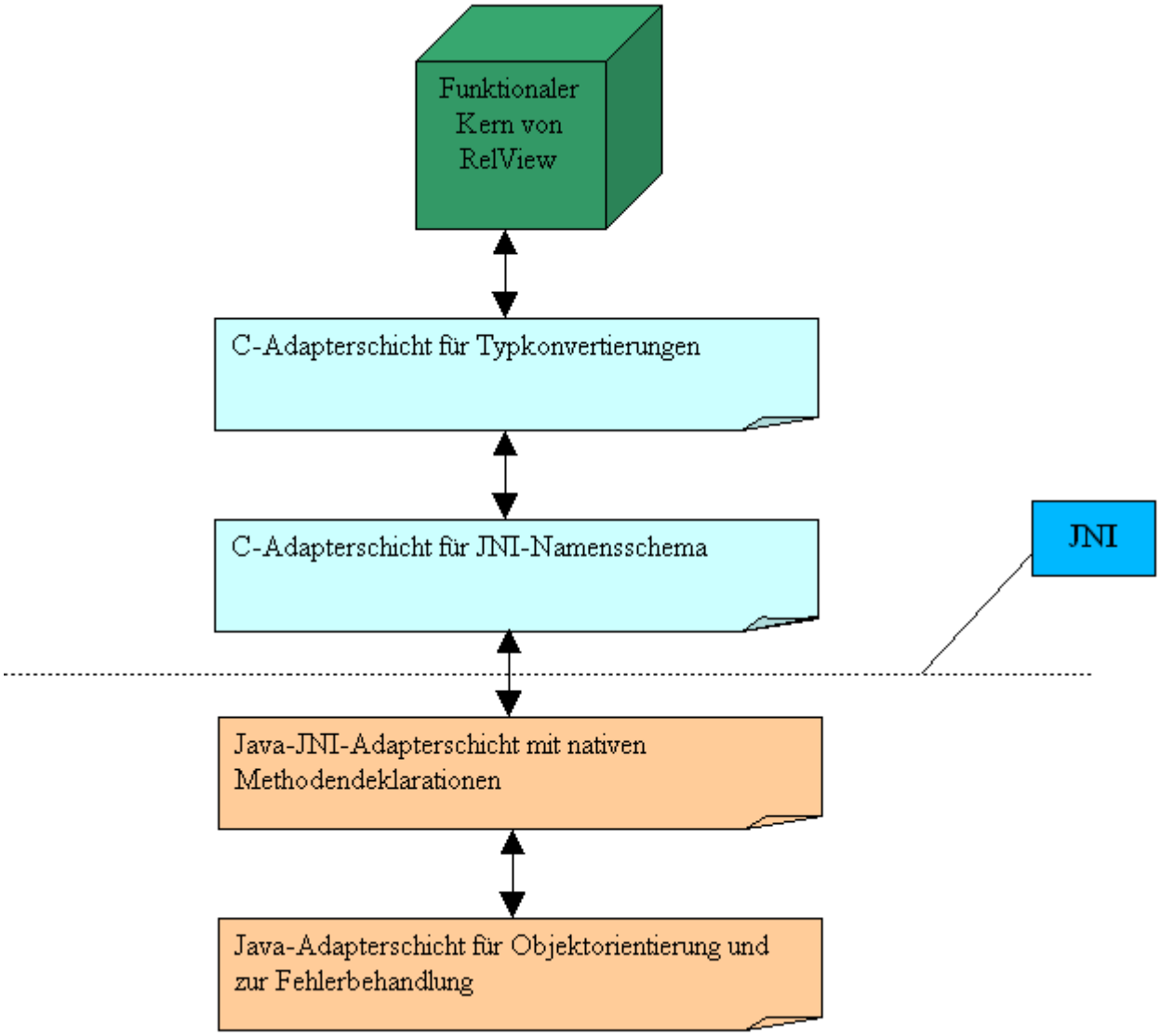

Abbildung 4.1: Die Java-Adapterschicht ist die unterste Schicht in diesem Diagramm. Die oberen zwei Schichten vor dem funktionalen Kern stellen die Adapterschichten in C dar, die dritte ist die zu den C-Adapterschichten direkt gehörende JNI-Adapterschicht in Java; alle drei wurden im letzten Kapitel erläutert.

ermöglicht und damit die erwartete Schnittstelle einer Client-Anwendung anbietet, aber keine neue fachliche Funktionalität beinhaltet, sondern lediglich die Funktionalität der C-Bibliothek durch die bereits vorliegende Schnittstelle nutzt. Ohne sie wäre eine komfortable objektorientierte Handhabung mit Ausnahmen bei aufgetretenen Fehlern nicht möglich. Eine Alternative zu dem Adapterentwurfsmuster gibt es nicht, wenn man nicht in der Java-Client-Anwendung auf die Nutzung von Objektorientierung und Ausnahmebehandlung für den Umgang mit der Bibliothek verzichten möchte. Die Klasse besteht aus einer Sammlung statischer Methoden, die eineindeutig den nativen Methoden der Klasse `Kurejava` zugeordnet sind, sowie drei zusätzlicher Methoden, die alle im Folgenden näher erläutert werden. Für den objektorientierten Zugriff auf die nativen Funktionen wurden die zwei Klassen `RelationImpl` und `RelManagerImpl` als Datenhalter für C-Zeiger mit den zugehörigen Java-Schnittstellen `Relation` und `RelManager` entwickelt, welche die aus den nativen Funktionen in Form von `Byte-Arrays` zurückgelieferten C-Zeiger in privaten Objektattributen kapselt. Für die Fehlerbehandlung durch Ausnahmen wurden die Java-Exceptions (vgl. Abbildung 4.4 auf Seite 42)

- `NotInitializedException`,
- `AlreadyInitializedException`,
- `FunctionException`,
- `ProgramException`,
- `RelationException`,
- `RelManagerException` und
- `TermException`

entwickelt. Die genannten Klassen und Schnittstellen gehören zu dem Java-Paket

`de.osz.kurejava`

und werden nachfolgend näher erläutert.

4.2 Die Datenhalter `RelManagerImpl` und `RelationImpl`

Die Klasse `RelManagerImpl` dient zum objektorientierten Kapseln von C-Zeigern, die aus der nativen Bibliothek in Form von `Byte-Arrays` geliefert werden. Dazu wird in dieser Klasse ein Objektattribut definiert, welches den C-Zeiger in Form eines `Byte-Arrays` zwischenspeichern kann. Klassen im Java-Paket `de.osz.kurejava` können direkt auf dieses Objektattribut zugreifen, es also auslesen, manipulieren und schreiben. Dadurch kann die in der Klasse `KureJavaOO` implementierte Adapterschicht mit Objekten dieses Typen arbeiten und beim Aufruf nativer Funktionen den eigentlichen C-Zeiger als `Byte-Array` übergeben. Die Klasse `RelationImpl` speichert C-Zeiger auf die gleiche Art und kapselt dabei Zeiger auf Relationen. Beide Klassen sind wichtig, um in Java-Anwendungen mit Objekten arbeiten zu können. Die Schnittstellen `RelManager` und `Relation`, welche von den zwei genannten Klassen implementiert werden, definieren zusätzliche Methoden. Mit Hilfe dieser Methoden ist es Java-Anwendungen, welche die Bibliothek nutzen wollen, möglich, vollständig über die Instanzen, die Relationsmanager und Relationen darstellen, auf die

Funktionen der Bibliothek zuzugreifen. Die Java-Anwendungen müssen selbst keine Methoden der Klasse `KurejavaOO` aufrufen. Diese Schnittstellen und die Methoden werden im nachfolgenden Kapitel 5 erläutert und in diesem Kapitel vernachlässigt.

4.3 Die Ausnahmeklassen

Die Ausnahmeklassen sind in der Abbildung 4.4 auf Seite 42 gezeigt. Die Ausnahmeobjekte der `Exception`-Klassen

- `NotInitializedException`,
- `AlreadyInitializedException`,
- `FunctionException`,
- `ProgramException`,
- `RelationException`,
- `RelManagerException` und
- `TermException`

werden im jeweiligen Fehlerfall bei Methoden der Klasse `KurejavaOO` geworfen. Die einzelnen Fehlerfälle werden nachfolgend noch erläutert. Dadurch ist eine Fehlerbehandlung durch Ausnahmen, wie in Java gewohnt, beim Benutzen der Bibliothek möglich. Alle entwickelten Fehlerklassen wurden direkt von der Klasse `Exception` des Java-Frameworks abgeleitet. Jeder der Ausnahmen beinhaltet eine textuelle Fehlermeldung als `String`, welche vor dem Werfen einer Ausnahme gesetzt wird und mit der Methode `getMessage()` des Ausnahmeobjektes abgefragt werden kann. Dadurch ist es möglich, dem Benutzer eine Fehlermeldung anzuzeigen oder diese zu Debugzwecken zu protokollieren.

Im Folgenden wird kurz erläutert, was das Auftreten von Ausnahmen der einzelnen Klassen bedeutet.

- `NotInitializedException`

  Diese Ausnahme tritt auf, wenn Funktionen der nativen Bibliothek genutzt werden sollen, die Bibliothek allerdings nicht vorher initialisiert wurde.

- `AlreadyInitializedException`

  Diese Ausnahme tritt auf, wenn die native Bibliothek bereits initialisiert wurde, und dies erneut geschehen soll. Die Bibliothek wird dann nicht erneut initialisiert.

- `FunctionException`

  Diese Ausnahme tritt auf, wenn eine relationale Funktion definiert wird und bei der Definition Fehler auftreten.

- `ProgramException`

  Diese Ausnahme tritt auf, wenn ein relationales Programm definiert wird und bei der Definition Fehler auftreten.

- `RelationException`

  Diese Ausnahme wird beim Anlegen oder Abfragen einer Relation geworfen, falls dabei Fehler auftreten. Dies kann u.a. geschehen, wenn die Relation nicht angelegt werden konnte oder die Relation, die abgefragt werden soll, nicht existiert.

- `RelManagerException`

  Diese Ausnahme tritt auf, wenn beim Anlegen eines Relationsmanagers ein Fehler auftritt.

- `TermException`

  Diese Ausnahme wird geworfen, wenn bei der Auswertung eines relationalen Terms ein Fehler auftritt. Dies kann der Fall sein, wenn zu mindestens einem im Term benutzten Relationsnamen keine Relation mit diesem Namen existiert oder der Term syntaktisch falsch ist.

In welchen Methoden der Klasse `KurejavaOO` die Ausnahmen geworfen werden können, wird bei der Beschreibung der Methoden genannt.

4.4 Die Klasse `KurejavaOO`

In der Klasse `KurejavaOO` sind die Adaptermethoden zu den in der Klasse `Kurejava` deklarierten nativen Methoden implementiert. Die Adaptermethoden haben die Aufgabe die C-Zeiger, welche in Java in Form von `Byte-Arrays` vorliegen, in Instanzen der Datenhalterklassen `RelManagerImpl` und `RelationImpl` zu kapseln. Daher haben diese Adaptermethoden als Parameter- und Rückgabetypen Instanzen, welche die Schnittstellen `RelManager` und `Relation` implementieren. Für eine Fehlerbehandlung in Form von Java-Ausnahmen werden die Rückgabewerte der nativen Methoden in den Adaptermethoden geprüft und im Fehlerfall wird eine entsprechende Ausnahme ausgelöst.

Die Klasse `KurejavaOO` ist in der Abbildung 4.4 auf Seite 42 dargestellt. Sie speichert in dem Klassenattribut `init` vom Typen `boolean`, ob die Bibliothek bereits initialisiert wurde und vermerkt die angelegten Relationsmanager mit deren eindeutig zugeordneten Namen in dem Klassenattribut `relManagers` vom Typen `Map`.

Der gemeinsame Ablauf aller Adaptermethoden ist wie folgt. Zuerst werden die Initialisierung und die Vorbedingungen geprüft, die entsprechende native Methode aufgerufen, die Rückgabewerte geprüft und bei aufgetretenen Fehlern entsprechende Ausnahmen ausgelöst. Da der Ablauf in den einzelnen Methodenimplementierungen gleich aufgebaut ist, wird dieser Ablauf im Folgenden exemplarisch am Beispiel der Methode `initRelManager` gezeigt, welche einen neuen Relationsmanager anlegt:

```
protected static RelManagerImpl initRelManager(String name) throws
NotInitializedException, RelManagerException {
        if (!init) throw
          new NotInitializedException("Kurejava not initialized");
        if (relManagers.containsKey(name))
          throw new RelManagerException
```

```
            ("relManager␣with␣this␣name␣already␣initialized");
        byte[] relManagerPointer
          = Kurejava.initRelManager();
        if (relManagerPointer == null)
          throw new RelManagerException("error␣initializing␣a␣relManager");
        RelManagerImpl relManager
          = new RelManagerImpl(name, relManagerPointer);
        relManagers.put(name, relManager);
        return relManager;
    }
```

Zuerst wird in dieser Adaptermethode über das statische Attribut `init` geprüft, ob die Bibliothek bereits initialisiert wurde, ansonsten wird die Ausnahme `NotInitializedException` geworfen. Ist die Bibliothek korrekt initialisiert wird in der `Map relManagers` geprüft, ob bereits ein Relationsmanager mit dem übergebenen Namen existiert. Ist dies der Fall, wird die Java-Ausnahme `RelManagerException` geworfen. Ansonsten wird über die Klasse `Kurejava` über die native Methode `initRelManager` ein neuer Relationsmanager angelegt. Der Rückgabewert der nativen Methode wird geprüft. Wenn kein Relationsmanager angelegt wurde, also `null` zurückgeliefert wurde, wird die Ausnahme `RelManagerException` aufgerufen. Ansonsten wird eine neue Instanz der Klasse `RelManagerImpl` als Datenhalter für den C-Zeiger, der den neuen Relationsmanager repräsentiert, angelegt, und dieser der `Map relManagers` hinzugefügt, bevor die Adaptermethode die Instanz zurückliefert.

Die Erläuterung der einzelnen Adaptermethoden befindet sich im Anhang A, ebenso die der drei weiteren Methoden, die in der Klasse implementiert sind, welche lediglich ermöglichen, alle angemeldeten Relationsmanager abzufragen oder die vorher angelegte Instanz eines Relationsmanagers unter Angabe des Namens zu ermitteln sowie zu prüfen, ob die Bibliothek bereits initialisiert wurde.

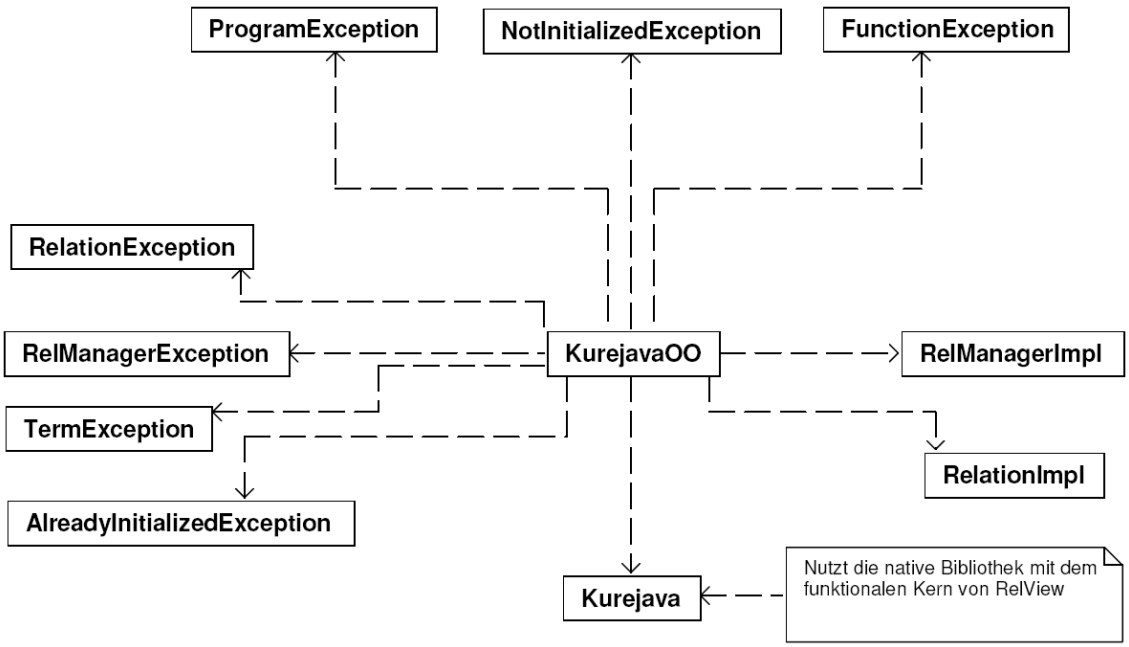

Abbildung 4.2: Die Klassen `Kurejava`, `KurejavaOO`, `RelManagerImpl`, `RelationImpl`, sowie die Ausnahmeklassen

Kapitel 5

Schnittstelle zu Java-Anwendungen

Dieses Kapitel beschreibt die Klassen und Schnittstellen, die innerhalb von Client-Anwendungen benötigt werden. Dies sind die beiden Klassen `RelationImpl` und `RelManagerImpl` mit den zugehörigen Java-Schnittstellen `Relation` und `RelManager`, die in den UML-Klassendiagrammen in Abschnitt 5.2 auf Seite 45 und Abschnitt 5.3 auf Seite 46 abgebildet sind.

5.1 Die Schicht für Client-Anwendungen

Die in den vorherigen Kapiteln beschriebenen Schichten ermöglichen bereits Relationen und Relationsmanager als Objekte anzusehen und Fehlerbehandlung durch Java-Ausnahmenbehandlung vorzunehmen. Allerdings ist es immer noch notwendig, für jede zu nutzende Funktion die statischen Methoden der Klasse `KurejavaOO` aufzurufen. Dies widerspricht dem Grundgedanken der Objektorientierung. Damit die Bibliothek vollständig objektorientiert genutzt werden kann, wurden zu den statischen Methoden entsprechende Methoden in den Schnittstellen `Relation` und `RelManager` deklariert und in den Klassen `RelationImpl` und `RelManagerImpl` implementiert, welche die Methoden der Klasse `KurejavaOO` aufrufen. Somit müssen die statischen Methoden der Klasse `KurejavaOO` nicht mehr von einer Client-Anwendung aufgerufen werden (siehe Abbildung 5.1 auf Seite 44).

Die Klassen `RelationImpl` und `RelManagerImpl` besitzen lediglich Paketsichtbarkeit und sind somit nicht direkt von der Client-Anwendung ansprechbar. Wie man Instanzen der Klassen erhält, wird im Kapitel 6 näher erläutert. Dieses Kapitel ist auf die Beschreibung der Klassen und Schnittstellen beschränkt. Die Client-Anwendung kann auf die Instanzen der Klassen über die als `public` deklarierten Schnittstellen `Relation` und `RelManager` zugreifen. Das Schlüsselwort `public` bedeutet, dass die Schnittstellen für Pakete in anderen Klassen verfügbar sind.

Innerhalb eines Java-Programmes, hier in dem Eclipse-Plugin, kann auf diese Anwendungsschicht zugegriffen werden. Auf diese Weise muss sich der Entwickler des Eclipse-Plugins nicht darum kümmern, wie der funktionale Kern intern angesprochen wird, sondern kann wie gewohnt Java-Methoden von entsprechenden Objekte aufrufen und Fehlerbehandlung über Ausnahmen implementieren. Im Folgenden sind die beiden Schnittstellen und ihre implementierenden Klassen erläutert, die Auflistung der Methoden ist im Anhang A ersichtlich.

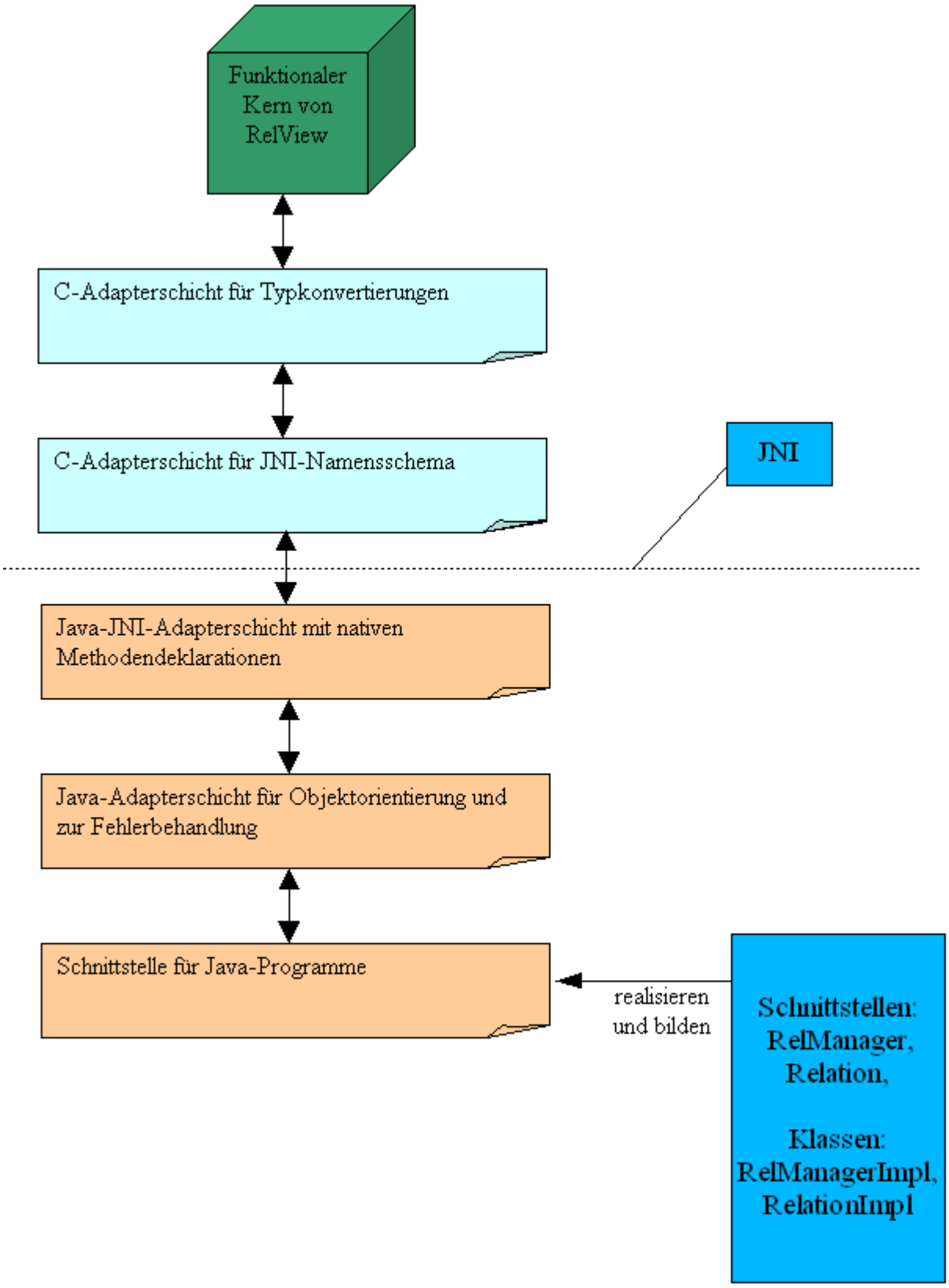

Abbildung 5.1: Die Schnittstellenschicht zu Client-Anwendungen (unterste Schicht).

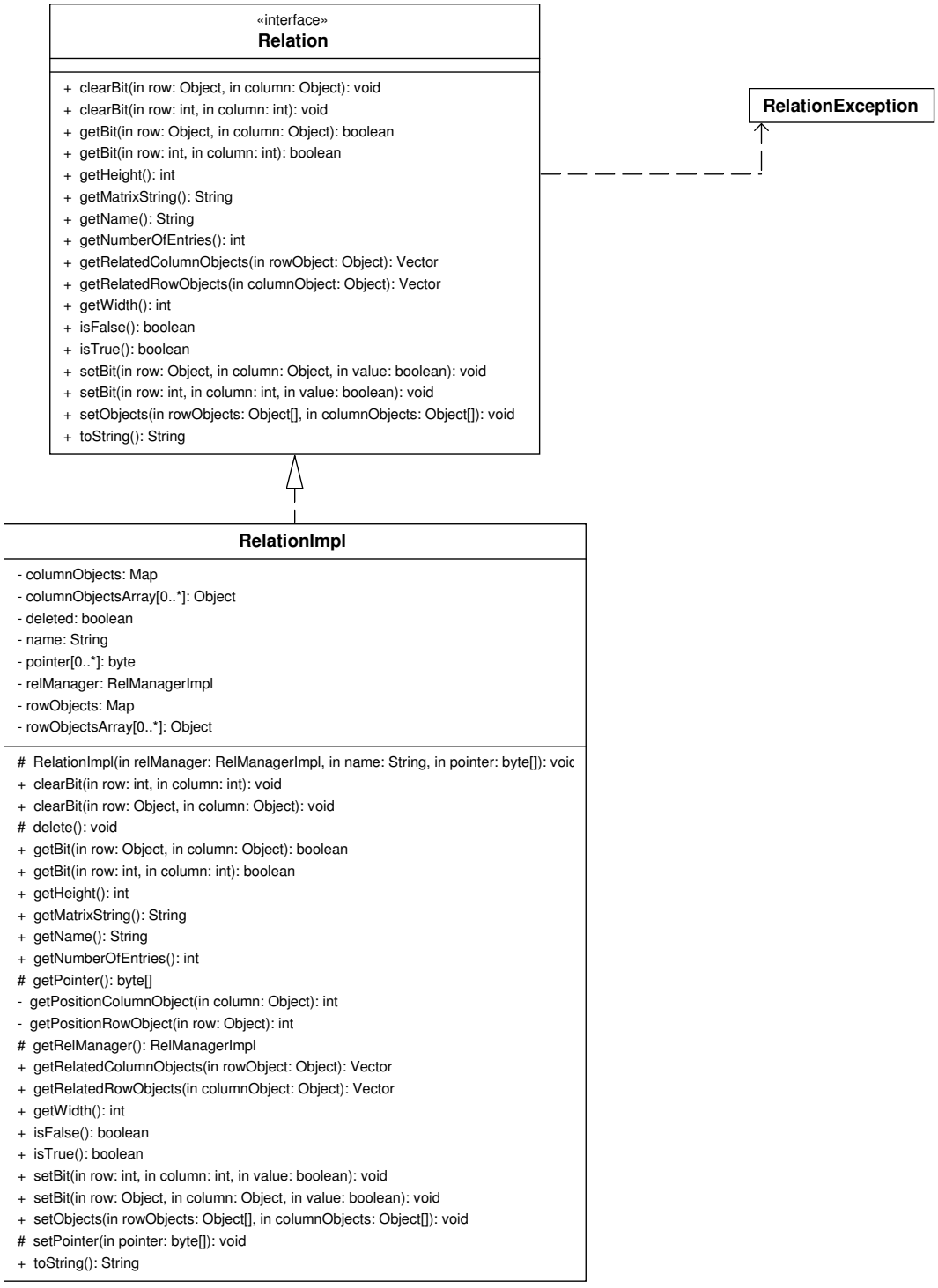

Abbildung 5.2: Die Schnittstelle `Relation` und die dazugehörige Implementierungsklasse `RelationImpl`

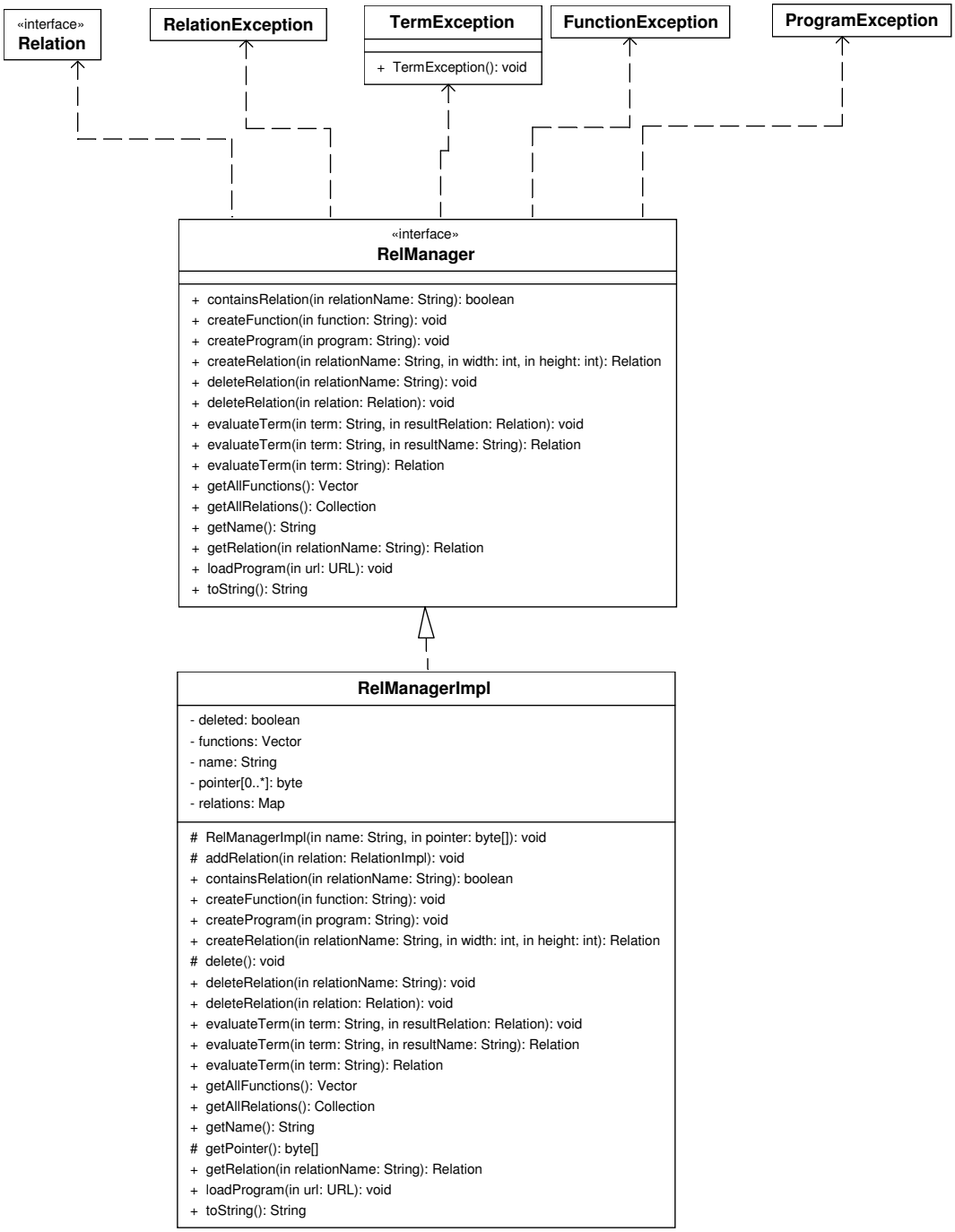

Abbildung 5.3: Die Schnittstelle `RelManager` und die dazugehörige Implementierungsklasse `RelManagerImpl`

5.1.1 Die Schnittstelle `Relation`

Die Schnittstelle `Relation`, die in dem UML-Klassendiagramm in Abbildung 5.2 auf Seite 45 dargestellt ist, deklariert die Methoden, welche die Bibliothek für den Umgang mit einer Relation zur Verfügung stellt. Diese beinhaltet alle Methoden, um Informationen über eine Relation abzufragen, Einträge in einer Relation zu setzen und zu löschen. Eine Instanz dieser Schnittstelle, also eine Instanz der implementierenden Klasse `RelationImpl`, stellt dabei eine Relation als Java-Objekt dar. Die Methoden werden in der Klasse `RelationImpl` implementiert. Zusätzlich zu der bekannten Art, einen Eintrag in einer Relation über Spalten- und Reihennummer anzugeben, wurde eine Methode vorgesehen, um Java-Objekte den Reihen- und Spaltennummern zuzuordnen. Dabei handelt es sich um die Methode `setObjects`, deren Implementierung im Anhang A erläutert ist. Diese Methode vermerkt zu den Reihen- und Spaltennummern die ihr übergebenen Objekte. Durch Überladung der anderen Methoden, die sonst Reihen- und Spaltennummern als Parameter bekommen, ist es damit möglich, Einträge zu setzen und abzufragen, in dem man statt der Reihen- und Spaltennummer die korrespondierenden Objekte angibt. Da die Bibliothek dazu genutzt werden soll, Beziehungen zwischen Objekten in Relationen festzuhalten und diese auszuwerten, ist dies ein wichtiger Schritt, um die Bibliothek komfortabel nutzen zu können. Statt in der Client-Anwendung zu vermerken, welche Nummern zu den Objekten gehören, übernimmt dies die Bibliothek.

5.1.2 Die Klasse `RelationImpl`

Die Klasse `RelationImpl`, die in dem UML-Klassendiagramm in Abbildung 5.2 auf Seite 45 ersichtlich ist, implementiert die Schnittstelle `Relation` und definiert zusätzliche Methoden, welche nur innerhalb der Bibliothek genutzt werden. Die zusätzlichen Methoden dienen zum Setzen und Abfragen des `Byte-Arrays`, welche den C-Zeiger beinhalten und zum Ermitteln des Relationsmanagers, dem die Relation untergeordnet ist sowie zum Löschen des Objektes. Diese zusätzlichen Funktionen werden in der Klasse `KurejavaOO` benötigt. Allen diesen Methoden ist gemeinsam, dass zu Beginn überprüft wird, ob die Relation bereits als gelöscht markiert wurde. Dies ist wichtig, da eine Relation von der Client-Anwendung gelöscht werden kann, sich das entsprechende Relationsobjekt aber noch im Speicher befindet, da in Java keine Objekte direkt gelöscht werden können. Falls die Client-Anwendung eine Methode des Relationsobjektes aufruft, nachdem die Relation bereits gelöscht wurde, wird nach der Prüfung eine Java-Laufzeitausnahme ausgelöst. Würde man auf diesen Mechanismus verzichten, könnte es zum Systemabsturz führen, da so eventuell in den nativen C-Funktion auf bereits freigegebenen Speicherbereich zugegriffen wird.

5.1.3 Die Schnittstelle `RelManager`

Die Schnittstelle `RelManager`, die in dem UML-Klassendiagramm in Abbildung 5.3 auf Seite 46 ersichtlich ist, deklariert die Methoden eines Relationsmanager-Objektes, die zur Verwaltung von Relationen, relationalen Funktionen und Programmen und zur Termauswertung benötigt werden. Diese können von einer Client-Anwendung aufgerufen werden. Eine Instanz dieser Klasse stellt einen Relationsmanager als Java-Objekt dar. Die Methoden werden in der Klasse `RelManagerImpl` implementiert. Der Relationsmanager delegiert Aufrufe an die Klasse `KurejavaOO` und dient lediglich dazu, einer Client-Anwendung ein Objekt anzubieten, so dass diese nicht auf statische Methode zugreifen muss.

5.1.4 Die Klasse `RelManagerImpl`

Die Klasse `RelManagerImpl`, die in dem UML-Klassendiagramm in Abbildung 5.3 auf Seite 46 ersichtlich ist, implementiert die Schnittstelle `RelManager`, welche in Abschnitt 5.1.3 auf Seite 47 beschrieben wurde.

5.2 Zusammenfassung

Mit Hilfe der Schnittstellen `RelManager` und `Relation` ist es Client-Anwendungen möglich, Relationsmanager und Relationen objektorientiert zu nutzen, Informationen über sie abzufragen und die Relationen zu manipulieren. Die Implementierung dieser Schnittstellen befindet sich in den Klassen `RelManagerIml` und `RelationImpl`. Diese Implementierungsklassen sind vor den Client-Anwendungen verborgen. Client-Anwendungen bekommen Instanzen dieser Implementierungsklassen von einer im nächsten Kapitel 6 erläuterten Fabrikklasse. Sie sprechen diese Instanzen ausschließlich über die Schnittstellen an.

Kapitel 6

Einstiegspunkt für Client-Anwendungen

Im vorherigen Kapitel 5 wurde erläutert wie eine Client-Anwendung mit Hilfe von Objekten, die die Schnittstelle `RelManager` und `Relation` implementieren, auf die Funktionalität der Bibliothek zugreifen kann. Es bleibt noch aus, wie eine Client-Anwendung Zugriff auf Objekte dieses Typen bekommt, um mit diesen dann wie im vorherigen Kapitel beschrieben auf die Funktionalität der Bibliothek zuzugreifen. Dieses Kapitel klärt, wie man Instanzen dieser Objekte erzeugt, Zugriff auf diese für den Umgang mit der Bibliothek notwendigen Objekte bekommt und die Bibliothek initialisiert.

6.1 Die Klasse `Context`

In den vorherigen Kapiteln wurde beschrieben, wie eine Client-Anwendung alle Funktionen des funktionalen Kern über Instanzen, welche die Schnittstellen `RelManager` und `Relation` implementiert, nutzen kann. Um die Bibliothek vollständig nutzen zu können, muss noch Funktionalität zur Verfügung gestellt werden, diese Instanzen zu erhalten und vorher die native Bibliothek zu laden, damit *JNI* zu den als native deklarierten Methoden Implementierungen finden kann. Zusätzlich muss die Bibliothek initialisiert werden können. Darüberhinaus benötigt eine Client-Anwendung eine Methode zur Speicherbereinigung, wenn die Nutzung der Bibliothek abgeschlossen ist. Die Funktionalität der Initialisierung, des Beendens und des Ladens der nativen Bibliothek wurde in der Klasse `Context` implementiert. Über diese Klasse kann man ein Fabrikobjekt erhalten, welches Relationsmanagerobjekte nach dem Fabrikentwurfsmuster verwaltet. Die Klasse `Context` stellt den Einstiegspunkt für jede Client-Anwendung dar. Möchte man in einer beliebigen Client-Anwendung die Bibliothek nutzen, so muss die Client-Anwendung zuerst über die Klasse `Context` aus dem Paket `de.osz.kurejava` die Bibliothek initialisieren, die native Bibliothek laden und das Fabrikobjekt für die Relationsmanagerobjekte, welche die Schnittstelle `Home` implementiert, ermitteln. Der Grund für die Verwendung eines Fabrikobjekt ist in Abschnitt 6.1.1 auf Seite 51 beschrieben, in dem die Schnittstelle `Home` erläutert wird. Die Klasse `Context` ist in dem UML-Klassendiagramm in Abbildung 6.1 auf Seite 50 dargestellt. Die Methoden der Klasse `Context` werden im Anhang A beschrieben.

Eine beliebige Client-Anwendung ruft zu Beginn der Bibliotheksnutzung eine der statischen `init`-Methoden auf, um die Bibliothek zu initialisieren und die native Bibliothek zu laden. Danach kann sie über die statische Methode `getHome` ein im nachfolgenden Abschnitt erläutertes Fabrikobjekt für

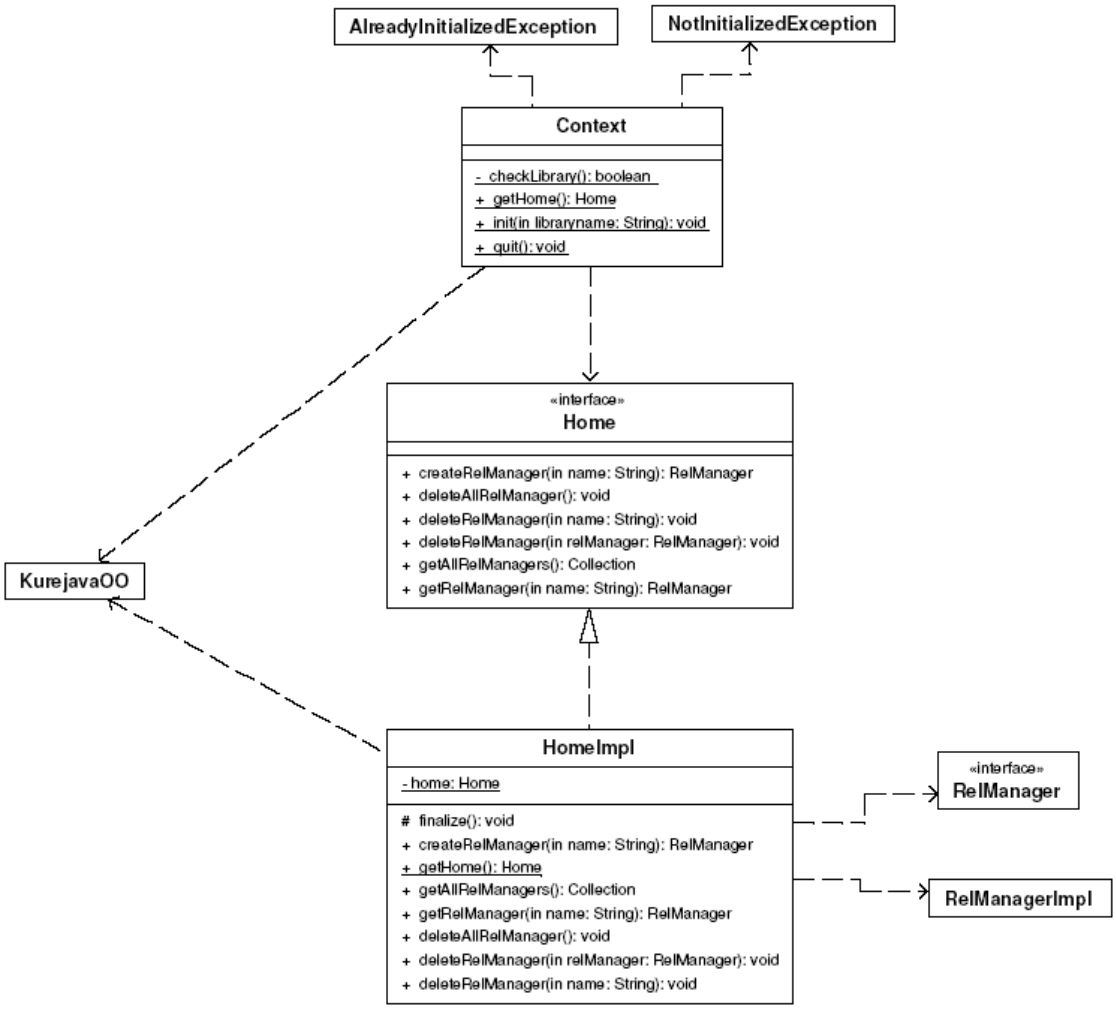

Abbildung 6.1: Die Klassen `Context`, `HomeImpl` und die Schnittstelle `Home`

die Relationsmanager-Objekte erhalten.

6.1.1 Die Schnittstelle `Home`

Die Schnittstelle `Home` ist in dem UML-Klassendiagramm in Abbildung 6.1 auf Seite 50 abgebildet und deklariert Methoden, welche von einer Client-Anwendung benutzt werden, um Relationsmanager zu verwalten. Dazu gehört das Erzeugen von Relationsmanagern, das Ermitteln von existierenden Relationsmanagern und das Löschen von Relationsmanagern. Java-Objekte, die diese Schnittstelle implementieren, sind also Fabrikobjekte für Relationsmanager. Bei Relationsmanagern handelt es sich dabei, wie in Kapitel 5 beschrieben, um Java-Objekte, die über die Schnittstelle `RelManager` angesprochen werden können. Die in Abschnitt 6.1.2 auf Seite 51 beschriebene Klasse `HomeImpl` implementiert die Schnittstelle `Home`, so dass Instanzen dieser Klasse die Fabrikobjekte darstellen. Hierbei handelt es sich um eine Umsetzung des Fabrikentwurfsmusters. Dabei entspricht die Schnittstelle `Home` der abstrakten Fabrik, während ihre Implementierung `HomeImpl` einer konkreten Fabrik entspricht (siehe Abbildung 6.2 auf Seite 52). Das abstrakte Produkt ist durch die Schnittstelle `RelManager` und die konkrete Implementierung durch die Klasse `RelManagerImpl` gegeben. Bei dem üblichen Fabrikentwurfsmuster gilt, dass die abstrakte Fabrik eine Methode enthält, um ein Produkt zu erzeugen. Bei der Umsetzung ist dies ein Relationsmanagerobjekt. Erst eine konkrete Fabrik gibt an, welches konkrete Produkt erstellt wird, also in welchen Ausprägungen es vorliegt. Die konkrete Entscheidung was erzeugt wird, wird somit in den Spezialisierungen der abstrakten Fabrik getroffen. Damit ist es möglich, andere Spezialisierungen der Schnittstelle zu entwickeln, die dann spezielle Relationsmanagerobjekte liefern. Auf diese Weise bleibt die Java-Bibliothek offen für zukünftige Erweiterungen. Im vorliegenden Fall wurde das Muster aber nicht ausschließlich dafür verwendet, offen zu lassen, welche Spezialisierungen erzeugt werden können, sondern es wird hier ebenfalls dazu genutzt, in der Fabrikmethode, die das Relationsmanagerobjekt erzeugt, noch Verwaltungsaufgaben für den Relationsmanager vorzunehmen. Der Relationsmanager muss vermerkt werden, damit er später, wenn er nicht mehr benötigt wird, gelöscht werden kann. Da beim Anlegen eines Relationsmanagers durch den funktionalen Kern von RELVIEW in C Speicher allokiert wird, muss dieser auch wieder freigegeben werden. Somit wird innerhalb der Fabrikmethode das Relationsmanagerobjekt erzeugt und zurückgegeben, aber auch vermerkt, dass es erzeugt wurde, so dass es verwaltet, bei Bedarf gelöscht und der in C allokierte Speicher freigegeben werden kann. Um also eine erweiterungsfähige Bibliothek zu schreiben und die Verwaltung und Speicherbereinigung der Objekte effizient implementieren zu können, wurde an dieser Stelle das Fabrikentwurfsmuster gewählt. Alternativ hätten die Objekte über den Java-Operator `new` angelegt werden können, dann wäre die Bibliothek aber nicht so einfach erweiterbar, und die Verwaltung der Objekte hätte umständlich in Konstruktoren implementiert werden müssen, auch in den Konstruktoren jeder weiteren zusätzlichen Spezialisierung.

6.1.2 Die Klasse `HomeImpl`

Die Klasse `HomeImpl` ist in dem UML-Klassendiagramm in Abbildung 6.1 auf Seite 50 abgebildet und implementiert die Schnittstelle `Home`. Diese Klasse ist nach dem Singleton-Entwurfsmuster implementiert. Es kann also nur eine Instanz dieser Klasse gleichzeitig existieren. Bei Instanzen dieser Klasse handelt es sich um Fabrikobjekte für Relationsmanager-Objekte, welche über die Schnittstelle `RelManager` ansprechbar sind. Zusätzlich zu den in der Schnittstelle `Home` deklarierten Metho-

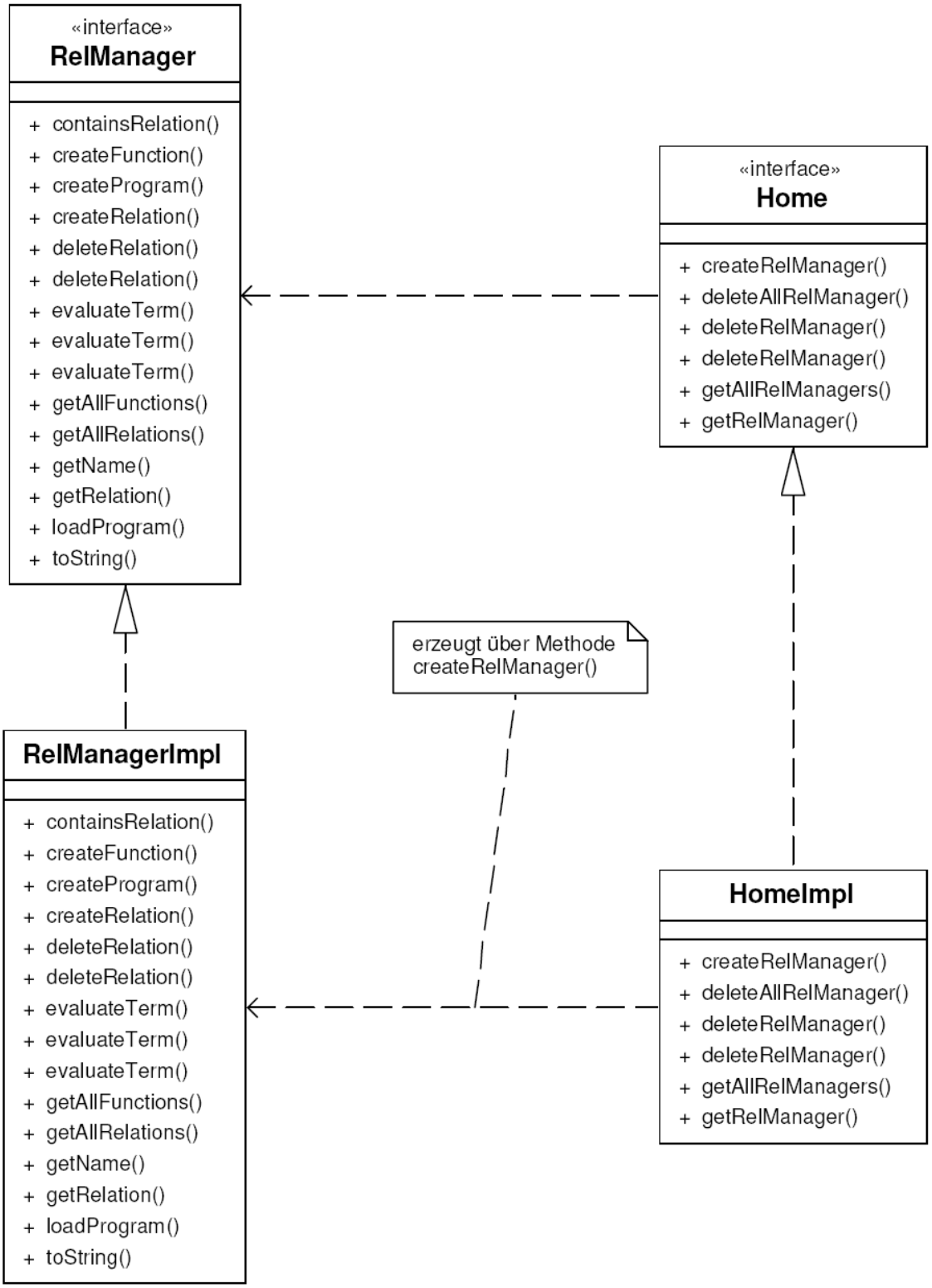

Abbildung 6.2: Das Diagramm zeigt die Umsetzung des Fabrikentwurfsmuster für das Erzeugen von Relationsmanagern. Die Schnittstelle `Home` entspricht einer abstrakten Fabrik, die Klasse `HomeImpl` entspricht einer konkreten Fabrik, die Schnittstelle `RelManager` entspricht einem abstrakten Produkt und die Klasse `RelManagerImpl` entspricht einem konkreten Produkt aus dem Fabrikentwurfsmuster.

den, wird in der Klasse `HomeImpl` die statische Methode `getHome` implementiert, welche von der Klasse `Context`, die in Abschnitt 6.1 auf Seite 49 erläutert ist, aufgerufen wird. Diese Methode liefert stets dieselbe, beim ersten Aufruf der Methode erzeugte Instanz der Klasse `HomeImpl`. Das Singleton-Entwurfsmuster wurde an dieser Stelle verwendet, da man das Fabrikobjekt nicht in mehreren Instanzen benötigt und auf diese Weise, da nur ein Objekt existiert, Speicherbereich einspart.

6.2 Zusammenfassung

Eine Client-Anwendung nutzt als ersten Einstiegspunkt in die Bibliothek die Klasse `Context`, um über die statische Methode `getHome` eine Instanz zu bekommen, die die Schnittstelle `Home` implementiert. Dabei wird genau eine solche Instanz erzeugt und bei jedem Aufruf der Methode `getHome` diese Instanz zurückgeliefert. Mit dieser Instanz kann eine Client-Anwendung Instanzen von Relationsmanagern bekommen und diese nutzen. Mit Hilfe der Relationsmanager können Relationen angelegt, abgefragt, manipuliert und Terme ausgewertet werden. Die Relationsmanager-Objekte wurden in dem Kapitel `Schnittstelle zu Java-Anwendungen` beschrieben.

6.2.1 Beispielabläufe

Im diesem Abschnitt werden fünf Beispielabläufe beschrieben, welche das Benutzen der Bibliothek verdeutlichen. Jeder der Beispielabläufe wird durch ein UML-Sequenzdiagramm dargestellt und beschrieben.

- Initialisierung der Bibliothek

  Um die Bibliothek zu initialisieren muss innerhalb einer Client-Anwendung, wie in dem UML-Diagramm in Abbildung 6.3 auf Seite 54 ersichtlich, die statische Methode `init` der Klasse `Context` aufgerufen werden. Dieser Methode wird die zu benutzende native Bibliothek mit dem gekapselten funktionalen Kern von `RelView` als Java-Dateiobjekt oder als Pfadangabe übergeben. Innerhalb dieser Methode wird die Methode `loadLibrary` der Klasse Kurejava aufgerufen, um die native dynamische Bibliothek zu laden. Danach wird die geladene Bibliothek mit dem Aufruf der Methode checkLibrary geprüft. Die Prüfung besteht aus Aufrufen der Funktionen, welche die native Bibliothek implementieren muss. Auf diese Weise wird sichergestellt, dass die richtige native Bibliothek angegeben und geladen wurde. Danach wird die Bibliothek durch den Aufruf der Methode `init` der Klasse `KurejavaOO` initialisiert. In der Klasse `KurejavaOO` wird daraufhin die Methode `init` der Klasse `Kurejava` aufgerufen. Nachdem die Bibliothek korrekt initialisiert wurde, kann man innerhalb der Client-Anwendung mit einem Aufruf der Methode `getHome` der Klasse `Context`, welche eine Fabrik für Relationsmanager darstellt, erhalten. Die Methode ermittelt die einzige Instanz dieser Fabrik über die gleichnamige statische Methode der Klasse `HomeImpl`.

- Umgang mit Relationsmanagern

  Um einen Relationsmanager zu erzeugen, muss innerhalb einer Client-Anwendung die Methode `createRelManager` des `Home`-Objektes aufgerufen werden. Dies ist in dem UML-Diagramm in Abbildung 6.4 auf Seite 55 ersichtlich. Innerhalb dieser Methode wird die Methode `initRelManager` der Klasse `KurejavaOO` aufgerufen. Die Methode ruft zuerst die

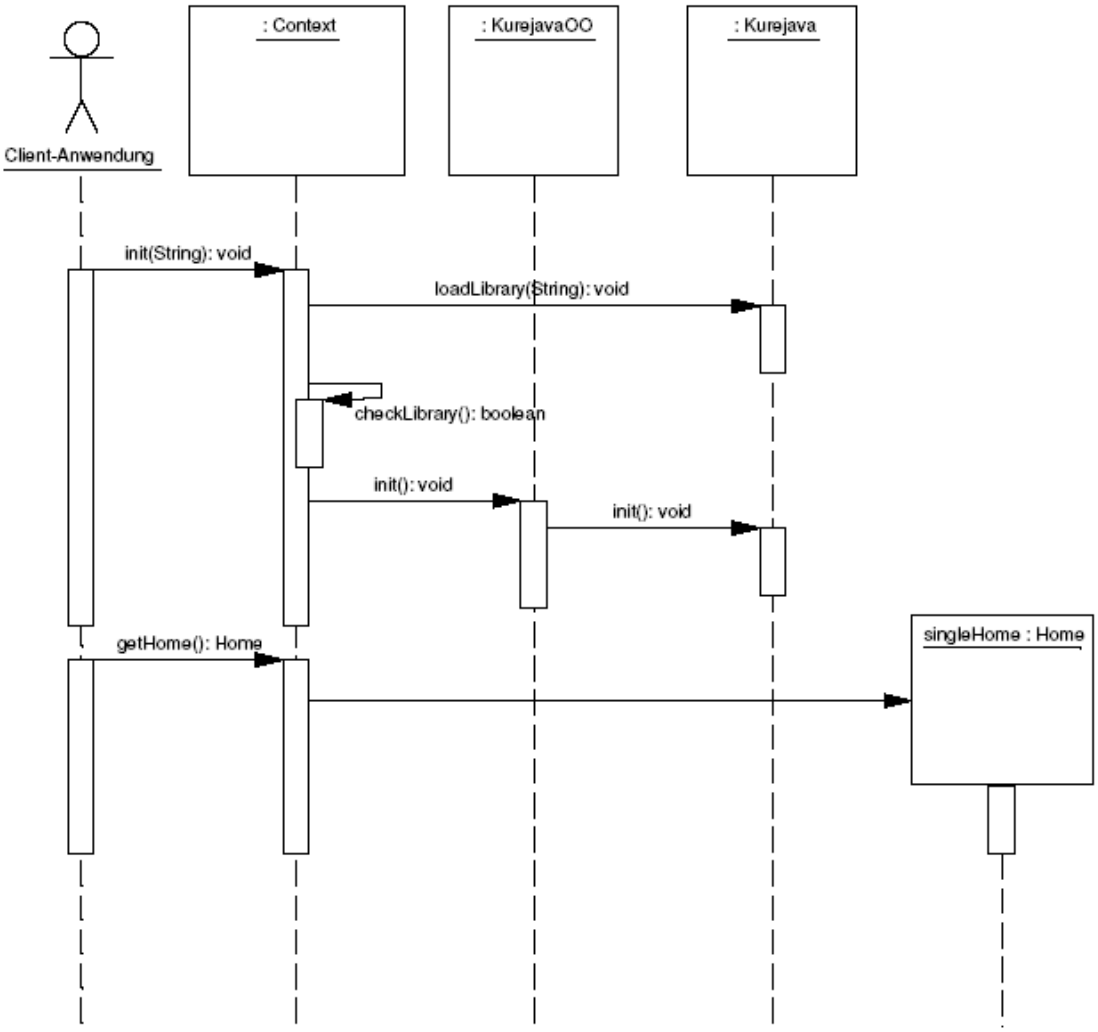

Abbildung 6.3: Initialisierung der Bibliothek

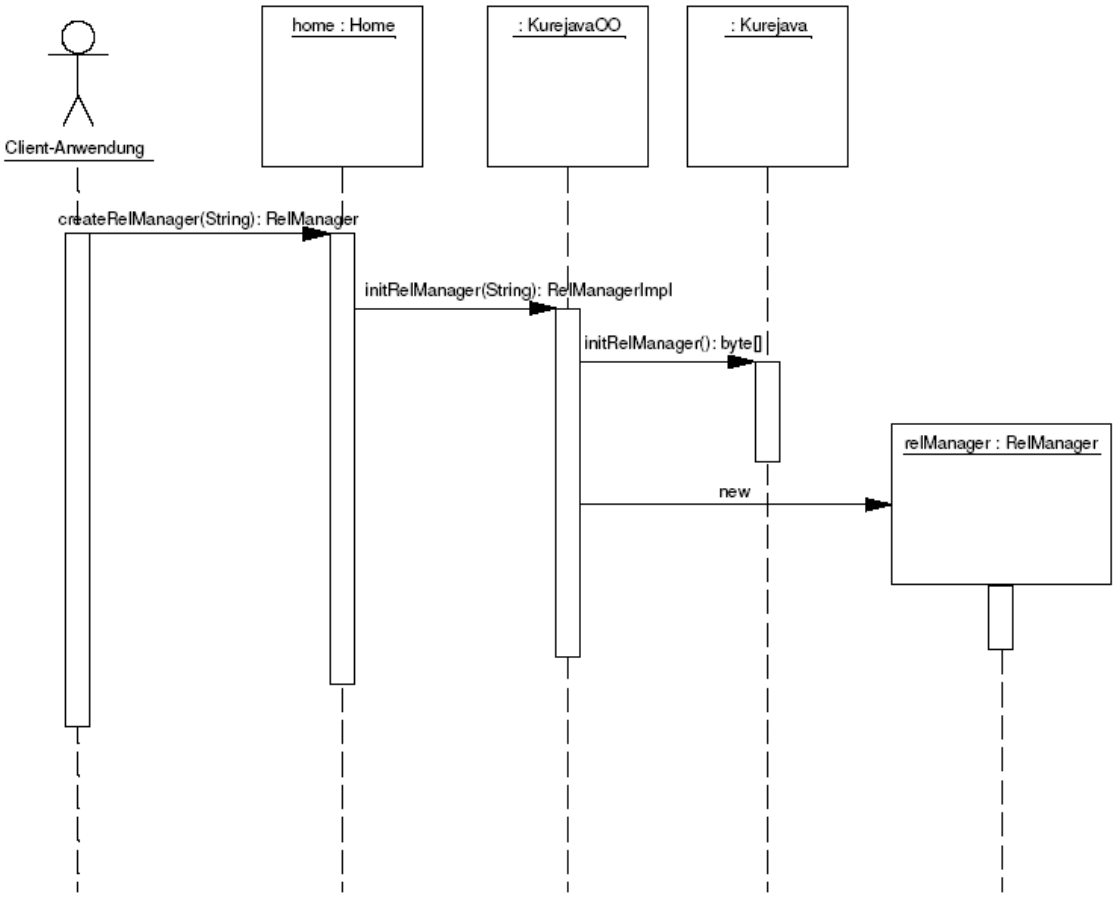

Abbildung 6.4: Umgang mit Relationsmanagern

gleichnamige Methode der Klasse `Kurejava` auf, und kapselt den durch die Methode zurückgebenen Zeiger in Form eines Java-`byte`-Arrays in einem neuen Relationsmanagerobjekt. Dieser neu erzeugte Relationsmanager wird der Client-Anwendung zurückgegeben.

- Umgang mit Relationen

  Um eine Relation zu erzeugen, muss in Client-Anwendungen die Methode `createRelation` eines Relationsmanagerobjektes aufgerufen werden. Dies ist in dem UML-Diagramm in Abbildung 6.5 auf Seite 56 dargestellt. Innerhalb dieser Methode wird die Methode `relatioNew` der Klasse `KurejavaOO` aufgerufen. In dieser Methode wird die gleichnamige Methode der Klasse `Kurejava` aufgerufen und damit die Relation über den funktionalen Kern von REL-VIEW erzeugt. Danach wird eine neues Relationsobjekt erzeugt, und der durch die native Funktion zurückgegebene Zeiger in Form eines Java-`byte`-Arrays darin gekapselt. Dieses neu erzeugte Relationsobjekt wird der Client-Anwendung zurückgegeben.

- Termauswertung

  Um einen Term der relationalen Algebra auszuwerten, muss man in einer Client-Anwendung die Methode `evaluateTerm` eines Relationsmanagerobjektes aufrufen. Dies ist in dem UML-Diagramm in Abbildung 6.6 auf Seite 58 ersichtlich. Von dieser Methode gibt es unterschied-

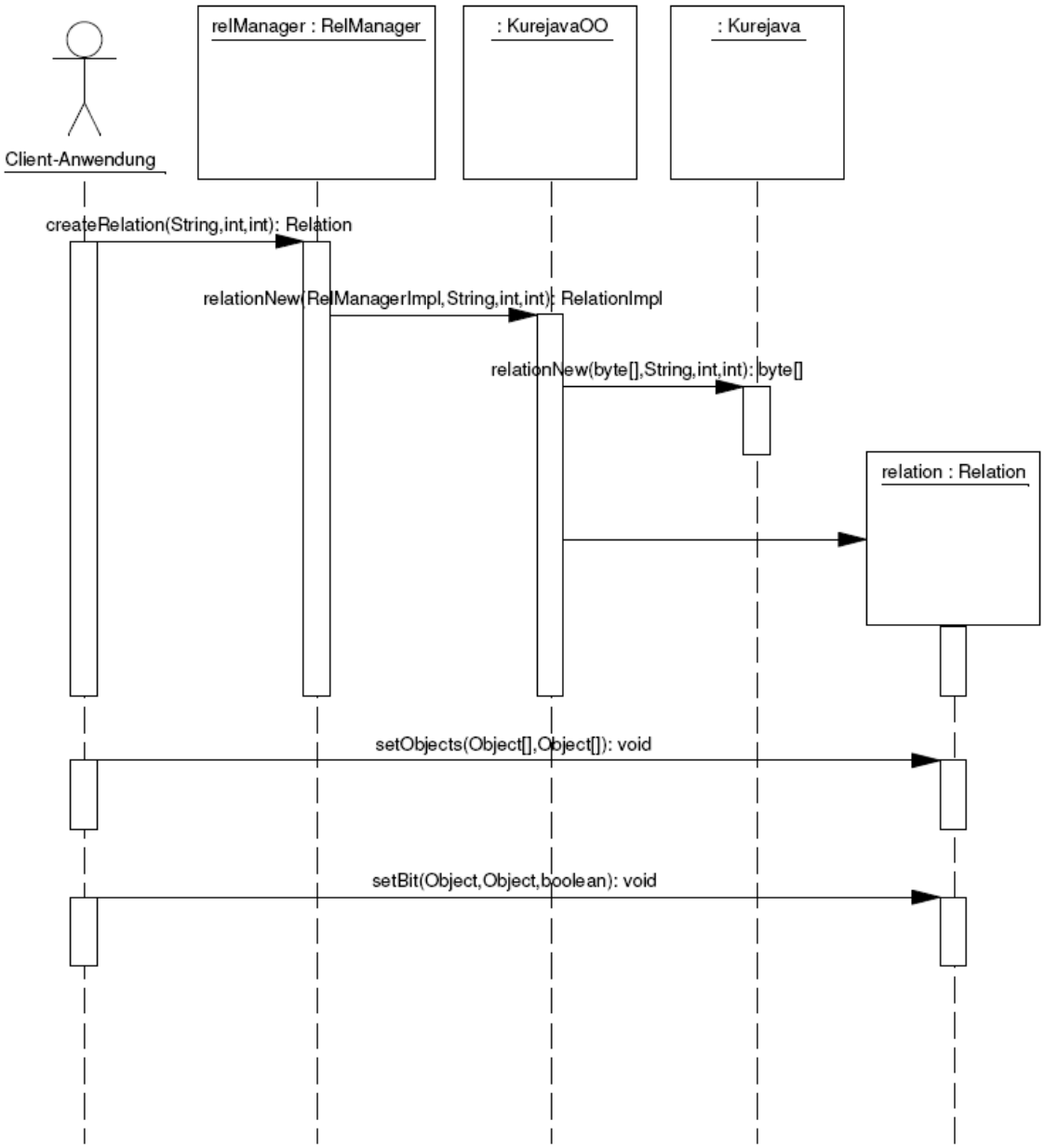

Abbildung 6.5: Umgang mit Relationen

liche Ausprägungen, in diesem Beispiel wird die Methode, welche nur den Term als Parameter in Form einer Zeichenkette erhält, und die das Ergebnis der Auswertung als neue Relation zurückgibt, genutzt. Dazu wird zuerst in der Methode ein neues Relationsobjekt über die Methode `createRelation` des Relationsmanagerobjektes erstellt. In dieser Methode wird die Methode `relationNew` der Klasse `Kurejava` aufgerufen. Diese Methode ruft die gleichnamige Methode der Klasse `Kurejava` auf, um eine neue Relation mit dem Namen `result` zu erzeugen und kapselt den durch die Methode zurückgebenen Zeiger in Form eines Java-`byte`-Arrays in einem neuen Relationsmanagerobjekt.

Danach wird in der Methode `evaluateTerm` des Relationsmanagerobjektes die gleichnamige Methode der Klasse `KurejavaOO` aufgerufen. In dieser Methode wird der Term durch einen Aufruf der Methode `evaluateTerm` der Klasse `Kurejava` ausgewertet und in der Ergebnisrelation `result` gespeichert. Der Zeiger auf diese Ergebnisrelation wird durch einen Aufruf der Methode `relationGet` ermittelt und in dem vorher erzeugten Relationsobjekt gesichert. Dieses Relationsobjekt wird an die Client-Anwendung zurückgegeben.

- Beenden der Bibliotheksbenutzung

  Wenn die Bibliothek nicht mehr benötigt wird, kann man den Speicherbereich, der durch die nativen C-Bibliothek allokiert wurde, freigeben, in dem eine Client-Anwendung die statische Methode `quit` der Klasse `Context` aufruft (vgl. Abbildung 6.7 auf Seite 59). In dieser Methode wird die Methode `deleteAllRelManager` des Fabrikobjektes für Relationsmanager aufgerufen. In dieser Methode werden alle vorhandenen Relationsmanager durchgegangen und von jedem die Methode `delete` aufgerufen. In der `delete`-Methode wird für alle von dem jeweiligen Relationsmanager verwalteten Relationsobjekte die Methode `delete` aufgerufen. Diese Methode vermerkt in den Relationsobjekten, dass sie nicht mehr benutzt werden dürfen. Da man bei Java-Objekten das endgültige Löschen des Objektes aus dem Speicher nicht selbst veranlassen kann, sondern die *Java Virtual Machine* dies selbst verwaltet, sind die Objekte nicht real aus dem Speicher gelöscht. Daher existiert die Lebenslinie in dem Diagramm weiterhin und wird nach dem Aufruf der `delete`-Methode nicht beendet. Nachdem in der `delete`-Methode des Relationsmanagerobjektes alle Relationsobjekte durchgegangen sind, wird die Methode `quitRelManager` der Klasse `KurejavaOO` und von dieser die gleichnamige Methode der Klasse `Kurejava` aufgerufen um den von dem Relationsmanager und seinen verwalteten Relationsobjekte allokierten Speicherbereich in der nativen Bibliothek freizugeben. Nachdem alle Relationsmanagerobjekte auf diese Weise bereinigt wurden, wird in der Methode `quit` der Klasse `Context` die gleichnamige Methode der Klasse `KurejavaOO` und von dieser die gleichnamige der Klasse `Kurejava` aufgerufen. Hierüber werden die restlichen von der nativen Bibliothek belegten Speicherbereiche freigegeben und der Zugriff auf die Bibliothek wurde korrekt beendet.

  Die `quit`-Methode der Klasse `Context` in einer Java-Client-Anwendung aufzurufen ermöglicht es, den Zugriff auf die Bibliothek zu beenden und alle belegten Speicherbereiche freizugeben. Vorher angelegte Relations- und Relationsmanagerobjekte werden dabei nicht gelöscht, da dies in Java nicht möglich ist. Aber sie können von da an nicht mehr benutzt werden. Evtl. später folgende Methodenaufrufe an diesen Objekte führen zu Java-Laufzeitausnahmen. Wird die `quit`-Methode nicht aufgerufen, beendet sich die Bibliothek von alleine, sobald ihre Objekte vom Java-Garbage-Collector gelöscht werden. Nach dem Beenden einer Java-Anwendung, welche die Bibliothek nutzt, ist also auf jeden Fall sichergestellt, dass der allokierte Speicherbereich wieder freigegeben wurde.

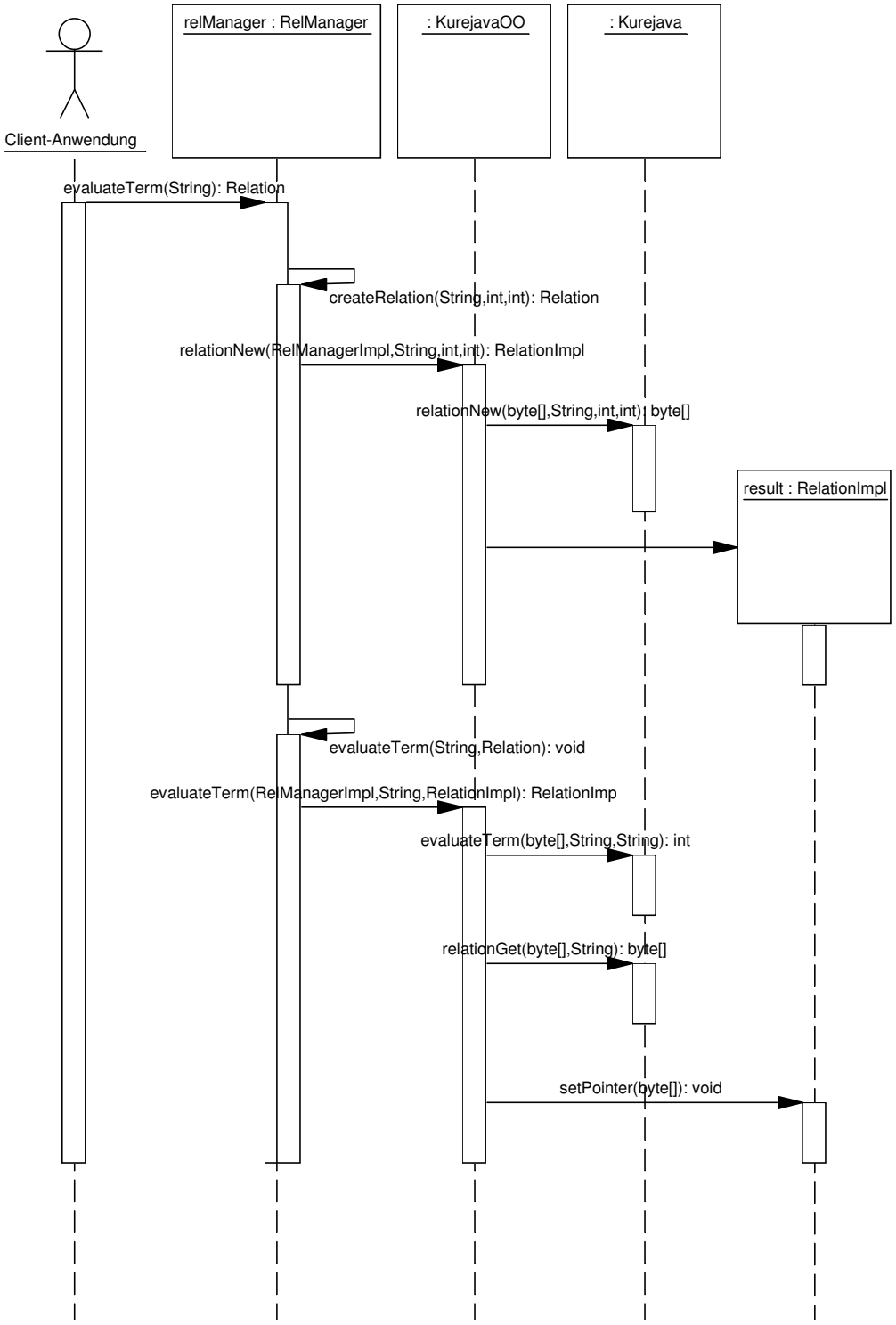

Abbildung 6.6: Termauswertung

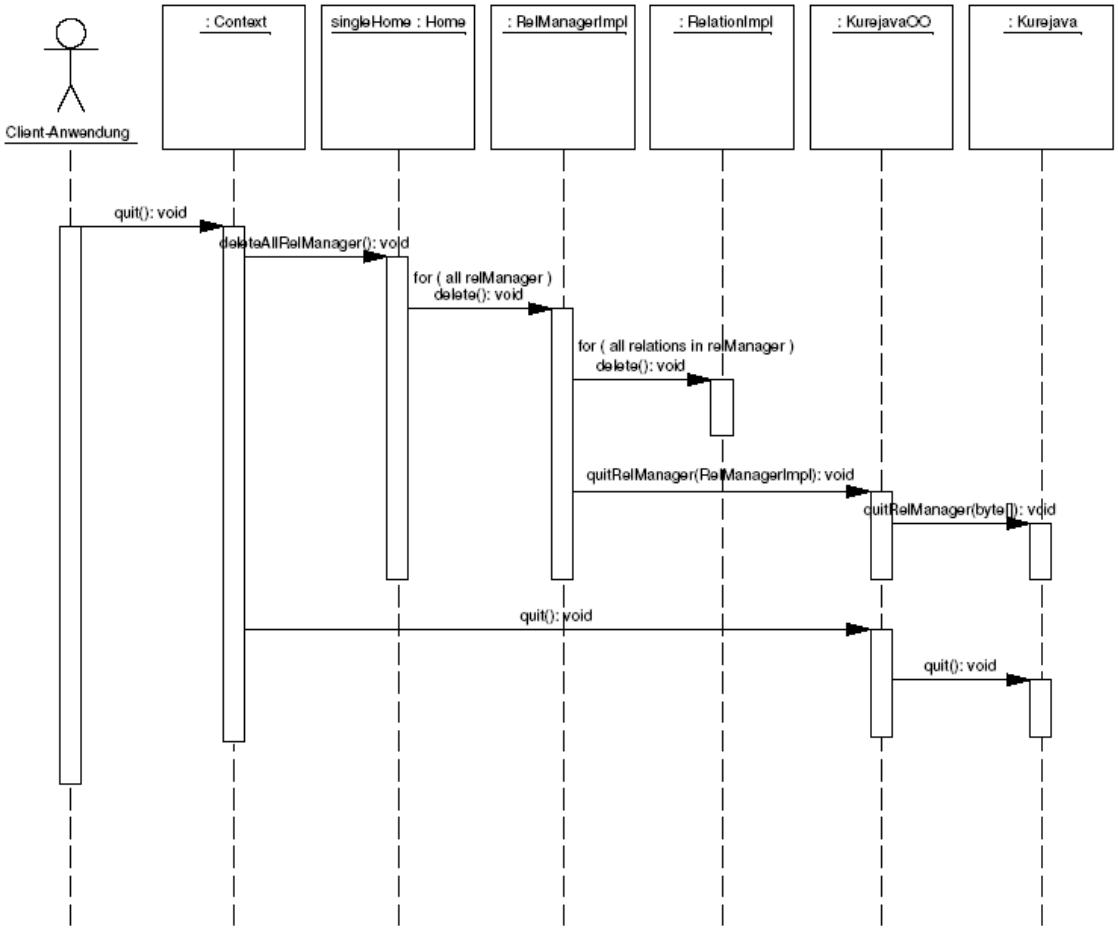

Abbildung 6.7: Beenden der Bibliotheksbenutzung

Kapitel 7

Der Eclipse-Plugin-Prototyp

In diesem Kapitel wird der Eclipse-Plugin-Prototyp beschrieben, welcher dazu benötigt wird, die in der vorliegenden Arbeit entwickelte Java-Bibliothek zu testen und für die weitere Arbeit am Lehrstuhl Software-Technologie als Basis dienen kann.

7.1 Die Arbeitsweise des Prototypen

Der Prototyp ist in die Oberfläche von Eclipse eingebunden. Er kann im Projektbaum von Eclipse über das Kontextmenü aufgerufen werden. Er bekommt als Eingabe vom Benutzer über ein Dateiauswahlmenü eine XML-Datei, welche eine dreidimensionale Szene beschreibt, die statische Beziehungen von Paketen, Schnittstellen und Klassen in Java visualisiert. Dabei werden die im Kapitel 1 beschriebenen Darstellungsformen genutzt. Der Prototyp visualisiert die Beziehungen nicht, er verarbeitet lediglich die XML-Datei, ermittelt die dadurch beschriebenen Beziehungen und generiert daraus Relation. Die Relationen werden auf später beschriebene Eigenschaften geprüft. Ergibt die Prüfung, dass die Visualisierung korrekten Java-Code darstellt, werden die darin beschriebenen Pakete, Schnittstellen und Klassen in Eclipse generiert. Während der gesamten Ablaufdauer protokolliert der Prototyp den Fortschritt und eventuelle Fehlermeldungen. Die Klassen des Prototypen liegen in dem Java-Paket `de.osz.eclipseprototype`.

7.2 Der Prototyp

Der Prototyp für das Eclipse-Plugin wurde so aufgebaut, dass er leicht anzupassen und leicht zu erweitern ist, damit der Prototyp auch für die weitere Entwicklung des Plugins genutzt werden kann. Die Architektur des Prototypen sieht vor, dass der Prototyp grob in mehrere Komponenten gegliedert ist, wie in Abbildung 7.1 auf Seite 61 ersichtlich. Der Begriff Komponente wird hier für thematisch direkt zusammengehörige Klassen verwendet, die eine bestimmte Aufgabe lösen, für die die Komponente verantwortlich ist. Die Komponenten werden nachfolgend einzeln erläutert. Es gibt eine zentrale Komponente (im Folgenden als `Prototype`-Komponente bezeichnet), die den Ablauf des Prototypen nach dem Starten des Eclipse-Plugins kontrolliert, eine Komponente für die Einbindung des Prototypen in Eclipse (im Folgenden als `Eclipse-Plugin`-Komponente bezeichnet), eine Kom-

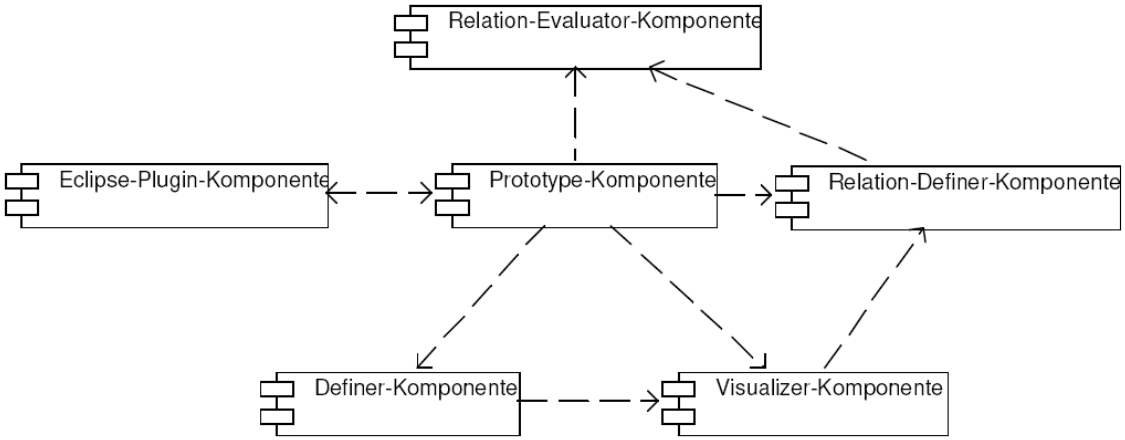

Abbildung 7.1: Die Architektur des Prototypen, die Pfeile geben die Abhängigkeiten unter den Komponenten an. Diese Abhängigkeiten ergeben sich dadurch, dass die `Prototype`-Komponente den Ablauf steuert sowie die anderen Komponenten Daten der Komponenten, von denen sie abhängig sind, voraussetzen.

ponente um die Informationen zu Paketen, Schnittstellen und Klassen aus der XML-Eingabedatei zu verarbeiten (im Folgenden als `Definer`-Komponente bezeichnet), eine Komponente um die in der XML-Eingabedatei beschriebenen `Walls`, `Cone Trees` und `Information Cubes` zu verarbeiten (im Folgenden als `Visualizer`-Komponente bezeichnet), eine weitere Komponente um aus den verarbeiteten Informationen Relationen anzulegen (im Folgenden als `Relation-Definer`-Komponente bezeichnet), und letztlich noch eine Komponente, welche die angelegten Relationen auswertet (im Folgenden als `Relation-Evaluator`-Komponente bezeichnet).

Durch die lose Kopplung sind die Komponenten nicht direkt aneinander gebunden. Dies wurde dadurch realisiert, dass lediglich die `Prototype`-Komponente den Ablauf des Prototypen und aller Komponenten steuert. Von ihr ausgehend werden die anderen Komponenten aufgerufen. Jede Komponente legt Daten dann zentral in der Klasse `ComponentMap` ab (später näher erläutert), auf die andere Komponenten dann bei Bedarf zugreifen können. Auf diese Weise können einzelne Komponenten ausgetauscht oder hinzugefügt werden, die andere Arbeitsschritte vornehmen, aber die gleichen Daten zentral ablegen, ohne dass die anderen Komponenten dazu geändert werden müssen.

Die lose Kopplung zwischen den Komponenten, welche zur leichten Anpassung und für zukünftige Erweiterungen erforderlich ist, führt zu einer längeren Einarbeitungsphase, wenn man den Prototypen analysiert. Eine andere Möglichkeit hätte darin bestanden, den Prototypen in wenigen stark miteinander durch Aufrufe verbundenen Klassen zu entwickeln, welche direkt aus der XML-Eingabedatei Relationen ableiten und die Paket-, Schnittstellen- und Klassenstrukturen in Eclipse erstellen. Diese Möglichkeit, welche unter Umständen schneller abgearbeitet ist, wäre jedoch nicht als Basis für ein späteres Eclipse-Plugin zu benutzen. Damit der Prototyp für eine spätere Neuentwicklung genutzt werden kann, ist es erforderlich, dass man ihn auch mit anderen Daten als der beschriebenen XML-Eingabedatei starten kann. Dabei wird ein richtiges Eclipse-Plugin zur Geschwindigkeitssteigerung für die Dateneingabe keine Dateien benutzen, sondern Objekte im Hauptspeicher. Dies ist mit der gewählten Architektur möglich, da einzelne Komponente gegen entsprechende anders implementierte ausgetauscht werden können.

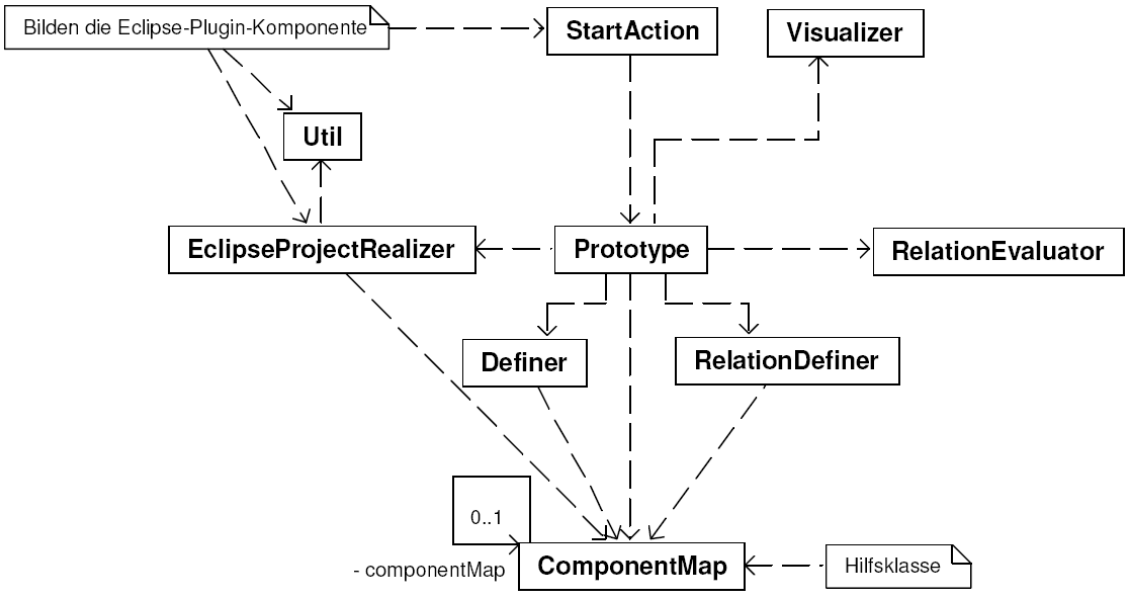

Abbildung 7.2: Der Entwurf des Prototypen

7.2.1 Die zentrale `Prototype`-Komponente

Die zentrale Komponente des Prototypen ist für den Arbeitsablauf des Prototypen verantwortlich. Sie ist in der Klasse `Prototype` realisiert. Die Klasse wird von der später beschriebenen Elipse-Plugin-Komponente aufgerufen. Diese Klasse liest die XML-Datei über die *Java API for XML Binding* ein (nachfolgend in Abschnitt 7.2.1 auf Seite 65 erläutert), und nutzt andere Klassen um die XML-Struktur zu verarbeiten, Relationen anzulegen und die Relationen auf Eigenschaften zu prüfen. Verlaufen die Prüfungen zufriedenstellend, so legt der Prototype die in der XML-Datei beschriebenen Pakete, Schnittstellen und Klassen in Eclipse an, ordnet sie in den richtigen Paketen an und trägt die Beziehungen `extends` und `implements` korrekt ein.

Die Klasse `Prototype` ist in dem UML-Klassendiagramm in Abbildung 7.2 auf Seite 62 ersichtlich. Die einzelnen Methoden werden im Anhang A erläutert.

XML-Struktur der Szenen-Eingabedatei

Die XML-Struktur der Eingabedatei unterliegt dem in der Datei `prototype.xsd` abgelegten XML-Schema. Ein XML-Schema gibt Regeln vor, wie eine XML-Datei aussehen darf, welche Elemente vorkommen, wie diese angeordnet sein können und welche Attribute die Elemente besitzen. Das Schema, sowie eine Beispiel-XML-Datei, welche nach den Regeln des Schemas aufgebaut ist, ist im Anhang B ersichtlich. Durch das genannte XML-Schema ist folgende XML-Struktur vorgeschrieben (die Struktur ist in Abbildung 7.3 auf Seite 63 schematisch dargestellt):

Das Wurzelelement der XML-Struktur ist das Element `root`. Dieses Element hat keine Attribute, aber die drei Kindelemente `config`, `components` und `visualize`. In dem Element `config` wird der Prototyp konfiguriert, in dem Element `components` die in der Visualisierung benutzten Komponenten definiert und in dem Element `visualize` die Beschreibung der Szene angegeben.

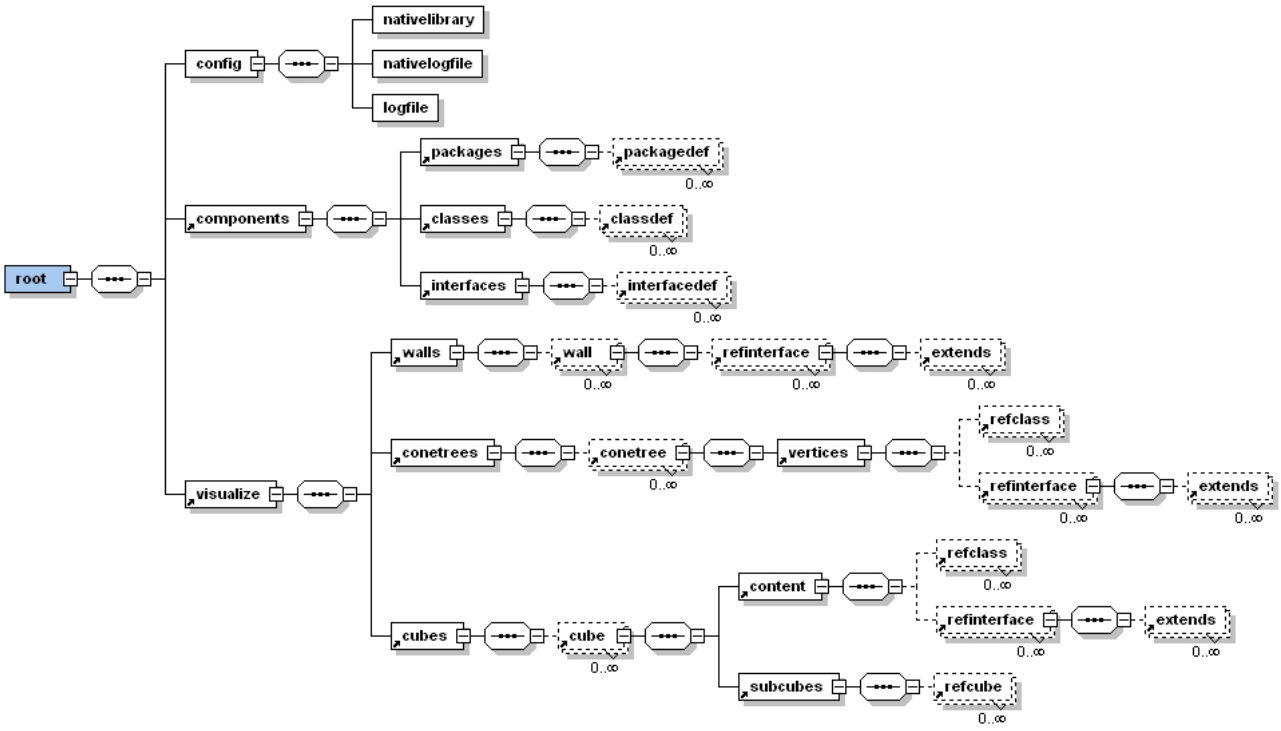

Abbildung 7.3: Die Struktur einer XML-Datei des Prototypen

Die drei Elemente werden im Folgenden näher erläutert.

Das Element `config` besitzt keine Attribute. Es hat die drei Kindelemente `nativelibrary`, `nativelogfile` und `logfile`. Über das Element `nativelibrary` wird mit dem Attribut `uri` der absolute Dateiname der Java-Bibliothek für den Zugriff auf den funktionalen Kern von RELVIEW angegeben. Das Element `nativelogfile` gibt in dem Attribut `uri` den absoluten Dateinamen der Datei an, in den alle Meldungen der Bibliothek geschrieben werden. Das Element `logfile` gibt in dem Attribut `uri` den absoluten Dateinamen der Datei an, in die alle weiteren Ausgaben des Prototyps erfolgen. Die drei Elemente besitzen keine weiteren Attribute oder Kindelemente.

Das Element `components` besitzt keine Attribute und hat die Kindelemente `packages`, `classes` und `interfaces`. Diese Kindelemente geben jeweils die Pakete, Klasseb und Schnittstellen an, für die die Visualisierung im später beschriebenen Element `visualize` angegeben ist. Die drei Kindelemente besitzen keine Attribute.

Das Element `packages` beinhaltet beliebig viele Elemente `packagedef`. Jedes dieser Elemente steht für ein Paket. Jedes Element besitzt die zwei Attribute `id` und `name`. Das Attribut `id` ist ein innerhalb der XML-Datei eindeutiger Bezeichner, mit dessen Hilfe in anderen Teilen der XML-Datei auf das Paket verwiesen werden kann. Das Attribut `name` beinhaltet den Java-Paketnamen.

Das Element `interfaces` besitzt beliebig viele Elemente `interfacedef`, ebenfalls mit dem Attribut `id` und `name`. Das Attribut `name` beinhaltet dabei den Java-Schnittstellennamen ohne Paketpfad.

Das Element `classes` besitzt beliebig viele Elemente `classdef` mit den gleichen zwei Attributen.

Das Attribut `name` gibt den Java-Klassennamen ohne Paketpfad an.

Das Element `visualize` besitzt keine Attibute aber die drei Kindelemente `walls`, `conetrees` und `cubes`, in denen jeweils die Darstellungen von `Walls`, `Cone Trees` und `Information Cubes` beschrieben werden. Alle drei Kindelemente besitzen keine Attribute. Das Element `walls` besitzt beliebig viele Kindelemente `wall`, das Element `conetrees` beliebig viele Kindelemente `conetree` und das Element `cubes` beliebig viele Kindelemente `cube`.

Das Element `wall` hat das Attribut `id` mit einem innerhalb der XML-Struktur eindeutigen Bezeichner und beliebig viele Kindelemente `refinterface`. Ein Element `refinterface` repräsentiert dabei eine Java-Schnittstelle, welche in die `Wall` gehört. Das Element `refinterface` hat als Attribut `ref-id` einen Verweis auf eine vorher unterhalb des Elementes `components` angegebenen Java-Schnittstelle. Als Kindelement sind beliebig viele Elemente `extends` möglich, welche die `extends`-Verbindungspfeile innerhalb der `Wall` darstellen und anzeigen, welche anderen Schnittstellen die Schnittstelle erweitert. Das Element `extends` hat dabei keine Kindelemente und nur das Attribut `ref-id` mit dem Verweis auf die vorher angegebenen Schnittstellen.

Das Element `conetree` hat das Attribut `id` mit einem innerhalb der XML-Struktur eindeutigen Bezeichner, das Attribut `ref-id` mit einem Verweis auf eine vorher unter dem Element `components` angegebenen Klasse, welche die Spitze des `Cone Trees` repräsentiert und beliebig viele Kindelemente `vertices`.

Das Element `vertices` hat keine Attribute und beliebig viele Kindelemente. Als Kindelemente sind dabei ausschließlich `refinterface` und `refclass` erlaubt.

Das Element `cube` hat das Attribut `id` mit einem innerhalb der XML-Struktur eindeutigen Bezeichner, das Attribut `ref-id` mit einem Verweis auf eine vorher unter dem Element `components` angegebenen Paket, welches dieser `Information Cube` repräsentiert und die Kindelemente `content` und `subcubes`. Das Element `content` enthält die oben beschriebenen Elemente `refinterface` und `refclass` und repräsentiert die Schnittstellen und Klassen, welche in dem dem `Information Cube` zugeordneten Paket liegen.

Das Element `subcubes` beinhaltet beliebig viele Elemente `refcube`, die keine Kindelemente und nur das Attribut `ref-id` auf einen bereits vorher angegebenen `Information Cube` besitzen. Dieses Element wird für die Beschreibung von verschachtelten Paketen benutzt.

Die beschriebene Struktur der XML-Szenendatei ist für die Problemstellung geeignet, da sie ermöglicht, die drei notwendigen dreidimensionalen Darstellungsformen `Information Cubes`, `Cone Trees` und `Walls` zu beschreiben. Es können Identifizierer, also eindeutige Stellvertreter für Schnittstellen und Klassen angegeben werden, die bei der Definition von `Information Cubes`, `Cone Trees` und `Walls` benutzt werden können. Dies entspricht dem Vorgang, eine Klasse als Box und eine Schnittstelle durch eine Kugel zu repräsentieren, welche dann in den dreidimensionalen Diagrammen als Symbole für Klassen und Schnittstellen auftreten. `Information Cubes` können dabei Schnittstellen und Klassen beinhalten und selbst verschachtelt werden. Somit können Paketstrukturen beschrieben werden. `Cone Tree` haben genau eine Klasse an der Spitze und es kann angegeben werden, welche weiteren Klassen oder Schnittstellen in dem `Cone Tree` liegen, um Vererbungshierarchien zu beschreiben. `Walls` beinhalten Schnittstellen die untereinander verbunden sein können, um Schnittstellenhierarchien zu beschreiben. Damit kann die XML-Struktur alle notwendigen Aspekte der dreidimensionalen Szenen aus `Information Cubes`, `Cone Trees` und `Walls` beschreiben.

Java-XML-Binding

Um die XML-Datei im Prototypen zu verarbeiten wurde die *Java Architecture for XML Binding*, kurz *JAXB* verwendet. Mit *JAXB* generiert man aus einem XML-Schema Java-Klassen, welche dann der möglichen Struktur der XML-Dateien entsprechen. *JAXB* erstellt zu jedem Element, welches in dem Schema beschrieben wird, eine zugehörige Schnittstelle und eine Klasse, welche die Schnittstelle implementiert. Eine Instanz der Klassen repräsentiert ein konkretes Element aus einer XML-Datei, welche zu dem Schema gehört. In generierten Attributen der Klasse wird der Inhalt des Elementes vermerkt. Mit *JAXB* läßt sich eine XML-Datei einlesen. Für jedes Element in der XML-Datei legt *JAXB* dabei eine Instanz der entsprechenden generierten Klasse an. Das Wurzelelement wird nach dem Einlesen der XML-Datei als Instanz der dafür generierten Klasse zurückgeliefert. Mit diesem Wurzelobjekt kann man im Folgenden die komplette XML-Datei verarbeiten, da die generierten Klassen Methoden besitzen, um Kindelemente und Attribute abzufragen. Die generierten Klassen und Schnittstellen für den Prototypen liegen im Java-Paket `de.osz.eclipseprototype.jaxb`.

Die generierte Schnittstelle für das Wurzelelement der XML-Szenendatei des Prototypen sieht wie folgt aus:

```
public interface RootType {

    RootType.ConfigType getConfig();
    void setConfig(RootType.ConfigType value);
    ComponentsType getComponents();
    void setComponents(ComponentsType value);
    VisualizeType getVisualize();
    void setVisualize(VisualizeType value);

    public interface ConfigType {
        RootType.ConfigType.LogfileType getLogfile();
        void setLogfile(RootType.ConfigType.LogfileType value);
        RootType.ConfigType.NativelibraryType getNativelibrary();
        void setNativelibrary(RootType.ConfigType.NativelibraryType value);
        RootType.ConfigType.NativelogfileType getNativelogfile();
        void setNativelogfile(RootType.ConfigType.NativelogfileType value);

        public interface LogfileType {
            String getUri();
            void setUri(String value);
        }

        public interface NativelibraryType {
            String getUri();
            void setUri(String value);
        }

        public interface NativelogfileType {
            String getUri();
            void setUri(String value);
        }
    }
}
```

In der generierten Schnittstelle ist ersichtlich, dass es Methoden gibt, um auf die Kindlelemente des Wurzelelementes zuzugreifen und auch den Inhalt von Elementen und ihren Attributen auszulesen und zu manipulieren.

Wenn man mit *JAXB* die XML-Szenendatei einliest, erzeugt *JAXB* also Instanzen der generierten Klassen und füllt die Attribute dieser Instanzen mit den eingelesenen Werten der XML-Szenendatei. Diese Objekte werden dann im Prototypen weiter verarbeitet. Die von *JAXB* generierten Klassen und Schnittstellen befinden sich im Prototypen im Paket `de.osz.eclipseprototype.jaxb`. Nähere Information zu *JAXB* sind unter [18] ersichtlich.

7.2.2 Hilfsklassen für die Komponenten

Bei den im Folgenden beschriebenen Klassen `PackageComponent`, `InterfaceComponent` `ClassComponent` sowie `WallComponent`, `ConetreeComponent`, `CubeComponent` und `ComponentMap` handelt es sich um Hilfsklassen, die zur Speicherung und Verfügungstellung von Informationen benötigt werden.

Die Klassen `PackageComponent`, `InterfaceComponent` und `ClassComponent`

Instanzen der Klassen `PackageComponent`, `InterfaceComponent` und `ClassComponent` sind reine Datenhalter und werden jeweils über einen Konstuktor mit den zwei Parametern `String id` und `String name` erzeugt. Bei dem Parameter `id` handelt es sich um den in der XML-Datei eindeutig vergebenen Identifizierer des Paketes, der Schnittstelle, bzw. der Klasse. Bei dem Parameter `name` handelt es sich um den zugehörigen Java-Bezeichner. Die Instanzen kapseln die beiden Zeichenketten und besitzen entsprechende `get`-Methoden um die Werte abzufragen. Diese Hilfsklassen werden nicht in den UML-Diagrammen abgebildet, da sie ausser dem Kapseln von Daten keine weitere Funktionalität ausüben.

Die Klasse `WallComponent`

Die Klasse `WallComponent` kapselt die Angaben zu einer `Wall` und ist im UML-Klassendiagramm in Abbildung 7.4 auf Seite 69 abgebildet. Instanzen der Klasse `WallComponent` werden über einen Konstruktor mit dem Parameter `id` erzeugt. Bei dem Parameter `id` handelt es sich um den in der XML-Datei eindeutig vergebenen Identifizierer der `Wall`. Instanzen kapseln den Wert und bieten eine entsprechende `get`-Methode um den Wert abzufragen. Darüber hinaus kapseln Instanzen Schnittstellenidentifizierer, die in der `Wall` in dem Attribut `private Map interfaces` enthalten sind. In der `Map` werden alle Schnittstellen die Teil der `Wall` sind eingetragen und jeweils dazu in einer Menge vermerkt, zu welchen anderen Schnittstellen in der `Wall` `extends`-Beziehungen bestehen. Die Methoden, die die Klasse dazu anbietet, sind im Anhang A ersichtlich.

Die Klasse `ConetreeComponent`

Die Klasse `ConetreeComponent` kapselt die Angaben zu einem `Cone Tree`, (siehe Abbildung 7.4 auf Seite 69). Die Instanzen dieser Klasse werden über einen Konstruktor mit dem Parameter `id`

und dem Parameter `topid` erzeugt. Bei dem Parameter `id` handelt es sich um den in der XML-Datei eindeutig vergebenen Identifizierer des `Cone Trees`. Bei dem Parameter `topid` handelt es sich um den in der XML-Datei eindeutig vergebenen Identifizierer der Klasse, welche die Spitze des `Cone Trees` repräsentiert. Wie bei der Klasse `WallComponent` werden diese Werte, sowie die Schnittstellenidentifizierer und Klassenidentifizierern, die auf Schnittstellen und Klassen verweisen, welche Knoten des `Cone Trees` darstellen, gekapselt. Die Schnittstellenidentifizierer werden in dem Attribut `private Set interfaces` und die Klassenidentifizierer in dem Attribut `private Set classes` abgelegt. Die Klasse besitzt entsprechende `add`- und `get`-Methoden, um Identifizierer den Mengen hinzuzufügen und die gesamten Mengen abzufragen.

Die Klasse `CubeComponent`

Die Klasse `CubeComponent` kapselt die Angaben zu einem `Information Cube` und ist im UML-Klassendiagramm in Abbildung 7.4 auf Seite 69 abgebildet. Die Instanzen der Klasse werden über einen Konstruktor mit dem Parameter `id` und dem Parameter `packageid` erzeugt. Bei dem Parameter `id` handelt es sich um den in der XML-Datei eindeutig vergebenen Identifizierer des `Information Cubes`. Bei dem Parameter `packageid` handelt es sich um den in der XML-Datei eindeutig vergebenen Identifizierer des Paketes, zu welchem der `Information Cube` gehört. Instanzen kapseln die Werte und bieten entsprechende `get`-Methoden um diese Werte abzufragen. Darüber hinaus kapseln Instanzen je eine Menge von Schnittstellenidentifizieren, Klassenidentifizierern und Identifizieren auf andere `Information Cubes`, die auf Schnittstellen, Klassen und untergeordnete `Information Cubes` verweisen, die in dem repräsentierten `Information Cube` enthalten sind. Die Schnittstellenidentifizierer werden in dem Attribut `private Set interfaces`, die Klassenidentifizierer in dem Attribut `private Set classes` und die `Information Cube`-Identifizierer in dem Attribut `private Set cubes` abgelegt. Die Klasse besitzt entsprechende `add`- und `get`-Methoden um Identifizierer den Mengen hinzuzufügen und die gesamten Mengen abzufragen.

Die Klasse `ComponentMap`

Die Klasse `ComponentMap` dient als gemeinsamer Datenspeicher der Komponenten und ist in dem UML-Klassendiagramm in Abbildung 7.2 auf Seite 62 abgebildet. Objekte dieser Klasse erlauben es andere Objekte unter einem Namen abzulegen und unter diesem Namen auf die abgelegten Objekte zuzugreifen. Von der Klasse `ComponentMap` kann zur Laufzeit nur ein Objekt existieren, welches in dem statischen Attribut `componentMap` abgelegt ist; es handelt sich dabei um das Singleton-Entwurfsmuster. Das Singleton-Objekt bekommt man bei einem Aufruf der statischen Methode `getComponentMap` dieser Klasse zurückgeliefert. Existiert das Objekt nicht, wird es in dieser Methode angelegt und in dem statischen Attribut `componentMap` abgelegt. Das Singleton-Entwurfsmuster ist erforderlich, um in den anderen Klassen des Prototypen genau die Instanz zu bekommen, in der bereits Objekte abgelegt wurden. Damit die einzelnen Komponenten des Prototypen alle Daten zur Verfügung haben, welche die anderen Komponenten während des Arbeitsablaufes des Prototypen für sie bereitstellen, aber dennoch mit den anderen Komponenten nicht direkt in Aktion treten, ist ein gemeinsamer Datenspeicher, hier in Form des Singleton-Objektes, notwendig. Jede Komponente kann in dem Objekt die Ergebnisse seiner Arbeitsschritte ablegen, so dass die anderen darauf zugreifen können. Eine Alternative hätte in jedem anderen gemeinsamen Datenspeicher

bestanden, wie z.B. einer Datenbank. Der Zugriff auf ein simples Singleton-Objekt, in dem Daten abgelegt werden können ist effizienter, da ein Methodenaufruf an diesem Objekt deutlicher schneller verarbeitet werden kann, als eine Anfrage an eine Datenbank. Daher bietet sich die Realisierung des zentralen Datenspeichers als Singleton-Objekt, auf das alle Komponenten zugreifen können, an.

Eine Instanz dieser Klasse verwaltet sechs `Maps`:

- `protected Map packagedef`
  `Map` mit Paketdefinition aus der XML-Struktur
- `protected Map classdef`
  `Map` mit Klassendefinition aus der XML-Struktur
- `protected Map interfacedef`
  `Map` mit Schnittstellendefinition aus der XML-Struktur
- `protected Map wall`
  `Map` mit `Wall`-Definition aus der XML-Struktur
- `protected Map conetree`
  `Map` mit `Cone Tree`-Definition aus der XML-Struktur
- `protected Map cube`
  `Map` mit `Information Cube`-Definition aus der XML-Struktur

Diese `Maps` werden von den Komponenten des Prototypen mit Werten gefüllt, bzw. abgefragt. Die Methode `reset` dieser Klasse leert die sechs `Maps` und wird vor einem erneuten Arbeitsablauf des Prototypen benötigt, um nicht alte Daten des vorherigen Arbeitsablaufes zu benutzen.

7.2.3 Die `Definer`-Komponente

Die `Definer`-Komponente ist dafür verantwortlich, die Angaben zu Paketen, Schnittstellen und Klassen, welche in der XML- Eingabedatei abgelegt sind, zu verarbeiten. Diese werden von der `Visualizer`-Komponente benötigt, um die Strukturen von `Information Cubes`, `Walls` und `Cone Trees` zu verarbeiten, welche aus diesen Paketen, Schnittstellen und Klassen bestehen. Die `Definer`-Komponente ist in der Klasse `Definer` realisiert. Die Klasse ist in dem UML-Klassendiagramm in Abbildung 7.2 auf Seite 62 abgebildet. Eine Instanz der Klasse `Definer` bekommt beim Erzeugen das Wurzelobjekt und den Ausgabestrom für Meldungen im Konstruktor übergeben. Die Klasse greift über das Wurzelobjekt auf das Element `components` zu und vermerkt die dort angegebenen Pakete, Schnittstellen und Klassen jeweils als Instanzen der Klassen `PackageComponent`, `InterfaceComponent` und `ClassComponent` in den `Maps` `packagedef`, `interfacedef` und `classdef` der Instanz der Klasse `ComponentMap`. Dies geschieht jeweils in den Methoden `definePackages`, `defineInterfaces` und `defineClasses`, die im Anhang A erläutert sind.

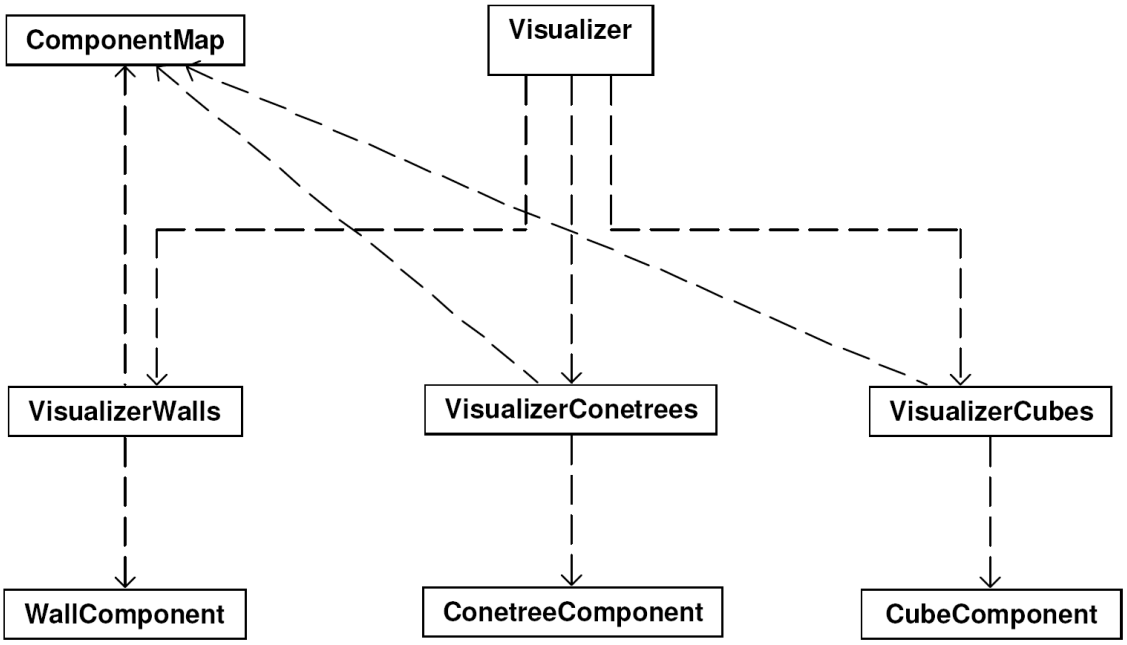

Abbildung 7.4: Die Klassen `Visualizer`, `VisualizerWalls`, `VisualizerConetrees` und `VisualizerCubes` sowie die Hilfsklassen `WallComponent`, `ConetreeComponent` und `CubeComponent`, welche die Komponente `Visualizer` bilden und die Hilfsklasse `ComponentMap`

7.2.4 Die `Visualizer`-Komponente

Die `Visualizer`-Komponente ist verantwortlich, die in der XML-Datei gemachten Angaben zur Definition von `Information Cubes`, `Walls` und `Cone Trees`, also zu der Visualisierung von Paket-, Schnittstellen- und Klassenstrukturen, zu verarbeiten. Die Komponente besteht aus den Klassen `Visualizer`, `VisualizerWalls`, `VisualizerConetrees` und `VisualizerCubes`, die im Folgenden erläutert werden.

Die Klasse Visualizer

Die Klasse `Visualizer` ist in dem UML-Klassendiagramm in Abbildung 7.4 auf Seite 69 abgebildet, sowie in dem UML-Klassendiagramm 7.2 auf Seite 62.

Eine Instanz der Klasse `Visualizer` bekommt beim Erzeugen das von *JAXB* erstellte Objekt, welches das Element `visualize` der XML-Datei repräsentiert und den Ausgabestrom für Meldungen im Konstruktor übergeben. Die Klasse erstellt die einzelnen Instanzen der oben beschriebenen Hilfsklassen `WallComponent`, `ConetreeComponent` und `CubeComponent`, die später genutzt werden um die Relation zu erstellen. Die Objekte werden nach den Angaben in der XML-Datei erstellt. Dazu nutzt sie die drei Klassen `VisualizerWalls`, `VisualizerConetrees` und `VisualizerCubes`. Die Klasse hat nur die Methode `defineVisualize`. In dieser Methode werden Objekte der drei gerade genannten Klassen erstellt. Es werden jeweils die entsprechenden `define`-Methoden dieser Objekte aufgerufen.

Die Klasse VisualizerWalls

Die Klasse `VisualizerWalls` ist in dem UML-Klassendiagramm in Abbildung 7.4 auf Seite 69 abgebildet. Die Klasse `VisualizerWalls` ist dafür verantwortlich, die `wall`-Elemente der XML-Struktur zu verarbeiten und entsprechende Instanzen der Klasse `WallComponent` zu erzeugen. Dazu geht sie in der Methode `defineWalls` alle `wall`-Elemente durch und erzeugt jeweils eine Instanz der Klasse `WallComponent` und ruft damit die Methode `defineWallInterfaces` auf. Die Methode `defineWallInterfaces` verarbeitet für ein `wall`-Element die Kindelemente `refinterface`, die Schnittstellen repräsentieren, welche sich innerhalb der `Wall` befinden. Diese werden bei der Instanz der Klasse `WallComponent` hinzugefügt. Für jede in der `Wall` enthaltenen Schnittstelle werden die Kindelemente `extends` verarbeitet und über entsprechende `add`-Methoden der Instanz der Klasse `WallComponent` vermerkt, zu welchen anderen Schnittstellen die Schnittstelle in der `Wall` eine `extends`-Beziehung hat.

Die Klasse VisualizerConetrees

Die Klasse `VisualizerConetrees` ist in dem UML-Klassendiagramm in Abbildung 7.4 auf Seite 69 abgebildet. Die Klasse `VisualizerConetrees` ist verantwortlich, die `conetree`-Elemente der XML-Struktur zu verarbeiten und entsprechende Instanzen der Klasse `ConetreeComponent` zu erzeugen. Dazu verarbeitet sie in der Methode `defineConetrees` alle `conetree`-Elemente und erzeugt jeweils eine Instanz der Klasse `ConetreeComponent` und ruft damit die Methoden `defineConetreeVerticesInterface` und `defineConetreeVerticesClass` auf. Diese Methoden ermitteln jeweils die Kindelemente `refinterface` und `refclass` und fügen die aus dem Attribut `ref-id` erhaltenen Identifizierer über entsprechende `add`-Methoden der vorher erzeugten Instanz der Klasse `ConetreeComponent` hinzu.

Die Klasse VisualizerCubes

Die Klasse `VisualizerCubes` ist in dem UML-Klassendiagramm in Abbildung 7.4 auf Seite 69 abgebildet. Die Klasse `VisualizerCubes` ist dafür verantwortlich, die `cube`-Elemente der XML-Struktur zu verarbeiten und entsprechende Instanzen der Klasse `CubeComponent` zu erzeugen. Dazu geht sie in der Methode `defineCubes` alle `cube`-Elemente durch und erzeugt jeweils eine Instanz der Klasse `CubeComponent` und ruft damit die Methoden `defineCubePackageEntries`, `defineCubeInterfaceEntries` und `createCubeClassEntries` auf. Diese Methoden ermitteln jeweils die Kindelemente `refcube`, `refinterface` und `refclass` und fügen die aus dem Attribut `ref-id` erhaltenen Identifizierer der Instanz der Klasse `CubeComponent` über entsprechende `add`-Methoden hinzu.

7.2.5 Die `Relation-Definer`-Komponente

Die `Relation-Definer`-Komponente ist dafür verantwortlich, aus den von der `Visualizer`-Komponente ermittelten Daten zur Struktur von `Information Cubes`, `Walls` und `Cone Trees` alle benötigten Relationen zu generieren, welche die `Relation-Evaluator`-Komponente anschließend auswerten kann. Diese Relationen werden mit der in der vorliegenden Arbeit entwickelten

Java-Bibliothek angelegt. Die Komponente wird durch die Klasse `RelationDefiner` realisiert. Die Klasse ist in dem UML-Klassendiagramm in Abbildung 7.2 auf Seite 62 abgebildet. Sie bekommt den Ausgabestrom für Meldungen im Konstruktor übergeben. Hat man eine Instanz dieser Klasse erstellt, dient die Methode `defineRelations` dazu, alle Relationen zu erstellen. Die Methode ruft dazu weitere im Anhang A erläuterte spezielle Methoden mit dem Präfix `define` auf, um jeweils eine der benötigten Relationen zu erstellen. In diesen Methoden werden entsprechende, ebenso im Anhang A erläuterte Methoden mit dem Präfix `setEntries` aufgerufen, um die Einträge in den Relationen zu setzen.

7.2.6 Die `Relation-Evaluator`-Komponente

Die `Relation-Evaluator`-Komponente ist verantwortlich, die von der `Relation-Definer`-Komponente erstellten Relationen auszuwerten. Diese Komponente zur Auswertung wird durch die Klasse `RelationEvaluator` realisiert. Die Klasse `RelationEvaluator` ist in dem UML-Klassendiagramm in Abbildung 7.2 auf Seite 62 abgebildet. Die Klasse `RelationEvaluator` prüft die definierten Relationen auf Eigenschaften. Mit Hilfe dieser Klasse ermittelt der Prototyp, ob die ihm übergebene Szene gültige statische Beziehungen von Paketen, Schnittstellen und Klassen in Java visualisiert und somit zu der Szene Java-Quellcode generiert werden kann. Dem Konstruktor der Klasse wird der Ausgabestrom für Meldungen als Parameter übergeben.

- ```
  private void createFunctions()
      throws FunctionException
  ```

  In dieser Methode werden nachfolgend erläuterte relationale Funktionen definiert, die in Termauswertungen in der Methode `evaluateJavaBindingRelations` benötigt werden. Die Methode `createFunctions` wird in dem Konstruktor der Klasse aufgerufen.

  – Eigenbeziehung

    Diese Funktion ermittelt, ob in einer Relation, in welcher der Vor- und Nachbereich der Relation gleich ist, ein Objekt zu sich selbst in Beziehung steht. Die Funktion ist wie folgt definiert:

    $selfrelationed(R) = -empty(I(R)\&R)$

    Die Definition wurde gebildet, da falls in einer Relation ein Objekt mit sich selbst verbunden ist, der Durchschnitt der Relation mit der Identität der Relation nicht leer ist. D.h. ein Eintrag auf der Hauptdiagonalen der Relationsmatrix muss gesetzt sein.

  – Geschlossener Zyklus

    Diese Funktion ermittelt, ob in einer Relation ein geschlossener Zyklus vorliegt. Dies ist der Fall, wenn ein Objekt `o1` zu einem anderen Objekt `o2` in Relation steht, dieses zu einem weiteren Objekt, usw. bis eines dieser Objekte wieder zu dem ersten Objekt `o1` in Relation steht. Dies ist äquivalent zu der Frage, ob ein Objekt in dem transitiven Abschluss der Relation zu sich selbst in Relation steht. Somit wurde die Funktion wie folgt definiert:

    $cycle(R) = selfrelationed(trans(R))$

  – Injektiv

    Diese Funktion ermittelt, ob eine Relation injektiv ist. Eine Relation ist injektiv, wenn jedes Objekt maximal zu einem Objekt in Relation steht. Wenn eine Relation R nicht
  ```

injektiv ist, dann existieren für ein Element x im Vorbereich der Relation R mindestens zwei verschiedene Elemente y und z aus dem Nachbereich. Das heißt, in der Relation $R^\wedge$ ist x im Nachbereich und y und z im Vorbereich und die beiden Paare (y, x) und (z, x) sind in $R^\wedge$ enthalten. In der Komposition $R * R^\wedge$ sind die Paare (y, z) und (z, y) enthalten. Da das Element y ungleich dem Element z ist, ist die Komposition nicht in der Identität enthalten.

Daher wird der Test auf Injektivität durch folgende Definition realisiert:
$injective(R) = incl(R * R^\wedge, I(R * R^\wedge))$

- `protected void evaluateJavaBindingRelations()`
  `throws RelationException,`
  `TermException,`
  `FunctionException,`
  `CreateException`

  In dieser Methode werden die vorher beim Arbeitsablauf des Prototypen erstellten Relationen auf bestimmte Eigenschaften geprüft und somit festgestellt, ob aus der dem Prototypen übergebene Szene gültige Pakete, Schnittstellen und Klassen in Java generiert werden können. Wird der Aufruf dieser Methode korrekt beendet, sind alle Prüfungen zufriedenstellend verlaufen, ansonsten wird in der Methode die Ausnahme `CreateException` geworfen. Innerhalb der Methode werden folgende Eigenschaften geprüft:

  – Eigenbeziehung

    Zuerst werden die drei Relation `packages`, `interfaces` und `classes`, bei denen Vor- und Nachbereich gleich sind, darauf geprüft, ob ihre Objekte nicht mit sich selbst in Beziehung stehen. Dies geschieht mit Hilfe der definierten Funktion $selfrelationed$. Wenn ein Objekt zu sich selbst in Beziehung steht, verletzt das die Java-Syntax. Weder kann ein Paket in Java Unterpaket von sich selbst sein, noch kann eine Klasse oder eine Schnittstelle sich selbst erweitern. Gilt für eine der Relationen, dass sie die Eigenschaft erfüllt, wird eine Ausnahme der Klasse `CreateException` ausgelöst.

  – Zyklenfreiheit

    Danach wird geprüft, ob die drei Relation `packages`, `interfaces` und `classes` zyklenfrei sind. Dies geschieht mit Hilfe der vorher definierten Funktion $cycle$. Enthält eine der Relationen einen Zyklus, hat dies dieselbe Bedeutung wie in dem Falle, dass ein Objekt mit sich selbst in Beziehung steht, nur ist diese Beziehung nicht direkt durch die Szenenbeschreibung sondern indirekt durch die Kette von Beziehungen zwischen den Objekten gegeben. Dies würde bedeuten, dass in dem Vererbungsgraphen der Klassen oder Schnittstellen oder in dem Paketbaum Kreise existieren, was in Java ausgeschlossen ist. Ist eine der Relationen nicht zyklenfrei, wird die Ausnahme `CreateException` geworfen.

  – Injektivität

    Die Relationen `packages`, `classes`, `packagesInt` und `packagesClasses` werden darauf geprüft, ob ihre Transpositionen injektiv sind. Dazu wird von jeder dieser Relationen die Transposition mit dem Operator $\wedge$ gebildet und darauf die Funktion $injective$ angewendet. Die Eigenschaft ob die Transposition injektiv ist, bedeutet bei der Relation `packages`, dass jedes Paket maximal in einem übergeordneten Paket vorhanden sein darf. Bei der Relation `classes` bedeutet es, dass jede Klasse maximal eine Superklasse

haben darf, die sie erweitert. Bei der Relation `packagesInt` ist die Eigenschaft injektiv notwendig, damit jede Schnittstelle höchstens in einem Paket liegt, gleiches gilt bei der Relation `packagesClasses` für Klassen. Ist eine der Transpositionen der genannten Relationen nicht injektiv, so wird eine Ausnahme vom Typ `CreateException` ausgelöst.

7.2.7 Die `Eclipse-Plugin`-Komponente

Die Anbindung an Eclipse geschieht über die `Eclipse-Plugin`-Komponente. Die Komponente ist dafür verantwortlich, den Prototypen in die Entwicklungsumgebung und die graphische Oberfläche von Eclipse zu integrieren und innerhalb von Eclipse Pakete, Schnittstellen und Klassen aus den durch die `Visualizer`-Komponente zur Verfügung gestellten Informationen zu erzeugen. Die `Eclipse-Plugin`-Komponente wird durch die Klassen `Util`, `EclipseProjectRealizer` `PrototypePlugin` und `StartAction` realisiert, welche nachfolgend erläutert werden.

Die Klasse `Util`

Die Klasse `Util` ist eine weitere Hilfklasse. Sie ist in dem UML-Klassendiagramm in Abbildung 7.2 auf Seite 62 abgebildet. Sie besitzt vier öffentliche statische Methoden zum Erzeugen von Paketen, Schnittstellen und Klassen in Eclipse, die im Anhang A erläutert werden.

Die Klasse `EclipseProjectRealizer`

Die Klasse `EclipseProjectRealizer` ist in dem UML-Klassendiagramm in Abbildung 7.2 auf Seite 62 abgebildet. Die Klasse `EclipseProjectRealizer` ist dafür verantwortlich, die Pakete, Schnittstellen und Klassen in Eclipse zu generieren, welche in der dem Prototypen als XML-Datei übergebenen Szene angegeben sind. Dazu nutzt sie die oben beschriebene Klasse `Util`. Um eine Instanz der Klasse `EclipseProjectRealizer` zu erzeugen, muss man dem Konstruktor das Eclipse-Projekt in Form eines Objektes, das die von Eclipse definierte Schnittstelle `IJavaProject` implementiert, übergeben. Innerhalb dieses Projektes werden die Pakete, Schnittstellen und Klassen generiert. Zusätzlich wird im Konstruktor der Ausgabestrom für Meldungen übergeben. Man startet das Erzeugen mit dem Aufruf der Methode `realize`. Diese Methode ruft nacheinander die folgenden drei Methoden `realizePackages`, `realizeInterfaces` und `realizeClasses` auf, die in Eclipse die Pakete, Schnittstellen und Klassen über die Hilfklasse `Util` erzeugen. Die Methoden sind im Anhang A beschrieben.

Die Klasse `PrototypePlugin`

Für die Klasse `PrototypePlugin` ist kein UML-Klassendiagramm abgebildet, da die Klasse lediglich aus einem Konstruktor besteht, wie nachfolgend erläutert.

Ein Plugin für die Entwicklungsumgebung Eclipse besteht aus einer Klasse für das Plugin, optional weiteren Klassen für einzelne Aktionen, die das Plugin einem Benutzer zur Verfügung stellt und einer XML-Datei, welche das Plugin beschreibt. Die XML-Datei muss den Dateinamen `plugin.xml`

besitzen und wird von Eclipse verarbeitet. Eclipse wird damit ermöglicht, alle für das Plugin notwendigen Pakete, Schnittstellen, Klassen und Bibliotheken aufzufinden. In der XML-Datei wird auch definiert, welche Aktionen das Plugin zur Verfügung stellt. Die zu dem Plugin gehörende Klasse muss von der Klasse `AbstractUIPlugin` abgeleitet werden. Die in der Superklasse definierten Methoden können zur Anpassung des Plugins überschrieben werden, dies ist jedoch bei dem Prototypen nicht notwendig und die Eclipse-Basisklasse reicht von der Funktionalität her völlig aus. Dazu wird die Klasse `PrototypePlugin` erstellt, welche von der Klasse `AbstractUIPlugin` abgeleitet ist. In dieser Klasse wird nur der Konstruktor implementiert, in dem lediglich der Konstruktor der Basisklasse aufgerufen wird.

Die Datei `plugin.xml` muss einer von Eclipse vorgegebenen Struktur folgen. Die Datei für den Prototypen ist in diesem Abschnitt auf Seite 75 aufgeführt. Die folgende Erklärung bezieht sich auf diese Datei. Das Wurzelelement der XML-Datei muss das Element `plugin` sein. Dieses Element hat die Attribute `id` mit einem frei zu wählenden Bezeichner zur Identifikation des Plugins, `name` mit dem gewählten Namen des Plugins, `version` mit der Version, `provider-name` mit dem Namen des Herstellers, bzw. Entwicklers des Plugins und `class` mit der Klasse des Plugins, also der oben erläuterten Klasse `PrototypePlugin`.

Das Element `plugin` hat die Kindelemente: `runtime`, `requires` und `extension`. Letzteres Elemente ist mehrfach erlaubt, in dem Prototypen wird es nur einmal angegeben, da der Prototyp wie unten beschrieben nur eine Aktion definiert.

In dem Element `runtime` wird angegeben, aus welchen Bibliotheken, also Java-Archiven das Plugin besteht. Dazu erhält das Element `runtime` pro Bibliothek ein Kindelement `library` mit dem Attribut `name`, in dem der Name der Java-Archivdatei eingetragen wird.

In dem Kindelement `requires` werden andere genutzte Eclipse-Plugins angegeben. Dazu erhält das Element `requires` Kindelemente `import` mit dem Attribut `plugin`, welches den Namen der benötigten Eclipse-Plugins enthält. Der Prototyp nutzt die in Eclipse selbst vorhandenen Plugins `org.eclipse.core.resources`, `org.eclipse.ui` und `org.eclipse.jdt.core`, um die Pakete, Schnittstellen und Klassen anzulegen.

Unter dem Element `extension` werden die sogenannten Erweiterungen des Plugins angegeben. Dies entspricht den Aktionen, die das Plugin anbietet. Im Falle des Prototypen wird über das Attribut `point` mit der Angabe von `org.eclipse.ui.popupMenus` angegeben, dass das Plugin das Kontextmenü von Eclipse erweitert. Über das Kindelement `objectContribution` wird diese Erweiterung beim Prototypen darauf eingeschränkt, dass die Erweiterung des Kontextmenüs lediglich vorhanden ist, wenn die Selektion, die der Benutzer vor dem Öffnen des Kontextmenü getroffen hat, Java-Elementen entspricht. Dazu wird in dem Attribut `objectClass` die entsprechend von Eclipse definierte Schnittstelle für Java-Elemente angegeben. Über das Kindelement `menu` wird die Erweiterung des Kontextmenüs definiert. Dazu wird hier mit dem Attribut `label` der Text angegeben, der in dem Kontextmenü aufgenommen werden soll. Mit dem Attribut `path` wird durch die Angabe von `additions` bestimmt, dass der Menüeintrag unten im Kontextmenü aufgeführt wird. Das Attribut `id` weist dem Menüeintrag einen eindeutigen Bezeichner zu.

Das Element `action` definiert die durch den Prototypen neu hinzugekommene Aktion, die ein Benutzer über das Kontextmenü aufrufen kann. Über das Attribut `label` wird der Aktion ein Bezeichner zugeordnet, der in dem Untermenü des Kontextmenüs erscheint. In dem Attribut `class` wird der komplette Klassenname inklusive Paketpfad, der für die Aktion verantwortlichen Klasse angegeben.

Bei dem Prototypen ist dies die Klasse `StartAction`, die nachfolgend im Abschnitt 7.2.7 auf Seite 76 erläutert ist. Mit Hilfe des Attributes `menubarPath` wird angegeben, wo im Kontextmenü der Eintrag für die Aktion erscheinen soll, hier ist also die Identifikation des Kontextmenüeintrages des Prototypen gefolgt von dem Untermenü angegeben. Das Attribut `id` weist der Aktion einen eindeutigen Bezeichner zu.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="de.osz.eclipseprototype.PrototypePlugin"
   name="PrototypePlugin"
   version="1.0.0"
   provider-name="Oliver Szymanski"
   class="de.osz.eclipseprototype.PrototypePlugin">

   <runtime>
      <library name="Prototype.jar">
      </library>
      <library name="kurejava.jar">
      </library>
      <library name="jaxb-libs.jar">
      </library>
      <library name="jaxb-ri.jar">
      </library>
      <library name="jaxb-api.jar">
      </library>
   </runtime>

   <requires>
      <import plugin="org.eclipse.core.resources"/>
      <import plugin="org.eclipse.ui"/>
      <import plugin="org.eclipse.jdt.core"/>
   </requires>

   <extension
         point="org.eclipse.ui.popupMenus">
      <objectContribution
            objectClass="org.eclipse.jdt.core.IJavaElement"
            id="de.osz.eclipseprototype.IJavaElement">
         <menu
               label="Prototype"
               path="additions"
               id="de.osz.eclipseprototype.menuPrototype">
         </menu>
         <action
               label="Start Prototype"
               class="de.osz.eclipseprototype.StartAction"
               menubarPath=
```

```
                "de.osz.eclipseprototype.menuPrototype/start"
              id="de.osz.eclipseprototype.StartAction">
        </action>
      </objectContribution>
   </extension>

</plugin>
```

Weitere Angaben, wie man Eclipse-Plugins entwickelt, stehen unter [10] zur Verfügung.

Die Klasse `StartAction`

Die Klasse `StartAction` ist in dem UML-Klassendiagramm in Abbildung 7.2 auf Seite 62 abgebildet. Die Klasse `StartAction` des Prototypen wird von Eclipse aufgerufen, wenn das Plugin über das zu dem Plugin gehörende Kontextmenu gestartet wird. Eine Klasse, welche in Eclipse für Aktionen innerhalb eines Plugins zuständig ist, muss die Schnittstelle

`org.eclipse.ui.IObjectActionDelegate`,

die von Eclipse definiert wird, implementieren. Die Schnittstelle deklariert drei Methoden, dabei spielt die Methode `setActivePart` für die Entwicklung des Prototypen keine Rolle und wurde mit einer leeren Implementierung versehen. Nachfolgend werden die beiden anderen Methoden erläutert:

- `public void selectionChanged (IAction action, ISelection selection)`

  Diese Methode wird von Eclipse aufgerufen, wenn der Eclipse-Benutzer einen Eintrag in dem Projektbaum auswählt. Die Methode sichert die Auswahl des Benutzers in dem privaten Attribut `private ISelection selection`. Der Typ `ISelection` ist eine von Eclipse definierte Schnittstelle für die Auswahl von Einträgen des Projektbaumes.

- `public void run(IAction action)`

  Diese Methode wird von Eclipse aufgerufen, wenn der Eclipse-Benutzer das Plugin über das zu dem Plugin gehörende Kontextmenü startet. In der Methode wird die vorher in der Methode `selectionChanged` gesicherte und vom Benutzer getroffene Auswahl geprüft. Über das Attribut `selection` werden die einzelnen Auswahlobjekte und das zugehörige Eclipse-Java-Projekt ermittelt. Konnte zu der vom Benutzer getroffenen Auswahl kein existierendes Java-Projekt ermittelt werden, wird das Plugin gestoppt. Ansonsten wird dem Benutzer ein Dialog angezeigt, mit dem er eine Szenario-Datei auswählen kann. Wählt der Benutzer eine Datei aus, wird der Prototyp über den Aufruf der Methode `run` der Klasse `Prototype`, die in Anhang A erläutert ist, gestartet. Ist der Durchlauf des Prototypen erfolgreich beendet, wird dem Benutzer die Meldung `scene generated`, ansonsten eine Meldung mit der aufgetretenen Ausnahme angezeigt.

7.3 Starten des Prototypen

Um den Prototypen in Eclipse zu integieren, müssen alle erforderlichen Klassen kompiliert und in einem Java-Archiv zusammengefasst werden. Dieses Archiv und die Plugin-Beschreibungsdatei `plugin.xml` müssen in das Verzeichnis `plugins` im Eclipse-Verzeichnis in ein neues Unterverzeichnis gelegt werden. Das neu zu erstellende Unterverzeichnis wird im Allgemeinen nach dem jeweiligen Plugin benannt. Hier heißt das Verzeichnis demnach

`de.osz.eclipseprototype.PrototypePlugin`.

Ebenfalls müssen die benötigten Bibliotheken in dieses Unterverzeichnis kopiert werden. Dazu gehört das Java-Archiv, welches die Bibliothek, die den Zugriff auf den funktionalen Kern von RELVIEW kapselt und die Java-Archive mit den *JAXB*-Bibliotheken `jaxb-libs.jar`, `jaxb-ri.jar` und `jaxb-api.jar`. Diese Dateien sind unter [18] verfügbar. Alle diese Dateien müssen in der Datei `plugin.xml`, wie im Abschnitt 7.2.7 auf Seite 73 ist, unter dem Element `runtime` angegeben werden. Wo man die native Bibliothek hinterlegt ist frei wählbar, da man ihren Pfad in den jeweiligen XML-Szene-Dateien wie im Abschnitt 7.2.1 auf Seite 62 angibt.

Nachdem man die Entwicklungsumgebung Eclipse neu startet, ist das Plugin aktiviert. Um es auszuführen, muss man im Eclipse-Projektbaum ein Java-Projekt oder ein Eintrag innerhalb eines Java-Projektes selektieren und das Kontextmenü mit einem Klick der rechten Maustaste öffnen. Innerhalb des Kontextmenüs kann man das Untermenü `Prototype` öffnen und den Prototypen über den Eintrag `Start Prototype` starten. Dies ist in dem UML-Diagramm in Abbildung 7.5 auf Seite 78 dargestellt. Durch den Aufruf im Kontextmenü wird über Eclipse die `run`-Methode der Klasse `StartAction` aufgerufen. In dieser Methode wird dem Benutzer ein Dateiauswahlmenü angezeigt. Der Benutzer muss eine XML-Datei, welche eine Szene definiert, auswählen und der Prototyp beginnt mit dem Arbeitsablauf. Dabei wird die `run`-Methode der Klasse `Prototype` aufgerufen. In dieser Methode wird die `init`-Methode aufgerufen und die XML-Datei eingelesen. Danach wird die `define`-Methode aufgerufen und darin über die Klasse `Definer` die Pakete, Schnittstellen und Klassen, welche in der XML-Datei definiert sind, ermittelt. Danach wird die Methode `visualize` aufgerufen, welche über die Klasse `Visualizer` die dreidimensionale Darstellung, die in der XML-Datei angegeben ist, verarbeitet. Nachfolgend wird in der Methode `checkRelations` mit Hilfe der Klassen `RelationDefiner` und `RelationEvaluator` die Relation aus der Darstellung ermittelt und diese auf Eigenschaften geprüft. Verliefen die Prüfungen zufriedenstellend, wird ein Objekt der Klasse `EclipsePrototypeRealizer` erstellt, und mit dem Aufruf der Methode `realize` dieses Objektes die Pakete, Schnittstellen und Klassen in Eclipse erstellt.

7.4 Zusammenfassung

Der Prototyp kann Szenen aus einer vom Benutzer angegebenen XML-Datei einlesen. Diese Szenen, die per XML definiert werden, entsprechen Visualisierungen wie sie im dreidimensionalen Software-Entwurf verwendet werden. Der Prototyp kann daraus Relationen erstellen und diese Relationen auf Eigenschaften prüfen. Die Prüfung erfolgt durch die Auswertung von relationalen Termen mit Hilfe der Java-Bibliothek, die den Zugriff auf den funktionalen Kern von RELVIEW ermöglicht. Ergibt die Auswertung, dass aus der Szene korrekt Pakete, Schnittstellen und Klassen in Java generiert werden können, erstellt der Prototyp die Pakete, Schnittstellen und Klassen, wie sie in der XML-Datei

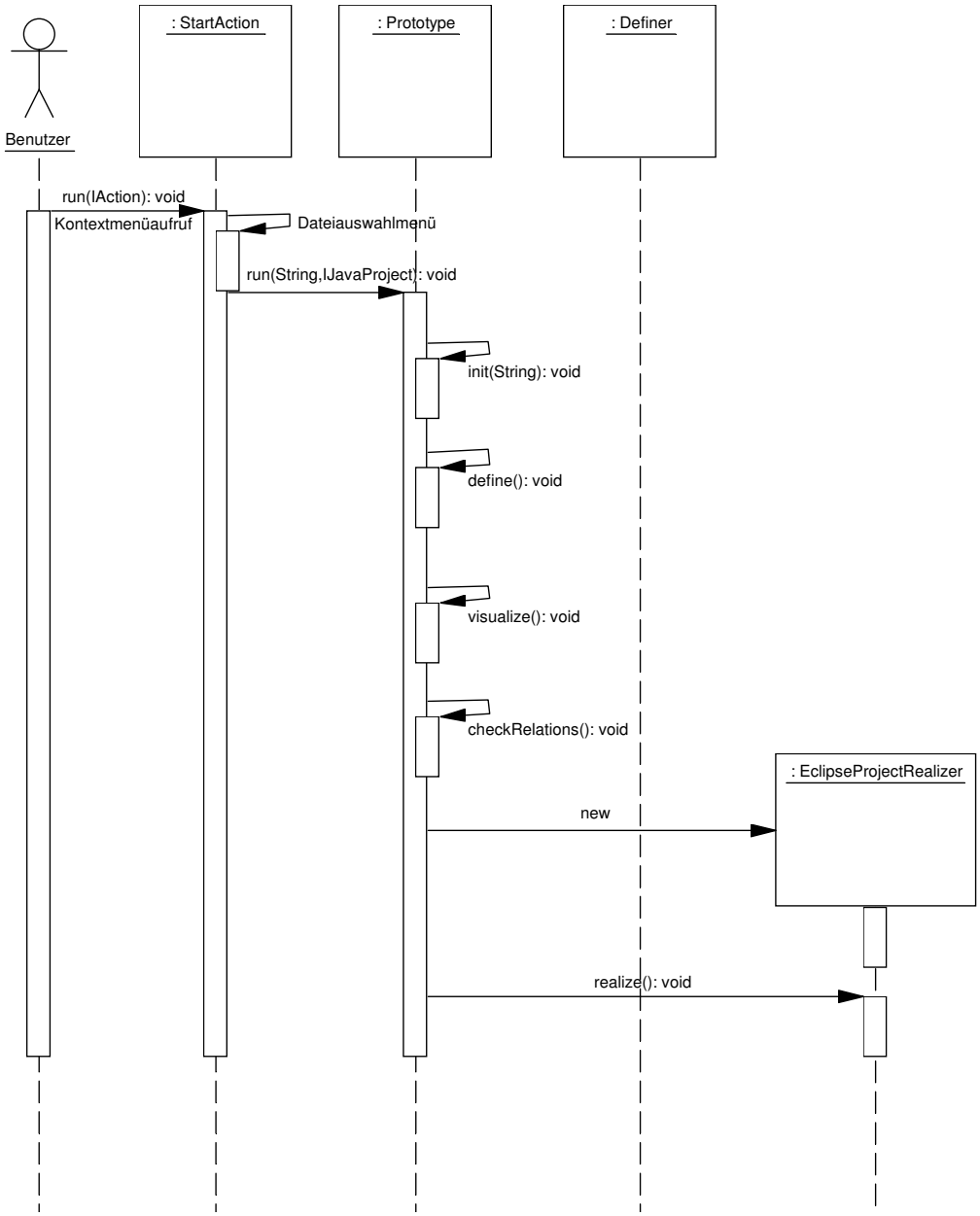

Abbildung 7.5: Arbeitsablauf des Prototypen

angegeben sind. Der Prototyp nimmt während des Arbeitsablaufes intensiven Gebrauch von der Java-Bibliothek, die sich bei der Entwicklung und Anwendung des Prototypen als nützlich erwiesen hat.

Der Prototyp selbst ist modular aufgebaut und kann somit leicht überarbeitet bzw. in Teilen in anderen Projekten des Lehrstuhles genutzt werden. Einzelne Teile des Prototypen wie die Klassen `RelationDefiner`, `RelationEvaluator` und `EclipsePrototypeRealizer` können für die Entwicklung einer Applikation zur Darstellung und Manipulationen von Paket-, Schnittstellen- und Klassenbeziehungen genutzt werden.

Der Prototyp ist effizienter zu gestalten, wenn Teile von ihm weiterhin genutzt werden sollen. Bestimmte häufig in Termen temporär verwendete Ausdrücke, müssen einmalig berechnet werden, so dass man in Weiteren Termen die berechnete Teillösung nutzen kann. Im Prototypen ist dies absichtlich nicht der Fall, da hier der Schwerpunkt nicht auf Geschwindigkeit sondern auf Verständlichkeit der Terme liegt. Ebenso wird momentan bei jedem erneuten Durchlauf des Prototypen die native Bibliothek neu initialisiert und am Ende der Speicher bereinigt. Dies müsste beim Starten und Beenden von Eclipse geschehen. Hier wurde dieser Vorgang absichtlich so gewählt, um die Arbeitsweise der Bibliothek zu verdeutlichen.

Kapitel 8

Ergebnisse und Ausblick

Am Lehrstuhl Software-Technologie der Universität Dortmund wird ein Programm für den dreidimensionalen Software-Entwurf entwickelt. Das Programm wird als Plugin für die Entwicklungsumgebung Eclipse realisiert und soll die Darstellung von Java-Paket-, Schnittstellen- und Klassenstrukturen in dreidimensionalen Diagrammen ermöglichen, wie sie in [1],[3] und [2] beschrieben werden. Darüber hinaus soll den Benutzern des Programmes ermöglicht werden, die Diagramme manipulieren zu können und daraus wieder gültige Java-Quellcodegerüste für die Paketstruktur, die Schnittstellen und Klassen zu erzeugen. Damit nach einer Manipulation der dreidimensionalen Darstellung gültiger Java-Quellcode erzeugt werden kann, muss geprüft werden, ob die Darstellung nach der Manipulation gültigen Java-Strukturen entspricht. Die dazu notwendigen Eigenschaften, welche die Darstellung erfüllen muss, müssen folglich geprüft werden, um zu entscheiden, ob die Manipulation der Darstellung zulässig ist. Diese Eigenschaften können in Ausdrücken der relationalen Algebra formalisiert werden. Zur Prüfung der Gültigkeit von Manipulation können diese Ausdrücke ausgewertet werden. Damit die Auswertung in einem interaktiven Programm bei jeder vom Benutzer ausgeführten Manipulation ausgeführt werden kann, ist es notwendig, dass diese Ausdrücke effizient ausgewertet werden können, damit ein Benutzer nicht lange auf eine Reaktion des Programmes warten muss. Das Programm RELVIEW kann solche Ausdrücke effizient auswerten.

Die vorliegende Diplomarbeit ermöglicht den Zugriff von Java-Programmen auf den funktionalen Kern des C-Programmes RELVIEW. Bei RELVIEW handelt es sich um ein interaktives, GUI-basiertes C-Programm mit dem Relationen eingegeben und relationale Ausdrücke über den Relationen ausgewertet werden können. Der funktionale Kern des Programmes RELVIEW ist die Menge der Funktionen von RELVIEW, die benötigt werden, um Relationen nicht GUI-basiert zu definieren und Terme auszuwerten. Die Funktionen sind dabei plattformunabhängig implementiert. Um den Zugriff auf den funktionalen Kern zu ermöglichen, wurde der funktionale Kern aus RELVIEW in Form einer Bibliothek extrahiert. Von Java aus wird auf den funktionalen Kern über das *Java Native Interface* zugegriffen. Dazu wurde eine Adapterschicht in C entwickelt, welche notwendige Typumwandlungen zwischen Java und C vornimmt. Desweiteren wurde eine Adapterschicht in C implementiert, um ein spezielles Namensschema für Funktionsnamen zu erfüllen, welches das *Java Native Interface* für native Funktionen, die über Java aufgerufen werden sollen, spezifiziert. Eine weitere Adapterschicht in Java ermöglicht den Zugriff auf den funktionalen Kern mit der Unterstützung von Java-Objekten für Relationen und Relationsmanager. In dieser Adapterschicht werden Fehler, welche die nativen Funktionen durch spezielle Rückgabewerte signalisieren, abgefangen und durch neu erzeugte Java-

Ausnahmen behandelt. Damit ist eine in Java übliche Fehlerbehandlung durch das Fangen von Ausnahmen möglich.

Da die Aufgabe der Bibliothek ist, von Java aus auf Funktionen in C zuzugreifen aber zwischen Java und C unterschiedliche Datentypen vorliegen, ist die von Java erwartete Schnittstelle anders als die in Form der nativen Bibliothek vorliegende Schnittstelle. Da das Adapterentwurfsmuster das Problem von inkompatiblen Schnittstellen löst, bietet es sich an dieses zu nutzen. Für die Typkonvertierungen zwischen Java und C und um auf Java-Seite eine Schnittstelle zu ermöglichen, die im Fehlerfall eine Java-Ausnahme liefert, und nicht wie im Fall der nativen Bibliothek die Fehlercodes in Form von Rückgaben, muss man auf jeden Fall einen Adapter implementieren, welcher die durch die native Bibliothek vorliegende Schnittstelle nutzt. Der Adapter muss auch für die Java-Seite eine Schnittstelle anbieten, welche Objekte als Parameter- und Rückgabetypen annimmt, sowie die Java-Ausnahmebehandlung ermöglicht. Dieser Adapter steht zwischen der aufrufenden Methode in Java und der aufzurufenden Funktion der nativen Bibliothek. D.h. es gibt keine echte Alternative zu den gewählten Adaptern.

Um die Java-Bibliothek in einer Java-Client-Anwendung wie in Java üblich objektorientiert nutzen zu können, sieht die Bibliothek Relations- und Relationsmanagerobjekte vor. Relationsobjekte kapseln dabei eine Relation und bieten Methoden um die Relation zu manipulieren und abzufragen. Bei den Relationsmanagerobjekten handelt es sich um Fabriken für Relationen nach dem Fabrikentwurfsmuster. Um Relationen anlegen und mit ihnen arbeiten zu können, benötigt man ein Relationsmanagerobjekt, welches man über eine Fabrikklasse für diese erzeugen kann. Statt Fabriken zur Verwaltung von Relationen und Relationsmanagern zu implementieren, hätte man die entsprechenden Objekte auch einfach direkt anlegen können (durch den Java-`new`-Operator). Das Anlegen dieser Objekte hat aber zur Folge, dass der in C implementierte funktionale Kern von RELVIEW Speicherbereich allokiert. Da dieser Speicherbereich wieder freigegeben werden muss, wenn die Objekte nicht mehr benötigt werden (spätestens wenn der Java-`GarbageCollector` die Objekte bereinigt), muss noch auf der Java-Seite vermerkt werden, welche Objekte angelegt wurden, und die Möglichkeit bestehen diese Objekte zu verwalten. Dies bedeutet, dass es auf jeden Fall eine Klasse geben muss, welche für die Verwaltung der Objekte verantwortlich ist. Da diese Klasse informiert werden muss, wenn die Objekte erzeugt werden, ist es am effizientesten, wenn sie direkt als Fabrik für die Objekte implementiert ist. Somit gibt es auch zu dem gewählten Fabrikentwurfsmuster keine echte Alternative.

Insgesamt ermöglicht die in der Diplomarbeit entwickelte Java-Bibliothek, welche den Zugriff auf den funktionalen Kern von RELVIEW objektorientiert kapselt und die Java-Ausnahmenbehandlung ermöglicht, in beliebigen Java-Client-Programmen mit Relationen zu arbeiten und relationale Terme auf die Relationen anzuwenden und diese effizient auszuwerten. RELVIEW ermöglicht auch die Definition von relationalen Funktionen und Programmen, dies kann ebenfalls über die Java-Bibliothek von Java-Programmen aus genutzt werden. Über den reinen Zugriff auf den funktionalen Kern von RELVIEW hinaus wurde Funktionalität in Relationsmanager- und Relationsobjekten gekapselt, so dass man direkt Java-Objekte in Relation zueinander setzen und abfragen kann, welche Objekte zu einem Objekt in Relation stehen. Die Java-Bibliothek kann in jeder Java-Client-Anwendung genutzt werden, in dem man auf die Relationale Algebra zurückgreifen möchte, nicht nur im Kontext des dreidimensionalen Software-Entwurfs.

Als Teil der Diplomarbeit wurde ein Prototyp für den dreidimensionalen Software-Entwurf entwickelt. Mit seiner Hilfe wurde die Java-Bibliothek getestet. Ferner wird verdeutlicht, wie man diese benutzt. Der Prototyp selbst wurde bereits als Eclipse-Plugin realisiert und kann aus den, über eine

XML-Struktur angegebenen, dreidimensionalen Darstellungen von Paket-, Schnittstellen- und Klassenbeziehungen Relationen erstellen. Der Prototyp prüft diese Relationen auf notwendige Eigenschaften, die gültig sein müssen, damit die angegebene Darstellung korrekten Beziehungen in einem Java-Programm entspricht. Die Prüfung der Eigenschaften erfolgt mit Hilfe von relationalen Termen über die Java-Bibliothek. Der Prototyp ist in der Lage, aus der gegebenen Darstellung in Eclipse Pakete, Schnittstellen und Klassen zu erstellen, falls alle Prüfungen zufriedenstellend verlaufen sind.

Mit der implementierten Funktionalität kann der Prototyp als Basis für das am Lehrstuhl Software-Technologie zu entwickelnde Eclipse-Plugin im Kontext des dreidimensionalen Software-Entwurfes verwendet werden. Das Plugin soll es in Zukunft ermöglichen, Paket-, Schnittstellen- und Klassenstrukturen von Java-Programmen dreidimensional darzustellen und Manipulation mit anschließender Generierung von Quellcodegerüsten zu ermöglichen. Der Prototyp wurde modular durch lose gekoppelte Komponenten, die über gemeinsam genutzte Datenstrukturen Daten austauschen, realisiert, damit einzelne Komponenten leicht ausgetauscht oder das Plugin durch neue Komponenten um Funktionalität ergänzt werden kann. Hat man z.B. im späteren Eclipse-Plugin eine Datenstruktur, welche die graphischen Cubes, Walls und Cone Trees beschreibt, ist es möglich durch Austausch einer Komponente dem Prototypen nicht mehr eine XML-Datei als Eingabe zur Verfügung zu stellen, sondern ihm direkt diese Datenstruktur zu liefern. Auch könnte man eine zusätzliche Komponente einbinden, welche nicht nur Vererbungs- und Implementierungsbeziehungen bei Klassen und Schnittstellen verarbeitet, sondern Methodensignaturen betrachtet und somit weitere Assoziationen, welche durch die angegebenen Parameter- und Rückgabetypen gegeben sind, beachtet. Eine Alternative statt der losen gekoppelten Komponentenstruktur bei der Entwicklung des Prototypen direkt aufeinander abgestimmte und zugreifende Klassen zu entwickeln, hätte die Entwicklungszeit verkürzt und ließe eine schnellere Einarbeitung in den kürzeren Quellcode zu. Auf diese Vorteile wurde absichtlich verzichtet, da man sonst auf die oben genannten Vorteile der losen Komponentenstruktur verzichten müsste.

Das Programm RELVIEW bietet neben den bereits von der entwickelten Java-Bibliothek unterstützten Funktionen die Möglichkeit, zu Relationen unterschiedliche Sichten als Graphen zu erzeugen. Dabei werden verschiedene Grapharten unterstützt. Als Ergänzungen zur vorliegenden Arbeit kann man den Zugriff auch auf diese Funktionen implementieren und so Java-Client-Programmen, welche die Bibliothek benutzen, die Möglichkeit geben, definierte Relationen unterschiedlich als Graphen darzustellen. Desweiteren würde es sich anbieten, in Java ein interaktives graphisches Programm zu entwickeln, welches wie das Programm RELVIEW ermöglicht, Relationen zu definieren, zu verwalten, in unterschiedlichen Darstellungen zu betrachten und relationale Terme innerhalb der Oberfläche anzugeben und diese auszuwerten. Ein solches Programm könnte eine Alternative zu dem Programm RELVIEW sein, und den Nachteil, dass RELVIEW nur auf wenigen Plattformen zur Verfügung steht, lösen. Ebenfalls ist es aufgrund den hohen Anzahl von Bibliotheken, welche in Java zur Verfügung stehen, einfacher, in Java ein neues Programm mit graphischer Oberfläche zu entwickeln, welches die Java-Bibliothek für die Relationale Algebra nutzt, als eine neue Version von RELVIEW zu implementieren.

Anhang A

Methodenbeschreibungen

A.1 Deklaration und Implementierung der Adapterfunktionen

Folgende aufgelistete Funktionen besitzen alle das Präfix (vgl. 3.2.1 auf Seite 26).

`Java_de_osz_kurejava_Kurejava_`

Dieser Präfix ist zur Erfüllung des *JNI*-Namensschemas unbedingt notwendig, wird aber zur besseren Lesbarkeit im Folgenden nicht aufgeführt.

Den nachfolgenden Funktionen ist gemeinsam, dass sie Eingabeparameter vom Typ `jbyteArray` in C-Zeiger und `jstring` in C-Zeichenketten über die in Abschnitt 3.3 auf Seite 29 beschriebenen Funktionen konvertieren. Danach rufen die Funktionen die eigentlichen namensgleichen Funktionen des funktionalen Kerns auf und liefern die ermittelten Rückgabewerte zurück. Wenn von Funktionen des funktionalen Kerns dabei C-Zeiger zurückgegeben werden, werden diese in ein `jbyteArray` konvertiert und zurückgeliefert, da in Java keine Zeigerdatentypen existieren.

- ```
  JNIEXPORT jbyteArray JNICALL initRelManager
    (JNIEnv* env, jclass cl)
  ```
  Diese Funktion kapselt die Funktion `Kure_Init` des funktionalen Kerns.
- ```
  JNIEXPORT void JNICALL quitRelManager
    (JNIEnv* env, jclass cl, jbyteArray relManager)
  ```
  Diese Funktion kapselt die Funktion `Kure_Quit` des funktionalen Kerns.
- ```
  jbyteArray JNICALL relationNew
    (JNIEnv* env, jclass cl, jbyteArray relManager,
     jstring relationName, jint height, jint width)
  ```
  Diese Funktion kapselt die Funktion `Kure_RelationNew` des funktionalen Kerns.
- ```
  JNIEXPORT jint JNICALL relationExists
    (JNIEnv* env, jclass cl,
     jbyteArray relManager, jstring relationName)
  ```
  Diese Funktion kapselt die Funktion `Kure_RelationExists` des funktionalen Kerns.

- `JNIEXPORT jbyteArray JNICALL relationGet`
  `(JNIEnv* env, jclass cl,`
  `jbyteArray relManager, jstring relationName)`

  Diese Funktion kapselt die Funktion `Kure_RelationGet` des funktionalen Kerns.

- `JNIEXPORT jint JNICALL relationDelete`
  `(JNIEnv* env, jclass cl,`
  `jbyteArray relManager, jbyteArray relation)`

  Diese Funktion kapselt die Funktion `Kure_RelationDelete` des funktionalen Kerns.

- `JNIEXPORT jint JNICALL relationGetHeight`
  `(JNIEnv* env, jclass cl,`
  `jbyteArray relManager, jbyteArray relation)`

  Diese Funktion kapselt die Funktion `Kure_RelationGetHeight` des funktionalen Kerns.

- `JNIEXPORT jint JNICALL relationGetWidth`
  `(JNIEnv* env, jclass cl,`
  `jbyteArray relManager, jbyteArray relation)`

  Diese Funktion kapselt die Funktion `Kure_RelationGetWidth` des funktionalen Kerns.

- `JNIEXPORT jint JNICALL relationGetNumberOfEntries`
  `(JNIEnv* env, jclass cl,`
  `jbyteArray relManager, jbyteArray relation)`

  Diese Funktion kapselt die Funktion `Kure_RelationGetNumberOfEntries` des funktionalen Kerns.

- `JNIEXPORT jint JNICALL relationGetBit`
  `(JNIEnv* env, jclass cl,`
  `jbyteArray relManager, jbyteArray relation,`
  `jint row, jint column)`

  Diese Funktion kapselt die Funktion `Kure_RelationGetBit` des funktionalen Kerns.

- `JNIEXPORT jint JNICALL relationSetBit`
  `(JNIEnv* env, jclass cl,`
  `jbyteArray relManager, jbyteArray relation,`
  `jint row, jint column)`

  Diese Funktion kapselt die Funktion `Kure_RelationSetBit` des funktionalen Kerns.

- `JNIEXPORT jint JNICALL relationClearBit`
  `(JNIEnv* env, jclass cl,`
  `jbyteArray relManager, jbyteArray relation,`
  `jint row, jint column)`

  Diese Funktion kapselt die Funktion `Kure_RelationClearBit` des funktionalen Kerns.

- `JNIEXPORT jint JNICALL functionNew`
  `(JNIEnv* env, jclass cl,`
  `jbyteArray relManager, jstring function)`

  Diese Funktion kapselt die Funktion `Kure_FunctionNew` des funktionalen Kerns.

- `JNIEXPORT jint JNICALL programNew`
  `(JNIEnv* env, jclass cl,`
  `jbyteArray relManager, jstring program)`

  Diese Funktion kapselt die Funktion `Kure_ProgramNew` des funktionalen Kerns.

- `JNIEXPORT jint JNICALL programFileRead`
  `(JNIEnv* env, jclass cl,`
  `jbyteArray relManager, jstring filename)`

  Diese Funktion kapselt die Funktion `Kure_ProgramFileRead` des funktionalen Kerns.

- `JNIEXPORT jint JNICALL evaluate`
  `(JNIEnv* env, jclass cl,`
  `jbyteArray relManager, jstring term,`
  `jstring resultRelationName)`

  Diese Funktion kapselt die Funktion `Kure_EvaluateTerm` des funktionalen Kerns. Die Ergebnisrelation läßt sich danach mit der Adapterfunktion `relationGet` ermitteln.

A.2 Die Klasse `KurejavaOO`

Im Folgenden werden die Methoden der Klasse `KurejavaOO` erläutert.

- `protected static void init()`
  `throws AlreadyInitializedException`

  Diese Methode initialisiert die native Bibliothek und merkt sich in einer Klassenvariablen, dass die Bibliothek initialisiert wurde. Der Debugmodus wird per Standard deaktiviert und die Liste der angelegten Relationsmanager geleert.

  Falls die Bibliothek über diese Methode bereits vorher initialisiert und nicht mit der Methode `quit()` beendet wurde, wird die Ausnahme `AlreadyInitializedException` ausgelöst.

Alle nachfolgend aufgelisteten Methoden lösen, wenn sie aufgerufen werden, ohne dass die Bibliothek vorher mit der `init()`-Methode initialisiert wurde, die Ausnahme `NotInitializedException` aus.

- `protected static void quit()`
  `throws NotInitializedException`

  Diese Methode muss aufgerufen werden, um den Speicher zu bereinigen, wenn die Arbeit mit der Bibliothek abgeschlossen ist. Da es sich um eine native, in C entwickelte Bibliothek handelt, deren Speicherbereich die *Java Virtual Machine* nicht verwalten kann, ist dies unbedingt erforderlich. Soll danach noch auf Funktionen der Bibliothek zugegriffen werden, muss die Bibliothek zuerst erneut mit der Methode `init()` initialisiert werden.

- `protected static void setDebug(boolean value)`
  `throws NotInitializedException`

  Mit dieser Methode kann man den Debugmodus aktivieren, bzw. deaktivieren. Übergibt man der Methode als Parameter den Wert `true` ist der Debugmodus aktiviert. Im aktivierten Debugmodus werden auf der Java-Standardkonsolenausgabe Meldungen über die Aufrufe der Funktionen der nativen Bibliothek ausgegeben. Der Debugmodus ist nur für die Entwicklungsphase der Bibliothek und zum Nachvollziehen von aufgetretenen Fehlern gedacht.

- `protected static RelManagerImpl initRelManager(String name)`
  `throws NotInitializedException, RelManagerException`

  Diese Methode wird aufgerufen, um einen Relationsmanager anzulegen. Dabei muss dem Relationsmanager ein eindeutiger Name zu dessen Identifikation gegeben werden. In der Methode wird daher zuerst überprüft, ob bereits ein Relationsmanager mit diesem Namen angelegt wurde. Falls dies der Fall ist, löst die Methode eine Ausnahme `RelManagerException` aus. Ansonsten legt die Methode über die native Bibliothek einen neuen Relationsmanager an, erstellt eine Instanz der Klasse `RelManagerImpl`, welche den C-Zeiger auf den eigentlichen Relationsmanager innerhalb der nativen Bibliothek als `Byte-Array` als Objektattribut beinhaltet, vermerkt den neu angelegten Relationsmanager unter dem angegebenen Namen in dem Klassenattribut `private static Map relManagers` und liefert die erzeugte Instanz zurück.

- `protected static void quitRelManager(RelManagerImpl relManager)`
  `throws NotInitializedException`

  Diese Methode wird mit einem Relationsmanager-Objekt als Parameter aufgerufen, um den in der nativen Bibliothek beim Erzeugen verwendeten Speicherbereich freizugeben. Auch wird der Vermerk über den Relationsmanager in dem Klassenattribut `private static Map relManagers` aufgehoben. Die Methode wird verwendet, wenn man einen einzelnen Relationsmanager nicht mehr benötigt. Benötigt man keinen Relationsmanager mehr, besteht die Möglichkeit, direkt die Methode `quit()` aufzurufen.

- `protected static RelationImpl relationNew(`
  `RelManagerImpl relManager, String relationName,`
  `int height, int width)`
  `throws NotInitializedException, RelationException`

  Mit dieser Methode kann eine neue Relation angelegt werden. Diese Relation wird in dem übergebenen Relationsmanager verwaltet. Die Relation benötigt einen eindeutigen Namen, der innerhalb des Relationsmanagers zu ihrer Identifizierung benötigt, und zusammen mit der Breite und Höhe der der Relation zugrunde liegenden Relationsmatrix als Parameter der Methode übergeben wird. Kann die neue Relation nicht von der nativen Bibliothek angelegt werden, so wirft die Methode eine Ausnahme der Klasse `RelationException`. Ansonsten wird ein neues Objekt der Klasse `RelationImpl` erzeugt und zurückgeliefert, welche den C-Zeiger auf die eigentliche Relation innerhalb der nativen Bibliotek als `Byte-Array` als Objektattribut beinhaltet.

- `protected static boolean relationExists(`
  `RelManagerImpl relManager, String relationName)`
  `throws NotInitializedException`

Mit dieser Methode kann über Zugriff auf die native Bibliothek geprüft werden, ob eine Relation unter dem Relationsnamen in einem Relationsmanager angelegt wurde. Dazu wird innerhalb der Methode der C-Zeiger als `Byte-Array` aus dem Parameterobjekt der Klasse `RelManagerImpl` gelesen und der nativen Bibliothek übergeben. Die Methode liefert `true` zurück, wenn die Relation in dem Relationsmanager existiert, sonst `false`.

- ```
  protected static RelationImpl relationGet(
    RelManagerImpl relManager, String relationName)
      throws NotInitializedException, RelationException
  ```

  Diese Methode liefert zu einem Relationsmanager und einem Relationsnamen, welche beide als Parameter übergeben werden, die in dem Klassenattribut `private static Map relManagers` eingetragene Relation, falls eine Relation mit dem Namen unter dem Relationsmanager angelegt worden ist. Ansonsten löst die Methode die `RelationException`-Ausnahme aus.

- ```
  protected static void relationDelete(
    RelManagerImpl relManager, RelationImpl relation)
      throws NotInitializedException, RelationException
  ```

  Diese Methode löscht eine Relation aus einem Relationsmanager. Dazu werden die C-Zeiger als `Byte-Array` aus den Objekten gelesen und der nativen Bibliothek übergeben. Konnte die Relation in der nativen Bibliothek nicht gelöscht werden, wird die `RelationException`-Ausnahme ausgelöst.

- ```
  protected static int relationGetWidth(
    RelManagerImpl relManager, RelationImpl relation)
      throws NotInitializedException
  ```

  Diese Methode liest die C-Zeiger als `Byte-Array` aus dem Relationsmanager- und Relationsobjekt, ermittelt damit die Breite der Relationsmatrix über die native Bibliothek und liefert diese Breite zurück.

- ```
  protected static int relationGetHeight(
    RelManagerImpl relManager, RelationImpl relation)
      throws NotInitializedException
  ```

  Diese Methode liefert die Höhe der Relationsmatrix.

- ```
  protected static int relationGetNumberOfEntries(
    RelManagerImpl relManager, RelationImpl relation)
      throws NotInitializedException
  ```

  Diese Methode liefert die Anzahl der Einträge in der Relationsmatrix.

- ```
  protected static void relationSetBit(
    RelManagerImpl relManager, RelationImpl relation,
    int row, int column)
      throws NotInitializedException, RelationException
  ```

  Diese Methode setzt mit Hilfe der nativen Bibliothek einen Eintrag in der Relationsmatrix. Dazu ermittelt sie aus Relationsmanager und Relation die C-Zeiger auf die Speicherbereiche der nativen Relationsmanager und Relation als `Byte-Array`. Konnte der Eintrag in der nativen Bibliothek nicht gesetzt werden, so wirft die Methode die Ausnahme `RelationException`.

- `protected static boolean relationGetBit(`
  `RelManagerImpl relManager, RelationImpl relation,`
  `int row, int column)`
  `throws NotInitializedException, RelationException`

  Diese Methode liest über die native Bibliothek einen Eintrag aus der Relationsmatrix. Trat beim Abfragen des Eintrages in der nativen Bibliothek ein Fehler auf, wird die Ausnahme `RelationException` geworfen. Es wird `true` zurückgeliefert, wenn der Eintrag gesetzt ist, und `false`, sonst.

- `protected static void relationClearBit(`
  `RelManagerImpl relManager, RelationImpl relation,`
  `int row, int column)`
  `throws NotInitializedException, RelationException`

  Diese Methode löscht über die native Bibliothek einen Eintrag aus der Relationsmatrix. Dazu ermittelt sie aus Relationsmanager und Relation die C-Zeiger auf die Speicherbereiche der nativen Relationsmanager und Relation als `Byte-Array` und ruft damit die Löschfunktion der nativen Bibliothek auf. Wenn der Eintrag nicht gelöscht werden konnte, wird die Ausnahme `RelationException` ausgelöst.

- `protected static void functionNew(`
  `RelManagerImpl relManager, String function)`
  `throws NotInitializedException, FunctionException`

  Diese Methode trägt in einem Relationsmanager eine Funktion ein. Die übergebene Zeichenkette `function` besteht dabei aus dem Funktionsnamen, gefolgt von einem Gleichheitszeichen und der Funktionsdefinition. Kann die Funktion über die native Bibliothek nicht angelegt werden, wird eine Ausnahme der Klasse `FunctionException` geworfen.

- `protected static void programNew(`
  `RelManagerImpl relManager, String program)`
  `throws NotInitializedException, ProgramException`

  Diese Methode definiert über die native Bibliothek neue relationale Programme, die in dem übergebenen Zeichenkettenparameter angegeben werden. Das Programm kann danach in Termen verwendet werden. Meldet die native Bibliothek Fehler bei der Programmdefinition, wird von der Methode die Ausnahme `ProgramException` ausgelöst.

- `protected static void programFileRead(`
  `RelManagerImpl relManager, String filename)`
  `throws NotInitializedException, ProgramException`

  Diese Methode verhält sich analog zur Methode `programNew`. Anstelle der Programmdefinition wird dieser Methode ein Dateiname inklusive Pfadangaben übergeben, um Programme aus dieser Datei einzulesen. Dabei handelt es sich um `ASCII`-codierte Dateien, welche relationale Programme beinhalten. Mit Hilfe der nativen Bibliothek wird das Programm eingelesen.

- `protected static RelationImpl evaluateTerm(`
  `RelManagerImpl relManager, String term,`
  `RelationImpl resultRelation)`
  `throws NotInitializedException, TermException`

Diese Methode nutzt die native Bibliothek, um einen relationalen Term auszuwerten. Ein Term kann nur im Kontext eines Relationsmanagers ausgeführt werden. Dies hat zur Folge, dass in einem Term nur Relationen verwendet werden können, die innerhalb eines Relationsmanagers verwaltet werden. Relationsmanager und Term werden der Methode als Parameter übergeben. Auch muss der Methode eine weitere Relation übergeben werden, welche auch in dem Relationsmanager verwaltet wird. In diese Relation wird das Ergebnis der Termauswertung von der nativen Bibliothek abgelegt. Treten bei der Termauswertung in der nativen Bibliothek Fehler auf, wird eine Ausnahme der Klasse `TermException` geworfen.

- `protected static boolean isInitialized()`

  Mit Hilfe dieser Methode kann man ermitteln, ob die Bibliothek bereits initialisiert worden ist. Sie ermittelt dazu den Wert des entsprechenden Klassenattributes und gibt diesen zurück. Dabei handelt es sich um `true`, wenn die Initialisierung bereits erfolgt ist, ansonsten um `false`.

- ```
  protected static RelManagerImpl getRelManager(String name)
      throws NotInitializedException, RelManagerException
  ```

  Diese Methode ermittelt in den Klassenattributen ob ein Relationsmanager unter dem übergebenen Namen angelegt wurde und liefert diesen, falls er existent ist, zurück. Ansonsten wird eine Ausnahme der Klasse `RelManagerException` geworfen.

- ```
  protected static Collection getAllRelManagers()
      throws NotInitializedException
  ```

  Diese Methode liefert alle in den Klassenattributen der Klasse `KurejavaOO` vermerkten Relationsmanager zurück.

A.3 Die Schnittstelle `Relation`

Im Folgenden wird die Funktionalität der Methoden, welche einer Client-Anwendung durch die Schnittstelle `Relation` zur Verfügung gestellt werden, beschrieben. Die Schnittstelle wird durch die Klasse `RelationImpl` implementiert.

- `public String getName()`

  Diese Methode liefert den Namen der Relation, der bei ihrer Erstellung angegeben wurde.

- `public int getWidth()`

  Diese Methode liefert die Breite der Relation.

- `public int getHeight()`

  Diese Methode liefert die Höhe der Relation.

- `public int getNumberOfEntries()`

  Diese Methode liefert die Anzahl der Einträge in der Relation.

- ```
  public void setBit (int row,
    int column,
  ```
  ```

```
  boolean value)
    throws RelationException
```

Diese Methode setzt oder löscht einen Eintrag in der Relationsmatrix, abhängig von dem Parameter `value` vom Typ `boolean`.

- ```
  public void clearBit (int row,
    int column)
      throws RelationException
  ```

  Die Methode löscht einen Eintrag in der Relationsmatrix.

- ```
  public boolean getBit(int row,
    int column)
      throws RelationException
  ```

  Diese Methode ermittelt, ob ein Eintrag in der Relationsmatrix gesetzt ist.

- ```
  public String getMatrixString() throws RelationException
  ```

  Diese Methode liefert eine Zeichenkettendarstellung der Relationsmatrix.

- ```
  public String toString()
  ```

  Diese Methode, welche jedes Objekt besitzt, liefert eine textuelle Repräsentation einer Relation. Die Darstellung wird später bei der Implementierung der Schnittstelle beschrieben.

- ```
  public void setObjects(Object[] rowObjects,
    Object[] columnObjects)
      throws RelationException
  ```

  Diese Methode ermöglicht es, den Reihen und Spalten der Relationsmatrix Java-Objekte zuzuordnen. Dies ermöglicht es, nicht immer den Umweg über Reihen- und Spaltennummern gehen zu müssen, sondern direkt beliebige Java-Objekte in Relation zu setzen.

- ```
  public Collection getRelatedRowObjects(Object columnObject)
      throws RelationException
  ```

  Diese Methode liefert die, zu einem mit der Methode `setObjects` gesetzten Objekt, die in Relation stehenden Objekte.

- ```
  public Collection getRelatedColumnObjects(Object rowObject)
      throws RelationException
  ```

  Diese Methode liefert die, zu einem mit der Methode `setObjects` gesetzten Objekt, die in Relation stehenden Objekte.

- ```
  public boolean isTrue() throws RelationException
  ```

  Diese Methode ermittelt, ob alle Einträge in der Relationsmatrix gesetzt sind.

- ```
  public boolean isFalse() throws RelationException
  ```

  Diese Methode ermittelt, ob mindestens ein Eintrag in der Relationsmatrix nicht gesetzt ist.
  ```

Die folgenden drei Methoden sind synonym zu den gleichnamigen oben beschriebenen Methoden und werden daher nicht im Einzelnen erläutert. Die Position der Matrix wird in den folgenden Methoden aber nicht durch Reihen- und Spaltennummer, sondern durch Java-Objekte angegeben. Diese Java-Objekte müssen vorher über die Methode `setObjects` zu den Reihen- und Spaltennummer zugeordnet werden.

- ```
  public void setBit (Object row,
    Object column,
    boolean value)
      throws RelationException
  ```
- ```
  public void clearBit (Object row,
    Object column)
      throws RelationException
  ```
- ```
  public boolean getBit(Object row,
    Object column)
      throws RelationException
  ```

A.4 Die Klasse `RelationImpl`

Im Folgenden werden die Methoden der Klasse `RelationImpl` erläutert.

- `public String getName()`

  Diese Methode gibt den Namen der Relation zurück, der bei ihrer Erstellung im Konstruktor gesetzt und in dem privaten Attribut `private String name` festgehalten wurde.

- `public int getWidth()`

  Diese Methode ermittelt durch einen Aufruf der Methode `relationGetWidth` der Klasse `KurejavaOO` die Breite der Relationsmatrix und liefert diese zurück.

- `public int getHeight()`

  Diese Methode ermittelt durch einen Aufruf der Methode `relationGetHeight` der Klasse `KurejavaOO` die Höhe der Relationsmatrix und liefert diese zurück.

- `public int getNumberOfEntries()`

  Die Methode ruft die Methode `relationGetNumberOfEntries` der Klasse `KurejavaOO` auf und liefert den ermittelten Wert zurück.

- ```
  public void setBit (
    int row,
    int column,
    boolean value)
      throws RelationException
  ```

  Ist die Methode mit dem Wert `false` für den Parameter `value` aufgerufen worden, löscht sie über die Methode `relationClearBit` der Klasse `KurejavaOO` den entsprechenden,

durch die Parameter `row` und `column` angegebenen Eintrag in der Relationsmatrix. Ansonsten wird der Eintrag gesetzt.

- ```
  public void clearBit (
    int row,
    int column)
      throws RelationException
  ```

  Die Methode löscht einen Eintrag der Relationsmatrix über die Methode `relationClearBit` der Klasse `KurejavaOO`.

- ```
  public boolean getBit(
    int row,
    int column)
      throws RelationException
  ```

  Diese Methode prüft über die Methode `getBit` der Klasse `KurejavaOO`, ob ein Eintrag in der Relationsmatrix gesetzt ist.

- `public String getMatrixString() throws RelationException`

  Diese Methode liefert eine Zeichenkettenrepäsentation der Relationsmatrix. Dies ermöglicht es, die Relation zu Debugzwecken auszugeben. Die Darstellung besteht aus Zeilen und Spalten, welche aus `0` und `1` gebildet werden. Die einzelnen Einträge der Relationsmatrix werden durch Aufrufe der oben beschriebenen Methode `getBit` ermittelt.

- `public String toString()`

  Diese Methode konstruiert eine Zeichenkette, welche den Namen der Relation, ihre Breite, ihre Höhe, die Anzahl ihrer Einträge und den Namen des Relationsmanagers, dem sie zugeordnet ist, beinhaltet.

  Der Rückgabewert der Methode für eine unter dem Relationsmanager namens `relManager1` angelegte Relation namens `relation1`, mit zwei Zeilen und zwei Spalten in der Relationsmatrix und keinen Einträgen, sieht wie folgt aus:

  ```
  Relation: relation1
  width  = 2
  height = 2
  entries = 0
  from RelManager relManager1
  ```

- ```
  public void setObjects(Object[] rowObjects,
    Object[] columnObjects)
      throws RelationException
  ```

  Diese Methode erwartete zwei `Object-Arrays` als Parameter. Das erste `Array` beinhaltet Java-Objekte, die den einzelnen Reihen der Relationsmatrix zugeordnet werden sollen, und daher muss die `Array`-Länge mit der Höhe der Relationsmatrix übereinstimmen. Gleiches gilt für den zweiten Parameter, der die Objekte für die Spalten der Relation beinhaltet. Die Methode speichert, welche der übergebenen Java-Objekte zu welcher Reihen-, bzw. Spaltennummer zugeordnet werden sollen. Dabei ist die Reihenfolge der Java-Objekte in den `Arrays` entscheidend.
  ```

In dem folgenden Beispiel wird der Reihennummer `1` das Objekt `r1`, der Reihennummer `2` das Objekt `r2`, der Spaltennummer `1` das Objekt `c1` und der Spaltennummer `2` das Objekt `c2` zugeordnet. Die Methoden `setBit`, `clearBit` und `getBit` können - dank Überladung der Methoden mit Objekten als Parametern - mit den zugeordneten Objekten statt der Reihen- und Spaltennummern aufgerufen werden.

```
String r1 = "r1";
String r2 = "r2";
String c1 = "c1";
String c2 = "c2";
Object[] rowObjects    = new Object[]{r1, r2};
Object[] columnObjects = new Object[]{c1, c2};
relation.setObjects(rowObjects, columnObjects);
```

- `public void setBit (Object row,`
  `Object column,`
  `boolean value)`
  `throws RelationException`

  Diese Methode ermittelt die Reihen- und Spaltennummer zu den übergebenen Objekten und ruft damit oben beschriebene gleichnamige `setBit`-Methode auf. Wurden die übergebenen Objekte keiner Reihen- und Spaltennummer zugeordnet, wird eine `RelationException` ausgelöst.

- `public void clearBit (Object row,`
  `Object column)`
  `throws RelationException`

  Diese Methode verhält sich analog zur Methode `setBit`.

- `public boolean getBit(Object row,`
  `Object column)`
  `throws RelationException`

  Diese Methode verhält sich analog zur Methode `setBit`.

- `public boolean isTrue() throws RelationException`

  Diese Methode prüft, ob alle Einträge in der Relationsmatrix `1`-Einträge sind.

- `public boolean isFalse() throws RelationException`

  Diese Methode prüft, ob alle Einträge in der Relationsmatrix `0`-Einträge sind.

Im Folgenden werden die zusätzlichen Methoden erläutert, welche in der Klasse `RelationImpl` deklariert und implementiert werden und die nicht bereits in der Schnittstelle `Relation` deklariert wurden. Diese Methoden können nicht von der Client-Anwendung aufgerufen werden, sondern nur innerhalb der Bibliothekklassen selbst.

- `protected RelationImpl(RelManagerImpl relManager,`
  `String name,`
  `byte[] pointer)`

  Der Konstruktor legt eine Instanz der Klasse `RelationImpl` an, vermerkt, welchem Relationsmanager die Instanz zugeordnet ist, welcher Name der Relation gegeben wurde und den C-Zeiger auf den Speicherbereich der Relation in der nativen Bibliothek als `Byte-Array`.

- `protected byte[] getPointer()`

  Diese Methode liefert den vermerkten C-Zeiger als `Byte-Array`.

- `protected void setPointer(byte[] pointer)`

  Diese Methode setzt einen neuen C-Zeiger als `Byte-Array`.

- `protected RelManagerImpl getRelManager()`

  Diese Methode liefert das Relationsmanagerobjekt zurück, das die Relation verwaltet.

- `protected void delete()`

  Diese Methode vermerkt, dass die Relation gelöscht wurde und ermöglicht damit, bei jedem Methodenaufruf an der Instanz zu prüfen, ob die repäsentierte Relation bereits entfernt wurde und die Methoden nicht aufgerufen werden dürfen.

A.5 Die Schnittstelle `RelManager`

Im Folgenden werden die Methoden beschrieben, welche die Schnittstelle `RelManager` deklariert. Die Schnittstelle wird in der Klasse `RelManagerImpl` implementiert.

- `public Relation getRelation (String relationName)`
  `throws RelationException`

  Diese Methode ermittelt die unter dem übergebenen Relationsnamen in dem Relationsmanager verwaltete Relation und liefert diese zurück.

- `public void deleteRelation (String relationName)`
  `throws RelationException`

  Diese Methode löscht die unter dem übergebenen Relationsnamen in dem Relationsmanager verwaltete Relation.

- `public void deleteRelation (Relation relation)`
  `throws RelationException`

  Diese Methode löscht die durch den Parameter `relation` angegebene Relation.

- `public Relation createRelation(String relationName,`
  `int width,`
  `int height)`
  `throws RelationException;`

  Diese Methode erstellt eine neue Relation, welche bei dieser Relationsmanager-Instanz verwaltet wird und gibt diese neue Relation zurück.

- `public String getName()`

  Diese Methode liefert den Namen der Relationsmanager-Instanz.

- `public boolean containsRelation(String relationName)`

  Diese Methode ermittelt, ob die Relationsmanager-Instanz eine Relation mit dem übergebenen Namen beinhaltet.

- `public Collection getAllRelations()`

  Diese Methode liefert alle von dem Relationsmanager-Objekt verwalteten Relationen.

- ```
  public void createFunction(String function)
      throws FunctionException
  ```

  Diese Methode definiert eine neue relationale Funktion, die unter dieser Relationsmanager-Instanz in relationalen Termen genutzt werden kann.

- `public Collection getAllFunctions()`

  Diese Methode liefert alle innerhalb dieses Relationsmanager-Objektes definierten relationalen Funktionen als Java-Datentyp `Collection`, welcher die Funktionsdeklarationen als `String`s beinhaltet.

- ```
  public void createProgram(String program)
      throws ProgramException
  ```

  Diese Methode definiert ein neues relationales Programm, welches unter dieser Relationsmanager-Instanz in relationalen Termen genutzt werden kann.

- ```
  public void loadProgram (URL url)
      throws ProgramException
  ```

  Die Methode lädt relationale Programme aus einer Datei, welche dann unter der Relationsmanager-Instanz in relationalen Termen genutzt werden können.

- ```
  public void evaluateTerm(String term,
    Relation resultRelation)
      throws TermException, RelationException
  ```

  Diese Methode wertet einen relationalen Term aus. Innerhalb des Termes können lediglich Relationen, Funktionen und Programme genutzt werden, die innerhalb dieses Relationsmanager-Objektes verwaltet werden. Das Ergebnis der Termauswertung befindet sich danach in dem übergebenen Relations-Objekt.

- ```
  public Relation evaluateTerm(String term)
      throws TermException, RelationException
  ```

  Diese Methode ist analog zu der gleichnamigen oben beschriebenen Methode. Das Ergebnis wird in eine Relation mit dem Namen `result` geschrieben, welche entweder überschrieben, falls sie bereits vorhanden ist oder sonst neu erstellt wird. Diese Relation wird zurückgeliefert.

- ```
  public Relation evaluateTerm(String term, String resultName)
      throws TermException, RelationException
  ```

Diese Methode arbeitet analog zu der gleichnamigen oben beschriebenen Methode. Das Ergebnis wird in eine Relation mit dem übergebenen Namen geschrieben, welche entweder überschrieben, falls sie bereits vorhanden ist oder sonst neu erstellt wird. Diese Relation wird zurückgeliefert.

- `public String toString()`

  Liefert eine Zeichenkettenrepräsentation des Relationsmanagers.

A.6 Die Klasse `RelManagerImpl`

Im Folgenden werden die Methoden der Klasse `RelManagerImpl` erläutert.

- ```
  public Relation getRelation (String relationName)
      throws RelationException
  ```

  Diese Methode ermittelt aus einem Objektattribut des Relationsmanager-Objektes, in der alle verwalteten Relationen abgelegt sind, die Relation mit dem übergebenen Namen und gibt das entsprechende Relationsobjekt zurück. Verwaltet der Relationsmanager keine Relation mit diesem Namen, wird eine `RelationException` ausgelöst.

- ```
  public void deleteRelation (String relationName)
      throws RelationException
  ```

  Diese Methode löscht die von der Relationsmanager-Instanz verwaltete Relation mit dem übergebenen Namen. Verwaltet der Relationsmanager keine Relation des übergebenen Namens, wird eine `RelationException` ausgelöst.

- ```
  public void deleteRelation (Relation relation)
      throws RelationException
  ```

  Diese Methode löscht die übergebene Relation, wenn sie vom Relationsmanager verwaltet wird. Sonst wird eine `RelationException` ausgelöst.

- ```
  public Relation createRelation(String relationName,
    int width,
    int height)
      throws RelationException;
  ```

  Diese Methode erstellt über die Methode `relationNew` der Klasse `KurejavaOO` eine neue Relation und speichert einen Vermerk auf die Relation in dem Objektattribut `private Map relations` des Relationsmanager-Objektes. Hiermit kann geprüft werden, welche Relationen von dem Relationsmanager verwaltet werden. Verwaltet das Relationsmanager-Objekt bereits eine Relation mit diesem Namen, wird eine `RelationException` ausgelöst.

- `public String getName()`

  Diese Methode gibt den Namen des Relationsmanagers zurück, der beim Erstellen des Relationsmanagers angegeben wurde.

- `public boolean containsRelation(String relationName)`

  Diese Methode ermittelt, ob in dem Relationsmanager-Objekt eine Relation mit dem übergebenen Namen verwaltet wird.

- `public Collection getAllRelations()`

  Diese Methode liefert alle von dem Relationsmanager-Objekt verwalteten Relationen zurück. Die Vermerke auf die verwalteten Relationen sind in Objektattributen der Relationsmanager-Instanz abgelegt. Wenn der Relationsmanager keine Relationen verwaltet, wird eine leere Java-`Collection` zurückgegeben. Bei Java-`Collection` handelt es sich in Java um Sammlungen von Objekten. Die Elemente, die ein Java-`Collection`-Objekt beinhaltet, können einzeln abgefragt oder wie in einer Liste durchlaufen werden.

- `public void createFunction(String function)`
  `throws FunctionException`

  Diese Funktion erstellt über die Methode `functionNew` der Klasse `KurejavaOO` eine neue relationale Funktion, die zu dieser Relationsmanager-Instanz gehört und vermerkt diese in Objektattributen des Relationsmanager-Objektes. Die erstellte Funktion kann nur in Termen benutzt werden, welche über diesen Relationsmanager ausgewertet werden. Kann die Funktion nicht erstellt werden, wird eine `FunctionException` ausgelöst.

- `public Collection getAllFunctions()`

  Diese Methode ermittelt die Liste der von dieser Relationsmanager-Instanz verwalteten Funktion aus Objektattributen des Relationsmanagers und liefert diese als Java-`Collection` zurück.

- `public void createProgram(String program)`
  `throws ProgramException`

  Diese Funktion erstellt über die Methode `programNew` der Klasse `KurejavaOO` ein neues relationales Programm, welches zu dieser Relationsmanager-Instanz gehört. Das Programm kann nur in Termen benutzt werden, welche über diesen Relationsmanager ausgewertet werden. Kann das Programm nicht erstellt werden, wird eine `ProgramException` ausgelöst.

- `public void loadProgram (URL url)`
  `throws ProgramException`

  Diese Methode nutzt die Methode `programFileRead` der Klasse `KurejavaOO`, um ein relationales Programm aus einer Datei einzulesen.

- `public void evaluateTerm(String term,`
  `Relation resultRelation)`
  `throws TermException, RelationException`

  Diese Methode wertet über die Methode `evaluateTerm` der Klasse `KurejavaOO` einen relationalen Term aus und legt das Ergebnis der Auswertung in die übergebene Relation ab. Verwaltet der Relationsmanager die übergebene Relation nicht, wird eine `RelationException` ausgelöst. Treten Fehler bei der Termauswertung auf, wird eine `TermException` ausgelöst.

- `public Relation evaluateTerm(String term)`
  `throws TermException, RelationException`

Diese Methode verhält sich analog zur gleichnamigen oben beschriebenen Methode, aber hier wird das Ergebnis in eine Relation mit dem Namen `result` beim Relationsmanager-Objekt abgelegt. Gibt es bereits eine Relation mit diesem Namen, wird sie überschrieben, ansonsten wird die Relation neu erstellt.

- `public Relation evaluateTerm(String term, String resultName) throws TermException, RelationException`

  Diese Methode verhält sich analog zur gleichnamigen oben beschriebenen Methode, aber hier wird das Auswertungsergebnis in eine Relation mit dem übergebenen Namen unter diesem Relationsmanager-Objekt abgelegt. Gibt es bereits eine Relation mit diesem Namen, wird sie überschrieben, ansonsten wird die Relation neu erstellt.

- `public String toString()`

  Liefert eine Zeichenkettenrepräsentation des Relationsmanagers in der Form `RelManager: NAME with x Relation entries`.

A.7 Die Klasse `Context`

Im Folgenden sind die Methoden erläutert, welche die Klasse `Context` anbietet. Dabei handelt es sich ausschließlich um statische Methoden, von der Klasse `Context` selbst wird kein Objekt instantiiert.

- ```
  public static void init (File library)
      throws AlreadyInitializedException,
             NotInitializedException;
  ```

  Die Methode initialisiert die Bibliothek und lädt die native Bibliothek, welche die C-Funktionen zur Verfügung stellt, die die Bibliothek nutzt. Die zu ladende native Bibliothek wird als Instanz der Klasse Java-`File`, in Java ein Objekt das eine physische Datei repräsentiert, der Methode übergeben.

  Ist die native Bibliothek geladen, prüft die Methode durch Aufruf der unten beschriebenen Methode `checkLibrary`, ob die geladene native Bibliothek alle benötigten Funktionen zur Verfügung stellt. Erst wenn dieser Test korrekt beendet wird, ist die Bibliothek initialisiert und kann von der Client-Anwendung genutzt werden.

- ```
  public static void init(String libraryname)
      throws AlreadyInitializedException,
             NotInitializedException;
  ```

  Diese Methode ist analog zur gleichnamigen, oben beschriebenen Methode. Hier wird der Verweis auf die zu ladende native Bibliothek als Zeichenkette, welche den vollständigen Pfad enthält, übergeben.

- ```
  public static KurejavaHome getHome()
      throws NotInitializedException;
  ```
  ```

Diese Methode liefert ein Objekt zurück, welches über die nachfolgend erläuterte Schnittstelle `Home` von der Client-Anwendung angesprochen werden kann. Dieses Objekt dient als Fabrik für Relationsmanager-Objekte. Das Objekt wird über die statische Methode der Klasse `HomeImpl` ermittelt. Bei jedem Aufruf der Methode wird stets dasselbe Objekt zurückgegeben, das beim ersten Aufruf der Methode erzeugt wird (vgl. dazu auch die Beschreibung der Klasse `HomeImpl` in Abschnitt 6.1.2 auf Seite 51).

Wurde die Bibliothek vorher nicht über eine der beiden `init`-Methoden erfolgreich initialisiert, so wird eine `NotInitializedException` geworfen. Die Prüfung, ob die Bibliothek bereits initialisiert wurde, erfolgt mit der weiter unten beschriebenen Methode `checkInit`.

- `private static boolean checkLibrary();`

  Diese Methode prüft, ob die Bibliothek korrekt auf die Funktionen der nativen Bibliothek zugreifen kann. Ist dies der Fall liefert sie `true` zurück, sonst `false`. Für den Test ruft sie jede der benötigten Funktionen der nativen Bibliothek über die nativen Methoden der Klasse `Kurejava` auf.

  Auf die Methode `checkLibrary` kann nur innerhalb der Klasse `Context` selbst zugegriffen werden. Dies geschieht in den beiden `init`-Methoden.

- ```
  public static void quit()
      throws NotInitializedException
  ```

  Diese Methode beendet die Bibliothek. Der Zugriff auf die Bibliothek ist danach nicht mehr möglich. Soll die Bibliothek erneut benutzt werden, muss sie zuerst wieder über eine der beiden oben beschriebenen `init`-Methoden initialisiert werden.

  Wird diese Methode aufgerufen und wurde die Bibliothek vorher noch nicht initialisiert, wird eine Ausnahme vom Typ `NotInitializedException` ausgelöst. Wie auch in der Methode `getHome` erfolgt die Prüfung mit der unten beschriebenen Methode `checkInit`.

- `private static void checkInit()`

  Diese Methode prüft, ob die native Bibliothek bereis initialisiert wurde. Auf die Methode `checkLibrary` kann nur innerhalb der Klasse `Context` selbst zugegriffen werden. Sie wird von den Methoden `getHome` und `quit` genutzt, um zu prüfen, ob die Bibliothek bereits initialisiert wurde, und um dann eine Ausnahme vom Typ `NotInitializedException` auszulösen.

A.8 Die Schnittstelle `Home`

Im Folgenden wird die Funktionalität der Methoden der Klasse `Home`, welche einer Client-Anwendung zur Verfügung gestellt wird, beschrieben. Auf ihre konkrete Implementierung wird im Abschnitt A.9 auf Seite 100 eingegangen. Allen diesen Methoden ist gemeinsam, dass sie eine Ausnahme vom Typ `NotInitializedException` auslösen, wenn sie aufgerufen werden, ohne dass die Bibliothek initialisiert wurde.

- ```
  public RelManager createRelManager(String name)
      throws NotInitializedException,
  ```

```
RelManagerException
```

Diese Methode erzeugt einen neuen Relationsmanager mit dem übergebenen Namen. Konnte der Relationsmanager nicht angelegt werden, löst die Methode eine Ausnahme vom Typ `RelManagerException` aus.

- ```
  public Collection getAllRelManagers()
      throws NotInitializedException
  ```

  Diese Methode liefert eine Java-`Collection` aller erzeugten Relationsmanager, welche nicht bereits wieder gelöscht wurden.

- ```
  public RelManager getRelManager(String name)
      throws RelManagerException,
             NotInitializedException
  ```

  Diese Methode ermittelt zu dem übergebenen Namen den vorher erzeugten Relationsmanager. Wurde kein Relationsmanager mit dem übergebenen Namen angelegt oder dieser bereits gelöscht, wird die Ausnahme `RelManagerException` geworfen.

- ```
  public void deleteRelManager(RelManager relManager)
      throws RelManagerException
  ```

  Diese Methode löscht den übergebenen Relationsmanager. Tritt beim Löschvorgang ein Fehler auf, wird eine Ausnahme vom Typ `RelManagerException` ausgelöst.

- ```
  public void deleteRelManager(String name)
      throws RelManagerException,
             NotInitializedException
  ```

  Diese Methode verhält sich analog zur gleichnamigen oben beschriebenen Methode, nur dass dieser Methode der zu löschende Relationsmanager mit Namen übergeben wird.

- ```
  public void deleteAllRelManager()
      throws NotInitializedException
  ```

  Diese Methode löscht alle existierenden Relationsmanager.

A.9 Die Klasse `HomeImpl`

Im Folgenden werden die Methoden der Klasse `HomeImpl` erläutert. Allen im Folgenden beschriebenen Methoden ist gemeinsam, dass sie eine Ausnahme vom Typ `NotInitializedException` auslösen, wenn sie aufgerufen werden, ohne dass die Bibliothek initialisiert wurde.

- ```
  public RelManager createRelManager(String name)
      throws NotInitializedException,
             RelManagerException
  ```

  Diese Methode erstellt mit der Methode `initRelManager` der Klasse `KurejavaOO` einen neuen Relationsmanager und gibt diesen zurück. Ein Relationsmanager wird durch ein Objekt repräsentiert, welches über die Schnittstelle `RelManager` angesprochen werden kann. Beim

Erzeugen aufgetretene Ausnahmen vom Typ `RelManagerException` werden an den Aufrufer weitergeleitet.

- ```
  public Collection getAllRelManagers()
      throws NotInitializedException
  ```

  Diese Methode ermittelt alle vermerkten Relationsmanager-Objekte und liefert diese zurück. Dazu greift sie auf die Methode `getAllRelManagers` der Klasse `KurejavaOO` zu.

- ```
  public RelManager getRelManager(String name)
      throws RelManagerException,
             NotInitializedException
  ```

  Die Methode sucht unter den vermerkten Relationsmanager-Objekten einen Relationsmanager mit dem übergebenen Namen über die Methode `getRelManager` der Klasse `KurejavaOO`. Existiert ein solcher Relationsmanager wird dieser zurückgeliefert, ansonsten eine Ausnahme vom Typ `RelManagerException` geworfen.

- ```
  public void deleteRelManager(RelManager relManager)
      throws RelManagerException
  ```

  Diese Methode löscht das übergebene Relationsmanager-Objekt über die Methode `delete` der Klasse `RelManagerImpl`.

- ```
  public void deleteRelManager(String name)
      throws RelManagerException,
             NotInitializedException
  ```

  Diese Methode verhält sich analog zur gleichnamigen oben beschriebenen Methode. Zu dem übergeben Namen wird zusätzlich über die oben beschriebene Methode `getRelManager` das zu löschende Relationsmanager-Objekt ermittelt.

- ```
  public void deleteAllRelManager()
      throws NotInitializedException
  ```

  Diese Methode ermittelt über die oben beschriebene Methode `getAllRelManagers` alle existierenden Relationsmanager-Objekte und löscht diese über die Methode `delete` der Klasse `RelManagerIml`.

- ```
  public static Home getHome()
  ```

  Diese Methode erzeugt bei ihrem ersten Aufruf eine Instanz der Klasse `HomeImpl`. Diese Instanz wird bei jedem Aufruf der Methode zurückgeliefert.

  Ausschließlich über diese Methode ist es möglich, eine Instanz der Klasse `HomeImpl` zu erzeugen. Diese Methode wird von der Klasse `Context` genutzt, die in Abschnitt 6.1 auf Seite 49 beschrieben ist, um ein Fabrikobjekt für Relationsmanager anzulegen.

A.10 Die Klasse `Prototype`

Im Folgenden werden die Methoden der Klasse `Prototype` erläutert:

- `public void run(String xmlURL, IJavaProject project)`
  `throws AlreadyInitializedException, NotInitializedException,`
  `FileNotFoundException, JAXBException, TermException,`
  `RelManagerException, RelationException,`
  `CreateException, FunctionException`

  Diese Methode ruft die unten beschriebenen Methoden `init`, `define`, `visualize` und `checkRelations` nacheinander auf. Wenn bis zu diesem Zeitpunkt kein Fehler ausgelöst wurde, und der Methode ein Eclipse-Java-Objekt durch den Parameter `project` übergeben wurde, legt die Methode Pakete, Schnittstellen und Klassen mit Hilfe der später beschriebenen Klasse `EclipseProjectRealizer` an, die, in der durch den Parameter `xmlURL` referenzierten XML-Datei, genannt sind. Zuletzt wird die Methode `shutdown` aufgerufen. Eventuelle Ausnahmen, die in einem der gekapselten Methodenaufrufe auftreten, werden an den Aufrufer weitergeleitet.

- `private void init(String xmlURL)`
  `throws AlreadyInitializedException, NotInitializedException,`
  `FileNotFoundException, JAXBException`

  In dieser Methode wird die XML-Datei, deren absoluter Dateiname in dem Parameter `xmlURL` angegeben ist, mit *JAXB* eingelesen. *JAXB* erzeugt beim Einlesen der Datei Instanzen der vorher aus dem genannten XML-Schema generierten Klassen. Das Objekt, welches das Wurzelelement der XML-Struktur repräsentiert, wird in dem Attribut `root` gesichert und im Folgenden als Wurzelobjekt bezeichnet. Über dieses Objekt können im weiteren die einzelnen Elemente und Attribute der XML-Struktur abgefragt werden. Die Methode ermittelt den Pfad zu der Bibliothek über das Wurzelobjekt und initialisiert die Bibliothek. Auch Java-Ausgabeströme in die Dateien, in die Ausgaben erfolgen sollen, werden angelegt. Eventuelle Ausnahmen, die auftreten, werden an den Aufrufer weitergeleitet.

- `private void define() throws CreateException`

  Diese Methode erzeugt ein Objekt der Klasse `Definer` und ermittelt darüber die in der XML-Struktur angegebenen Pakete, Schnittstellen und Klassen. Eventuelle Ausnahmen, die auftreten, werden an den Aufrufer weitergeleitet.

- `private void visualize() throws CreateException`

  Diese Methode erzeugt ein Objekt der Klasse `Visualizer` und verarbeitet darüber den Visualisierungsabschnitt der XML-Struktur. Eventuelle Ausnahmen, die auftreten, werden an den Aufrufer weitergeleitet.

- `private void checkRelations()`
  `throws CreateException, NotInitializedException,`
  `RelManagerException, RelationException,`
  `TermException, FunctionException`

  Diese Methode legt eine Objekt der Klasse `RelationDefiner` an und lässt dieses Objekt alle benötigten Relationen bilden. Danach wird ein Objekt der Klasse `RelationEvaluator` erzeugt, und die Relationen darüber auf Eigenschaften geprüft. Die zu prüfenden Eigenschaften sind in der Klasse `RelationEvaluator` fest als relationale Terme und Funktionen implementiert und bei deren Beschreibung erläutert. Eventuelle Ausnahmen, die auftreten, werden an den Aufrufer weitergeleitet.

- `public void shutdown()`
  `throws NotInitializedException`

  Diese Methode bereinigt den Speicher und beendet die Bibliothek. Eventuelle Ausnahmen, die auftreten, werden an den Aufrufer weitergeleitet.

A.11 Die Klasse `RelationDefiner`

Im Folgenden werden die Methoden der Klasse `RelationDefiner` erläutert:

- `private void defineInterfacesRel()`
  `throws RelationException`

  Diese Methode erstellt über die Bibliothek eine neue Relation mit dem Namen `interfaces`. Diese Relation steht für die `extends`-Beziehungen zwischen Schnittstellen. Die Breite und Höhe der Relationsmatrix beträgt die Größe der `Map interfacedef` der einzigen Instanz der Klasse `ComponentMap`. Die entsprechenden Objekte in der `Map`, welche die Schnittstellen repräsentieren, werden als Relationsobjekte über die Methode `setObjects` der Klasse Relation angegeben. Auf diese Weise müssen beim Setzen und Abfragen von Einträgen zur Relation keine Matrixpositionen mehr angegeben werden, sondern es können direkt Relationen zwischen Objekten gesetzt und abgefragt werden. Das Setzen der Einträge der Relation geschieht in der Methode `setEntriesInterfacesRelByWalls`, die weiter unten beschrieben ist.

- `private void defineClassesRel()`
  `throws RelationException`

  Diese Methode erstellt über die Bibliothek eine neue Relation mit dem Namen `classes`. Die Relation steht für die Vererbungsbeziehungen zwischen Klassen. Die Breite und Höhe der Relationsmatrix beträgt die Größe der `Map classdef` der einzigen Instanz der Klasse `ComponentMap`. Die entsprechenden Objekte in der `Map`, welche die Klassen repräsentieren, werden als Relationsobjekte über die Methode `setObjects` der Klasse `Relation` angegeben. In der Methode `setEntriesClassesRelByConetrees`, die weiter unten beschrieben ist, ist das Setzen der Einträge der Relation realisiert.

- `private void defineClassesInterfacesRel()`
  `throws RelationException`

  In dieser Methode wird eine neue Relation mit dem Namen `classesInt` erstellt. Diese Relation beinhaltet die `implements`-Beziehungen zwischen Klassen und Schnittstellen. Die Höhe der Relationsmatrix beträgt dabei die Größe der `Map interfacedef`, die Breite die Größe der `Map classdef` der einzigen Instanz der Klasse `ComponentMap`. Die entsprechenden Objekte in der `Map`, welche die Schnittstellen, bzw. die Klassen repräsentieren, werden als Relationsobjekte über die Methode `setObjects` der Klasse `Relation` angegeben. In der Methode `setEntriesClassesInterfacesRelByConetrees`, die weiter unten beschrieben ist, ist das Setzen der Einträge der Relation realisiert.

- `private void definePackagesRel()`
  `throws RelationException`

Diese Methode erstellt über die Bibliothek eine neue Relation mit dem Namen `packages`. Diese Relation steht für die Unterpaket-Beziehungen zwischen Paketen. Die Breite und Höhe der Relationsmatrix beträgt die Größe der `Map packagedef` der einzigen Instanz der Klasse `ComponentMap`. Die entsprechenden Objekte in der `Map`, welche die Pakete repräsentieren, werden als Relationsobjekte über die Methode `setObjects` der Klasse `Relation` angegeben. In der Methode `setEntriesPackagesRelByCubes`, die weiter unten beschrieben ist, ist das Setzen der Einträge der Relation realisiert.

- `private void definePackagesClassesRel()`
  `throws RelationException`

  Die Methode erstellt über die Bibliothek eine neue Relation mit Namen `packagesClasses`. Diese Relation steht für die Beziehung von Paketen zu den in ihnen enthaltenen Klassen. Die Höhe der Relationsmatrix beträgt dabei die Größe der `Map classdef`, die Breite die Größe der `Map packagedef` der einzigen Instanz der Klasse `ComponentMap`. Die entsprechenden Objekte in der `Map`, welche die Pakete repräsentieren, werden als Relationsobjekte über die Methode `setObjects` der Klasse `Relation` angegeben. Das Setzen der Einträge der Relation geschieht in der Methode `setEntriesPackagesClassesRelByCubes`, die weiter unten beschrieben ist.

- `private void definePackagesInterfacesRel()`
  `throws RelationException`

  Diese Methode erstellt über die Bibliothek eine neue Relation mit dem Namen `packagesInt`. Diese Relation steht für die Beziehung von Paketen zu den in ihnen enthaltenen Schnittstellen. Die Höhe der Relationsmatrix beträgt dabei die Größe der `Map interfacedef`, die Breite die Größe der `Map packagedef` der einzigen Instanz der Klasse `ComponentMap`. Die entsprechenden Objekte in der `Map`, welche die Pakete repräsentieren, werden als Relationsobjekte über die Methode `setObjects` der Klasse `Relation` angegeben. Das Setzen der Einträge der Relation geschieht in der Methode `setEntriesPackagesInterfacesRelByCubes`, die weiter unten beschrieben ist.

- `private void setEntriesInterfacesRelByWalls()`
  `throws RelationException`

  In dieser Methode werden alle `WallComponent`-Objekte, die aus der Menge `wall` der einzigen Instanz der Klasse `ComponentMap` ermittelt werden, durchlaufen. Für jedes dieser Objekte werden alle zu der repräsentierten `Wall` gehörenden Schnittstellen ermittelt und entsprechende Einträge in der Relation `interfaces` gesetzt, wenn eine Schnittstelle in der `Wall` eine Verbindung zu einer anderen Schnittstelle hat.

- `private void setEntriesClassesRelByConetrees()`
  `throws RelationException`

  In der Methode werden alle `ConetreeComponent`-Objekte, die aus der Menge `conetree` der einzigen Instanz der Klasse `ComponentMap` ermittelt werden, durchlaufen. Für jedes dieser Objekte werden die zu dem repräsentierten `Cone Tree` gehörenden Klasse der `Cone Tree`-Spitze ermittelt und entsprechende Einträge in der Relation `classes` zu den weiteren Klassen, die den `Cone Ttree` bilden, gesetzt.

- `private void setEntriesPackagesRelByCubes()`

```
    throws RelationException
```

In dieser Methode werden alle `CubeComponent`-Objekte, die aus der Menge `cube` der einzigen Instanz der Klasse `ComponentMap` ermittelt werden, durchlaufen. Für jedes dieser Objekte wird das zu dem repräsentierten `Cube` gehörende Paket ermittelt und entsprechende Einträge in der Relation `packages` zu den weiteren Paketen der `Cube`s, die in dem `Cube` liegen, gesetzt.

- `private void setEntriesPackagesClassesRelByCubes()`
  `throws RelationException`

  In dieser Methode werden alle `CubeComponent`-Objekte, die aus der Menge `cube` der einzigen Instanz der Klasse `ComponentMap` ermittelt werden, durchlaufen. Für jedes dieser Objekte wird das zu dem repräsentierten `Cube` gehörende Paket ermittelt und entsprechende Einträge in der Relation `packagesClasses` zu den Klassen, die in dem `Cube` liegen, gesetzt.

- `private void setEntriesPackagesInterfacesRelByCubes()`
  `throws RelationException`

  In dieser Methode werden alle `CubeComponent`-Objekte, die aus der Menge `cube` der einzigen Instanz der Klasse `ComponentMap` ermittelt werden, durchlaufen. Für jedes dieser Objekte wird das zu dem repräsentierten `Cube` gehörende Paket ermittelt und entsprechende Einträge in der Relation `packagesInt` zu den Schnittstellen, die in dem `Cube` liegen, gesetzt.

- `private void setEntriesClassesInterfacesRelByConetrees()`
  `throws RelationException`

  In dieser Methode werden alle `ConetreeComponent`-Objekte, die aus der Menge `conetree` der einzigen Instanz der Klasse `ComponentMap` ermittelt werden, durchlaufen. Für jedes dieser Objekte wird die zu dem repräsentierten `Cone Ttree` gehörenden Klasse der `Cone Tree`-Spitze ermittelt und entsprechende Einträge in der Relation `classesInt` zu den Schnittstellen, die in dem `Cone Tree` liegen, gesetzt.

A.12 Die Klasse `WallComponent`

Im Folgenden werden die Methoden der Klasse `WallComponent` erläutert:

- `protected void addInterface(String id)` Die Methode fügt eine Schnittstelle unter dem Wert `id` der `Map interfaces` hinzu und vermerkt eine leere Menge für die `extends`-Beziehungen.

- `protected void addExtendsIDToInterfaceID`
  `(String id, String extendid)`
  `throws CreateException}`

  Die Methode ermittelt aus der `Map interfaces` die zu der mit `id` referenzierten Schnittstelle gehörige Menge an `extends`-Beziehungen und trägt den Wert von `extendid` dazu ein. Existiert keine Schnittstelle, dem der Wert `id` zugeordnet ist, so wird die Ausnahme `CreateException` ausgelöst.

- `protected Set getInterfacesID()` Die Methode gibt eine Menge aller in der repräsentierten `Wall` zugehörigen Schnittstellen-IDs in Form des Parameter `id` der Methode `addInterface` zurück.
- `protected Set getExtendsIDFromInterfaceID(String id)` Die Methode gibt die zu dem Parameter `id` in der `Map interfaces` hinterlegte Menge von Schnittstellenidentifizierern zurück.

A.13 Die `Definer`-Komponente

Im Folgenden werden die Methoden der Klasse `Definer` erläutert:

- `protected void definePackages() throws CreateException`

  Über das Wurzelobjekt werden die in der XML-Datei angegebenen Pakete ermittelt und von jedem dieser Pakete das Attribut `id` und `name` ausgelesen. Es wird geprüft, ob bereits ein Paket mit dieser `id` in der `Map packagedef` vorhanden ist. Ist dies der Fall, wird die Ausnahme `CreateException` geworfen, ansonsten eine Instanz der Klasse `PackageComponent` mit den Werten `id` und `name` erzeugt und in der `Map` abgelegt.
- `protected void defineInterfaces() throws CreateException`

  Verhält sich analog zur Methode `definePackages`. Hier werden die Schnittstellen ausgelesen und in der `Map interfacedef` abgelegt.
- `protected void defineClasses() throws CreateException`

  Verhält sich analog zur Methode `definePackages`. Hier werden die Klassen ausgelesen und in der `Map classdef` abgelegt.

A.14 Die Klasse `Util`

Im Folgenden werden die Methoden der Klasse `Util` erläutert. Drei dieser Methoden sind öffentlich, die vierte ist nur innerhalb dieser Klasse gültig und wird als Hilfsmethode benutzt.

- ```
  public static boolean createClass
    (IJavaProject project, String packageName,
     String name, String[] imp,
     String[] ext, String[] impl)
  ```

  Diese Methode generiert eine Klasse in einem Eclipse-Projekt. Sie ruft dazu die Methode `createClassOrInterface` auf, wobei der Parameter `isInterface` mit `false` als Wert belegt wird.
- ```
  public static boolean createInterface
    (IJavaProject project, String packageName,
     String name, String[] imp,
  ```

```
   String[] ext)
```

Diese Methode generiert eine Schnittstelle in einem Eclipse-Projekt. Sie ruft dazu die Methode `createClassOrInterface` auf, wobei der Parameter `isInterface` mit `true` als Wert belegt wird.

- ```
  private static boolean createClassOrInterface
    (IJavaProject project, String packageName,
     String name, boolean isInterface,
     String[] imp, String[] ext,
     String[] impl)
  ```

  Diese Methode erstellt eine neue Schnittstelle, bzw. eine neue Klasse in dem angegebenen Paket in einem Eclipse-Projekt. Ob es sich bei der Erstellung um eine Schnittstelle oder eine Klasse hält, wird mit dem Parameter `isInterface` angegeben. Der Methode wird das Elicpse-Projekt in Form eines Objektes, welche die von Eclipse definierte Schnittstelle `IJavaProject` implementiert, übergeben. Über dieses Objekt wird zu dem angegebenen Paket ein das Paket repräsentierendes Objekt ermittelt. Mit diesem Objekt wird eine Quellcode-Datei in dem Eclipse-Projekt erstellt. Diese Quellcode-Datei wird mit dem für Java-Schnittstellen, bzw. Java-Klassen üblichen Dateikopf versehen. Dazu gehören die Anweisungen `package` und `import`. Die `package`-Anweisung ist gefolgt von dem zugehörigen Paketnamen und einem Semikolon. `import`-Anweisungen können mehrere auftreten. Für jede Zeichenkette in dem Parameter `imp` wird eine `import`-Anweisung, gefolgt von der Zeichenkette, die die zu importierende Klasse angibt, und einem Semikolon im Dateikopf aufgenommen. Danach folgt das Schlüsselwort `public`, gefolgt von dem Schlüsselwort `interface`, wenn eine Schnittstelle und `class`, wenn eine Klasse generiert werden soll. Danach kommt der Name der zu erstellenden Schnittstelle, bzw. Klasse, welcher durch den Parameter `name` angegeben wird. Enthält der Parameter `ext` mindestens eine Zeichenkette, folgt hinter dem Namen das Schlüsselwort `extends` und die Zeichenketten aus dem Parameter `ext`, jeweils von einem Komma getrennt. Diese Zeichenketten geben dabei im Falle einer zu generierenden Schnittstelle andere Schnittstellen an, welche erweitert werden sollen. Im Falle einer Klasse ist dies lediglich eine Basisklasse. Enthält der Parameter `impl` mindestens eine Zeichenkette, wird dem Dateikopf noch das Schlüsselwort `implements` angehängt, gefolgt von den Zeichenketten in dem Parameter `impl`. Diese Zeichenketten repräsentieren im Falle einer zu generierenden Klasse Schnittstellen, welche diese implementiert. Die Schnittstelle, bzw. die Klasse wird nicht mit Methodendeklarationen oder Implementierungen gefüllt. Die Methode prüft nicht, ob die ihr übergebenen Werte korrekter Java-Syntax unterliegen. Wenn die Datei angelegt werden konnte, liefert die Methode `true` zurück, sonst `false`.

  Folgendes Beispiel zeigt die generierte Klasse `Class3` aus dem Paket

  `de.test.eclipseprototype`

  aus dem oben beschriebenen Beispielszenario aus Abschnitt B.2 auf Seite 115:

  ```
  package de.test.eclipseprototype;

  import Class1;

  public class Class3 extends
  ```
  ```

```
    Class1
{ }
```

- `public static boolean createPackage (IJavaProject project, String packageName)`

  Diese Methode erstellt eine neues Java-Paket in einem Eclipse-Projekt. Dazu wird der Methode das Elicpse-Projekt in Form eines Objektes, welche die von Eclipse definierte Schnittstelle `IJavaProject` implementiert, übergeben. In der Methode wird geprüft, ob das Paket bereits existiert. Ist dies der Fall liefert die Methode `false` zurück. Ansonsten wird in der Methode das Paket erstellt. Dazu wird über das Projekt-Objekt ein Objekt, welches den Quellcode-Ordner repräsentiert, ermittelt und darüber das Paket angelegt.

A.15 Die Klasse `EclipseProjectRealizer`

Im Folgenden werden die Methoden der Klasse `EclipseProjectRealizer` erläutert:

- `private void realizePackages()`

  Die Methode ermittelt aus jedem in der `Map packagedef` abgelegten `PackageComponent` den Namen des Paketes und erzeugt das Paket über die statische Methode `createPackage` der Klasse `Util`, die im Abschnitt 7.2.7 auf Seite 73 bei den Methoden der Klasse beschrieben ist.

- `private void realizeInterfaces()`

  Die Methode ermittelt alle in der `Map interfacedef` abgelegten `InterfaceComponent`-Objekte. Aus diesen Objekten wird der zugehörige Paketname der repräsentierten Schnittstelle ermittelt. Aus der Relation `interfaces` werden alle Schnittstellen in Form von Objekten vom Typ `InterfaceComponent` ermittelt, welche diese Schnittstelle erweitert, also zu welchen eine `extends`-Beziehung besteht. Die Objekte werden durch einen Aufruf der Methode `getRelatedColumnObjects` der Klasse `Relation` ermittelt. Über die Objekte wird der Paketname zu diesen Schnittstellen bestimmt. Mit Hilfe der Paketnamen und der erweiterten Schnittstellen werden entsprechende Bezeichner aus Paketnamen gefolgt von einem Punkt und dem Schnittstellenname für die zu generierenden Java-`Import`-Anweisungen gebildet. Mit den ermittelten Informationen wird die statische Methode `createInterface` der Klasse `Util` aufgerufen. Um zu einem Objekt des Typen `InterfaceComponent` den Paketnamen zu ermitteln, wird die nachfolgend noch erläuterte Methode `getPackageFromInterface` aufgerufen.

- `private void realizeClasses()`

  Die Methode ermittelt alle in der `Map classdef` abgelegten `ClassComponent`-Objekte. Aus diesen Objekten wird der zugehörige Paketname der repräsentierten Schnittstelle ermittelt. Aus der Relation `classes` werden über die Methode `getRelatedColumnObjects` der Klasse `Relation` alle Klassen in Form von Objekten vom Typ `ClassComponent` ermittelt, welche diese Klasse erweitert, also zu welchen eine `extends`-Beziehung besteht.

Über die Objekte wird der Paketname zu diesen Klassen bestimmt. Mit Hilfe der Paketnamen und der erweiterten Schnittstellen werden entsprechende Bezeichner aus Paketnamen, gefolgt von einem Punkt und dem Schnittstellenname für die zu generierenden Java-`Import`-Anweisungen gebildet. Genauso wird mit den von der Klasse implementierten Schnittstellen verfahren, welche über die Relation `classesInt` ermittelt werden. Mit den ermittelten Informationen wird die statische Methode `createClass` der Klasse `Util` aufgerufen. Um zu einem Objekt des Typen `ClassComponent` den Paketnamen zu ermitteln, wird die nachfolgend noch erläuterte Methode `getPackageFromClass` aufgerufen, für Objekte vom Typen `InterfaceComponent` die Methode `getPackageFromInterface`.

- private String getPackageFromClass (ClassComponent cc) throws RelationException

Diese Methode ermittelt aus der Relation `packagesClasses` durch einen Aufruf der Methode `getRelatedColumnObjects` der Klasse `Relation` das zu der von dem Parameter `cc` repräsentierten Klasse gehörige Paket.

- private String getPackageFromInterface (InterfaceComponent ic) throws RelationException

Diese Methode ermittelt aus der Relation `packagesInt` durch einen Aufruf der Methode `getRelatedColumnObjects` der Klasse `Relation` das zu der von dem Parameter `ic` repräsentierten Schnittstelle gehörige Paket.

Anhang B

XML-Schema und -Beispiel für Szenenbeschreibungen

B.1 Das Schema für den Prototypen

Es folgt das Schema, welches Regeln für die Struktur der XML-Szenendatei für den Prototypen angibt. Pro möglichem Element in der Szenen-XML-Struktur gibt es hier ein Element mit dem (XML-Namensraum-) Präfix `xs:`, gefolgt von dem Elementnamen. Darin ist festgehalten, wie die Struktur dieses Elementes aufgebaut ist, und wie oft es in der XML-Szenendatei vorkommen darf. Die Regeln selbst sind im Abschnitt 7.2.1 auf Seite 62 erläutert.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
    <xs:element name="classes">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="classdef"
                            minOccurs="0"
                            maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="classdef">
        <xs:complexType>
            <xs:attribute name="id" type="xs:ID"
                          use="required"/>
            <xs:attribute name="name" type="xs:string"
                          use="required"/>
        </xs:complexType>
    </xs:element>
```

```
<xs:element name="components">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="packages"/>
            <xs:element ref="classes"/>
            <xs:element ref="interfaces"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="conetree">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="vertices"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:string"
                      use="required"/>
        <xs:attribute name="ref-id" type="xs:IDREF"
                      use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="conetrees">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="conetree"
                        minOccurs="0"
                        maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="content">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="refclass"
                        minOccurs="0"
                        maxOccurs="unbounded"/>
            <xs:element ref="refinterface"
                        minOccurs="0"
                        maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="cube">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="content"/>
            <xs:element ref="subcubes"/>
        </xs:sequence>
```

```
        <xs:attribute name="id" type="xs:ID"
                      use="required"/>
        <xs:attribute name="ref-id" type="xs:IDREF"
                      use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="cubes">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="cube"
                        minOccurs="0"
                        maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="extends">
    <xs:complexType>
        <xs:attribute name="ref-id" type="xs:IDREF"
                      use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="interfacedef">
    <xs:complexType>
        <xs:attribute name="id" type="xs:ID"
                      use="required"/>
        <xs:attribute name="name" type="xs:string"
                      use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="interfaces">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="interfacedef"
                        minOccurs="0"
                        maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="packagedef">
    <xs:complexType>
        <xs:attribute name="id" type="xs:ID"
                      use="required"/>
        <xs:attribute name="name" type="xs:string"
                      use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="packages">
```

```
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="packagedef"
                        minOccurs="0"
                        maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="refclass">
    <xs:complexType>
        <xs:attribute name="ref-id" type="xs:IDREF"
                      use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="refcube">
    <xs:complexType>
        <xs:attribute name="ref-id" type="xs:IDREF"
                      use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="refinterface">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="extends"
                        minOccurs="0"
                        maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="ref-id" type="xs:IDREF"
                      use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="root">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="config">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="nativelibrary">
                            <xs:complexType>
                                <xs:attribute name="uri"
                                              type="xs:anyURI"
                                              use="required"/>
                            </xs:complexType>
                        </xs:element>
                        <xs:element name="nativelogfile">
                            <xs:complexType>
                                <xs:attribute name="uri"
```

```
                                                type="xs:anyURI"
                                                use="required"/>
                                    </xs:complexType>
                                </xs:element>
                                <xs:element name="logfile">
                                    <xs:complexType>
                                        <xs:attribute name="uri"
                                                type="xs:anyURI"
                                                use="required"/>
                                    </xs:complexType>
                                </xs:element>
                            </xs:sequence>
                        </xs:complexType>
                    </xs:element>
                    <xs:element ref="components"/>
                    <xs:element ref="visualize"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="subcubes">
            <xs:complexType>
                <xs:sequence>
                    <xs:element ref="refcube"
                                minOccurs="0"
                                maxOccurs="unbounded"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="vertices">
            <xs:complexType>
                <xs:sequence>
                    <xs:element ref="refclass"
                                minOccurs="0"
                                maxOccurs="unbounded"/>
                    <xs:element ref="refinterface"
                                minOccurs="0"
                                maxOccurs="unbounded"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="visualize">
            <xs:complexType>
                <xs:sequence>
                    <xs:element ref="walls"/>
                    <xs:element ref="conetrees"/>
                    <xs:element ref="cubes"/>
                </xs:sequence>
```

```
        </xs:complexType>
    </xs:element>
    <xs:element name="wall">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="refinterface"
                            minOccurs="0"
                            maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="id" type="xs:string"
                          use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="walls">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="wall"
                            minOccurs="0"
                            maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

B.2 Beispiel XML-Szenendatei

Eine XML-Datei, welche dem genannten Schema unterliegt, ist die Datei `prototype.scene`, welche dem Prototypen beiliegt. Die Datei ist im Folgenden ersichtlich und wird nachfolgend erläutert:

```
<?xml version="1.0" encoding="UTF-8"?> <root
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="prototype.xsd">
       <config>
        <nativelibrary
          uri="kurejava.dll">
        </nativelibrary>
        <nativelogfile uri="kurejava_native.log">
        </nativelogfile>
        <logfile uri="prototype.log"></logfile>
    </config>
    <components>
        <packages>
            <packagedef id="package1" name="de.test.eclipseprototype"/>
            <packagedef id="package2"
              name="de.test.eclipseprototype.subpackage1"/>
```

```
        <packagedef id="package3"
          name="de.test.eclipseprototype.subpackage2"/>
    </packages>
    <classes>
        <classdef id="class1" name="Class1"/>
        <classdef id="class2" name="Class2"/>
        <classdef id="class3" name="Class3"/>
        <classdef id="class4" name="Class4"/>
    </classes>
    <interfaces>
        <interfacedef id="interface1" name="Interface1"/>
        <interfacedef id="interface2" name="Interface2"/>
        <interfacedef id="interface3" name="Interface3"/>
        <interfacedef id="interface4" name="Interface4"/>
    </interfaces>
</components>
<visualize>
    <walls>
        <wall id="wall1">
            <refinterface ref-id="interface1">
                <extends ref-id="interface3"/>
                <extends ref-id="interface4"/>
                <extends ref-id="interface2"/>
            </refinterface>
            <refinterface ref-id="interface2">
                <extends ref-id="interface3"/>
                <extends ref-id="interface4"/>
            </refinterface>
            <refinterface ref-id="interface3">
            </refinterface>
            <refinterface ref-id="interface4">
            </refinterface>
        </wall>
    </walls>
    <conetrees>
        <conetree id="conetree1" ref-id="class1">
            <vertices>
                <refclass ref-id="class3"/>
                <refinterface ref-id="interface3"/>
                <refinterface ref-id="interface4"/>
            </vertices>
        </conetree>
        <conetree id="conetree2" ref-id="class2">
            <vertices>
                <refclass ref-id="class4"/>
                <refinterface ref-id="interface3"/>
                <refinterface ref-id="interface4"/>
```

```
            </vertices>
        </conetree>
    </conetrees>
    <cubes>
        <cube id="cube1" ref-id="package2">
            <content>
                <refinterface ref-id="interface1"/>
            </content>
            <subcubes/>
        </cube>
        <cube id="cube2" ref-id="package1">
            <content>
                <refclass ref-id="class4"/>
                <refclass ref-id="class3"/>
                <refinterface ref-id="interface2"/>
            </content>
            <subcubes>
                <refcube ref-id="cube1"/>
            </subcubes>
        </cube>
    </cubes>
  </visualize>
</root>
```

In der XML-Struktur werden im Element `config` in den entsprechenden Kindelementen die zu nutzende Bibliothek und die Dateien, in denen Ausgaben erfolgen sollen, angegeben. In dem Element `components` werden in den entsprechenden Kindelementen die drei Pakete

`de.test.eclipseprototype`,

`de.test.eclipseprototype.subpackage1` und

`de.test.eclipseprototype.subpackage2`,

die vier Klassen namens `Class1`, `Class2`, `Class3` und `Class4` und die vier Schnittstellen namens `Interface1`, `Interface2`, `Interface3` und `Interface4` angegeben. Allen wird jeweils ein eindeutiger Bezeichner über das Attribut `id` zugeordnet, um im weiteren Verlauf auf sie zu verweisen. In dem Element `visualize` wird im Kindelement `walls` eine `Wall` angegeben, welche alle vier vorher genannten Schnittstellen beinhaltet. Zusätzlich wird in der `Wall` angegeben, dass in ihr `implements`-Beziehungen von `Interface1` mit `Interface2`, `Interface3`, `Interface4` und von `Interface2` mit `Interface3`, `Interface4` bestehen.

Unter dem Element `conetrees` werden zwei `Cone Trees` beschrieben. Der erste hat die Klasse `Class1`, der zweite die Klasse `Class2` als Spitze. Der erste `Cone Tree` hat dann die Klasse `Class3` und die Schnittstellen `Interface3` und `Interface4` als Knoten. Dies visualisiert, dass die Klasse `Class1` die Klasse `Class3` erweitert und die Schnittstellen `Interface3` und `Interface4` implementiert. Der zweite `Cone Tree` hat dann die Klasse `Class4` und die Schnittstellen `Interface3` und `Interface4` als Knoten. Dies stellt dar, dass die Klasse `Class2` die Klasse `Class4` erweitert und die Schnittstellen `Interface3` und `Interface4` implementiert.

Unter dem Element `cubes` werden zwei `Cubes` angegeben. Der erste `Cube` repräsentiert das Paket `de.test.eclipseprototype.subpackage1`, welches über das Attribut `ref-id` angegeben ist und hat als Inhalt die Schnittstelle `Interface1`. Dies bedeutet, dass sich diese Schnittstelle im entsprechenden Paket befindet. Der zweite `Cube` ist dem Paket `de.test.eclipseprototype` zugeordnet und hat als Inhalt die Schnittstelle `Interface2` und die Klassen `Class3` und `Class4`, sowie den kompletten ersten `Cube`. Letzteres verdeutlicht, dass

`de.test.eclipseprototype.subpackage1` Unterpaket von

`de.test.eclipseprototype` ist.

Literaturverzeichnis

[1] K. Alfert and A. Fronk. 3-Dimensional Visualization of Java Class Relations. In M. M. Tanik and A. Ertas, editors, *Proceedings of the 5th World Conference on Integrated Design Process Technology*. Society for Design and Process Science, 2000. on CD-ROM.

[2] K. Alfert and A. Fronk. Manipulation of 3-Dimensional Visualization of Java Class Relations. In M. M. Tanik and A. Ertas, editors, *Proceedings of the 6th World Conference on Integrated Design Process Technology*. Society for Design and Process Science, 2002.

[3] K. Alfert, A. Fronk, and F. Engelen. Experiences in 3-dimensional visualization of Java class relations. *Transactions of the SDPS: Journal of Integrated Design and Process Science*, 5(3):91–106, September 2001.

[4] R. Behnke. Extending relational specifications by sequential algebras - prototyping with relview. In: Berghammer R., Simon F. (eds.): Programming languages and fundamentals of programming. Report 9717, Institut für Informatik und Praktische Mathematik, Universität Kiel, 12-22, 1997.

[5] R. Behnke, R. Berghammer, E. Meyer, and P. Schneider. RELVIEW - A System for Calculating with Relations and Relational Programming.
`http://www.informatik.uni-kiel.de/~progsys/relview.html/`.
Zuletzt zugegriffen am 09.11.2003.

[6] R. Behnke, R. Berghammer, and P. Schneider. Machine Support of Relational Computations: The Kiel RELVIEW System.
`http://www.informatik.uni-kiel.de/~progsys/relview.html/`.
Zuletzt zugegriffen am 09.11.2003.

[7] R. Berghammer, B. von Karger, and C. Ulke. Relation-algebraic analysis of petri nets with relview. In: Margaria T., Steffen B. (eds.): Proc. TACAS '96, LNCS 1055, Springer, 49-69, 1996.

[8] Rudolf Berghammer and Alexander Fronk. Applying relational algebra in 3d graphical software design. In *Proceedings of the 7th International Seminar on Relational Methods in Computer Science (RelMiCS 7)*, 2003. To appear.

[9] C. Brink, W. Kahl, and G. Schmidt. Relational methods in computer science. New York: Springer, 1997.

[10] Eclipse.org Consortium. eclipse.org.
`http://www.eclipse.org/`.
Zuletzt zugegriffen am 09.11.2003.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, 1995.

[12] R. Helm and K. Marriott. A Declarative Specification and Semantics for Visual Languages. *Journal of Visual Languages and Computing*, 1(2):311–331, 1991.

[13] U. Milanese. KURE: Kiel University Relation Package Release 1.0.
`http://cvs.informatik.uni-kiel.de/~kure/`.
Zuletzt zugegriffen am 09.11.2003.

[14] U. Milanese. Rechnergestützte Programmentwicklung - RelView System.
`http://http://www.informatik.uni-kiel.de/~progsys/relview.shtml`.
Zuletzt zugegriffen am 09.11.2003.

[15] Inc. Neva Object Technology. Java2COM.
`http://www.nevaobject.com/_docs/_java2com/java2com.html/`.
Zuletzt zugegriffen am 09.11.2003.

[16] G. Schmidt and T. Ströhlein. Relations and graphs. Discrete Mathematics for Computer Scientists, EATCS Monographs on Theoret. Comput. Sci., Springer, 1993.

[17] F. Somemzi. CUDD: CU Decision Diagram Package, Release 2.3.1.
`http://vlsi.colorado.edu/~fabio/CUDD/`.
Zuletzt zugegriffen am 09.11.2003.

[18] SUN. Java Technology and XML Downloads - Java Architecture for XML Binding.
`http://java.sun.com/xml/downloads/jaxb.html/`.
Zuletzt zugegriffen am 09.11.2003.

[19] Inc. Sun Microsystems. JavaBeans Bridge for ActiveX.
`http://java.sun.com/products/javabeans/software/bridge/`.
Zuletzt zugegriffen am 09.11.2003.