Lehrstuhl für Software-Technologie, Fachbereich Informatik, Universität Dortmund

Diplomarbeit

Modifikation von dreidimensionalen Visualisierungen objektorientierter Software-Strukturen

Jin-Ha Tchoe

8. September 2004

Erstgutachter:
Prof. Dr. E.-E. Doberkat

Zweitgutachter:
Dr. Alexander Fronk

Preliminary version – 8. September 2004

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Java Quellcode Verzeichnis

Teil I.

Einleitung

1. Einführung in die Thematik

Software wird oft mit Hilfe grafischer Sprachen visualisiert, durch die der Entwurf prägnanter als mit textuellen Beschreibungen veranschaulicht werden kann. Häufig beruhen diese Sprachen auf zweidimensionalen Diagrammen, was die Wahl zulässt, sie mit Papier und Bleistift oder am Computer zu erstellen. Als Standard dient dabei die *Unified Modeling Language* (UML, [Uml]), die beispielsweise Klassen-, Sequenz- und Use-Case-Diagramme bereitstellt, um verschiedene Aspekte der Software wie Struktur, Dynamik und Kommunikation mit dem Benutzer zu beschreiben.

In [AF00] und [AFE01] wird eine dreidimensionalen Darstellung als alternativer Ansatz vorgestellt, die mit dem J3Browser (s. Kapitel 1.2) prototypisch umgesetzt worden ist. Dieser Ansatz visualisiert Java-Quellcode in Form von Klassen, Schnittstellen und Paketen mit ihren statischen Beziehungen Assoziation, Implementierung, Vererbung und Benutzung. Dabei wird für die Darstellung nicht einfach eine zweidimensionale Notation wie die UML ins Dreidimensionale übertragen. Vielmehr sollen die Vorteile der dritten Dimension effektiv ausgenutzt werden. Diese sind laut [AF00] und [AFE01]:

- *Transparenz von Objekten*, um Inhaltsbeziehungen darzustellen, wie beispielsweise die in einem Paket enthaltenen Klassen.
- *Räumliche Tiefe*, um Objekte hintereinander anzuordnen.
- *Räumliche Anordnung*, bei der man sich wegen einer perspektivischen Anordnung auf Objekte im Vordergrund konzentriert, aber immer noch Objekte im Hintergrund im Blick hat.
- *Beweglichkeit*, mit deren Hilfe der Benutzer durch das Diagramm navigieren kann, als ob er ein Teil dessen wäre.

Dreidimensionale Anordnungsmethoden, die die sichtbaren grafischen Objekte im virtuellen Raum nach bestimmten Regeln positionieren, sollen die Realisierung der genannten Vorteile sicherstellen. In [AF00] und [AFE01] werden daher folgende Anordnungsmethoden betrachtet:

- *Cone Trees* stellen Objekte baumartig dar. Jeder Teilbaum bildet einen Kegel, in dem die Wurzel des Teilbaumes die Spitze und die Kinder die kreisförmige Basis bilden. Durch rekursive Nutzung können *Cone Trees* auf diese Weise Vererbungshierarchien visualisieren.

- *Information Cubes* gruppieren verwandte Informationen in transparenten Würfeln, die beliebig ineinander verschachtelt werden können. Dabei muss stets ein Würfel vollständig in einem anderen enthalten sein. Mit *Information Cubes* kann die Paketzugehörigkeit von Klassen dargestellt werden.
- Informationslandschaften nutzen die Metapher der Landschaft. Objekte werden auf einer Ebene, dem „Boden" der Darstellung, platziert, so dass der Eindruck von Gebäuden in einer Landschaft entsteht. Linien, die zwischen den Objekten verlaufen, symbolisieren deren Beziehungen und entsprechen in der Metapher Straßen, die von einem Gebäude zu einem anderen führen.
- *Walls* ordnen Objekte in einer zweidimensionalen Ebene an. Durch die räumliche Anordnung können Teile der *Wall* im Vordergrund dargestellt werden, während der Rest kleiner im Hintergrund sichtbar ist. *Walls* eignen sich dazu, Schnittstellenhierarchien darzustellen.

1.1. Modifikation

Im Laufe der Softwareentwicklung können Anpassungen aus vielfältigen Gründen an neue oder veränderte Anforderungen nötig werden, die unter Umständen auch eine Veränderung der Klassen bzw. deren Struktur erfordern. Zum Beispiel könnte eine Klasse überflüssig werden und entfernt werden müssen. In diesem Fall wäre es wünschenswert, den Code der Klasse zu entfernen, indem wie bei zweidimensionalen computergestützten Entwicklungswerkzeugen seine grafische Repräsentation in der Visualisierung ausgewählt und gelöscht wird.

Die Möglichkeit, die Visualisierung zu modifizieren, kann allerdings problematisch sein. So muss sichergestellt werden, dass die Visualisierung stets korrekt die Strukturen der benutzten Programmiersprache darstellt, um aus ihr syntaktisch korrekten Quellcode zu erhalten. Wie in [AF00] und [AFE01] wird im weiteren Java als benutzte Programmiersprache betrachtet.

Aus der visuellen Modifikation muss anschließend die resultierende Veränderung des Quellcodes berechnet und durchgeführt werden. Dabei müssen eventuelle Abhängigkeiten beachtet werden.

1.1.1. Gültigkeit der Visualisierung

Um sicherzustellen, dass die Visualisierung immer korrekt die Strukturen von Java-Klassen repräsentiert, muss die Syntax der Darstellung definiert werden. Es muss bestimmt werden, welche Elemente der Visualisierung welchen Teilen des Quellcodes entsprechen, und wie diese Elemente zusammengesetzt bzw.

verbunden werden dürfen. Nur wenn die modifizierte Visualisierung immer noch diese Regeln erfüllt, kann die Modifikation als gültig anerkannt werden.

In [BF03] werden zu diesem Zweck relationenalgebraische Formeln vorgeschlagen. In ihnen wird beispielsweise definiert, dass ein Würfel eine Klasse und ein *Information Cube* ein Paket repräsentiert. Analog zu der Bedingung, dass eine Klasse immer in genau einem Paket sein muss, muss dann gelten, dass ein Würfel immer in genau einem *Information Cube* liegt. Weitere Formeln verknüpfen anschließend Visualisierung und Quellcode miteinander. Zum Beispiel muss die Klasse, die durch einen bestimmten Würfel dargestellt wird, zu dem Paket gehören, das durch den umgebenden *Information Cube* vertreten wird. Mit Hilfe von KURE bzw. RELVIEW ([KUR]) können diese Formeln effizient auf Validität geprüft werden, das im Rahmen einer Diplomarbeit ([Szy03]) als Java-Bibliothek zur Verfügung steht.

1.1.2. Änderung des Quellcodes

Ist die Gültigkeit der visuellen Modifikation gewährleistet, muss die daraus resultierende Änderung des Quellcodes bestimmt werden. Diese beschränkt sich nicht nur auf das betrachtete Element, sondern kann den gesamten Quellcode betreffen. Soll beispielsweise eine Klasse aus einem Paket entfernt werden, genügt es nicht, nur den Quellcode der Klasse zu löschen. Vielmehr muss ebenfalls jede Referenz, also in unserem Fall jede Assoziations-, Vererbungs-, Implementierungs- (falls eine Schnittstelle entfernt wird) und Benutzt-Beziehung zu dem nicht mehr existierendem Element, eliminiert werden.

Sind die Klassen über den Coderahmen hinaus teilweise oder vollständig implementiert, so entstehen weitere Folgeprobleme, da ebenfalls alle Aufrufe von Methoden und Referenzen auf Variablen des entfernten Elementes beseitigt werden müssen. Um die syntaktische Korrektheit zu gewähren, muss festgestellt werden, ob die verwendeten Aufrufe und Referenzen für weitere Anweisungen im Quellcode relevant sind, zum Beispiel indem die Rückgabe einer Methode ausgewertet wird oder der Wert einer Referenz einer lokalen Variable zugewiesen wird. Wird zum Beispiel eine Klasse `B` gelöscht, so würde von der Anweisung `int i = instanzVonB.berechne()` nur die Deklaration von `i` bleiben. Zur Erhaltung der syntaktischen Korrektheit müsste dieser Variable einfach ein beliebiger Wert zugewiesen werden. Dies würde allerdings nicht garantieren, dass die Implementierung weiterhin den ihr bestimmten Zweck erfüllt.

Unter Umständen kann die syntaktische Korrektheit aber gar nicht gewährleistet werden. Soll beispielsweise eine Klasse `B` von einer anderen Klasse `A` erben, so muss `B` in seinen Konstruktoren ebenfalls den der Superklasse aufrufen. Besitzt `A` aber weder den leeren noch einen anderen Konstruktor, der mit den Konstruktorparametern von `B` aufgerufen werden kann, kann kein `super`-Aufruf konstruiert werden. Hier muss der Benutzer dann selbst den Quellcode anpassen.

1.2. Der J3Browser

In [Eng00] wird der in [AF00] und [AFE01] vorgestellte Ansatz der dreidimensionalen Vorstellung prototypisch in Form des J3Browsers realisiert. Er analysiert Java-Quellcode mit Hilfe von Doclets ([Jav]) und stellt die so gesammelten Informationen mit Hilfe von VRML ([Vrm]) dar.

Der J3Browser nutzt eine modifizierte Informationslandschaft zur Darstellung der Paketstruktur, in der die sichtbaren Objekte nicht an den Boden gebunden sind, sondern frei im Raum schweben. In der Darstellung werden Pakete als *Information Cubes* visualisiert, welche Würfel für Klassen sowie Kugeln für Schnittstellen enthalten. Zusätzlich werden die statischen Beziehungen Assoziation, Implementierung, Benutzt und Vererbung in Form verschiedenfarbiger Röhren bzw. Pfeile integriert. In getrennten Visualisierungen können weiterhin Vererbungs- und Implementierungshierarchien durch *Cone Trees* für Klassen bzw. *Walls* für Schnittstellen beschrieben werden.

Der J3Browser ist ein reines Betrachtungswerkzeug und bietet somit keine Möglichkeit zur Modifikation. Wird während der Betrachtung deutlich, dass der Entwurf geändert werden soll, indem beispielsweise eine Klasse gelöscht werden soll, muss der Benutzer diese selbst im Quellcode entfernen und anschließend den J3Browser neu starten. Wünschenswert wäre jedoch, wie in Kapitel 1.1 beschrieben, Modifikationen direkt in der Visualisierung auszuführen.

Bedingt durch den Aufbau und der benutzten Komponenten ist ein nachträglicher Einbau von Modifikationsmöglichkeiten nicht realisierbar. Der J3Browser trennt die Analyse des Quellcodes durch Doclets von der Darstellung mit Hilfe eines VRML-Betrachters in einem Webbrowser. Um durch eine Modifikation der Visualisierung auch den Quellcode modifizieren zu können, muss die Darstellung aber Bestandteil des Programms sein, um so die Veränderungen in der Visualisierung zu empfangen, zu prüfen und eventuell den Quellcode zu ändern. VRML ist jedoch nicht in ein Java-Programm integrierbar, so dass eine andere Visualisierungstechnik gefunden werden muss.

Ebenfalls muss ein Ersatz für die Komponente zur Analyse des Quellcodes gefunden werden. Da Doclets dazu konzipiert sind, Klassendokumentationen anhand von JavaDoc-Kommentaren zu erstellen, können sie nur lesend auf den Quellcode zugreifen und somit nicht zur Modifikation benutzt werden.

2. Ziel und Vorgehen

2.1. Ziel

Ziel der Diplomarbeit ist, aufbauend auf der Grundidee des J3Browsers die Möglichkeit zu schaffen, durch Modifikationen in der Visualisierung den Quellcode zu ändern. Dieses soll zusätzlich in einem prototypischen Programm umgesetzt werden.

Aufgrund der in Kapitel 1.2 beschriebenen Beschränkungen des J3Browsers, kann dieser nicht einfach erweitert werden. Vielmehr muss das Programm, das in Anlehnung an den J3Browser J3Creator genannt wird, komplett neu entwickelt werden. Dabei soll der Schwerpunkt auf der Realisierung der Modifikationsmöglichkeiten liegen, wohingegen darstellerische Aspekte wie ergonomisches Layout nur eine minimale Rolle spielen sollen.

VRML wird als Visualisierungstechnik durch Java3D ([J3D]) ersetzt. Als Erweiterungs-API ([Api]) ist Java3D per Instanziierung und Methodenaufruf in ein Java-Programm integrierbar und stellt Mechanismen zur Verfügung, durch die das Programm über Ereignisse und Veränderungen in der Visualisierung benachrichtigt werden kann. Auf diese Weise kann der J3Creator auf Modifikationen in der Visualisierung reagieren und den Quellcode dementsprechend verändern.

Der J3Creator wird als PlugIn für die Entwicklungsumgebung ECLIPSE ([Ecl]) realisiert. Ein PlugIn stellt eine Erweiterungskomponente dar, die die Funktionalität erweitert und für diesen Zweck auf Schnittstellen von ECLIPSE zurückgreifen kann, die APIs genannt werden. Insbesondere wird von ECLIPSE mit den *Java Development Tools* ([Jdt]) eine API geboten, mit dessen Hilfe Java-Quellcode in Form von Datenstrukturen und abstrakten Syntaxbäumen analysiert und modifiziert werden kann. Dadurch entfällt für den J3Creator die Aufgabe, selbst die Quellcodedateien zu laden und zu analysieren. Veränderungen des Quellcodes können mit Hilfe des JDT auf den Datenstrukturen und Syntaxbäumen durchgeführt werden, anstatt selbst in den Quellcodedateien die Textabschnitte zu suchen und zu ersetzen.

Zur Überprüfung der Modifikationen werden die in Abschnitt 1.1.1 angesprochenen relationenalgebraischen Formeln benutzt, die mit Hilfe der Java-Bibliothek KURE ([Szy03]) zur Verfügung stehen.

2.2. Vorgehen

Das Vorgehen teilt sich grob in zwei Abschnitte: die technischen Voraussetzungen und die Modifikation. Da der J3Creator komplett neu entwickelt wird, müssen zunächst die Analyse des Quellcodes und die Darstellung der erfassten Information umgesetzt werden, bevor die Modifikation behandelt werden kann.

Die Modifikation kann wieder in drei Teile gegliedert werden: Navigation, einfache und semantikverändernde Modifikationsmöglichkeiten. Die Navigation modifiziert die Sicht des Benutzers auf die Darstellung und behandelt die Interaktion des Benutzers mit dem J3Creator. Durch einfache Modifikationsmöglichkeiten werden anschließend die sichtbaren Objekte modifiziert, ohne jedoch Auswirkungen auf den Quellcode zu haben. Diese entstehen erst mit den semantikverändernden Modifikationsmöglichkeiten.

Zusammenfassend ergeben sich folgende Arbeitsschritte:

1. Analyse des Quellcodes mit ECLIPSE
2. Visualisierung mit Hilfe von Java3D
3. Realisierung einer einfachen Navigation in der Visualisierung
4. Realisierung einfacher Modifikationsmöglichkeiten
5. Realisierung semantikverändernder Modifikationsmöglichkeiten

Im Folgenden werden die einzelnen Arbeitsschritte näher beschrieben und motiviert.

2.2.1. Analyse des Quellcodes mit ECLIPSE

Das ECLIPSE *-Framework* ([Jdt]) ermöglicht einem PlugIn, Quellcode zu analysieren, ohne selbst auf die Dateien direkt zuzugreifen und den Quellcode parsen zu müssen, sondern bildet die Java-Strukturen in Form von hierarchisch angeordneten Datenstrukturen und abstrakten Syntaxbäumen ab.

In Kapitel 3 wird durch eine Analyse des ECLIPSE *-Frameworks* ein Konzept erarbeitet und realisiert, mit dem alle zur Visualisierung nötigen Informationen gesammelt werden können. Dazu wird zuerst eine Übersicht über die für die Analyse des Quellcodes relevanten Teile des *Frameworks* gegeben. Anschließend wird kurz dargestellt, auf welche Weise die Informationen, die für die Darstellung nötig sind, mit Hilfe des ECLIPSE *-Frameworks* erfasst werden. Zum Schluss werden als Ausblick auf den letzten Arbeitsschritt die Möglichkeiten zur Modifikation des Quellcodes mit Hilfe des *Frameworks* vorgestellt.

2.2.2. Visualisierung mit Java3D

Nach der Analyse des Quellcodes wird in Kapitel 4 die Darstellung mit Hilfe von Java3D realisiert. Java3D basiert auf einem Szenegraphen, der von der eigentlichen Darstellung am Bildschirm in Form eines hierarchischen Baumes abstrahiert. Es wird ein Konzept vorgestellt, das den Aufbau des Szenegraphen soweit möglich vom J3Creator versteckt, indem dieser sich auf Instanzierung von Objekten und Aufruf von Delegationsmethoden beschränkt.

Die Darstellung lehnt sich größtenteils an der Informationslandschaft des J3Browsers (s. Kapitel 1.2) an. Als grafische Repräsentationen werden dementsprechend genutzt:

- für Pakete: *Information Cubes*
- für Klassen: Würfel
- für Schnittstellen: Kugeln

Die visualisierten Beziehungen werden ebenfalls übernommen. Diese sind im Einzelnen:

- Assoziation
- Implementierung
- Vererbung
- Benutzung

Neu hinzu kommt die Subpaketbeziehung, da anders als im J3Browser *Information Cubes* nicht ineinander verschachtelt werden, sondern lediglich Würfel und Kugeln enthalten.

Im J3Creator wird in der Darstellung auf Details, die im J3Browser angezeigt wurden, verzichtet. So wird bespielsweise die Markierung der Elemente (s. [Eng00] Anhang D), die beispielsweise die Sichtbarkeit von Java-Elementen kennzeichnet, nicht beachtet. Die Vererbungs- und Implementierungshierarchien des J3Browsers in Form von *Cone Trees* und *Walls* werden nicht umgesetzt.

2.2.3. Realisierung der Navigation

Beweglichkeit ist, wie in der Einführung (s. Kapitel 1) geschildert, ein wesentlicher Vorteil der dreidimensionalen Darstellung. Java3D stellt diese Funktionalität allerdings standardmäßig nicht zur Verfügung. Deswegen gilt es in Kapitel 5 die Navigation nachzubilden. Der Benutzer soll in der Lage sein, sich frei im virtuellem Raum zu bewegen und die gesamte Visualisierung zu drehen, um sich auf diese Weise auf die für ihn interessanten Teile konzentrieren zu können.

2.2.4. Realisierung einfacher Modifikationsmöglichkeiten

Dieser Arbeitsschritt ist als Vorbereitung zum letzten Schritt (s. 2.2.6) gedacht, indem in Kapitel 6 Modifikationsmöglichkeiten geboten werden, die keine Änderung des Quellcodes zur Folge haben, sondern stattdessen nur den Szenegraphen betreffen. Der Benutzer erhält mit der Realisierung die Möglichkeit, die Anordnung der Elemente in der Visualisierung durch Verschieben zu ändern, sofern diese die Semantik erhält. Zusätzlich kann er die Farbe einzelner Elemente zur Markierung oder Gruppierung modifizieren, solange die gewünschte Farbe nicht semantisch belegt ist.

2.2.5. Überprüfung der Modifikation

Wie in Abschnitt 1.1 erläutert, kann eine Modifikation in der Darstellung dazu führen, dass sie nicht mehr korrekt die Klassenstrukturen von Java-Quellcode visualisiert. Aus diesem Grund wird in Kapitel 7 mit Hilfe von relationenalgebraischen Formeln und KURE eine Überprüfung der Visualisierung realisiert, mit der ebenfalls die Gültigkeit einer Modifikation entschieden werden kann. Dazu wird zuerst definiert, welche grafischen Elemente existieren und welche Java-Elemente sie repräsentieren. Für diese grafischen Elemente wird anschließend festgelegt, wie sie visuell in Beziehung stehen dürfen und müssen. Auf diese Weise wird die Syntax der Visualisierung formuliert.

2.2.6. Realisierung semantikverändernder Modifikationsmöglichkeiten

Nach der gezielten Modifikation des Szenegraphen in Abschnitt 2.2.6 und der Überprüfung der Modifikation, wird in Kapitel 8 durch eine Modifikation in der Darstellung gezielt der Quellcode verändert. Die möglichen und grundlegensten Operationen umfassen:

- Hinzufügen von Elementen
- Verschieben von Elementen
- Umbenennen von Elementen
- Entfernen von Elementen

Für jedes Element und für jede Operation wird analysiert, auf welche Weise und an welchen Stellen der Quellcode verändert werden muss. Auf diese Weise ergibt sich für jede Kombination aus Elementen und Operationen ein Plan, der beschreibt in welche Pakete, Klassen und Schnittstellen betrachtet und wie sie bearbeitet werden müssen. Unter Berücksichtigung der in Kapitel 1.1.2 beschriebenen Problematik werden anschließend die Pläne mit Hilfe des ECLIPSE -*Frameworks* umgesetzt.

Teil II.

Technische Voraussetzungen

3. Analyse des Quellcodes mit ECLIPSE

Dieses Kapitel behandelt die erste grundlegende Komponente des J3Creators: die Analyse von Java-Quellcode. Mit ihr werden die zu visualisierenden Strukturen und Beziehungen im betrachteten Quellcode analysiert und gesammelt. Zu diesem Zweck wird der J3Creator als PlugIn für ECLIPSE realisiert, wodurch er in der Lage ist, auf andere PlugIns aufzubauen, die ihn bei der Analyse untersützen können.

Die *Java Development Tools* (JDT, [Jdt]) sind ein solches PlugIn für ECLIPSE und stellen eine API ([Api]) zur Verfügung, die einem PlugIn die Möglichkeit bietet, Java-Quellcode zu analysieren ohne mit seiner textuellen Form oder dem Dateisystem in Berührung zu kommen. Mit ihrer Hilfe werden die Informationen gesammelt, die benötigt werden, um die Visualisierung in Kapitel 4 zu konstruieren. In Anlehnung an Abschnitt 2.2.2 werden Pakete, Klassen und Schnittstellen im Quellcode identifiziert mit ihren Beziehungen:

- Assoziation
- Implementierung
- Vererbung
- Benutzung
- Subpaketbeziehung

Die JDT abstrahieren vom Quellcode in Form spezieller Datenstrukturen, die beispielsweise ein Paket repräsentieren. Auf diese Weise kann im Falle eines Pakets der Name und die Liste der enthaltenen Klassen und Schnittstellen abgefragt werden. Zusätzlich kann die Implementierung einer Klasse mit Hilfe eines abstrakten Syntaxbaums (AST) analysiert werden. Kapitel 3.1 gibt eine Übersicht über die Datenstrukturen der JDT, mit denen der Quellcode analysiert werden kann.

Anschließend wird in Kapitel 3.2 beschrieben, wie der J3Creator den Quellcode mit Hilfe der JDT untersucht. Kapitel 3.3 stellt als Ausblick auf die semantikverändernden Modifikationen in Kapitel 8 die Möglichkeiten der JDT dar, den Quellcode zu modifizieren.

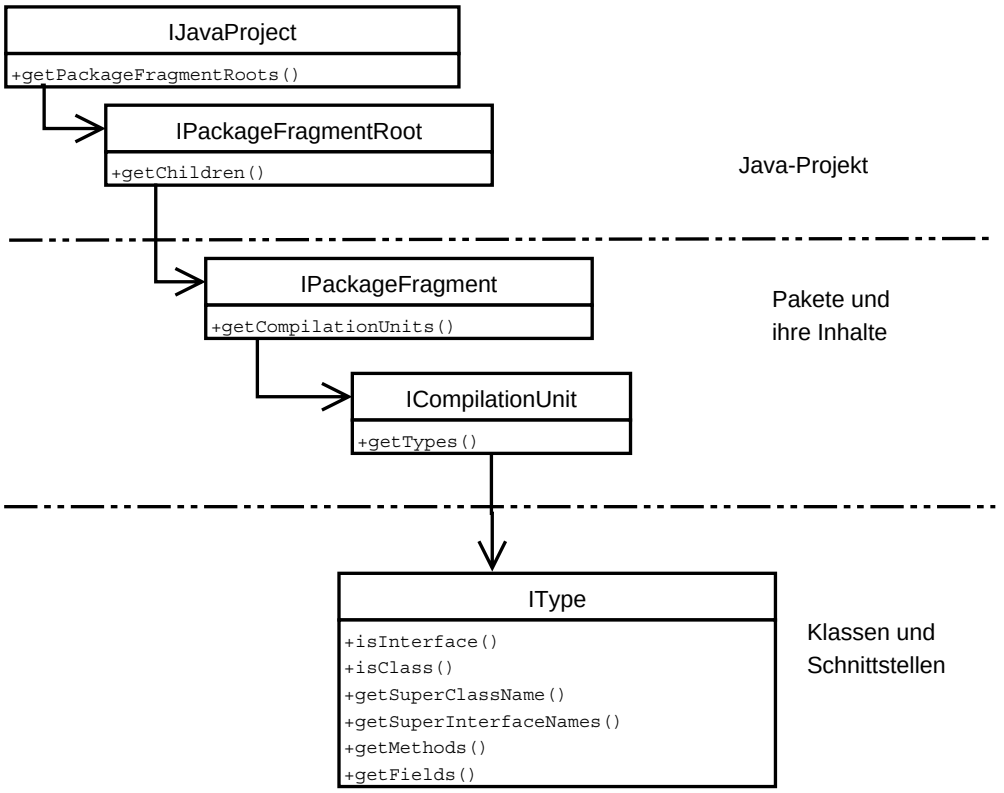
3.1. Übersicht über die JDT

Die für die Quellcodeanalyse wichtigen Klassen der JDT-API sind in zwei Pakete unterteilt: das *Core*- und das *DOM*-Paket. Das *Core*-Paket stellt Möglichkeiten zur Verfügung, die Paketstruktur des Quellcodes zu untersuchen. Für die in den Paketen enthaltenen Klassen und Schnittstellen kann weiterhin die Signatur analysiert werden. Durch das *DOM*-Paket können mit Hilfe eines abstrakten Syntaxbaumes (AST) zu jeder Klasse ihre Implementierungsdetails wie Methodenaufrufe, Variablenzuweisungen oder *try/catch*-Konstrukte analysiert werden.

3.1.1. Übersicht über das *Core*-Paket

Mit Hilfe des *Core*-Pakets kann ein PlugIn die Hierarchie von Paketen und die Signatur von Klassen und Schnittstellen untersuchen. Abbildung 3.1 gibt eine schematische Übersicht über die Klassen, die zur Analyse benötigt werden.

Abbildung 3.1. Übersicht über die Klassen des *Core*-Pakets der JDT-API



Die Klassen können wie in der Abbildung dargestellt in drei Teile gegliedert werden. Der erste Teil beschreibt die Elemente eines Java-Projekts, das in ECLIPSE alle Ressourcen für ein Java-Programm zusammenfasst. Der zweite Teil

repräsentiert die Paketstruktur, während durch den dritten Teil Klassen und Schnittstellen dargestellt werden. Im Folgenden werden die einzelnen Klassen näher beschrieben.

Ein *IJavaProject* repräsentiert ein Java-Projekt und gruppiert in ECLIPSE den Quellcode und die zur Erstellung des Java-Programms benötigten Bibliotheken. Von ihm können mit `getPackageFragmentRoots()` alle *IPackageFragmentRoots* erhalten werden.

IPackageFragmentRoots sind allgemeine Behälter für Pakete. Sie stellen entweder Bibliotheken in Form von .jar-Dateien oder Verzeichnisse dar, die nicht zur Paketstruktur gehören. Beispiele für Letzteres sind das *src*-Verzeichnis für den Quellcode oder das *Build*-Verzeichnis für die kompilierten `.class`-Dateien. Zur Unterscheidung von Bibliotheken und Verzeichnissen dient ein `isArchive()`-Kennzeichner. Um zusätzlich den Quellcode- vom *Build*-Ordner abzuheben, kann die Art des *IPackageFragmentRoot* mit `getKind()` abgefragt werden, die entsprechend ihrem Inhalt entweder die Ausprägung `Source` oder `Binary` trägt.

Mit der Methode `getChildren()` erhält man alle enthaltenen Paket-Verzeichnisse in Form von *IPackageFragments* und gelangt so zu den eigentlich zu analysierenden Java-Strukturen. In den *IPackageFragments* befinden sich *ICompilationUnits* (im weiteren Übersetzungseinheit genannt) als Repräsentationen von `.java`-Dateien, die mit `getCompilationUnits()` gesammelt werden können. Da in einer `.java`-Datei mehrere Klassen/Schnittstellen definiert werden können, sind sie in *ITypes* (im folgenden Typ genannt) gekapselt, die man mit *getTypes()* erhält.

Die Ausprägung der Klasse bzw. Schnittstelle kann von einem *IType* mittels `isClass()` bzw. `isInterface()` erfragt werden. Um zu erfahren, ob der betrachtete Typ lokal oder anonym ist, können die Methoden `isLocal()` bzw. `isAnonymous()` genutzt werden. Weiterhin kann der Typ über die Methoden `getSuperClassName()` und `getSuperInterfaceNames()` Informationen bezüglich seiner Superklasse und seinen implementierten Schnittstellen geben. Zusätzlich können auch Inhalte von Klassen bzw. Schnittstellen wiedergegeben werden, indem mit den Methoden `getMethods()` und `getFields()` Deklarationen von Methoden und Attributen erfragt werden.

3.1.2. Übersicht über das *DOM*-Paket

Das *DOM*-Paket stellt einen AST zur Verfügung, durch den alle Details einer Klasse bzw. Schnittstelle erfasst werden können. Dieser wird aus der entsprechenden *ICompilationUnit* gebildet. Die Elemente eines AST stellen logische Blöcke des Quellcodes dar wie beispielsweise eine *for*-Schleife oder einen Methodenaufruf.

Im Folgenden werden nur die für die Analyse und spätere Modifikation des Quellcodes relevanten Klassen anhand eines Beispiels beschrieben. Jeder Knotentyp im AST stellt eine Klasse im *DOM*-Paket dar. Als Beispiel dient die

`HelloWorld`-Klasse in Abbildung 3.2. Wird aus ihr ein AST gebildet, entspricht seine Struktur dem Baum in Abbildung 3.3. Zur Veranschaulichung steht in Klammern unter der Bezeichnung des AST-Knotens entweder der korrespondierende Teil des Quellcodes oder eine kurze Erklärung.

Abbildung 3.2. Einfache HelloWorld-Klasse

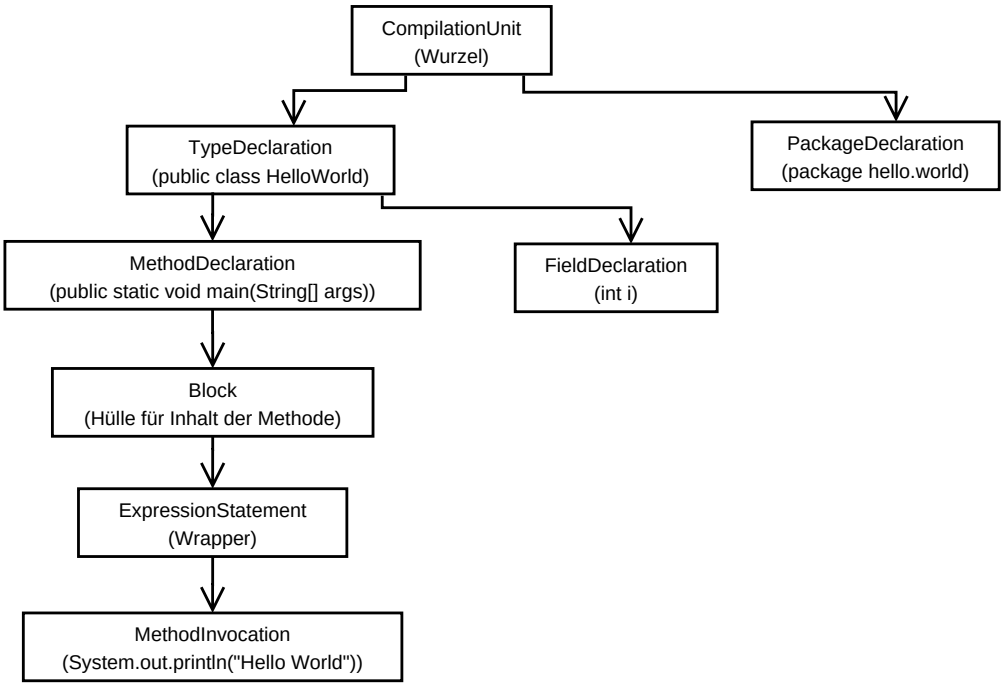
```
package hello.world;

public class HelloWorld{

   private int i;

   public static void main(String[] args){
     System.out.println("Hello World");
   }
}
```

Abbildung 3.3. AST für die HelloWorld-Klasse



Die Wurzel des AST bildet die *CompilationUnit*. Sie repräsentiert analog zur *ICompilationUnit* den Inhalt der `.java`-Datei, und hat als Kinder eine *PackageDeclaration*, die entsprechend ihrem Namen die Paketdeklaration darstellt, und eine *TypeDeclaration*, in der die Klasse `HelloWorld` deklariert wird. Damit ein PlugIn feststellen kann, ob es sich um eine Klasse oder Schnittstelle handelt, besitzt eine *TypeDeclaration* die Methode `isInterface()`. Weiterhin kann sie über die Methoden `getSuperClass()` und `superInterfaces()`

Informationen zu Vererbung und implementierten Schnittstellen geben. Würden hier mehrere Klassen bzw. Schnittstellen deklariert werden, besäße die Wurzel für jede Deklaration ein Kind dieses Typs. Die *TypeDeclaration* besitzt zwei Kinder: eine *FieldDeclaration* für das Attribut `i` und eine *MethodDeclaration* für die *main*-Methode. Analog zur *TypeDeclaration* hätten beide Knoten Geschwister für jedes in HelloWorld enthaltene Attribut bzw. für jede Methode. Der Inhalt einer Methode wird in einem *Block* zusammengefasst, dessen Kinder ein bzw. mehrere *Statements* sind. Da ein Methodenaufruf wie die im Beispiel gezeigte Systemausgabe aber vom Typ *Expression* ist, ist er in ein *ExpressionStatement* gehüllt.

Als Hilfe bei der Analyse des AST stellen die JDT den *ASTVisitor* zur Verfügung, der die Funktionalität des Besucher-Entwurfsmusters (siehe [GHJV96]) anbietet. Durch Erweiterung dieser Klasse muss ein PlugIn nicht selbst durch den Baum iterieren, sondern kann sich auf die Analyse der interessanten Knoten des AST konzentrieren.

Beide Teile der JDT-API überschneiden sich augenscheinlich in der Darstellung von Informationen. Sowohl über den AST als auch über die Kombination von *ICompilationUnit* und seinen enthaltenen *ITypes* können Imports, Klassendeklaration, Superklasse, implementierte Schnittstellen, Attribute und Methodendeklarationen gesammelt werden. *ICompilationUnits* und *ITypes* sind allerdings nicht in der Lage, den Typ beispielsweise eines Attributs aufzulösen. Ist in einer Klasse ein Attribut vom Typ *String* deklariert, kann dieses nicht auf den voll qualifizierten Namen *java.lang.String* zurückgeführt werden. Dadurch würde für den J3Creator die eindeutige Identifikation der beteiligten Java-Elemente einer Beziehung stark erschwert werden. Diese kann zwar mit Hilfe des `import`-Namensraums der Name manuell durchgeführt werden, bedeutet aber einen erheblichen Aufwand, der im AST auf Anforderung automatisch durchgeführt wird. Für den J3Creator empfiehlt sich somit der AST zur Analyse des Quellcodes.

3.2. Analyse des Quellcodes

Dieses Kapitel erläutert, wie mit Hilfe der JDT-API der Quellcode im J3Creator analysiert wird. Für die Visualisierung in Kapitel 4 werden hier Pakete, Klassen und Schnittstellen identifiziert. Sie werden zusätzlich nach folgenden Beziehungen untersucht:

- Assoziation
- Implementierung
- Vererbung
- Benutzung
- Subpaketbeziehung

Entsprechend der Struktur in Abbildung 3.1 beginnt die Analyse im J3Creator mit dem *IJavaProject*. Von ihm ausgehend werden alle *IPackageFragmentRoots* untersucht, die Quellcodeverzeichnisse darstellen. Alle darin enthaltenen *IPackageFragments* werden als Pakete in den J3Creator aufgenommen, sofern sie nicht leer sind. Von ihnen werden alle *ICompilationUnits* dazu benutzt, die deklarierten *ITypes* zu erhalten und den AST zu bilden. Die *ITypes* werden abhängig von der *isInterface()*-Methode als Klasse oder Schnittstelle im J3Creator gespeichert, falls sie weder anonym noch lokal sind. Der AST dient anschließend dazu, die Beziehungen zu identifizieren, da Typen wie der eines Attributs nur im AST auf den voll qualifizierten Namen aufgelöst werden können.

Zur Analyse des AST wird eine Spezialisierung des *ASTVisitors* erstellt, die sich hier auf die Betrachtung der Knoten *TypeDeclaration*, *FieldDeclaration* und *MethodDeclaration* beschränkt. Die *TypDeclaration* wird abhängig vom Rückgabewert der Methode `isInterface()` untersucht. Wird der Wert `true` zurückgegeben, ist der betrachtete Typ eine Schnittstelle. Dann wird für jede Schnittstelle, die in der Liste, die mit `superInterfaces()` angefordert werden kann, enthalten ist, eine Vererbungsbeziehung vermerkt. Ist der betrachtete Typ dagegen eine Klasse, wird für jeden Eintrag dieser Liste eine Implementierungsbeziehung notiert. Hier wird mit Hilfe der Methode `getSuperClass` die Vererbungsbeziehung untersucht.

Mit Hilfe von *FieldDeclarations* werden im AST Assoziationen identifiziert, indem der Typ des deklarierten Attributs betrachtet wird. Ist dieser nicht der betrachtete Typ selbst oder ein primitiver Datentyp, wird eine Assoziation zu ihm vermerkt.

MethodDeclaration-Knoten dienen schließlich dazu, Benutzt-Beziehungen zu analysieren. Dafür wird in jeder betrachteten Methode die Liste der Parameter untersucht, die mit der Methode `parameters()` angefordert werden kann. Wie bei der Analyse von Assoziationen werden Benutzt-Beziehungen zum Typ des Parameters vermerkt, falls der Typ nicht der betrachtete Typ selbst oder ein primitiver Datentyp ist.

Für alle Referenzen auf andere Klassen oder Schnittstellen (z.B. bei Superklassen oder Attributen) gilt, dass sie nur dann weiter betrachtet werden, wenn deren Quellcode Teil des betrachteten *IJavaProjects* sind. Verweise auf Klassen aus der Java-Laufzeitumgebung oder eventuell benutzten Bibliotheken werden nicht weiter im J3Creator behandelt oder dargestellt.

Weitere Einschränkungen sind, dass der J3Creator bei Assoziationen und Benutzt-Beziehungen weder die Kardinalität der Beziehung noch die dafür verantwortlichen Teile des Quellcodes vermerkt. Die erste Einschränkung bewirkt, dass zwischen zwei Klassen `A` und `B` höchstens eine Assoziation und eine Benutzt-Beziehung in der Visualisierung dargestellt wird, unabhängig davon, ob eine oder mehrere Attribute bzw. Methodenparameter vom Typ `B` in `A` existieren. Die zweite Einschränkung führt in Kombination mit der ersten dazu, dass später bei der Modifikation des Quellcodes nicht gezielt nur ein Teil der Beziehung entfernt werden kann. Existieren beispielsweise zwei Attribute

vom Typ B in A, so führt das aufgrund der ersten Beschränkung dazu, dass nur eine Assoziation von A nach B vermerkt wird. Soll diese Assoziation durch eine semantikverändernde Modifikation entfernt werden, müssen alle Attribute des Typs B aus A entfernt werden.

Problematisch bei der Analyse der Assoziationen und Benutzt-Beziehungen ist die Verwendung von *Collection*-Klassen wie *Vector* oder *Hashtable*. Werden sie benutzt, um eine beliebige Menge von Objekten einer Klasse zu verwalten, kann das bei der Analyse des Quellcodes nicht nachvollzogen werden. Enthält beispielsweise eine Klasse A in einem *Vector* eine beliebige Menge von Instanzen einer Klasse B, so müsste A eine Assoziation zu B haben. Da die `get`-Operation des *Vectors* laut Signatur die allgemeinste Java-Klasse *Object* zurück gibt, wird diese Assoziation bei der Analyse nicht erkannt. Typisierte *Collections* in Form von *Generics* (s. [Hei]) könnten das verhindern, werden aber erst ab Java-Version 1.5 zur Verfügung gestellt, das sich allerdings noch im Beta-Test-Stadium befindet.

3.3. Möglichkeiten zur Modifikation des Quellcodes

Im Hinblick auf die semantikverändernden Modifikationen in Kapitel 8 muss der analysierte Quellcode auch modifiziert werden. Dieses Kapitel bietet eine kurze Übersicht über die Möglichkeiten zur Modifikation des Quellcodes, die die JDT zur Verfügung stellen. So wie die Analyse in den beiden Paketen *Core* und *DOM* verschieden durchgeführt wird, unterscheiden sich ebenfalls die Möglichkeiten zur Modifikation des Quellcodes. Dementsprechend werden beide Pakete in den beide folgenden Abschnitten getrennt behandelt.

3.3.1. Modifikation im *Core*-Paket

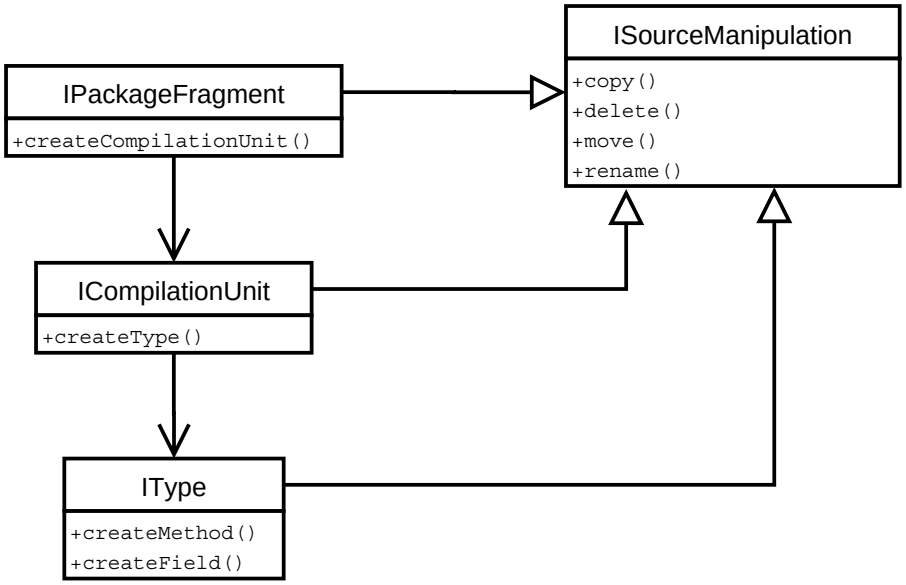
Das *Core*-Paket unterstützt lediglich die Modifikation des gesamten betrachteten Elements. Pakete, CompilationUnits und Typen können über die Schnittstelle *ISourceManipulation*, wie Abbildung 3.4 zeigt, kopiert, gelöscht, versetzt und umbenannt werden. Zusätzlich kann jedes Element mit Hilfe einer *create*-Methode neue Kinder erzeugen.

Eigenschaften eines Elements wie etwa seine Superklasse können lediglich ausgelesen, nicht aber verändert werden. Sollte also mit den Möglichkeiten, die das *Core*-Paket zur Verfügung stellt, die Superklasse eines Typs verändert werden, müsste der alte Typ gelöscht und ein Neuer mit demselben Namen, aber einer anderen Superklasse erstellt werden.

3.3.2. Modifikation im *DOM*-Paket

Um Details einer Klasse bzw. Schnittstelle zu verändern, muss auf das *DOM*-Paket zurückgegriffen werden. Dort kann im AST in jedem Knoten jede Ei-

Abbildung 3.4. Modifikation im *Core*-Paket



genschaft, die ausgelesen werden kann, nach einem *get/set*-Prinzip ebenfalls verändert werden.

> *Beispiel*
> In der *FieldDeclaration* für die Variable `i` der Klasse HelloWorld aus Abbildung 3.3 (HelloWorld-AST) kann über *get*-Methoden sowohl der Typ (`int`) als auch der Name (`i`) erfragt werden. Über ihre *set*-Pendants können beide Eigenschaften verändert werden, z.B. zum Typen `String` und den Namen `hello`.

Um Knoten aus dem AST zu entfernen oder neue hinzuzufügen, gibt es zwei Möglichkeiten, abhängig davon, ob es mehrere Exemplare des betrachteten Knotentyps geben darf oder nicht. Falls mehrere Knoten erlaubt sind, kann vom Vater eine Liste dieses Kindknotentyps angefordert werden. In dieser Liste können anschließend Elemente entfernt oder eingesetzt werden, wobei jegliche Änderung direkt auf den AST übertragen wird. Falls nur ein Knoten erlaubt ist, kann über die *set*-Methode im Elter mit dem Parameter `null` der Knoten entfernt werden.

> *Beispiele*
>
> 1. Soll aus der HelloWorld-Klasse der Inhalt der *main*-Methode verändert werden, so kann vom zugehörigen *Block* die Liste der *Statements* angefordert werden. Aus dieser Liste kann man nun entweder das vorhandene *ExpressionStatement* löschen oder neue hinzufügen. Ebenso kann die Reihenfolge der Statements verändert werden.

2. Da es in einer Klasse/Schnittstelle höchstens eine PaketDeklaration geben darf, stellt die *CompilationUnit* keine Liste von Deklarationen zur Verfügung. Stattdessen kann die Paketangabe über *setPackageDeclaration()* mit dem Parameter `null` entfernt werden.

Änderungen im AST wirken sich normalerweise nicht auf den Quellcode aus. Wird also ein AST modifiziert, müssen die Änderungen manuell übertragen werden. Dies geschieht mit Hilfe eines *TextEdit*-Objekts, das allgemein eine Textmanipulation beschreibt. Im Falle des ASTs beinhaltet er alle Quellcodemodifikationen, die durch die Modifikation des ASTs ausgelöst werden. Diese Textmanipulation muss dann auf den Quellcode angewendet werden, der durch ein *Document*-Objekt in den JDT repräsentiert wird. Abbildung 3.3.2 zeigt den Java-Quellcode, der benötigt wird, um Modifikationen im AST in den Quellcode zu übertragen. Ein *Document*-Objekt wird mit Hilfe der *ICompilationUnit* erzeugt, aus der auch der AST generiert wird, und beinhaltet den gesamten Quellcode der Übersetzungseinheit.

Abbildung 3.5. Übertragung der Modifikationen eines ASTs in Quellcode

```
// Document aus der ICompilationUnit erzeugen
ICompilationUnit cu = ... ; // Inhalt ist z.B. "public class X {\n}"
String source = cu.getBuffer().getContents();
Document document= new Document(source);

// AST erzeugen
ASTParser parser = ASTParser.newParser(AST.JLS2);
parser.setSource(cu);
CompilationUnit astRoot = (CompilationUnit) parser.createAST(null);

// Anweisung, Modifikationen zu speichern
astRoot.recordModifications();

// Modifikationen am AST
...

// TextEdit-Objekt der Modifikationen anfordern
TextEdit edits =
     astRoot.rewriteAST(document, cu.getJavaProject().getOptions(true));
// neuen Quellcode berechnen
edits.apply(document);
String newSource = document.get();
// ICompilationUntit aktualisieren
cu.getBuffer().setContents(newSource);
```

Damit alle Änderungen im AST erfasst werden, muss vor den Modifikationen mittels der Methode `recordModifications()` signalisiert werden, dass später ein *TextEdit* zur Quellcodemodifikation erwünscht wird. Dieses kann nach den Modifikationen des ASTs mit Hilfe der Methode `rewriteAST()` angefordert werden, das als Parameter das zugrunde liegende *Document* und die Optionen des Java-Projekts erhält. Die Textmanipulationen werden anschließend durchgeführt, indem die Methode `apply()` aufgerufen wird, die den Quellcode im Speicher ändert. Der so geänderte Quellcode muss dann abschließend

mit `setContents()` in die betrachtete Übersetzungseinheit übertragen werden.

4. Visualisierung mit Java3D

Nachdem der Java-Quellcode in Kapitel 3 mit Hilfe der *Java Development Tools* verarbeitet worden ist, behandelt dieses Kapitel die Darstellung der erfassten Informationen. Aus den gesammelten Paketen, Klassen und Schnittstellen mit ihren Beziehungen wird eine dreidimensionale Darstellung nach dem Vorbild des J3Browsers ([Eng00]) konstruiert. Als grafische Repräsentationen werden dementsprechend genutzt:

- für Pakete: *Information Cubes*
- für Klassen: Würfel
- für Schnittstellen: Kugeln

Die visualisierten Beziehungen werden ebenfalls übernommen und als Röhren dargestellt. Diese sind im Einzelnen:

- Assoziation
- Implementierung
- Vererbung
- Benutzung

Neu hinzu kommt die Subpaketbeziehung, da anders als im J3Browser *Information Cubes* nicht ineinander verschachtelt werden, sondern lediglich Würfel und Kugeln enthalten.

Als Visualisierungstechnik wird im J3Creator Java3D ([J3D]) genutzt, das als Erweiterungs-API ([Api]) für Java einfach in ein Java-Programm integrierbar ist. Dadurch ist der J3Creator in der Lage, im Gegensatz zum J3Browser Modifikationen der Visualisierung zu verarbeiten (vgl. Kapitel 1.2), die in den nachfolgenden Kapiteln behandelt werden.

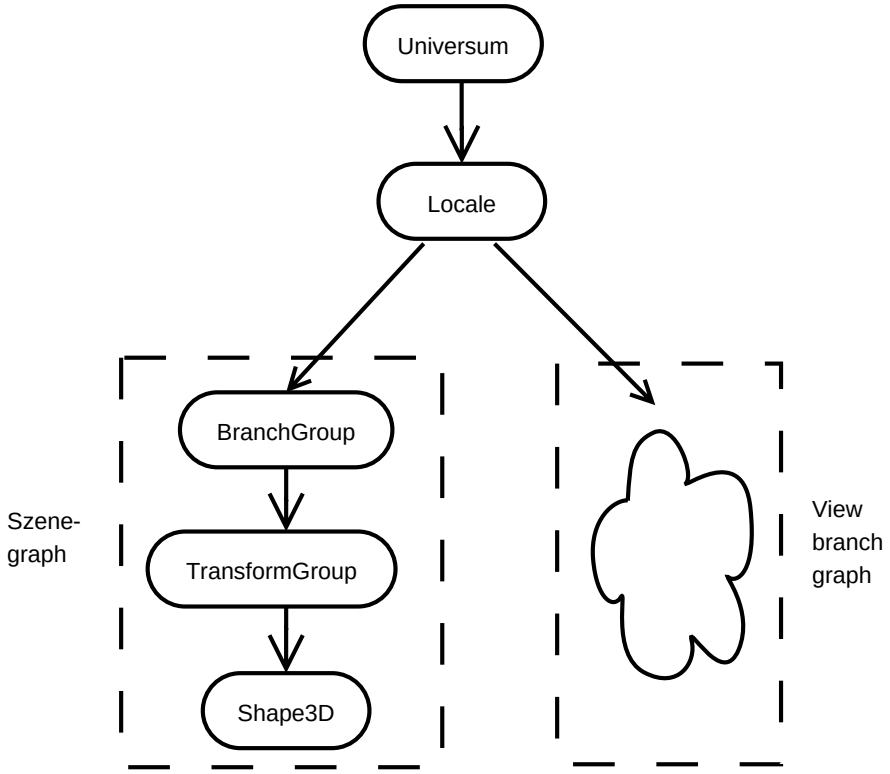
Kapitel 4.1 gibt eine Übersicht über Java3D und stellt den Szenegraphen vor, der thematisch den Schwerpunkt bildet. Anschließend wird in Kapitel 4.2 gezeigt, wie mit Hilfe des Szenegraphen die Visualisierung nach Vorbild des J3Browsers konstruiert wird.

4.1. Aufbau von Java3D

Java3D ist eine Erweiterungs-API für Java und somit einfach per Instanziierung und Methodenaufruf in ein Java-Programm integrierbar. Ähnlich wie VRML ist Java3D szenengraphorientiert (vgl. [Eng00] Kapitel 11 und [Cou99]). Ein Szenegraph beschreibt eine Szene mittels eines hierarchisch geordneten Baumes, in dem nur die Blätter grafische Objekte darstellen. Innere Knoten dienen entweder der Gruppierung oder der Transformation der dargestellten Objekte. Im folgenden Unterkapitel (4.1.1) werden der Szenegraph von Java3D und seine Elemente näher beschrieben.

Neben dem Szenegraphen existiert in Java3D noch der *view branch graph*. Er definiert die Sicht des Benutzers auf die dargestellte Szene in Form des sichtbares Bildausschnittes und der Blickrichtung auf die dargestellten Objekte.

Abbildung 4.1. Aufbau von Java3D



Beide Graphen liegen in einem prinzipiell unendlich grossem Universum, in dem das Locale den Nullpunkt markiert. Abbildung 4.1 stellt den Aufbau von Java3D mit einem einfachen Szenegraphen schematisch dar.

Im Mittelpunkt dieses Kapitels steht der Szenegraph, da er so aufgebaut werden muss, dass er die mittels der JDT erfasste Paket- und Klassenstruktur wiedergibt.

Da die Sicht in diesem Kapitel nicht verändert wird, wird hier nicht näher auf den *view branch graph* eingegangen, sondern erst in Kapitel 5.

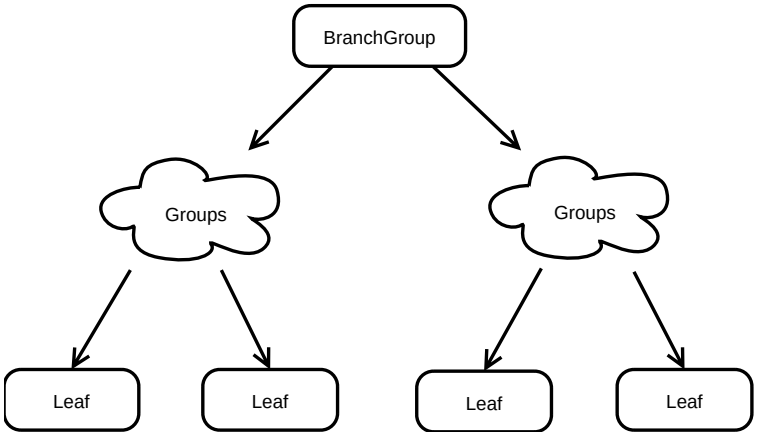
4.1.1. Struktur des Szenegraphen

Ein Szenegraph beschreibt eine Szene, die wiederum alle grafischen Objekte des virtuellen Raums der Visualisierung darstellt. Formal beschrieben ist er ein gerichteter Baum mit zwei Knotenmengen $Group$ und $Leaf$ und einer Kantenmenge, die eine Teilmenge ist von $Group \times Group \cup Group \times Leaf$.

Die Menge $Group$ setzt sich aus zwei disjunkten Mengen $BranchGroup$ und $TransformGroup$ zusammen, so dass $Group := BranchGroup \cup TransformGroup$ gilt. Ebenso besteht $Leaf$ aus zwei disjunkten Mengen $Shape3D$ und $Behavior$ mit $Leaf := Shape3D \cup Behavior$.

Die Wurzel `w` des Szenegraphen ist immer ein Knoten aus der Menge $BranchGroup$, so dass $w \in BranchGroup$ gilt. Jeder innere Knoten `i` ist aus der Menge $Group$. $Leaf$ bildet hingegen die Grundlage für jedes Blatt `b`, es gilt also $b \in Leaf$. Abbildung 4.2 veranschaulicht den soeben beschriebenen allgemeinen Aufbau eines Szenegraphen.

Abbildung 4.2. Allgemeiner Aufbau eines Szenegraphen



Zur sprachlichen Vereinfachung wird im weiteren die Art eines Knotens synonym mit dem Namen der Menge identifiziert, zu der er gehört. Anstatt also zu beschreiben, dass ein Knoten `k` ein Element der Menge $BranchGroup$ ist, wird `k` direkt als BranchGroup bzw. BranchGroup-Knoten bezeichnet.

Group-Knoten dienen allgemein dazu, ihre Kinder als logische Einheit in sich zusammenzufassen. BranchGroups besitzen als einziger Knotentyp darüber hinaus die Fähigkeit, nachträglich wieder aus dem Szenegraphen entfernt werden zu können. TransformGroups beinhalten eine Transformation, die eine Positionierung, eine Rotation und eine Skalierung der Größte umfassen kann, und wenden diese auf allen Kindern an.

Leaf-Knoten weisen allgemein betrachtet über die Tatsache hinaus, dass sie als Blätter keine Kinder besitzen, keine besonderen Eigenschaften auf. Behavior-Knoten ermöglichen als Spezialisierung die Interaktion mit dem Benutzer, indem sie auf Ereignisse wie einen Mausklick reagieren können. Shape3Ds re-

präsentieren im Szenegraphen das eigentliche zu visualisierende grafische Objekt.

Durch eine Kante wird ein Knoten `k` einem Group-Knoten `g` untergeordnet. Ist `g` eine BranchGroup, bedeutet eine Kante, dass `k` im Falle der Entfernung von `g` aus dem Szenegraphen ebenfalls entfernt wird. Ist `g` hingegen eine TransformGroup, wird die in ihm enthaltene Transformation auf `k` übertragen. Abhängig vom Knotentyp von `k` wird diese Transformation auf dem Knoten angewendet, hinzugefügt oder einfach weitergegeben. Ist `k` ein Leaf-Knoten, wird die Transformation auf dem grafischen Objekt angewendet, so dass es positioniert, rotiert oder skaliert wird. Falls `k` eine TransformGroup ist, wird die Transformation von `g` zu derjenigen von `k` hinzugefügt. Wenn `k` eine BranchGroup ist, wird die Transformation an die Kinder weitergegeben.

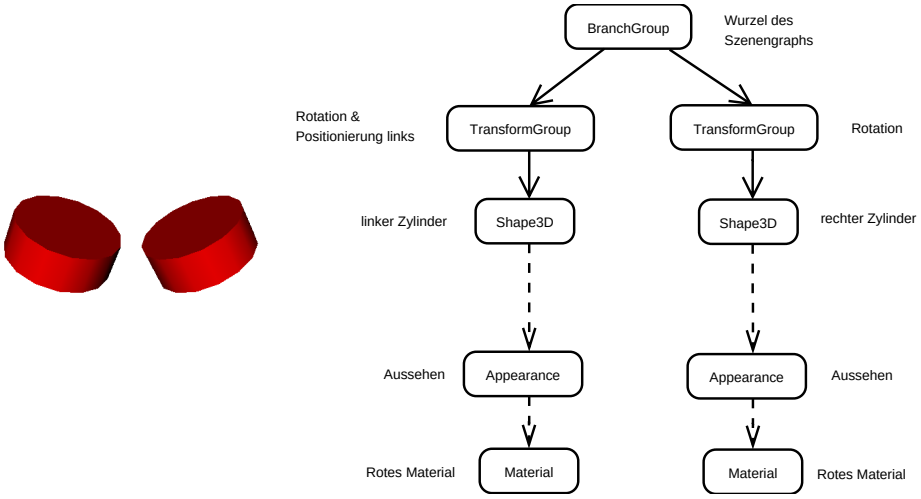
Ein Szenegraph beschreibt auf diese Weise genau eine Szene. Allerdings gilt nicht umgekehrt, dass zu einer Szene genau ein Szenegraph existiert. Grafische Objekte, die durch Shape3D repräsentiert werden, werden durch TransformGroups positioniert, rotiert oder skaliert. Die endgültige Transformation, die auf einem Shape3D angewendet wird, ergibt sich aus dem Pfad von der Wurzel zum betrachteten Blatt. Alle auf diesem Pfad betrachteten TransformGroups tragen zu dieser Transformation bei. Daraus folgt, dass in einem Szenegraphen lediglich die Summe der Transformationen für jedes Blatt gleich sein muss, um dieselbe Szene zu beschreiben, und die Struktur ansonsten beliebig ist. Im Hinblick auf die Modifikation der Visualisierung ist aber eine fest definierte Struktur zu empfehlen, um durch Modifikationen im Szenegraphen nur exakt die beabsichtigten Elemente der Visualisierung zu verändern.

4.1.2. Beispielhafter Szenegraph

Der soeben beschriebende Szenegraph wird in diesem Abschnitt anhand eines Beispiels veranschaulicht. Abbildung 4.3 zeigt zwei rote Zylinder und den zugehörigen Szenegraphen. Die Wurzel des Szenegraphen bildet eine BranchGroup, der einen Teilbaum für jeden Zylinder erhält. Beide Zylinder werden mit Hilfe je einer TransformGroup leicht gedreht, wobei die Linke zusätzlich den linken Zylinder versetzt, damit beide Objekte nebeneinander und nicht ineinander dargestellt werden. Ein Zylinder ist in seiner Standardausprägung lichtgrau, so dass sein Aussehen mittels seiner *Appearance* verändert werden muss, indem in ihr durch das *Material* das rote Äußere gesetzt wird. *Appearance* und *Material* sind nicht direkt Knoten des Szenegraphen, sondern Komponenten des Zylinders, weswegen die Pfeile zu ihnen gestrichelt sind.

Abbildung 4.1.1 zeigt den Java-Quellcode, der den Szenegraphen aus Abbildung 4.3 konstruiert. Zur Erzeugung eines Zylinders dient die Klasse `Cylinder`, die eine Erweiterung von Shape3D ist und als Parameter Radius und Höhe erhält. Sie wird von Java3D vorgefertigt zur Verfügung gestellt und enthält die geometrischen Informationen, um einen Zylinder darzustellen. Neben ihr werden noch folgende vorgefertigte Klassen zur Verfügung gestellt:

Abbildung 4.3. Darstellung von zwei Zylindern und dem zugehörigen Szenegraph



Box stellt einen Würfel dar.

Sphere stellt eine Kugel dar.

Cone stellt einen Kegel dar.

Wird eine komplizierte Form benötigt, kann diese entweder aus den soeben aufgezählten Klassen zusammengestellt oder selbst definiert werden. Dazu muss ein Shape3D erstellt werden, in dem mit Hilfe einer Punktemenge die Eckpunkte der zu visualisierenden Form angegeben werden. Für diese Punkte muss anschließend bestimmt werden, ob sie durch Linien verbunden werden sollen oder beispielsweise je drei Punkte ein gefülltes Dreieck bilden. Der J3Creator greift zur Visualisierung komplett auf die vorgefertigten Klassen zurück, weswegen die Konstruktion kompliziertere Formen nicht weiter behandelt wird.

Die rote Färbung des Zylinders wird mit der Methode `setDiffuseColor()` erzielt, indem ihr 1 als Rotwert und 0 als Grün- und Blauwert übergeben werden. Eine TransformGroup enthält eine `Transform3D`, in der die eigentliche Transformation definiert wird. In ihr kann eine Positionierung mit `setTranslation()` oder eine Rotation mit `rot`-Methoden definiert werden. Um verschiedene Transformationen zu kombinieren, müssen sie mit Hilfe der Methode `mul()` miteinander multipliziert werden.

4.2. Darstellung der analysierten Strukturen mit Java3D

Nach der Vorstellung von Java3D beschreibt dieses Unterkapitel, wie die Visualisierung aufgebaut wird. Um die mit Hilfe der JDT analysierten Strukturen und Beziehungen des Quellcodes darzustellen, muss ein Szenegraph, wie

Java Quellcode 4.1.1 Java-Quellcode zu Abbildung 4.3

```
public class RoteZylinder{
   public void konstruiereSzenegraph(){
      BranchGroup graph = new BranchGroup();    // die Wurzel des Szenegraphen

      Cylinder rechterZylinder = new Cylinder(0.2f, 0.15f);// rechter Zylinder

      Appearance aussehenRechts = new Appearance();     // Aussehen definieren
      Material materialRechts = new Material();
      materialRechts.setDiffuseColor(1f, 0, 0);       // rote Farbe definieren
      aussehenRechts.setMaterial(materialRechts);
      rechterZylinder.setAppearance(aussehenRechts);      // Aussehen zuordnen

      TransformGroup tgRechts = new TransformGroup();
      Transform3D rotRechts = new Transform3D(); // eigentliche Transformation
      rotRechts.rotX(Math.PI/6d);               // zuerst Rotation um X-Achse
      Transform3D rotZRechts = new Transform3D();
      rotZRechts.rotZ(Math.PI/8d);                      // Rotation um Z-Achse
      rotRechts.mul(rotZRechts);           // kombinieren durch Multiplikation
      tgRechts.setTransform(rotRechts);            // Transformation einsetzen

      tgRechts.addChild(rechterZylinder); // Zylinder zur TransformGroup hinzu
      graph.addChild(tgRechts);       // TransformGroup zum Szenegraphen hinzu


      Cylinder linkerZylinder = new Cylinder(0.2f,0.15f);   // linker Zylinder

      Appearance aussehenLinks = new Appearance();
      Material materialLinks = new Material();
      materialLinks.setDiffuseColor(1f, 0, 0);
      aussehenLinks.setMaterial(materialLinks);
      linkerZylinder.setAppearance(aussehenLinks);

      TransformGroup tgLinks = new TransformGroup();
      Transform3D transform = new Transform3D();
      transform.setTranslation(new Vector3f(-0.5f,0,0));// links        positionieren

      Transform3D rotLinks = new Transform3D();
      rotLinks.rotX(Math.PI/6d);
      Transform3D rotZLinks = new Transform3D();
      rotZLinks.rotZ(-Math.PI/8d);
      rotLinks.mul(rotZLinks);
      transform.mul(rotLinks);
      tgLinks.setTransform(transform);

      tgLinks.addChild(linkerZylinder);
      graph.addChild(tgLinks);
   }
}
```

in Kapitel 4.1 vorgestellt, konstruiert werden. Aus dieser Aufgabe entstehen zwei Probleme:

Konvertierung jedes Quellcodeelements in Szenegraphelemente Mit Hilfe der in Abschnitt 4.1.1 vorgestellten Elemente BranchGroup, TransformGroup und Shape3D (bzw. seiner Unterklassen) muss analog zu Abbildung 4.3 ein Teilbaum des Szenegraphen konstruiert werden. Für jedes Paket, jeden Typ und jede Beziehung muss eine geeignete Kombination der drei Elemente zur gewünschten Darstellung führen. Die Lösung dieses Pro-

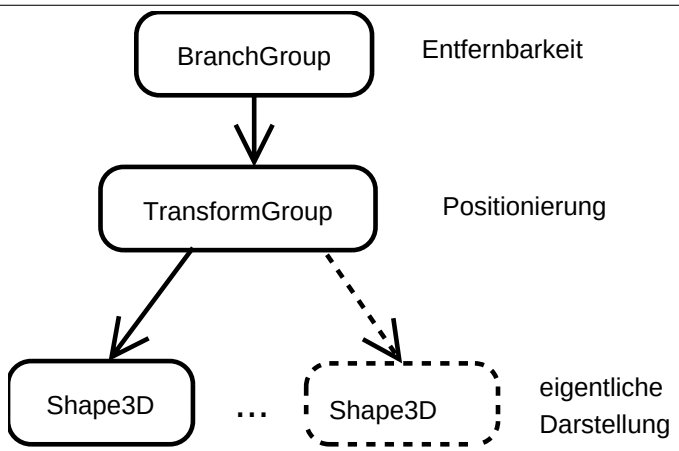
blems wird in Kapitel 4.2.1 erörtert.

Konstruktion des gesamten Szenegraphen Nachdem für jedes Element des Modells eine Entsprechung in Form von Szenegraphelementen existiert, wird in Kapitel 4.2.2 darauf eingegangen, wie die Einzelteile zum gesamten Szenegraphen zusammengefügt werden, der anschließend mit Java3D angezeigt wird. Hier ist es im Hinblick auf die Manipulation der Visualisierung wichtig, dass Modifikationen des Szenegraphen nicht zu unerwünschten Nebenwirkungen in der Darstellung führen.

4.2.1. Konvertierung in Szenegraphelemente

Um ein Element der analysierten Strukturen und Beziehungen des Quellcodes darzustellen, wird zunächst mindestens eine Klasse vom Typ Shape3D benötigt. Da der J3Creator in der Lage sein muss, diese Darstellung zu positionieren, müssen die Shape3Ds einer TransformGroup untergeordnet werden. Die Möglichkeit zur Manipulation der Visualisierung macht es weiterhin erforderlich, dass die Darstellung aus dem Szenegraphen entfernt werden kann, was nur möglich ist, wenn eine BranchGroup die Wurzel des dargestellten Elements bildet. Abbildung 4.4 gibt diesen Aufbau in einer Übersicht wieder. Neben den Knoten ist die jeweilige Rolle angegeben.

Abbildung 4.4. Allgemeiner Aufbau eines anzuzeigendes Elements in Java3D



Dieser Aufbau gilt allgemein für jedes visualisierte Element unabhängig davon, ob es ein Paket, eine Klasse oder eine Beziehung ist. Bezüglich der Darstellung in Java3D unterscheiden sie sich lediglich durch die konkreten Shape3Ds.

Daraus resultierend kann die Kombination aus BranchGroup als Wurzel und TransformGroup zur Positionierung als allgemeinstes Konstrukt für die Visualisierung eines Elements dienen, die im Weiteren als *J3DObject* bezeichnet wird. Sie stellt sicher, dass sie aus dem Szenegraphen entfernt werden kann und im virtuellen Raum positionierbar ist. Alle Darstellungen können

anschließend das *J3DObject* so erweitern, dass sie nur noch die entsprechenden Shape3Ds an die TransformGroup anhängen müssen, um so Pakete oder andere Komponenten zu beschreiben. Im Bemühen, möglichst nah an der Visualisierung des J3DBrowsers (vgl. [Eng00] Anhang D) zu bleiben, werden dabei für die Elemente des Quellcodes folgende Shape3Ds gewählt:

Paket: blaue, transparente Box

Klasse: rote, nicht-transparente Box

Schnittstelle: roter Sphere

Beziehung allgemein: langer, dünner Cylinder u.U. mit Cone als Pfeilspitze (falls gerichtet) in den Farben:

Assoziation: orange

Vererbung: rot

Implementierung: gelb

Benutzt: grau

Subpaket: schwarz

Die Transparenz eines Shape3Ds kann mit Hilfe der *TransparencyAttributes* gesetzt werden, die Teil der *Appearance* sind.

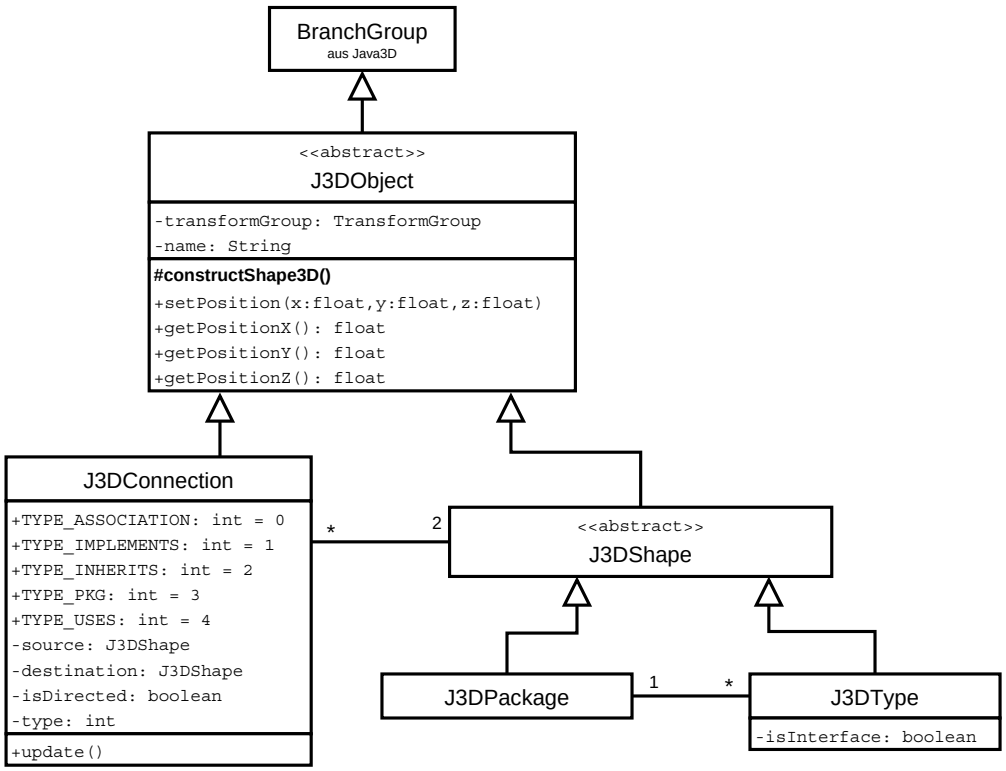
Abbildung 4.5 stellt in einem Klassendiagramm dar, wie *J3DObject* erweitert wird, um die Darstellung eines Elements zu erreichen (zur Bewahrung der Übersicht wurde auf die Darstellung von einfachen get-/set-Methoden verzichtet).

Wie bereits erwähnt stellt *J3DObject* das allgemeinste Konstrukt eines visualisierten Elements dar. Diese Klasse ist eine Erweiterung von BranchGroup und ist somit die Wurzel in Abbildung 4.4. Mit Hilfe von `setPosition`- und `getPosition`-Methoden kann die enthaltene TransformGroup manipuliert werden.

Da das *J3DObject* selbst keine Elemente zur eigentlichen Darstellung enthält, ist es eine abstrakte Klasse, die mit Hilfe der abstrakten Methode `constructShape3D()` sicherzustellen versucht, dass eine erbende Klasse auch eine Visualisierung konstruiert. Die zwei direkten Erben sind *J3DShape* und *J3DConnection*, die ein Element bzw. eine Beziehung zwischen zwei Elementen visualisieren.

J3DShape ist wiederum eine abstrakte Klasse, die als Verallgemeinerung für *J3DPackage* und *J3DType* dient. Sie stellt eine Komponente dar, die Beziehungen zu anderen J3DShapes verwaltet. Ein *J3DPackage* ist die Java3D-Repräsentation eines Java-Pakets, das Referenzen auf alle Entsprechungen der in diesem Paket enthaltenen Typen besitzt. Umgekehrt weiß ein J3DType ebenfalls, zu welchem Paket-*Information Cube* es gehört. Es visualisiert abhängig vom `isInterface`-Kennzeichner eine Klasse bzw. eine Schnittstelle.

Abbildung 4.5. Vererbungsstruktur von J3DObject



Beziehungen zwischen J3DShapes werden durch J3DConnections verkörpert. Da sie sich in ihrer Visualisierung nur geringfügig unterscheiden, wird hier auf weitere Vererbungen verzichtet und stattdessen die Art der Beziehung durch das Attribut `type` vermerkt, das die Werte der `TYPE_`-Variablen annehmen kann. Trotz der Bezeichnungen `source` und `destination` für die beiden Endpunkte wird die Beziehung als ungerichtet angenommen, kann aber mit Hilfe des `isDirected`-Kennzeichners als gerichtet markiert werden, durch die am Ende von `destination` eine Pfeilspitze erscheint. Die Methode `update()` veranlasst die J3DConnection dazu, die Positionen der Endpunkte neu auszulesen und anschließend ihre Darstellung zu aktualisieren. Sie wird nach einer Positionsänderung eines J3DShapes aufgerufen, da die Verbindung von J3DConnection und J3DShape in Java3D nicht existiert, sondern nur dementsprechend gezeichnet wird.

4.2.2. Konstruktion des Szenegraphen

Nachdem die einzelnen Teile des Szenegraphen realisiert sind, müssen sie zu einem Ganzen kombiniert werden. Da ihre Anordnung im Szenegraphen nicht der Struktur der eigentlichen Visualisierung entsprechen muss, ist sie prinzipiell beliebig.

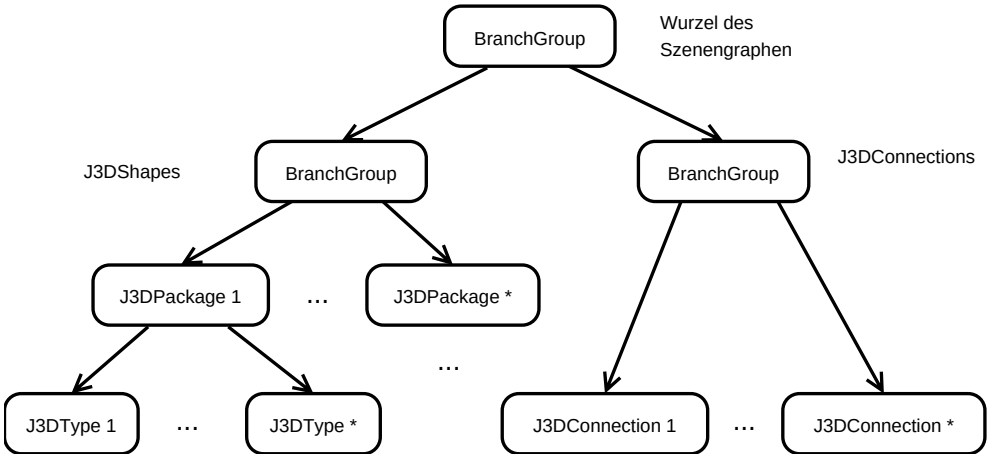
Eine unstrukturierte Zusammensetzung kann allerdings dazu führen, dass sich

die TransformGroups zur Positionierung in ungewünschter Weise beeinflussen, so dass sich beispielsweise eine Box nicht in dem ihr angedachten *Information Cube* befindet.

Mit der Möglichkeit, die Visualisierung und damit den Szenegraphen zu manipulieren, entstehen weitere Fehlermöglichkeiten, die dazu führen können, dass die Darstellung nicht mehr dem zugrunde liegenden Java-Quellcode entspricht. So kann eine chaotische Kombination von J3DPackages und J3DTypes dazu führen, dass die Entfernung eines J3DPackages ebenfalls ein J3DType entfernt, dessen Typ gar nicht Teil des zu entfernenden Pakets ist. Umgekehrt könnte ebenfalls eine Box übrig bleiben, die keinen umgebenden *Information Cube* mehr besitzt. In solch einem Fall müsste der Quellcode erneut analysiert und der Szenegraph komplett neu aufgebaut werden, damit beide wieder synchron sind.

Um solche Fehlerquellen zu vermeiden, wird der Szenegraph wie in Abbildung 4.6 unterteilt.

Abbildung 4.6. Unterteilung des Szenegraphen



Entsprechend der Struktur in Abbildung 4.5 existiert ein Zweig für J3DConnections und ein Zweig für J3DShapes.

Die J3DConnections unterliegen keiner weiteren Struktur, sondern sind alle parallel auf einer Ebene angeordnet. So ist sichergestellt, dass das Entfernen oder Hinzufügen einer Beziehung keine andere J3DConnection beeinflusst.

Der Teilbaum des J3DShapes besitzt als Kinder zunächst J3DPackages, deren Kinder entsprechend der Paketzugehörigkeit J3DTypes sind. Diese Entsprechung hat den Vorteil, dass sie relativ zum *Information Cube* ausgerichtet und bei einer Positionsänderung des J3DPackage automatisch mit verschoben werden. Weiterhin wird dadurch bei der Entfernung der Paketvisualisierung ihr Inhalt mit beseitigt. Allerdings wir die Pakethierarchie nicht analog in der Struktur des Szenegraphen abgebildet. Zwar könnten ebenso Unterpakete relativ zu ihrem übergeordneten Paket positioniert werden, diese müssten allerdings bei Entfernung des Oberpakets im Szenegraphen versetzt werden.

Teil III.

Modifikation

5. Realisierung der Navigation

Existiert eine geeignete Visualisierung, ist die Möglichkeit zur Navigation in ihr wünschenswert, um sich auf bestimmte Ausschnitte zu konzentrieren oder die Darstellung aus einer anderen Perspektive zu betrachten. Der Benutzer soll sich zu diesem Zweck frei in der Visualisierung bewegen können, als ob er ein Teil dessen ist. Dies erfordert zwei Konzepte: die Manipulation der Blickrichtung und der Position des Betrachters sowie die Möglichkeit, Eingaben vom Benutzer zu erhalten und zu verarbeiten.

Die Position und Blickrichtung des Benutzers auf die beschriebene Szene wird in Java3D durch den *view branch graph* beschrieben, der bereits in Kapitel 4.1 eingeführt worden ist und in Kapitel 5.1 näher beschrieben wird. Kapitel 5.2 erläutert danach, wie der Benutzer in Java3D mit der Visualisierung interagieren kann. Abschließend stellt Kapitel 5.3 dar, wie beide Elemente zum Zwecke der Navigation kombiniert werden.

5.1. Der *view branch graph*

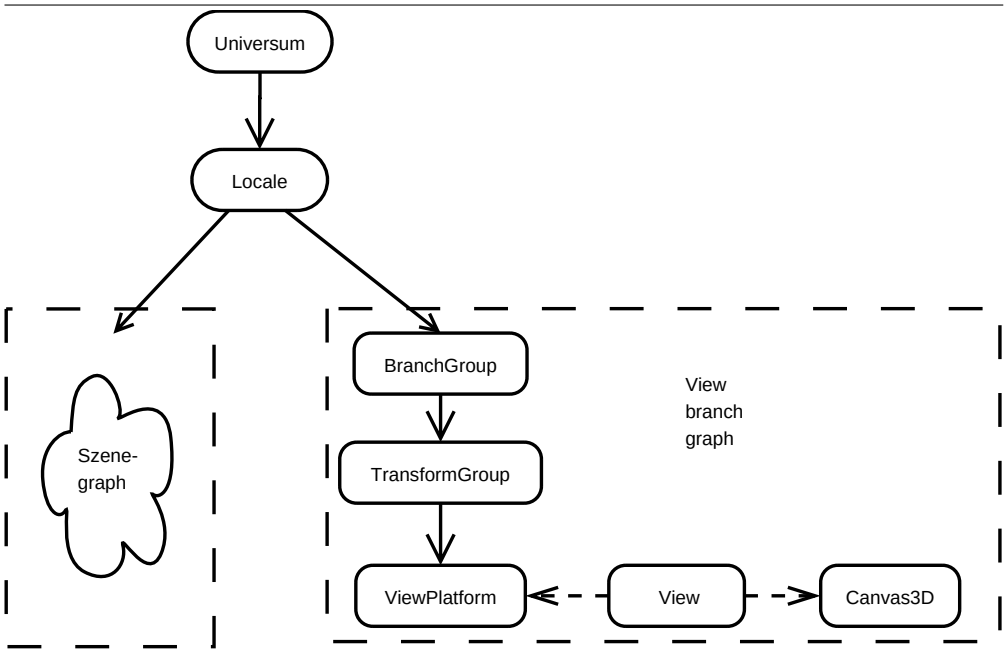
Der *view branch graph* besteht aus einem dem Szenegraph ähnlichen Baum, einem *View*-Element und einer *Canvas3D*. Er wird in Abbildung 5.1 schematisch dargestellt.

Der Baum besitzt wie der Szenegraph eine BranchGroup als Wurzel, enthält aber einen speziellen Knoten: die *ViewPlatform*. Sie repräsentiert den Betrachter im virtuellen Raum und definiert zusammen mit der ihr übergeordneten TransformGroup seine Position und Blickrichtung.

Das *View*-Element berechnet aus der Position und Blickrichtung den für den Betrachter sichtbaren Bereich des virtuellen Raums und projiziert diesen auf eine zweidimensionale Fläche in Form eines *Canvas3D*, der anschließend in einem Swing- oder AWT-Fenster angezeigt werden kann.

Zur Realisierung der Navigation muss die TransformGroup des *view branch graph* so modifiziert werden, dass die Modifikation der gewünschten Änderung der Position oder der Blickrichtung des Betrachters entspricht. Kapitel 5.3 beschreibt, wie die benötigten Modifikationen berechnet und umgesetzt werden.

Abbildung 5.1. Schema des View Branch Graphs



5.2. Interaktionsmöglichkeiten in Java3D

Dynamische Ereignisse wie das Drücken einer Taste können in Java3D mit Hilfe eines Behaviors verarbeitet werden, das in Kapitel 4.1 bereits eingeführt worden ist. Ein Behavior ist ein spezielles Element des Szenegraphen, das für Ereignisse bei Java3D registriert werden kann und bei Eintreten dieser Ereignisse über eine spezielle Methode notifiziert wird. Abschnitt 5.2.1 beschreibt den Aufbau und das Konzept eines Behaviors. Anschließend wird in Abschnitt 5.2.2 das *ViewPlatformAWTBehavior* vorgestellt, dass eine Spezialisierung des Behaviors ist und Unterstützung bietet für die Verarbeitung von Ereignissen, die durch die Tastatur oder die Maus ausgelöst werden.

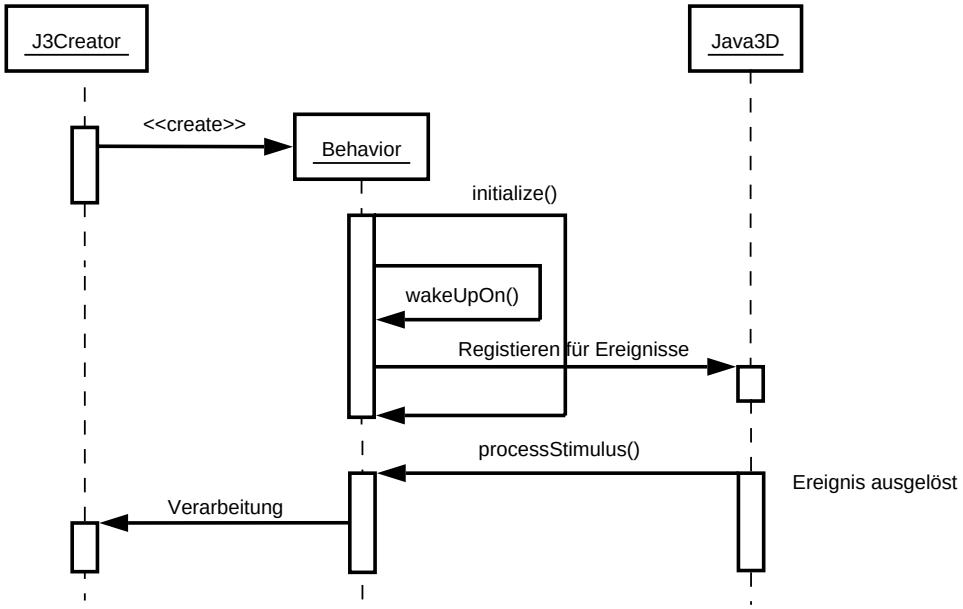
5.2.1. Aufbau und Benutzung eines Behaviors

Ein Behavior ist eine abstrakte Klasse, das wie ein Shape3D ein Blatt des Szenegraphen ist. Erbende Klassen müssen die Methoden `initialize()` und `processStimulus()` erfüllen.

Die Initialisierungsmethode dient dazu, das Behavior für bestimmte Ereignisse bei Java3D mittels Nutzung der Methode `wakeUpOn()` zu registrieren. Mögliche Ereignisse sind unter anderem das Drücken einer Taste, die Benutzung der Maus, die grafische Kollision zweier Objekte oder der Eintritt eines Elements in den Sichtbereich. Wird eines der registrierten Ereignisse ausgelöst, wird das Behavior mittels Aufruf der Methode `processStimulus()`

benachrichtigt, in der bestimmt wird, wie das Behavior auf das Ereignis reagieren soll. Abbildung 5.2 veranschaulicht das Vorgehen anhand eines Sequenzdiagramms.

Abbildung 5.2. Sequenzdiagramm eines Behaviors



(J3Creator und Java3D sind der Einfachheit halber als einfache Objekte dargestellt)

Hat das Behavior die Ereignisse verarbeitet und soll weiterhin über sie benachrichtigt werden, muss es sich abschließend erneut registrieren. Dies ermöglicht durch Unterlassen der Neuregistrierung ein Behavior, das nur einmal auf ein Ereignis reagiert.

5.2.2. Das *ViewPlatformAWTBehavior*

Das *ViewPlatformAWTBehavior* ist eine von Java3D zur Verfügung gestellte Spezialisierung des Behaviors. Wie das Behavior ist es eine abstrakte Klasse und bietet besondere Unterstützung für die Interaktion sowohl mit dem *view branch graph* als auch mit dem Benutzer.

Zur Interaktion mit dem *view branch graph* besitzt eine Referenz auf dessen TransformGroup, so dass einfach die in ihr enthaltene Transform3D ausgelesen und gesetzt werden kann, um die Position und Blickrichtung zu vermerken bzw. zu ändern. Da die TransformGroup prinzipiell auch von anderen Teilen des Szenegraphen modifiziert werden kann, wird die Methode `integrateTransforms()` zu Verfügung gestellt, mit deren Aufruf signalisiert wird, dass eventuell lokal gehaltene Daten aktualisiert werden müssen.

Um das Behavior nicht manuell nach jedem Ereignis der Tastatur oder der Maus neu zu registrieren, wie es eigentlich nötig wäre, erfüllt das *ViewPlat-*

formAWTBehavior bereits die im vorigen Abschnitt vorgestellten abstrakten Methoden des Behaviors. Es sammelt in `processStimulus()` die Ereignisse und registriert sich anschließend automatisch neu für sie. An eine erbende Klasse werden die gesammelten Ereignisse mit Hilfe der abstrakten Methode `processAWTEvents()` weitergegeben. Sie muss dann entscheiden, ob und wie die TransformGroup des *view branch graph* verändert werden soll.

5.3. Realisierung der Navigation

Zur Realisierung der Navigation wird das *ViewPlatformAWTBehavior* erweitert. Dabei soll ein Tastendruck auf eine Pfeil-Taste den Betrachter entsprechend vorwärts, rückwärts oder seitwärts bewegen. Durch betätigen der Bildlauf-Tasten „BildAuf" und „BildAb" kann die Position in der Höhe versetzt werden. Wird bei den genannten Tasten zusätzlich die STRG-Taste gedrückt, soll stattdessen die Blickrichtung verändert werden. In Kombination mit den Pfeil-Tasten senkt oder hebt sich der Blickwinkel oder dreht sich zur Seite. Die Bildlauf-Tasten dienen hier dazu, die Darstellung entlang der Sichtachse zu rotieren. Dabei wird die TransformGroup bei jedem Tastendruck nur minimal verändert, so dass das Gedrückthalten der jeweiligen Taste zu einer flüssigen und nachvollziehbaren Bewegung führt.

Um die Navigation auf die beschriebene Weise mit Hilfe eines *ViewPlatformAWTBehaviors* umzusetzen, muss die TransformGroup des *view branch graphs* entsprechend verändert werden. Dabei muss stets die aktuelle Position und Blickrichtung des Betrachters berücksichtigt. Im folgenden werden die dafür nötigen Berechnung beschrieben.

Änderung der Position

Um den Betrachter in eine bestimmte Richtung zu bewegen, muss die Bewegung (vorwärts, rückwärts, links, rechts) auf die momentane Blickrichtung projiziert werden. Der folgende Java Quellcode 5.3.1 beschreibt die nötigen Anweisungen für eine Bewegung vorwärts.

Die Variable `move` definiert mittels des Vektors eine Bewegung vorwärts, die jedoch nur dann in die gewünschte Richtung führt, wenn der Betrachter in Richtung der z-Achse blickt. Zur Projektion wird von der TransformGroup des *view branch graphs* die Transform3D angefordert und in `vpTransform3D` abgelegt, aus der wiederum die Blickrichtung in Form von Rotationsangaben extrahiert und in der Matrix `sight` gespeichert werden. Mit Hilfe dieser Matrix wird `move` so transformiert, dass er eine Bewegung vorwärts in Blickrichtung repräsentiert. Diese muss abschließend zur aktuellen Position addiert werden, um die endgültige Position zu erreichen.

Java Quellcode 5.3.1 Berechnung der Änderung der Position

```
TransformGroup vpTG = ...              // TransformGroup des view branch graph
Transform3D vpTransform3D = new Transform3D();
vpTG.getTransform(vpTransform3D);      // aktuelle Transform3D anfordern

Vector3f move = new Vector3f(0,0,-0.1f);  // Bewegung vorwaerts
Matrix3f sight = new Matrix3f();
vpTransform3D.get(sight);              // Blickrichtung der aktuellen Sicht in
                                          Matrix sight ablegen
sight.transform(move);                 // Transformation der Bewegung
                                          anhand der Blickrichtung

Vector3f position = new Vecor3f();
vpTransform3D.get(position);           // Position der ViewPlatform anfordern
position.add(move);                    // Bewegung zur Position addieren

vpTransform3D.setTranslation(position);
vpTG.setTransform(vpTransform3D);      // neue Positionü bertragen
```

Änderung der Blickrichtung

Das Drehen der Visualisierung erfordert im Gegensatz zur Bewegung keine aufwendige Projektion, wie Java Quellcode 5.3.2 zeigt, der eine Rotation des Blickwinkels in x-Richtung beschreibt.

Java Quellcode 5.3.2 Berechnung der Änderung der Blickrichtung

```
TransformGroup vpTG = ...              // TransformGroup des view branch graph
Transform3D vpTransform3D = new Transform3D();
vpTG.getTransform(vpTransform3D);      // aktuelle Transform3D anfordern

Transform3D rotate = new Transform3D();
rotate.rotX(Math.PI/180);              // Rotation um 1 Grad um x-Achsen
vpTransform3D.mul(rotate);             // vorherige Rotationen werden durch
                                       // mul-Operation beruecksichtigt

vpTG.setTransform(vpTransform3D);      // neue Blickrichtung uebertragen
```

Für die Änderung der Blickrichtung wird die Variable `rotate` genutzt. Mit Hilfe der Methode `rotX()` wird die Rotation in x-Richtung gesetzt. Durch Multiplikation mit der bisherigen Blickrichtung, wird anschließend der Blickwinkel modifiziert.

6. Realisierung einfacher Modifikationsmöglichkeiten

Dieses Kapitel behandelt Möglichkeiten, die Visualisierung zu manipulieren, ohne jedoch ihre semantische Bedeutung und damit den Quellcode zu verändern. Der Benutzer wird in die Lage versetzt, die Anordnung in der Visualisierung zu verändern, solange die Veränderung die Semantik erhält. Weiterhin kann er die Farbe einzelner Elemente zum Zweck der Markierung oder Gruppierung verändern, sofern die gewünschte Farbe nicht schon semantisch belegt ist.

Um einzelne Elemente verändern zu können, muss der J3Creator zuerst vom Benutzer erfahren, welches Element ausgewählt worden ist. Dieser Vorgang wird *Picking* genannt und näher in Kapitel 6.1 erläutert. Kapitel 6.2 beschreibt anschließend, wie der J3Creator die einfachen Modifikationsmöglichkeiten durchführt.

6.1. Picking

6.1.1. Funktionsweise des Pickings

Ausgehend von einem Mausklick in der zweidimensionalen Projektion müssen zunächst die Koordinaten dieses ausgewählten Punktes in der virtuellen dreidimensionalen Welt berechnet werden. Grundlagen dieser Berechnung sind außer der Position des Klicks noch die virtuelle Position des Betrachters, seine Blickrichtung und die Größe des Bildausschnitts. Von diesem Punkt aus wird anschließend ein Strahl entlang der Perspektive zum Fluchtpunkt konstruiert. Jedes Element der Visualisierung, das diesen Strahl kreuzt, ist ein potentieller Kandidat des Pickings. Die konkrete Auswahl hängt dabei vom gewünschten Kriterium des Pickings ab, das eines der Folgenden sein kann:

- Das dem Betrachter nächste Element
- Alle Elemente sortiert nach ihrer Entfernung vom Betrachter
- Alle Elemente in beliebiger Reihenfolge
- Irgendein Element

6.1.2. Picking in Java3D

In Java3D gibt es zwei Arten, Picking zu realisieren: das Programm realisiert selbst den Vorgang, wie er im vorigen Kapitel 6.1.1 beschrieben ist, oder nutzt den *PickCanvas* aus dem *utility*-Paket.

Wird das Picking vom Programm selbst realisiert, kann vom Canvas3D des *view branch graphs* eine Transform3D angefordert werden, die zur Umrechnung eines Punktes im Canvas3D in einen Punkt der visualisierten Szene benutzt werden kann. Der Mittelpunkt des Canvas3D stellt den Fluchtpunkt dar, zu der ebenfalls seine dreidimensionalen Koordinaten berechnet werden müssen, um den Vektor des Strahles zur Auswahl der Elemente zu erhalten. Dieser Vektor dient einem *PickRay* als Berechnungsgrundlage, der zur eigentlichen Berechnung der getroffenen Elemente einem *Locale* übergeben wird. Als Ergebnis erhält man je nach gewähltem Auswahlkriterium einen bzw. mehrere *SceneGraphPaths*, der den Pfad vom Locale zum Knoten, der das getroffene Element darstellt, repräsentiert. Dieser Knoten ist in der Regel ein Shape3D. Da die Berechnung dieses Pfades sehr rechenintensiv ist, werden nur Knoten berücksichtigt, die die Fähigkeit `ENABLE_PICK_REPORTING` besitzen. Standardmäßig besteht der *SceneGraphPath* daher nur aus dem Locale und dem ausgewählten Knoten.

Da das Vorgehen des Pickings stets gleich ist, gibt es im *utility*-Paket von Java3D mit dem *PickCanvas* eine Hilfsklasse, die die Berechnung automatisch ausführt. Ausgehend von den Koordinaten des Klicks im Canvas3D liefert er direkt einen oder mehrere *PickResults*, der einen Treffer des imaginären Strahls repräsentiert. Es enthält den *SceneGraphPath* und darüber hinaus Informationen, wo ein Element getroffen worden ist. Bei einem Würfel kann es beispielsweise Auskunft darüber geben, welche Seiten der Strahl durchkreuzt hat.

6.1.3. Nutzung des Pickings im J3Creator

Der J3Creator verlässt sich bei der Berechnung ausgewählter Elemente auf das *PickCanvas*, anstatt die Berechnung selbst vorzunehmen. Dabei sind weniger die Shape3Ds von Interesse, sondern vielmehr diejenigen BranchGroups, die zum J3DObject erweitert worden sind. Daher erhält jedes J3DObject die Fähigkeit `ENABLE_PICK_REPORTING`, um so beim Picking im *SceneGraphPath* enthalten zu sein. Die zusätzlichen Details, die das *PickCanvas* mit dem *PickResult* zur Verfügung stellt, werden nicht weiter berücksichtigt.

Als Auswahlkriterium möglicher Kandidaten kann grundsätzlich das dem Benutzer nächste Element gewählt werden. Problematisch sind dabei allerdings *Information Cubes*. Da sie in ihrem Inneren weitere Elemente enthalten, ist es möglich, dass der Strahl ebenfalls eines der enthaltenen Komponenten kreuzt. Aus diesem Grund müssen stets alle getroffenen Elemente sortiert nach ihrer Entfernung berücksichtigt werden, die wie folgt algorithmisch betrachtet werden:

1. Betrachte das erste Element `e`.
2. Ist `e` ein J3DPackage und existieren weitere Elemente? Dann betrachte ebenfalls das auf `e` folgende Element `e'`.
3. Falls `e'` ebenfalls ein J3DPackage ist, ist `e` das ausgewählte Element.
4. Falls `e'` ein J3DType oder eine J3DConnection ist, ist `e'` das ausgewählte Element.

6.2. Umsetzung der einfachen Modifikationsmöglichkeiten

Ist ein Element mit Hilfe des *Pickings* ausgewählt, kann es grafisch verschoben werden oder eine neue Farbe erhalten.

Um ein Element zu verschieben, wird es mit Hilfe der Maus bei gedrückter linker Maustaste an seine neue gewünschte Position gezogen. Zu diesem Zweck wird J3DObject um eine Methode `move(float x, float y, float z)` erweitert. Sie wird mit den `x`- und `y`-Werten aufgerufen, die sich der Mauszeiger bewegt hat. Auf diese erweckt das gezogene grafische Objekt den Eindruck, dass es dem Mauszeiger folgt. Da die Maus nur in zwei Richtungen bewegt werden kann, bleibt die `z`-Koordinate fest, wodurch ein Element nicht in der Tiefe verschoben werden kann. Dadurch erhält der Parameter `z` immer den Wert `0`, ist aber der Vollständigkeit halber in der Methode deklariert.

Bei der Änderung der Farbe ist zu beachten, dass sie nicht bereits semantisch belegt ist. Das ist dann der Fall, wenn eines der im vorigen Kapitel definierten grafischen Objekte mit der gewünschten Farbe definiert worden ist. Diese Einschränkung stellt sicher, dass nach der Änderung beispielsweise eine Assozation nicht fälschlicherweise wie Vererbungsbeziehung aussieht, und so vom Benutzer verwechselt wird. Entsprechend Kapitel 4.2.1 dürfen also die Farben Blau, Rot, Orange, Gelb, Grau und Schwarz nicht ausgewählt werden. Zusätzlich müssen auch ihnen ähnliche Farben ausgeschlossen werden, die Gefahr der Verwechslung zu minimieren. Farben werden in Java3D über ein Dreiertupel `(r,g,b)`definiert, deren einzelne Parameter den Rot-, den Grün- und den Blauwert der Farbe angeben. Der Wert jedes Parameters kann sich im Bereich [0;1] bewegen, wobei der Wert 0 bedeutet, dass der betrachtete Wert nicht vorkommt, und der Wert 1 aussagt, dass er mit voller Intensität in die Farbe einfließt. Der Wert (0,0,0) steht demnach für die Farbe Schwarz, wohingegen (1,1,1) weiß bedeutet. Um also bestimmte Farben aus der Wahl auszuschließen, müssten nicht nur die exakten Dreiertupel verboten werden, sondern auch ihnen ähnliche. Der Einfachheit halber werden hier allerdings verschiedene Zielfarben für die Änderung vorgegeben, für die eine Verwechslung ausgeschlossen werden kann. Diese sind ohne besondere Gründe:

- Grün

- Magenta
- Cyan
- Braun
- Violett

J3DObject wird zu diesem Zweck um eine Methode `setColor(float r, float g, float b)` erweitert, der als Parameter die Rot-, Grün- und Blauwerte als Parameter übergeben werden.

7. Überprüfung der Modifikation

Wird die dargestellte Szene verändert, stellt sich die Frage, ob sie weiterhin „korrekt" ist. Dies kann nicht allein durch die Visualisierung entschieden werden, sondern folgt vielmehr aus ihrer Zielsetzung - die Darstellung von Java-Strukturen, die implizit gewisse Gültigkeitsbedingungen an die Darstellung knüpft.

Abschnitt 7.1 beschreibt grob, was zu tun ist, damit die Visualisierung überprüft werden kann. Abschließend werden dort die Bedingungen festgelegt, die die Visualisierung des J3Creators erfüllen muss. Anschließend werden in 7.2 Möglichkeiten zur Überprüfung vorgestellt, aus denen RELVIEW bzw. KURE ausgewählt wird. Sie sind Systeme zur Auswertung von relationenalgebraischen Programmen und Formeln und werden in 7.3 vorgestellt werden. In Abschnitt 7.4 werden Mengen und Relationen definiert, die die Basis für die Überprüfung bilden. Schrittweise werden dort ebenfalls die in 7.1 vorgestellten Bedingungen in relationenalgebraische Terme überführt. Abschließend wird in Abschnitt 7.5 beschrieben, wie die Visualisierung in die vorher definierten Relationen übertragen wird. Damit ist KURE dann in der Lage, die Visualisierung zu überprüfen und zu entscheiden, ob eine modifizierte Visualisierung weiterhin gültig ist.

7.1. Überprüfung

Um die Korrektheit der Visualisierung zu entscheiden, muss ihre Syntax so definiert werden, dass sie die dargestellten Java-Strukturen widerspiegelt. Dies fängt bereits bei der Wahl der grafischen Repräsentation der Java-Elemente Paket, Klasse und Schnittstelle an, durch die festgelegt ist, dass nur genau diese Elemente durch die gewählten Symbole dargestellt werden dürfen. Eine Klasse darf dementsprechend nicht durch eine Kugel (das Symbol für eine Schnittstelle) verkörpert werden.

Für visualisierte Beziehungen muss auf Basis der Symbole definiert werden, welche Symbolarten miteinander verbunden werden dürfen, um gültige Beziehungen in Java zu beschreiben. Um eine Implementierungs-Beziehung zu erhalten, müssen beispielsweise eine Kugel und ein Würfel miteinander verknüpft sein.

Neben der Überprüfung der syntaktischen Regeln müssen ebenfalls strukturelle Eigenschaften kontrolliert werden. Dazu werden die Elemente und Beziehungen nicht mehr einzeln, sondern zusammenhängend in einer Menge

betrachtet. Diese Menge muss gewissen Merkmale aufweisen, damit die Visualisierung weiter als gültig angesehen werden kann. So darf beispielsweise eine Klasse durch Vererbungen nicht durch eine zyklische Abhängigkeit indirekt von sich selbst erben. Entsprechend darf die Menge der in der Visualisierung dargestellten Vererbungsbeziehungen keinen Kreis bilden.

Eine Sonderrolle in den visualisierten Beziehungen kommt der Paketzugehörigkeit zu, da diese Beziehung als Einzige nicht durch Röhren visualisiert wird, sondern sich durch die Positionierung innerhalb eines *Information Cubes* ausdrückt. Sie kann deshalb nicht über miteinander verbundene Symbolarten definiert werden, sondern muss anhand von Positionen und Symbol-Ausmaßen entschieden werden.

Die Bedingungen, die an die Visualisierung geknüpft sind, basieren auf den syntaktischen Regeln, die beliebiger Quellcode einhalten muss, um syntaktisch korrekter Java-Quellcode zu sein. Werden diese Regeln auf die grafischen Elemente übertragen, die in Kapitel chap:java3d für die Visualisierung von Paketen, Klassen, Schnittstellen und ihren Beziehungen gewählt worden sind, ergeben sich folgende Bedingungen:

1. Überprüfung der syntaktischen Korrektheit der Beziehungen:
   a) Eine **Vererbungsbeziehung** muss entweder einen Würfel mit einem Würfel oder eine Kugel mit einer Kugel verbinden:
   b) Eine **Implementierungsbeziehung** darf nur von einem Würfel zu einer Kugel führen:
   c) Eine **Paketbeziehung** darf nur *Information Cubes* miteinander verbinden:
   d) **Assoziationen und Benutztbeziehungen** dürfen weder als Quelle noch Ziel der Beziehung einen *Information Cubes* besitzen.
2. Für jeden Würfel b in einem *Information Cube* c muss die von b repräsentierte Klasse $C(b)$ in dem Paket enthalten sein, das durch c dargestellt wird. Dasselbe gilt analog für Kugeln und Schnittstellen.
3. Jeder Würfel und jede Kugel befindet sich in genau einem *Information Cube*.
4. Die Beziehungen für Vererbung und Implementierung dürfen keine Kreise bilden.
5. Die Package-, die Verberbungs- und die Implementierungs-Beziehung dürfen kein Element mit sich selbst verbinden.
6. Ein Paket darf höchstens ein übergeordnetes Paket besitzen.

Werden diese Bedingungen verletzt, ist die Visualisierung aus Sicht des Java-Quellcodes ungültig. Jede Übertragung der Änderungen in der Visualisierung

auf den Quellcode führt dann dazu, dass er ebenfalls ungültig - also syntaktisch nicht korrekt - ist. Diese Bedingungen stellen allerdings nicht das gesamte Regelwerk dar, das Java-Quellcode erfüllen muss. Vielmehr werden hier grundlegende Eigenschaften überprüft.

Die bis zu diesem Schritt realisierten einfachen Modifikationen wirken sich zwar nicht auf den Quellcode aus, doch kann schon das Verschieben eines Elements schon eine inkorrekte Visualisierung bewirken. Verschiebt man beispielsweise einen *Information Cube*, so kann es sein, dass sich anschließend zwei *Information Cubes* überschneiden, wodurch Bedingung 2 verletzt werden könnte. Befänden sich im Schnittbereich Würfel oder Kugeln, würden sie laut Visualisierung zu beiden *Information Cubes* gehören. Da eine Klasse bzw. Schnittstelle aber nur in einem Paket sein kann, würde in einem Fall nicht wie in der Bedingung gefordert die Klasse bzw. Schnittstelle nicht zu dem Paket gehören, dass durch den *Information Cube* dargestellt wird, in dem sich der betrachtete Würfel bzw. die betrachete Kugel befindet.

7.2. Möglichkeiten zur Überprüfung

Nachdem die Bedingungen, die die Visualisierung erfüllen muss, formuliert sind, muss nun entschieden werden, wie die Gültigkeit überprüft werden kann. In der naheliegendsten Lösung überprüft der J3Creator die Modifikation in seiner Programmlogik. Abhängig von Modifikationsoperation (Verschieben, Hinzufügen, Löschen, usw.) und ausgewähltem Element müssen dann die Bedingungen berücksichtigt werden, die möglicherweise verletzt werden könnten, wie Abbildung 7.2.1 anhand eines beispielhaften Pseudocodes darstellt.

Dieses Vorgehen bedeutet allerdings, dass jede mögliche Kombination von Operationen und ausgewählten Elementen im J3Creator berücksichtigt werden muss. Darüber hinaus muss bei Hinzufügen neuer Operationen oder Visualisierungselemente die Abfrage um alle möglichen neuen Kombinationsmöglichkeiten erweitert werden, in denen wiederum einzeln die Bedingungen abgefragt werden müssen. Dieses Vorgehen ist sehr aufwändig, fehlerhaft, da leicht in der Menge der Abfragen eine Einzelne vergessen werden kann, und besonders im Hinblick auf Erweiterungen schlecht wartbar, da unter Umständen auch die schon bestehenden Abfragen verändert oder ergänzt werden müssen.

Ein alternativer Lösungsweg ist die Verwendung von relationenalgebraischen Formeln, wie sie in [BF03] und [AF02] vorgeschlagen werden. Sie ermöglichen eine dynamische Überprüfung der Visualisierung, anstatt das Vorgehen wie im ersten Lösungsansatz fest in den J3Creator einzuprogammieren. Dies ist besonders im Hinblick auf Wartbarkeit und Erweiterung der Prüfung vorteilhaft.

Bei diesem Vorgehen wird von der Visualisierung selbst abstrahiert. Stattdessen werden nur die Beziehungen betrachtet, die die Elemente der Visualisierung zueinander haben, und als Relationen dargestellt.

Java Quellcode 7.2.1 beispielhafter Pseudocode als „schlechtes" Beispiel zur Überprüfung der Modifikation

```
if (operation == hinzufuegen){
   ...
}
if (operation == verschieben){
   if (element == information_cube){
      pruefe: Information Cube schneidet keinen anderen
                  Information Cube
   }
   if (element == wuerfel || element == kugel){
      ist element in einem Information Cube?
      wenn nein:
         Operation nicht erlaubt, da keine Paketzugehoerigkeit mehr
      wenn ja:
         pruefe: element vollstaendig enthalten (zur Eindeutigkeit)
         pruefe: element schneidet keine anderen Elemente im
                     Information Cube
   }
}
...
```

Eine Relation ist eine Teilmenge des kartesischen Produktes $X \times Y$ auf Mengen X und Y, notiert als $R : X \leftrightarrow Y$. Ist ein Tupel (x,y) Element der Relation R, so schreibt man $R_{x,y}$ statt $(x,y) \in R$. Die Menge aller $x \in X$, für die ein $y \in Y$ existiert mit $R_{x,y}$, heißt Definitionsbereich. Umgekehrt wird die Menge aller $y \in Y$ für die ein $x \in X$ existiert als Bildbereich bezeichnet.

Im J3Creator wird jede Beziehungsart (Implementierung, Assoziation, usw.) durch eine Relation repräsentiert, deren Definitions- und Bildbereich aus sichtbaren *Information Cubes*, Würfeln oder Kugeln besteht. Dabei gilt genau dann $R_{x,y}$ für zwei Elemente dieser Menge, wenn es in der Visualisierung eine sichtbare Beziehung vom Typ, die R repräsentiert, von x zu y gibt. Falls die Beziehung bidirektional ist, gilt ebenfalls $R_{y,x}$. Auf diese Weise kann die gesamte Visualisierung durch eine Menge von Relationen charakterisiert werden.

Um die Visualisierung zu überprüfen, müssen noch die Gültigkeitsbedingungen in relationenalgebraische Ausdrücke überführt werden. Diese Ausdrücke überprüfen Relationen auf bestimmte relationale Eigenschaften, die den geforderten Eigenschaften in den Gültigkeitsbedingungen entsprechen. Wird beispielsweise verlangt, dass kein visualisiertes Element eine Beziehung zu sich selbst besitzen darf, so muß die zugehörige Relation irreflexiv sein. Erfüllt eine Relation eine geforderte Eigenschaft nicht, gibt es eine visualisierte Beziehung, die die betrachtete Gültigkeitsbedingung verletzt.

7.3. RELVIEW /KURE

RELVIEW [Rel] ist ein interaktives System zur Manipulation von Relationen, das diese ebenfalls effizient analysieren kann. Da es durch eine grafische Oberfläche gesteuert wird und in C geschrieben ist, ist es in dieser Form zur Nutzung durch ein Java-Programm unbrauchbar.

Im Rahmen einer Diplomarbeit [Szy03] ist deshalb der sogenannte funktionale Kern von RELVIEW als Java-Bibliothek in Form von KURE [KUR] zugänglich gemacht worden. Zum funktionalen Kern gehören diejenigen Funktionen von RELVIEW , die weder Komponenten der grafischen Oberfläche bilden noch diese ansprechen.

KURE übernimmt für ein Programm zwei Aufgaben: erstens, es verwaltet die Anbindung an den Kern und stellt zweitens die Funktionalität von RELVIEW objektorientiert und um eine Fehlerbehandlung mittels Java-*Exceptions* erweitert zur Verfügung. Relationen werden in einem *RelManager* verwaltet, der darüber hinaus Möglichkeiten bietet, relationale Funktionen zu deklarieren und relationale Terme auszuwerten, die hier als relationenalgebraische Ausdrücke eingeführt worden sind. Eine Relation selbst wird durch ein Objekt des Typs *Relation* repräsentiert, das Methoden zum Setzen und Löschen von Einträgen und zur Abfrage von Informationen zur Verfügung stellt. RELVIEW und KURE betrachten dabei eine Relation als eine binäre Boole'sche Matrix mit festen, endlichen Dimensionen. Soll die Relation $R : X \leftrightarrow Y$ durch ein *Relation*-Objekt dargestellt werden und sind $\#X$ bzw. $\#Y$ die Kardinalitäten der Mengen X bzw. Y, so besitzt die zugehörige Matrix $\#X$ Zeilen und $\#Y$ Spalten. Die Zugehörigkeit eines Tupels (x,y) zu R kann anschließend entschieden werden, indem der Boole'sche Wert in Zeile x und Spalte y betrachtet wird. Besitzt dieser Eintrag den Wert `true`, gilt $R_{x,y}$. Alternativ bietet die *Relation*-Klasse die Möglichkeit, die Einträge der Matrix über eine Menge von Java-Objekten zu identifizieren, und erspart so die Abbildung auf die Zeilen- und Spaltenindizes der Matrix. Dazu muss ihr über die Methode `setObjects` jeweils ein *Array* von Java-Objekten für Definions- und Bildbereich übergeben werden. Anschließend können alle Einträge der Relation durch Angabe der jeweiligen Java-Objekte verwaltet werden.

7.4. Überprüfung mit Hilfe von KURE

Um die Visualisierung mit Hilfe von KURE auf ihre Gültigkeit zu überprüfen, müssen neben den visualisierten Beziehungen auch die Gültigkeitsbedingungen in relationenalgebraischer Form vorliegen. Dazu müssen die Gültigkeitsbedingungen als prädikatenlogische Terme formalisiert werden, aus denen anschließend relationenalgebraische Formeln abgeleitet werden können. Da die relationenalgebraische Notation Zeichen enthält, die in einem normalen ASCII-basierten Quelltext nicht darstellbar sind, müssen diese durch Symbole ersetzt werden, damit sie von KURE eingelesen und verarbeitet werden können. Darüber hinaus müssen konstante Relationen wie die Identitätsrelation und bestimmte Tests wie die Probe auf Gleichheit durch spezielle Funktionen ersetzt werden.

Die in diesem Kapitel vorgestellten Mengen, Relationen und Formeln sind angelehnt an [BF03], [AF02] und den KURE -Prototyp [Szy03] und sollen keine erschöpfende Überprüfung sicherstellen. Vielmehr soll anhand von einigen

ausgewählten Bedingungen gezeigt werden, auf welche Weise die Modifikation einer Visualisierung mit Hilfe von Relationenalgebra kontrolliert werden kann. Dies kann in anderen Arbeiten als Grundlage für weitere Überlegungen und Formeln dienen.

7.4.1. Grundlagen

Als Basis für die prädikatenlogische Formalisierung und die relationalgebraischen Terme werden zunächst folgende disjunkte Mengen definiert. Sei:

$CUBE$	die Menge aller *Information Cubes*,
BOX	die Menge aller Würfel,
$SPHERE$	die Menge aller Kugeln,
$PACK$	die Menge aller Pakete des betrachteten Quellcodes,
$CLASS$	die Menge aller Klassen des betrachteten Quellcodes,
$INTFACE$	die Menge aller *Interfaces* des betrachteten Quellcodes.

$CUBE$, BOX und $SPHERE$ werden zusammengefasst zu der Menge aller Entitäten, die definiert ist als:

$$ENTITY := CUBE \cup BOX \cup SPHERE.$$

Für jede Entitäts-Art wird weiterhin in einem Vektor festgehalten, ob sie in der Visualisierung sichtbar ist. Ein Vektor ist eine Relation $v : X \leftrightarrow \mathbb{1}$ einer Menge X auf die einelementige Menge $\mathbb{1} := \{\diamond\}$. v repräsentiert einen Spaltenvektor und beschreibt auf diese Weise die Menge $\{x \in X | (x, \diamond) \in v\}$. Im Weiteren wird $(x, \diamond) \in v$ durch v_x ausgedrückt. Sei demnach:

$vC : CUBE \leftrightarrow \mathbb{1}$	die Menge aller sichtbaren *Information Cubes*,
$vB : BOX \leftrightarrow \mathbb{1}$	die Menge aller sichtbaren Würfel,
$vS : SPHERE \leftrightarrow \mathbb{1}$	die Menge aller sichtbaren Kugeln.

Dabei gilt vC_e, vB_e bzw. vS_e für eine Entität e genau dann, wenn e ein *Information Cube*, ein Würfel bzw. eine Kugel und zudem sichtbar ist.

Analog zu $ENTITY$ werden vC, vB und vS in der Menge aller sichtbaren Entitäten zusammengefasst:

$$vENTITY := dom(vC) \cup dom(vB) \cup dom(vS),$$

wobei $dom(R) := \{x | R_x\}$ den Domain der Relation R symbolisiert.

Anhand dieser Mengen werden anschließend die visualisierten Beziehungen durch Relationen ausgedrückt, deren Definitions- und Wertebereich jeweils $ENTITY$ ist, um syntaktisch falsch visualisierte Beziehungen identifizieren

zu können. Jede der folgenden Relationen, deren Namen an die englische Bezeichnung der Beziehungsart angelehnt ist, ist also eine Teilmenge von $ENTITY \times ENTITY$. Sei:

$Assoc$	die Menge aller Assoziationen,
$Impl$	die Menge aller Implementierungsbeziehungen,
$Uses$	die Menge aller Benutztbeziehungen,
Ext	die Menge aller Erweiterungsbeziehungen,
$Pckg$	die Menge aller Unterpaketbeziehungen.

Ein Paar von Entitäten (e, e') gehört genau dann zu einer der obigen Relationen, wenn eine visualisierte Beziehung von e nach e' existiert. Im Einzelnen gilt:

$Assoc_{e,e'}$	$\Leftrightarrow$	e besitzt eine Assoziationsbeziehung zu e'
$Impl_{e,e'}$	$\Leftrightarrow$	e implementiert e'
$Uses_{e,e'}$	$\Leftrightarrow$	e besitzt eine Benutztbeziehung zu e'
$Ext_{e,e'}$	$\Leftrightarrow$	e erweitert e'
$Pckg_{e,e'}$	$\Leftrightarrow$	e ist Unterpaket von e'

Nachdem Mengen und Relationen definiert worden sind, die die Visualisierung widerspiegeln, müssen die Elemente der Visualisierung dem Quellcode zugeordnet werden können. Dazu dienen folgende Relationen:

$P : CUBE \leftrightarrow PACK$	Zuordnung *Information Cubes* zu Paketen
$C : BOX \leftrightarrow CLASS$	Zuordnung Würfel zu Klassen
$I : SPHERE \leftrightarrow INTFACE$	Zuordnung Kugeln zu *Interfaces*

Dabei gilt:

$P_{ic,p}$	$\Leftrightarrow$	der *Information Cube ic* repräsentiert das Paket p
$C_{b,c}$	$\Leftrightarrow$	der Würfel b repräsentiert die Klasse c
$I_{s,i}$	$\Leftrightarrow$	die Kugel s repräsentiert das *Interface i*

Für diese Relationen muss zusätzlich gelten, dass die Entitäten des Domain sichtbar sind, da nur sichtbare Elemente dem Quellcode zugeordnet werden können. Es gilt also zusätzlich:

$$\begin{aligned} \text{für } P_{ic,p} &: P_{ic,p} \rightarrow vC_{ic} \\ \text{für } C_{b,c} &: C_{b,c} \rightarrow vB_b \\ \text{für } S_{s,i} &: S_{s,i} \rightarrow vS_s \end{aligned}$$

Diese Relationen ordnen einer Entität genau ein Element des Quellcodes zu und sind somit rechtseindeutig. Aus diesem Grund werden sie im Weiteren als Funktionen betrachtet und in der dafür üblichen Schreibweise benutzt. $C(b)$

stellt dementsprechend die Java-Klasse dar, die durch den Würfel b repräsentiert wird.

Darüber hinaus werden noch folgende Hilfsrelationen definiert:

$cM : CUBE \leftrightarrow CLASS \cup INTFACE$ Zugehörigkeit zu *Information Cubes*
$in : ENTITY \leftrightarrow ENTITY$ Inhaltsbeziehung von Entitäten

cM ordnet einem *Information Cube* diejenigen Elemente des Java-Quellcodes zu, die durch ihn bzw. in ihm dargestellt werden. Es gilt $cM_{ic,c}$ bzw. $cM_{ic,i}$ genau dann, wenn die Entität in Form eines Würfels bzw. einer Kugel, die die Klasse c bzw. das *Interface* i repräsentiert, Teil des *Information Cubes* ic ist. in drückt aus, ob sich eine Entität in einer anderen Entität befindet. $in_{e,e'}$ bedeutet demnach, dass e innerhalb von e' liegt.

7.4.2. Prädikatenlogische Form

Mit diesen Mengen und Relationen als Grundlage können nun die Gültigkeitsbedingungen durch prädikatenlogische Formeln ausgedrückt werden. Zur Übersicht werden in der folgenden Aufzählung die textuellen Gültigkeitsbedingungen vom Anfang des Kapitels (7.1) wiederholt :

1. Überprüfung der syntaktischen Korrektheit der Beziehungen:

   a) Eine **Vererbungsbeziehung** muss entweder einen Würfel mit einem Würfel oder eine Kugel mit einer Kugel verbinden:

   $$\forall e, e' : Ext_{e,e'} \rightarrow (vB_e \wedge vB_{e'}) \vee (vS_e \wedge vS_{e'})$$

   b) Eine **Implementierungsbeziehung** darf nur von einem Würfel zu einer Kugel führen:

   $$\forall e, e' : Impl_{e,e'} \rightarrow vB_e \wedge vS_{e'}$$

   c) Eine **Paketbeziehung** darf nur *Information Cubes* miteinander verbinden:

   $$\forall e, e' : Pckg_{e,e'} \rightarrow vC_e \wedge vC_{e'}$$

   d) **Assoziationen und Benutztbeziehungen** dürfen weder als Quelle noch Ziel der Beziehung einen *Information Cubes* besitzen. Es gilt

   $$\forall e, e' : Assoc_{e,e'} \cup Uses_{e,e'} \rightarrow \{e, e'\} \subseteq dom(vB) \cup dom(vS)$$

2. Definition der *in*-Relation: Für jeden Würfel b in einem *Information Cube* c muss die von b repräsentierte Klasse $C(b)$ in dem Paket enthalten sein, das durch c dargestellt wird. Dasselbe gilt analog für Kugeln und *Interfaces*. Es gilt

   $$\forall e, c : in_{e,c} \leftrightarrow cM_{c,C(e)} \vee cM_{c,I(e)}$$

3. Definition der *cM*-Relation: Jeder Würfel und jede Kugel befindet sich in genau einem *Information Cube*. Es gilt

$$\forall b : |\{c : cM_{c,C(b)}\}| = 1$$

für Würfel und

$$\forall s : |\{c : cM_{c,I(s)}\}| = 1$$

für Kugeln.

4. Die Beziehungen für Vererbung und Implementierung dürfen keine Kreise enthalten. Das heißt, dass die transitiven Hüllen von Ext und $Impl$ irreflexiv sein müssen. Es gilt

$$\forall x, y : Ext^{+}_{x,y} \cup Impl^{+}_{x,y} \rightarrow x \neq y$$

5. Die Package-, die Verberbungs- und die Implementierungs-Beziehung dürfen kein Element mit sich selbst verbinden. Die zugehörigen Relationen müssen also irreflexiv sein.

$$\forall x, y : R_{x,y} \rightarrow x \neq y$$

für $R \in \{Pckg, Ext, Impl\}$.

6. Ein Paket darf höchstens ein übergeordnetes Paket besitzen. $Pckg$ muss demnach injektiv sein.

$$\forall x, y \exists z : Pckg_{x,z} \wedge Pckg_{y,z} \rightarrow x = y$$

7.4.3. Relationenalgebraische Form

Basierend auf diesen prädikatenlogischen Formeln werden ihre relationenalgebraischen Formen mit Hilfe von Äquivalenzumformungen gebildet. Tabelle 7.1 gibt eine Übersicht über spezielle Symbole und Operationen der relationalen Algebra.

Basierend auf der Aufzählung der prädikatenlogischen Formeln werden im Folgenden schrittweise die relationenalgebraischen Formen hergeleitet:

1. Überprüfung der syntaktischen Korrektheit der Beziehungen:

Tabelle 7.1. Symbole und Operationen der relationenalgebraischen Notation

Symbol / Operation	Bedeutung
$\mathbb{I}$	die Identitätsrelation
$\mathbb{O}$	die leere Relation
$\mathbb{L}$	die Allrelation
$R \cup S$	die Vereinigung von R und S
$R \cap S$	die Schnittmenge von R und S
$\overline{R}$	die Negation von R
R^T	die Transposition von R
$R;S$	die Komposition von R und S
R^+	die transitive Hülle von R

a) **Vererbung**:

$$\begin{aligned}
&\forall e,e' : Ext_{e,e'} \rightarrow (vB_e \wedge vB_{e'}) \vee (vS_e \wedge vS_{e'}) \\
\Leftrightarrow &\forall e,e' : Ext_{e,e'} \rightarrow \exists u : (vB_{e,u} \wedge vB_{e',u}) \vee (vS_{e,u} \wedge vS_{e',u}), \\
&\quad \text{da } R_x \Leftrightarrow \exists y : R_{x,y} \text{ mit } y = \diamond \text{ für } R \text{ ein Vektor} \\
\Leftrightarrow &\forall e,e' : Ext_{e,e'} \rightarrow \exists u : (vB_{e,u} \wedge vB^T_{u,e'}) \vee (vS_{e,u} \wedge vS^T_{u,e'}), \\
&\quad \text{da } R_{x,y} \Leftrightarrow R^T_{y,x} \\
\Leftrightarrow &\forall e,e' : Ext_{e,e'} \rightarrow (vB;vB^T)_{e,e'} \vee (vS;vS^T)_{e,e'}, \\
&\quad \text{da } \exists z : R_{x,z} \wedge S_{z,y} \Leftrightarrow (R;S)_{x,y}
\end{aligned}$$

$$\begin{aligned}
\Leftrightarrow &Ext \subseteq (vB;vB^T) \cup (vS;vS^T), \\
&\quad \text{da } \forall x,y : R_{x,y} \rightarrow S_{x,y} \vee T_{x,y} \Leftrightarrow R \subseteq S \cup T
\end{aligned}$$

b) **Implementierung**:

$$\begin{aligned}
&\forall e,e' : Impl_{e,e'} \rightarrow vB_e \wedge vS_{e'} \\
&\text{(Umformung analog zu 1a)}
\end{aligned}$$

$$\Leftrightarrow Impl \subseteq vB;vS^T$$

c) **Paketbeziehung**:

$$\begin{aligned}
&\forall e,e' : Pckg_{e,e'} \rightarrow vC_e \wedge vC_{e'} \\
&\text{(Umformung analog zu 1a)}
\end{aligned}$$

$$\Leftrightarrow Pckg \subseteq vC;vC^T$$

d) **Assoziation und Benutzt**:

$$
\begin{aligned}
&\forall e,e' : Assoc_{e,e'} \cup Uses_{e,e'} \rightarrow \{e,e'\} \subseteq dom(vB) \cup dom(vS) \\
\Leftrightarrow &\forall e,e' : Assoc_{e,e'} \cup Uses_{e,e'} \rightarrow \\
&\quad e \in (dom(vB) \cup dom(vS)) \wedge e' \in (dom(vB) \cup dom(vS)), \\
&\quad \text{da } \{x,y\} \subseteq X \Leftrightarrow x \in X \wedge y \in X \\
\Leftrightarrow &\forall e,e' : Assoc_{e,e'} \cup Uses_{e,e'} \rightarrow (vB_e \vee vS_e) \wedge (vB_{e'} \vee vS_{e'}), \\
&\quad \text{da } x \in (dom(R) \cup dom(S)) \Leftrightarrow R_x \vee S_x \\
\Leftrightarrow &\forall e,e' : Assoc_{e,e'} \cup Uses_{e,e'} \rightarrow (vB \cup vS)_e \wedge (vB \cup vS)_{e'}, \\
&\quad \text{da } R_x \vee S_x \Leftrightarrow (R \cup S)_x \\
\Leftrightarrow &\forall e,e' : Assoc_{e,e'} \cup Uses_{e,e'} \rightarrow \exists u (vB \cup vS)_{e,u} \wedge (vB \cup vS)_{e',u} \\
\Leftrightarrow &\forall e,e' : Assoc_{e,e'} \cup Uses_{e,e'} \rightarrow \exists u (vB \cup vS)_{e,u} \wedge (vB \cup vS)^T_{u,e'} \\
\Leftrightarrow &\forall e,e' : Assoc_{e,e'} \cup Uses_{e,e'} \rightarrow (vB \cup vS);(vB \cup vS)^T_{e,e'} \\
\\
\Leftrightarrow &Assoc \cup Uses \subseteq (vB \cup vS);(vB \cup vS)^T
\end{aligned}
$$

2. Definition der *in*-Relation:

$$
\begin{aligned}
&\forall e,c : in_{e,c} \leftrightarrow cM_{c,C(e)} \vee cM_{c,I(e)} \\
\Leftrightarrow &\forall e,c : in_{e,c} \leftrightarrow \exists u : (C_{e,u} \wedge cM_{c,u}) \vee (I_{e,u} \wedge cM_{c,u}), \\
&\quad \text{da } R_{x,F(y)} \Leftrightarrow \exists u : (F_{y,u} \wedge R_{x,u}), \text{ (falls F univalent)} \\
\Leftrightarrow &\forall e,c : in_{e,c} \leftrightarrow \exists u : (C_{e,u} \wedge cM^T_{u,c}) \vee (I_{e,u} \wedge cM^T_{u,c}) \\
\Leftrightarrow &\forall e,c : in_{e,c} \leftrightarrow (C;cM^T)_{e,c} \cup (I;cM^T)_{e,c} \\
\\
\Leftrightarrow &in = C;cM^T \cup I;cM^T, \\
&\quad \text{da } \forall x,y : R_{x,y} \leftrightarrow S_{x,y} \Leftrightarrow R = S
\end{aligned}
$$

3. Definition der *cM*-Relation:
   Für Würfel soll gelten:

$$
\begin{aligned}
&\forall b : |\{c : cM_{c,C(b)}\}| = 1 \\
\Leftrightarrow &\forall b : |\{c : (\exists u : C_{b,u} \wedge cM_{c,u})\}| = 1 \\
\Leftrightarrow &\forall b : |\{c : (\exists u : C_{b,u} \wedge cM^T_{u,c})\}| = 1 \\
\Leftrightarrow &\forall b : |\{c : (C;cM^T)_{b,c}\}| = 1
\end{aligned}
$$

   Das bedeutet, dass jedem b genau ein c in $R := C;cM^T$ zugewiesen wird. Also ist R eine totale Funktion bzw. eine Abbildung. In der Relationenalgebra ist R eine Abbildung genau dann, wenn $R;\overline{\mathbb{I}} = \overline{R}$ gilt. Für unseren Fall gilt dann:

$$C;cM^T;\overline{\mathbb{I}} = \overline{C;cM^T}$$

Mit Hilfe von Schröders Äquivalenz ($R;Q \subseteq S \Leftrightarrow R^T;\overline{S} \subseteq \overline{Q}$) und der Transposition der Konkatenation ($(R;S)^T \Leftrightarrow S^T;R^T$) erhält man durch Umformung

$$cM;C^T;C;cM^T = \mathbb{I}.$$

Analog gilt für Kugeln

$$\forall s : |\{c : cM_{c,I(s)}\}| = 1$$

$$\Leftrightarrow cM;I^T;I;cM^T = \mathbb{I}$$

4. Die transitive Hülle von Ext und $Impl$ muss irreflexiv sein:

$$\begin{aligned} &\forall x,y : Ext^+_{x,y} \cup Impl^+_{x,y} \rightarrow x \neq y \\ \Leftrightarrow &\forall x,y : Ext^+_{x,y} \cup Impl^+_{x,y} \rightarrow \overline{\mathbb{I}}_{x,y} \end{aligned}$$

$$\Leftrightarrow Ext^+ \cup Impl^+ \subseteq \overline{\mathbb{I}}$$

5. $Pckg, Ext$ und $Impl$ müssen irreflexiv sein:

   $\forall x,y : R_{x,y} \rightarrow x \neq y$
   (Umformung analog zu 4)

   $$\Leftrightarrow R \subseteq \overline{\mathbb{I}}$$

   für $R \in \{Pckg, Ext, Impl\}$

6. $Pckg$ muss injektiv sein:

$$\begin{aligned} &\forall x,y \exists z : Pckg_{x,z} \wedge Pckg_{y,z} \rightarrow x = y \\ \Leftrightarrow &\forall x,y \exists z : Pckg_{x,z} \wedge Pckg^T_{z,y} \rightarrow x = y \\ \Leftrightarrow &\forall x,y : (Pckg;Pckg^T)_{x,y} \rightarrow x = y \\ \Leftrightarrow &\forall x,y : (Pckg;Pckg^T)_{x,y} \rightarrow \mathbb{I}_{x,y} \end{aligned}$$

$$\Leftrightarrow (Pckg;Pckg^T) \subseteq \mathbb{I}$$

7.4.4. Übersetzung in KURE -Terme

Im letzten Transformationsschritt werden die relationenalgebraischen Terme in KURE -ausführbare Konstrukte übersetzt. Tabelle 7.2 zeigt gibt eine Übersicht über die Funktionen, die statt der Symbole und Operationen aus Tabelle 7.1 verwendet werden.

Tabelle 7.2. Ersetzung der Symbole/Operationen aus Tabelle 7.1

Symbol / Operation	RELVIEW-Funktion
$\mathbb{I}$	`I(R)`
$\mathbb{O}$	`O(R)`
$\mathbb{L}$	`L(R)`
$R \cup S$	`R|S`
$R \cap S$	`R&S`
$\overline{R}$	`-R`
R^T	`R^`
$R;S$	`R*S`
R^+	`trans(R)`

Die Funktionen `I(R)`,`O(R)` und `L(R)` liefern eine Identitätsrelation, eine leere Relation bzw. eine Allrelation mit den Dimensionen der Relation R.

Die Komposition $R;S$ ist zwar in ASCII-basiertem Text darstellbar, allerdings hat das Semikolon in KURE bereits eine andere Bedeutung. In relationalen Programmen (s. [Rel] „Examples for RelView programs") markiert es, wie für viele Programmiersprachen üblich, das Ende eines Befehls. Zur Eindeutigkeit wird deshalb die Komposition durch `R*S` ausgedrückt.

Neben den in der Tabelle dargestellten Ersetzungen müssen ebenfalls Mengenoperatoren durch Funktionen ersetzt werden. Die Teilmengenbeziehung $R \subseteq S$ wird im Folgenden durch `incl(R,S)`, die Gleichheit $R = S$ durch `eq(R,S)` und die Ungleichheit $R \neq S$ durch `-eq(R,S)` ausgedrückt.

Diesen Ersetzungen entsprechend werden die Gültigkeitsbedingungen diesmal in einer KURE -konformen Notation wiederholt:

1. Überprüfung der syntaktischen Korrektheit der Beziehungen:
   a) **Vererbung**:
      ```
      incl(Ext, vB*vB^|vS*vS^)
      ```
   b) **Implementierung**:
      ```
      incl(Impl, vB*vS^)
      ```
   c) **Paketbeziehung**:
      ```
      incl(Pckg, vC*vC^)
      ```
   d) **Assoziation und Benutzt**:
      ```
      incl(Assoc|Impl, (vB|vS)*(vB|vS)^)
      ```
2. Definition der *in*-Relation:
   ```
   eq(in, C*cM^|I*cM^)
   ```

3. Definition der *cM*-Relation:

   ```
   eq(cM*C^*C*cM^, I(cM*C))
   ```

   bzw.

   ```
   eq(cM*I^*I*cM^, I(cM*I))
   ```

4. Die transitive Hülle von *Ext* und *Impl* muss irreflexiv sein:

   ```
   incl(trans(Ext)|trans(Impl), -I(Ext|Impl))
   ```

5. *Pckg*, *Ext* und *Impl* müssen irreflexiv sein:

   ```
   incl(R,-I(R))
   ```

   für $R \in \{Pckg, Ext, Impl\}$

6. Pckg muss injektiv sein:

   ```
   incl(Pckg*Pckg^,I(Pckg))
   ```

7.5. Übertragung der Visualisierung in Relationen

Dieses Unterkapitel schildert die Konstruktion der benötigten Mengen und Relationen aus der Visualisierung und dem betrachtetem Quellcode. Auf Basis des Szenegraphs und des Java-Projekts des ECLIPSE-*Frameworks* werden methodisch die Grundlagen für die Überprüfung, wie sie im vorigen Unterkapitel 7.4 beschrieben worden sind, gebildet. Der so beschriebene Ablauf soll veranschaulichen, wie die Mengen und Relationen im J3Creator aufgebaut werden, um sie mit Hilfe von KURE überprüfen zu können.

Die im Folgenden beschriebenen Abläufe beginnen mit der Bildung der Entitätsmengen wie *CUBE*, *BOX* und *SPHERE*. Anschließend werden die Quellcodemengen *PACK*, *CLASS* und *INTFACE* behandelt. Auf diesen Mengen aufbauend werden die Relationen für die visualisierten Beziehungen und die Zuordnungen der Entitäten zum Quellcode konstruiert. Abschließend werden die Hilfsrelationen *cM* und *in* gebildet.

Entitätsmengen

CUBE, *BOX* und *SPHERE* beschreiben laut ihrer Definition die Menge aller möglichen *Information Cubes*, Würfel und Kugeln. Sie sind Hilfsmengen, durch die Entitäten erst in der relationalen Algebra existieren. Als solche müssen sie nicht gebildet werden, sondern werden als gegeben betrachtet. Für die Überprüfung einer aktuell sichtbaren Visualisierung wie es beim J3Creator der Fall

ist, reicht es aus, sich nur auf die sichtbaren Entitäten zu beschränken. Aus diesem Grund werden hier die Entitätsmengen $CUBE$, BOX, $SPHERE$ und ihre zusammenfassende Menge $ENTITY$ synonym verwendet mit vC, vB, vS, und $vENTITY$. Das bedeutet, dass fortan in diesem Unterkapitel das jeweilige Pendant der sichtbaren Elemente anstatt der formal definierten Entitätsmenge als Basis für die Relationen dient.

Um vC, vB, vS, und $vENTITY$ zu konstruieren, muss über den Szenegraph, wie er in Abbildung 4.6 dargestellt ist, iteriert werden. Dabei ist für die Entitätsmengen nur der Teilbaum für J3DShapes von Interesse. Jedes betrachtete J3DPackage wird in die Relation vC aufgenommen. J3DTypes werden anhand ihres `isInterface`-Kennzeichners unterschieden. Falls er auf `true` gesetzt ist, wird das betrachtete J3DType Teil von vS, ansonsten von vB. $vENTITY$ wird anschließend durch Vereinigung der drei Mengen vC, vB und vS gebildet.

Quellcodemengen

Die Quellcodemengen $PACK$, $CLASS$ und $INTFACE$ werden auf Basis des analyiserten Java-Projekts des ECLIPSE -*Framework s* aufgebaut. Dafür müssen diejenigen *IPackageFragments* und *ITypes* aus dem Projekt extrahiert werden, die bereits in Kapitel 3.2 beim Erfassen der für die Visualisierung nötigen Informationen betrachtet worden sind. Da eine nochmalige Analyse des Projekts unsinnig ist, werden die betrachteten Java-Elemente während der Erfassung in drei Listen gesammelt, die die Quellcodemengen verkörpern. Die *IPackage-Fragments* bilden die Menge $PACK$, *ITypes* werden ähnlich wie J3DTypes nach Rückgabewert der `isInterface()`-Methode unterschieden. Ist der Wert `true`, gehört der betrachtete Typ zur Menge $INTFACE$ ansonsten zur Menge $CLASS$.

Beziehungs-Relationen

Für die Beziehungs-Relationen wird der J3DConnection-Teilbaum des Szenegraphen betrachtet. Je nach Typ (s. Abbildung 4.5) der Verbindung wird das Paar $(source, destination)$ der verbundenen Entitäten wie in Tabelle 7.3 dargestellt der entsprechenden Relation zugeordnet. Falls der `isDirected`-Kennzeichner aussagt, dass die Verbindung ungerichtet ist, wird ebenfalls das Paar $(destination, source)$ in die Relation aufgenommen.

Zuordnungen

Nachdem Visualisierung und Quellcode durch Mengen und Relationen formalisiert sind, wird hier die Zuordnung der Entitäten zu ihren entsprechenden Strukturen im Quellcode aufgebaut. Da die Entitäts- und die Quellcodemengen schon konstruiert sein müssen, wird bei der Zusammenstellung der Relationen P, C und I auf sie zurückgegriffen.

Tabelle 7.3. Zuordnung der J3DConnection-Typen zu den Java-Relationen

J3DConnection-Typ	Java-Relation
`TYPE_ASSOCIATION`	*Assoc*
`TYPE_IMPLEMENTS`	*Impl*
`TYPE_INHERITS`	*Ext*
`TYPE_PKG`	*Pckg*
`TYPE_USES`	*Uses*

Exemplarisch wird hier die Konstruktion der Relation P gezeigt. C und I werden analog aufgebaut. Um P zu konstruieren, wird über den Domain von vC $dom(vC)$ iteriert. Mit Hilfe des Namen des J3DPackages kann eindeutig das zugehörige *IPackageFragment* aus $PACK$ identifiziert werden. Das so ermittelte Paar von J3DPackage und *IPackageFragment* wird als Tupel in die Relation P aufgenommen. Zur Konstruktion der Relationen C und I muss vC durch vB bzw. vS ersetzt werden und $PACK$ durch $CLASS$ bzw. $INTFACE$

Hilfsrelationen

Um cM zusammenzustellen, wird mit Hilfe von vC jedes sichtbare J3DPackage betrachtet. Von ihm werden alle enthaltenen J3DTypes benutzt, um mit Hilfe der Zuordnungen C und I die *ITypes* zu erhalten, die dem J3DPackage zugeordnet werden.

Hierbei muss mittels einer Positionskontrolle darauf geachtet werden, dass alle J3DTypes wirklich innerhalb des *Information Cubes* liegen. Dazu werden die Ausmaße des *Information Cubes* genommen und halbiert, um einen Radius für jede Dimension um den Mittelpunkt zu erhalten. Diese Radien müssen für jeden J3DType um seine Größe gekürzt werden, um kontrollieren, ob sie nur teilweise im Inneren liegen. Für eine erfolgreiche Positionskontrolle muss der Positionierungsvektor des J3DTypes innerhalb der gekürzten Radien liegen.

Diese Positionskontrolle wird ebenfalls zur Konstruktion der *in*-Relation benutzt. Ähnlich wie bei cM werden zu jedem J3DPackage die als Kinder zugeordneten J3DTypes betrachtet. Liegt das betrachtete J3DType innerhalb des *Information Cubes*, wird das Vater-Kind-Paar zur Relation hinzugefügt.

8. Realisierung semantikverändernder Modifikationsmöglichkeiten

Nachdem die Visualisierung vom Benutzer semantikerhaltend modifiziert und vom J3Creator auf Gültigkeit geprüft werden kann, wird in diesem Kapitel die Möglichkeit behandelt, die Visualisierung semantikverändernd zu modifizieren und damit ebenfalls den Quellcode zu ändern. Als Modifikationen werden hier folgende Operationen betrachtet:

- Hinzufügen eines Elements
- Verschieben eines Elements
- Umbenennen eines Elements
- Entfernen eines Elements

Ein Element kann dabei eine der Entitäten *Information Cube*, Würfel und Kugel oder eine Röhre, die zwei Entitäten miteinander verbindet, sein. Diese Verbindung kann eine Assoziations-, Implementierungs-, Vererbungs-, Benutzt- oder Subpaketbeziehung sein.

In Kapitel 8.1 wird zuerst eine allgemeine Vorgehensweise für semantikverändernde Modifikationen vorgestellt, die unabhängig von der Operation und dem ausgewählten Element. Bevor diese in Kapitel 8.3 in Bezug auf Operation und Element konkretisiert werden, zeigt Kapitel 8.2 die Grenzen, die dem J3Creator bei der semantikverändernden Modifikation gesetzt sind.

8.1. Allgemeine Vorgehensweise

Unabhängig von der konkreten Operation und des selektierten Elements ist für die Durchführung einer semantikverändernden Modifikation eine geordnete Vorgehensweise sinnvoll, um zu verhindern, dass die Visualisierung nicht mehr syntaktisch korrekt oder nicht mehr konsistent mit dem Quellcode ist. Diese Vorgehensweise stellt einen groben Ablauf dar, der für jede Operation durchlaufen wird, deren einzelne Schritte aber abhängig von Operation und selektiertem Element durchgeführt werden. Der Ablauf umfasst folgende Schritte:

1. Vorbedingungen prüfen

2. Veränderung des Szenegraphen durchführen
3. Gültigkeit der Veränderung prüfen
4. Veränderung des Quellcodes durchführen

Die einzelnen Schritte werden im Folgenden motiviert und erklärt.

8.1.1. Vorbedingungen prüfen

Da eine semantikverändernde Modifikation aufwendig sein kann, ist eine einfache Prüfung der Operation empfehlenswert, um bei einem absehbaren Misserfolg vorzeitig abzubrechen. Dadurch wird verhindert, dass in den folgenden Schritten der Szenegraph und die Relationen geändert werden, um anschließend festzustellen, dass die Operation nicht durchführbar ist, und die Änderungen wieder rückgängig machen zu müssen. Dies ist insbesondere bei Bedingungen sinnvoll, die durch die Gültigkeitsprüfung mittels KURE (s. Kapitel 7) nicht überprüft werden, aber im Java-Quellcode trotzdem gelten müssen.

Soll beispielsweise eine Klasse umbenannt werden, kann eine Namenskollision auftreten, falls bereits eine Klasse oder Schnittstelle mit dem gewünschten Namen existiert. Dieser Fehler würde erst bei der Veränderung des Quellcodes bemerkt, da die Gültigkeitsprüfung Namen nicht kontrollieren kann.

Um solche einfachen Fehler frühzeitig zu erkennen, kann der Szenegraph genutzt werden, der durch seine gewählte Struktur (s. Abbildung 4.6) in der Lage ist, einfache Aussagen über die Struktur des Quellcodes zu machen. Im Falle der Umbenennungs-Operation kann im Szenegraph im J3DPackage, dass das Paket der Klasse darstellt, nach einem J3DType gesucht werden, der bereits den gewählten neuen Namen besitzt. Trifft das zu, muss die Operation abgebrochen werden, da bereits eine Klasse mit dem neuen Namen existiert.

8.1.2. Veränderung des Szenegraphen

Ist die Prüfung der Vorbedingungen erfolgreich abgeschlossen, wird der Szenegraph entsprechend der gewählten Operation verändert. Dafür wird unter Beachtung der in Abbildung 4.6 gezeigten Struktur ein *J3DObject* hinzugefügt, verschoben, umbenannt oder entfernt, so dass der Szenegraph bereits die Visualisierung nach erfolgreichem Abschluss der Operation repräsentiert. Dies erscheint wenig sinnvoll, da dadurch auch Ergebnisse regelwidriger Operationen dargestellt werden. Da der Szenegraph aber Grundlage der Gültigkeitsprüfung ist, muss kurzzeitig zugelassen werden, dass er die an die Visualisierung geknüpften Regeln verletzt. Stellt sich im nächsten Schritt heraus, dass die Operation auf dem gewählten Element nicht möglich ist, muss die hier durchgeführte Veränderung rückgängig gemacht werden.

8.1.3. Gültigkeit der Veränderung prüfen

Bevor der Quellcode geändert werden kann, muss mit Hilfe von KURE und der relationenalgebraischen Formeln, die in Kapitel 7 formuliert worden sind, geprüft werden, ob die veränderte Visualisierung noch gültig ist.

Zur Überprüfung werden aus dem im vorigen Schritt veränderten Szenegraph die Relationen, wie in Kapitel 7.5 beschrieben, gebildet und anschließend überprüft. Existieren aus einer vorherigen Überprüfung der Visualisierung bereits Relationen, kann der Ablauf verkürzt werden, indem gezielt nur diejenigen Relationen angepasst werden, die durch die Modifikation der Visualisierung betroffen sind.

Ausgenommen von dieser Optimierung sind Modifikationen, durch die die Dimensionen von Relationen verändert werden müssen, da RELVIEW und damit KURE eine Relation als fest dimensionierte Matrix verwaltet. Dies ist dann der Fall, wenn sich eine der Entitätsmengen vC, vB und vS vergrößtert oder verkleinert. Dadurch verändert sich die Größe der Menge $ENTITY$, da die Bildung der konkreten Relationen für die Überprüfung auf den Mengen der sichtbaren Entitäten beruhen (vgl. Kapitel UebertragungRelationen). In Folge dessen ändern sich ebenfalls die Dimensionen der Beziehungs-Relationen. Soll beispielsweise ein Würfel hinzugefügt werden, vergrößert sich die Menge vB der sichtbaren Würfel. Dadurch ändert sich ebenfalls die Menge $ENTITY$, durch die dann auch die Dimensionen aller Beziehungs-Relationen betroffen sind. In diesem Fall müssen die Relationen komplett neu aus dem Szenegraph gebildet werden.

Bei der Prüfung der Gültigkeit müssen diejenigen Bedingungen ausgelassen werden, die die Konsistenz von Visualisierung und Quellcode überprüfen, da sie erst wieder im nächsten Schritt wiederhergestellt wird. Im Einzelnen sind das die Punkte 2 und 3 der Gültigkeitsbedingungen in Kapitel 7.1.

8.1.4. Veränderung des Quellcodes

Ist die Gültigkeit der Veränderung gewährleistet, wird in diesem Schritt der Quellcode mit Hilfe der *Java Development Tools* (s. Kapitel 3.3) entsprechend der auszuführenden Operation geändert.

Die Änderung umfasst zwei Bereiche: 1. den Teil des Quellcodes, der das Element darstellt, und 2. die Teile, deren grafischen Repräsentationen in Verbindung mit dem Element stehen. Die Modifikation im ersten Bereich führt lediglich die Operation an sich aus, also das Hinzufügen, Entfernen, Umbenennen oder Verschieben. Durch den zweiten Bereich wird versucht, die syntaktische Korrektheit zu sichern, indem der restliche Quellcode an die Operation angepasst wird. Dies kann beschränkt werden auf alle Klassen und Schnittstellen, deren grafische Repräsentationen in der Visualisierung in Beziehung zum modifiziertem Element stehen.

Wird beispielsweise eine Klasse umbenannt, müssen in allen Klassen und Schnittstellen, die von ihr erben oder eine Assoziations- oder Benutztbeziehung zu ihr besitzen, die `import`-Anweisungen, die Deklarationen von Attributen und Konstrukturaufrufe der umbenannten Klasse angepasst werden.

8.2. Grenzen der semantikverändernden Modifikation

Dieses Unterkapitel erläutert die Grenzen, die dem J3Creator bei der semantikverändernden Modifikation gesetzt sind. Insbesondere bei der Gewährleistung der syntaktischen Korrektheit des Quellcodes sind Abstriche zu machen.

Für den J3Creator ist die Überprüfung mit Hilfe von KURE (abgesehen von den Vorbedingungen) die einzige Stelle, die die Validität einer Modifikation entscheiden kann. Jede Modifikation, die von KURE bestätigt wird, gilt demnach als durchführbar. Die in Kapitel 7 aufgestellten Bedingungen sind allerdings keineswegs erschöpfend und können auch nur diejenigen Beziehungen im Quellcode untersuchen, die in der Quellcodeanalyse auch erfasst worden sind.

Dadurch ist es möglich, dass der Quellcode nach einer Modifikation nicht mehr kompilierfähig ist, obwohl diese aus Sicht des J3Creators fehlerfrei durchgeführt worden ist. Ein Beispiel dafür sind Java-Ausnahmen, sogenannte *Throwables*, die von Methoden als Reaktion auf einen Fehler ausgelöst werden können, und mittels einer `throws`-Deklaration angegeben werden. Eine Klasse `K` kann durch direktes oder indirektes Erben von *Throwable* dann ebenfalls in einer Methode ausgelöst werden. Wird `K` entfernt, muss ebenfalls die `throws`-Deklaration beseitigt werden. Da der J3Creator aber nur von den in der Analyse erfassten Beziehungen ausgeht, existiert keine Möglichkeit, diese Deklaration zu finden und zu entfernen. Um diese Beschränkung aufzulösen, müsste für den betrachteten Quellcode bezüglich seiner Klassen und Schnittstellen ein *dependency graph* [ASU86] konstruiert werden. Jede Klasse und Schnittstelle würde darin durch einen Knoten repräsentiert werden. Eine Kante von einem Knoten `K'` zu `K` würde bedeuten, dass `K'` von `K` abhängt. Zusätzlich müsste allerdings die Information zur Verfügung stehen, worin genau die Abhängigkeit besteht, um wissen zu können, wie sie beseitigt werden kann. Im Beispiel des *Throwable* müsste die Kante also aussagen, dass `K'` durch eine `throws`-Deklaration von `K` abhängt. Um den *dependency graph* zu konstruieren, müsste der gesamte Quellcode Anweisung für Anweisung nach Abhängigkeiten untersucht werden und entsprechend festgehalten werden.

Eine andere Beschränkung bezieht sich auf die in 1.1.2 erwähnten Folgeprobleme. Sie entstehen durch Abhängigkeiten von zu entfernenden Deklarationen wie die eines Attributs innerhalb der betrachteten Klasse. Der folgende Quellcodeausschnitt soll dies verdeutlichen:

```
public class Klasse{
   private AndereKlasse assoziation;
```

```
    public void tue(){
       int i = assoziation.berechne();
    }
}
```

In dem vorliegenden Quellcodeausschnitt wird eine Klasse namens `Klasse` deklariert, die ein Attribut `assoziation` besitzt und in der Methode `tue()` benutzt wird, um eine Methode aufzurufen, deren Rückgabewert, der Variable `i` zugewiesen wird. Soll das Attribut `assoziation` entfernt werden, müsste entschieden werden, wie die Zuweisung in `tue()` modifiziert werden soll. Zuerst müsste untersucht werden, ob die Zuweisung in nachfolgenden Anweisungen benutzt wird oder (wie in diesem Fall) nicht. Dies ist ein Problem, das beispielsweise mit *flow graphs* [ASU86] mit Hilfe der *next-use*-Information untersucht werden kann. Mit ihnen kann der Kontrollfluß analysiert werden, durch die Abhängigkeiten von Anweisungsblöcken erkennbar werden. Im vorliegenden Beispiel wird die Zuweisung nicht weiter genutzt, wodurch die Anweisung komplett gelöscht werden könnte. Wird `i` allerdings weiterverwendet, muss ein willkürlicher Ersatzwert für die Rückgabe der Methode `berechne()` in die Zuweisung eingesetzt werden. Dieser willkürliche Wert führt allerdings mit hoher Wahrscheinlichkeit dazu, dass die Klasse nicht mehr die ihr bestimmte Aufgabe erfüllt, also fehlerhaft arbeitet. Durch diesen Fehler ist der Benutzer gezwungen, die Zuweisung selbst nochmal durch einen sinnvolleren Wert zu ergänzen. Das heißt, dass auch die Analyse der Abhängigkeiten nicht den Benutzer von der Not befreit, selbst den Quellcode zu ändern.

Da der Aufwand, die hier gezeigten Grenzen zu beseitigen, sehr hoch und dabei nicht einmal erfolgversprechend ist, wird auf eine automatische Korrektur verzichtet. Abhängigkeiten, seien sie Klassenübergreifend oder -intern, die nicht im J3Creator durch Beziehungen repräsentiert werden, können bei der Modifikation des Quellcodes nicht berücksichtigt werden. Beim Entfernen einer Klasse oder Schnittstelle können daher nur die Teile des Quellcodes modifiziert werden, die laut Szenegraph in Verbindung zum entfernten Element stehen. Ebenso können beim Entfernen einer Assoziation oder Benutzt-Beziehung Anweisungen innerhalb der Klasse, die die entfernten Variablen referenzieren, nicht korrigiert werden.

8.3. Detaillierte Vorgehensweise

In diesem Kapitel werden die Schritte der in 8.1 vorgestellten allgemeinen Vorgehensweise gegliedert nach Operation und Element näher beschrieben. Entsprechend ihrer Aufzählung ist jede Beschreibung in verschiedene Absätze unterteilt. Der Absatz **Vorbedingung** erörtert konkret für die betrachtete Operation und das betrachtete Element die in 8.1.1 vorgeschlagene Prüfung der Vorbedingungen. Analog erörtern die Absätze **Szenegraph**, **Relationen** und **Quellcode** die Abschnitte 8.1.2, 8.1.3 und 8.1.4.

Bei der Beschreibung der detaillierten Vorgehensweise wird nicht jede mögliche Kombination von Operation und Element betrachtet, da sie in manchen Fällen unmöglich oder unsinnig ist. Beispielsweise kann keine Subpaketbeziehung explizit durch den Benutzer entfernt oder hinzugefügt werden, da diese sich aus den Namen der Pakete ergibt. Technisch wäre es zwar möglich, die Röhren für die Subpaketbeziehung durch den Benutzer modifizieren zu lassen, hätte aber aus genanntem Grund keinen Sinn.

Andere Einschränkungen ergeben sich auch aus der Art wie der J3Creator den Quellcode analysiert und darstellt. Dieser ignoriert beispielsweise die Kardinalität von Beziehungen (vgl. [Alb98] Kapitel 7). Dadurch werden mehrere Klassenattribute desselben Typs als eine einzige Assoziation erfasst und dargestellt. Wird diese Assoziation gelöscht, müssen alle Attribute entfernt werden, um die Beziehung wirklich aus dem Szenegraph entfernen zu können. Solche Einschränkungen werden jeweils am Anfang jedes der vier folgenden Abschnitte erklärt.

8.3.1. Hinzufügen eines Elements

Wie bereits geschildert, ist das Hinzufügen von Subpaketbeziehungen nicht möglich. Assoziationen und Benutzt-Beziehungen werden nur hinzugefügt, wenn noch keine solche Beziehung vom gewünschten Start- zum Zielelement existiert. Dies begründet sich in der Einschränkung des J3Creators, die Kardinalitäten von Beziehungen zu ignorieren. Da im Szenegraph und in den Relationen bereits eine Beziehung der gewünschten Art vom Start- zum Zielelement existiert, wäre das Hinzufügen einer der beiden Beziehungen eine reine Quellcodemodifikation und keine Modifikation der Visualisierung.

Hinzufügen eines *Information Cubes*

Vorbedingung Als Vorbedingung muss im Szenegraph geprüft werden, ob kein J3DPackage existiert, das bereits den gewünschten Namen trägt.

Szenegraph Ein neues J3DPackage $\texttt{P}_{neu}$ mit dem gewünschten Namen wird erzeugt und als Kind zum Teilbaum des Szenegraphen hinzugefügt, der die J3DShapes enthält (siehe Abbildung 4.6).

Anhand des neuen Namens muss geprüft werden, ob sich Subpaketbeziehungen ändern. $\texttt{P}_{neu}$ kann sowohl Subpaket als auch übergeordnetes Paket für jedes bereits im Szenegraph existierende J3DPackage werden. Um dies festzustellen wird der Name in seine durch Punkte getrennte Bezeichner aufgetrennt. Der Name `de.ls10.paket` beispielsweise würde dann in die drei Bezeichner `de`, `ls10` und `paket` aufgeteilt werden. Mit Hilfe dieser Bezeichner wird im Szenegraph nach einem übergeordneten Paket gesucht, indem sukzessive immer der letzte Bezeichner inklusive des Punkt-Trennzeichens aus

dem Namen entfernt und anhand des so gebildeten Teilnamens nach einem entsprechend benanntem J3DPackage gesucht wird. Ist solch ein J3DPackage P gefunden, wird eine neue Subpaketbeziehung von P_{neu} zu P erstellt und in den Teilbaum eingefügt, der die J3DConnections enthält. Für die aktuellen Subpakete von P muss einzeln geprüft werden, ob sie weiterhin P untergeordnet sind oder ob die bestehende Subpaketbeziehung entfernt und eine neue zu P_{neu} erstellt werden muss. Zu diesem Zweck wird für jeden Namen untersucht, ob er mit dem Namen von P_{neu} anfängt.

Wird kein übergeordnetes J3DPackage P gefunden, muss der Szenegraph in ähnlicher Weise durch den soeben beschriebenen Namensvergleich nach untergeordneten Exemplaren durchsucht werden. Für jedes gefundene neue Subpaket von P_{neu} muss die alte Subpaketbeziehung entfernt und eine neue zu P_{neu} erstellt werden.

Relationen Da sich hier die Menge vC und damit $ENTITY$ vergrößert, werden die Relationen komplett neu aus dem Szenegraph gebildet.

Quellcode Sind neue Subpaketbeziehungen entstanden, wird das neue *IPackageFragment* im *IPackageFragmentRoot*, das den verbundenen Paketen gemeinsam ist, erzeugt. Existiert dies nicht, kann aus den *IPackageFragmentRoots*, die bei der Analyse näher betrachtet worden sind (s. 3.2), eines frei gewählt werden.

Hinzufügen eines Würfel oder einer Kugel

Vorbedingung Für das Hinzufügen eines Würfels oder einer Kugel muss zunächst ein *Information Cube* als Stellvertreter für das Paket gewählt werden. Das zugehörige J3DPackage darf keinen J3DType mit dem Namen des neuen Elements besitzen.

Szenegraph Ein neuer J3DType wird erstellt, der abhängig von Würfel oder Kugel den `isInterface`-Kennzeichner erhält. Er wird zum J3DPackage hinzugefügt, das durch den gewählten *Information Cube* dargestellt wird.

Relationen Wie beim Hinzufügen eines *Information Cubes* vergrößert sich auch hier die Menge $ENTITY$ (allerdings auf Grund von vB bzw. vS), wodurch auch hier die Relationen neu aus dem Szenegraph gebildet werden müssen.

Quellcode Zum gewählten *Information Cube* wird mit Hilfe des vollen Namen das zugehörige *IPackageFragment* gesucht, in dem mittels der create-Methode eine neue *ICompilationUnit* erstellt, in der anschließend der neue *IType* produziert wird. Abhängig davon, ob ein Würfel oder eine Kugel hinzugefügt worden ist, wird der `isInterface`-Kennzeichner gesetzt.

Hinzufügen einer Assoziation

Vorbedingung Da Kardinalitäten von Beziehungen wie eingangs erwähnt nicht verwaltet werden, sollte noch keine Assoziationsbeziehung vom gewählten Quell-J3DObject zum Ziel führen.

Szenegraph Eine neue J3DConnection mit gewählter Quelle und Ziel mit dem Typ `TYPE_ASSOCIATION` (s. Abbildung 4.5) wird erstellt und zum J3D-Connections-Teilbaum hinzugefügt.

Relationen In der Relation *Assoc* wird das Tupel (Quelle, Ziel) hinzugefügt.

Quellcode Die Modifikation des Quellcodes beschränkt sich völlig auf die Klasse, die durch das Quell-J3DObject dargestellt wird. Zur Realisierung der Assoziation wird dort ein Attribut vom Typ der Zielklasse erstellt mit Hilfe der Methode `createField()`. Zusätzlich wird ihr `import`-Namensraum um den voll qualifizierten Namen der Zielklasse erweitert.

Hinzufügen einer Vererbungsbeziehung

Vorbedingung Existiert bereits eine Vererbungsbeziehung von der gewählten Quelle zum Ziel, kann die Operation abgebrochen werden.

Szenegraph Analog zur Assoziation wird eine neue J3DConnection vom erbenden J3DShape zum geerbten erstellt und dem J3DConnections-Teilbaum hinzugefügt. Der Typ wird hier allerdings auf `TYPE_INHERITS` gesetzt.

Relationen Die Relation *Ext* erhält eine neue Zuordnung von der Quelle zum Ziel.

Quellcode Wie beim Hinzufügen einer Assoziation beschränkt sich die Quellcodemodifikation auf die Quelle. Ihr Namensraum wird um den Namen des Typs, von dem geerbt wird, erweitert. Die Vererbungsbeziehung an sich wird mit Hilfe eines AST realisiert, in dem der *TypeDeclaration*-Knoten um die Erweiterung ergänzt wird. Im weiteren muss unterschieden werden, ob die Quelle eine Schnittstelle oder eine Klasse ist. Ist sie eine Schnittstelle wird der Typname des Ziels in die Liste der geerbten Schnittstellen hinzugefügt, die mit der Methode `superInterfaces()` erhalten wird. Falls sie eine Klasse ist, wird über die Methode `setSuperClass()` der Klassenname der gewählten Superklasse gesetzt. Darüber hinaus müssen die Konstruktoren den `super`-Konstruktor aufrufen, für den die richtigen Parameter gefunden werden müssen. Existiert der leere Konstruktor, wird er der Einfachheit halber gewählt. Ist

der allerdings nicht definiert, kann nur anhand der Konstruktorparameter, die die erbende Klasse selbst erhält, nach einem passenden `super`-Aufruf gesucht werden. Ist diese Suche erfolglos, muss der Benutzer den Instanziierungsaufruf an die Superklasse selbst eintragen.

Hinzufügen einer Implementierungsbeziehung

Vorbedingung Wie beim Hinzufügen einer Assoziations- und einer Vererbungsbeziehung sollte noch keine Implementierungsbeziehung von der Quelle zum Ziel führen.

Szenegraph Eine neue J3DConnection mit gewählter Quelle und Ziel mit dem Typ `TYPE_IMPLEMENTS` wird erstellt und zum J3DConnection-Teilbaum hinzugefügt.

Relationen Das Paar (Quelle,Ziel) wird zur Relation *Impl* hinzugefügt.

Quellcode Wie beim Hinzufügen einer Vererbungsbeziehung wird der AST der Quellklasse gebildet und die Liste der Schnittstellen mit der Methode `superInterfaces()` aus dem *TypeDeclaration*-Knoten angefordert. Diese Liste stellt diesmal nicht die geerbten, sondern die implementierten Schnittstellen dar, da der AST aus einer Klasse und keiner Schnittstelle gebildet worden ist. An diese Liste wird der Name der Zielschnittstelle angefügt.

Um die Implementierungsbeziehung vollständig zu realisieren, müssen noch die in der Schnittstelle deklarierten Methoden implementiert werden. Für jede der deklarierten Methoden wird eine *MethodDeclaration* mit der gleichen Signatur erzeugt und zum AST hinzugefügt.

Abschließend wird der `import`-Namensraum der Quellklasse erweitert um den vollqualifizierten Namen der implementierten Schnittstelle.

Hinzufügen einer Benutzt-Beziehung

Vorbedingung Wie bei den vorgehenden Beziehungen darf nicht bereits eine Benutzt-Beziehung von der Quelle zum Ziel führen.

Szenegraph Eine neue J3DConnection mit dem Typ `TYPE_USES` von der gewählten Quelle zum Ziel wird erzeugt und in den J3DConnection-Teilbaum eingefügt.

Relationen Das Tupel (Quelle, Ziel) wird zur Relation *Uses* hinzugefügt.

Quellcode Im *IType*, das durch die Quelle dargestellt wird, wird mit Hilfe der Methode `createMethod()` eine Methode deklariert, die einen Parameter erhält vom Typ der Zielklasse.

8.3.2. Verschieben eines Elements

Das Verschieben eines Elements macht im J3Creator nur für Würfel und Kugeln Sinn. Da *Information Cubes* hier nicht ineinander geschachtelt werden, können sie nicht durch Ziehen in ein anderes Exemplar zum Subpaket werden. Ebenso können die visualisierten Beziehungen nicht verschoben werden. Die Röhren, die die Beziehungen darstellen, sind fest an ihre Endpunkte gebunden, um zu verhindern, dass sie nach visuellen Modifikationen keine Quelle oder kein Ziel mehr besitzen. Dies stellt allerdings keine wesentliche Beschränkung dar, da semantikveränderndes Verschieben von Beziehungen durch Entfernen und anschließendes Hinzufügen einer Röhre nachgebildet werden kann.

Würfel oder Kugel

Vorbedingungen Das Verschieben eines Würfels oder einer Kugel ist erst dann semantikverändernd, wenn sich die verschobene Entität innerhalb eines *Information Cubes* befindet, der zudem nicht der ursprüngliche sein darf. Wie beim Hinzufügen muss mit Hilfe des Szenegraphs geprüft werden, ob bereits ein J3DType mit dem Namen der verschobenen Entität im Ziel-J3DPackage existiert.

Szenegraph Um die Inhaltsbeziehung von Paketen und Java-Typen auch im Szenegraph korrekt widerzuspiegeln, wird das betroffene J3DType aus dem alten J3DPackage entfernt und zu demjenigen wieder hinzugefügt, das den Ziel-*Information Cube* darstellt.

Relationen Da die verschobene Entität nach der Operation in einem anderen *Information Cube* liegt, muss die alte Zuordnung des J3DType zum J3DPackage aus der *in*-Relation gelöscht und durch eine Zuordnung zum neuen J3DPackage ersetzt werden. Dieses muss ebenso in der Relation *cM* mit dem *IType* gepaart werden, das durch den J3DType dargestellt wird, nachdem die Zuordnung mit dem alten J3DPackage gelöscht worden ist.

Quellcode Wird eine Klasse oder Schnittstelle verschoben, so muss nicht nur ihre Deklaration, sondern auch ihre Quellcodedatei verschoben werden. Aus diesem Grund wird mittels des `move`-Befehls (s. Abbildung 3.4) nicht der *IType*, sondern die *ICompilationUnit* verschoben.

Für die Klassen und Schnittstellen, die eine Beziehung zum verschobenen Java-Element besitzen, ändert sich diese zwar nicht, jedoch muss der mit den `import`-Anweisungen definierte Namensraum angepasst werden an den neuen vollqualifizierten Namen.

In den nicht betrachteten Klassen und Schnittstellen besteht die Möglichkeit, dass das verschobene Element in den Namensraum eingebunden ist, aber nicht weiter verwendet wird. Diese ungenutzten `import`-Anweisungen müssten eigentlich ebenfalls angepasst werden. Dafür muss allerdings jede Klasse und Schnittstelle im Quellcode betrachtet werden, da sie keine Beziehung zum verschobenen Element besitzen und aus diesem Grund keine Möglichkeit besteht, sie auf eine effiziente Art und Weise zu finden. Da das je nach Umfang des Quellcodes sehr aufwändig ist, werden Klassen und Schnittstellen mit ungenutzten `import`-Anweisungen nicht behandelt.

8.3.3. Umbenennen eines Elements

Analog zu Verschiebe-Operationen ist hier die Umbenennung von Beziehungen nicht möglich. Vererbungs- und Implementierungsbeziehungen werden im Quellcode nicht durch einen Namen repräsentiert, sondern durch die Schlüsselwörter `extends` bzw. `implements`, und können daher auch nicht umbenannt werden. Benutzt-Beziehungen charakterisieren die generelle Nutzung einer Klasse durch Methodenaufruf und sind als solche nicht zwangsläufig an einen Namen gebunden. Assoziationen hingegen können zwar durch den Namen des Klassenattributs umbenannt werden, aufgrund der Beschränkung des J3Creators, der diese nicht erfasst, ist das hier aber nicht möglich.

Umbenennen eines *Information Cube*

Vorbedingungen Der Szenegraph muss daraufhin überprüft werden, ob bereits ein J3DPackage existiert, das den gewünschten Namen besitzt.

Szenegraph Wie beim Hinzufügen eines *Information Cubes* wird hier anhand des neuen Namens geprüft, ob sich die Subpaketbeziehungen ändern. Da der Ablauf gleich ist, wird dieser hier erneut beschrieben.

Relationen Im Gegensatz zur Hinzufügungs-Operation müssen hier die Relationen nicht neu aufgebaut werden. In der Relation *Pckg* müssen lediglich die Änderungen der Subpaketbeziehungen, die im vorigen Schritt durchgeführt worden sind, entsprechend festgehalten werden.

Quellcode Das *IPackageFragment*, das das umzubennende Paket darstellt, erhält durch die `rename`-Operation (s. Abbildung 3.4) den neuen Namen. In

allen enthaltenen *ICompilationUnits* wird die Paketdeklaration auf den neuen Namen geändert. Dadurch müssen wie in 8.3.2 in allen Klassen und Schnittstellen, die zu den aktualisierten Java-Elementen eine Beziehung besitzen, die `import`-Namensräume entsprechend aktualisiert werden. Hier gilt ebenso die Einschränkung, dass Klassen und Schnittstellen mit ungenutzten `import`-Anweisungen nicht betrachtet werden.

Umbenennen eines Würfel oder einer Kugel

Vorbedingungen Das übergeordnete J3DPackage darf kein J3DType enthalten, das bereits den gewünschten Namen trägt.

Szenegraph Durch das Umbenennen eines J3DTypes ändert sich nichts in der Struktur des Szenegraphen. Lediglich das ausgewählte J3DType, das den Würfel bzw. die Kugel darstellt, muss umbenannt werden.

Relationen Da die Operation die Struktur des Szenegraphen nicht beeinflusst, bleiben alle Relationen erhalten.

Quellcode Das *IType*, das durch den J3DType dargestellt wird, wird mit Hilfe der `rename`-Methode umbenannt. Auf die gleiche Weise wird der Name der übergeordneten *ICompilationUnit* geändert. Mit Hilfe des Szenegraphen werden alle in Beziehung stehenden Klassen und Schnittstellen betrachtet, in denen die `import`-Anweisung für den umbenannten Typ auf den neuen Namen korrigiert wird.

8.3.4. Entfernen eines Elements

Beim Entfernen eines Elements existiert nur die Einschränkung, dass keine Subpaketbeziehungen entfernt werden können. Jedes Element, das in Abschnitt 8.3.1 hinzugefügt werden konnte, kann hier wieder entfernt werden.

Bei den Entfernungs-Operationen gibt es bei Beziehungen keine zu prüfenden Vorbedingungen, weswegen der entsprechende Absatz in den entsprechenden Beschreibungen fehlt.

Entfernen eines *Information Cubes*

Vorbedingungen Um die Modifikation im Quellcode möglichst minimal zu halten, wird verlangt, dass das J3DPackage, das den *Information Cube* darstellt, keine Kinder enthält. Würde diese Einschränkung nicht gelten, müssten für alle im zu entfernenden Paket enthaltenen Klassen und Schnittstellen alle Beziehungen entfernt werden, für die ihre Quelle außerhalb des Pakets liegen.

Szenegraph Das J3DPackage, das den ausgewählten *Information Cube* darstellt, wird aus dem Szenegraph entfernt.

Relationen Durch die Entfernung des *Information Cubes* wird die Menge *vC* verkleinert. Diese Modifikation wäre ohne neue Kontruktion der Relationen möglich, indem in *vC* die Zuordnung des J3DPackages zu $\diamond$ entfernt wird. Um allerdings sicherzustellen, dass die Relationen nur exakt die Elemente umfassen, die im Szenegraph existieren, werden hier trotzdem die Relationen aus dem Szenegraph neu gebildet.

Quellcode Das *IPackageFragment* des betrachteten Pakets wird mit Hilfe seiner `remove()`-Methode entfernt.

Entfernen eines Würfel oder einer Kugel

Vorbedingungen Der J3DType, der den Würfel oder die Kugel im Szenegraph repräsentiert, darf keine eingehenden Beziehungen mehr besitzen. Dies schränkt den Umfang der nötigen Quellcodemodifikationen ein.

Szenegraph Das ausgewählte J3DType wird mit seinen ausgehenden J3DConnections aus dem Szenegraph entfernt.

Relationen Wie beim Entfernen eines *Information Cubes* werden die Relationen aus dem modifiziertem Szenegraph neu erstellt.

Quellcode Die *ICompilationUnit*, das zur entfernten Klasse bzw. Schnittstelle gehört, wird mit der `remove()`-Methode entfernt.

Entfernen einer Assoziation

Szenegraph Die J3DConnection, die die Assoziation repräsentiert, wird aus dem Szenegraph entfernt.

Relationen Das Tupel (Quelle,Ziel) der Assoziation wird aus der Relation *Assoc* entfernt. Ist die Assoziation ungerichtet, wird ebenso das umgekehrte Tupel (Ziel, Quelle) entfernt.

Quellcode In der Klasse oder Schnittstelle, die die Quelle der Assoziation darstellt, werden mit Hilfe eines AST alle *FieldDeclaration*-Knoten untersucht. Diejenigen, die eine Variable vom Typ, die dem Ziel der Assoziation entsprechende, deklarieren, werden aus dem AST entfernt. Ist die Assoziation ungerichtet, wird der Vorgang analog für die Klasse oder Schnittstelle, die das Ziel darstellt, wiederholt.

Abschließend wird mit Hilfe des Szenegraph geprüft, ob die `import`-Anweisung ebenfalls entfernt werden kann. Zu diesem Zweck wird geprüft, ob das J3DType, das die Quelle repräsentiert, noch eine Beziehung zum Ziel der zu entfernenden Assoziation besitzt. Gilt das nicht, kann der vollqualifizierte Name des Ziels aus dem Namensraum entfernt werden.

Entfernen einer Vererbungsbeziehung

Szenegraph Wie beim Entfernen einer Assoziation wird die J3DConnection, die die ausgewählte Beziehung repräsentiert, aus dem Szenegraph entfernt.

Relationen Für das Paar (Quelle, Ziel) der Beziehung das Tupel aus der Relation *Ext* entfernt.

Quellcode Von der laut Beziehung erbenden Klasse oder Schnittstelle wird ein AST erzeugt, in dem der *TypeDeclaration*-Knoten modifiziert wird. Abhängig vom `isInterface()`-Bezeichner wird der Typ des Ziels im positiven Fall aus der Liste der ererbten Schnittstellen, die mit der Methode `superInterfaces()` angefordert werden kann, entfernt. Repräsentiert der AST hingegen eine Klasse, wird die Vererbungsbeziehung eliminiert, indem die Methode `setSuperClass()` mit Parameter `null` aufgerufen wird. Wie bei der Entfernung einer Assoziation wird abschließend geprüft, ob der Name des Ziels aus dem Namensraum der Quelle entfernt werden kann.

Entfernen einer Implementierungsbeziehung

Szenegraph Aus dem Szenegraph wird diejenige J3DConnection entfernt, die die ausgewählte Implementierungsbeziehung darstellt.

Relationen Das Tupel von Quelle und Ziel der zu entfernenden Beziehung wird aus der Relation *Impl* entfernt.

Quellcode Aus der implementierenden Klasse wird ein AST erzeugt. Im *TypeDeclaration*-Knoten wird der Name des Ziels aus der Liste der `superInterface` entfernt. Wie beim Entfernen einer Assoziation und einer Vererbungsbeziehung wird anhand des Szenegraphen geprüft, ob der Namensraums verkleinert werden kann.

Entfernen einer Benutzt-Beziehung

Szenegraph Aus dem Szenegraph wird diejenige J3DConnection entfernt, die die ausgewählte Benutzt-Beziehung darstellt.

Relationen Das Tupel, das Quelle und Ziel der Benutzt-Beziehung bilden, wird aus der Relation *Uses* entfernt. Ist die Benutzt-Beziehung ungerichtet, wird ebenfalls das umgekehrte Tupel entfernt.

Quellcode Aus der Klasse, die die Quelle der Beziehung bildet, wird ein AST erzeugt. In ihm wird jeder *MethodDeclaration*-Knoten dahingehend untersucht, ob die deklarierte Methode einen oder mehrere Parameter vom Typ des Ziels enthält. Ist dies der Fall, werden diese Parameter aus der Deklaration entfernt.

Auch hier wird abschließend untersucht, ob die `import`-Anweisung der Zielklasse aus dem Namensraum entfernt werden kann.

Teil IV.

Der J3Creator

9. Der Prototyp

Dieses Kapitel gibt einen Überblick über Aufbau und Implementierung des J3Creators. Kapitel 9.1 beschreibt den groben Aufbau des J3Creators und geht besonders auf die benutzten APIs ein. Anschließend werden in Kapitel 9.2 die einzelnen Komponenten des J3Creators näher betrachtet.

9.1. grober Aufbau des J3Creators

Der J3Creator baut auf vier APIs auf, um seine Aufgabe zu erfüllen. Mit Hilfe der API von ECLIPSE wird er als PlugIn in die Oberfläche eingegliedert, wodurch er erst in der Lage ist die *Java Development Tools* zu nutzen. Diese dienen dazu, den Quellcode zu analysieren, der anschließend mit Java3D visualisiert wird. Mit Hilfe von KURE können Modifikationen der Visualisierung überprüft werden, indem der Aufbau der grafischen Objekte kontrolliert wird.

ECLIPSE erlaubt Java-Programmen über eine API, sich als PlugIn einbinden zu lassen, und so auf Funktionalitäten anderer PlugIns, wie die JDT zurückzugreifen. Die JDT übernehmen das Einlesen der Dateien des Java-Projektes und stellen den Quellcode in Form von Datenstrukturen zur Verfügung. Dadurch ist der J3Creator in der Lage, Informationen über Struktur und Eigenschaften des Quellcodes einfach über Methodenaufrufe zu sammeln. Modifikationen des Quellcodes werden ebenfalls über Methodenaufrufe ausgeführt.

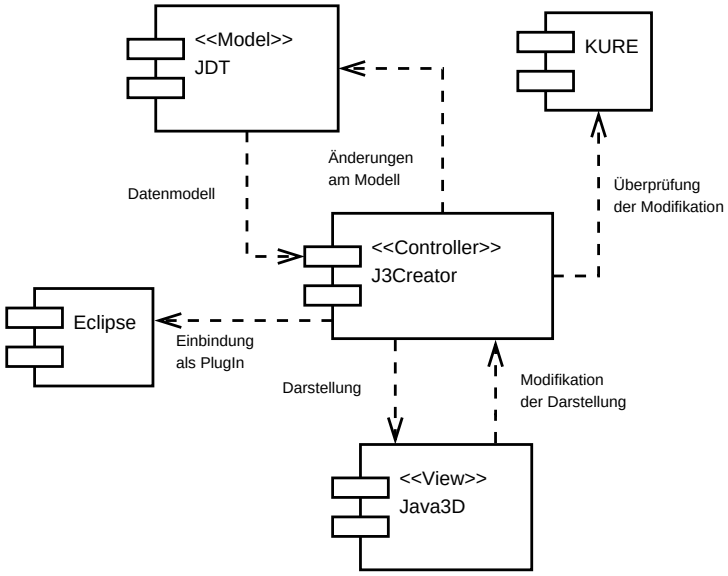
Zur Visualisierung wird die Java3D-API genutzt. Dazu baut der J3Creator einen Szenegraph auf, der die analysierten Strukturen des Quellcodes wiedergibt. Zusätzlich wird dieser so strukturiert, dass spätere Modifikationen nur die beabsichtigten Teile des Szenegraphen verändern.

Modifikationen der Visualisierung werden mit Hilfe von KURE auf Gültigkeit überprüft. Zu diesem Zweck werden zunächst relationenalgebraische Regeln formuliert, die erfüllt sein müssen, damit die Visualisierung als valide anerkannt wird. Der J3Creator konstruiert auf dieser Basis die Mengen und Relationen, die für die Überprüfung der Regeln benötigt werden.

Abbildung 9.1 stellt den groben Aufbau des J3Creators in einem Komponentendiagramm dar, in dem der J3Creator und die von ihm benutzten APIs durch Komponenten repräsentiert werden. Aufruf-Beziehungen symbolisieren die wichtigsten Interaktionen zwischen ihnen.

Der dargestellte Aufbau unterteilt die Komponenten in ein Modell des Quellcodes in Form der Datenstrukturen der JDT, einer Sicht auf diese unter Zuhilfenahme von Java3D und einer Verbindung zwischen beiden durch den

Abbildung 9.1. grober Aufbau des J3Creators



J3Creator. Dies entspricht dem Entwurfsmuster des *Model/View/Controller*-Schemas ([GHJV96]), dessen Elemente den Komponenten in Abbildung 9.1 als Stereotypen zugeordnet sind. Dem J3Creator steht dabei in der Rolle des *Controllers* die Aufgabe zu, die Elemente von *Model* und *View* eindeutig zueinander zuzuordnen und konsistent zu halten.

9.2. Die Komponenten des J3Creators

In diesem Unterkapitel wird der realisierte Prototyp - der J3Creator - beschrieben, der die in den vorigen Kapiteln gezeigten Schritte umsetzt. Anhand von Beschreibungen der Pakete und Klassen wird die Implementierung des J3Creators erläutert. Anschließend werden die wichtigsten Vorgänge im J3Creator mit Hilfe von Sequenzdiagrammen veranschaulicht.

Entsprechend den Komponenten in Abbildung 9.1 ist der J3Creator ebenfalls in vier Komponenten gegliedert, die je eine der vorgestellten APIs in den J3Creator einbinden. In den folgenden Abschnitten 9.2.1 bis 9.2.4 werden diese stückweise beschrieben.

9.2.1. Die ECLIPSE -Komponente

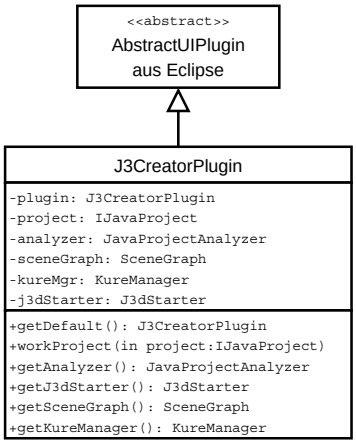
Die ECLIPSE -Komponente besteht aus vier Teilen: der PlugIn-Klasse `J3CreatorPlugin`, der Klasse `J3CreatorView`, die den J3Creator als ECLIPSE -*View* in die Benutzeroberfläche einbindet, einer `StartAction`, die das PlugIn startet, und dem Paket `vise3d.j3creator.plugin.viewActions`, das ECLIPSE -*ViewActions*

enthält, um die Verarbeitung der semantikverändernden Modifikationen auszulösen.

Die Klasse `J3CreatorPlugin`

Die Klasse `J3CreatorPlugin` stellt das eigentliche PlugIn dar und ist für ECLIPSE durch Erweiterung der Klasse `org.eclipse.ui.plugin.AbstractUIPlugin` als solche erkennbar. Da von ihr nur eine Instanz existieren darf, ist sie ein *Singleton*-Element [GHJV96], das mit Hilfe der Methode `getDefault()` angefordert werden kann. Die folgende Abbildung 9.2 stellt die Klasse in einem Klassendiagramm dar.

Abbildung 9.2. Klassendiagramm der Klasse `J3CreatorPlugin`



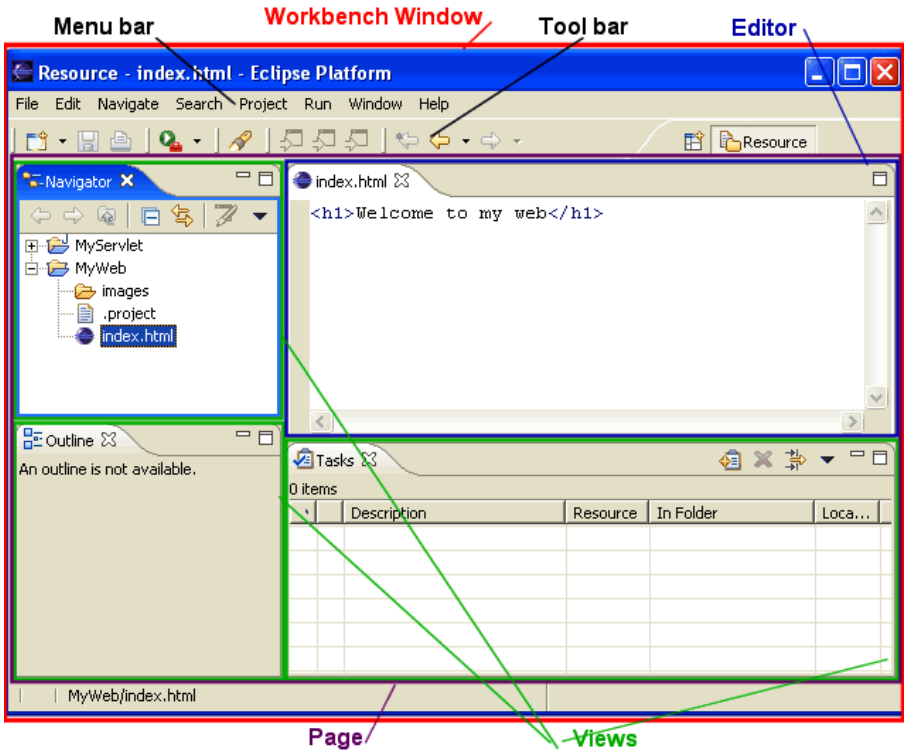
`J3CreatorPlugin` ist die zentrale Klasse des J3Creators. Sie koordiniert die anfängliche Arbeit in der Methode `workProject()`, in der die Analyse des Quellcodes des übergebenen *IJavaProjects* und die Anzeige mit Java3D angestoßen wird, und ist zentrale Anlaufstelle für Klassen, die auf andere Komponenten zugreifen müssen. Dazu besitzt sie Referenzen auf diejenige Klassen der Komponenten, die für einer übergreifende Zusammenarbeit nötig sind. Die Variable `analyzer` stellt die Verbindung in die JDT-Komponente dar, während `j3dStarter` und `sceneGraph` Referenzen zur Java3D-Komponente sind. Zur Überprüfung der Visualisierung mit Hilfe von KURE wird die Variable `kureMgr` genutzt. Das Attribut `project` ist eine Referenz auf das *IJavaProject*, das in `workProject()` übergeben worden ist.

Die Klasse `J3CreatorView`

Ein PlugIn besitzt standardmäßig keine grafische Repräsentation in der Benutzeroberfläche, weswegen die Klasse `J3CreatorView` ein ECLIPSE -*View* realisiert, in dem Java3D ihre Visualisierung darstellen kann. Ein *View* ist in ECLIPSE ein Teil der Benutzeroberfläche, das allgemein für die Anzeige von

Informationen genutzt wird. Abbildung 9.3 gibt einen Überblick über die grafischen Komponenten der Benutzeroberfläche.

Abbildung 9.3. Übersicht über die Benutzeroberfläche von ECLIPSE (Quelle: Hilfesystem von ECLIPSE)



Zur Realisierung erweitert `J3CreatorView`, wie in Abbildung 9.4 dargestellt, die abstrakte Klasse `org.eclipse.ui.part.ViewPart`, die die Basis für ein *View* darstellt. Dazu muss sie die Methode *createPartControl()* erfüllen, in der der Inhalt des *Views* aufgebaut wird. Im Falle des J3Creators wird hier das *Canvas3D* des *view branch graphs* (s. Kapitel 5) von Java3D eingesetzt. Um über Veränderungen des Fensters, das das *View* darstellt, benachrichtigt zu werden, implementiert `J3CreatorView` zusätzlich die Schnittstelle `org.eclipse.swt.events.ControlL` signalisiert in ihren zwei Methoden `controlMoved()` und `controlResized()`, dass das umgebende Fenster bewegt bzw. in der Größe verändert worden ist. Als Reaktion darauf wird das *Canvas3D* aufgefordert, seinen Inhalt zu aktualisieren.

Die Klasse `StartAction`

Die Klasse `StartAction` startet im `J3CreatorPlugin` die Analyse eines Java-Projekts. Zu diesem Zweck erweitert sie das Kontextmenü für Java-Projekte um eine Anweisung, im J3Creator seine Arbeit zu beginnen (s. Abbildung 9.5).

Abbildung 9.4. Klassendiagramm der Klasse `J3CreatorView`

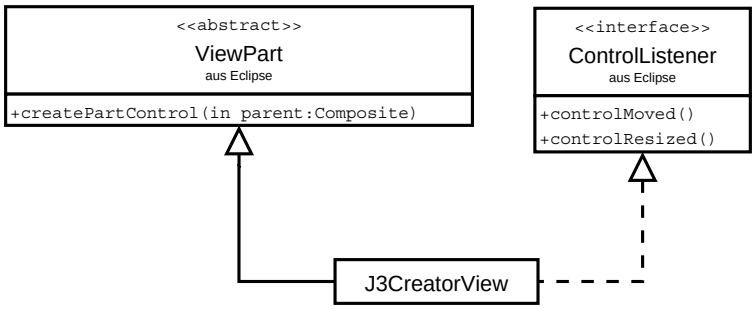
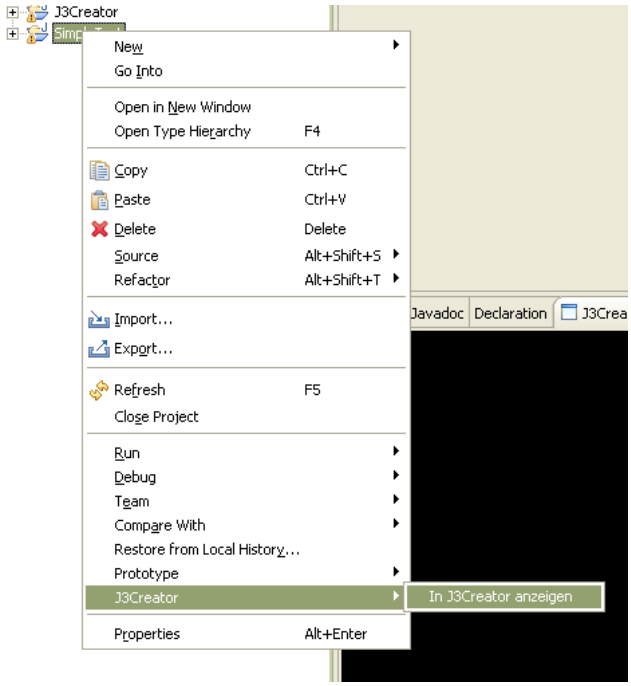
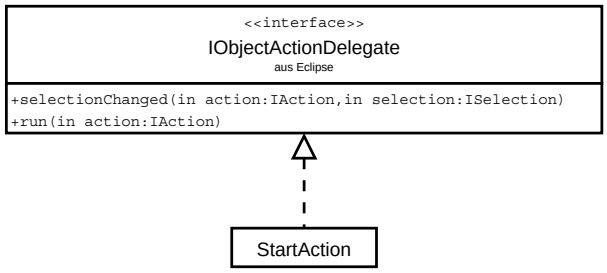


Abbildung 9.5. Kontextmenüerweiterung durch die Klasse `StartAction`



Um das Kontextmenü zu erweitern, implementiert `StartAction` die Schnittstelle `org.eclipse.ui.IObjectActionDelegate`, die die beiden Methoden `selectionChanged()` und `run()` definiert. Abbildung 9.6 zeigt in einem Klassendiagramm die Klasse `StartAction` und die Schnittstelle, die sie implementiert. Die Methode `selectionChanged()` teilt `StartAction` mit, wenn für ein Java-Projekt das Kontextmenü aufgerufen worden ist und übergibt dieses im Parameter `selection` gekapselt. Der Parameter `action` repräsentiert den Teil der Benutzeroberfläche, in dem die Selektion durchgeführt worden ist, was in diesem Fall das Kontextmenü ist. Die Methode `run()` wird aufgerufen, wenn der Benutzer den Punkt „In J3Creator anzeigen" auswählt. Als Reaktion darauf wird die Methode `workProject()` im `J3CreatorPlugin` aufgerufen, die als Parameter das Java-Projekt erhält, dass in `selectionChanged()` übergeben worden ist.

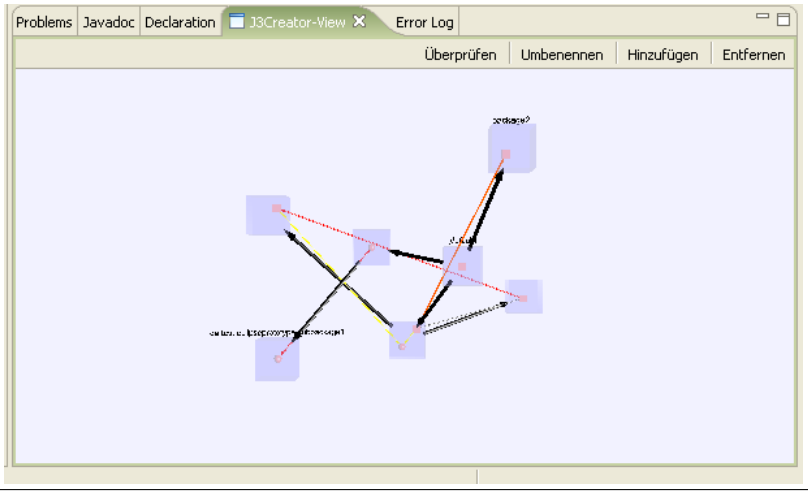
Abbildung 9.6. Klassendiagramm der Klasse `StartAction`



Das Paket `vise3d.j3creator.plugin.viewActions`

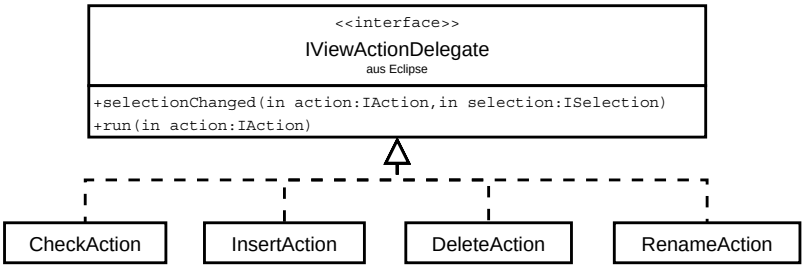
Das Paket `vise3d.j3creator.plugin.viewActions` enthält *ViewActions*, die ein ECLIPSE -*View* durch Knöpfe in der Titelleiste erweitern. Abbildung 9.7 stellt in einem Bildschirmfoto die *ViewActions* des J3Creators dar. Entsprechend den semantikverändernden Modifikationen sind dort Knöpfe zum Hinzufügen, Umbenennen und Entfernen zu sehen. Für Verschiebe-Operation existiert kein Knopf, da die durch grafisches Selektieren und Ziehen eines Elementes ausgeführt wird. Als zusätzliche *ViewAction* ist die Anweisung „Überprüfen" hinzugekommen, die die aktuelle Visualisierung mit Hilfe von KURE überprüft.

Abbildung 9.7. Darstellung der *ViewActions* im J3Creator



Damit ECLIPSE diese Knöpfe einbinden kann, implementieren alle Klassen in diesem Paket die Schnittstelle `org.eclipse.ui.IViewActionDelegate`. Für jeden sichtbaren Knopf in Abbildung 9.7 existiert in Abbildung 9.8 eine `Action`-Klasse, die als Präfix die englische Bezeichnung der zu realisierenden Operation besitzt. Ein `IViewActionDelegate` deklariert dieselben Methoden wie ein `IObjectActionDelegate`, die dieselbe Funktionalität definieren. Da sie bereits im vorigen Abschnitt erläutert worden sind, wird hier auf eine nochmalige Beschreibung verzichtet.

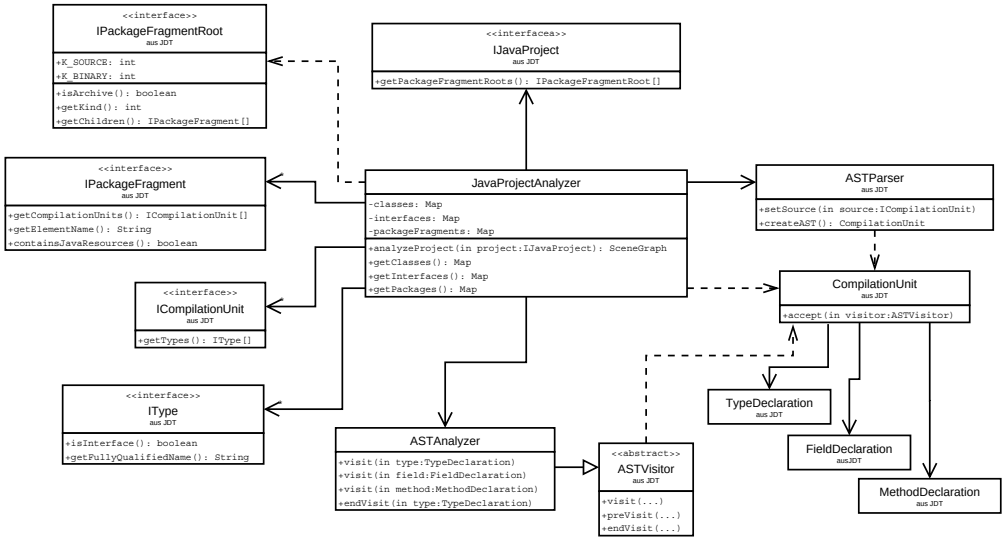
Abbildung 9.8. Klassendiagramm des Pakets `viewActions`



9.2.2. Die *Java Development Tools* -Komponente

Die JDT-Komponente besteht im Grunde nur aus den zwei Klassen `JavaProjectAnalyzer` und `ASTAnalyzer`. Da die JDT den Quellcode strukturiert in Form der Klassen des *Core*- und des *DOM*-Pakets zur Verfügung stellt, müssen die beiden Klassen dieser Komponente lediglich über sie navigieren, um die gewünschten Informationen über den Quellcode zu gewinnen. Abbildung 9.9 stellt alle bei der Analyse, die in Kapitel 3.2 beschrieben ist, benutzten Klassen in einem Klassendiagramm dar.

Abbildung 9.9. Klassendiagramm der JDT-Komponente



Der `JavaProjectAnalyzer` iteriert in der Methode `analyzeProject` über die Hierarchie des *Core*-Pakets, das in Kapitel 3.1 vorgestellt worden ist. Während der Analyse sammelt er in `classes`, `interfaces` und `packages` die betrachteten Klassen, Schnittstellen und Pakete, damit später bei Modifikationen schnell auf sie zugegriffen werden kann. Zusätzlich bilden diese drei Mengen die Grundlage für die Quellcodemengen bei der Überprüfung mit KURE
.

Aus den betrachteten `ICompilationUnits` werden mit Hilfe des `ASTParsers`

ihre ASTs gebildet. Dafür wird dem `ASTParser` mit der Methode `setSource()` die Übersetzungseinheit übergeben, zu der ein AST gewünscht wird. Dieser kann anschließend in Form der `CompilationUnit` mit der Methode `createAST()` angefordert werden. Der `ASTAnalyzer` kann diesen dann analysieren, indem er die Klasse `ASTVisitor` erweitert, der die Funktionalität des Besucher-Entwurfsmusters bietet. Für jeden Knotentyp besitzt der `ASTVisitor` eine `visit()`-Methode, von denen für die Analyse nur diejenigen benötigt werden, die `TypeDeclaration`-, `FieldDeclaration`- und `MethodDeclaration`-Knoten besuchen. Angestoßen wird die Analyse im `CompilationUnit`-Knoten mit der Methode `accept()`, der der `ASTAnalyzer` übergeben wird.

Als Ergebnis der Analyse gibt der `JavaProjectAnalyzer` ein Objekt vom Typ `SceneGraph` zurück, das seinem Namen entsprechend den Szenegraphen darstellt, der den Quellcode visualisiert.

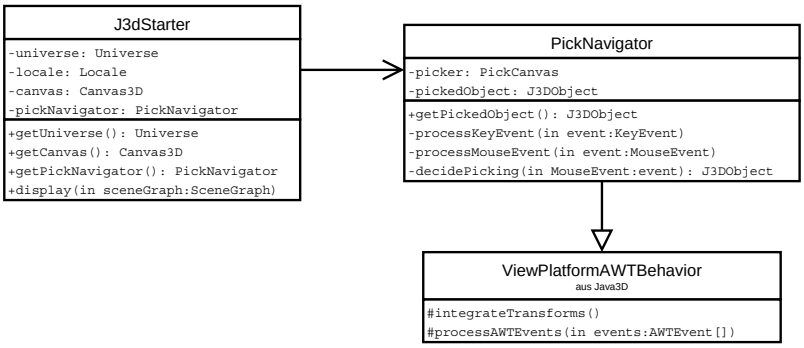
9.2.3. Die Java3D-Komponente

Die Java3D-Komponente umfasst drei Pakete: das Paket `vise3d.j3creator.j3d` und seine beiden untergeordneten Pakete `vise3d.j3creator.j3d.scenegraph` und `vise3d.j3creator.j3d.objects`. Das erste Paket stellt sicher, dass alle Klassen instanziiert sind, die bezüglich Java3D für den J3Creator nötig sind. Das `scenegraph`-Paket enthält die Basisklassen für den Szenegraph, während das `objects`-Paket die in Kapitel 4 vorgestellten *J3DObjects* enthält.

Das Paket `vise3d.j3creator.j3d`

Das Paket `vise3d.j3creator.j3d` stellt sicher, dass die Infrastruktur existiert, damit die Visualisierung mit Hilfe von Java3D angezeigt werden kann. Es besteht aus den zwei Klassen `J3dStarter` und `PickNavigator` (s. Abbildung 9.10). `J3dStarter` instanziiert mit Ausnahme des Szenegraphen die Klassen, die in den Abbildungen 4.1 und 5.1 dargestellt sind, also das Universum `Universe`, das `Locale` und den gesamten *view branch graph*. In der Methode `display()` wird der übergebene Szenegraph dem `Locale` als Kind hinzugefügt, wodurch die beschriebene Szene schließlich angezeigt wird. Weiterhin wird der Szenegraph um den `PickNavigator` erweitert, der für die Navigation und das *Picking* zuständig ist. Die Klasse `PickNavigator` erweitert die Java3D-Klasse `ViewPlatformAWTBehavior`, wodurch er über Aktionen des Benutzers mit Tastatur und Maus notifiziert wird. Die Ereignisse in Form von `AWTEvents`, die der geerbten Methode `processAWTEvents()` übergeben werden, werden abhängig von der Quelle an die Methoden `processKeyEvent()` und `processMouseEvent` weiter delegiert. Die Methode `processKeyEvent()` bewältigt die Umrechnungen für die Navigation, während `processMouseEvent` das *Picking* mit Hilfe des `PickCanvas` und der Methode `decidePicking` erledigt. Der Vorgang des *Pickings* ist in Kapitel 6 beschrieben.

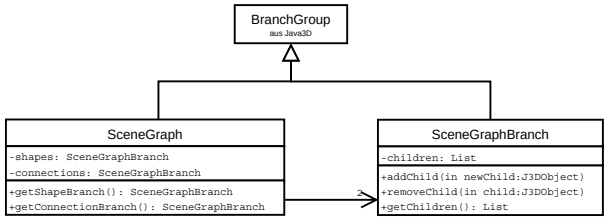
Abbildung 9.10. Klassendiagramm des Pakets `vise3d.j3creator.j3d`



Das Paket `vise3d.j3creator.j3d.scenegraph`

Das Paket `vise3d.j3creator.j3d.scenegraph` enthält die Basisklassen `SceneGraph` und `SceneGraphBranch` für den Szenegraph, der in Abbildung 4.6 dargestellt ist. `SceneGraph` stellt die Wurzel des Szenegraphen dar. Er erweitert die Java3D-Klasse `BranchGroup` und besitzt für den einfacheren Zugriff zwei Kinder vom Typ `SceneGraphBranch`, die die Wurzeln für die Teilbäume der *J3DShapes* und *J3DConnections* darstellen. `SceneGraphBranch` erbt ebenfalls von der Klasse `BranchGroup` und erweitert sie um eine Liste, in der alle direkten `J3DObject`-Kinder verwaltet werden. Abbildung 9.11 stellt die beiden Klassen in einem Klassendiagramm dar.

Abbildung 9.11. Klassendiagramm des Pakets `vise3d.j3creator.j3d.scenegraph`



Das Paket `vise3d.j3creator.j3d.objects`

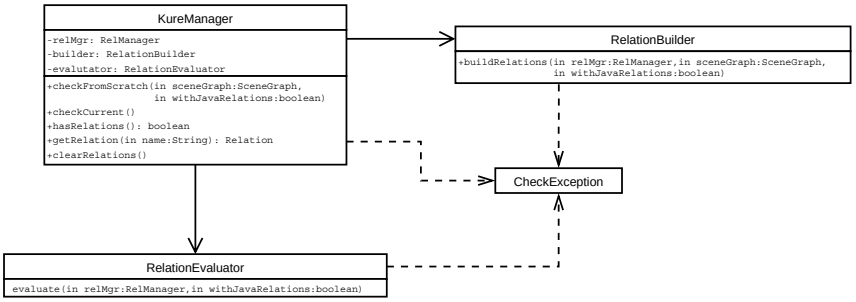
Das Paket `vise3d.j3creator.j3d.objects` enthält die in Kapitel 4.2 eingeführten *J3DObjects*. Da sie dort schon beschrieben worden sind, wird hier auf eine Beschreibung verzichtet.

9.2.4. Die KURE -Komponente

Die KURE -Komponente behandelt die in Kapitel 7 beschriebene Überprüfung der Visualisierung mit Hilfe von relationenalgebraischen Formeln. Sie besteht

aus den drei Klassen `KureManager`, `RelationBuilder` und `RelationEvaluator`. Die Klasse `KureManager` stellt sicher, dass KURE richtig initialisiert wird. Sie lädt die C-Bibliothek, die den funktionalen Kern von RELVIEW darstellt, und erstellt den *RelManager*. Sie stellt ebenso die Schnittstelle für alle Klassen dar, die außerhalb dieser Komponente liegen. Abbildung 9.12 gibt eine Übersicht über die Klassen.

Abbildung 9.12. Klassendiagram der KURE -Komponente



Zum Überprüfen werden die Methoden `checkFromScratch()` und `checkCurrent()` zur Verfügung gestellt, die im Fehlerfall eine `CheckException` schmeißen. Die Methode `checkFromScratch()` konstruiert die Relationen auf Basis des übergebenen Szenegraphen von Grund auf neu. Besitzt der Parameter `withJavaRelations` den Wert `true`, werden ebenfalls die Relationen erstellt, die die Konsistenz von Visualisierung und Quellcode prüfen. Die Methode `checkCurrent()` überprüft die zum Zeitpunkt des Aufrufs vorhandenen Relationen auf Gültigkeit. Zusammen mit der Methode `getRelation()`, die die Relation mit dem angegebenen Namen zurück gibt, können gezielt einzelne Relationen verändert werden, und so die Gültigkeit einzelner Modifikations-Operationen überprüft werden. Die Methode `hasRelations()` zeigt an, ob der *RelManager* zur Zeit Relationen besitzt. Ist die Überprüfung fehlgeschlagen, können zur Sicherheit mit der Methode `clearRelations()` alle Relationen gelöscht werden.

Aufgebaut werden die Relationen in der Klasse `RelationBuilder`. Wie in Kapitel 7.5 beschrieben iteriert er zu diesem Zweck über den übergebenen Szenegraph und gegebenfalls auch über die im `JavaProjectAnalyzer` gesammelten Pakete, Klassen und Schnittstellen, die er über das `J3CreatorPlugin` anfordern kann. Der `RelationEvaluator` übernimmt die eigentliche Überprüfung, indem er mit Hilfe des *RelManagers* die Gültigkeit der relationenalgebraisch formulierten Bedingungen kontrolliert.

Teil V.

Abschluss

10. Fazit

Ziel der vorliegenden Diplomarbeit war, aufbauend auf dem Konzept des J3Browsers, Modifikationen einer dreidimensionalen Visualisierung zu ermöglichen und in einem Prototyp - dem J3Creator - zu realisieren.

Aufgrund der Beschränkungen der verwendeten Komponenten des J3Browsers konnte dieser nicht wiederverwendet werden. Stattdessen wurde die Quellcodeanalyse und die Visualisierung mit Hilfe der *Java Development Tools* und von Java3D neu entwickelt, um im Anschluss daran Modifikationen ermöglichen zu können. Da der Schwerpunkt auf den Modifikationen lag, wurde die Visualisierung im Vergleich zum J3Browser in einer stark vereinfachten Variante realisiert.

Modifikationsmöglichkeiten können bewirken, dass die Visualisierung nicht mehr gültigen Java-Strukturen entspricht. Aus diesem Grund wurden die Regeln, die für gültigen Java-Quellcode gelten müssen, auf die grafischen Objekte der Visualisierung übertragen, so dass ihre Syntax so definiert ist, dass ihre Semantik Java-Strukturen entspricht. Diese Regeln wurden in relationenalgebraische Formeln übertragen, die mit KURE überprüft werden können. Auf diese Weise kann mit einem festen Regelsatz unabhängig von der konkreten Modifikation die Gültigkeit der Visualisierung entschieden werden.

Die aus der Modifikation der Visualisierung resultierende Modifikation des Quellcodes sollte die syntaktische Korrektheit gewährleisten. Dabei ist der J3Creator aufgrund von schwer erkennbaren und lösbaren Abhängigkeiten an Grenzen gestoßen.

Der J3Creator baut zur Erfüllung seiner Aufgabe auf vier APIs auf: ECLIPSE , *Java Development Tools* , Java3D und KURE . Eine der Hauptaufgaben war daher, diese vier APIs zu einem Ganzen zusammenzufügen. Die Benutzbarkeit war dabei in hohem Maße von der Qualität der Dokumentation abhängig. Insbesondere bei den JDT und bei Java3D existierten Lücken, die den Zusammenhang der Klassen schwer erkennbar machten. Den JDT mangelte es anfangs an einer Möglichkeit Modifikationen des ASTs in den Quellcode zu überführen. Als Notlösung existierte die Möglichkeit eine interne Klasse der JDT zu benutzen, um den AST in eine *String*-Zeichenkette umzuwandeln. Diese musste anschließend manuell durch Textersetzungen in dem Quellcode überführt werden. Mit der aktuellen Version 3 von ECLIPSE ist allerdings auch eine offizielle Möglichkeit realisiert worden. Java3D zeichnete sich durch Fehler in der Implementierung aus, die die Verwendung gewisser Klassen verbot, was sich meistens erst nach einer langen Suche nach einem Fehler in der eigenen Implementierung herausstellte. Hinderlich war bei der Fehlersuche auch der

Umstand, dass Java3D oft auf Grund falscher oder fehlender Aufrufe den Szenegraph nur teilweise angezeigt und den Fehler nicht mitgeteilt hat.

Mit der Realisierung der Modifikationsmöglichkeiten ist es nun möglich, dreidimensionale Visualisierungen von Software-Strukturen nicht nur als nettes Beiwerk zu betrachten, sondern so wie bei zweidimensionale *Computer Aided Software Engineering*-Werkzeugen (*CASE-Tools*) auch im Entwicklungsprozess einzusetzen.

11. Ausblick

Da der J3Creator ein Prototyp ist, bietet er zahlreiche Möglichkeiten zum Ausbau. Zu aller erst ist da die Realisierung der Visualisierung des J3Browsers zu nennen. Dieser gibt zusätzliche Informationen über Entitäten und Beziehungen durch eine besondere Markierung. So wird durch einen grünen Kreis auf einem Würfel symbolisiert, dass es sich um eine abstrakte Klasse handelt. Zusätzlich verfügt der J3Browser über Mechanismen zur Reduktion der Darstellungskomplexität, indem Entitäten und Beziehungen zusammengefasst werden. In Bezug auf die Modifikation setzt dies aber die Möglichkeit voraus, die Zusammenfassung manuell wieder aufzulösen, um gezielt ein zu modifizierendes Element selektieren zu können.

Zudem kann die Quellcodeanalyse erweitert werden, so dass beispielsweise auch erkennbar ist, welche Methoden in einer Klasse existieren und welche *Exceptions* sie auslösen. Dies hilft auch zum Teil dabei, die Grenzen der Quellcodemodifikation ein bisschen aufzuweichen, da mit mehr analysierten Strukturen und Beziehungen Abhängigkeiten besser erkennbar sein können. Da eine Visualisierung von Quellcode abstrahiert, indem es diesen in Form von Entitäten und Beziehungen darstellt, hat die Erweiterung der Analyse auch ihre Grenzen. Ab einem gewissen Maß ist die Menge an Informationen so groß, dass die Abstraktion nicht mehr gegeben ist, sondern der gesamte Quellcode in anderer Form dargestellt wird.

Die Überprüfung der Visualisierung mit KURE hat hier nur auf einer Basis von relationenalgebraischen Formeln aufgebaut. Es ist zu prüfen, ob alle Regeln, die für Java-Quellcode gelten müssen, mit Hilfe dieser Formeln erfasst werden können. Dies wird momentan im Rahmen einer Diplomarbeit untersucht.

Durch die Einbindung als ECLIPSE -PlugIn ergeben sich schließlich weitere Ausbaumöglichkeiten. So wäre denkbar, die Warnungen und Fehler, die der Compiler meldet, in der dreidimensionalen Visualisierung erkennbar zu machen. Diese werden bereits in der Benutzeroberfläche in Form von kleinen Warndreiecken und rot hinterlegten X-Symbolen im *Package-Explorer* deutlich gemacht. Möglich wäre auch eine Kopplung der Visualisierung an Editor-Fenster, so dass ein Doppelklick mit der Maus beispielsweise auf einem Würfel, den Editor für die visualisierte Klasse öffnet.

Literaturverzeichnis

[AF00] ALFERT, KLAUS und ALEXANDER FRONK: *3-Dimensional Visualization Of Java Class Relations*. Proceedings of the 5th World Conference on Integrated Design & Process Technology, 2000.

[AF02] ALFERT, KLAUS und ALEXANDER FRONK: *Manipulation of 3-dimensional Visualizations of Java Class Relations*. Integrated Design and Process Technology, 2002.

[AFE01] ALFERT, K., A. FRONK und 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.

[Alb98] ALBIR, SINAN SI: *UML in a Nutshell*. O'Reilly & Associates, 1998.

[Api] WIKIPEDIA: *Application Programming Interface*. `http://de.wikipedia.org/wiki/Application_Programming_Interface` (Zuletzt gesichtet am 01.09.2004).

[ASU86] AHO, ALFRED V., RAVI SETHI und JEFFREY D. ULLMAN: *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BF03] BERGHAMMER, RUDOLF und ALEXANDER FRONK: *Applying Relational Algebra in 3D Graphical Software Design*. Proceedings 7th International Seminar on Relational Methods in Computer Science, 2003.

[Cou99] COUCH, JUSTIN: *Raw J3D Tutorial*. The Java3D Community, 1999. `http://www.j3d.org/tutorials/raw_j3d/` (Zuletzt gesichtet am 25.09.2003).

[Ecl] ECLIPSE.ORG: *The Eclipse Project*. `http://www.eclipse.org` (Zuletzt gesichtet am 25.09.2003).

[Eng00] ENGELEN, FRANK: *Konzeption und Implementierung eines dreidimensionalen Klassenbrowsers für Java*. Diplomarbeit, Lehrstuhl für Software-Technologie, Universität Dortmund, 2000.

[GHJV96] GAMMA, E., R. HELM, R. JOHNSON und J. VLISSIDES: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 1996.

[Hei] HEISSE, JANICE J.: *New Language Features for Ease of Development in the Java 2 Platform, Standard Edition 1.5*. Sun Microsystems. `http://java.sun.com/features/2003/05/bloch_qa.html` (Zuletzt gesichtet am 19.08.2004).

[J3D] SUN MICROSYSTEMS: *Java3D API*. `http://java.sun.com/products/java-media/3D/` (Zuletzt gesichtet am 18.08.2004).

[Jav] SUN MICROSYSTEMS: *Javadoc Tool*. `http://java.sun.com/j2se/javadoc/index.jsp` (Zuletzt gesichtet am 17.08.2004).

[Jdt] THE ECLIPSE PROJECT: *Java Development Tools Subproject*. `http://www.eclipse.org/jdt/` (Zuletzt gesichtet am 18.08.2004).

[KUR] INSTITUT FÜR INFORMATIK UND PRAKTISCHE MATHEMATIK, CHRISTIAN-ALBRECHTS-UNIVERSITY OF KIEL: *Kiel University Relation Package*. `http://www.informatik.uni-kiel.de/~progsys/relview/kure` (Zuletzt gesichtet am 15.06.2004).

[Rel] INSTITUT FÜR INFORMATIK UND PRAKTISCHE MATHEMATIK, CHRISTIAN-ALBRECHTS-UNIVERSITY OF KIEL: *RelView System*. `http://www.informatik.uni-kiel.de/~progsys/relview/` (Zuletzt gesichtet am 15.06.2004).

[Szy03] SZYMANSKI, OLIVER: *Relationale Algebra im dreidimensionalen Software-Entwurf - ein werkzeugbasierter Ansatz*. Diplomarbeit, Lehrstuhl für Software-Technologie, Universität Dortmund, 2003.

[Uml] OBJECT MANAGEMENT GROUP: *Unified Modeling Language*. `http://www.uml.org` (Zuletzt gesichtet am 17.08.2004).

[Vrm] WEB3D CONSORTIUM: *The Virtual Reality Modeling Language*. `http://www.web3d.org/x3d/specifications/vrml/ISO_IEC_14772-All/index.html` (Zuletzt gesichtet am 17.08.2004).