A Hierarchical Eclipse-based Editor for System Dependency Graphs

Masterarbeit

Dean J. Finkes

29. Oktober 2016

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe Software Engineering

Betreut durch: Prof. Dr. Wilhelm Hasselbring
M. Sc. Christian Wulf

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Zusammenfassung

In der heutigen Zeit basiert ein Großteil der Computersysteme auf Mehrkernprozessoren. Obwohl dies die Ausführung von sequentiellen Programmen zwar nicht beeinträchtigt, lässt sich das Potential der parallelen Berechnung nur in Verbindung mit einem parallelisierten Quellcode ausnutzen. Durch die Generierung des Systemabhängigkeitsgraphen eines beliebigen Programms ist es möglich parallelisierbare Programmabschnitte zu erkennen und bei Bedarf zu bearbeiten. Der modifizierte Graph stellt anschließend die Grundlage der, mittels eines semi-automatischen Ansatzes generierten, parallelen Version des ursprünglichen Programms dar.

Der im Rahmen dieser Arbeit entwickelte Grapheditor ermöglicht die Visualisierung und Bearbeitung beliebiger Graphen, wie etwa den erwähnten Systemabhängigkeitsgraphen. Der Im- und Export erfolgt über die Anbindung einer Neo4j-Graphdatenbank, welche auf dem lokalen System vorliegt oder mittels einer URL erreichbar ist. Neben der flachen Ansicht eines Graphen ist der Nutzer in der Lage Kanten zu typisieren und eine hierarchische Repräsentation zu generieren. Das Ausdrücken von Kanten durch Eltern- bzw. Kindknoten entschlackt den visualisierten Graphen und gewährt eine verbesserte Übersicht. Für das Bearbeiten eines neu erstellten oder importierten Graphen steht ein Kontextmenü zur Verfügung, welches sämtliche zulässigen Aktionen beinhaltet. Ein möglichst universeller Einsatz des Grapheditors wird über die Implementierung als erweiterndes Eclipse-Plugin realisiert.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1

1.2 Kontext 1

1.3 Ziele 3

1.3.1 Z1: Abbildung von hierarchischen Strukturen eines SDG 4

1.3.2 Z2: Import- und Exportfunktionalität von SDG für eine Neo4j-Datenbank 4

1.3.3 Z3: Anpassungen an das Kontextmenü 5

1.3.4 Z4: Erweiterung der bisherigen Bedienbarkeit des Editors 5

1.3.5 Z5: Evaluation des Grapheditors 6

1.4 Aufbau 7

2 Grundlagen und Technologien **9**

2.1 Graphentheorie 9

2.1.1 Der Eigenschaftsgraph 9

2.1.2 Der Systemabhängigkeitsgraph 10

2.1.3 Der Java-spezifische Systemabhängigkeitsgraph 11

2.2 Die Entwicklungsumgebung Eclipse 14

2.2.1 Die Rich Client Platform 15

2.2.2 Der Lebenszyklus eines Plugins 16

2.3 Das Eclipse Modeling Framework 17

2.3.1 Das Ecore (Meta) Modell 18

2.3.2 Die Code Generierung 19

2.4 Die Neo4j-Datenbank als Anwendungsbeispiel der NoSQL-Systeme 19

2.4.1 Die Graphdatenbank Neo4j 20

2.4.2 Die Anfragesprache Cypher 21

2.5 Die Programmiersprache Xtend 22

2.6 Das KIELER Projekt 23

2.6.1 Das (Graph-) Visualisierungsframework KLighD 24

2.6.2 Die Synthese 25

2.7 Der Grapheditor von L. Blümke und Y. Benekov 26

2.7.1 Das Modell des Propertygraphen 28

2.7.2 Die Architektur des Grapheditors 29

3 Die hierarchische Konzepte des Grapheditors 33
3.1 Die Kantentypen . . . 33
3.1.1 Die Hierarchie-Kanten . . . 34
3.1.2 Die Vererbungs-Hierarchie-Kanten . . . 34
3.1.3 Die Stop-Hierarchie-Kanten . . . 35
3.1.4 Die Aggregations-Tupel . . . 38
3.2 Das Hierarchie-Menü . . . 39
3.3 Der erweiterte Propertygraph . . . 41
3.4 Die erweiterte Synthese . . . 43

4 Das Neo4j-Reader Plugin zum Importieren eines Graphen 47
4.1 Die Architektur des Neo4j-Reader Plugins . . . 47
4.2 Das Importieren eines SDG . . . 48
4.2.1 Import über das lokale Dateisystem . . . 51
4.2.2 Import über die REST-Schnittstelle von Neo4j . . . 51

5 Das Neo4j-Writer Plugin zum Exportieren eines Graphen 55
5.1 Die Architektur des Neo4j-Writer Plugins . . . 55
5.2 Das Exportieren eines SDG . . . 55
5.2.1 Export über das lokale Dateisystem . . . 56
5.2.2 Export über die REST-Schnittstelle von Neo4j . . . 57

6 Die Erweiterungen des bisherigen Grapheditors 59
6.1 Das Kontextmenü . . . 59
6.1.1 Neue Optionen zum Einfügen von Kindknoten . . . 59
6.1.2 Ausblenden von einzelnen Menüeinträgen . . . 61
6.2 Einfärbung eines Graphen . . . 63
6.3 Löschen von hierarchischen Knoten . . . 65
6.4 Periodisches Speichern der geöffneten Graphen . . . 67

7 Evaluation 69
7.1 Methodik . . . 69
7.1.1 Beschreibung der Experimente . . . 69
7.1.2 Szenarien . . . 76
7.1.3 Setup . . . 79
7.2 Ergebnisse und Diskussion . . . 81
7.2.1 Ergebnisse und Diskussion zu E1: Import über den Neo4j-Reader . . . 81
7.2.2 Ergebnisse und Diskussion zu E2: Export über den Neo4j-Writer . . . 84
7.2.3 Ergebnisse und Diskussion zu E3: Hierarchische Konzepte des Grapheditors . . . 87
7.3 Unsicherheiten bezüglich der Validität . . . 89
7.3.1 Interne Validität . . . 89

7.3.2 Externe Validität 90

8 Verwandte Arbeiten **91**
8.1 Arbeiten in Bezug auf den semi-automatischen Ansatz von C. Wulf 91
8.2 Weitere Grapheditoren 92

9 Fazit und Ausblick **95**
9.1 Fazit 95
9.2 Ausblick 96

Bibliografie **97**

Kapitel 1

Einleitung

1.1. Motivation

Obwohl die heutigen Mehrkernprozessoren bereits vor mehr als einem Jahrzehnt vorgestellt und eingeführt worden sind, wird ihr Potenzial meist nicht vollends ausgeschöpft. Für das parallele Ausführen von mehreren Programmteilen oder einer gesamten Anwendung ist neben dem benötigten Multi-Core-Prozessor ebenfalls ein parallel ausführbarer Programmcode notwendig. Allerdings wird noch heute aufgrund der Komplexität des Paradigmas zur parallelen Programmierung eine Vielzahl von Programmen als sequentieller Quellcode entwickelt. Probleme wie *Dead-* und *Livelocks* oder *Race Conditions* treten bei dieser Art von Code nicht auf, wodurch der Prozess der Softwareentwicklung erleichtert wird.

Um sowohl die Anwendungen aus früherer Zeit als auch die noch heute entwickelten sequentiellen Programme performant auf einem Mehrkernprozessor auszuführen, bedarf es einer Parallelisierung der sich eignenden Programmteile. Schleifenkonstrukte oder der gleichzeitige Aufruf unabhängiger Methoden sind nur einige Beispiele solcher Abschnitte im Quellcode. Allerdings ist das manuelle Parallelisieren von Programmcode sehr fehleranfällig und setzt ein gewisses Maß an Erfahrung voraus. Mittels des im folgenden Abschnitt 1.2 beschriebenen Ansatzes von C. Wulf [Wulf 2014] lässt sich der Prozess der Parallelisierung partiell automatisieren. In Folge dessen wird der Entwickler beim Implementieren paralleler Software entlastet.

1.2. Kontext

Der Ansatz von C. Wulf [Wulf 2014] befasst sich mit einer semi-automatischen Transformation von sequentiellen Programmen zu parallel ausführbaren Programmen. Der Prozess zur Parallelisierung besteht aus mehreren Phasen, welche in der Abbildung 1.1 schematisch dargestellt sind. In der an der CAU zu Kiel entwickelten Anwendung namens *NeoSuit*[1] ist der im Folgenden beschriebene Ansatz implementiert worden.

[1] https://build.se.informatik.uni-kiel.de/chw/integration-project Letzter Zugriff: 29. Oktober 2016

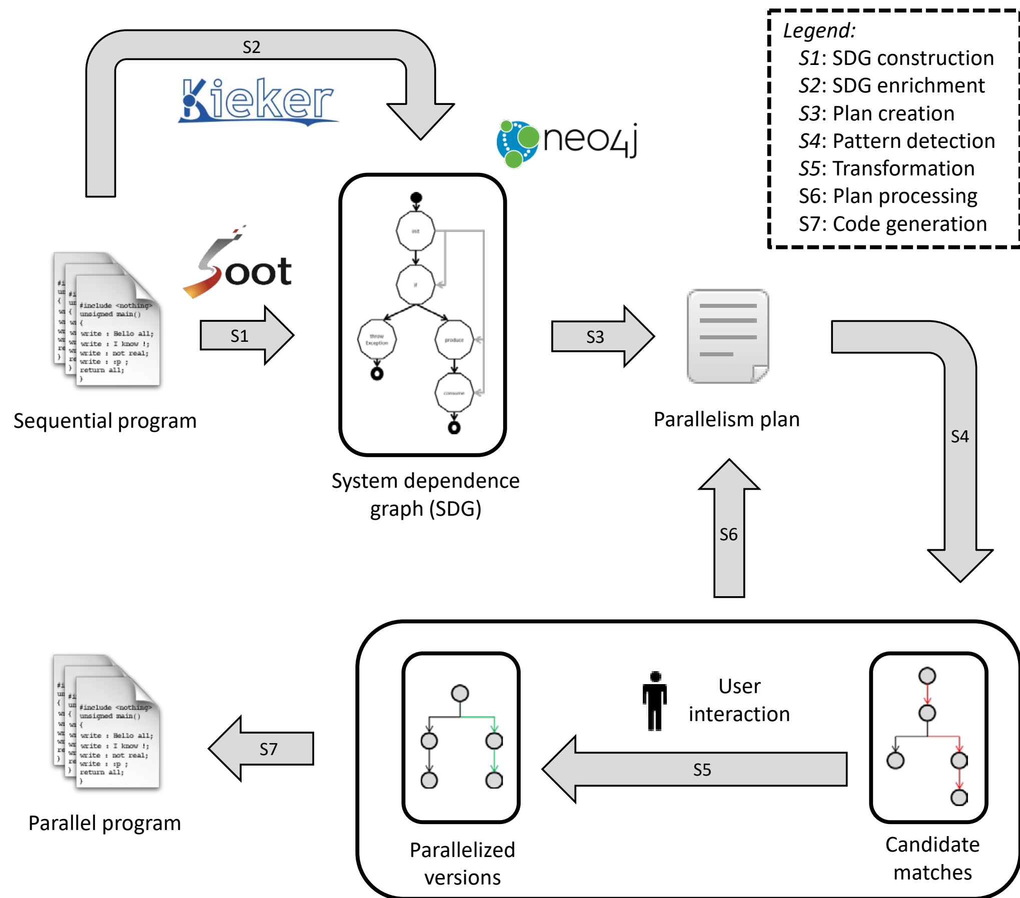

Abbildung 1.1. Semi-automatischer Ansatz zur Parallelisierung eines Programms von C. Wulf [Wulf 2014]

Der erste Schritt besteht in der Erzeugung und persistenten Speicherung eines Systemabhängigkeitsgraph (SDG) von der zu transformierenden Applikation (S1). Dieser wird auf Basis einer statischen Analyse mit dem Java-spezifischen Analyseframework Soot[2] erzeugt. Für zusätzliche Laufzeitinformationen wird das sequentielle Programm mit dem Monitoring-Framework Kieker[3] analysiert und der SDG entsprechend angereichert (S2). Als Backend wird die in Kapitel 2.4 beschriebene Graphdatenbank Neo4j eingesetzt, wodurch der generierte Graph nicht in ein anderes Speicherformat, wie etwa ein Tabellensystem, übertragen wird.

In dem anschließenden Schritt S3 dient der vollständige SDG als Grundlage für einen

[2]https://sable.github.io/soot/ Letzter Zugriff: 29. Oktober 2016
[3]http://kieker-monitoring.net/ Letzter Zugriff: 29. Oktober 2016

sogenannten Parallelisierungsplan. Inhalt dieses Plans ist eine Liste von Teilgraphen, welche, gemessen an zuvor definierten Kriterien, über das größte Potential zur Parallelisierung und einhergehenden Optimierung verfügen. Beispiele für solche Kriterien ist die Laufzeit oder Anzahl an Aufrufen einer Methode. Des Weiteren wird die Reihenfolge der in Schritt S4 auf die Teilgraphen anzuwendenden Candidate-Pattern festgelegt.

Ein Candidate-Pattern beschreibt den zu einem allgemein parallelisierbaren Programmabschnitt korrespondierenden Teil eines SDG. Mittels einer Datenbankabfrage werden, falls vorhanden, die zu dem Parallelisierungsmuster passenden Teilgraphen ermittelt. Zu jedem Candidate-Pattern existiert mindestens ein Parallelization-Pattern, welches eine Modifizierung des vorher selektierten Teilgraphen definiert. Nach der Anwendung des ebenfalls als Datenbankabfrage realisierten Parallelization-Pattern (S5) stellt der resultierende Graph eine parallelisierte Version des sequentiellen SDG bzw. Quellcodes dar. Falls für ein Candidate-Pattern mehr als ein Parallelization-Pattern existiert, entscheidet der Nutzer auf welche Weise der Graph parallelisiert wird.

Sobald eine Transformation abgeschlossen ist, wird der Parallelisierungsplan in Schritt S6 aktualisiert und eventuelle Auswirkungen vermerkt. Beispielsweise könnte durch die Modifikation eines Teilgraphen ein anderer Graphabschnitt nicht mehr parallelisierbar sein. Die noch übrigen Teilgraphen werden wiederum mittels der angegebenen Candidate- und Parallelization-Pattern bearbeitet.

Dieser Kreislauf zum Parallelisieren des ursprünglichen SDG wird unterbrochen, falls keine weiteren Einträge im Parallelisierungsplan existieren oder der Nutzer keine Modifikationen mehr vornehmen möchte. Die Generierung des parallelisierten Quellcodes (S7) erfolgt auf Basis des resultierenden SDG.

Der aus dieser Thesis hervorgehende Grapheditor wird es dem Nutzer ermöglichen ein verbessertes Verständnis für das als SDG dargestellte Programm zu erlangen. Durch die Visualisierung eines komplexen Systems als SDG kann beispielsweise die Erkennung von potenziell parallelisierbaren Schleifen oder Zuweisungen gefördert werden. Zusätzlich wird die Entwicklung der angesprochenen Pattern, welche generische Teilgraphen eines SDG ausdrücken, erleichtert. Die Einführung von Hierarchien dient der Strukturierung der dargestellten Graphen.

1.3. Ziele

Das Ziel dieser Thesis ist die Entwicklung eines, auf den Arbeiten von L. Blümke und Y. Benekov (siehe [Blümke 2015] und [Benekov 2015]) aufbauender, hierarchischer Grapheditor. Dieser Editor wird als Eclipse Plugin entwickelt und soll in erster Linie als unterstützendes Tool für den in Kapitel 1.2 beschriebenen Ansatz von C.Wulf verwendet werden. Im Folgenden werden die dafür benötigten Ziele genauer unterteilt und erläutert.

1.3.1. Z1: Abbildung von hierarchischen Strukturen eines SDG

Der zu entwickelnde Grapheditor wird den bereits bestehenden um hierarchische Strukturen ergänzen. Für einen gröberen Überblick über den aktuellen Graphen oder ein detaillierteres Verständnis von Klassen und Paketen, können Knoten eingekapselt bzw. entfaltet werden. Die Sichtbarkeit von enthaltenen Kindknoten hängt von dem Status des Elternknotens ab. Kanten, die zwischen eingekapselten Knoten verlaufen, werden entsprechend der ausgeführten Aktion ebenfalls sichtbar bzw. verborgen. Der Zustand eines Knoten, ob entfaltet oder eingekapselt, wird vermerkt und gegebenenfalls gespeichert, sodass die für den Nutzer relevanten Knoten beim erneuten Öffnen des Graphen bereits sichtbar sind. Mittels dieser hierarchischen Darstellung kann sich der Nutzer stets auf den für ihn relevanten Teil des Graphen beschränken und erhält aufgrund der geringeren Kantenmenge einen übersichtlicheren Graphen.

Neben der hierarchischen Visualisierung kann ebenfalls auf die flache Darstellung des Graphen, d.h. ohne Verwendung von Eltern- und Kindknoten, zurückgegriffen werden. In diesem Fall wird die Elternknoten-Kindknoten-Relation durch eine Kante vom Elternknoten zum entsprechenden Kindknoten repräsentiert. Der Wechsel zwischen hierarchischer und flacher Darstellung wird jederzeit möglich sein und unterliegt keinen weiteren Bedingungen.

1.3.2. Z2: Import- und Exportfunktionalität von SDG für eine Neo4j-Datenbank

Für eine möglichst flexible Verwendung des Grapheditors werden bei diesem Ziel zwei unabhängige Erweiterungen entwickelt. Somit kann ein Nutzer frei entscheiden, ob er die Funktionalitäten zum Im- und Exportieren verwendet bzw. installiert, oder nur einen Teil dieser.

Der *Neo4j-Reader* wird als eigenständiges Plugin implementiert und kann bei installiertem Grapheditor als Ergänzung verwendet werden. Durch den *Reader* wird die Funktion zum Importieren eines SDG aus einer Neo4j-Datenbank bereitgestellt, sodass der Datenbankzustand als Graph visualisiert wird. Für ein flexibles Arbeiten mit dem Editor wird es möglich sein, den Graphen zeitgleich anzeigen zu lassen und mit einem externen Programm bearbeiten zu können. Dies erlaubt es den Editor als ergänzendes Werkzeug zu verwenden und den gesamten Prozess der Parallelisierung zu erleichtern (vgl. Kapitel 1.1).

Zum Exportieren eines Graphen wird der *Neo4j-Writer*, wie beim *Reader*, ebenfalls als ein eigenes Plugin implementiert und kann dem installierten Grapheditor nachträglich hinzugefügt werden. Folglich können beliebige SDG im Editor erzeugt und für weitere Verwendungen zur Verfügung gestellt werden.

1.3.3. Z3: Anpassungen an das Kontextmenü

Um die hierarchischen Konzepte aus Kapitel 1.3.1 vollständig zu unterstützen, wird das mittels Rechtsklick erreichbare Kontextmenü erweitert. Neben der bereits vorhandenen Menge an Optionen zum Editieren eines Java-spezifischen SDG (JSysDG) aus Abschnitt 2.1.3, wird das Zuweisen von Kindknoten oder Nachfolgern an entfaltete Knoten über neue Menüeinträge erreichbar sein. In Verbindung mit den Zielen aus Kapitel 1.3.2 ist es somit möglich einen SDG zu importieren, ihn nach Belieben zu modifizieren und als neuen Datenbankzustand zu exportieren.

Des Weiteren wird die Übersicht des Kontextmenüs überarbeitet. Zum derzeitigen Stand des Grapheditors werden dem Nutzer, unabhängig von der Auswahl an Elementen, sämtliche Optionen zum Bearbeiten eines Graphen als verfügbar dargestellt. Um eine bessere Übersicht der möglichen Aktionen für die aktuell ausgewählten Elemente zu gewährleisten, sollen nicht verfügbare Menüeinträge ausgeblendet werden. So wird die Option *connect nodes* beispielsweise nur zur Verfügung stehen, falls der Nutzer zuvor genau zwei Knoten markiert hat.

1.3.4. Z4: Erweiterung der bisherigen Bedienbarkeit des Editors

In diesem Kapitel sind mehrere Unterziele zusammengefasst, wobei jedes Ziel für sich genommen eine Erweiterung der bisherigen Bedienbarkeit des Editors darstellt.

1.3.4.1. Z4.1: Erweiterung der bisherigen Optionen zum Einfärben eines Graphen

Zum derzeitigen Stand des Editors ist es möglich die Komponenten eines Graphen unterschiedlich einzufärben. Dabei wird keine Überprüfung vorgenommen, ob bestimmten Arten von Knoten oder Kanten mehrere Farben zugeordnet worden sind, wodurch stets die zuletzt auftretende Farbe verwendet wird. Um dies zu unterbinden, wird eine Überprüfung eingeführt, die den Nutzer darauf hinweist, dass einer Komponente verschiedene Farben zugewiesen worden sind.

Die Farbauswahl für ein Element wird derzeitig über die Eingabe von HTML-Farbcodes durchgeführt. Um dem Nutzer diese Auswahl zu erleichtern, wird zusätzlich das Eingeben von Farbnamen als Strings für die geläufigsten Farbwerte unterstützt.

1.3.4.2. Z4.2: Löschen von hierarchischen Knoten

Die Funktionalität zum Löschen von Knoten wird erweitert, sodass beim Entfernen eines Knotens in der hierarchischen Darstellung eine Unterscheidung bezüglich vorhandener Kindknoten getroffen wird. Falls der zu löschende Knoten über keine Kindknoten verfügt, wird sowohl der Knoten selbst als auch die ein- und ausgehende Kanten bei den Nachfolgern bzw. Vorgängern entfernt. Bei vorhandenen Kindknoten hingegen wird neben dem ursprünglichen Knoten ebenfalls jeder enthaltene Knoten mitsamt den entsprechenden Verweisen rekursiv gelöscht. Somit unterliegt das Löschen von hierarchischen Knoten

einem intuitiven Verständnis. Der Nutzer muss beispielsweise beim Entfernen eines Klassenknotens nicht jeden enthaltenen Methodenknoten manuell löschen.

1.3.4.3. Z4.3: Periodisches Speichern der Graphen

Für dieses Ziel wird wie bei dem *Neo4j-Reader* und *Writer* aus Kapitel 1.3.2 ein eigenständiges Plugin entwickelt. Bei Verwendung dieser Erweiterung kann der Nutzer eine Funktion zum periodischen Speichern der geöffneten Graphen aktivieren bzw. deaktivieren. Über ein beliebig einstellbares Zeitintervall wird festgelegt, in welchen Abständen die Graphen im Hintergrund gespeichert werden. Der Speichervorgang wird dabei in einem separaten Thread ausgeführt, sodass für den Nutzer keine Einschränkungen entstehen.

1.3.5. Z5: Evaluation des Grapheditors

In diesem Kapitel werden die angestrebten Ziele zur Evaluation des Editors beschrieben. Für das Evaluieren der hierarchischen Darstellung aus Kapitel 1.3.1 wird angenommen, dass die importierten Graphen in der flachen Repräsentation ein genaues Abbild der ursprünglichen Datenbank sind. Um diese Voraussetzung zu schaffen, wird die Evaluation der Import- und Exportfunktionalität, wie in der Reihenfolge der nächsten Kapitel zu sehen, als erstes durchgeführt. Grundlage sämtlicher durchzuführender Evaluationen ist das Goal-Question-Metric-Modell [Van Solingen u. a. 2002], welches in Kapitel 7.1 genauer erläutert wird.

1.3.5.1. Evaluation der Import- und Exportfunktionalität

In diesem Ziel wird die Evaluation der in Kapitel 1.3.2 beschriebenen Funktionen zum Im- und Exportieren von allgemeinen Neo4j-Datenbanken beschrieben. Beim Evaluieren der Importfunktionalität wird auf ein breites Spektrum an unterschiedlichen Datenbankmodellen, wie etwa einer Smartphonedatenbank und einem SDG einer Java-Anwendung, zurückgegriffen. Jede Datenbank wird mittels des *Neo4j-Readers* auf mehreren Wegen importiert und anschließend dem im Neo4j-Browser visualisierten Original gegenübergestellt. Über einen systematischen Vergleich mit der flachen Darstellung des aus dieser Arbeit resultierenden Editors kann entschieden werden, ob beim Importieren ein exaktes Abbild der ursprünglichen Datenbank erstellt worden ist.

Für die Evaluation der Exportfunktionalität, werden die zuvor importierten und als korrekt verifizierten Graphen in eine neue Neo4j-Datenbank exportiert. Diese neu erstellten Datenbanken werden anschließend auf ihre interne Korrektheit überprüft und mit den im vorherigen Schritt ursprünglich importierten Datenbanken verglichen. Mittels des Neo4j-Browsers kann ein systematischer Vergleich sowohl über die vorhandenen Knoten und Kanten als auch über die Metainformationen der beteiligten Datenbanken durchgeführt werden.

1.3.5.2. Evaluation der hierarchischen Konzepte

Nachdem das Importieren von SDG erfolgreich evaluiert worden ist, wird die in Kapitel 1.3.1 beschriebene hierarchische Darstellung eines Graphen überprüft. Für die Evaluation der in Kapitel 3.1 beschriebenen Kantentypen reicht eine Vielzahl an kleineren Programmen mit verschiedenen Kontrollflussstrukturen, wie Verzweigungen oder Schleifen, aus. Neben den klassischen Paket-Klassen- oder Klasse-Methoden-Hierarchien kann so sichergestellt werden, dass die hierarchischen Konzepte auch auf der Ebene des Kontrollflusses korrekt angewendet werden. Die Analyse ganzer Frameworks ist somit nicht notwendig.

1.4. Aufbau

Die Grundlagen für den aus dieser Arbeit resultierenden Grapheditor werden in Kapitel 2 erläutert. Neben einer Einführung in die Graphentheorie wird ebenfalls auf die technischen Komponenten eingegangen. In Kapitel 3 werden die implementierten Konzepte zur Realisierung der Hierarchie aus Abschnitt 1.3.1 beschrieben. Die beiden folgenden Kapitel 4 und 5 befassen sich mit dem Ziel 1.3.2 zum Im- und Exportieren eines Graphen aus bzw. in eine Neo4j-Datenbank. Die allgemeinen Anpassungen an den Grapheditor, wie etwa das Erweitern des Kontextmenüs (siehe Abschnitt 1.3.3) oder das Löschen von hierarchischen Knoten, werden in Kapitel 6 erläutert. Für jedes der untergeordneten Ziele aus Abschnitt 1.3.4 besteht ein separates Unterkapitel, in dem auf die jeweilige Implementierung eingegangen wird. In Kapitel 7 wird die systematisch durchgeführte Evaluation der in Abschnitt 1.3.5 aufgeführten Ziele beschrieben. Die verwandten Arbeiten werden in Kapitel 8 erwähnt und in einen Zusammenhang gebracht. In Kapitel 9 wird sowohl ein abschließendes Fazit als auch ein Ausblick auf mögliche nachfolgende Arbeiten gegeben.

Kapitel 2

Grundlagen und Technologien

2.1. Graphentheorie

Obwohl die Graphentheorie ursprünglich aus der Mathematik kommt, gibt es in der Informatik viele Anwendungsfälle für Graphen. Sie werden im Bereich der Softwareanalyse und Softwarearchitektur sowohl zum Repräsentieren von Systemen als auch zum Aufdecken von ungelösten Problemen verwendet.

Nach R. Diestel [Diestel 1996] besteht ein Graph G aus zwei Mengen E und V, wobei E die Menge der Knoten und V die Menge der Kanten von G darstellt. Eine Kante v_1 besteht immer aus genau zwei Knoten e_1, e_2 und beschreibt eine vorhandene Relation zwischen diesen. Je nachdem, ob es sich bei G um einen gerichteten oder ungerichteten Graphen handelt, lassen sich e_1 und e_2 als Start- bzw. Endknoten definieren. Sollten in dem Graphen G mehrere Kanten zwischen e_1 und e_2 existieren, so handelt es sich um einen Multigraphen.

Diese mathematischen Definitionen eines Graphen sind die Grundlage der im folgenden beschriebenen Graphmodelle, wie etwa dem Java-spezifischen SDG aus Abschnitt 2.1.3. Bei allen drei Modellen handelt es sich um gerichteten Multigraphen, welche jeweils aufeinander aufbauen.

2.1.1. Der Eigenschaftsgraph

Mit dem Eigenschafts- oder auch Propertygraphen wird eine sehr allgemeine Art von gerichteten Multi-Graphen referenziert, bei denen die Knoten und Kanten mit Labels und Eigenschaften (meist Schlüssel-Wert-Paare) angereichert werden [Rodriguez und Neubauer 2010]. Für Knoten bzw. Kanten gibt es keine unterschiedlichen Typen, sodass die Unterscheidung lediglich auf Basis der angefügten Informationen möglich ist. Aus dem Modell des Eigenschaftsgraphen lassen sich eine Vielzahl anderer Graphmodelle ableiten. Beispielsweise können ungerichtete oder gewichtete Graphen durch das Entfernen der Kantenrichtungen oder das Hinzufügen von Gewichtsinformation erstellt werden. Abbildung 2.1 stellt eine Übersicht der auf dem Propertygraph-Modell aufbauenden Graphen dar.

Aufgrund des allgemein gehaltenen Modells des Eigenschaftsgraph wird dieses oftmals für Graphdatenbanken, wie das in Kapitel 2.4 erläuterte Neo4j-Datenbanksystem, verwendet. Um die Verwendung des resultierenden Grapheditors möglichst flexibel zu gestalten, dient ein modifiziertes Propertygraphmodell (siehe Kapitel 3.3) als grundlegendes Format

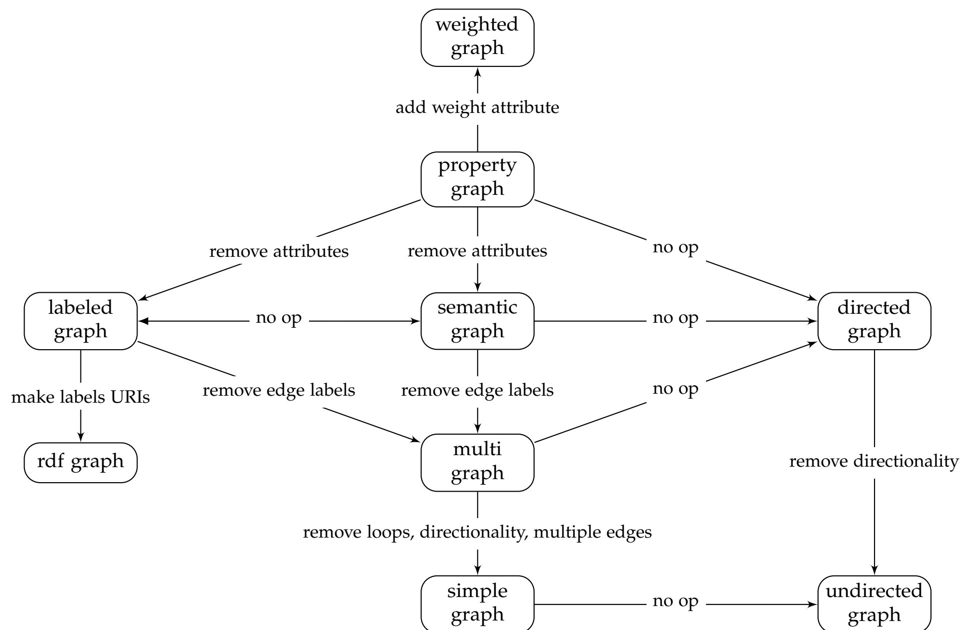

Abbildung 2.1. Zusammenhang von verschiedenen Graphmodellen und dem Propertygraph als Zentrum [Rodriguez und Neubauer 2010]

zum Speichern der visualisierten Graphen. Neben den Knoten- und Kanteninformationen werden zusätzlich allgemeine Metadaten über den Graphen und die beinhalteten Elemente gespeichert.

2.1.2. Der Systemabhängigkeitsgraph

Der Systemabhängigkeitsgraph (SDG) ist eine interprozedurale Repräsentation eines auf Funktionen und Funktionsaufrufen basierenden Programms [Horwitz u. a. 1988]. Grundlegend hierfür sind die einzelnen Programmabhängigkeitsgraphen der unterschiedlichen Funktionen, welche interne Kontroll- und Datenflussabhängigkeiten darstellen [Ottenstein und Ottenstein 1984]. Dabei besteht für zwei Statements S1 und S2 eine Datenabhängigkeit, falls beim Vertauschen der Statements einer vorkommenden Variable ein falscher Wert zugewiesen wird [Ferrante u. a. 1987]. Ein Beispiel hierfür ist in Listing 2.1 angegeben.

Listing 2.1. Datenabhängigkeit zwischen zwei Statements

```
1    Y = X * X
2    Z = Y * Y
```

Ist die Ausführung eines Statements S hingegen abhängig von dem Wert eines Prädikates P, besteht eine Kontrollflussabhängigkeit zwischen P und S, siehe Listing 2.2.

Listing 2.2. Kontrollabhängigkeit zwischen einem Prädikat (1) und einem Statement (2)

```
1   if (Y) then
2      X = X + X
3   endif
```

Der finale Systemabhängigkeitsgraph setzt sich schließlich aus den Programmabhängigkeitsgraphen und zusätzlichen Caller-Callee-Kanten, welche den Zusammenhang zwischen aufrufenden Statements und aufgerufenen Funktionen herstellen, zusammen. Zusätzlich werden für entstandene, transitive Abhängigkeiten weitere Kanten in den SDG eingefügt.

2.1.3. Der Java-spezifische Systemabhängigkeitsgraph

Das Modell des im vorherigen Kapitel 2.1.2 beschriebenen allgemeinen SDG unterliegt keiner Programmiersprache und kann daher für beliebige Programme erstellt werden. Um die Vorteile und Besonderheiten einzelner Sprachen besser in einem Graphen abbilden zu können, gibt es verschiedene Modelle, welche den allgemeinen SDG erweitern und für spezifische Sprachen anpassen. In diesem Kapitel wird der Java-spezifische SDG (JSysDG) erläutert [Walkinshaw u. a. 2003]. Dieser stellt das zugrundeliegende Modell der mit dem Ansatz von C. Wulf (siehe 1.2) generierten Graphen dar.

Der JSysDG basiert auf mehreren Ebenen, wodurch der Graph stets auf das für den Betrachter Wesentliche und Relevante reduziert werden kann. Die unterste der insgesamt fünf Ebenen ist die Anweisungsebene. Darauf aufbauend existiert die Methodenebene, welche wiederum die Grundlage der übergeordneten Klassenebenen ist. Die Schnittstellenebene erweitert die Klassenebene und bildet die Basis für die abschließende Paketebene. Je nach gewählter Ebene und dem korrespondierenden Graphen stehen unterschiedliche Typen von Knoten und Kanten zur Verfügung. Die Tabelle 2.1 stellt einen Überblick der für die einzelnen Graphen verfügbaren Knoten- und Kantentypen dar.

Jeder *Statement Node* in einem Anweisungsgraphen repräsentiert genau eine Anweisung aus der zum Graphen gehörenden Funktion. Diese Knoten können sowohl über eine *Control Dependence Edge*, als auch über eine *Data Dependence Edge* miteinander verbunden sein. Die Semantik dieser Kanten ist dabei gleich zu der Kontroll- bzw. Datenabhängigkeit aus Kapitel 2.1.2.

Um Zusammenhänge zwischen einem funktionsaufrufenden Statement und der entsprechenden Funktion darstellen zu können, muss der Methodenabhängigkeitsgraph erstellt werden. Der *Method Entry Node* repräsentiert den Einstiegspunkt der Funktion und ist mit jedem zugehörigen (Statement-) Knoten, über eine *Control Dependence Edge* verbunden. Die Parameterübergabe wird mittels *Actual In/Out* und *Formal In/Out* dargestellt. Über weitere *Control Dependence Edges* werden dem funktionsaufrufenden Knoten *Actual In/Out Nodes* angehängt, die das Schreiben und Lesen einer temporären Variable repräsentieren. Auf der Gegenseite werden ebenfalls über *Control Dependence Edges* dem *Method Entry Node Formal In/Out Nodes* angehängt, um das korrespondierende Lesen bzw. Schreiben dieser temporären Variablen darzustellen. Diese Knotenpaare von *Actual In / Formal In* und

Tabelle 2.1. Unterschiedliche Ebenen mit dazugehörigen Knoten- und Kantentypen des JSysDG [Walkinshaw u. a. 2003]

Paketabhängigkeitsgraph:		
Knoten	Package Entry	Repräsentation eines Pakets
Kanten	Package Member	Verknüpfung von Paketen und Klassen bzw. Schnittstellen
Schnittstellenabhängigkeitsgraph:		
Knoten	Interface Entry	Repräsentation einer Schnittstelle
	Parameter	Repräsentation von Eingabe- bzw Rückgabeparametern einer Schnittstelle
Kanten	Abstract Member	Verknüpfung von Schnittstelle und abstrakter Methodendeklaration
	Implement Abstract Method	Verknüpfung von abstraker Methodendeklaration und Implementierung
	Implements	Verknüpfung von Schnittstelle und implementierender Klasse
Klassenabhängigkeitsgraph:		
Knoten	Class Entry	Repräsentation einer Klasse
Kanten	Class Membership	Verknüpfung von Klasse und Methoden
	Data Member	Verknüpfung von Klasse und Attributen
	Class Dependence	Vererbungsrelation
Methodenabhängigkeitsgraph:		
Knoten	Method Entry	Repräsentation einer Methode
	Actual In	Schreiben einer temporären Variable (funktionsaufrufender Statement-Knoten)
	Actual Out	Lesen einer temporären Variable (funktionsaufrufender Statement-Knoten)
	Formal In	Lesen einer temporären Variable (aufgerufener Methoden-Knoten)
	Formal Out	Schreiben einer temporären Variable (aufgerufener Methoden-Knoten)
Kanten	Call Dependence	Verknüpfung zwischen funktionsaufrufendem Statement-Knoten und aufgerufenem Methoden-Knoten
	Parameter In	Verknüpfung von korrespondierenden Atual In/Formal In Knotenpaaren
	Parameter Out	Verknüpfung von korrespondierenden Atual Out/Formal Out Knotenpaaren
Anweisungsgraph:		
Knoten	Statement	Repräsentation eines Statements
Kanten	Control Dependence	Verknüpfung von Statements bei Kontrollflussabhängigkeit
	Data Dependence	Verknüpfung von Statements bei Datanabhängigkeit

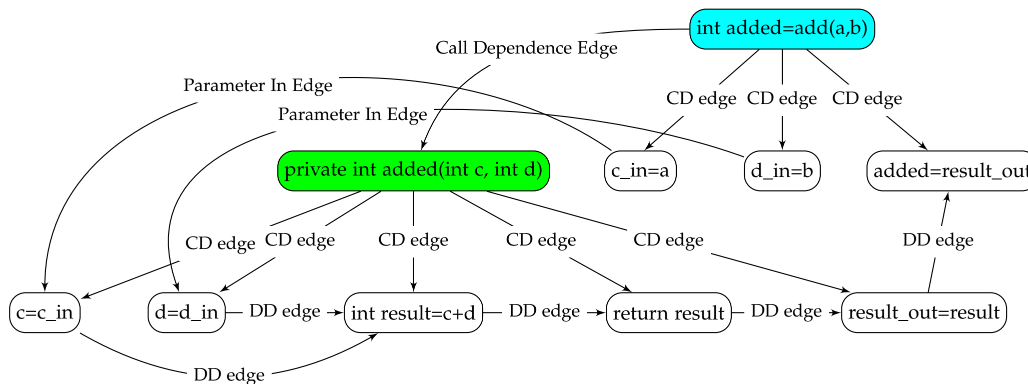

Abbildung 2.2. Beispiel eines Methodenabhängigkeitsgraphen mit einem funktionsaufrufenden Knoten (hellblau) und einem *Method Entry Node* (grün) [Walkinshaw u. a. 2003]. *Data Dependence Edge* und *Control Dependence Edge* wurde mit *DD Edge* bzw. *CD Edge* abgekürzt.

Actual Out / Formal Out werden schließlich mittels *Parameter In/Out Edges* verbunden und verdeutlichen das Java-spezifische Call-by-Value-Verhalten. Die Caller-Callee-Beziehung zwischen dem funktionsaufrufenden Knoten und dem *Method Entry Node* wird über eine *Call Dependence Edge* repräsentiert.

In Abbildung 2.2 ist ein simpler Methodenabhängigkeitsgraph für die Funktion *private int added(int c, int d)* zu sehen. Der grün dargestellte *Method Entry Node* ist mit allen zur Methode gehörenden *Statement Nodes* (*int result=c+d* und *return result*) über eine *Control Dependence Edge* verknüpft. Der Methodenaufruf ist durch die eingehende *Call Dependence Edge* visualisiert, welche als Startknoten das zu einer anderen Methode gehörende funktionsaufrufende Statement (hellblau) hat. Die übrigen Knoten, wie *c_in=a* und *c=c_in*, stellen die bereits erwähnten *Actual In / Formal In* bzw. *Actual Out / Formal Out* Paare dar und sind jeweils mit einer *Parameter In/Out Edge* verbunden. Die Datenabhängigkeit innerhalb der Methode wird durch die *Data Dependence Edges* zwischen den einzelnen *Statement Nodes* repräsentiert.

Für das Abbilden der Klassenhierarchie eines Java Programms muss der auf dem Methodenabhängigkeitsgraphen aufbauende Klassenabhängigkeitsgraph erstellt werden. Jede Klasse wird durch einen *Class Entry Node* repräsentiert und ist mit den zur Klasse gehörenden *Method Entry Nodes* mittels *Class Membership Edges* verknüpft. Attribute einer Klasse werden durch *Data Member Edges* mit dem Klassenknoten verbunden. Je nach Sichtbarkeit dieser Methoden und Attribute können die Kanten mit einem *public*, *protected* oder *package* Tag versehen werden. Zum Ausdruck einer Vererbungsrelation zwischen zwei Klassen werden die beteiligten *Class Entry Nodes* durch eine *Class Dependence Edge* verbunden.

Der Schnittstellenabhängigkeitsgraph wird verwendet, um Schnittstellen und abstrakte

Klassen zu repräsentieren. Eine Schnittstelle wird mit einem *Interface Entry Node* dargestellt, welcher über *Abstract Member Edges* zu jeder im Interface angegebenen abstrakten Methode (*Method Entry Node*) verknüpft ist. Für Eingabeparameter oder Rückgabewerte werden *Parameter Nodes* eingeführt, die mit dem *Method Entry Node* der entsprechenden abstrakten Methode verbunden sind. Der Zusammenhang zwischen einer abstrakten Methodendeklaration und ihrer Implementierung wird durch eine *Implement Abstract Method Edge* zwischen den beiden *Method Entry Nodes* ausgedrückt. Falls eine Schnittstelle von einer Klasse implementiert sein sollte, wird dies mit einer *Implements Edge* von dem *Interface Entry Node* zum *Class Entry Node* dargestellt. Das Abbilden von abstrakten Klassen ist analog, nur dass anstatt eines *Interface Entry Node* ein *Class Entry Node* als Repräsentation der abstrakten Klasse gewählt wird.

Um die Paketstruktur eines Java Programms darzustellen, muss der Paketabhängigkeitsgraph konstruiert werden. Neben all den bereits erwähnten Knoten- und Kantentypen verfügt dieser über sogenannte *Package Entry Nodes*, zur Repräsentation der Pakete. Klassen und Schnittstellen, die zu einem Paket gehören, sind mit einer *Package Member Edge* mit dem entsprechenden *Package Entry Node* verbunden.

In Abgrenzung zu dem von C. Wulf verwendeten Graphmodell bietet der Java-spezifische SDG keine Kanten zum Abbilden eines direkten Kontrollflusses. Sowohl die *Control* als auch die *Data Dependence Edge* eignen sich lediglich zum Darstellen von Abhängigkeiten zwischen verschiedenen Statements. Diese Kantentypen reichen jedoch für den in Kapitel 1.2 vorgestellten Ansatz zur semiautomatische Parallelisierung einer Anwendung nicht aus. Das Modell der vom Editor importierten Graphen ist demnach eine um Kontrollflusskanten angereicherte Erweiterung des hier beschriebenen JSysDG.

2.2. Die Entwicklungsumgebung Eclipse

Bei *Eclipse* handelt es sich um ein *Open Source*-Werkzeug zur Entwicklung von Software unterschiedlicher Art. Obwohl es anfänglich mit seiner integrierten Entwicklungsumgebung (IDE) für die Erstellung von Programmen in der Programmiersprache *Java* vorgesehen war, wird die *Eclipse-Platform* heutzutage auch als Grundlage für reguläre (Geschäfts-) Anwendungen verwendet [Steppan 2015]. Dies liegt an der einfachen Erweiterbarkeit des zugrundeliegenden Plugin-Systems und dem minimalen Kern. Die Grundfunktionalitäten von *Eclipse* können somit an beliebige Bedürfnisse und gestellte Aufgaben angepasst werden, um beispielsweise die Entwicklung von Programmen in den Sprachen *Xtend* oder *C++* zu erleichtern.

Aufgrund der Weiterentwicklung des in Kapitel 2.7 beschriebenen Grapheditors, wird die aus dieser Arbeit resultierende Applikation ebenfalls ein Eclipse-Plugin sein. Folglich wird für Verwendung des erweiterten Editors eine installierte Eclipse-Umgebung vorausgesetzt.

In den folgenden Kapiteln wird einerseits die für die Plugin-Entwicklung grundlegende *Rich Client Platform* beschrieben, sowie der Lebenszyklus eines implementierten Plugins.

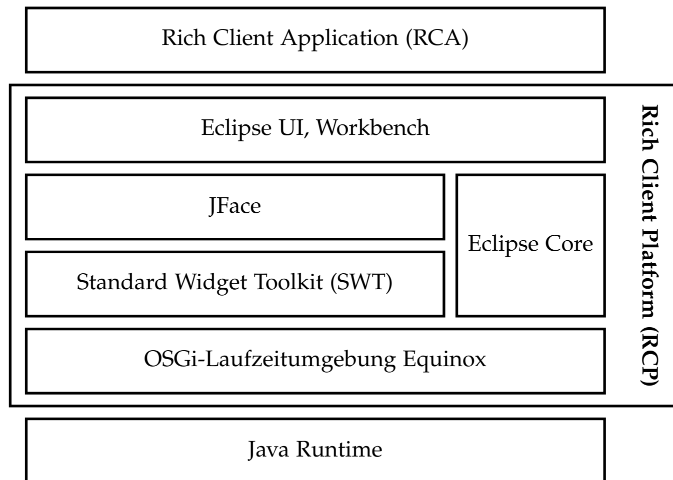

Abbildung 2.3. Zusammenhang der RCP-Komponenten [Ebert 2011]

2.2.1. Die Rich Client Platform

Die *Rich Client Platform* bildet die Grundlage für das Entwickeln von *Rich Client Application* (RCA), wie dem aus dieser Arbeit resultierende Grapheditor. Für ein besseres Verständnis der im folgenden beschriebenen Komponenten, stellt Abbildung 2.3 eine Übersicht der enthaltenen Bibliotheken und Umgebungen dar.

Über die Laufzeitumgebung *Equinox* werden die verschiedenen Plugins eines laufenden Systems verwaltet und können unter anderem dynamisch gestartet oder beendet werden. *Equinox* bildet das Fundament der RCP und implementiert die *Open Services Gateway initiative*-Spezifikation (OSGi), welche ein internationales Standard für Modulplattformen ist. Die genauen Zustände eines Plugins werden im folgenden Kapitel 2.2.2 über den Lebenszyklus eines Plugins erläutert. Des Weiteren stellt die *Eclipse-Core*-Bibliothek allgemeine Funktionalitäten, wie den Zugriff auf das lokale Dateisystem oder das Erstellen von aussagenlogischen Formeln, für Eclipse-Anwendungen bereit.

Für das Erstellen von graphischen Oberflächen wird die GUI-Bibliothek *Standard Widget Toolkit* (SWT) und das darauf aufbauende *JFace*-Toolkit verwendet. SWT bildet über das *Java Native Interface* (JNI) eine Schnittstelle zu den nativen Basiskomponenten des Betriebssystems, wie Buttons oder einfachen Tabellen. Der Vorteil an diesem Ansatz ist, dass im Gegensatz zu den Java-Bibliotheken AWT/Swing das Zeichnen der GUI-Elemente direkt von dem Betriebssystem übernommen wird. Die Folge hingegen ist das manuelle Freigeben von Speicher für nicht mehr benötigte Widgets, da SWT kein Bestandteil der Java-Laufzeitumgebung ist.

Die reine Java-Bibliothek *JFace* bietet eine weitere Abstraktionsschicht und Zugriff auf höherwertige, aus SWT-Elementen zusammengesetzte, UI-Komponenten. Dabei ist

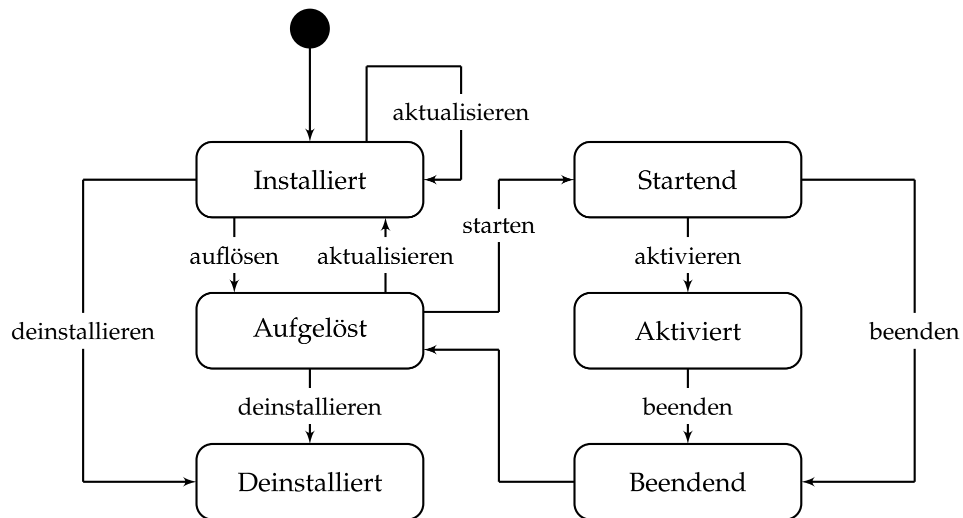

Abbildung 2.4. Lebenszyklus eines Plugins [Steppan 2015]

festzuhalten, dass *JFace* dem Entwickler als ergänzendes und nicht SWT-kapselndes Toolkit zur Verfügung steht. In Verbindung mit der *Eclipse-Core*-Bibliothek ist es ebenfalls möglich, eine *Model-View-Controler*-Architektur (MVC) zu implementieren.

Die für den Nutzer sichtbare *Workbench* wird mittels der *Eclipse-UI*-Bibliothek erstellt. Sie stellt ein erweiterbares Grundgerüst für die verschiedenen GUI-Elemente aus den zuvor beschriebenen Toolkits dar. Neben Menüleisten oder Dialogen können auch Fenster (Parts) zum Anzeigen von Inhalten (Views) bzw. Editieren von Inhalten (Editoren) erstellt und in einer Perspektive beliebig angeordnet werden. Das Erstellen einer Benutzeroberfläche unterliegt somit keinen Einschränkungen.

2.2.2. Der Lebenszyklus eines Plugins

Für das Verwalten der Plugins wird jedes Modul über eine *Service Registry* in der Laufzeitumgebung *Equinox* eingetragen. Dieser Registrierungsmechanismus basiert auf einer Datenbank, welche für jedes Plugin einen Eintrag mit entsprechender Versionsnummer beinhaltet. So kann bei Abhängigkeiten zwischen verschiedenen Plugins überprüft werden, ob die geforderte Mindestversion zur Verfügung steht. Des Weiteren ist es möglich, das gleiche Plugin mit unterschiedlichen Versionen zu verwalten.

Nach der Registrierung eines Moduls in dem OSGi-Framework nimmt das Plugin den Zustand *Installiert* an (siehe Abbildung 2.4). Falls die Abhängigkeiten nicht aufgelöst werden können, bleibt dieser Zustand bestehen und die enthaltenen Pakete bzw. Funktionalitäten stehen nicht zur Verfügung. Sollten alle angegebenen Abhängigkeiten aufgelöst worden sein, wechselt der Status des Plugins in *Aufgelöst*. Hiermit wird ausgedrückt, dass

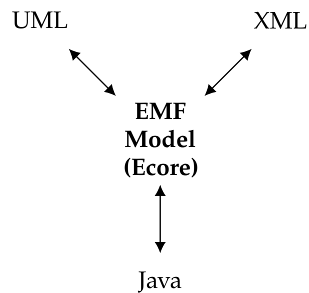

Abbildung 2.5. Zusammenhang der unterstützen Repräsentationen [Steinberg u. a. 2009]

die Erweiterungen des Moduls verwendet werden können. Allerdings kann der Zustand *Installiert* jederzeit durch ein Update und den damit verbundenen neuen Abhängigkeiten erreicht werden.

Ist ein Plugin gestartet worden, nimmt es den Übergangsstatus *Startend* an. Sobald es betriebsbereit ist und die Rich Client Platform um die entsprechenden Funktionalitäten erweitert worden ist, wechselt der Zustand des Moduls zu *Aktiviert*. Der *Beendend* Status kann während des Startvorgang oder nach dem Aktivieren durch das Stoppen des Plugins erreicht werden. Nach erfolgreichem Beenden, wird erneut der Zustand *Aufgelöst* angenommen.

Deinstalliert werden kann ein Plugin nur solange es nicht gestartet worden ist. Dabei ist es unerheblich, ob die angegebenen Abhängigkeiten bereits aufgelöst worden sind oder nicht.

2.3. Das Eclipse Modeling Framework

Modelle sind in der Informatik ein häufig verwendetes Mittel um Strukturen bzw. Sachverhalte, wie den Aufbau eines Computersystems oder Datenbankschemas, zu repräsentieren. Für die Verwendung eines spezifizierten Modells ist es im Bereich der Softwareentwicklung oftmals notwendig auf mehrere Repräsentationen zurückzugreifen. Beispielsweise genügt ein UML-Klassendiagramm nicht aus, falls erzeugte Klasseninstanzen einheitlich in einer XML-Datei gespeichert werden soll. Das *Eclipse Modeling Framework* (EMF) ermöglicht es dem Entwickler eine Repräsentation eines Modells in eine andere Form zu überführen. Neben annotierten Java Schnittstellen werden die Technologien UML und XML-Schema unterstützt (siehe Abbildung 2.5).

Die Grundlage der Modelltransformation bildet das *Ecore*-Modell, welches aus den drei erwähnten Formen erzeugt wird. Neben den drei aufgezählten Formaten lässt sich ebenfalls der Quellcode zur Implementierung des beschriebenen Modells automatisch generieren. Zusätzlich können Editoren zum Bearbeiten einer generierten Modellinstanz

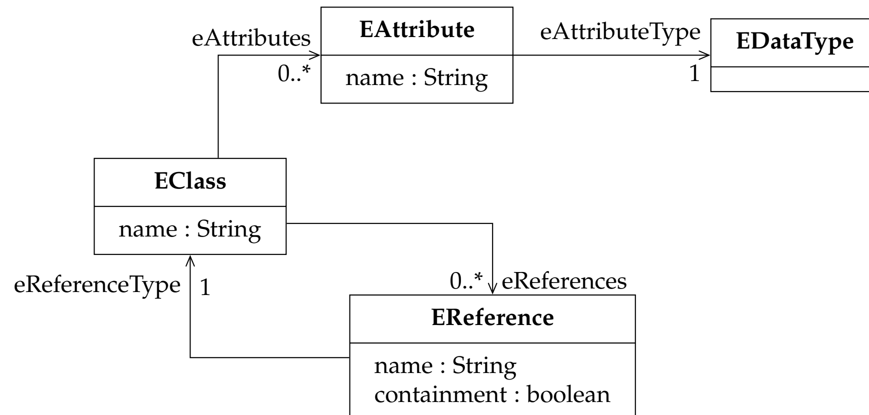

Abbildung 2.6. Eine vereinfachte Teilmenge des ursprünglichen *Ecore*-Metamodells [Steinberg u. a. 2009]

erzeugt werden.

In dieser Arbeit wird das EMF zum Erstellen des in Kapitel 2.7.1 eingeführten *Propertygraph*-Modells verwendet. Dies stellt die Grundlage der in dem bisherigen Editor visualisierten Graphen dar und wird in Abschnitt 3.3 zu einem erweiterten Modell ergänzt.

2.3.1. Das Ecore (Meta) Modell

Wie bereits erwähnt, bildet das *Ecore*-Modell die Grundlage sowohl für das Generieren der Implementierung als auch der drei Repräsentationen: Java-Schnittstellen, XML-Schema und UML. In dem Modell sind die zulässigen Strukturen der im *Eclipse Modeling Framework* repräsentierten Modelle beschrieben, sodass es sich bei *Ecore* ebenfalls um ein sogenanntes Metamodell handelt. In Abbildung 2.6 ist eine vereinfachte Version des Metamodells dargestellt [Steinberg u. a. 2009].

Der Knoten *EClass* wird für die Repräsentation von Klassen verwendet. Neben dem Namen kann eine Klasse über beliebig viele Attribute und Referenzen auf andere Klassenknoten verfügen. Ein modelliertes Attribut wird über den Knoten *EAttribute* repräsentiert und ist mit genau einem *EDataType*-Objekt verknüpft, welches sowohl einen primitiven als auch objektbezogenen Datentypen referenzieren kann. Über die Knoten *EReference* werden Relationen zwischen Klassen ausgedrückt. Das Feld *containment* beschreibt den Zusammenhang der beteiligten Klassen und lässt sich im Zustand *wahr* mit der Kompositionsrelation aus UML vergleichen.

Um den Eindruck eines bevorzugten Standardmodells zu unterbinden, wird für das Serialisieren einer *Ecore*-Modellinstanz eine vierte Art der Repräsentation verwendet: *XML Metadata Interchange* (XMI). Anders als bei den bisherigen Formaten ist XMI ein etablierter Standard zum Austausch von Metadaten und verhindert die Anreicherung des

spezifizierten Modells mit zusätzlichen Informationen.

2.3.2. Die Code Generierung

Bei der Generierung des Java-Quellcodes wird für jede *Ecore*-Klasse (*EClass*) ein Interface und eine korrespondierende, implementierende Klasse erzeugt. Nach selbigem Schema wird zusätzlich eine *Factory* bereitgestellt, wodurch das Erstellen von Objektinstanzen dem Entwurfsmuster *factory method* unterliegt. Der Zugriff auf die *Ecore*-Metadaten eines Modells, wie etwa der Anzahl an Operationen einer Klasse im zugrundeliegenden Modell, ist mittels einer weiteren *Package*-Klasse und Schnittstelle realisiert.

Der generierte Quellcode kann jederzeit vom Entwickler angepasst werden, sodass auch bei einer erneuten Generierung die manuell erstellten Methoden und Attribute beibehalten werden. Über ein *@generated*-Tag kann entschieden werden, welche Teile des Quellcodes vom Generator erzeugt worden sind und ob diese überschrieben werden sollen. Zusätzlich ist es möglich neben einer eigens geschriebenen Methode ebenfalls eine mit dem gleichen Namen generierte Methode zu verwalten. Durch das Erweitern eines Methodennamens mit dem Suffix *gen* wird der generierte Methodenrumpf an die entsprechende Stellen im Quellcode weitergeleitet und die eigene Methode nicht überschrieben.

Neben der Quellcodegenerierung eines spezifizierten Modells, unterstützt der EMF-Generator auch das Erzeugen von Werkzeugen zum Umgang mit einer Modellinstanz. Hierfür werden die zwei Module *EMF.Edit* und *EMF.Editor* generiert, wobei die UI-abhängigen Komponenten in das Letztere ausgelagert werden. Über einen Wizard oder Editor mit Baumstruktur können neue Modellinstanzen angelegt bzw. bestehende dargestellt und bearbeitet werden. In dem Modul *EMF.Edit* werden die für eine Visualisierung benötigten *Content-* und *Label-Provider* zum Zugriff auf die anzuzeigenden Informationen eines Objektes bereitgestellt.

Jedes der erwähnten Module wird als eigenständiges Eclipse-Plugin mit den entsprechend benötigten Dateien, wie der *MANIFEST*-Datei, erstellt. Auf diese Weise kann das Modell ohne nachträgliche Modifikationen in andere Eclipse-Anwendungen integriert werden. Grundlage der Generierung ist neben der *Ecore*-Instanz ein um zusätzliche Informationen angereichertes *Generator*-Modell, welches das spezifizierte Modell umschließt. Dieser Ansatz erlaubt ein separates Verwalten der lediglich für das Generieren des Quellcodes relevanten Daten, wie beispielsweise die Speicherorte der erzeugten Plugins.

2.4. Die Neo4j-Datenbank als Anwendungsbeispiel der NoSQL-Systeme

Die Vielzahl an verfügbaren Datenbankensystemen implementieren ein Modell, in dem gespeicherte Daten in Form von Tabellen vorliegen. Obwohl dies heutzutage ein etablierter Standard für Datenbanken ist, gibt es neben den relationalen Systemen eine Reihe weiterer, die über einen anderen Ansatz verfügen. NoSQL (Not only SQL) ist ein Begriff, der

für Datenbanken verwendet wird, bei denen das zugrundeliegende Datenmodell nicht relational ist. Aufgrund der verschiedenen nicht-relationalen Datenmodellen unterliegen NoSQL-Systeme jedoch keiner eindeutigen Definition. Stattdessen müssen gewisse Eigenschaften erfüllt werden, um als No-SQL-System ausgezeichnet zu werden. Als Beispiel sollte das Datenbanksystem neben dem nicht-relationalen Datenmodell als Open Source frei verfügbar sein [Edlich u. a. 2011].

Allgemein lassen sich NoSQL-Systeme, gemäß dem verwendeten Datenmodell, in verschiedene Kategorien einteilen. Graphdatenbanken, dokumentbasierte Datenbanken und Key-Value-Datenbanken sind nur einige dieser Kategorien und bringen unterschiedliche Vor- und Nachteile mit sich, die bei verschiedenen Anwendungsfällen unterschiedlich stark ins Gewicht fallen. Daher lässt sich die Frage, ob relationale Datenbanken oder NoSQL-Systeme besser geeignet sind, auch nicht einheitlich beantworten. Ein wichtiger Faktor für die Wahl des verwendeten Datenbanksystems sind die vorliegenden Daten. Oftmals ist es nicht zu empfehlen Informationen eines Datenmodells (z.B. tabellenbasiert) in ein anderes, unnatürlicheres Modell (z.B. graphenbasiert) zu überführen, da es bei großen Datenmengen zu Performanceproblemen kommen kann. Außerdem kann das korrekte Formulieren von präzisen Datenbankanfragen wesentlich schwieriger ausfallen, zumal gewisse Operationen, wie zum Beispiel die Berechnung des kürzesten Weges zwischen zwei Knoten, nicht unterstützt werden. Bei SDG ist es somit sinnvoll ein System, wie Neo4j zu wählen, welches Knoten und Kanten auf möglichst natürliche Weise darstellen kann.

2.4.1. Die Graphdatenbank Neo4j

Neo4j[1] ist ein NoSQL-System und gliedert sich in die Reihe der Graphdatenbank ein. Daten liegen somit nicht in Form von Tabellen vor, sondern werden durch Knoten und Kanten beschrieben, wobei jedes abgelegte Objekt über eine eindeutige ID verfügt[2]. Knoten stehen in diesem Zusammenhang für die konkreten Entitäten wie Methoden oder Klassen eines SDG. Kanten beschreiben die Beziehungen zwischen den einzelnen Knoten, wodurch beispielsweise Abhängigkeiten oder Zusammenhänge ausgedrückt werden. Des Weiteren verfügt Neo4j über Konzepte wie Eigenschaften und Labels. Eine Eigenschaft besteht dabei immer aus einem Schlüssel-Wert-Paar und kann an einen Knoten oder eine Kante angehängt werden. So können die Elemente innerhalb eines Graphen genauer beschrieben und von anderen Objekten der gleichen Art abgegrenzt werden. Labels hingegen sind für das Gruppieren von mehreren Knoten vorgesehen. Diese Art der Gruppierung erleichtert sowohl das Schreiben als auch das Auswerten von Anfragen an eine bestehende Datenbank. Des Weiteren lassen sich auf Labels basierende Indexe erstellen, wodurch Knoten mit einem ausgewählten Attribute effizienter ermittelt werden. Der Nachteil dieser Performancesteigerung ist das redundante Speichern von Informationen in der Datenbank.

Seit 2007 ist Neo4j als Open-Source-Projekt erhältlich und wird von der Firma Neo-

[1]https://neo4j.com/ Letzter Zugriff: 29. Oktober 2016
[2]http://neo4j.com/docs/developer-manual/current/ Letzter Zugriff: 29. Oktober 2016

Technology weiterentwickelt. Neben der frei verfügbaren Variante wird außerdem eine kommerzielle Version mit exklusivem Support angeboten [Edlich u. a. 2011]. Neo4j ist eine hauptsächlich in Java geschriebene Datenbank und wurde früher ausschließlich als eingebettete Bibliothek innerhalb einer JVM (Java Virtual Machine) benutzt. Heutzutage lässt sich Neo4j ebenfalls als eigenständiger Server konfigurieren und kann durch verschiedenen APIs, beispielsweise Java, jRuby, Python und C#, oder die HTTP/REST-Schnittstelle in Verbindung mit dem Datenformat JSON genutzt werden. Des Weiteren stellt Neo4j ein Transaktionsmanagement zur Verfügung, wodurch Änderungsoperationen in Transaktionen gekapselt werden und den ACID-Eigenschaften unterliegen. Anfragen an die Datenbank können entweder über die bereitgestellten Schnittstellen oder die deklarative Abfragesprache Cypher, welche in Kapitel 2.4.2 beschrieben wird, gestellt werden.

Seit der Version 2.0 implementiert Neo4j einen Neo4j-Browser, der über die URL `http://localhost:7474/browser/` von einem regulären Browser aus erreicht werden kann [Hunger 2014]. Hierfür muss die Datenbank als externer Server konfiguriert worden sein und darf nicht innerhalb einer JVM gestartet werden. Der Neo4j-Browser ist eine JavaScript-Anwendung und erleichtert dem Benutzer die interaktive Arbeit mit der bestehenden Datenbank. Beispielsweise wird ein Editor mit Syntaxhervorhebung für Cypher-Anfragen bereitgestellt und es besteht die Möglichkeit, häufig gestellte Anfragen zu speichern. Außerdem werden Antworten seitens des Servers dem Benutzer graphisch aufbereitet und Knoten/Kanten durch farbige Kreise bzw. Pfeile repräsentiert. Serverkonfigurationen oder Metadaten über die Datenbank sind ebenfalls über den Neo4j-Browser zu erreichen.

Die in Ziel 1.3.1 beschriebenen Konzepte der Hierarchie sind in dem zugrundeliegenden Datenmodell von Neo4j hingegen nicht vorhanden, wodurch dargestellte Graphen stets flach aufgebaut sind. Zusätzlich werden durch die für Knoten und Kanten verwendeten Layoutalgorithmen teils überlappende Graphen erstellt, welche es dem Nutzer erschweren ein Verständnis für die Visualisierung zu entwickeln. Der aus dieser Arbeit resultierende Editor soll diese Probleme der Visualisierung mittels einer optionalen hierarchischen Darstellung und der Verwendung von Layoutalgorithmen aus dem KLighD-Framework beheben.

2.4.2. Die Anfragesprache Cypher

Statt der weit verbreiteten *Structured Query Language* verfügt Neo4j seit Version 1.4 über eine eigene deklarative Anfragesprache. Mit Hilfe von Cypher werden neben einfachen Anfragen, wie die Ausgabe bestimmter Knoten oder Kanten, auch komplexe Operationen zur Bestimmung eines kürzesten Pfades gestellt bzw. durchgeführt. Die Struktur von Cypher orientiert sich an SQL hinsichtlich der unterschiedlichen Bedingungsklauseln, die je nach Art der Anfrage verwendet oder ignoriert werden können. Bei der Auswertung von Anfragen findet ein Pattern-Matching statt, sodass der Graph nur entlang jener Knoten und Kanten traversiert wird, die ein bestimmtes angegebenes Muster erfüllen. Um Cypher möglichst zugänglich und intuitiv zu gestalten, haben sich die Entwickler bei der Syntax für Kreise und Pfeile entschieden. So werden Knoten innerhalb einer Anfrage stets mit

runden Klammern () und Kanten mit dem Pfeilsymbol –> dargestellt. Ein simples Pattern mit der Struktur () –> () ist in Listing 2.3 aufgeführt.

Listing 2.3. Anfrage in Cypher: Wer sind Tims Freunde?

```
1    MATCH (Tim {Name: 'Tim'})-[:kennt]->(Freund)
2    RETURN Tim, Freund
```

Im Allgemeinen beginnt eine selektierende Anfrage mit einer Menge an Startknoten von denen aus durch den Graph traversiert wird (start-Klausel). Sind keine expliziten Startknoten genannt, wie in Listing 2.3, wird stattdessen jeder Knoten im Graph als potentieller Startknoten betrachtet. Anschließend werden zu erfüllende Muster in der match-Klausel angegeben, wodurch zutreffende Knoten und Kanten selektiert werden. In diesem Beispiel muss der Starknoten jeweils das Attribut *name* mit dem Wert *Tim* aufweisen und die ausgehende Kante vom Typ *kennt* sein. Falls die Menge der Ergebnisse noch weiter gefiltert werden soll, kann dies mit Hilfe der where-Klausel geschehen. Die return-Klausel definiert abschließend eine Projektion auf die gewünschten Rückgabewerte der Anfrage. Je nachdem, ob die Ausgabe noch sortiert oder eine maximale Anzahl an Ergebnissen haben soll, kann dies mit entsprechenden order by- bzw. limit-Klauseln realisiert werden. Eine Anwendung dieser zusätzlichen Klauseln ist in Listing 2.4 zu sehen.

Listing 2.4. Selektierende Anfrage mit den wichtigsten Klauseln

```
1    MATCH (Person)-[:kennt]->(Freund)
2    WHERE Person.Name = 'Tim' AND Freund.Alter > 18
3    RETURN Freund.Name, Freund.Alter, Freund.Wohnort
4    ORDER BY Freund.Name
5    LIMIT 3
```

Neben den selektierenden Anfragen gibt es in Cypher auch die Möglichkeit Knoten und Kanten mittels der create- oder delete-Klausel zu inserieren bzw. zu löschen. Attribute von Objekten können ebenfalls mit set- und remove-Klauseln verändert werden.

In dieser Arbeit wird Cypher für den in Kapitel 1.3.2 beschriebenen *Neo4j-Reader* zum Auslesen der im SGD vorhandenen Knoten und Kanten benötigt. Für das Exportieren eines SDG in eine neu erstellte Datenbank, durch den *Neo4j-Writer* aus Kapitel 1.3.2, wird ebenfalls auf die deklarative Anfragesprache zurückgegriffen. Mittels der erwähnten create-Klausel können die vorhandenen Graphelemente in die Datenbank kopiert werden.

2.5. Die Programmiersprache Xtend

Xtend[3] ist eine statisch getypte, objektorientierte Programmiersprache, welche auf Java aufbaut. Sie umfasst sowohl imperative als auch funktionale Konzepte und verfügt neben

[3]https://eclipse.org/xtend/index.html Letzter Zugriff: 29. Oktober 2016

einer kompakteren Syntax über zahlreiche Erweiterungen, wie Typinferenz oder Operatorüberladung. Mittels des Xtend-to-Java Compilers[4] wird Java-Quellcode entsprechend des im Projekt festgelegten Java-Compilers generiert, wodurch es zu keinen Kompatibilitätsproblemen zwischen den Sprachen kommt. Lambda Ausdrücke in Xtend werden somit entweder in anonyme Klassen oder, falls ein Java 8 Compiler verwendet wird, in Java-Lambdas transformiert. Dies ermöglicht einen flexiblen Einsatz des Java-Dialekts und zwingt den Nutzer nicht sich für eine der Sprachen entscheiden zu müssen. Bei dem resultierenden Grapheditor wird Xtend in dem *Neo4j-Reader*, *-Writer* und der im folgenden Kapitel 2.6.2 beschriebenen Synthese verwendet. Über eine spezielle Methodendeklaration mit dem Schlüsselwort *create* wird ein Konzept zum Erkennen von bereits erstellten Objektinstanzen, wie etwa Knoten oder Kanten eines zu kopierenden Graphen, unterstützt. Die entsprechende Methode wird nur ausgeführt, falls zuvor kein Aufruf mit den gleichen Argumenten stattgefunden hat. Der Code in Listing 2.5 repräsentiert ein Beispiel für das Erstellen von Knoten.

Listing 2.5. Anwendungsbeispiel einer create-Methode von Xtend

```
1   def create result : NodeFactory.getInstance().createNode() copy(Node n) {
2       result.name = n.name
3       result.id = n.id
4   }
```

Sofern bisher keine Kopie des Knotens *n* über die *NodeFactory* mit anschließender Zuweisung der Attribute erzeugt worden ist, wird die Methode *copy* gewöhnlich ausgeführt. Sollte allerdings bereits eine Kopie erzeugt worden sein, ist der Rückgabewert der bereits erstellte und im Cache unter dem Eintrag *n* gespeicherte Knoten.

2.6. Das KIELER Projekt

KIELER[5] (Kiel Integrated Environment for Layout Eclipse RichClient) ist ein an der Christian-Albrechts-Universität zu Kiel entstandenes Forschungsprojekt zur grafischen Visualisierung von komplexen modellbasierten Systemen. Das Open-Source-Projekt ist in die drei Bereiche Layout, Pragmatics und Semantics unterteilt, wobei im Rahmen dieser Arbeit nur die ersteren beiden von Relevanz sind. Die zentrale Datenstruktur von KIELER ist ein EMF-Modell names *KGraph* [Schneider u. a. 2012]. Es bietet neben den Graphelementen wie *KNode* und *KEdge* zum Abbilden der Struktur eines Graphen auch Klassen für layout-spezifische Informationen, welche insbesondere bei den im weiteren Verlauf erwähnte Layoutalgorithmen zum Tragen kommen. Die Struktur der auf dem Graphmodell aufbauenden und im folgenden beschriebenen Komponenten von KIELER ist in Abbildung 2.7 dargestellt.

[4]https://eclipse.org/xtend/documentation/index.html Letzter Zugriff: 29. Oktober 2016
[5]http://www.rtsys.informatik.uni-kiel.de/en/research/kieler/ Letzter Zugriff: 29. Oktober 2016

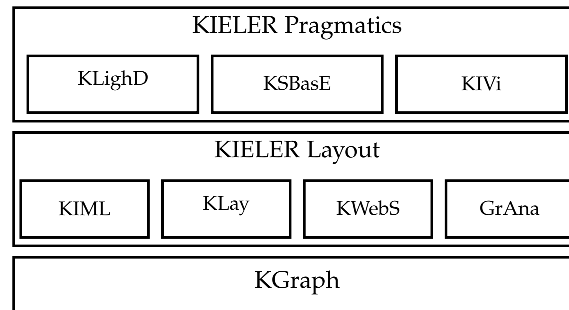

Abbildung 2.7. Zusammenhang der beschriebenen Komponenten von KIELER

KIELER Layout stellt die Grundlage des Projekts dar und umfasst Hintergrundaktivitäten, welche dem Nutzer die Verwendung des Frameworks erleichtern. Beispielsweise wird über die Subkomponente *KIML* (Infrastucture for Meta Layout) eine Verbindung zwischen den Layout-Algorithmen und den Editoren bzw. Views zum Darstellen der angeordneten Graphen hergestellt. Die Menge der Algorithmen setzt sich aus frei zugänglichen Layout-Bibliotheken, wie Graphviz oder OGDF, und eigens entwickelten Implementierungen, welche in *KLay* enthalten sind, zusammen. Über die Komponenten *KWebS* hat der Nutzer die Möglichkeit sämtliche Algorithmen auch als Web Service zu verwenden. Falls Analysen eines Graphen benötigt werden, können diese mittels des *GrAna*-Projekts (Graph Analysis) durchgeführt werden.

Der Bereich KIELER Pragmatics umfasst Werkzeuge, welche den Nutzer im Kontext der modellbasierten Entwicklung unterstützen. Hierzu zählt neben dem Erstellen und Bearbeiten eines Modells über das KSBasE-Projekt (KIELER Structure-based Editing) auch das für diese Arbeit relevante Visualisieren eines Graphen mittels KLighD (KIELER Lightweight Diagrams). Die Interaktion mit dem Modell und der Zugriff auf die Layout-Optionen wird durch die Schnittstellen der Komponente KIVi (KIELER View Management) gewährleistet.

Aufgrund der aktuell noch andauernden Umstrukturierungen innerhalb des KIELER-Projekts gilt es zu erwähnen, dass in dieser Arbeit auf die im Februar 2016 fertiggestellte Version Bezug genommen wird. Die Auslagerung des *KGraph*-Modells in die neue *ELK*-Bibliothek (Eclipse Layout Kernel[6]) ist demnach kein Bestandteil dieser Thesis.

2.6.1. Das (Graph-) Visualisierungsframework KLighD

KLighD ist ein Framework, dessen Fokus auf der temporären Visualisierung eines beliebigen Modells liegt [Schneider u. a. 2013]. Im Gegensatz zu anderen grafischen Editoren steht bei KLighD die in Echtzeit generierte Darstellung im Vordergrund, sodass auf komplexe Optionen zum Bearbeiten des Modells verzichtet wird. Die Grundlage der grafischen

[6]https://www.eclipse.org/elk/ Letzter Zugriff: 29. Oktober 2016

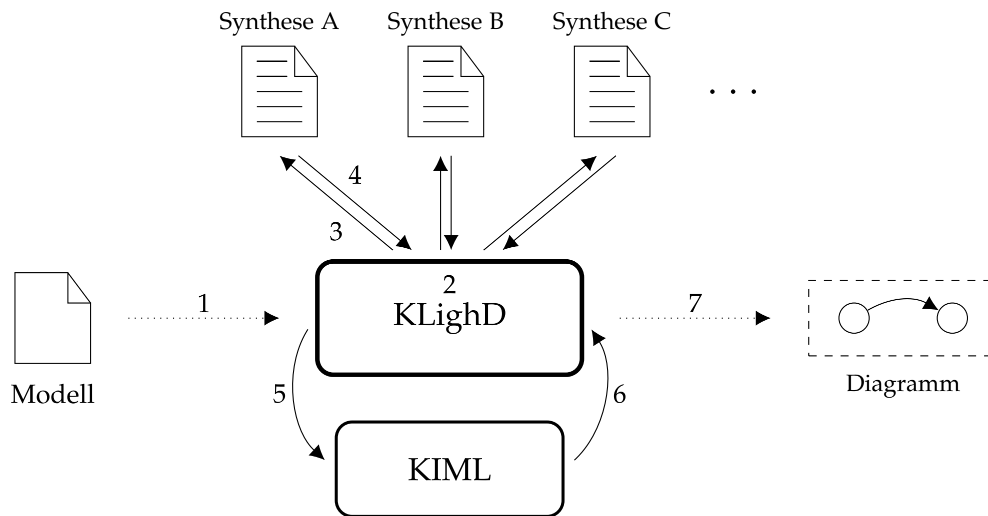

Abbildung 2.8. Der Datenfluss in KLighD [Schneider u. a. 2013]

Repräsentation in eine vom Nutzer geschriebene Synthese zwischen dem darzustellenden Modell und dem *KGraph*-Modell. Falls nicht explizit angegeben, wird das generierte Diagramm nach Gebrauch direkt verworfen und muss für eine wiederholte Betrachtung erneut generiert werden. Der Zusammenhang zwischen den einzelnen Komponenten, die bei einer Visualisierung durch KLighD verwendet werden, ist in Abbildung 2.8 dargestellt.

Im ersten Schritt (1) wird das zu visualisierende Modell in das *KLighD*-Framework eingelesen. Auf Basis dessen wird anschließend eine korrespondierende Synthese für die Transformation in das *KGraph*-Modell ermittelt (2) und auf die Modellinstanz angewendet (3). Der resultierende KGraph und die für das Rendern benötigten Informationen (4) werden durch *KLighD* an das *KIML*-Framework übergeben, in welchem die automatische Anordnung der Graphelemente mittels der verfügbaren Layoutalgorithmen stattfindet (5). Nach dem Empfang der gesamten für das Diagramm benötigten Daten (6), werden diese in ein zum Zeichnen geeignetes Format überführt und in einer View dargestellt (7).

2.6.2. Die Synthese

Die Synthese besteht aus einer Reihe von Transformationsvorschriften zwischen einem beliebigen Modell und dem *KGraph-Modell*, welches die Basis für das Zeichnen und Anordnen von Graphen in *KIELER* darstellt. Die Regeln der Transformation werden eigens durch den Nutzer in dem aus Kapitel 2.5 bekannte Java-Dialekt *Xtend* implementiert, wodurch er keinen Einschränkungen unterlegen ist. Neben der Zuordnung von den Objekten der unterschiedlichen Modellen, werden auch die für eine Visualisierung relevanten Eigenschaften der darzustellenden Elemente festgelegt.

In den Zeilen 1 bis 3 des Listings 2.6 ist zu sehen, dass jeder *State* aus dem ursprünglichen Modell durch einen *KNode* repräsentiert und durch ein Rechteck visualisiert wird. Über den Zugriff auf die Attribute des Objekts *state* können dem neu erstellten Knoten Elemente wie *Labels* zum Anzeigen des Namens (Zeile 6) oder *ChildAreas* für enthaltene Kindknoten hinzugefügt werden. Die an eine Synthese anschließende Kalkulation der Struktur eines Diagramms kann mit Hilfe von sogenannten *Layout-Paramatern* (Zeile 9 - 11) vom Nutzer beeinflusst werden.

Listing 2.6. Auszug einer Synthese in Xtend

```
1   private def KNode transform(State state) {
2      val KNode stateNode = state.createNode().associateWith(state);
3      stateNode.addRoundedRectangle(4, 4, 2);
4
5      stateNode.addInsideCenteredNodeLabel(state.name,
6         KlighdConstants.DEFAULT_FONT_SIZE,
7         KlighdConstants.DEFAULT_FONT_NAME);
8
9      stateNode.addLayoutParam(
10        LayoutOptions.SIZE_CONSTRAINT,
11        EnumSet.of(SizeConstraint.MINIMUM_SIZE));
12     return stateNode;
13  }
```

2.7. Der Grapheditor von L. Blümke und Y. Benekov

Der von L. Blümke und Y. Benekov entwickelte Grapheditor ist eine Menge von mehreren, untereinander abhängigen Plugins, welche die Basis für den aus dieser Arbeit resultierenden Editor bilden. Die einzelnen Plugins und die bestehenden Abhängigkeiten zwischen ihnen werden in Kapitel 2.7.2 erläutert. Das intern verwendete Modell der zu visualisierenden Graphen ist auf einen aus Abschnitt 2.1.1 bekannten Eigenschaftsgraphen zurückzuführen, welches um Metainformationen ergänzt worden ist. Die exakte Struktur des modifizierten Propertygraphen wird im folgenden Kapitel 2.7.1 beschrieben.

In der Abbildung 2.9 ist die Oberfläche des entwickelten Grapheditors zu sehen. Für ein einfacheres Beschreiben der Elemente und ihrer Funktionalitäten wurden dem Screenshot referenzierende Nummern hinzugefügt.

Die Visualisierung der Graphen erfolgt über eine in Abschnitt 2.6.2 erwähnte Synthese in das interne Modell von KLighD, welches anschließend in der *KlighD Diagram View* (1) dargestellt wird. Die Transformation zwischen dem Propertygraph und KGraph wurde mittels Xtend und dem angesprochenen Konzept zum Erkennen von bereits erstellten Objektinstanzen (vgl. Kapitel 2.5) implementiert. Für das Bearbeiten eines Graphen bzw. der beinhalteten Knoten und Kanten enthält das per Rechtsklick aufrufbare Kontextmenü

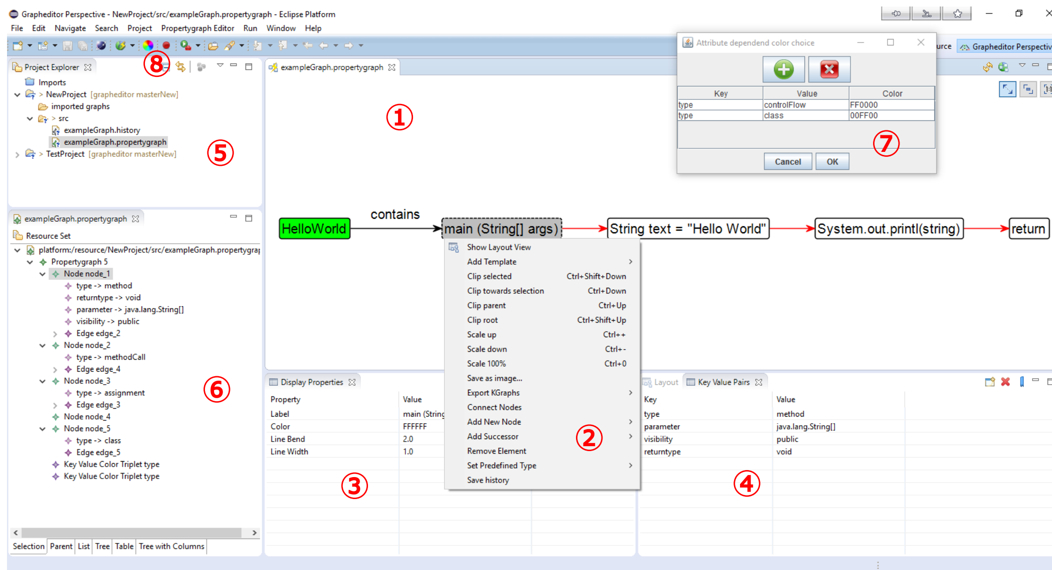

Abbildung 2.9. Benutzeroberfläche des Grapheditors von L.Blümke und Y. Benekov [Blümke 2015]

mehrere Optionen (2). Beispielsweise können neue Graphelemente erstellt, vorhandene Knoten miteinander verbunden oder den Elementen vordefinierte Typen zugewiesen werden.

Um eine stets übersichtliche Darstellung der Graphen zu gewährleisten, werden die Eigenschaften der Elemente nicht direkt an den Knoten bzw. Kanten angezeigt, sondern in mehreren Views (3 + 4) tabellarisch aufbereitet. Über die *Key-Value-View* (4) hat der Nutzer Zugriff auf die mit einem Graphelement verknüpften Schlüssel-Wert-Paare. Vorhandene Paare können so verändert oder gelöscht und neue Paare hinzugefügt werden. Die *Display-Properties-View* (3) ist für das Anzeigen und Verändern der für die Visualisierung relevanten Eigenschaften eines selektierten Elements zuständig. Beispielsweise kann die Breite einer Kante oder die Krümmung der Ecken eines Knotens eingestellt werden, um das jeweilige Objekt von anderen abzugrenzen.

Um mehrere Graphen bzw. Projekte verwalten zu können steht ein aus der Java-Entwicklung bekannter *Package-Explorer* (5) zur Verfügung. Neben einer grafischen Repräsentation in der *KlighD Diagram View* kann die zugrundeliegende Graphinstanz des aktiv ausgewählten Graphen in dem *Tree Editor* (6) betrachtet werden. Mittels einer Baumstruktur werden neben Knoten und Kanten auch einige Metainformationen der Graphelemente angezeigt.

Ein weiteres Feature ist die attributabhängige Einfärbung verschiedener Graphelemente, wodurch dem Nutzer ein weiteres Mittel zur Abgrenzung der für ihn relevanten Elemente zur Verfügung steht. Dies kann insbesondere bei Graphen mit einer hohen Anzahl an Kno-

ten oder Kanten hilfreich sein. Über einen Knopf in der Menüleiste (8) wird ein separates Fenster zur Farbauswahl (7) geöffnet. Die enthaltene Tabelle besteht aus drei Spalten mit den Überschriften: *Key*, *Value* und *Color*. Die ersten beiden Einträge eines *Farbtripels* geben die selektierenden Bedingungen der einzufärbenden Gruppe von Elementen an. Jedes Objekt, welches über das festgelegte *Schlüssel-Wert-Paar* verfügt, wird mit der entsprechenden Farbe zu dem aus der dritten Spalte stammenden HTML-Farbcode eingefärbt.

Des Weiteren wurde ein Rekorder integriert, mit dem Änderungen am Graphen protokolliert und Sicherungen durchführt werden können. Diese Funktion kann über einen roten Knopf (8), welcher ebenfalls der Menüleiste hinzugefügt worden ist, gestartet und beendet werden. Das aufgezeichnete Protokoll wird in einer separaten **.history*-Datei gespeichert.

2.7.1. Das Modell des Propertygraphen

Das vom Editor intern verwendete Modell zur Repräsentation eines Graphen basiert weitestgehend auf dem aus Abschnitt 2.1.1 bekannten Eigenschaftsgraphen. Die Elemente und vorhandenen Abhängigkeiten wurden in einem *Ecore*-Modell beschrieben und durch das Eclipse Modeling Framework (siehe Kapitel 2.3) generiert. In dem Diagramm 2.10 ist das zugrundeliegende Modell dargestellt.

Der zentrale Bestandteil eines *Propertygraphen* sind die enthaltenen Knoten (*Nodes*). Ein Knoten verfügt über das Attribut *lineBend* zum Einstellen der Randkrümmung und wird mittels gerichteten Kanten (Edges) mit Vorgängern bzw. Nachfolgern verbunden. Die Richtung einer Kante wird nicht als Attribut gespeichert, sondern über die Felder *sourceNode* und *targetNode* kodiert. Auffallend ist, dass ein Knoten neben einer beliebigen Anzahl an ein- und ausgehenden Kanten keine anderen Knoten enthalten kann und demzufolge die hierarchischen Konzepte aus Kapitel 1.3.1 bisher nicht unterstützt werden.

Über die Oberklasse *GraphElement* wird einem Knoten bzw. einer Kante weitere allgemeine Attribute vererbt. Der Wert des *label* steht für den im visualisierten Graphen angezeigten Text eines Objektes. In den Attribute *lineWidth* und *color* ist die Linienstärke bzw. Farbe eines Graphelements gespeichert. Das Feld *ID* ermöglicht eine eindeutige Referenzierung sowohl von Knoten als auch von Kanten und wird beispielsweise für die bereits erwähnten *source-* und *targetNode* Attribute benötigt. Um sicherzustellen, dass jede *ID* eindeutig ist, erfolgt die Vergabe der Werte über den *node-* bzw. *edgeCounter* in der Klasse *Propertygraph*. Die Werte dieser Felder entsprechen stets der kleinsten, noch nicht vergebenen *ID*.

Jedem Graphelement kann mit einer unbegrenzte Anzahl an *KeyValuePairs* verknüpft werden, wodurch die zugrundeliegende Struktur des Eigeschaftsgraphen deutlich wird. Obwohl es sich bei den bereits erwähnten Felder, wie dem *label*, streng genommen ebenfalls um Schlüssel-Wert-Paare handelt, werden diese aufgrund ihrer für die Visualisierung notwendigen Informationen direkt in den Objektinstanzen gespeichert.

Um die attributabhängige Einfärbung eines *Propertygraphen* aus Kapitel 2.7 zu unterstützen, kann einem Graphen neben den enthaltenen Knoten eine beliebige Anzahl von *KeyValueColorTriplets* zugewiesen werden. Eine Instanz dieser Klasse verfügt über die

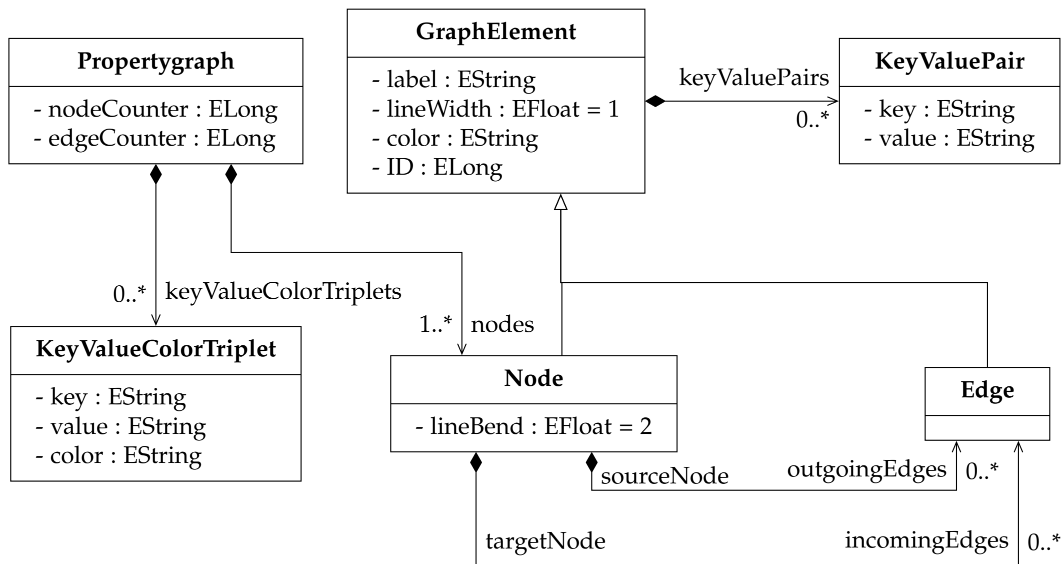

Abbildung 2.10. Ein Diagramm des *Ecore*-Modells mit den vorhandenen Abhängigkeiten [Blümke 2015]

Attribute *key*, *value* und *color* und repräsentiert genau einen Einträge aus der Tabelle des Dialogfensters zur Farbauswahl.

2.7.2. Die Architektur des Grapheditors

Der Grapheditor besteht nicht aus einem einzelnen Plugin, sondern setzt sich aus den Funktionalitäten verschiedener Module zusammen. Wie in der Abbildung 2.11 zu sehen ist, besteht der Kern dieser Menge aus den durch das *Eclipse Modeling Framework* generierten Klassen (siehe Kapitel 2.7.1).

Sie verteilen sich auf die Module *propertygraph*, *propertygraph.edit* und *propertygraph.editor*. Die Klassen und Schnittstellen der in dem *Ecore*-Modell spezifizierten Objekt sind im Plugin *propertygraph* enthalten, welches folglich über keine Abhängigkeiten zu anderen Modulen verfügt. Die Module *propertygraph.edit* und *propertygraph.editor* stellen Funktionen zum Umgang mit den Klassen, wie das Laden und Speichern eines Graphen aus einer bzw. in eine Datei, bereit. Sie können nur in Verbindung mit dem *propertygraph*-Plugin verwendet werden. Der *Tree Editor*, welcher den aktuell ausgewählten Graphen in einer Baumstruktur darstellt und Optionen zum Erstellen und Löschen von Knoten bereitstellt, ist eine Erweiterung des *editor*-Moduls.

Durch das *workbench*-Modul stehen dem Nutzer die im vorherigen Abschnitt 2.7 erläuterten Views zum Anzeigen und Bearbeiten der Schlüssel-Wert-Paare eines Graphelements zur Verfügung. Sobald die Menge an Attributen modifiziert worden ist, wird diese Ände-

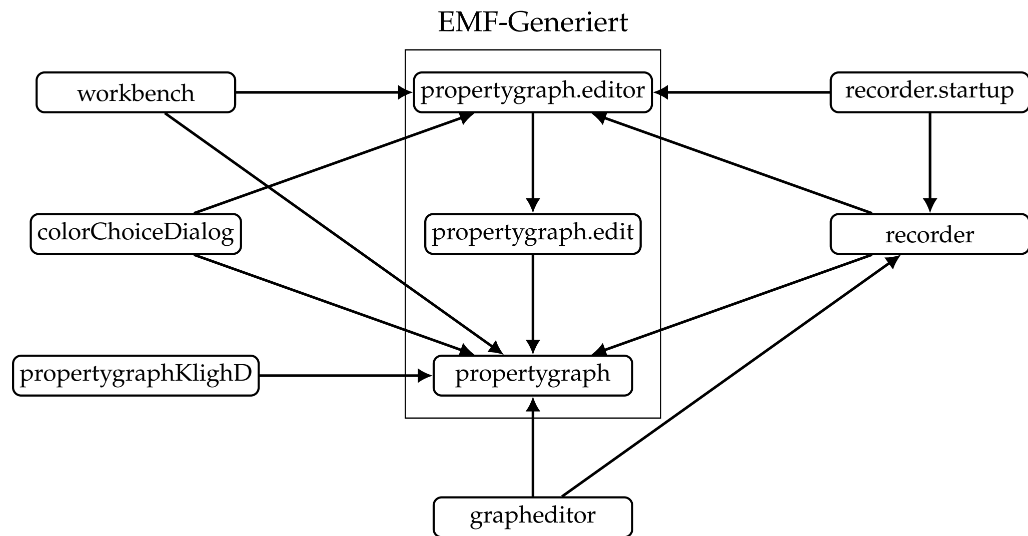

Abbildung 2.11. Abhängigkeiten zwischen den Modulen des Grapheditors von L. Blümke und Y. Benekov [Blümke 2015]

rung in der entsprechenden Klasse vermerkt. Aufgrund des anschließenden Speicherns des Graphen ist das Plugin von den Modulen *propertygraph* und *propertygraph.editor* abhängig.

Das Plugin *colorChoiceDialog* erweitert den Editor um die Funktionalität Graphelemente unterschiedlich einzufärben. Sowohl das Fenster zur Farbauswahl als auch die dahinterstehende Logik wird von diesem Modul bereitgestellt. Für den Zugriff auf die *KeyValueColorTriplet* und das Speichern eines Graphen nach erfolgreichem Einfärben bestehen die gleichen Abhängigkeiten wie beim *workbench*-Modul.

Zum Visualisieren eines Graphen wird das Plugin *propertygraphKLighD* benötigt. Es beinhaltet neben der in Kapitel 2.6.2 erläuterte Synthese ebenfalls die *KLighD Diagram View* zum Anzeigen der Graphen. Durch die Transformation vom *propertygraph*-Modell in das intern von KLighD verwendete *KGraph*-Modell müssen die Klassen des Moduls *propertygraph* zur Verfügung stehen.

Über das *grapheditor*-Plugin hat der Nutzer Zugriff auf das Kontextmenü (siehe Kapitel 2.7) zum Bearbeiten eines Graphen. Für jeden Eintrag des Menüs ist die entsprechend durchzuführende Aktion in einer eigenen Klasse implementiert worden. Diese Klassen wurden in Paketen, welche mit den Untermenüs korrespondieren, zusammengefasst. Für das Protokollieren von durchgeführten Aktionen an einem Graphen, wie beispielsweise das Löschen von Objekten, muss neben dem *propertygraph*-Modul auch die Funktionalität des *recorder*-Plugins gewährleistet sein. Des Weiteren wird eine neue Perspektive zur Verfügung gestellt, in der die Views und Editoren anderer Module in einer festgelegten Anordnung eingebunden sind. Um direkte Abhängigkeiten zu den jeweiligen Plugins zu vermeiden,

werden in der neu erstellten Ansicht lediglich die verfügbaren Komponenten angezeigt.

Wie bereits erwähnt, ist die Funktionalität zum Aufzeichnen zeitlicher Änderungen eines Graphen in dem Plugin *recorder* implementiert worden. Zu den benötigten Klassen für das Erkennen von Modifikationen stellt es außerdem die Funktionalität zum Speichern einer Historie bereit. Sollte beim Starten des Editors bereits eine Aufzeichnung für den aktiven Graphen vorhanden sein, wird diese über das Modul *recorder.startup* in den Rekorder geladen. Durch den Zugriff auf die zu protokollierenden Klassen und das Ermitteln, ob es sich bei der fokussierten View um einen visualisierten Propertygraphen handelt, bestehen die Abhängigkeiten zu den Modulen *propertygraph* und *propertygraph.editor*.

Kapitel 3

Die hierarchische Konzepte des Grapheditors

In diesem Kapitel werden die neu eingeführten Konzepte für die hierarchische Darstellung eines Graphen beschrieben. In Hinblick auf die weiteren Ziele aus Abschnitt 1.3.2 zum Im- und Exportieren von Graphen war eine Modifikation des *Propertygraph*-Modells aus Kapitel 2.1.1 nur eingeschränkt möglich. Der Ansatz einem Knoten neben Kanten ebenfalls Kindknoten zuzuordnen, erwies sich aufgrund der verwendeten Neo4j-Datenbank, als nicht sinnvoll. Das stets nur flache Graphstrukturen unterstützende Datenbanksystem hätte in Verbindung mit dieser Vorgehensweise sowohl beim Import als auch Export zu Komplikationen geführt. Stattdessen wurde ein Konzept von typisierten Kanten entwickelt, welche in der hierarchischen Darstellung teilweise ausgeblendet und über die Elternknoten-Kindknoten-Relation ausgedrückt werden.

Für einen flexiblen Umgang mit den Hierarchien eines Graphen wurden vier verschiedene Kantentypen eingeführt (siehe Kapitel 3.1). Die hierarchische Ansicht eines Graphen ist von der seitens des Nutzers vorgenommenen Typisierung der vorhandenen Kanten abhängig und wird über das Hierarchie-Menü aus Kapitel 3.2 konfiguriert. Im Gegensatz zu der flachen Repräsentation eines Graphen können somit je nach Konfiguration unterschiedliche Darstellungen konstruiert werden.

Um zu gewährleisten, dass beim Betrachten des Graphen auf einem anderen System die vorgenommenen Anpassungen nicht erneut durchgeführt werden müssen, wurde das Modell aus Abschnitt 2.7.1 modifiziert. Der erweiterte Propertygraph wird in Kapitel 3.3 beschrieben. In dem Abschnitt 3.4 wird auf die erweiterte Synthese zum Generieren der hierarchischen Repräsentation eingegangen.

3.1. Die Kantentypen

Dieses Kapitel umfasst die vier neu eingeführten Kantentypen für die hierarchische Darstellung eines Graphen. Der Verwendungszweck und die Notwendigkeit der Typen wird in den jeweiligen Unterkapiteln anhand von Systemabhängigkeitsgraphen (SDG) erläutert. Im Abschnitt 3.4 wird auf die angepasste Synthese eingegangen und das Zusammenspiel der Kantentypen anhand des implementierten Algorithmus beschrieben.

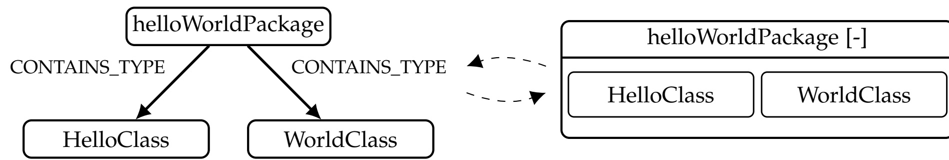

Abbildung 3.1. Vergleich zwischen einer flachen (links) und hierarchischen (rechts) Darstellung eines Graphen. Die Kante *CONTAINS_TYPE* ist als aktive Hierarchie-Kante gesetzt.

3.1.1. Die Hierarchie-Kanten

Für die Implementierung der hierarchischen Konzepte stellen die Hierarchie-Kanten das grundlegende Werkzeug dar. Durch die Typisierung einer Kante als Hierarchie-Kante wird diese in der hierarchischen Ansicht nicht mehr dargestellt und stattdessen über eine hierarchische Beziehung repräsentiert. Der Endknoten wird als Kind in den Startknoten eingekapselt und ist nur zu sehen, falls der Start- bzw. Elternknoten entfaltet ist. In der flachen Darstellung des Graphen hat die Typisierung keine Auswirkungen, sodass die Knoten gemäß dem Modell über eine Kante verbunden sind. Wie in Abbildung 3.1 zu sehen ist, eignet sich dieser Kantentyp zum Repräsentieren von Beziehungen, die eine direkte Hierarchie ausdrücken. Die beiden Klassenknoten aus der linken Ansicht werden in der hierarchischen Repräsentation aufgrund einer Typisierung als Kindknoten dargestellt.

Die Hierarchie-Kanten alleine reichen jedoch nicht aus, um alle möglichen Zusammenhänge innerhalb eines SDG zu beschreiben. Beispielsweise ist es nicht möglich, einen zu einer Methode gehörigen Statement-Knoten als Kind dieser zu visualisieren, falls es keine direkte Kante zwischen den entsprechenden Knoten gibt. Für diese Szenarien wurden die im folgenden Abschnitt 3.1.2 beschriebenen Vererbungs-Hierarchie-Kanten eingeführt.

3.1.2. Die Vererbungs-Hierarchie-Kanten

Vererbungs-Hierarchie-Kanten sind eine Erweiterung zu den bereits eingeführten Hierarchie-Kanten. Sie werden verwendet, falls der Endknoten einer Kante als Kind eines vom Startknoten verschiedenen Knotens dargestellt werden soll. Ein Beispiel für diese Situation ist das Listing 3.1 und der relevante Auszug des zugehörigen SDG (siehe Abbildung 3.2).

Der Methodenknoten *main* ist weder mit den *println*- noch mit dem *return*-Knoten über eine direkte Kante verbunden. Die hierarchische Beziehung dieser Knoten lässt sich demnach nicht mit den eingeführten Hierarchie-Kanten ausdrücken. Stattdessen werden sie entlang der *CONTROL_FLOW*-Kanten vererbt. Der Elternknoten einer vererbten Hierarchie wird durch eine Kombination von verschieden typisierten Kanten definiert. Sobald ein Knoten über zwei ausgehende Hierarchie- und Vererbungs-Hierarchie-Kanten verfügt, stellt dieser den Anfang einer vererbten Hierarchie dar. Sämtliche entlang der Vererbungskanten erreichbaren Knoten werden diesem als Kindknoten hinzugefügt. Innerhalb einer vererbten Hierarchie sind lediglich die Vererbungs-Hierarchie-Kanten zu sehen. Die Hierarchie-Kante

Listing 3.1. HelloWorld-Programm in Java

```
1   package helloWorld;
2
3   public class HelloWorld {
4
5      public static void main(final String[] args) {
6         System.out.println("Hello_World");
7         System.out.println("Hello_World");
8      }
9   }
```

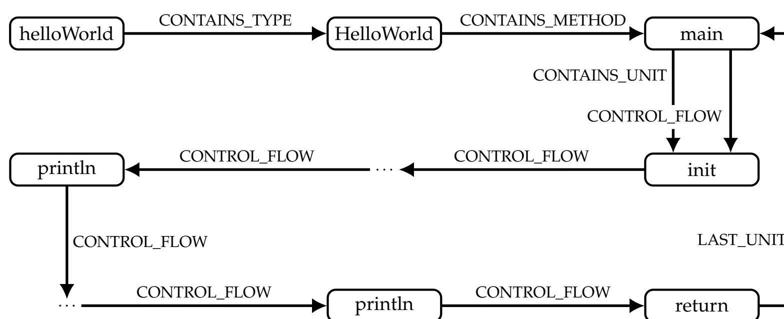

Abbildung 3.2. Der reduzierte flache SDG zu Listing 3.1

der beiden ausgehenden Kanten, welche den Elternknoten einer vererbten Hierarchie markieren, wird folglich nicht visualisiert.

Um eine korrekte hierarchische Repräsentation des SDG aus Abbildung 3.2 zu erhalten, müssen sowohl für die *CONTAINS_UNIT*- als auch die *CONTROL_FLOW*-Kanten entsprechende Einträge in dem Hierarchie-Menü vorgenommen werden. Die hierarchischen Relationen zwischen den Paket- und Klassenknoten werden wie zuvor über Hierarchie-Kanten ausgedrückt. Die resultierende hierarchische Darstellung des SDG und die dazugehörige Typisierung der Kanten ist in Abbildung 3.3 zu sehen.

Bei der Kante *LAST_UNIT* handelt es sich um eine sogenannte Stop-Hierarchie-Kante, welche in Kapitel 3.1.3 genauer erläutert wird und für dieses Beispiel nicht relevant ist. Die Verwendung dieser Kantenart verhindert, dass eine Hierarchie zu weit vererbt wird, wodurch es zu keiner falschen Zuteilung zwischen Eltern- und Kindknoten kommt.

3.1.3. Die Stop-Hierarchie-Kanten

Aufgrund der individuellen Festlegung von Elternknoten können neben Methodenknoten auch Strukturknoten wie Verzweigungs- oder Schleifenknoten als Startknoten einer Hier-

Hierarchie-Kanten:	CONTAINS_TYPE CONTAINS_METHOD CONTAINS_UNIT
Vererbungs-Hierarchie-Kanten:	CONTROL_FLOW
Stop-Hierarchie-Kanten:	LAST_UNIT

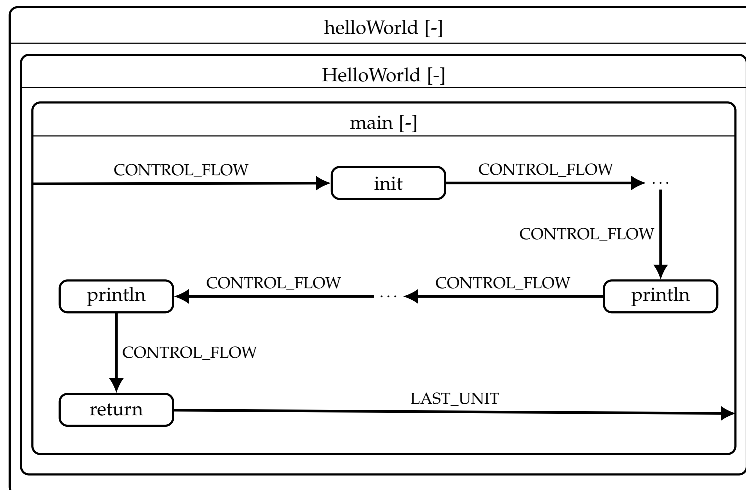

Abbildung 3.3. Die hierarchische Darstellung des SDG zu Listing 3.1 mit vorgenommener Typisierung der Kanten

archie definiert werden. Komplexe Schleifendurchläufe lassen sich somit einkapseln und beinhaltete Statements bleiben bis zum Entfalten verborgen. Die Stop-Hierarchie-Kanten wurden eingeführt, um zu garantieren, dass Statementknoten stets dem korrekten Elternknoten hinzugefügt werden. Andernfalls wäre es nicht möglich das Ende einer vererbten Hierarchie zu definieren, woraus eine fehlerhafte Zuweisung sämtlicher Kindknoten, welche Nachfolger einer abgeschlossenen Struktur sind, resultieren würde. Ein Beispiel hierfür sind die Anweisungen, welche im ursprünglichen Quellcode nach aber nicht innerhalb einer Schleife stehen. Die entsprechenden Statementknoten würden fälschlicherweise als Kindknoten in den Schleifenknoten hineingezogen werden.

Durch das Typisieren einer Kante als Stop-Hierarchie-Kante wird das Ende einer vererbten Hierarchie definiert und der Startknoten dieser Kante als letzter Kindknoten interpretiert. Die nachfolgenden Knoten werden in Abhängigkeit von den typisierten Vererbungs-Hierarchie-Kanten einem anderen Elternknoten, wie zum Beispiel einem übergeordneten Struktur- oder Methodenknoten, hinzugefügt. Die Verwendung der drei bisher eingeführten Kantentypen ist in der hierarchischen Darstellung des zu Listing 3.2 gehörenden SDG (siehe Abbildung 3.4) zu sehen.

Listing 3.2. Add-Programm in Java

```java
public static void add() {
    int sum = 0;
    for(int i = 0; i < 5; i = i + 1){
        sum = sum + i;
    }
    System.out.println(sum);
}
```

Hierarchie-Kanten:	CONTAINS_UNIT
Vererbungs-Hierarchie-Kanten:	CONTROL_FLOW AGGREGATED_CONTROL_FLOW
Stop-Hierarchie-Kanten:	LAST_UNIT

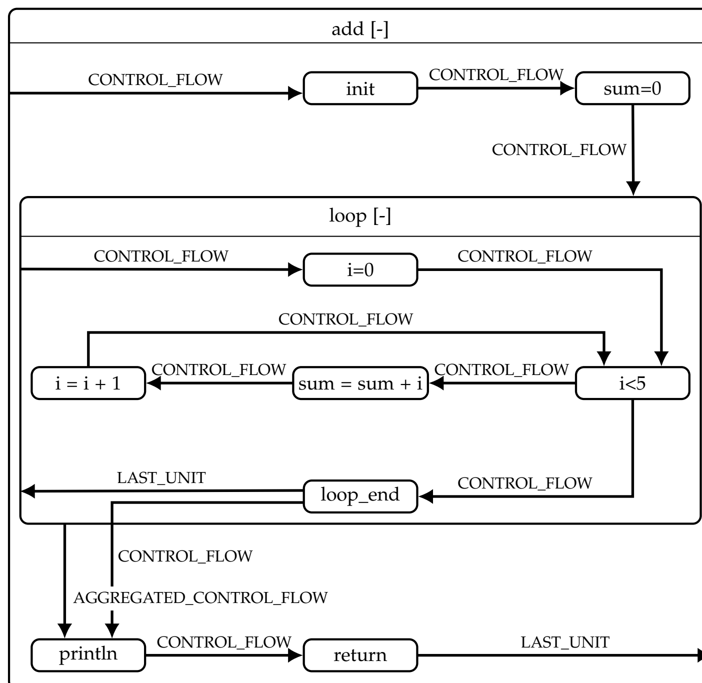

Abbildung 3.4. Hierarchische Darstellung des reduzierten SDG aus Listing 3.2

Sowohl der *init-* als auch der *i=0*-Knoten verfügt neben der *CONTROL_FLOW*-Kante über eine weitere eingehende *CONTAINS_UNIT*-Kante, die wegen der Typisierung nur in der flachen Darstellung des Graphen sichtbar ist. Wie bereits erläutert, definiert diese Kombination von Hierarchie- und Vererbungs-Hierarchie-Kante den Anfang einer vererbten Hierarchie und erweitert die jeweiligen Knoten zu Elternknoten. Durch das Typisieren der *LAST_UNIT*-Kanten als Stop-Hierarchie-Kanten, endet die Hierarchie der Schleife bei dem *loop_end*-Knoten. Die folgenden Knoten *println* und *return* sind dem *add*-Knoten entlang der ebenfalls als Vererbungs-Hierarchie-Kante typisierten *AGGREGATED_CONTROL_-FLOW*-Kante als Kindknoten hinzugefügt. Diese neue Art von Kanten stellen jeweils eine Verknüpfung zwischen dem entfaltbaren Strukturknoten und seinem Nachfolger her.

Obwohl die zwei in den *println*-Knoten eingehenden Kanten für eine korrekte hierarchische Darstellung notwendig sind, verfügen beide Kanten über den gleichen Informationsgehalt. Folglich würde für das Verständnis des Graphen die Visualisierung von nur einer der Kanten ausreichen. Mit dem Konzept der Aggregations-Tupel aus Kapitel 3.1.4 ist es möglich jeweils eine dieser beiden redundant erscheinenden Kanten in Abhängigkeit von dem Zustand des vorherigen *loop*-Knotens ein- bzw. auszublenden. Größere Graphen werden auf diese Weise entschlackt und für den Nutzer übersichtlich gehalten.

3.1.4. Die Aggregations-Tupel

Im Vergleich zu den bisherigen Kantentypen ist die Verwendung der Aggregations-Tupel optional und wird für eine korrekte hierarchische Darstellung nicht benötigt. Die in einem Graphen vorkommenden Kantentypen werden zu Paaren kombiniert, wodurch bei Knoten, welche über eingehende Kanten beiden Typs eines Tupels verfügen, nur eine Kante angezeigt wird. Durch das Ausblenden von überflüssigen Kanten wirkt der Graph entschlackt.

Die Aggregations-Tupel bestehen aus einer aggregierten und einer nicht-aggregierten Kante. Die Entscheidung, welche Kante verborgen wird, hängt von dem Zustand des Startknotens der aggregierten Kante ab. Ist dieser entfaltet und die nicht-aggregierte Kante sichtbar, wird die eingehende aggregierte Kante nicht visualisiert. Im Falle eines eingekapselten Vorgängers, bei dem folglich kein Kindknoten und keine ausgehende nicht-aggregierte Kante eines Kindknoten sichtbar ist, wird die aggregierte Kante dargestellt. Ein Beispiel hierfür ist der zu Listing 3.3 korrespondierende reduzierte SDG aus Abbildung 3.5.

Neben der gleichen Typisierung wie in Abbildung 3.3 wurden die Kanten *CONTROL_-FLOW* und *AGGREGATED_CONTROL_FLOW* zu einem Aggregations-Tupel kombiniert. Die Auswirkungen sind anhand des abschließenden *return*-Knotens zu sehen. Dieser verfügt im Gegensatz zum *println*-Knoten aus Abbildung 3.4 lediglich über eine eingehende Kontrollfluss-Kante. Die vorhandene *AGGREGATED_CONTROL_FLOW*-Kante zwischen dem entfalteten *if*- und *return*-Knoten (1) wird aufgrund der sichtbaren *CONTROL_-FLOW*-Kante ausgeblendet. Sobald eine Einkapselung des äußeren Verzweigungs-Knoten

Listing 3.3. Verschachtelte If-Verzweigung in Java

```
1  public class NestedIf {
2
3     public static void nestedIf() {
4        int sum = rand(37,47);
5        if(sum == 42){
6           System.out.println("Winner!");
7        } else {
8           if(sum < 42){
9              System.out.println("Too␣small!");
10          } else {
11             System.out.println("Too␣big!");
12          }
13       }
14    }
15 }
```

erfolgt, wird wie bei dem zu sehenden eingekapselten *if*-Knoten (2) die *AGGREGATED_-CONTROL_FLOW*-Kante dargestellt.

Durch eine Kombination der erläuterten Kantentypisierung und optionalen Bildung von Aggregations-Tupeln lassen sich beliebige flach aufgebaute Graphen bzw. SDG in einer hierarchischen Darstellung visualisieren. Die Konfiguration für eine hierarchische Ansicht wird mittels der im folgenden beschriebenen erweiterten Synthese (siehe Kapitel 3.4) berücksichtigt. Grundlage dieser bilden die Transformationsvorschriften aus den Arbeiten von L. Blümke [Blümke 2015] und Y. Benekov [Benekov 2015].

3.2. Das Hierarchie-Menü

Das Hierarchie-Menü ist die zentrale Komponente zum Verwalten der Einstellungen für die hierarchische Darstellung eines Graphen. Über einen Knopf in der Menüleiste wird das in Abbildung 3.6 zu sehende Fenster aufgerufen.

Das Feld *Activate Hierarchy* (1) ermöglicht dem Nutzer den in Ziel 1.3.1 erläuterten Wechsel zwischen der flachen und hierarchischen Repräsentation eines Graphen, wobei die Letztere von den Tabelleneinträgen abhängig ist. Jede Tabelle besteht aus zwei Spalten und beinhaltet die Kanten des Graphen, welche beim Synthetisieren als Hierarchie- (2), Vererbungs-Hierarchie- (3) oder Stop-Hierarchie-Kante (4) anerkannt werden soll. Mittels der Spalte *Active* (5) werden einzelne Einträge ignoriert, ohne sie dabei löschen zu müssen. Als Beispiel wurden in Abbildung 3.6 die Kanten *contains_method* und *contains_package* als Hierarchie-Kanten typisiert, aber nur eine als aktiv gesetzt.

Über die *Add-* oder *Remove-Buttons* (6) der Tabellen können neue Einträge angelegt bzw.

	nicht-aggregierte Kante	aggregierte Kante
Aggregations-Tupel:	CONTROL_FLOW	AGGREGATED_ CONTROL_FLOW

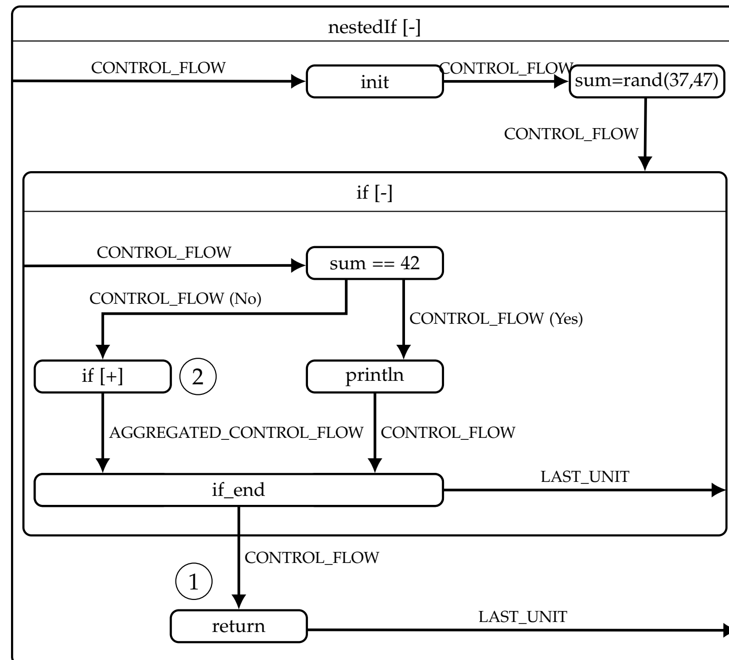

Abbildung 3.5. Beispielhafte Anwendung der Aggregations-Tupel

vorhandene gelöscht werden. Das Ändern eines Kantentyps ist mittels Doppelklick auf den Namen der Kante möglich. Falls der Nutzer den Graphen fortlaufend nach dem Entfalten bzw. Einkapseln eines Knotens speichern möchte, kann er diese Funktionalität über das Feld *Save graph on expand/collapse action* (7) aktivieren. Die Festlegung, welches Attribut einer Kante für die Typisierung verwendet wird, geschieht über die Eingabe in das Textfeld (8).

Anders als bei den drei Kantentypen, welche über das Plugin *hierarchyDialog* verwaltet werden, ist das Erstellen von Aggregations-Tupel (siehe Abschnitt 3.1.4) für eine hierarchische Darstellung nicht notwendig. Aus diesem Grund wurde das optionale Feature zum Ausblenden von überflüssigen Kanten in einem eigenständigen Modul namens *edgeDialog* implementiert. Mittels einer zu Abbildung 3.6 ähnlichen Benutzeroberfläche wird die Funktionalität aktiviert und der Zugriff auf die *Aggregations-Tupel* gewährleistet.

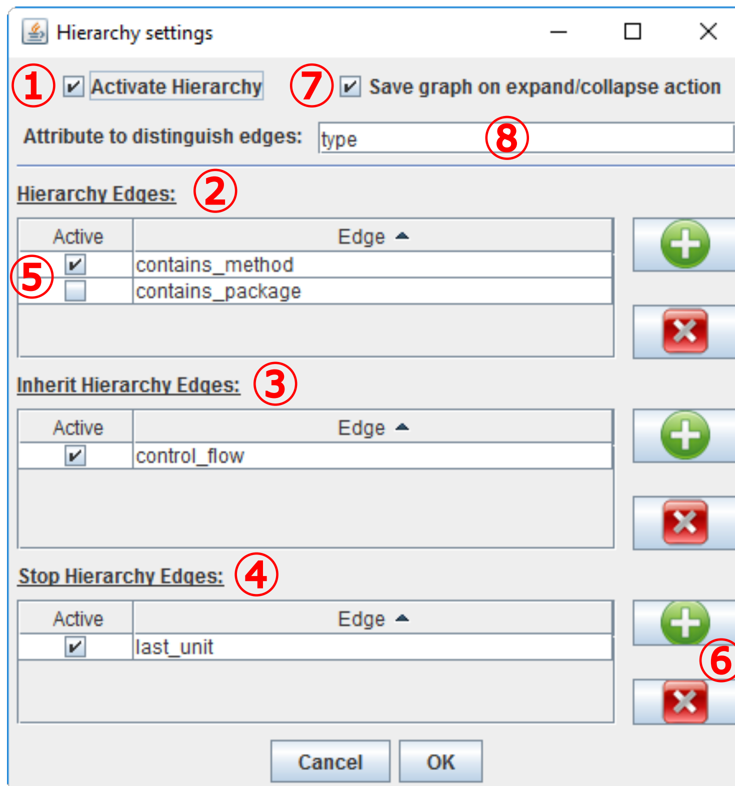

Abbildung 3.6. Screenshot des Menüs zur Einstellung der Kantentypen

3.3. Der erweiterte Propertygraph

Das ursprüngliche Modell des Propertygraphen wurde um Klassen und Felder erweitert, sodass sämtliche Informationen, die für eine hierarchische Darstellung nötig sind, persistent gehalten werden. Jeder Graph, der vom Nutzer für eine hierarchische Repräsentation vorbereitet worden ist, kann demnach auf einem beliebigen System betrachtet werden, ohne die Modifikationen erneut durchführen zu müssen. Dies kann beispielsweise bei der Analyse eines tief verschachtelten Knotens hilfreich sein, falls alle übergeordneten Knoten bereits entfaltet worden sind. In der Abbildung 3.7 zu dem erweiterten Modell des Propertygraphen sind die neu hinzugefügten Klassen farblich (gelb) von den bereits vorhandenen abgegrenzt.

Die Klasse *EdgeInformation* korrespondiert mit den Tabelleneinträgen aus dem zuvor erläuterten Hierarchie-Menü. Jeder Eintrag wird als eigene Instanz abgelegt und in einer der drei Listen *hierarchyEdges*, *inheritHierarchyEdges* oder *stopHierarchyEdges* gespeichert. Die Felder *active* und *edge* werden mit den entsprechenden Werten der Spalten einer Tabelle

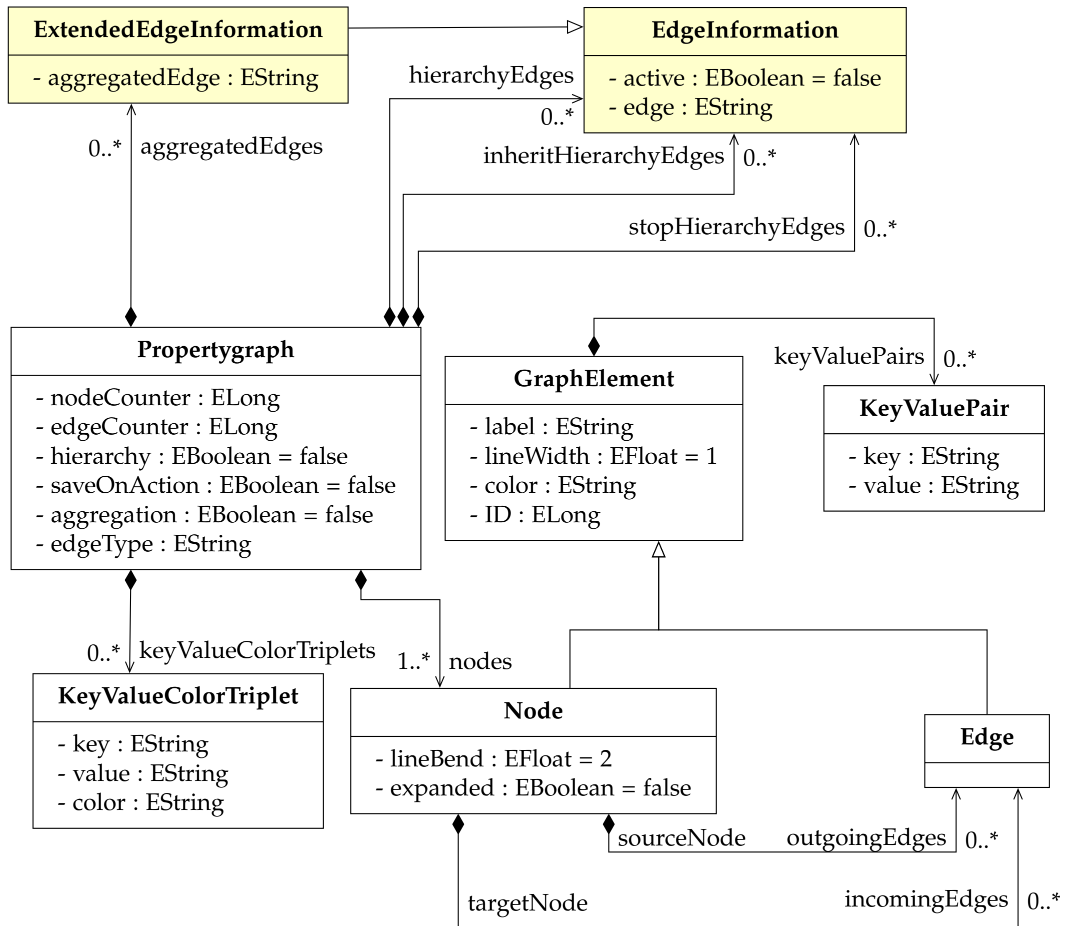

Abbildung 3.7. Ein Diagramm des erweiterten Modells des Propertygraphen, wobei ergänzte Klassen gelb eingefärbt worden sind.

belegt.

Für weitere Einstellungsmöglichkeiten wurde die Klasse *Propertygraph* um neue Felder erweitert. Das (De-)Aktivieren der hierarchischen Darstellung wird in dem Attribut *hierarchy* vermerkt, sodass der Graph bei einem erneuten Öffnen direkt in der zuletzt verwendeten Ansicht visualisiert wird. Der Wert des Feldes *edgeType* bestimmt auf Basis welchen Attributs die Typisierung der Kanten durchgeführt wird.

Die optionale Funktionalität zum Aggregieren von Kanten (siehe Kapitel 3.1.4) wird über das eigenständige Aggregations-Menü ein- bzw. ausgeschaltet und der Zustand in dem Attribut *aggregation* gespeichert. Ähnlich wie bei der Oberklasse *EdgeInformation* repräsentieren Instanzen von *ExtendedEdgeInformation* die Tabelleneinträge des Aggregations-Menüs. Die beinhalteten Felder *edge* und *aggregatedEdge* stellen jeweils einen Zusammenhang zwischen den als redundant markierten Kanten dar. Die Einträge werden dem Propertygraphen in einer beliebig langen Liste mit dem Namen *aggregatedEdges* angehängt.

Um den Zustand eines Knoten (entfaltet oder eingekapselt) zu speichern, wurde der Klasse *Node* das Feld *expanded* hinzugefügt. Dieses ist im initialen Zustand auf *false* gesetzt, sodass beim ersten Öffnen eines Graphen nur die obersten Knoten sichtbar sind.

3.4. Die erweiterte Synthese

In Anlehnung an den ursprünglichen Grapheditor wird die erweiterte Synthese zwischen dem zu visualisierenden Modell und dem *KGraph*-Modell ebenfalls in dem Java-Dialekt *Xtend* implementiert. Um ein grobes Verständnis für die einzelnen relevanten Transformationsschritte zu entwickeln, ist in Listing 3.4 eine auf die wesentliche Teile reduzierte Version der Synthese angegeben.

Eingabe und Ausgabe der Transformation sind Instanzen beider erwähnter Modelle, wobei als Rückgabewert stets der Wurzelknoten des resultierenden *KGraphs* verwendet wird. Falls die hierarchische Ansicht aktiviert worden ist, werden im ersten Schritt (Zeile 4) sämtliche Informationen aus den dafür vorgesehenen Tabellen ausgelesen und stehen im weiteren Verlauf der Synthese zur Verfügung. Anschließend wird für jeden Knoten des zu visualisierenden *Propertygraphen* eine neue *KNode*-Instanz erstellt und mit dem entsprechenden Knoten assoziiert (Zeile 6). Die Form, Farbe und Beschriftung eines Knoten wird hingegen erst in Zeile 27 hinsichtlich der im ursprünglichen Knoten definierten Werte festgelegt. Andernfalls würden die zwei implementierten Ansichten für einen Elternknoten (eingekapselt und entfaltet) wegen der noch nicht vorhandenen Kindknoten ignoriert werden.

Nach dem Erstellen des *KNodes* wird bei jedem Knoten wiederum über die Menge der ausgehenden Kanten iteriert und eine Fallunterscheidung basierend auf der vom Nutzer vorgenommenen Kantentypisierung durchgeführt. In Abhängigkeit, ob es sich bei der aktuellen Kante (*edge*) um eine Hierarchie-Kante handelt (Zeile 8), wird der Endknoten jener Kante dem derzeitigen Knoten *node* als Kindknoten oder Nachfolger hinzugefügt (Zeile 9-15). Im letzteren Fall wird neben der Erstellung und Assoziierung einer neuen

Listing 3.4. Erweiterte Synthese in Pseudocode

```
1  input  : propertygraph
2  output : kGraph
3  begin  :
4     collect information about hierarchy configuration
5     for each (propertygraph.node) do
6        kNode = create kNode and bound it to node
7        for each (node.outgoingEdge) do
8           if (edge == active HierarchyEdge && hierarchy is activated) do
9              targetKNode = recursively create kNode
10                         and bound it to edge.targetNode
11             add targetKNode to kNode.childNodes
12          else
13             kEdge = create kEdge and bound it to edge
14             targetKNode = recursively create kNode
15                         and bound it to edge.targetNode
16             kEdge.sourceNode = kNode
17             kEdge.targetNode = targetKNode
18             create a kEdge shape
19          endIf
20       endFor
21       if (node == starting node of inherited hierarchy &&
22           hierarchy is activated) do
23          nodes = get all reachable nodes through inherit hierarchy edges
24          kNodes = get all kNodes bound to nodes in the list
25          add kNodes to kNode.childNodes
26       endIf
27       create a knode shape
28       add kNode to kGraph.nodes
29    endFor
30    if (hierarchy is activated)
31       remove unnecessary aggregated edges
32    endIf
33 end
```

KNode-Instanz (*targetKNode*) ebenfalls die zur Visualisierung der Kante (edge) benötigte *KEdge*-Instanz (*kEdge*) erzeugt. Durch das Setzen des Start- und Endknotens in den Zeilen 16 bzw. 17 wird die kopierte Kante außerdem zu den Listen an aus- und eingehenden Kanten der entsprechenden *KNode*-Instanzen hinzugefügt. Die Form einer dargestellten *kEdge* wird in Zeile 18 auf Basis der ursprünglichen Kantenattribute definiert.

Das Erstellen der *targetKnodes* in Zeile 9,10 und 14,15 erfolgt über einen rekursiven Methodenaufruf der Statements zwischen Zeile 6 und 28, wobei der zu transformierende Knoten jeweils als Parameter übergeben wird. Der Rückgabewert dieser Methode ist stets die entsprechende *KNode*-Instanz.

Über das in Kapitel 2.5 vorgestellte Konzept zum Erkennen von bereits erstellten Objektinstanzen wird garantiert, dass für jeden Knoten des Propertygraphen stets nur ein korrespondierender *KNode* erstellt wird. Falls beispielsweise der Knoten *targetKNode* aus Zeile 9 bereits während der Iteration über die Knotenmenge erzeugt worden ist, wird beim Zuweisen des Kind- oder Endknotens eine Referenz auf die bereits vorhandene *KNode*-Instanz verwendet.

Im Anschluss an die Kanteniteration wird unterschieden, ob der aktive Knoten (*node*) ein Startknoten einer vererbten Hierarchie ist (Zeile 21). Jeder über ausgehende Kanten nachfolgende Knoten wird auf eine eingehende Hierarchie- und Vererbungs-Hierarchie-Kante untersucht. Falls die Kanten beiden Typs vorhanden sind, wird dieser mitsamt den weiteren entlang Vererbungs-Hierarchie-Kanten erreichbaren Knoten in einer Liste gespeichert (Zeile 23). Aufgrund der bereits durchgeführten Iteration über alle von dem aktuellen Knoten ausgehenden Kanten und den entsprechenden Endknoten, existiert für jeden ursprünglichen Knoten genau eine korrespondierende *KNode*-Instanz. Im nächsten Schritt wird für jeden Knoten der zuvor erstellten Liste die jeweilige Instanz ermittelt und dem aktuellen Knoten als Kindknoten hinzugefügt (Zeile 24-25). In Zeile 28 wird dieser schließlich dem resultierenden *KGraph* hinzugefügt.

Nachdem sämtliche Knoten und Kanten in dem *KGraph* abgebildet worden sind, werden die gemäß den Aggregations-Tupeln redundanten Kanten ausgeblendet (Zeile 30-32). Über einen Algorithmus werden die Zustände der Knoten untersucht und analysiert, welche der beiden Kanten eines Tupeleintrags ein- bzw. ausgeblendet wird. Falls der zu einem Knoten über die nicht-aggregierte Kante verknüpfte Vorgängerknoten sichtbar ist, wird die aggregierte Kante ausgeblendet. In diesem Zusammenhang ist ein Knoten sichtbar, sobald alle übergeordneten Elternknoten entfaltet sind.

Um zu verhindern, dass die Synthese bei dem Entfalten und Einkapseln eines visualisierten Knotens erneut komplett durchgeführt wird, wurde eine sogenannte *Expand-CollapseAction* implementiert. Hierbei handelt es sich um eine Erweiterung des Algorithmus aus Zeile 31, welche nur bei einer Interaktion mit dem dargestellten Graph ausgeführt wird. Sobald der Nutzer einen Knoten entfaltet oder einklappt, wird lediglich die Sichtbarkeit der Kanten des relevanten Teilgraphen neu berechnet. Somit bleiben die meisten Abschnitte eines visualisierten Graphen unangetastet und das Neuzeichnen wird performanter durchgeführt.

Kapitel 4

Das Neo4j-Reader Plugin zum Importieren eines Graphen

4.1. Die Architektur des Neo4j-Reader Plugins

Der *Neo4j-Reader* wurde als eigenständiges Plugin entwickelt und stellt eine Erweiterung der bisherigen Funktionalitäten des Grapheditors dar. Durch den *Reader* wird die Funktion zum Importieren eines SDG aus einer Neo4j-Datenbank bereitgestellt, sodass der Datenbankzustand als Graph visualisiert wird (siehe Abschnitt 1.3.2). Die Anbindung der Datenbank erfolgt über das lokale Dateisystem oder die seitens Neo4j bereitgestellte REST-Schnittstelle, wodurch der Import eines Graphen systemübergreifend möglich ist. Die Verwendung der verschiedenen Datenbankanbindungen wird in Abbildung 4.1 schematisch dargestellt.

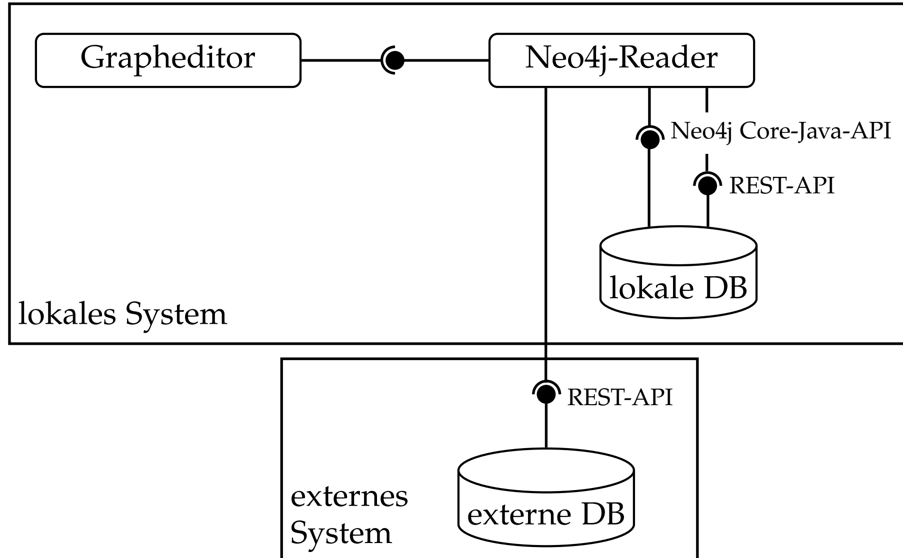

Abbildung 4.1. Zusammenhang der einzelnen Komponenten beim Importieren eines Graphen

Um einen flexiblen Umgang mit dem Grapheditor zu gewährleisten ist sowohl das Visualisieren eines Graphen als auch die Bearbeitung mit einem externen Programm zeitgleich ausführbar. Das Sperren der Datenbank ist lediglich bei Verwendung der *Neo4j Core-Java-API* für einen kurzen Zeitraum notwendig. Dies erlaubt es, den Editor als ergänzendes Werkzeug neben anderen Anwendungen zu verwenden und den gesamten Prozess der Parallelisierung zu erleichtern (siehe Kapitel 1.1).

In dem folgenden Abschnitt 4.2 wird zunächst der allgemeine Vorgang zum Importieren eines Graphen sowie die hierfür bereitgestellten Optionen beschrieben. Anschließend wird in den Unterkapiteln 4.2.1 und 4.2.2 auf die Besonderheiten der jeweiligen Datenbankanbindungen eingegangen.

4.2. Das Importieren eines SDG

Für das Importieren eines Graphen wurde ein unabhängiger Wizard implementiert, mit Hilfe dessen der Nutzer durch den gesamten Vorgang und die auswählbaren Einstellungen geführt wird. Über einen dem Import-Menü hinzugefügten Knopf namens *Import Neo4j Database* wird der in Abbildung 4.2 dargestellte Dialog geöffnet.

Das obersten Textfeld (1) dient unabhängig von der Art der Anbindung für die Auswahl der zu importierenden Datenbank. Je nachdem, ob ein lokaler Dateipfad oder eine URL, beginnend mit dem Prefix *http://*, eingegeben worden ist, wird im weiteren Verlauf die entsprechende Import-Synthese verwendet. Im letzteren Fall ist es von Einstellungen der Datenbank abhängig, ob sich der Nutzer über die hierfür bereitgestellten Felder *Username* und *Password* (2) vor dem Importieren zunächst autorisieren muss. Der Dateiname (3) und Speicherort (4) der neuen Datei sind über die beiden unteren Felder des Dialogs auswählbar. Hierbei ist zu beachten, dass die Dateiendung *.propertygraph* beibehalten wird.

Sobald alle Einstellungen vorgenommen worden sind, ist es möglich die Datenbank zu importieren und in einen Propertygraphen zu überführen. Jedoch ist bisher weder die Typisierung der Kanten noch die Auswahl des hierfür verwendeten Attributs durchgeführt worden, sodass der resultierende Propertygraph lediglich als flacher Graph visualisiert wird. Um die für eine hierarchische Ansicht benötigten Anpassungen bereits beim Importieren zu tätigen, wird über einen Klick auf den *Next*-Knopf die in Abbildung 4.3 dargestellte zweite Seite des Wizards geladen.

Während des Wechsels zwischen den Dialogseiten wird überprüft, ob es sich bei der ausgewählten Datenbank um eine Neo4j-Instanz handelt und eine erfolgreiche Verbindung hergestellt wurde. Sollte dies nicht der Fall sein, wird über einen Hinweis auf die vorhandenen Probleme verwiesen und die erste Dialogseite erneut geladen.

Ist es bei der Überprüfung zu keinen Fehlern gekommen, wird die zu importierende Datenbank auf sämtliche vorhandene Kantentypen untersucht. In Abhängigkeit von der Anbindung werden hierbei die unterschiedlichen Methoden aus Kapitel 4.2.1 und 4.2.2 verwendet. Anschließend wird jeder ermittelte Typ den drei Tabellen (1) der zweiten Dialogseite als eigener Eintrag hinzugefügt, sodass der Nutzer lediglich die für eine

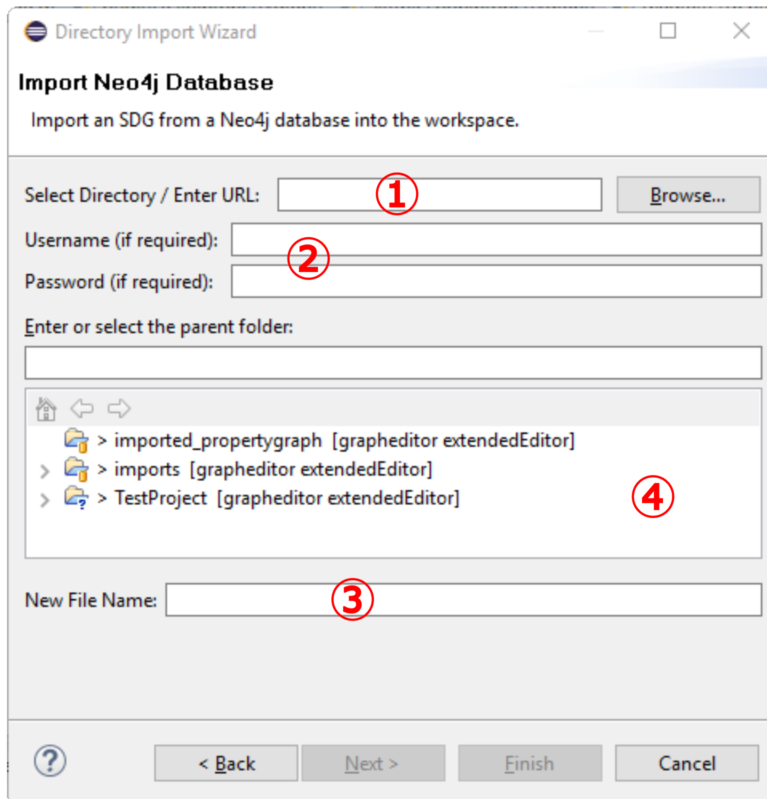

Abbildung 4.2. Dialog zur Auswahl der zu importierenden Datenbank des *Neo4j-Readers*

Typisierung relevanten Tupel aktivieren muss. Des Weiteren ist es möglich einer Tabelle neue Einträge hinzuzufügen (2) oder vorhandene zu löschen (3). Das Ändern der Namen erfolgt über einen Doppelklick auf den entsprechenden Eintrag.

Das Feld *Activate Hierarchy* (4) ist zur Auswahl der initialen Ansicht, ob hierarchisch oder flach, eines Graphen vorgesehen. Die Auswahl des für die Typisierung verwendeten Kantenattributs erfolgt durch das darunter angeordnete Eingabefeld (5). Um eine korrekte hierarchische Ansicht zu garantieren, ist bei der Eingabe auf ein bei allen Kanten vorhandenes Attribut zu achten. Sollte dies nicht der Fall sein, werden die entsprechenden Kanten nicht berücksichtigt und gegebenenfalls durch keine hierarchische Beziehung zwischen den beteiligten Knoten repräsentiert.

Sobald alle Einstellungen getätigt worden sind, wird der Import des Graphen über ein abschließendes Bestätigen des Knopfes *Finish* gestartet. Die Daten der Neo4j-Instanz werden mittels einer der nachfolgend erläuterten Verfahren in eine Instanz des in Kapitel 3.3 eingeführten erweiterten Propertygraphen überführt. Nach dem Abschluss der

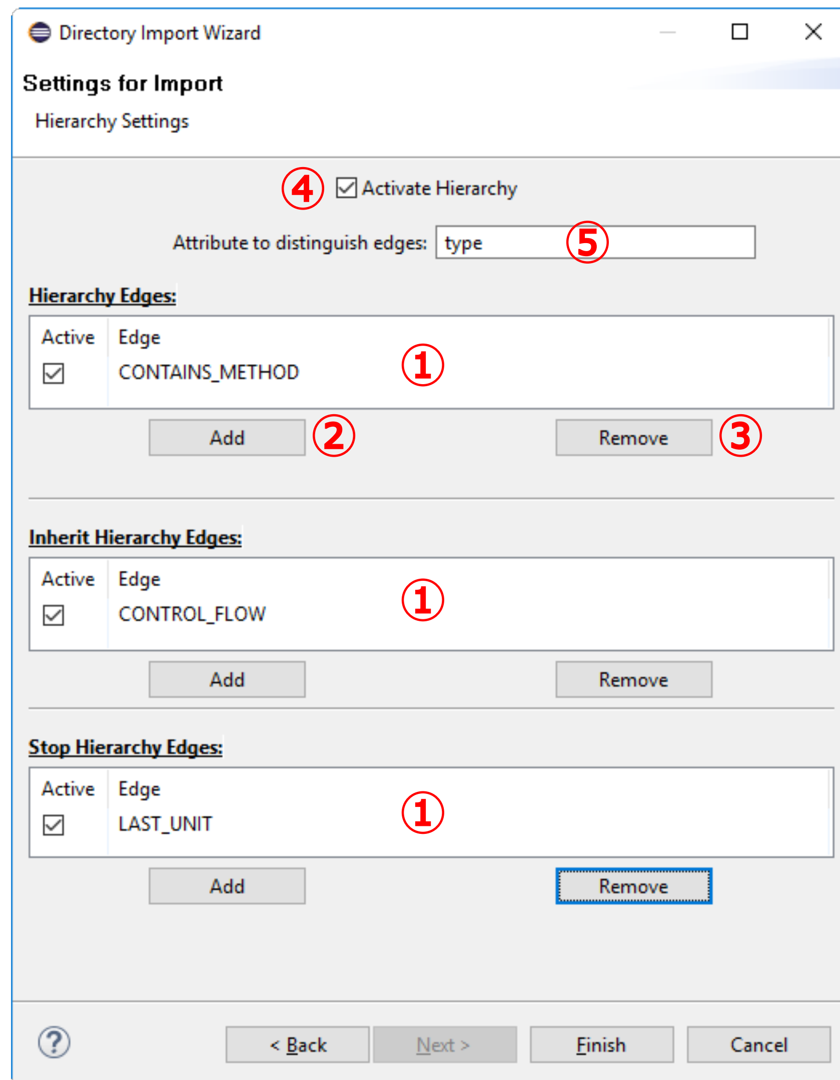

Abbildung 4.3. Dialog zur Typisierung der vorhandenen Kanten einer Neo4j-Datenbank

Import-Synthese wird die Datenbank gegebenenfalls wieder freigegeben und die erweiterte Synthese aus Kapitel 3.4 durchgeführt. Folglich kommt es während des Importierens einer Neo4j-Datenbank neben der einmaligen Neo4j-Propertygraph-Transformation zu einer Propertygraph-KGraph-Transformation.

4.2.1. Import über das lokale Dateisystem

Das Importieren eines Graphen aus einer Datenbank im lokalen Dateisystem erfolgt über die Import-Synthese-Klasse *Neo4jFileSynthesis*. Hierbei handelt es sich um eine in Xtend geschriebene Klassen, wodurch das in Kapitel 2.5 erläuterte Konzept zum Erkennen von bereits erstellten Objektinstanzen zur Verfügung steht.

Die vom Nutzer angegebene Datenbank wird zunächst durch die von Neo4j bereitgestellte *Neo4j Core-Java-API* als *GraphDatabaseService* geladen und mit einer temporären Sperre versehen. Anschließend stehen Methoden zum Bestimmen der vorhandenen Knoten- und Kantentypen zur Verfügung und werden beispielsweise zum Befüllen der zuvor erläuterten Tabellen verwendet.

Die Transformation zwischen der Neo4j-Datenbank und dem Propertygraph-Modell ist eine auf zwei flachen Graphstrukturen basierende Synthese, wodurch die Knoten und Kanten nur kopiert und in den resultierenden Propertygraphen eingefügt werden. In Anlehnung an den in Kapitel 3.4 vorgestellten Algorithmus wird mittels Iteration über die Knotenmenge der Datenbank jede *graphDb.Node*-Instanz in eine *propertygraph.Node*-Instanz überführt. Über rekursive Methodenaufrufe werden neben den vorhandenen Attributen ebenfalls die ausgehenden Kanten eines Knotens mitsamt den entsprechenden Endknoten dupliziert. Das Attribut *Label*, welches für die Beschriftung der im Editor visualisierten Knoten und Kanten zuständig ist, wird initial mit dem Wert des Attributs *displayname* belegt. Sollte dieses nicht bei allen Knoten bzw. Kanten der Neo4j-Datenbank vorhanden sein, werden die entsprechenden Objekte im Propertygraphen mit keinem Label versehen.

Nach dem erfolgreichen Importieren sämtlicher Daten wird die Neo4j-Instanz heruntergefahren und die kurzzeitige Sperre wieder freigegeben.

4.2.2. Import über die REST-Schnittstelle von Neo4j

Im Vergleich zu dem Import über das lokale Dateisystem steht bei der Anbindung über die REST-Schnittstelle keine Java-Bibliothek für den Umgang mit der Datenbank zur Verfügung. Das Auslesen der benötigten Daten geschieht durch das Senden von HTTP GET- und POST-Requests, dessen Inhalt sich aus unterschiedlichen Cypher-Anfragen zusammensetzt. Der Datenaustausch zwischen Datenbankserver und Grapheditor findet im JSON-Format statt, wodurch sowohl die gesendeten Anfragen als auch die Antworten für eine weitere Verarbeitung entsprechend codiert werden. Zur Autorisierung eines Datenbanknutzers werden die Felder *Username* und *Password* ausgelesen und dem Header einer Anfrage als Base64-verschlüsselter String hinzugefügt.

Listing 4.1. Antwort eines lokalen Datenbankservers im JSON-Format auf einen GET-Request an die Adresse *http://localhost:7474/db/data/*


```
1  {
2  "extensions" : { },
3  "node" : "http://localhost:7474/db/data/node",
4  "relationship" : "http://localhost:7474/db/data/relationship",
5  "node_index" : "http://localhost:7474/db/data/index/node",
6  "relationship_index" : "http://localhost:7474/db/data/index/relationship",
7  "extensions_info" : "http://localhost:7474/db/data/ext",
8  "relationship_types" : "http://localhost:7474/db/data/relationship/types",
9  "batch" : "http://localhost:7474/db/data/batch",
10 "cypher" : "http://localhost:7474/db/data/cypher",
11 "indexes" : "http://localhost:7474/db/data/schema/index",
12 "constraints" : "http://localhost:7474/db/data/schema/constraint",
13 "transaction" : "http://localhost:7474/db/data/transaction",
14 "node_labels" : "http://localhost:7474/db/data/labels",
15 "neo4j_version" : "3.0.4"
16 }
```


Listing 4.2. Cypher-Anfrage für Erhalt der zum Kopieren eines Graphen benötigten Informationen

```
1    MATCH (n) OPTIONAL MATCH (n)-[r]->(m) RETURN id(n), n, r, id(m), m
```

Für den Zugriff auf die am häufigsten verlangten Daten einer Datenbank stellt Neo4j verschiedene URLs bereit. Diese sogenannten Wurzel-Services sind in Listing 4.1 angegeben und über einen GET-Request an die Adresse *http://localhost:7474/db/data/* auflistbar. Voraussetzung hierfür ist, dass eine Neo4j-Instanz auf dem lokalen System gestartet wurde.

Wie zu sehen ist, wird für das benötigte Auslesen der in einer Datenbank vorhandenen Kantentypen die URL-Erweiterung *.../db/data/relationship/types* aufgeführt. Mittels einer GET-Anfrage werden die hierfür benötigten Berechnungen seitens des Servers angestoßen und als String-Einträge in einem JSON-Array verzeichnet. Die Tabellen der zweiten Seite des Import-Dialogs werden nach dem Erhalt der Antwort und dem erfolgreichen Parsen um die jeweiligen Typen ergänzt.

Anders als beim Ermitteln der Kantentypen steht für den Erhalt sämtlicher Knoten und Kanten kein direkter Service zur Verfügung. Stattdessen wird mittels der URL-Erweiterung *.../db/data/transaction* die in Listing 4.2 aufgeführte Cypher-Anfrage an die Datenbank gestellt.

Über die allgemeine Form *MATCH (n)* wird sichergestellt, dass jeder Knoten der Datenbank mitsamt den angefügten Attributen in der Antwort des Servers vorhanden

ist. Die eventuell vorhandenen ausgehenden Kanten eines Knotens inklusiver Endknoten werden über den optionalen Zusatz der Anfrage selektiert. Die explizite Ausgabe der Knoten-IDs ist für die erneute Nutzung des Mechanismus zum einmaligen Erzeugen von Objektinstanzen nötig. Ein Knoten im Propertygraphen wird nur erstellt, falls bisher keine Kopie für die ID des ursprünglichen Knotens vorhanden ist. Durch dieses Vorgehen wird garantiert, dass ein Knoten auch bei mehreren eingehenden Kanten stets nur ein Mal transformiert wird.

Aufgrund der Anbindung über die REST-Schnittstelle von Neo4j ist es bei diesem Ansatz nicht notwendig die angegebene Datenbank mit einer temporären Sperre zu versehen. Demnach ist sie jederzeit für andere verfügbar und beispielsweise ein zeitgleiches Importieren in unterschiedliche Grapheditoren denkbar.

Kapitel 5

Das Neo4j-Writer Plugin zum Exportieren eines Graphen

5.1. Die Architektur des Neo4j-Writer Plugins

Der *Neo4j-Writer* ist das Gegenstück zum in Kapitel 4 vorgestellten *Neo4j-Reader* und ermöglicht das Exportieren eines im Editor visualisierten Graphen in eine Neo4j-Datenbank (vgl. Kapitel 1.3.2). Wie beim *Reader* wurde ein eigenständiges Plugin implementiert, welches dem Grapheditor als Erweiterung hinzugefügt wird. Die Arten der Anbindung einer Datenbank sind ähnlich zu denen in Abbildung 4.1, wobei das Sperren der Datenbank wiederum nur beim Export über die *Neo4j Core-Java-API* notwendig ist.

In den folgenden Kapiteln 5.2.1 und 5.2.2 werden die beiden Verbindungsarten detailliert beschrieben. Der Vorgang des Exportierens wird mitsamt des Export-Dialogs in Abschnitt 5.2 erläutert.

5.2. Das Exportieren eines SDG

In Anlehnung an den Import-Wizard wurde für das Exportieren eines Graphen ebenfalls ein eigener Wizard implementiert. Er wird über den Menüeintrag *Export Propertygraph to Neo4j Database* aufgerufen und besteht lediglich aus der in Abbildung 5.1 zu sehenden Dialog-Seite.

Das erste Feld (1) dient zur Auswahl des Graphen, welcher in die unter *Destination* (2) angegebene Datenbank exportiert wird. Ähnlich wie bei der Angabe einer zu importierenden Datenbank unterscheidet sich auch beim Export die Art der Anbindung über den Prefix der Zieldatenbank. Falls eine URL eingegeben und die Verbindung demnach über die REST-Schnittstelle hergestellt wird, ist die eventuell anfallende Autorisierung über die bereitgestellten Felder *Username* und *Password* (3) durchzuführen.

Obwohl beim Exportieren lediglich die flache Struktur des Propertygraphen und nicht die Typisierung oder die Einfärbung eines Graphen in eine Datenbank übertragen wird, lassen sich dennoch zwei Anpassungen vornehmen. Über das Feld *Attribute for node labeling in Neo4j* (4) ist es möglich die exportierten Neo4j-Knoten mit den aus Kapitel 2.4 bekannten Labels zu versehen. Falls bei einem Knoten im Propertygraph der Wert des Feldes als Schlüssel in einem Schlüssel-Wert-Paar angegeben ist, wird dem korrespondieren Neo4j-

Abbildung 5.1. Dialog zum Exportieren eines Graphen

Knoten der Wert des Paares als Knotenlabel hinzugefügt. Für die seitens Neo4j geforderte Zuweisung eines Kantentyps steht das Feld *Attribute for edge types in Neo4j* (5) in ähnlicher Weise zur Verfügung. Sollte der angegebene Schlüssel bei keinem Schlüssel-Wert-Paar einer Kante vorhanden sein, erhält diese den gesonderten Wert *NO_REL_TYPE_FOUND* als Typ zugeordnet.

Durch das Bestätigen des *Finish*-Knopfes wird das Testen der Datenbankeinstellungen und anschließende Importieren mittels einer der in den folgenden Unterkapiteln erläuterten Export-Synthesen gestartet. Unabhängig von der Art der Anbindung wird der Inhalt der Zieldatenbank vor dem Einfügen der Daten komplett gelöscht.

5.2.1. Export über das lokale Dateisystem

Der Export eines Graphen über das lokale Dateisystem verläuft in ähnlicher Weise wie das Importieren eines Graphen aus einer Datenbank. Mittels der verwendeten *Neo4j Core-Java-API* werden sowohl Methoden für das bereits erwähnte temporäre Sperren der Datenbank als auch Java-Klassen für das Inserieren von Neo4j-Knoten und Kanten bereitgestellt. Aufgrund der erneut verwendeten Sprache *Xtend* ist außerdem die einmalige Erstellung einer Neo4j-Knoten- bzw. Kanten-Instanz mittels des Schlüsselworts *create* sichergestellt.

Analog zu der Vorgehensweise aus Kapitel 4.2 findet der Export anhand einer Iteration über die gesamte Knotenmenge des Propertygraphen statt. Sobald ein korrespondierender Neo4j-Knoten erstellt wurde und sämtliche Schlüssel-Wert-Paare exportiert worden sind,

Listing 5.1. Generisches Muster zum Erzeugen einzelner Knoten

```
1   CREATE (n: <nodeType> {properties})
2   RETURN <node>.ID as oldID, id(n) as newID
```

werden die ausgehenden Kanten mitsamt den zugehörigen Endknoten in die Datenbank übertragen. Durch diese ebenfalls beim Import verwendete rekursive Art der Traversierung wird der vollständige Export sämtlicher Objekte eines Propertygraphen gewährleistet.

Während des Exportierens werden die ursprünglichen Objekte des Propertygraphen auf die für das Labeln der Knoten bzw. Zuweisen von Kantentypen angegebenen Attribute untersucht. In Abhängigkeit, ob ein entsprechendes Schlüssel-Wert-Paar vorhanden ist, wird das korrespondierende Neo4j-Duplikat mit dem Wert des jeweiligen Paares oder einem Hinweistext angepasst.

5.2.2. Export über die REST-Schnittstelle von Neo4j

Anders als bei den bisherigen Import- und Export-Synthesen wird beim Exportieren eines Graphen über die REST-Schnittstelle das Konzept von *Xtend* zum Erkennen von bereits erstellten Instanzen nicht verwendet. Um den Export möglichst effizient zu gestalten, wurde stattdessen ein Ansatz gewählt mit dem ein Propertygraph durch zwei POST-Requests in eine Datenbank übertragen wird. Wie in Kapitel 4.2.2 erfolgt der Datenaustausch zwischen dem Grapheditor und der Datenbank im JSON-Format.

Mittels Iteration über die Knotenmenge des zu exportierenden Graphen wird für jeden Knoten zunächst eine *CREATE*-Klausel auf Basis des in Listing 5.1 zu sehenden Musters generisch erzeugt. Jede Anfrage wird in einem separaten JSON-Objekt gespeichert und anschließend einem gemeinsamen JSON-Array zugewiesen.

Durch die *CREATE*-Anweisung in der erste Zeile wird ausgedrückt, dass dem neu erstellten Neo4j-Knoten *n* das Label *nodeType* und die Attribute *properties* hinzugefügt werden. Die Xtend-Variable *<nodeType>* steht hierbei für den Wert des Schlüssel-Wert-Paares, welches als Schlüssel die Eingabe des entsprechenden Feldes aus den Einstellungen zum Export aufweist. Die Schlüssel-Wert-Paare eines zu exportierenden Knotens werden wiederum in das gemeinsame JSON-Objekt *properties* überführt und dem Objekt, welches die Cypher-Anfrage enthält, angefügt. Der Rückgabewert einer *CREATE*-Klausel ist stets ein Paar bestehend aus der ID des exportierten Propertygraph-Knoten, welcher durch die Xtend-Variable *<node>* repräsentiert wird, und der des neu erstellten.

Sobald sämtliche Knoten durch eine Cypher-Anfrage ausgedrückt worden sind, werden die gesammelten Anfragen über einen HTTP-POST-Request an die Datenbank übertragen. Um im nächsten Schritt den bisher noch nicht exportierten Kanten korrekte Start- und Endknoten zuzuweisen, wird die Antwort des Datenbankservers, bestehend aus einer Tabelle von alten und neuen Knoten-IDs, in einer Hash-Map gespeichert. Die resultierende Tabelle stellt folglich eine Zuordnung von IDs dar, wodurch der Zusammenhang zwischen

Listing 5.2. Generisches Muster zum Exportieren einer Kante

```
1   MATCH (a),(b) WHERE id(a) = sourceNodeID AND id(b) =  targetNodeID
2   CREATE (a)-[r: edgeType {props} ]->(b)
```

den Knoten im Propertygraph und den korrespondierenden Neo4j-Knoten gesichert ist.

Für das anschließende Exportieren sämtlicher Kanten wird analog mittels einer Iteration über sämtliche ausgehende Kanten ein gemeinsames JSON-Objekt erstellt, welches für jede Kante eine separate Cypher-Anfrage beinhaltet. Die Basis dieser generisch erzeugten Anfragen ist das in Listing 5.2 dargestellte Muster.

Die Werte der Variablen *sourceNodeID* und *targetNodeID* entsprechen den Neo4j-Knoten-IDs, welche in der Hash-Map unter den IDs der ursprünglichen Start-und Endknoten einer Kante eingetragen worden sind. Die zweite Zeile ist mit der *CREATE*-Anweisung aus Listing 5.1 vergleichbar und drückt das Erstellen der neuen Kante *r* zwischen den ermittelten Knoten *a* und *b* aus. Als Kantentyp wird der Wert des Schlüssel-Wert-Paares verwendet, welches über den im Export-Menu festgelegten Schlüssel verfügt. Ähnlich wie beim Erstellen der Knoten wird auch das Erzeugen der Kanten mit dem Senden der gesammelten Anfragen über einen POST-Request veranlasst.

Aufgrund der Iterationen über die Menge der Knoten und ausgehenden Kanten wird auch ohne die Erkennung von bereits erstellten Objektinstanzen sichergestellt, dass ein Knoten bzw. eine Kante nicht mehrfach exportiert wird. Außerdem wird durch das Gruppieren der unterschiedlichen Cyper-Anfragen die Anzahl an HTTP-Verbindungen möglichst gering gehalten und die Datenbank während des Exportierens entlastet.

Kapitel 6

Die Erweiterungen des bisherigen Grapheditors

Dieses Kapitel umfasst mehrere der für die Ziele 1.3.3 und 1.3.4 notwendigen Anpassungen des zugrundeliegenden Grapheditors aus Abschnitt 2.7. Jedes der folgenden Unterkapitel ist in sich abgeschlossen und beinhaltet die Umsetzung eines der erwähnten Ziele.

In dem Abschnitt 6.1 wird erläutert, in wie weit das Kontextmenü für das Einfügen von Kindknoten und Ausblenden einzelner Menüeinträge erweitert worden ist. Die Anpassungen bezüglich der im Editor bereits vorhandenen Funktionalität zum Einfärben eines Graphen wird in Kapitel 6.2 beschrieben. Die letzten beiden Abschnitte umfassen das durch die hierarchischen Konzepte resultierende Löschen von Elternknoten (siehe Kapitel 6.3) bzw. die Möglichkeit geöffnete Graphen automatisch zu speichern (siehe Kapitel 6.4).

6.1. Das Kontextmenü

Das Kontextmenü stellt wie in Kapitel 2.7 erläutert die Schnittstelle zum Bearbeiten eines visualisierten Propertygraphen dar. Neben dem Löschen von Objekten ist es hierüber beispielsweise möglich neue Knoten einzufügen oder vorhandenen einen vordefinierten Typ zuzuweisen. In der Abbildung 6.1 ist das ursprüngliche Kontextmenü mitsamt den bereits vorhandenen Optionen zum Bearbeiten eines Graphen dargestellt.

Obwohl das Erstellen eines Kindknotens über die Einträge *Add New Node* bzw. *Add Successor* und eine anschließende Typisierung der entsprechenden Kante denkbar ist, erscheint diese Vorgehensweise recht aufwändig. Für eine verbesserte Bedienbarkeit wurden die im folgenden Unterkapitel 6.1.1 erläuterten Menüeinträge implementiert und dem Nutzer verschiedene Optionen zum direkten Zuweisen von Kindknoten bereitgestellt.

Um trotz der erhöhten Anzahl an Menüeinträgen die Übersicht des Menüs zu bewahren, wurde das in Kapitel 6.1.2 beschriebene Ausblenden einzelner Einträge eingeführt.

6.1.1. Neue Optionen zum Einfügen von Kindknoten

Wie bereits erwähnt, ist das Erstellen von Kindknoten nur über die aneinandergereihte Ausführung verschiedener Aktionen möglich. Unabhängig davon, ob es sich um eine

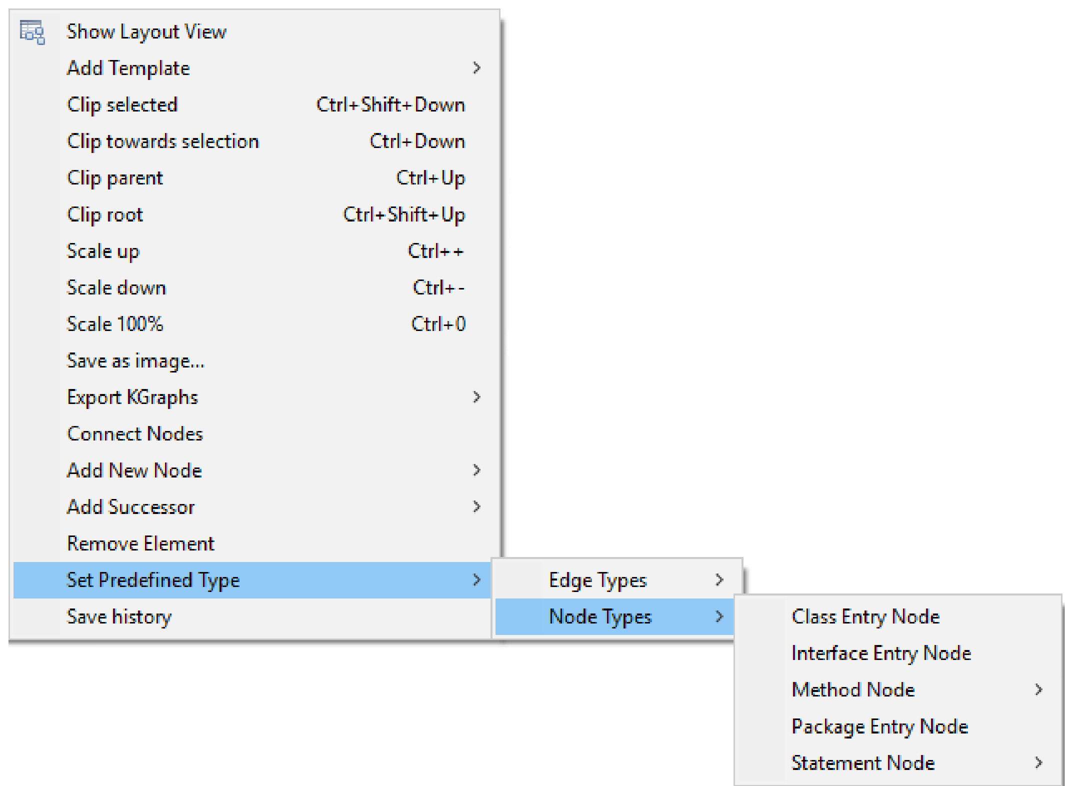

Abbildung 6.1. Das Kontextmenü des zugrundeliegenden Grapheditors

direkte oder vererbte Hierarchie handelt, muss dem Graphen zunächst ein neuer Knoten hinzugefügt werden und dieser anschließend über eine entsprechend getypte Kante mit dem Elternknoten verknüpft werden. Die Fehleranfälligkeit dieser nicht intuitiven Vorgehensweise erschwert dem Nutzer das korrekte Einfügen eines Kindknotens. Beispiele hierfür ist neben dem Verbinden der richtigen Knoten in einem komplexen Graphen auch das Zuweisen des Kantentyps unter dem zur Typisierung der Kanten verwendeten Schlüssel.

Über die neu eingeführten Einträge *Add Child Node* und *Add Inherited Child Node* unter dem Menüpunkt *Add New Child* werden die für das Einfügen eines Kindknotens benötigten Aktionen gekapselt und nacheinander ausgeführt. Analog zu den bereits vorhandenen Optionen wurden die neuen Funktionalitäten auf separate Java-Klassen aufgeteilt und die auszuführenden Anweisungen in einer *execute*-Methode implementiert.

Sobald der Nutzer einen der beiden Menüeinträge auswählt und bestätigt, erscheint der in Abbildung 6.2 dargestellte Dialog zur Auswahl einer eingehenden Hierarchie- oder Vererbungs-Hierarchie-Kante.

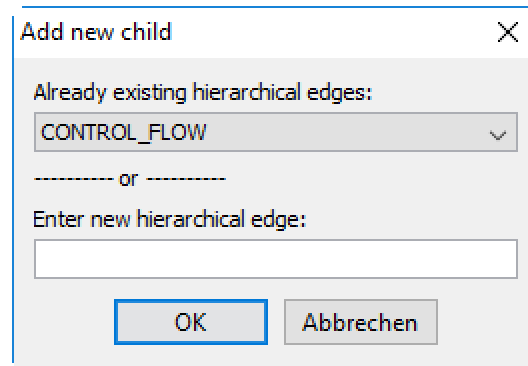

Abbildung 6.2. Dialog zur Auswahl des Kantentyps

In Abhängigkeit von den bereits ein- und ausgehenden Kanten des zuvor per Rechtsklick ausgewählten Elternknotens wird die im oberen Bereich dargestellte *Dropdown*-Liste erstellt. Die einzelnen Einträge der Liste dienen dem Nutzer als Vorschläge für den Wert, welcher in einem Schlüssel-Wert-Paar unter dem Schlüssel zum Typisieren von Kanten gespeichert wird. Sollte der geforderte Wert nicht vorhanden sein, ist ein beliebiger String über die Eingabe in das darunter angeordnete Textfeld spezifizierbar. Abschließend wird durch das Bestätigen des *OK*-Knopfes ein neuer Knoten angelegt, bevor dieser über eine Kante mit dem Elternknoten verbunden wird. Die Darstellung der hierarchischen Beziehung findet aufgrund des der Kante hinzugefügten Schlüssel-Wert-Paares statt.

Falls bei einer Eingabe in das untere Textfeld der Wert bisher nicht in der Tabelle für die Hierarchie- bzw. Vererbungs-Hierarchie-Kanten vorhanden ist, wird diese um einen neuen aktiven Eintrag erweitert.

6.1.2. Ausblenden von einzelnen Menüeinträgen

Bei dem Kontextmenü des ursprünglichen Grapheditors handelt es sich um ein statisches Menü, welches unabhängig von den fokussierten Objekten sämtliche implementierte Funktionen auflistet. Dieses Verhalten erschwert die Bearbeitung eines Graphen, weil dem Nutzer teilweise irritierende Optionen aufgeführt werden. Beispielsweise lässt sich der Menüeintrag *Connect Nodes* ausführen, obwohl in der aktuellen Auswahl an Objekten nur ein Knoten enthalten ist.

Das Ausblenden einzelner Menüeinträgen gewährt, trotz der aus dem vorherigen Kapitel ergänzenden Einträge, eine übersichtlichere Bedienung des Grapheditors. Die Berechnung der zu ignorierenden Einträge findet auf Basis der aktuell fokussierten Graphobjekte statt. Sie werden vor dem Aufruf des Menüs über eine Kombination von Shift- und linker Maustaste ausgewählt. Durch die Bestätigung der rechten Maustaste wird für jeden Menüeintrag überprüft, ob die korrespondierende Bedingung erfüllt ist und der Eintrag

Tabelle 6.1. Voraussetzungen zum Einblenden des entsprechenden Menüeintrags

Menüeintrag:	**Voraussetzungen zum Einblenden**
Connect Nodes	Auswahl besteht aus zwei Knoten
Add New Node	Auswahl besteht aus Wurzelknoten (Rechtsklick auf leeren Bereich des Graphen)
Add New Child	Auswahl besteht aus einem Knoten, aber nicht dem Wurzelknoten
Add Successor	Auswahl besteht aus einem Knoten, aber nicht dem Wurzelknoten
Remove Element	Auswahl besteht aus einem Graphelement, aber nicht dem Wurzelknoten
Set Predefined Node Types	Auswahl besteht aus einem Knoten, aber nicht dem Wurzelknoten
Set Predefined Edge Types	Auswahl besteht aus einer Kante

angezeigt wird. Die Tabelle 6.1 stellt eine Übersicht der zum Anzeigen einer Optionen gehörigen Voraussetzung dar.

Die Umsetzung der Bedingungen wurde mit dem *Eclipse Platform Expression Framework* realisiert, welches ein Bestandteil der in Kapitel 2.2.1 vorgestellten *Rich Client Platform* ist. Das Framework stellt verschiedene Ausdrucksformen und Werkzeuge, wie etwa die Iteration über Objekte, das Ermitteln der Kardinalität einer Menge oder das Implementieren von eigenen Tests, zur Verfügung. Die hierdurch erstellten aussagenlogischen Formeln werden den jeweiligen Einträgen über das vordefinierte Feld *visibleWhen* eines Menüeintrags angefügt. Ein Beispiel für die Kombination aus Eintrag und den zum Einblenden bestimmten Voraussetzungen ist in Listing 6.1 zu sehen.

In den Zeilen 2 bis 9 werden die Eigenschaften des Menüeintrags, wie der anzuzeigende Name (Zeile 4) oder die auszuführenden Java-Klasse (Zeile 8), definiert. Das für dieses Kapitel relevante Konzept zum Ausblenden des Eintrags ist in den darauffolgenden Zeilen 10 bis 25 umgesetzt worden. Um die Sichtbarkeit des Eintrags explizit durch eine aussagenlogische Formel auszudrücken, muss zunächst der Wert des Feldes *checkEnabled* (Zeile 11) des Ausdrucks *visibleWhen* auf *false* gesetzt werden. Andernfalls würde der Zustand des mit dem Menüpunkt verknüpften *Handlers*, über den der Befehl (*command*) aus Zeile 2 mit einer auszuführenden Klasse verknüpft wird, als Basis der Berechnung dienen. Ein *Handler* kann sowohl aktiviert als auch deaktiviert sein, wobei die implementierende Klasse nur im aktivierten Zustand ausgeführt wird.

Der Zugriff auf die aktuell fokussierten Graphelemente erfolgt durch die vordefinierte Variable *activeMenuSelection* aus Zeile 13. Mittels der Iteration über die bereitgestellte Menge (Zeile 14 - 20) wird überprüft, ob es sich bei jedem Objekt um KNode-Instanz handelt (Zeile 17 - 19) und folglich nur visualisierte Knoten ausgewählt worden sind. Die zusätzlich geforderte Anzahl von zwei Graphelementen wird anhand des nachfolgenden Ausdrucks *count* (Zeile 21 - 23) sichergestellt.

Listing 6.1. Implementierung des Menüeintrages *Connect Nodes*

```
1  <command
2     commandId="de.cau.cs.kieler.klighd.ui.actionExecution"
3     id="de.cau.cs.kieler.klighd.ui.connectNodes"
4     label="Connect Nodes"
5     style="push">
6     <parameter
7        name="de.cau.cs.kieler.klighd.ui.action"
8        value="de.cau.se.grapheditor.actions.ConnectNodesAction">
9     </parameter>
10    <visibleWhen
11       checkEnabled="false">
12       <with
13           variable="activeMenuSelection">
14           <iterate
15              ifEmpty="false"
16              operator="and">
17              <instanceof
18                 value="de.cau.cs.kieler.core.kgraph.KNode">
19             </instanceof>
20          </iterate>
21          <count
22             value="2">
23          </count>
24       </with>
25    </visibleWhen>
26 </command>
```

Für die bei den anderen Einträgen benötigte Überprüfung des Wurzelknotens wurde ein separater Test in der namensgebenden Java-Klasse *NoParentTester* implementiert. Dieser ist erfüllt, falls der zu analysierende Knoten über einen Elternknoten verfügt, sodass es sich folglich nicht um den Wurzelknoten eines *KGraphen* handelt.

In Verbindung mit dem vorherigen Kapitel 6.1.1 wurden die Ziele aus Abschnitt 1.3.3 demnach erfolgreich umgesetzt.

6.2. Einfärbung eines Graphen

Dieses Kapitel umfasst die Anpassungen an dem Plugin *colorChoiceDialog*, welches für das Einfärben eines dargestellten Graphen zuständig ist. Die in Abschnitt 1.3.4.1 beschriebenen Ziele wurden durch das Modifizieren der entsprechenden Stellen direkt im Plugin realisiert,

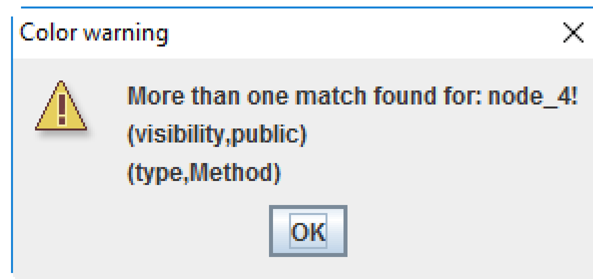

Abbildung 6.3. Farbkonflikt zwischen den Tupeln *(type,Method)* und *(visibility,public)* bei dem Knoten mit der ID *node_4*

sodass die grundlegende Architektur und Funktionsweise des Moduls nicht verändert wurde.

Als Basis einer Einfärbung werden sogenannte *Key-Value-Color*-Tripel verwendet. Verfügt ein Graphobjekt über ein Schlüssel-Wert-Paar mit den Werten *(Key,Value)*, wird es mit dem im Hexadezimalsystem angegeben Farbcode *Color* eingefärbt. Die hierfür anstehenden Berechnungen werden nach dem Verlassen des in Kapitel 2.7 beschriebenen Fenster zur Auswahl der Farben über den *OK*-Knopf durchgeführt. Neben der Iteration über die angegebenen *Key-Value-Color*-Tripel, wird ebenfalls durch sämtliche Graphobjekte und die zugehörigen Schlüssel-Wert-Paare iteriert.

Um den Nutzer auf eine mehrfache Einfärbung eines Knotens oder einer Kante hinzuweisen, wird jeder Schlüssel und Wert eines mit einem Tripel übereinstimmenden Attributs in einer temporären Liste gespeichert. Falls am Ende des Durchlaufs über die Attribute eines Graphobjekts mehr als ein Eintrag in den Listen vorhanden ist, erscheint ein diesbezüglich hinweisender Dialog. Es wird sowohl die ID des betroffenen Knotens als auch die in Konflikt stehenden Schlüssel-Wert-Paare angezeigt. In der Abbildung 6.3 ist ein Beispiel eines solchen Dialogs dargestellt.

Aus Gründen der Performance wird nach dem Bestätigen des Dialogs die Einfärbung des Graphen trotz eventueller mehrfacher Farbgebung komplett durchgeführt. Andernfalls müsste, um sicherzustellen, dass es zu keiner mehrfachen Zuordnung von Farben kommt, eine doppelte Iteration über sämtliche Graphelemente getätigt werden. Einem Objekt würde somit erst nach erfolgreichem Durchlauf der Test-Schleife eine Farbe zugeordnet werden.

Im Vergleich dazu ist bei dem implementierten Ansatz lediglich ein einziger Schleifendurchlauf nötig. Im Falle eines Konflikts wird jeweils der Farbwert der letzten Übereinstimmung von Farb-Tripel und Schlüssel-Wert-Attribut verwendet. Durch die Verwendung des in Kapitel 2.7 vorgestellten *Tree Editors* lassen sich die betroffenen Knoten schnell ermitteln.

Des Weiteren wurde die hexadezimale Farbauswahl um eine auf Strings basierende Eingabe erweitert. Gibt der Nutzer einen der vordefinierten Werte ein, wird beim Verlassen des Dialogs der entsprechende hexadezimale Farbwert in das jeweilige *Key-Value-Color*-

Tripel eingetragen. Die hierfür unterstützten Farben sind: *red, blue, cyan, grey, green, brown, black, white, yellow, purple* und *orange*. Wird ein von diesen Werten verschiedener String eingegeben, der nicht das Format eines HTML-Farbcodes aufweist, werden die jeweiligen Basisfarben (schwarz oder weiß) zum Einfärben verwendet. Zusätzlich bleibt der ungültige Wert beim wiederholten Öffnen des Farbdialogs bestehen und wird nicht mit dem korrespondierenden hexadezimalen Wert ersetzt.

6.3. Löschen von hierarchischen Knoten

Das Löschen von hierarchischen Knoten bzw. Elternknoten wurde für einen intuitiven Umgang mit der hierarchischen Ansicht eines Graphen entwickelt. Bisher wurde beim Löschen eines Knotens lediglich der Knoten selbst sowie die ein- und ausgehenden Kanten entfernt. Durch die Verwendung von nur einem Knotentyp waren sämtliche Knoten flach auf einer Ebene angeordnet und es kam zu keinen syntaktischen Abhängigkeiten.

Mit der Einführung der in Kapitel 3.1 erläuterten Kantentypen ist es in der hierarchischen Darstellungen hingegen möglich, Beziehungen ohne den Gebrauch von Kanten auszudrücken. Trotz des noch immer gemeinsamen Knotentyps lassen sich sowohl Elternknoten von Kindknoten abgrenzen als auch unterschiedliche Ebenen innerhalb eines Graphen definieren. Bei dem bisherigen Löschen eines Elternknoten, wie etwa einem Methodenknoten, würde dies allerdings nicht berücksichtigt werden, wodurch sämtliche Kindknoten (Statement-Knoten) bestehen blieben.

Um dem Nutzer einen möglichst flexiblen Umgang mit dem Editor zu gewährleisten, wurde neben dem Entfernen von einzelnen Objekten ein zweites Verfahren für das Löschen von hierarchischen Knoten bereitgestellt. Die Auswahl welcher der beiden Algorithmen verwendet wird, basiert auf der aktuell gewählten Ansicht eines Graphen. In Folge dessen bleibt das bisherige Löschen von Objekten in der flachen Darstellung vollständig unberührt. Der Vorgang bezüglich des Entfernen eines Elternknoten aus der hierarchischen Darstellung ist auf drei in Abbildung 6.4 zu sehende Aktivitäten verteilt worden.

Anfänglich wird analysiert, um welche Art von Graphobjekt es sich bei dem zu löschenden handelt und ob die hierarchische Ansicht aktiviert ist. Sollte dies nicht der Fall sein oder es sich bei dem ausgewählten Objekt um keinen Knoten handeln, wird das Löschen mittels der bereits im Editor implementierten Methoden durchgeführt [Blümke 2015], [Benekov 2015]. Beim Entfernen eines Knotens aus der flachen Darstellung werden sowohl die Verweise auf ein- und ausgehende Kanten als auch das Objekt selbst aus dem Propertygraph gelöscht. Kanten wiederum werden lediglich durch das Entfernen der Referenzen aus den Kantenmengen der verknüpften Knoten eliminiert.

Vorausgesetzt, dass die hierarchische Ansicht aktiviert und das ausgewählte Objekt ein Knoten ist, wird die Menge der ausgehenden Kanten des Knotens auf aktive Hierarchie-Kanten untersucht. Der ursprüngliche Knoten wird erst gelöscht, sobald sämtliche entlang dieser Kanten verbundenen Kindknoten mittels zweier im Folgenden erklärten rekursiven Methoden entfernt worden sind.

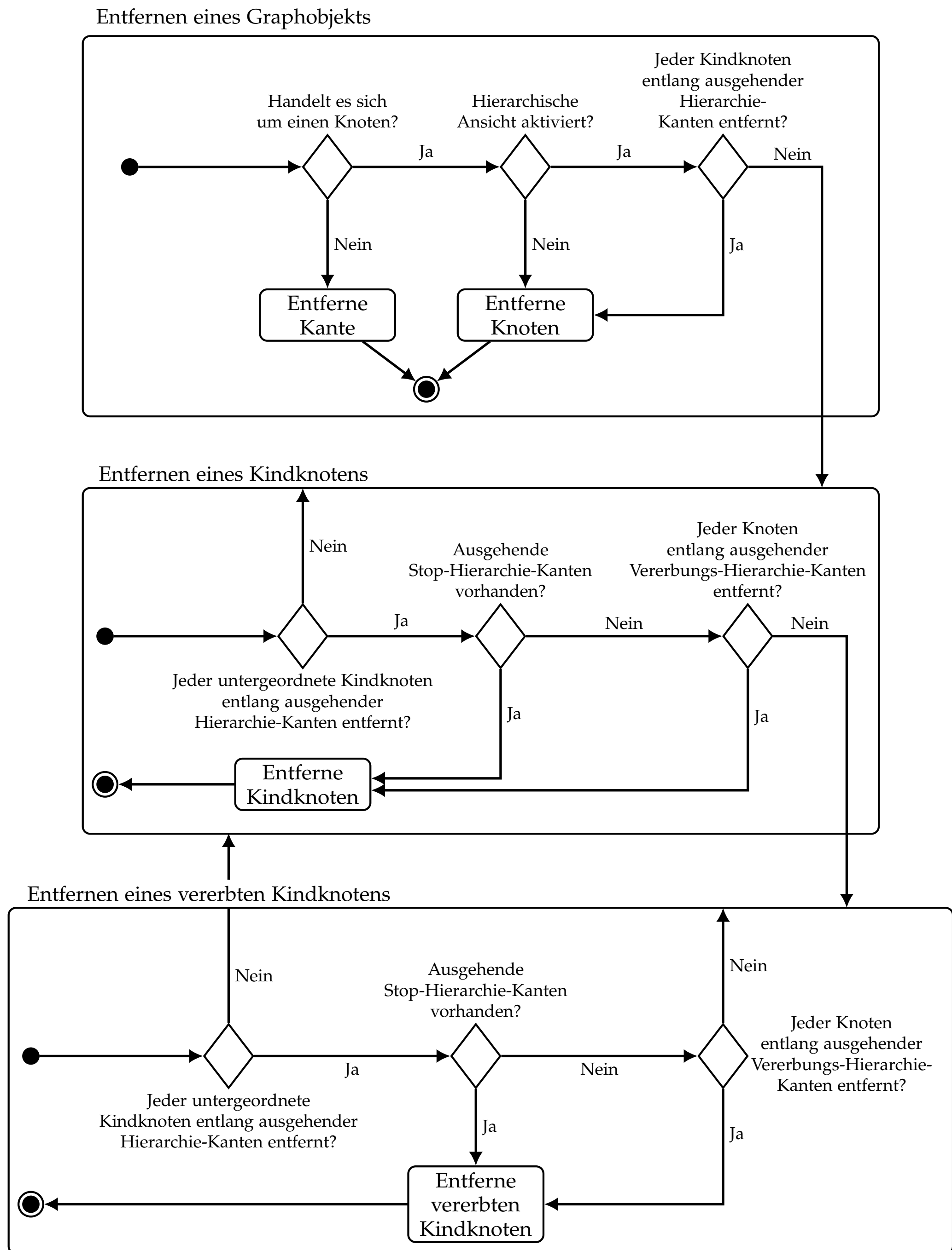

Abbildung 6.4. Aktivitäten beim Löschen eines Graphobjekts

Zunächst werden, wie in dem mittleren Kasten aus Abbildung 6.4 zu sehen ist, die wiederum über Hierarchie-Kanten verbundenen untergeordneten Kindknoten mittels rekursivem Methodenaufruf gelöscht. Sobald dies für einen Knoten geschehen ist, wird die Menge an ausgehenden Kanten auf Stop-Hierarchie-Kanten untersucht. Um keine Knoten außerhalb einer abgeschlossenen Hierarchie fälschlicherweise zu löschen wird bei der Existenz solch einer Kante lediglich der Knoten selbst mitsamt der ein- und ausgehenden Kanten aus dem Graph entfernt. Andernfalls wird abschließend die Menge an Kindknoten, welche entlang ausgehender Vererbungs-Hierarchie-Kanten erreichbar ist, gelöscht.

Diese auf einer Ebene mit dem ursprünglichen Kindknoten befindlichen Knoten werden schrittweise durch den Aufruf der rekursiven Methode, welche im untersten Kasten dargestellt ist, entfernt. Analog zu der vorherigen Methode werden im ersten Schritt sämtliche enthalte Kindknoten entlang der ausgehenden Hierarchie-Kanten entfernt. Ein Beispiel für das an dieser Stelle erneute Auftreten von Kindknoten sind geschachtelte Kontrollflussstrukturen. Anschließend werden erneut in Abhängigkeit von einer vorhandenen Stop-Hierarchie-Kante die mittels Vererbungs-Hierarchie-Kanten erreichbaren Nachfolgerknoten gelöscht. Sobald die Iterationen über die jeweiligen Kantenmengen abgeschlossen wurden und folglich keine der Hierarchie zugehörigen Knoten mehr vorhanden sind, wird schließlich der ursprünglich vererbte Kindknoten aus dem Graph entfernt.

In Folge der verwendeten Iterationen über die jeweiligen ausgehenden Kanten und den rekursiven Methoden wird garantiert, dass beim Löschen eines Elternknotens ebenfalls jegliche enthaltene Kindknoten aus dem Graph entfernt werden. Die Abgrenzung zu Knoten, welche anderen Hierarchien unterliegen, wird über die erwähnte Analyse der Stop-Hierarchie-Kanten gewährleistet.

Trotz des hier beschriebenen Verfahrens bleibt zu bemerken, dass das Entfernen von bestimmten Knoten aus einer hierarchischen Ansicht einen Einfluss auf die dargestellten Hierarchien hat. Beispielsweise wird durch das Löschen eines vererbten Kindknotens (Statementknoten innerhalb eines Methodenknoten) die Hierarchie unterbrochen und nachfolgende Knoten außerhalb des ursprünglichen Elternknoten (Methodenknoten) angeordnet. Um endlose rekursive Aufrufketten zu unterbinden, werden die bereits gelöschten Knoten in einer temporären Liste verzeichnet.

6.4. Periodisches Speichern der geöffneten Graphen

Das periodische Speichern erleichtert dem Nutzer die Bedienung des Editors, sodass ein Graph nach dem Bearbeiten nicht notwendigerweise manuell gespeichert werden muss. Sobald ein über das in Abbildung 6.5 dargestellte Fenster definiertes Zeitintervall abgelaufen ist, werden die geöffneten Graphen unter den entsprechenden Speicherorten gesichert. Der Dialog und die zugrundeliegende Funktionalität ist dem Grapheditor in einem weiteren ergänzenden *saveDialog*-Plugin hinzugefügt worden.

Das Speichern der Graphen ist in einen separaten Thread ausgelagert worden, wodurch der Grapheditor dem Nutzer weiterhin in vollem Umfang zur Verfügung steht. Nachdem

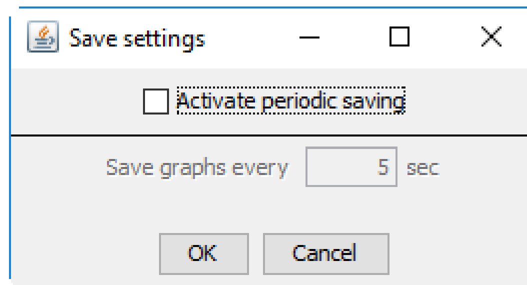

Abbildung 6.5. Der Dialog zum Aktivieren des periodischen Speicherns

die Menge an geöffneten *PropertygraphEditoren* bestimmt worden ist, werden die Graphen mittels einer vom *Eclipse Modeling Framework* bereitgestellten *doSave*-Methode gespeichert. Über das Feld *Activate periodic saving* wird der nebenläufig arbeitende Thread gestartet bzw. angehalten und das periodische Speichern gegebenenfalls unterbrochen.

Kapitel 7

Evaluation

In diesem Kapitel wird der entstandene Grapheditor hinsichtlich verschiedener Kriterien evaluiert. Die durchgeführten Experimente und das zugrundeliegende Setup werden in Abschnitt 7.1 beschrieben. Die verwendeten Datenbankmodelle, welche die Grundlage der Evaluation darstellen, sind mit einer kurzen Beschreibung in Kapitel 7.1.2 aufgelistet. Im Anschluss an die Diskussion der Ergebnisse (siehe Kapitel 7.2) werden eventuelle Unsicherheiten bezüglich der Validität in Abschnitt 7.3 besprochen.

7.1. Methodik

Die Evaluation des Editors wird anhand des *Goal-Question-Metric*-Modells [Van Solingen u. a. 2002] durchgeführt. Jedes der in Abschnitt 7.1.1 aufgeführten Experimente wurde hinsichtlich eines separat zu erreichenden Ziels entworfen. Mittels mehrerer systematischer Fragen wird überprüft, ob ein konkretes Ziel erfüllt wurde. Die Beantwortung dieser Fragen geschieht über ausgewählte Metriken, wie etwa der Anzahl an Knoten innerhalb eines importierten Graphen.

7.1.1. Beschreibung der Experimente

Für die Evaluation des Editors werden drei unabhängige Experimente durchgeführt, wobei mit jedem Experiment ein zu erreichendes Ziel untersucht wird. In den folgenden drei Abschnitten werden die jeweiligen Experimente mitsamt des zugrundeliegenden Ziels separat erläutert. Die beinhaltete Beschreibung der Tabellen dient einem besseren Verständnis der in Kapitel 7.2 beschriebenen und diskutierten Ergebnisse.

7.1.1.1. E1: Vollständiger Import eines Graphen mittels des Neo4j-Reader Plugins

Das erste Ziel ist der vollständige Import eines beliebigen Graphen mittels des entworfenen *Neo4j-Reader* Plugins. Hierfür werden mehrere Beispielgraphen (siehe Abschnitt 7.1.2.1) von der Neo4j GraphGists Website[1], welche eine Ansammlung von Datenbanken verschiedenster Domänen darstellt, ausgewählt. Diese werden über die beiden Anbindungsarten (URL und lokales Dateisystem) als Propertygraphen in den Grapheditor importiert. Um ein

[1]https://neo4j.com/graphgists/ Letzter Zugriff: 29. Oktober 2016

größtmögliches Spektrum an unterschiedlichen Graphen abzudecken, werden diese sowohl in ihrer Größe als auch dem zugrundeliegenden Modell variiert. Folglich wird neben dem Import des aus Kapitel 2.1.3 bekannten SDG auch der Import allgemeiner Modelle, wie beispielsweise einer Smartphonedatenbank, analysiert. Der Vergleich zwischen der ursprünglichen Datenbank und dem visualisierten Graphen erfolgt mittels des internen Neo4j-Browsers und des Grapheditors.

Für die Feststellung, ob ein Graph vollständig importiert worden ist, wurden im Sinne des *GQM*-Ansatzes drei Fragen entworfen. Um diese beantworten zu können, wurden je nach Frage unterschiedlich viele auszuwertende Metriken eingeführt. Aufgrund der Beschaffenheit eines importierten Graphen konnten jedoch nicht ausschließlich allgemein bekannte Metriken, wie etwa die Anzahl an Knoten oder das Verhältnis zwischen möglichen und vorhandenen Kanten eines Graphen, verwendet werden. Obwohl diese Werte teilweise durch den Neo4j-Browser und die erstellten **.propertygraph*-Dateien ersichtlich sind, können sie trotz inhaltlich komplett unterschiedlicher Graphen gleich ausfallen. In Folge dessen wurden für die Berücksichtigung der Attribute eines Knoten bzw. einer Kante unabhängige Metriken eingeführt, welche anhand einer manuellen und visuellen Überprüfung bestimmt werden. Diese erfolgt mittels einer textuelle Betrachtung der dem entsprechenden Propertygraphen zugrundeliegenden Datei. In der Tabelle 7.1 sind die zu beantwortenden Fragen und Metriken untereinander in der linken Spalte aufgeführt.

Die jeweils hervorgehobenen Einträge entsprechen den Fragen und beziehen sich auf die darunter angeordneten Metriken. Beispiele für die erwähnten visuellen Feststellungen sind die Metriken in den Zeilen *Korrespondenz bezüglich der Schlüssel-Wert-Paare* und *Korrekte Zuordnung der Schlüssel-Wert-Paare*. Sie geben Auskunft darüber, ob die neu erstellten Schlüssel-Wert-Paare mit den originalen Knoten- bzw. Kantenattributen korrespondieren und den korrekten Graphelementen zugeordnet worden sind. Für eine möglichst valide Beantwortung der Fragen werden zusätzlich quantitative Metriken, wie beispielsweise der Anzahl an vorhandenen Kanten, verwendet.

Die Spalten *Smartphone*, *Medizin*, *Autohersteller* und *SDG*, welche die in Abschnitt 7.1.2.1 erläuterten Szenarien repräsentieren, sind in jeweils drei weitere Spalten unterteilt. Die Kategorien *Orig. Datei* und *URL* symbolisieren die Arten der Graphen: Originale Datenbank (mittels internen Neo4j-Browser visualisiert), Import über das Dateisystem oder Import über eine URL. Bei der Besprechung der Ergebnisse (siehe Kapitel 7.2) ist demnach stets der Eintrag in der Spalte *Orig.* als korrekte Referenz anzusehen.

Tabelle 7.1. Auflistung der Fragen und Metriken bezüglich des Ziels: Vollständiger Import eines Graphen mittels *Neo4j-Reader*

	Smartphone			Medizin			Autohersteller			SDG		
	Orig.	Datei	URL	Orig.	Datei	URL	Orig.	Datei	URL	Orig.	Datei	URL
Wurden sämtliche Knoten importiert?												
Anzahl der Knoten												
Wurden sämtliche Kanten importiert?												
Anzahl der Kanten												
Korrekte Start- und Endknoten an den Kanten												
Wurden die Attribute korrekt importiert?												
Anzahl der Schlüssel-Wert-Paare an den Knoten												
Anzahl der Schlüssel-Wert-Paare an den Kanten												
Korrespondenz bzgl. der Schlüssel-Wert-Paare												
Korrekte Zuordnung der Schlüssel-Wert-Paare												

7.1.1.2. E2: Vollständiger Export eines Graphen mittels des Neo4j-Writer Plugins

Ein weiteres Ziel ist der vollständige Export eines im Editor visualisierten Graphen. Ähnlich wie beim vorherigen Ziel wird hierfür das neu entwickelte *Neo4j-Writer* Plugin verwendet und die Graphen in eine Neo4j-Datenbank exportiert. Um einen aussagekräftigen Vergleich zwischen dem originalen Datenbankzustand und der neu entstandenen Version zu gewährleisten, werden lediglich die im ersten Ziel importierten und auf Korrektheit geprüften Graphen exportiert. Anders als bei dem vorherigen Ziel, wird der Vergleich der zwei Datenbankzustände einheitlich anhand des Neo4j-Browsers durchgeführt.

Für die Beurteilung, ob das Ziel hinsichtlich eines vollständigen Exports erfüllt worden ist, wurde die vorherige Tabelle 7.1 adaptiert. Die einzigen Unterschiede liegen in der, auf den Export bezogenen Neuformulierung der Fragen und einer detaillierteren Aufteilung der zu analysierenden Datenbanken bzw. Graphen. Um Redundanzen zu vermeiden, wird die leicht umstrukturierte Tabelle an dieser Stelle jedoch nicht erneut dargestellt.

Im Vergleich zu den drei Spalten *Orig.*, *Datei* und *URL* existieren beim Export fünf Felder mit den Überschriften: *Orig.*, *D2D*, *D2U*, *U2D* und *U2U*. Diese stehen neben dem beibehaltenen Eintrag des Originals für vier auf unterschiedlichen Wegen erzeugte Datenbanken und gewährleisten eine ausführlichere Analyse des Experiments. Beispielsweise repräsentiert die Abkürzung *D2U* eine Datenbank, welche durch einen mittels des lokalen Dateisystems importierten Graphen anschließend über eine URL exportierte wurde. Die vier Abkürzungen stehen demnach für sämtliche Kombinationen zwischen der Anbindungsart Datei (D) und URL (U). Auf diese Weise wird sichergestellt, dass bei den importierten Datenbanken der vorherigen Evaluation 7.1.1.1 keine Abweichungen innerhalb eines Szenarios vorhanden sind. Andernfalls wäre es nicht sichergestellte, dass keine Unterschiede, welche aufgrund der gewählten Fragen und Metriken nicht messbar waren, zwischen den auf verschiedene Arten importierten Datenbanken existieren.

Aufgrund des zuvor angesprochenen einheitlichen Vergleichs mit dem Neo4j-Browser, lassen sich die bisherigen Fragen und Metriken erweitern, sodass ebenfalls die Neo4j-charakteristischen Knotenlabels und Kantentypen untersucht werden. Die Tabelle 7.2 stellt eine Ergänzung zu den aus dem Ziel 7.1.1.1 adaptierten Tabelleneinträgen dar und veranschaulicht die detailliertere Aufteilung der exportierten Datenbanken.

Die ersten Metriken beider Fragen beziehen sich jeweils auf die Anzahl der nach dem Im- und Export vorhandenen Knotenlabels bzw. Kantentypen und werden anhand zusätzlicher Informationen direkt im Neo4j-Browser abgelesen. Die Feststellungen, ob die exportierten Labels und Typen mit den ursprünglichen korrespondieren, werden ebenfalls über eine Auflistung sämtlicher enthaltener Einträge zum Kategorisieren von Graphelementen analysiert. Ausschließlich die Überprüfungen der korrekten Zuordnungen geschieht mittels einer manuellen Traversierung durch den Graphen.

Tabelle 7.2. Auflistung der weiteren Fragen und Metriken bezüglich des Ziels: Vollständiger Export eines Graphen mittels *Neo4j-Writer*

	Smartphone			Medizin			Autohersteller			SDG		
	Orig.	D2D	D2U	Orig.	D2D	D2U	Orig.	D2D	D2U	Orig.	D2D	D2U
		U2D	U2U		U2D	U2U		U2D	U2U		U2D	U2U
Übereinstimmung der Knotenlabels zwischen der originalen bzw. exportierten Datenbank?												
Anzahl der unterschiedlichen Knotenlabels												
Korrespondenz bzgl. der Knotenlabels												
Korrekte Zuordnung der Knotenlabels												
Übereinstimmung der Kantentypen zwischen der originalen bzw. exportierten Datenbank?												
Anzahl der unterschiedlichen Kantentypen												
Korrespondenz bzgl. der Kantentypen												
Korrekte Zuordnung der Kantentypen												

7.1.1.3. E3: Korrekte Funktionsweise der hierarchischen Konzepte

Das dritte Ziel befasst sich mit der hierarchischen Ansicht eines im Editor visualisierten Graphen. Durch eine visuelle Überprüfung mehrerer SDG, welche in Abschnitt 7.1.2.2 beschrieben sind, wird die Funktionalität der in Kapitel 3 vorgestellten hierarchischen Konzepte analysiert. Aufgrund des flachen Neo4j-Graphmodells stehen allerdings keine weiteren hierarchischen Darstellungen für einen Vergleich zur Verfügung. Die Überprüfung erfolgt demnach in Anlehnung an die erwarteten hierarchischen Ansichten bezüglich der jeweiligen Kantentypisierungen.

Im Vergleich zu den beiden vorherigen Experimenten basiert die Analyse dieses Experiments auf vier neuen Szenarien namens *if-then-else*, *loops*, *nested* und *method-call*. Hierbei handelt es sich um java-spezifische SDG, welche auf Basis der in Abschnitt 7.1.2.2 genauer erläuterten Beispielprogramme generiert worden sind. Aufgrund des nicht zur Verfügung stehenden Vergleichs mit einer originalen hierarchischen Ansicht repräsentieren die untergeordneten Spalten E und G jeweils den erwarteten bzw. gemessenen Wert. Für die Untersuchung, ob eine korrekte hierarchische Graphdarstellung mittels der implementierten Konzepte generiert wurde, sind die zwei in Tabelle 7.3 aufgelisteten Fragen definiert worden.

Die erste Frage befasst sich mit dem korrekten Abbilden von Hierarchien zwischen den im Graph enthaltenen Knoten. Mittels der quantitativen Metriken, wie etwa der Anzahl an Elternknoten, wird entschieden, ob die einer Kantentypisierung zugrundeliegende, erwartete Menge an Knoten vorhanden ist. Mit der Eigenschaft hinsichtlich der korrekten Zuordnung von Eltern- und Kindknoten wird sichergestellt, dass trotz eventuell richtiger Anzahl an Knoten keine fehlerhaften Hierarchien im visualisierten Graph vorhanden sind. Die beiden abschließenden Zeilen betreffen das Verhalten des Graphen beim Entfalten und Einkapseln eines Knotens. Diese Metriken werden ausschließlich anhand einer visuellen Überprüfung beim Auslösen der jeweiligen Aktion gemessen.

Die zweite der beiden Fragen bezieht sich auf das optionale Aggregieren von Kanten (siehe Kapitel 3.1.4). In Anlehnung an die erste Frage wird neben den quantitativen Messungen ebenfalls das Verhalten des Graphen beim Entfalten und Einkapseln von Knoten untersucht. Das Ermitteln der Werte erfolgt hierbei ausschließlich durch eine visuelle Überprüfung des dargestellten Graphen. Allerdings ist die Reliabilität angesichts der erhöhten Anzahl an zu messenden Metriken dennoch sichergestellt. Die Zeilen hinsichtlich der Anzahl an (nicht-) aggregierten Kanten bei entsprechend eingekapseltem Graph stellen ein Gegenstück zu jenen bei vollständig entfaltetem Graphen dar. Das Wort *entsprechend* beschreibt hierbei einen Graphen, welcher soweit eingekapselt ist, sodass lediglich die aggregierten Kanten sichtbar sind. Beispielsweise würde ein Schleifenknoten, welcher keine weiteren Kontrollflussstrukturen beinhaltet, eingekapselt werden, sodass im Gegensatz zur nicht-aggregierten *CONTROL_FLOW*-Kante des letzten enthaltenen Statementknoten die ausgehende *AGGREGATED_CONTROL_FLOW*-Kante sichtbar ist.

Tabelle 7.3. Auflistung der Fragen und Metriken bezüglich des Ziels: Korrekte Funktionsweise der hierarchischen Konzepte

	if-then-else		loops		nested		method-call	
	E	G	E	G	E	G	E	G
Korrekte Darstellung von Hierarchien?								
Anzahl der Elternknoten								
Anzahl der Kindknoten								
Korrekte Zuordnung zwischen Eltern- und Kindknoten								
Einblendung von Kanten sichtbarer Kindknoten								
Ausblendung von Kanten nicht sichtbarer Kindknoten								
Korrektes Verhalten bei Kanten-Aggregationen?								
Anzahl der sichtbaren aggregierten Kanten bei vollständig entfaltetem Graphen								
Anzahl der sichtbaren nicht-aggregierten Kanten bei vollständig entfaltetem Graphen								
Anzahl der sichtbaren aggregierten Kanten bei entsprechend eingekapseltem Graphen								
Anzahl der sichtbaren nicht-aggregierten Kanten bei entsprechend eingekapseltem Graphen								
Einblendung von aggregierten Kanten bei Einkapselung eines Elternknotens								
Ausblendung von nicht-aggregierten Kanten bei Einkapselung eines Elternknotens								
Ausblendung von aggregierten Kanten bei Entfaltung eines Elternknotens								
Einblendung von nicht-aggregierten Kanten bei Entfaltung eines Elternknotens								

7.1.2. Szenarien

7.1.2.1. Import bzw. Export

Für die Evaluation des Im- und Exports werden vier grundlegend von einander verschiedene Szenarien bzw. Datenbankmodelle verwendet. Zusätzlich wurde bei der Auswahl auf eine variierende Anzahl an Knoten, Kanten und Attributen bezüglich beider Graphelemente geachtet. Aufgrund der in Kapitel 7.2.1 angesprochenen manuellen und visuellen Überprüfung einiger Metriken, war es jedoch nicht möglich Datenbankzustände mit einer äußerst hohen Anzahl an Objekten zu verwenden.

Das erste Szenario repräsentiert eine Smartphone-Datenbank, in der unterschiedliche Handys mitsamt ihren Herstellern enthalten sind[2]. Die weitere Charakterisierung erfolgt beispielsweise durch eine Relation zu einem Betriebssystem oder Typ, wobei hiermit die Eingabeart gemeint ist: *Qwerty*, *Numpad* oder *Touch Screen*. Des Weiteren sind die Kanten mit einer Gewichtung versehen, welche zur Erstellung einer Rangliste von empfohlenen Handys dient.

Ein weiteres Datenbankmodell befasst sich mit der Klassifizierung von Medikamenten auf Basis des anatomischen Wirkungsbereichs[3]. Neben den Inhaltsstoffen sind außerdem unterschiedliche Dosierungen und die entsprechend einzunehmenden Produkte, wie etwa eine Tablettensorte, angegeben. Der Zusammenhang zwischen den enthaltenen Bestandteilen und den verschiedenen Körperbereichen wird über Instanzen des Knotentyps *Wirkungsweise* hergestellt. Eine Besonderheit dieses Modells liegt in der Verteilung der angefügten Attribute, welche ausschließlich bei den Knoten vorhanden sind.

Das nächste Szenario besteht aus einem, im Vergleich zu den vorherigen Modellinstanzen, dicht vernetzten Graphen von Autoherstellern aus dem Jahr 2013[4]. Die beiden enthaltenen Knotentypen Hersteller und Ausstatter sind mittels unterschiedlicher Relationen zu Kooperationen oder Besitztümern verknüpft worden. Das besondere Merkmal dieses Szenarios besteht neben dem angesprochenen Knoten-Kanten-Verhältnis zusätzlich in einem hohen Aufkommen an zu parsenden Knotenattributen.

Im letzten Szenario wird ein java-spezifischer SDG (siehe Kapitel 2.1.3) eines Programms, in dem fünf vorgegebene Zahlen sortiert werden, im- bzw. exportiert. Die Datenbank wurde mittels des in Kapitel 1.2 erwähnten Tools namens *NeoSuit*, welches den beschriebenen Ansatz von C. Wulf implementiert, generiert. Der hierfür zugrundeliegende Quellcode ist in Listing 7.1 aufgeführt.

Mit der Durchführung dieses Szenarios wird die primäre Verwendung des Grapheditors in Verbindung mit dem java-spezifischen SDG-Modell analysiert. Etwaige hierarchische Konzepte werden allerdings erst mittels der folgenden Szenarien aus Abschnitt 7.1.2.2 untersucht, sodass auch in diesem Szenario lediglich der Im- und Export überprüft wird.

[2]https://gist.github.com/axnux/8120907 Letzter Zugriff: 29. Oktober 2016

[3]https://github.com/neo4j-examples/graphgists/blob/master/medical/central_hospital_of_asturias.adoc Letzter Zugriff: 29. Oktober 2016

[4]https://neo4j.com/graphgist/e58948b2-4714-41b3-8f82-ff80e13be3cb#listing_category=transportation-and-logistics Letzter Zugriff: 29. Oktober 2016

Listing 7.1. Programm zum Sortieren eines vorgegebenen Arrays in Java

```
1  public class Sort {
2
3     public static void main(String[] input) {
4        int[] numbers = {4,1,2,3,0};
5        Arrays.sort(numbers);
6        for (int i = 0; i<numbers.length; i++) {
7           System.out.print(numbers[i]);
8        }
9     }
10 }
```

7.1.2.2. Hierarchie

Die Evaluation der hierarchischen Konzepte basiert auf vier, ebenfalls mittels des Tools *NeoSuit*, generierten SDG. Die Grundlage dieser Graphen sind wiederum vier simpel gehaltene Beispielprogramme, welche unterschiedliche Fokusse hinsichtlich der enthaltenen Programmstrukturen legen. Die Analyse des SDG einer komplexen, allumfassenden Anwendung ist demnach nicht notwendig. Stattdessen werden die verschiedenen Konzepte aus Kapitel 3.1 anhand mehrerer kleiner ausfallenden und folglich zuverlässig zu analysierenden Graphen überprüft.

In dem ersten Programm (siehe Listing 7.2) liegt der Schwerpunkt auf den enthaltenen Verzweigungen. Neben dem alleinigen *if*- und erweiterten *if-then-else*-Statement ist zusätzlich ein *case*-Ausdruck mit zwei Fällen vorhanden. Der Zweck dieses Szenarios liegt in der Feststellung, ob die jeweiligen Verzweigungen durch entsprechende Elternknoten ausgedrückt werden. Durch das Entfalten und Einkapseln sollte es demnach möglich sein, die Anweisungen aus den Zeilen 8, 10, 12 und 14 bis 19 darzustellen bzw. zu verbergen. Neben der Analyse der Hierarchie- und Vererbungs-Hierarchie-Kanten aus den Kapiteln 3.1.1 und 3.1.2 lässt sich ebenfalls die korrekte Funktionsweise der optionalen Aggregations-Tupel aus Abschnitt 3.1.4 untersuchen.

Das nächste Szenario, welches in Listing 7.3 zu sehen ist, befasst sich mit den unterschiedlichen Schleifenkonstrukten in Java. Ähnlich wie bei dem vorherigen Programm wird der SDG dieser Anwendung auf mehrere Elternknoten, die den Inhalt der entsprechenden Schleife als Kindknoten beinhalten, untersucht. Analog zum ersten Szenario wird ebenso das Darstellen von statischen Hierarchien zwischen Klassen- und Methodenknoten sowie die korrekte Funktionsweise der Kanten-Aggregationen überprüft. Folglich wird bei diesem Experiment ebenfalls die Korrektheit der Hierarchie- und Vererbungs-Hierarchie-Kanten und der Aggregations-Tupel analysiert. Obwohl das Konzept der Stop-Hierarchie-Kanten bei dem bisherigen SDG bereits verwendet worden ist, wird die ausführliche Evaluation dieser erst im folgenden Szenario durchgeführt.

Listing 7.2. Beispielprogramm zur Analyse der hierarchischen Darstellung von Verzweigungen in Java

```
1   public class IfThenElse {
2
3       public static void main(String[] args) {
4           int number1 = (int) (Math.random() * 100);
5           int number2 = (int) (Math.random() * 100);
6           int number3 = (int) (Math.random() * 2);
7           if ( number1 == number2 )
8               System.out.println("Same␣numbers!");
9           if ( number1 > number2 )
10              System.out.println( "Number␣1␣ist␣greater␣than␣number␣2!" );
11          else
12              System.out.println( "Number␣2␣ist␣greater␣than␣number␣1!" );
13         switch (number3) {
14             case 0:
15                 System.out.println( "Number␣3␣is␣0!" );
16                 break;
17             case 1:
18                 System.out.println( "Number␣3␣is␣1!" );
19                 break;
20         }
21      }
22  }
```

Das dritte Programm repräsentiert eine Kombination der beiden zuvor beschriebenen. Wie in Listing 7.4 zu erkennen ist, sind sowohl Schleifen als auch Verzweigungen in einander verschachtelt worden. Bei der Untersuchung wird daher verstärkt auf die korrekte Darstellung und Verhaltensweise der Elternknoten geachtet, welche selbst als Kindknoten eines weiteren Elternknotens existieren. Dies betrifft insbesondere die erstellten Knoten der Zeilen 8 bis 10, 12 bis 14 und 18 bis 20. Das richtige Aggregieren von Kanten und Visualisieren von statischen Hierarchien wird gleichermaßen wie bei den zuvor beschriebenen SDG untersucht. Die Besonderheit dieses Szenarios liegt in der bisher nicht überprüften Verschachtelung mehrerer Kontrollflussstrukturen, wodurch der Fokus auf den Stop-Hierarchie-Kanten aus Kapitel 3.1.3 liegt. Sollten diese nicht wie erwartet funktionieren, würden die verschiedenen Strukturen über keine klar definierten Abgrenzungen zueinander verfügen und die korrekte Zuordnung zwischen Eltern- und Kindknoten verloren gehen.

Im letzten Beispielprogramm liegt der Fokus auf der Interaktion zwischen verschiedenen Paketen, Klassen und Methoden. Neben den klassenübergreifenden Konstruktoraufrufen werden ebenfalls statische Methoden ausgeführt, welche jedoch allesamt relativ

Listing 7.3. Beispielprogramm zur Analyse der hierarchischen Darstellung von Schleifen in Java

```
1  public class Loops {
2
3     public static void main(String[] args) {
4        int sum = 0;
5        for (int i=1; i < 10; i++) {
6           sum += i;
7        }
8        while (sum > 42) {
9           System.out.println("This␣is␣not␣correct!");
10          sum /= 2;
11       }
12       do {
13          System.out.println("This␣is␣also␣not␣correct!");
14          sum++;
15       } while (sum < 42);
16       System.out.println("Now␣it's␣correct!");
17    }
18 }
```

simpel gehalten sind. Der Zweck dieses Szenarios liegt in der detaillierten Analyse der korrekten Funktionsweise von Aggregations-Tupel. In dem generierten SDG sind für einen Methodenaufruf jeweils zwei in den Methodenknoten eingehende Kanten vorhanden, wobei die *CALLS*-Kante das funktionsaufrufende Statement mit der Methode verbindet. Durch die zweite Kante (AGGREGATED_CALLS) hingegen wird die, jenes Statement umschließende Methode mit der aufgerufenen Methode verknüpft, wodurch ein gewisses Maß an Redundanz entsteht. Folglich bietet es sich an, die erwähnten Kanten zu einem Aggregations-Tupel zusammenzufassen und zu überprüfen, ob je nach Zustand der Knoten lediglich eine der beiden für den Nutzer sichtbar ist. Obwohl die korrekte Implementierung der Tupel bereits innerhalb der anderen Szenarien untersucht worden ist, lässt sich eine zuverlässige Aussage erst nach der Evaluation des diesem Programm zugrundeliegenden SDG treffen. Aufgrund der Menge an Programmzeilen und des nicht erkennbaren Kontrollflusses wurde bei diesem Szenario auf ein Listing bzw. Klassendiagramm verzichtet.

7.1.3. Setup

7.1.3.1. Hardware

Durchgeführt wird die Evaluation auf einem Acer-Aspire-7745G Laptop. Das System verfügt über einen x64-basierten Intel Core i5-Prozessor (430M) mit 2,26 GHz, wodurch bis zu vier Threads parallel verarbeitet werden. Als Speicher stehen 4 GB RAM und eine 500

Listing 7.4. Beispielprogramm zur Analyse der hierarchischen Darstellung von verschachtelten Kontrollflussstrukturen in Java

```
1   public class Nested {
2
3      public static void main(String[] args) {
4         int number = (int) (Math.random() * 100);
5         int sum = 0;
6
7         if (number > 50) {
8            for (int i=number; i > 0; i--) {
9               sum += i;
10           }
11        } else {
12           if (number < 25) {
13              sum = number * 2;
14           }
15        }
16
17        while (sum > 100) {
18           if (number < 50){
19              number *= 2;
20           }
21           sum -= number;
22        }
23        System.out.println("Done!");
24     }
25  }
```

GB große HDD Festplatte mit 5200 RPM zur Verfügung. Die Grafikausgabe erfolgt über eine ATI Mobility Radeon HD 5850 mit DirectX 11 Unterstützung und 1 GB Videospeicher.

7.1.3.2. Software

Das System wird über eine Windows 10 Home 64-Bit-Version betrieben. Zum Ausführen des Grapheditors wird das Eclipse-Modeling-Tools-Paket in der Version Mars.2 Release (4.5.2) verwendet. Als Java-Umgebung kommt das JRE 8 Update 101 zum Einsatz. Die visualisierten Graphen werden mit der Version 0.9.0.201608240737 des KLighD-Frameworks dargestellt. Des Weiteren erfolgt neben dem Importieren auch das Exportieren der Neo4j-Datenbanken mittels der frei erhältlichen Neo4j Community Edition in der Version 2.3.6. Der für den Vergleich der Graphen benötigte Neo4j-Browser stammt ebenfalls aus dieser Version.

7.2. Ergebnisse und Diskussion

Dieses Kapitel beinhaltet die Ergebnisse der drei in Abschnitt 7.1.1 beschriebenen Experimente. In den folgenden Unterkapiteln werden die ermittelten Werte bezüglich der zuvor eingeführten Metriken beschrieben und diskutiert. Die Fragestellung, ob das jeweilige Ziel erfüllt worden ist, wird in einer abschließenden Aussage beantwortet.

7.2.1. Ergebnisse und Diskussion zu E1: Import über den Neo4j-Reader

Zur Analyse des Experiments E1 werden die bereits aus Kapitel 7.1.1.1 bekannten Fragen und Metriken verwendet. In der Tabelle 7.4 sind die quantitativen Werte in den entsprechenden Zeilen und Spalten gegenübergestellt worden. Bei den zu ermittelnden Feststellungen, welche lediglich als erfüllt oder nicht erfüllt angesehen wurden, sind die Tabellenzellen mit aussagekräftigen Symbolen (✓ bzw. ✗) ausgefüllt worden. Die mit Klammern versehenden Häkchen sollen hervorheben, dass dieser Eintrag trivialer Weise erfüllt ist. Beispielsweise verfügt der originale Graph offensichtlich über richtige Start- und Endknoten an den Kanten.

Unabhängig von den Szenarien weisen die drei Tabellenzellen innerhalb einer quantitativen Metriken stets den gleichen Wert auf. Sowohl die Anzahl an Knoten und Kanten als auch die der zugehörigen Attribute stimmen restlos in den Spalten *Orig.*, *Datei* und *URL* überein. Szenarienübergreifend hingegen kommt es in allen vier Metriken zu unterschiedlichen Werten. Beispielsweise variiert je nach Datenbank die Anzahl der Schlüssel-Wert-Paare an den Kanten von 0 bis 111. Passend zu der Erklärung aus Kapitel 7.1.2.1 fallen die Zahlen hinsichtlich der Knoten- und Kantenanzahl nicht äußerst groß aus. Die Zellen der visuell zu ermittelnden Feststellungen, wie etwa der korrekten Zuordnung von Start- und Endknoten, wurden mit einem Häkchen ausgefüllt und sind folglich als erfüllt anzusehen.

Die erste Frage, ob sämtliche Knoten importiert wurden, lässt sich anhand der Übereinstimmungen innerhalb eines Szenarios ablesen. Sowohl bei den über das Dateisystem als auch den über eine URL importierten Graphen gleichen die Werte dem Original. Demnach wurde die Eigenschaft, dass bei den kopierten Graphen weder zusätzliche Knoten erstellt noch zuvor vorhandene entfernt worden sind, sichergestellt. Allerdings wird die Korrektheit der neu erstellten Knoten erst durch die Frage und Metriken der letzten Tabellenzeilen beantwortet.

Für die Beantwortung der zweiten Frage bedarf es neben der quantitativen Aussage noch einer qualitativen. Die Übereinstimmungen in den jeweiligen Zeilen stellen sicher, dass es sich bei dem importierten Graphen nicht um ein Abbild des originalen mit gleicher Knoten- und Kantenanzahl aber unterschiedlichen Abhängigkeiten handelt. Folglich sind lediglich die Kanten der ursprünglichen Datenbank in dem importierten Graphen wiederzufinden.

Die abschließende Frage bezüglich der kopierten Knoten- und Kantenattribute zielt auf die Korrektheit der importierten Graphelemente ab. Wie zu erkennen ist, stimmen die ermittelten Werte hinsichtlich beider Zahlen an Schlüssel-Wert-Paaren in den jeweiligen Szenarien überein. Jedes Attribut aus der ursprünglichen Neo4j-Datenbank wurde in

ein entsprechendes Schlüssel-Wert-Paar des Propertygraph-Modells (siehe Kapitel 2.1.1) überführt. Demnach ist sichergestellt, dass jeder Knoten und jede Kante mitsamt ihren ursprünglichen Attributen korrekt importiert worden ist.

Abschließend lässt sich sagen, dass alle importierten Graphen eine fehlerfreie Kopie des originalen Datenbankzustandes repräsentieren. Die für den Editor zur Visualisierung benötigten Propertygraphen wurden unabhängig von dem zugrundeliegenden Datenbankmodell des jeweiligen Szenarios erstellt. Folglich wurde belegt, dass sich der entstandene Grapheditor neben dem Import und der Bearbeitung eines SDG ebenfalls für den Einsatz in Verbindung mit anderen Graphmodellen eignet. Einzig das für Systemabhängigkeitsgraphen optimierte Kontextmenü stellt eine mögliche Einschränkung bezüglich der Modifikationen anderer Graphen dar.

Tabelle 7.4. Ermittelten Werte zu den Fragen und Metriken bezüglich des Ziels E1

	Smartphone			Medizin			Autohersteller			SDG		
	Orig.	Datei	URL	Orig.	Datei	URL	Orig.	Datei	URL	Orig.	Datei	URL
Wurden sämtliche Knoten importiert?												
Anzahl der Knoten	30	30	30	48	48	48	27	27	27	44	44	44
Wurden sämtliche Kanten importiert?												
Anzahl der Kanten	70	70	70	47	47	47	90	90	90	80	80	80
Korrekte Start- und Endknoten an den Kanten	(✓)	✓	✓	(✓)	✓	✓	(✓)	✓	✓	(✓)	✓	✓
Wurden die Attribute korrekt importiert?												
Anzahl der Schlüssel-Wert-Paare an den Knoten	47	47	47	114	114	114	149	149	149	282	282	282
Anzahl der Schlüssel-Wert-Paare an den Kanten	70	70	70	0	0	0	88	88	88	111	111	111
Korrespondenz bzgl. der Schlüssel-Wert-Paare	(✓)	✓	✓	(✓)	✓	✓	(✓)	✓	✓	(✓)	✓	✓
Korrekte Zuordnung der Schlüssel-Wert-Paare	(✓)	✓	✓	(✓)	✓	✓	(✓)	✓	✓	(✓)	✓	✓

7.2.2. Ergebnisse und Diskussion zu E2: Export über den Neo4j-Writer

Die Ergebnisse der Evaluation 7.1.1.2, welche sich auf die für den Export adaptierten Fragen und Metriken der Tabelle 7.4 beziehen, decken sich in sämtlichen Bereich mit den Werten des vorherigen Kapitels 7.2.1. Unabhängig von der gewählten Kombination an Datenbankanbindungen (D2D, D2U, U2D oder U2U) stimmen sowohl die quantitativen Einträge als auch die manuell und visuell zu untersuchenden Metriken überein. Bei allen vier Szenarien wurden die im Propertygraph enthaltenen Knoten und Kanten mitsamt ihren angehängten Attributen vollständig in eine Neo4j-Datenbank exportiert. Aufgrund des deckungsgleichen Inhalts mit der Tabelle 7.4 wurde allerdings darauf verzichtet die expliziten Werte an dieser Stelle erneut zu präsentieren. Stattdessen wird der Fokus in diesem Abschnitt auf die Einträge der folgenden Tabelle 7.5 hinsichtlich der Neo4j-spezifischen Knotenlabels und Kantentypen gelegt.

Bei den drei Szenarien Smartphone, Medizin und Autohersteller ist zu sehen, dass keine der exportierten Datenbank bezüglich der Anzahl an Knotenlabels oder Kantentypen mit dem Original übereinstimmt. Beide Mengen sind jeweils auf null Label bzw. einen Typ reduziert worden, welcher jedoch nicht mit einem der originalen Werte korrespondiert. Folglich wurde, wie zu erkennen, auch keine korrekte Zuordnung der Knotenlabels und Kantentypen an den jeweiligen Graphelementen durchgeführt. Im letzten Szenario (SDG) hingegen decken sich sowohl die gemessenen Quantitäten als auch die visuell überprüften Eigenschaften mit dem ursprünglichen Datenbankzustand.

Der Grund für die, unabhängig von der gewählten Datenbankanbindung, korrekten Exporte ist der im vorherigen Schritt importierte Datenbankzustand. Das dem SDG zugrundeliegende Datenbankmodell verfügt über ein Feld namens *type*, welches bei jedem im Graph enthaltenen Element als Attribut vorhanden ist. Falls einem Graphelement ein Knotenlabel respektive Kantentyp zugeordnet worden ist, wird der jeweilige Wert ebenfalls in diesem Attribut gespeichert. Obwohl es sich hierbei um redundante Informationen handelt, ist jene Art der Speicherung von großer Relevanz.

Das in Kapitel 3.3 beschriebene Modell des Propertygraphen unterstützt bis auf die Schlüssel-Wert-Paare keine weiteren Möglichkeiten zum Differenzieren von verschiedenen Knoten- oder Kantentypen. Dementsprechend lassen sich weder die Neo4j-spezifischen Knotenlabels noch die Kantentypen in eine neue Graphinstanz einfügen, ohne ein neues Schlüssel-Wert-Paar zu erstellen. Dieses künstlich generierte Paar könnte jedoch bei der im Grapheditor implementierten Darstellung sämtlicher Schlüssel-Wert-Paare eines Graphelements nicht von den ursprünglichen Attributen unterschieden werden. Beim anschließenden Exportieren wäre es erforderlich das entsprechende Schlüssel-Wert-Paar explizit anzugeben. Andernfalls würden die zusätzlich erzeugten Paare in gleicher Weise wie die ursprünglich vorhandenen als Neo4j-Attribute exportiert und nicht wieder entfernt werden.

Um diese Verschmelzung von Elementkategorisierungen und Attributen zu verhindern, werden sowohl die Knotenlabels als auch die Kantentypen beim Importieren einer Neo4j-Datenbank ignoriert. Ungeachtet dessen wurde das Exportieren um eine Schlüssel-Angabe

ergänzt, sodass es dennoch möglich ist Labels und Typen für die exportierten Datenbanken zu definieren. Allerdings muss hierfür ein entsprechendes Attribut, wie im Szenario SDG, bereits beim Importieren vorhanden sein oder im Anschluss manuell, mittels des Editors, ergänzt werden.

Die Unterschiede bezüglich der Anzahl an vorhandenen Labels und Typen basiert auf dem verwendeten Neo4j-Datenbanksystem, in dem jede Kante über mindestens einen Typ verfügen muss. Die Verwendung von Knotenlabels allerdings ist rein optional.

Obwohl bei drei der vier Szenarien die exportierten Datenbanken nicht dem Original entsprechen, sind die Anforderungen an den implementierten Grapheditor dennoch erfüllt worden. Besonders im letzten Szenario wird verdeutlicht, dass es möglich ist, einen bereits beim Importieren über die nötigen Attribute verfügenden Graphen als exakte Kopie zu exportieren. Bei den übrigen Datenbankmodellen stellen sich einzig die Neo4j-spezifischen Knotenlabels und Kantentypen als problematisch dar. Die grundlegenden Komponenten, wie Knoten oder Kanten, wurden hingegen mitsamt ihren Attributen fehlerlos exportiert. Die Berücksichtigung der Labels und Typen beim Import könnte Bestandteil einer auf dieser Thesis aufbauenden Arbeit sein.

Tabelle 7.5. Ermittelte Werte zu den ergänzenden Fragen und Metriken bezüglich des Ziels E2

	Smartphone			Medizin			Autohersteller			SDG		
	Orig.	D2D	D2U	Orig.	D2D	D2U	Orig.	D2D	D2U	Orig.	D2D	D2U
		U2D	U2U		U2D	U2U		U2D	U2U		U2D	U2U
Übereinstimmung der Knotenlabels zwischen der originalen bzw. exportierten Datenbank?												
Anzahl der unterschiedlichen Knotenlabels	5	0	0	6	0	0	2	0	0	11	11	11
		0	0		0	0		0	0		11	11
Korrespondenz bzgl. der Knotenlabels	(✓)	✗	✗	(✓)	✗	✗	(✓)	✗	✗	(✓)	✓	✓
		✗	✗		✗	✗		✗	✗		✓	✓
Korrekte Zuordnung der Knotenlabels	(✓)	✗	✗	(✓)	✗	✗	(✓)	✗	✗	(✓)	✓	✓
		✗	✗		✗	✗		✗	✗		✓	✓
Übereinstimmung der Kantentypen zwischen der originalen bzw. exportierten Datenbank?												
Anzahl der unterschiedlichen Kantentypen	4	1	1	5	1	1	3	1	1	13	13	13
		1	1		1	1		1	1		13	13
Korrespondenz bzgl. der Kantentypen	(✓)	✗	✗	(✓)	✗	✗	(✓)	✗	✗	(✓)	✓	✓
		✗	✗		✗	✗		✗	✗		✓	✓
Korrekte Zuordnung der Kantentypen	(✓)	✗	✗	(✓)	✗	✗	(✓)	✗	✗	(✓)	✓	✓
		✗	✗		✗	✗		✗	✗		✓	✓

7.2.3. Ergebnisse und Diskussion zu E3: Hierarchische Konzepte des Grapheditors

Die ermittelten Werte zu den Fragen und Metriken aus Kapitel 7.1.1.3 sind in der Tabelle 7.6 aufgelistet. Analog zu den vorherigen Ergebnissen sind die Zeilen, welche keine quantitativen Metriken repräsentieren, mit einem ✓ oder einem ✗ ausgefüllt worden. Zellen mit einem eingeklammerten Häkchen verdeutlichen wie zuvor, dass es sich um eine triviale Feststellung handelt. Sie sind demnach in den Feldern hinsichtlich der erwarteten Werte (E) vorhanden. Für die Erzeugung der in dieser Evaluation überprüften Hierarchien wurde in sämtlichen Szenarien eine einheitliche Kantentypisierung vorgenommen. Die Ergebnisse repräsentieren dementsprechend nicht die Werte sämtlicher eventuell durchführbaren Typisierungen.

Wie zu sehen ist, stimmen sämtliche gemessenen Werte (G) mit den ursprünglich erwarteten (E) innerhalb eines Szenarios überein. Unabhängig von der übergeordneten Frage einer Metrik gleichen sich beispielsweise sowohl die Anzahl an vorhandenen Eltern- und Kindknoten als auch an sichtbaren aggregierten Kanten innerhalb eines entfalteten Graphen. Die erhöhten quantitativen Werte des letzten Szenarios ist auf die in Abschnitt 7.1.2.2 beschriebene Evaluation der Aggregations-Tupel zurückzuführen. Die große Anzahl an Kanten repräsentiert die erwähnten Methodenaufrufe innerhalb des zugrundeliegenden Quellcodes.

Neben den Übereinstimmungen bezüglich der messbaren Werte, ist ebenfalls bei den manuell und visuell zu überprüfenden Zellen eine Korrespondenz ersichtlich. Jegliche ursprünglich als erfüllt angenommenen Feststellungen, wie etwa der richtigen Zuordnung von Eltern- und Kindknoten, wurden in allen vier Szenarien erfolgreich bestätigt.

Die Frage nach der korrekten Darstellung von Hierarchien im Graphen lässt sich mittels der ersten fünf Metriken als erfüllt ansehen. Unabhängig von der Größe der Graphen und den damit einhergehenden Mengen an Eltern- bzw. Kindknoten sind alle erwarteten Hierarchien abgebildet worden. Die Korrektheit dieser wurde durch die richtige Zuordnung zwischen den jeweiligen Knoten sichergestellt. Folglich ist keinem Elternknoten ein unerwarteter Kindknoten zugeordnet worden, wodurch lediglich die vom Nutzer spezifizierten hierarchischen Beziehungen im Graph enthalten sind. Die als erfüllt angesehenen Eigenschaften hinsichtlich der Ein- und Ausblendung von Kanten gewährleisten, dass lediglich die für die sichtbaren Knoten relevanten Kanten dargestellt werden. Dementsprechend verfügte ein Graph zu keinem Zeitpunkt über unnötige, nicht zuordenbare oder fehlende Relationen.

Die richtige Verhaltensweise eines Graphen bei einer Verwendung der Aggregations-Tupel aus Kapitel 3.1.4 wurde mittels der zweiten Frage analysiert. Anhand der ersten vier quantitativen Metriken ist die resultierende Visualisierung der bis zu einem bestimmten Grad entfalteten Graphen untersucht worden. Mittels der beiden Zeilen, in denen szenarienübergreifend ein Wert von null erwartet worden ist, wurde festgestellt, dass kein Graph in dem jeweiligen Zustand über unerwartete (nicht-) aggregierten Kanten verfügt. Auf diese Weise wurde beispielsweise bei einem vollständig entfalteten Graphen

Tabelle 7.6. Ermittelte Werte zu den Fragen und Metriken bezüglich des Ziels E3

	if-then-else		loops		nested		method-call	
	E	G	E	G	E	G	E	G
Korrekte Darstellung von Hierarchien?								
Anzahl der Elternknoten	13	13	12	12	15	15	39	39
Anzahl der Kindknoten	51	51	48	48	68	68	164	164
Korrekte Zuordnung zwischen Eltern- und Kindknoten	(✓)	✓	(✓)	✓	(✓)	✓	(✓)	✓
Einblendung von Kanten sichtbarer Kindknoten	(✓)	✓	(✓)	✓	(✓)	✓	(✓)	✓
Ausblendung von Kanten nicht sichtbarer Kindknoten	(✓)	✓	(✓)	✓	(✓)	✓	(✓)	✓
Korrektes Verhalten bei Kanten-Aggregationen?								
Anzahl der sichtbaren aggregierten Kanten bei vollständig entfaltetem Graphen	0	0	0	0	0	0	0	0
Anzahl der sichtbaren nicht-aggregierten Kanten bei vollständig entfaltetem Graphen	12	12	7	7	10	10	44	44
Anzahl der sichtbaren aggregierten Kanten bei entsprechend eingekapseltem Graphen	12	12	7	7	10	10	44	44
Anzahl der sichtbaren nicht-aggregierten Kanten bei entsprechend eingekapseltem Graphen	0	0	0	0	0	0	0	0
Einblendung von aggregierten Kanten bei Einkapselung eines Elternknotens	(✓)	✓	(✓)	✓	(✓)	✓	(✓)	✓
Ausblendung von nicht-aggregierten Kanten bei Einkapselung eines Elternknotens	(✓)	✓	(✓)	✓	(✓)	✓	(✓)	✓
Ausblendung von aggregierten Kanten bei Entfaltung eines Elternknotens	(✓)	✓	(✓)	✓	(✓)	✓	(✓)	✓
Einblendung von nicht-aggregierten Kanten bei Entfaltung eines Elternknotens	(✓)	✓	(✓)	✓	(✓)	✓	(✓)	✓

sichergestellt, dass keine aggregierten Kanten, wie die in Abschnitt 7.1.2.2 erwähnten *AGGREGATED_CALLS*-Kanten, dargestellt werden.

Die anderen beiden Metriken, bei denen in allen Szenarien ein von null verschiedener Wert erwartet worden ist, stellen die Gegenseite der zuvor erläuterten Zeilen dar. Anhand der in einem Szenario übereinstimmenden Werte ist zu sehen, dass bei der Aggregation von Kanten, genau die Richtige Anzahl an Kanten verborgen wird. Demzufolge werden sowohl beim entfalteten als auch teilweise eingekapselten Graphen sämtliche Relationen durch eine der beiden Kanten ausgedrückt. Dies ist ebenfalls anhand der zeilenübergreifenden Korrespondenz innerhalb eines Szenarios zu erkennen. Jede bei einem entfalteten Graphen dargestellte nicht-aggregierte Kante wird beim entsprechend eingekapselten Graphen durch die zugehörige aggregierte Kante repräsentiert.

Die Eigenschaften bezüglich des Ein- und Ausblendens von Kanten wurde anhand der letzten vier Metriken überprüft. Während die vorherigen quantitativen Werten lediglich den Graph in einem bestimmten Zustand repräsentieren, wurde bei diesen visuellen Überprüfungen das Verhalten während einer Aktion untersucht. Das Ergebnis der in allen vier Szenarien durchgeführten Aktionen, welche an einer Vielzahl von Knoten getestet worden sind, bestätigt den bereits erhobenen Verdacht. In allen vier Graphen wurde bei der Verwendung von Aggregations-Tupeln stets genau eine der beiden Kanten des Tupels eingeblendet bzw. verborgen.

Zusammenfassend ist zu sagen, dass die implementierten hierarchischen Konzepte erwartungsgemäß funktionieren. Sowohl das Visualisieren von Hierarchien als auch das Aggregieren von Kanten wurde erfolgreich mittels der beiden Fragen aus der Tabelle 7.6 evaluiert. Neben dem Darstellen von statischen Hierarchien wurde ebenfalls das Ein- und Ausblenden von (nicht) relevanten Kanten überprüft. Obwohl diese Evaluation lediglich anhand von SDG durchgeführt worden ist, eignet sich der Grapheditor wegen der universell einsetzbaren Typisierung gleichermaßen für andere Graphmodelle. Diese hätten jedoch manuell erzeugt werden müssen, da sie, im Gegensatz zu den verwendeten Szenarien, nicht mittels des Tools *NeoSuit* automatisch generiert werden konnten.

7.3. Unsicherheiten bezüglich der Validität

Obwohl der Grapheditor hinsichtlich der drei Experimente aus Kapitel 7.1.1 erfolgreich evaluiert worden ist, gibt es dennoch Faktoren, welche die Validität beeinflussen. In den folgenden beiden Abschnitten werden die in zwei Bereiche unterteilten Aspekte bezüglich der internen und externen Validität genauer beschrieben.

7.3.1. Interne Validität

Ein auffälliger Faktor im Hinblick auf die interne Validität stellen die manuell und visuell zu überprüfenden Metriken in allen drei Experimenten dar. Diese repräsentieren einerseits, wegen ihrer schlichten Beantwortung (✓ bzw. ✗), keinen allzu hohen Detailgrad und

sind andererseits fehleranfälliger gegenüber maschinell erzeugter Werte. Allerdings ist aufgrund der Fülle an gewählten Metriken eine fehlerbehaftete Evaluation der Graphen nahezu auszuschließen.

Des Weiteren gelten die ermittelten Ergebnisse nur in Zusammenhang mit der für die Evaluation durchgeführten Typisierung der Kanten. Zuteilungen der in Kapitel 3.1 erläuterten Kantentypen, welche hiervon verschieden sind, resultieren möglicherweise in einer abweichenden hierarchischen Ansicht der dargestellten Graphen. Bei den ermittelten Werten würden sich demzufolge Änderungen ergeben, sodass die Erwartungen nicht mehr mit den Messungen korrespondieren. Dementsprechend gilt die als erfolgreich evaluierte Funktionalität des Grapheditors stets nur in Verbindung mit einer korrekt durchgeführten Kantentypisierung eines Graphen. Die Voraussetzung hierfür ist die Verfügbarkeit von entsprechenden Schlüssel-Wert-Paaren an den jeweiligen Kanten.

Der letzte Faktor betrifft die Generierung der SDG, welche mittels *NeoSuit* generiert worden sind. Dieses unterstützt in der momentan aktuellen Version kein Java 8, wodurch die evaluierten Graphen jeweils auf Basis eines in Java 7 geschrieben Beispielprogramms erzeugt wurden. In Abhängigkeit von der Einbindung von Java 8 in eine spätere Version ist es demnach erdenklich, dass die generierten Graphen von den in dieser Arbeit verwendeten abweichen. In diesem Fall muss die durchgeführte Kantentypisierung entsprechend angepasst werden.

7.3.2. Externe Validität

In Bezug auf die externe Validität gehen die größten Unsicherheiten von der Anzahl an gewählten Szenarien aus, welche durch den für diese Thesis vorgesehenen Zeitraum begrenzt war. Wegen der unendlichen Menge an möglichen Datenbankzuständen und Graphkonstruktionen konnten weder beim Im- und Export noch beim Evaluieren der hierarchischen Konzepte eine allumfassende Korrektheit des Grapheditors sichergestellt werden. Folglich ist es trotz der als korrekt erwiesenen Funktionalitäten erdenklich, dass im Umgang mit anderen Datenbanken oder Graphen Fehler auftreten.

Zusätzlich waren die Szenarien sämtlicher Experimente aufgrund der manuellen und visuellen Überprüfung in ihrer Größe beschränkt. Bei zu vielen Knoten und Kanten fielen die resultierenden Graphen zu komplex aus, sodass die Korrektheit der ermittelten Messwerte nicht mehr garantiert werden konnte. Jedoch wird sowohl durch die Verwendung der vier grundlegend von einander verschiedenen Datenbankmodelle als auch durch die für unterschiedliche Fokusse generierten SDG ein gewisses Maß an externer Validität gewährleistet.

Eine Besonderheit bei der Evaluation des *Neo4j-Reader* und *-Writers* besteht in der Anbindung über eine URL. Entgegen der Möglichkeit die Datenbank und den Grapheditor auf getrennten Systemen operieren zu lassen, wurden während der Evaluation beide Anwendungen auf demselben Computer betrieben. Das Im- und Exportieren von Graphen ist folglich nur über die systemeigene URL *http://localhost:7474* und nicht systemübergreifend getestet worden.

Kapitel 8

Verwandte Arbeiten

Dieses Kapitel umfasst die mit dieser Masterthesis verwandten Arbeiten. In Abschnitt 8.1 sind neben dem zugrundeliegenden Ansatz von C. Wulf [Wulf 2014] ebenfalls die darauf aufbauenden Arbeiten aufgeführt. Anschließend werden in Kapitel 8.2 weitere Grapheditoren vorgestellt und zu dem aus dieser Arbeit resultierenden abgegrenzt.

8.1. Arbeiten in Bezug auf den semi-automatischen Ansatz von C. Wulf

Der Ansatz von C. Wulf bildet die grundlegende Motivation sowohl dieser Masterthesis als auch der im folgenden Abschnitt erwähnten Arbeiten. Das Ziel dieses Ansatzes ist die semi-automatische Parallelisierung eines sequentiellen Programmcodes. Als Basis dieser Transformation dient der ursprünglich generierte SDG, welcher mittels *Candidate-* und *Parallelization-Pattern* modifiziert wird (siehe Abschnitt 1.2). Für eine möglichst performante Berechnung des modifizierten Graphen wird im Back-End eine Neo4j-Graphdatenbank verwendet.

Wie bereits in Kapitel 2.7 erwähnt wurde, stellt der Editor aus den Arbeiten von L. Blümke und Y. Benekov die Grundlage dieser Thesis dar. Die Erstere der beiden Arbeiten [Blümke 2015] befasst sich mit der Erstellung von SDG, welche dem in Abschnitt 2.1.3 vorgestellten Modell entsprechen. Neben ausführbaren Aktionen zum Bearbeiten von Graphelementen bzw. deren Attributen wurde ebenfalls ein Menü zum Einfärben eines Graphen implementiert. Im Vergleich dazu beschäftigt sich die Arbeit von Y. Benekov [Benekov 2015] mit dem Verwalten der Graphen und den aufgezeichneten Bearbeitungshistorien. Ein erweiterndes *Recorder*-Plugin ermöglicht das Protokollieren und Speichern einzelner Bearbeitungsschritte eines Graphen in chronologischer Reihenfolge.

Eine weitere verwandte Arbeit stellt die Masterthesis von E. Koppenhagen [Koppenhagen 2016] dar. Entgegen der bisherigen Verwendung zur Visualisierung von vollständigen SDG, wird der Grapheditor hierbei für die Erstellung der zuvor erläuterten Pattern verwendet. Über die Transformation eines visualisierten Graphen in eine Cypher-Anfrage, wird das *Candidate-Pattern* automatisch generiert. Dieses ermöglicht anschließend das Ermitteln des ursprünglichen (Teil-)Graphen in einem beliebigen SDG. Die Basis der generierten *Parallelization-Pattern*, welche einen Graphen an gewissen Stellen modifizieren, sind die mittels des Rekorders aufgezeichneten Bearbeitungshistorien. Um in der Lage zu sein,

einen Graphen auf ungültige Knoten, Kanten oder Teilgraphen zu untersuchen, wurde zusätzlich ein optionaler Mechanismus zum Ausschließen von bestimmten Graphelementen implementiert.

8.2. Weitere Grapheditoren

Zur Visualisierung der in einer Neo4j-Datenbank gespeicherten Daten lassen sich eine Vielzahl an Editoren finden. Ein Vorteil des bereits in Kapitel 2.4.1 erwähnten Neo4j-Browsers ist die feste Integration und der damit verbundene versionsübergreifende Support. Das Traversieren durch einen Graphen wird über die Eingabe von Cypher-Anfragen und das farbliche Gestalten der Knoten bzw. Kanten ermöglicht. Aufgrund der fehlenden Layout-Algorithmen fallen die Graphen bei größeren Datenbanken jedoch meist unübersichtlich aus. Das manuelle Anordnen der Knoten behebt dieses Problem wegen der nicht persistenten Speicherung nur bedingt. Des weiteren werden wegen des flachen Modells einer Neo4j-Graphdatenbank keine hierarchischen Beziehungen (siehe Kapitel 1.3.1) zwischen den Knoten unterstützt.

Eine Alternative zur Visualisierung von Graphen stellt die frei verfügbare Software *Cytoscape*[1] dar. Obwohl die Java-Anwendung ursprünglich für den Bereich von molekularen Netzwerken vorgesehen war, lässt sie sich aufgrund der erweiterbaren Plugin-Struktur auch anderweitig nutzen. Über das Plugin *CyNeo4j*[2] wird der Zugriff auf eine Neo4j-Graphdatenbank mittels einer URL ermöglicht. Neben dem Im- und Export von Graphen, lassen sich diese ebenfalls mit den zur Verfügung gestellten Layoutalgorithmen neu anordnen. Ähnlich wie in dem Editor dieser Arbeit werden die Attribute eines Knoten bzw. einer Kante in einem separaten Panel dargestellt. Mit dem integrierten Tool *NetworkAnalyzer* lassen sich die visualisierten Graphen analysieren und zusätzliche Informationen bezüglich verschiedener Graphmetriken berechnen. Beispiele hierfür ist die Anzahl an eingehenden Kanten sämtlicher Knoten oder das Verhältnis zwischen möglichen und vorhandenen Kanten eines Graphen (*Density*-Wert). Allerdings werden infolge des direkten Zugriffs auf das Neo4j-Datenmodell wie bei dem integrierten Neo4j-Browser keine Hierarchien unterstützt.

Neben *Cytoscape* gibt es viele weitere Grapheditoren wie *Neoclipse*[3] oder die ebenfalls mit Plugins erweiterbare Platform *Gephi*[4], welche das Im- und Exportieren von Neo4j-Datenbanken ermöglichen. Obwohl meist auch das Anordnen der Graphelemente über bereitgestellte Layoutalgorithmen verfügbar ist, fehlt es den Anwendungen stets an einem Feature zum Darstellen von Hierarchien. Hinzu kommt der ausgebliebene Support aktueller Neo4j-Versionen, sodass ein Import der mittels *NeoSuit* generierten Graphen nicht durchführbar ist.

[1] http://www.cytoscape.org/ Letzter Zugriff: 29. Oktober 2016
[2] https://cyneo4j.wordpress.com/ Letzter Zugriff: 29. Oktober 2016
[3] https://github.com/neo4j-contrib/neoclipse Letzter Zugriff: 29. Oktober 2016
[4] https://gephi.org/ Letzer Zugriff: 29. Oktober 2016

Allgemeine Grapheditoren wie beispielsweise *yED*[5] bieten zwar vielseitige Optionen zum Zeichnen von Graphen, welche jedoch lediglich als Bilddateien im- und exportiert werden können. Trotz der Darstellung von Eltern- und Kindknoten ist eine Verwendung aufgrund der fehlenden Anbindung einer Neo4j-Datenbank nicht möglich. Der Inhalt einer Datenbank müsste zunächst als Bild exportiert werden, bevor es im weiteren Verlauf mit *yED* analysiert oder visualisiert wird. Nichtsdestotrotz würden Hierarchien wegen der flachen Neo4j-Graphvorlage lediglich durch ein manuelles Einfügen vorhanden sein.

Abschließend lässt sich sagen, dass keiner der hier vorgestellten Grapheditoren sämtliche Ziele aus Kapitel 1.3 implementiert. Die Kombination aus einer hierarchischen Visualisierung eines Graphen und der Anbindung an eine Neo4j-Graphdatenbank grenzt den aus dieser Arbeit resultierenden Editor von den bisherigen ab. Die Einbindung des *KLighD-Frameworks* zum Anordnen der Graphelemente stellt ebenfalls eine Besonderheit dar.

[5] https://www.yworks.com/products/yed/ Letzter Zugriff: 29. Oktober 2016

Kapitel 9

Fazit und Ausblick

9.1. Fazit

Der aus dieser Arbeit resultierende Grapheditor stellt eine Erweiterung des in Kapitel 2.7 beschriebenen Grapheditors dar. Durch die Realisierung als *Eclipse*-Plugin ist der Editor ein universelles Werkzeug, welches für viele Problemstellungen einsetzbar ist. Obwohl er in erster Linie zum grafischen Repräsentieren der SDG aus dem Ansatz von C. Wulf [Wulf 2014] entwickelt wurde, stellen diese lediglich eine beispielhafte Menge der visualisierbaren Graphen dar. Grund hierfür ist das in Abschnitt 3.3 erweiterte EMF-Modell eines Propertygraphen, welches dem Nutzer einen flexiblen Umgang mit dem Editor ermöglicht.

Sämtliche der in Kapitel 1.3 aufgelisteten Ziele, wurden erfolgreich implementiert, wodurch sowohl der Im- und Export als auch weitere Optionen zum Bearbeiten eines Graphen zur Verfügung stehen. In Ergänzung zu der flachen Darstellung eines Graphen wird dem Nutzer mit der hierarchischen Ansicht ein weiteres Mittel zur Analyse von relevanten Teilgraphen bereitgestellt. Obwohl die in Kapitel 3 erläuterte Typisierung von Kanten anfänglich nicht intuitiv wirkt, lassen sich hiermit jegliche erdenkliche hierarchische Beziehungen als Elternknoten-Kindknoten-Beziehung ausdrücken. Dies wurde in Kapitel 7 erfolgreich evaluiert.

Der Zugriff auf die Einstellungen eines zu visualisierenden Graphen ist auf unterschiedliche Dialoge aufgeteilt worden. Neben dem bereits vorhandenen Menü zur Einfärbung eines Graphen, wurde der Editor um drei weitere Dialogfenster erweitert: Typisierung von Kanten, Aggregation von Kanten und periodisches Speichern. Jedes der vier Fenster ist über einen separaten, in die Menüleiste eingebundenen Knopf aufrufbar.

Die Einbindung des *KIELER*-Unterprojekts *KlighD* gewährleistet das performante Zeichnen der Graphen und die Anordnung der einzelnen Graphobjekte. Über die bereitgestellten Graphalgorithmen wird es dem Nutzer ermöglicht die Visualisierung stets neu zu generieren, wodurch die Überschneidung von Elementen im Vergleich zum Neo4j-Editor reduziert wird.

9.2. Ausblick

Im Rahmen von zukünftigen Arbeiten ist es denkbar den entstandenen Grapheditor in Hinblick auf die Bedienbarkeit zu optimieren. Beim Erzeugen eines Kindknotens wäre es möglich, dem Nutzer eine Menge von bereits mit Informationen angereicherten Knoten vorzuschlagen, sodass beispielsweise nicht alle Schlüssel-Wert-Paare manuell hinzugefügt werden müssen. Die Basis dieser Menge ließe sich mittels der schon im Elternknoten vorhandenen Kindknoten oder der ausgewählten hierarchischen Kante schaffen.

Durch eine automatische Analyse könnte dem Nutzer die Typisierung der Kanten während des Importieren eines Graphen erleichtert werden. Sowohl sich wiederholende Graphstrukturen als auch typische Namen für hierarchische Beziehungen zwischen den Knoten wären Indizien für die Typisierung als einer der in Kapitel 3.1 eingeführten Kanten. Zusätzlich ließe sich die Analyse auf die Ermittlung von Aggregations-Tupeln ausweiten.

Ein weiterer Punkt besteht in der Berücksichtigung der Neo4j-charakteristischen Knotenlabels und Kantentypen. Um die entsprechenden Informationen ebenfalls zu importieren, müsste das Propertygraph-Modell erweitert werden. Obwohl die zusätzlichen Informationen auf die Visualisierung des Graphen keine Auswirkungen haben, würden die Knoten und Kanten eines exportierten Graphen direkt mit den beim Import vorhandenen Werten versehen werden. Demnach müssten die Objekte des zu importierenden Graphen über keine redundanten Felder verfügen, welche den Wert des Labels bzw. Typs beinhalten. Die Typisierung der Kanten für die hierarchische Ansicht könnte auf Basis der neu importierten Informationen stattfinden, falls neben den verbindlichen Kantentypen ebenso die Knoten mit den optionalen Labels markiert worden sind.

Für den Export einer Datenbank ist es vorstellbar neben dem eigentlichen Graphen auch die benutzerdefinierten Metainformationen in die Neo4j-Datenbank auszulagern. Auf diese Weise würde die hierarchische Darstellung und Einfärbung eines Propertygraphen nach einem Export-Import-Vorgang bestehen bleiben. Ein importierter Graph müsste demnach beispielsweise nicht erneut für eine hierarchische Ansicht angepasst werden.

Literaturverzeichnis

[Benekov 2015] Y. Benekov. Entwicklung eines Eclipse-Plugins zur Aufzeichnung von Benutzerinteraktionen mit Eigenschaftsgraphen. Bachelorarbeit. Institut für Informatik, Oktober 2015. (Siehe Seiten 3, 39, 65 und 91)

[Blümke 2015] L. E. Blümke. Konzeption und Implementierung eines Eclipse-Plugins zur Erstellung von graphbasierten Quellcodemustern. Bachelorarbeit. Institut für Informatik, Oktober 2015. (Siehe Seiten 3, 27, 29, 30, 39, 65 und 91)

[Diestel 1996] R. Diestel. Graphentheorie. Springer-Lehrbuch. Springer, 1996. (Siehe Seite 9)

[Ebert 2011] R. Ebert. Eclipse RCP - Entwicklung von Desktop-Anwendungen mit der Eclipse Rich Client Platform. 2011. (Siehe Seite 15)

[Edlich u. a. 2011] S. Edlich, A. Friedland, J. Hampe, B. Brauer und M. Brückner. NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken. München: Carl Hanser Verlag GmbH & CO. KG, Sep. 2011. (Siehe Seiten 20, 21)

[Ferrante u. a. 1987] J. Ferrante, K. J. Ottenstein und J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* (Juli 1987), Seiten 319–349. (Siehe Seite 10)

[Horwitz u. a. 1988] S. Horwitz, T. Reps und D. Binkley. Interprocedural Slicing Using Dependence Graphs. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. Atlanta, Georgia, USA: ACM, 1988, Seiten 35–46. (Siehe Seite 10)

[Hunger 2014] M. Hunger. Neo4j 2.0: Eine Graphdatenbank für alle. Frankfurt am Main: Entwickler.press, 2014. (Siehe Seite 21)

[Koppenhagen 2016] E. Koppenhagen. GUI-based Automated Generation of Neo4j Cypher Queries for Candidate Patterns and Parallelization Patterns. Masterarbeit. Institut für Informatik, Oktober 2016. (Siehe Seite 91)

[Ottenstein und Ottenstein 1984] K. J. Ottenstein und L. M. Ottenstein. The program dependence graph in a software development environment. In: *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. New York, USA: ACM, Apr. 1984, Seiten 177–184. (Siehe Seite 10)

[Rodriguez und Neubauer 2010] M. A. Rodriguez und P. Neubauer. Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology* 36.6 (2010), Seiten 35–41. (Siehe Seiten 9, 10)

[Schneider u. a. 2012] C. Schneider, M. Spönemann und R. von Hanxleden. Transient View Generation in Eclipse. *Technischer Bericht* (2012). (Siehe Seite 23)

[Schneider u. a. 2013] C. Schneider, M. Spönemann und R. von Hanxleden. Just model! - Putting automatic synthesis of node-link-diagrams into practice. In: *Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing*. Sep. 2013, Seiten 75–82. (Siehe Seiten 24, 25)

[Steinberg u. a. 2009] D. Steinberg, F. Budinsky, M. Paternostro und E. Merks. EMF: Eclipse Modeling Framework 2.0. 2nd. Addison-Wesley Professional, 2009. (Siehe Seiten 17, 18)

[Steppan 2015] B. Steppan. Eclipse Rich Clients und Plug-ins: Modulare Desktop-Anwendungen mit Java entwickeln. Carl Hanser Verlag GmbH & Company KG, 2015. (Siehe Seiten 14 und 16)

[Van Solingen u. a. 2002] R. Van Solingen, V. Basili, G. Caldiera und H. D. Rombach. Goal question metric (gqm) approach. *Encyclopedia of software engineering* (2002). (Siehe Seiten 6 und 69)

[Walkinshaw u. a. 2003] N. Walkinshaw, M. Roper und M. Wood. The Java system dependence graph. In: *Proceedings of the third IEEE International Workshop on Source Code Analysis and Manipulation*. Sep. 2003, Seiten 55–64. (Siehe Seiten 11–13)

[Wulf 2014] C. Wulf. Pattern-based Detection and Utilization of Potential Parallelism in Software Systems. In: *Proceeding of the Software Engineering 2014 Conference*. Feb. 2014. (Siehe Seiten 1, 2, 91 und 95)