Konzeption und Implementierung eines Eclipse-Plugins zur Erstellung von graphbasierten Quellcodemustern

Bachelorarbeit

Lars Erik Blümke

25. September 2015

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

Im Rahmen der semi-automatischen Parallelisierung sequentieller Programme kommt Graphen eine wichtige Rolle zu. Die Repräsentation eines Programms als Graph ermöglicht es, parallelisierbare Passagen nicht im Quellcode, sondern am Graphen zu erkennen. Dazu wird der Graph mit gegebenen Graphmustern verglichen, die potentiell parallelisierbare Quellcodemuster repräsentieren. Als parallelisierbar erkannte Teilgraphen können dann so modifiziert werden, dass aus ihnen paralleler Quellcode generiert werden kann.
Im Rahmen des Bachelorprojekts *„Konzeption und Implementierung eines Eclipse-Plugins zur Bearbeitung von Graphen“* befasst sich diese Arbeit mit der Entwicklung eines Eclipse-Plugins, welches Möglichkeiten bietet, derartige Graphmuster grafisch zu erstellen und zu bearbeiten. Dabei liegt der Schwerpunkt auf der Bearbeitung von Systemabhängigkeitsgraphen. Das heißt, es werden Funktionen entwickelt, um diesen spezifischen Typ von Graphen gezielt und effektiv bearbeiten zu können. Es werden spezifische Knoten- und Kantentypen für unterschiedliche Programmelemente entwickelt, um den Quellcode semantisch korrekt als Graphen abbilden zu können. Außerdem wird das Speichern zusätzlicher Daten in Form von Wertepaaren im Graphen ermöglicht, um auch Laufzeitinformationen über das repräsentierte Programm festhalten zu können.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Kontext . . . 1
- 1.3 Ziele . . . 2
  - 1.3.1 Z1: Grafischer Editor . . . 3
  - 1.3.2 Z2: Schlüssel-Wert-Tabelle . . . 3
  - 1.3.3 Z3: Vordefinierte Elemente für Systemabhängigkeitsgraphen . . . 4
  - 1.3.4 Z4: Attributabhängige Farbauswahl . . . 4
- 1.4 Aufbau . . . 4

2 Grundlagen und Technologien **5**
- 2.1 Graphen . . . 5
  - 2.1.1 Eigenschaftsgraph . . . 5
  - 2.1.2 Systemabhängigkeitsgraph . . . 6
- 2.2 Die Entwicklungsumgebung Eclipse . . . 8
  - 2.2.1 Die Rich Client Platform . . . 8
- 2.3 Das Eclipse Modeling Framework . . . 10
  - 2.3.1 Das Ecore (Meta-) Modell . . . 10
  - 2.3.2 Code Generierung . . . 11
- 2.4 Das KIELER Projekt und KLighD . . . 12
  - 2.4.1 Basismodell und Grafikmodell . . . 12
  - 2.4.2 Die Synthesis . . . 13

3 Übersicht über die Komponenten des Grapheditors **15**

4 Architektur des Basismodells **17**
- 4.1 Anforderungen an das Basismodell . . . 17
- 4.2 Umsetzung der Anforderungen . . . 17
  - 4.2.1 Elemente des Basismodells . . . 17
  - 4.2.2 Der EMF Tree Editor für Propertygraphen . . . 19

5 Transformation vom Basismodell zum Grafikmodell **21**
- 5.1 Allgemeiner Transformationsprozess in KLighD . . . 21
- 5.2 Umsetzung der Synthesis . . . 22
  - 5.2.1 Knoten und Kanten . . . 22
  - 5.2.2 Linienform und Linienstärke . . . 23

5.2.3 Farben 24

6 Interaktionsmöglichkeiten mit dem Graphen 27
6.1 Editierbarkeit und Vordefinierte Graphelemente 27
6.1.1 Darstellung des Graphen in der KLighD Diagram View 27
6.1.2 Editierbarkeit des Graphen 28
6.1.3 Vordefinierte Graphelemente 28
6.1.4 Unterschiedliche Elementtypen festlegen 29
6.1.5 Umsetzung der Editierbarkeit in einem Kontextmenü 30
6.2 Key/Value View 33
6.2.1 Umsetzung der Schlüssel-Wert-Tabelle 33
6.2.2 Die Toolbar 34
6.2.3 Datenaustausch mit dem Graphen 35
6.3 Display Properties View 36
6.3.1 Angezeigte Attribute 37
6.3.2 Abgrenzung zur Key/Value View 37
6.4 Attributabhängige Farbauswahl 38
6.4.1 Das Dialogfenster 38
6.4.2 Datenaustausch mit dem Graphen 39

7 Evaluierung 41
7.1 Anwendungsbeispiel 41
7.2 Bewertung ausgewählter Komponenten und mögliche Weiterentwicklungen 47
7.2.1 Graphmodell 47
7.2.2 EMF Tree Editor 47
7.2.3 Key/Value und Display Properties View 47
7.2.4 Attributabhängige Farbauswahl 48

8 Fazit und Ausblick 49
8.1 Fazit 49
8.2 Ausblick 49

Bibliografie 51

Kapitel 1

Einleitung

1.1. Motivation

Durch das Parallelisieren von Programmen kann deren Performance zum Teil deutlich verbessert werden. Zum einen können auftretende Latenzzeiten durch die Auslagerung in einen separaten Thread verborgen werden und zum anderen kann die vorhandene Hardware moderner Mehrkernarchitekturen besser genutzt werden.
Da das manuelle Parallelisieren von sequentiellen Programmen komplex und fehleranfällig ist, erfordert es oftmals Hilfe von Experten, um die dabei auftretenden Probleme zu lösen. Die in dieser Arbeit entwickelte Software ist eine der Grundlagen eines alternativen Parallelisierungsansatzes, der den Entwickler entlastet, indem weitestgehend nicht auf dem Quellcode gearbeitet wird, sondern auf dessen Repräsentation als Graph. Durch die Erkennung parallelisierbarer Muster am Graphen und die semi-automatische Transformation in tatsächlich parallele Muster kann Fehlern vorgebeugt und Zeit bei der Parallelisierung eingespart werden.

1.2. Kontext

Der Kontext dieser Arbeit ist ein semi-automatischer Parallelisierungsansatz von Wulf [2014], um Softwareentwicklern die Parallelisierung bestehender Softwaresysteme zu erleichtern. Der Ansatz basiert auf der Erkennung von potentiell parallelisierbaren Mustern am Systemabhängigkeitsgraphen des Softwaresystems und der Transformation der erkannten Muster zu parallelisierbaren Mustern. Abbildung 1.1 bietet einen Überblick über den kompletten Parallelisierungsprozess.

In den ersten drei Schritten S1-S3 wird für ein sequentielles Programm der zugehörige, das Programm repräsentierende Systemabhängigkeitsgraph (SDG) generiert. Dieser basiert auf statischen Informationen aus dem Quellcode (S1), wird aber auch mit dynamischen Informationen aus S2, wie der Laufzeit einzelner Methoden ergänzt. Der in S4 auf Grundlage des SDG konstruierte Parallelisierungsplan besteht aus einer geordneten Liste von Quellcodeabschnitten, die das größte Parallelisierungspotential bieten oder bestimmte Kriterien erfüllen. Angenommen, Methoden mit längerer Laufzeit bieten ein höheres Potenzial, so könnten im Parallelisierungsplan alle Methodendeklarationen nach ihrer Laufzeit sortiert aufgelistet werden.

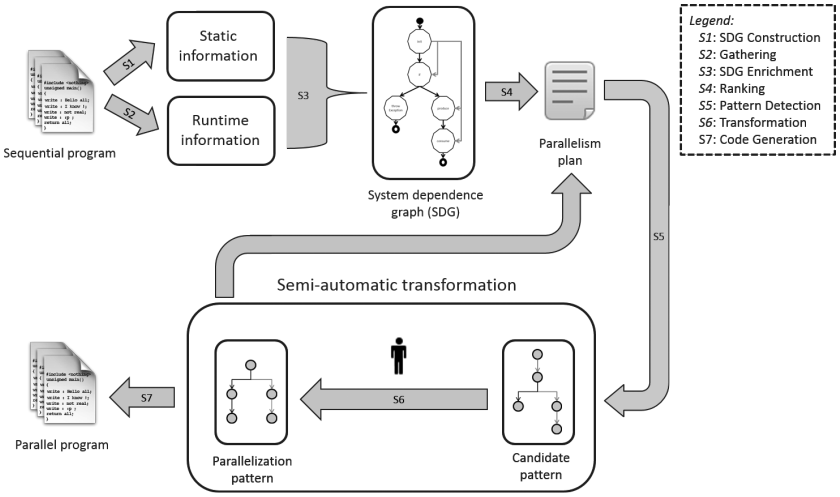

Abbildung 1.1. Übersicht des semi-automatischen Parallelisierungsansatzes von Wulf [2014]

Auf jedem der im Plan definierten Abschnitte werden dann die Schritte S5 und S6 ausgeführt. Dazu wird eine erweiterbare Menge von Candidate- und zugehörigen Parallelizationpatterns angeboten. Ein Candidatepattern ist ein Graphmuster, das einen potentiell parallelisierbaren Quellcodeabschnitt darstellt und zu dem jeweils ein Parallelizationpattern gehört, das einen semantisch gleichen, aber parallelen Quellcodeabschnitt darstellt. In S5 werden in den im Plan festgelegten Bereichen des SDG zu den gegebenen Candidatepatterns passende Muster gesucht und erkannt. S6 stellt dann die Transformation in das zugehörige Parallelizationpattern dar. S5 und S6 können automatisch ausgeführt werden, der Softwareentwickler hat aber vor S6 die Möglichkeit, die vorgeschlagene parallele Version zu prüfen und anzupassen, ehe sie in den SDG übernommen wird. In S7 wird schließlich aus dem parallelisierten SDG paralleler Programmcode generiert. Dieser Schritt kann nach jeder Ausführung von S5 und S6 durchgeführt werden.

1.3. Ziele

Das Eclipse Plugin, das in dieser Arbeit entwickelt wird, bietet Funktionen um mit Systemabhängigkeitsgraphen zu arbeiten. Damit soll es im Kontext des beschriebenen Parallelisierungsprozesses zunächst an den Stellen genutzt werden, wo entsprechende Candidatepatterns erstellt werden müssen.
Zudem wird in einer parallelen Arbeit [Benekov 2015] unter anderem eine Aufnahmefunktion entwickelt, welche es ermöglicht, die manuelle Transformation eines Candidatepattern in ein Parallelizationpattern aufzuzeichnen. Durch die Anwendung der aufgezeichneten Schritte kann die Umwandlung in parallelen Programmcode automatisiert werden. Im

Folgenden nennen und beschreiben wir die einzelnen Ziele dieser Arbeit.

1.3.1. Z1: Grafischer Editor

Es wird gemeinschaftlich im Rahmen des Bachelorprojekts *„Konzeption und Implementierung eines Eclipse-Plugins zur Bearbeitung von Graphen“* ein grafischer Editor entwickelt, der die Erstellung von Graphen mit der Maus ermöglicht. Der Graph wird auf einer Arbeitsfläche in Eclipse angezeigt wie in Abbildung 1.2. Es wird möglich sein, den Graphen mit der Maus zu bearbeiten. Um den Graphen zu bearbeiten, muss ein Knoten oder eine freie Fläche mit der rechten Maustaste angeklickt werden. Dadurch wird ein Kontextmenü geöffnet, welches Möglichkeiten zum Einfügen von Knoten und Kanten bietet. Knoten können zusammen mit einer Kante als Nachfolger hinter einem selektierten Knoten oder zunächst allein auf einer freien Fläche eingefügt werden. Zwei selektierte Knoten können mit einer Kante verbunden werden und ein selektiertes Element kann entfernt werden.

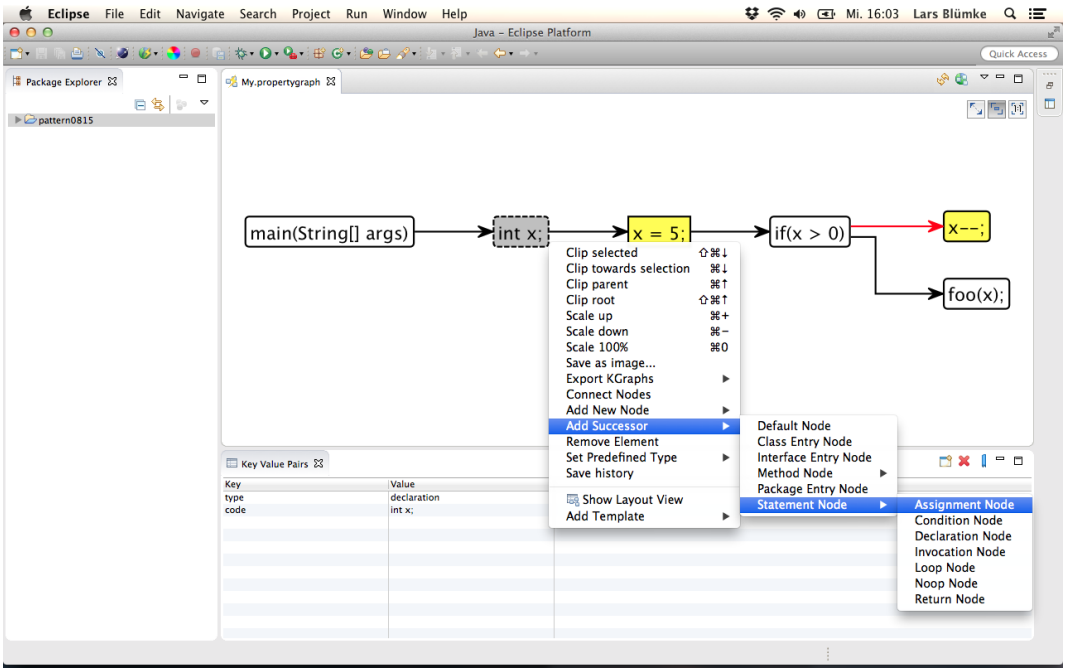

Abbildung 1.2. Grafischer Editor und Schlüssel-Wert-Tabelle

1.3.2. Z2: Schlüssel-Wert-Tabelle

Um zusätzliche Informationen im Graphen zu speichern, können Knoten und Kanten individuelle Schlüssel-Wert-Paare hinzugefügt werden, die beim Selektieren des jewei-

ligen Graphelements in einer Tabelle angezeigt werden. Über die Tabelle können den Graphelementen neue Schlüssel-Wert-Paare hinzugefügt werden. Vorhandene Paare können verändert und gelöscht werden. Ein einfaches Beispiel für ein Schlüssel-Wert-Paar um Knoten und Kanten zu unterscheiden, wäre der Schlüssel *type* und die Werte *node* beziehungsweise *edge* für Knoten beziehungsweise Kanten.

1.3.3. Z3: Vordefinierte Elemente für Systemabhängigkeitsgraphen

Das beschriebene Kontextmenü wird um speziell auf Systemabhängigkeitsgraphen bezogene Elemente erweitert, sodass direkt die in 2.1.2 beschriebenen, spezifischeren Kanten- oder Knotentypen eines SDG in den Graphen eingefügt werden können. Dabei enthalten diese dann insbesondere schon entsprechende Werte in der Schlüssel-Wert-Tabelle, wenn sie selektiert werden. Ein Knoten, der eine Zuweisung repräsentiert, hätte beispielsweise bereits das Paar *type: assignment* in seiner Schlüssel-Wert-Tabelle.

1.3.4. Z4: Attributabhängige Farbauswahl

Um unterschiedliche Knoten- und Kantentypen oder andere Schlüssel-Wert-Paare zu kennzeichnen, ist es möglich, für spezifische Schlüssel-Wert-Kombinationen Farben festzulegen. So werden bestimmte Graphelemente in Abhängigkeit ihrer Attribute in unterschiedlichen Farben dargestellt. Für das Schlüssel-Wert-Paar *type: assignment* kann zum Beispiel die Farbe rot definiert werden, sodass alle Knoten des Graphen, die eine Zuweisung repräsentieren, in rot dargestellt werden.

1.4. Aufbau

In Kapitel 2 werden die benötigten Grundlagen und die genutzten Softwarekomponenten vorgestellt. Kapitel 3 bietet eine kurze Übersicht über die Komponenten des entwickelten Plugins. In Kapitel 4 wird auf das dem Graphen zugrunde liegende Modell eingegangen und Kapitel 5 beschreibt die Transformation von diesem Modell zum dargestellten Graphen. Kapitel 6 geht auf die Editierbarkeit des Graphen mit der Maus (Z1) und die vordefinierten Graphelemente (Z3) ein. Außerdem wird die Umsetzung der Schlüssel-Wert-Tabelle (Z2) und der attributabhängigen Farbauswahl (Z4) erläutert. In Kapitel 7 werden die genannten Ziele anhand eines Anwendungsbeispiels evaluiert. Kapitel 8 zieht schließlich ein Fazit und gibt einen Ausblick für nachfolgende Arbeiten.

Kapitel 2

Grundlagen und Technologien

2.1. Graphen

Ein Graph ist eine Datenstruktur, die sich aus Knoten und Kanten zusammensetzt und mit der Objekte und ihre Beziehungen zueinander dargestellt werden können. [Rodriguez und Neubauer 2010] Dadurch bieten Graphen beispielsweise die Möglichkeit, einen sequentiellen Abschnitt Programmcode zu modellieren. Im Rahmen dieser Arbeit wird dafür mit einer speziellen Art von Graphen gearbeitet, dem *Eigenschaftsgraphen* und darauf aufbauend dem *Systemabhängigkeitsgraphen*, welche im Folgenden genauer erläutert werden.

2.1.1. Eigenschaftsgraph

Bei einem Eigenschaftsgraphen oder Propertygraph (PG) handelt es sich nach Rodriguez und Neubauer [2010] um einen gerichteten, mit Labeln und Attributen versehenen Graphen. Ein beispielhafter Eigenschaftsgraph ist in in Abbildung 2.1 zu sehen. Dort besitzt jeder Knoten ein Attribut *type* mit einem individuellen Attributwert. Die Attribute sind als Schlüssel-Wert-Paare dargestellt und erweitern den Graphen um zusätzliche Informationen. Außerdem sind die Graphelemente mit Labeln versehen, die beispielsweise Bezeichnungen der repräsentierten Objekte oder Beziehungen tragen und so die dargestellten Zusammenhänge verdeutlichen.

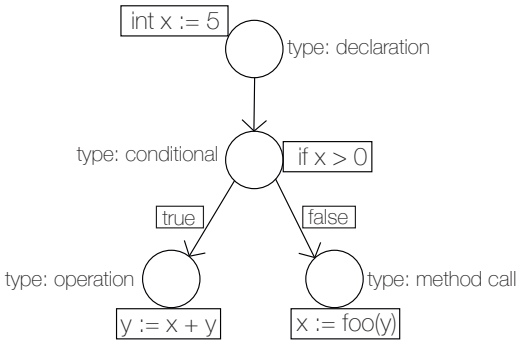

Abbildung 2.1. Beispiel für einen Eigenschaftsgraphen

2.1.2. Systemabhängigkeitsgraph

Eine Vorstufe des hier behandelten Systemabhängigkeitsgraphen oder System Dependence Graph (SDG) wurde erstmals von Ottenstein und Ottenstein [1984] vorgestellt: Der *Programmabhängigkeitsgraph*. Bei diesem repräsentieren die Knoten Programmelemente und die Kanten die Abhängigkeiten zwischen ihnen. Dadurch können für Programme aus einer einzigen Funktion programminterne Abhängigkeiten dargestellt werden.
Zur Repräsentation multiprozeduraler Programme wurde der SDG entwickelt, der jede einzelne Funktion als Programmabhängigkeitsgraphen darstellt. [Reps und Binkley 1990] Eine Java-spezifische Form des SDG entwickelten Walkinshaw u. a. [2003]. Diese soll im Folgenden kurz vorgestellt werden. Abbildung 2.2 verdeutlicht die grundlegende Architektur. Einem SDG liegt eine Ebenenstruktur zu Grunde, um komplexere Zusammenhänge von Programmen durch Nutzung unterschiedlicher Graphtypen pro Ebene übersichtlich repräsentieren zu können. Die unterste Ebene bilden die *Anweisungen*, darüber liegen die *Methoden*, dann die *Klassen*, darüber die *Schnittstellen* und in der obersten Ebene die *Pakete*.

Abbildung 2.2. Architektur von Systemabhängigkeitsgraphen

Um Programmabhängigkeiten und die Funktionalität genau repräsentieren zu können, stehen auf jeder Ebene des SDG unterschiedliche Knoten- und Kantentypen zur Verfügung, die in Tabelle 2.1 aufgelistet sind.

Um ein einfaches Programm ohne mehrere Methoden darstellen zu können, sind die Knoten und Kanten der Anweisungsebene ausreichend. Ein *Statement Knoten* repräsentiert einen einzelnen Ausdruck im Quellcode des repräsentierten Programms. Eine *Control Dependence Kante* verbindet zwei Statementknoten A und B, wenn die Ausführung von Anweisung B auf der Ausführung von Anweisung A aufbaut. Eine *Data Dependence Kante* verbindet zwei Knoten A und B, wenn die Ausführung von Anweisung B eine Variable referenziert, die in Anweisung A definiert oder geändert wurde.

Um ein Programm mit mehreren Methoden darzustellen, wird ein Methodenabhängigkeitsgraph konstruiert. Ein solcher nutzt intern die Knoten- und Kantentypen der Anweisungsebene. Hinzu kommt ein *Method Entry Knoten*, der den Einstiegspunkt für

Tabelle 2.1. Unterschiedliche Knoten- und Kantentypen eines SDG [Walkinshaw u. a. 2003]

Knoten	Kanten
Anweisungsgraph	
Statement	Control Dependence Data Dependence
Methodenabhängigkeitsgraph	
Method Entry Actual In/Out Formal In/Out	Call Dependence Parameter In/Out
Klassenabhängigkeitsgraph	
Class Entry	Class Membership Class Dependence Data Member
Schnittstellenabhängigkeitsgraph	
Interface Entry	Abstract Member Implement Abstract Method Implements
Paketabhängigkeitsgraph	
Package Entry	Package Member

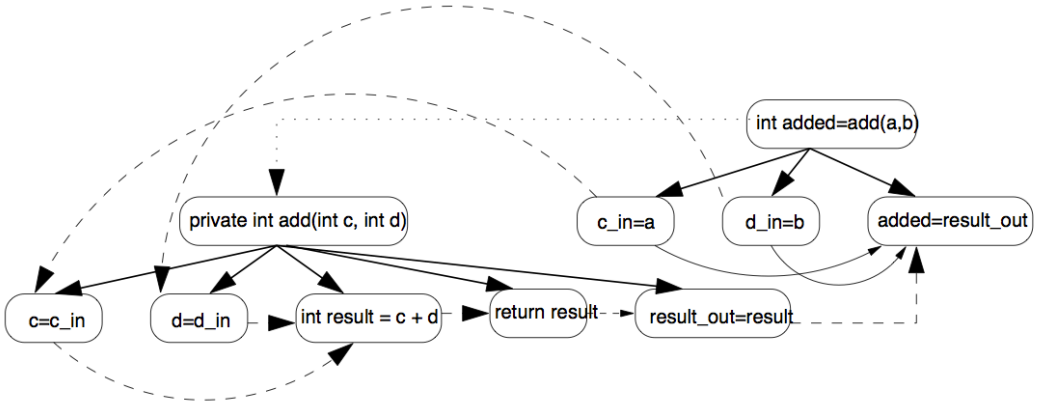

Abbildung 2.3. Beispiel für Methodenabhängigkeitsgraphen [Walkinshaw u. a. 2003]

die Methode darstellt. Wenn ein Parameter an eine Methode übergeben wird, liegt in der Methode nicht der tatsächliche Wert mit seiner Speicheradresse, sondern nur eine lokale (formale) Kopie vor. Daher werden im Graphen der aufrufenden Methode die Knoten der übergebenen und zurückgelieferten Ausdrücke durch *Actual In/Out Knoten* dargestellt und im Graphen der aufgerufenen Methode als *Formal In/Out Knoten*. Der Method Entry Knoten (`private int add(int c, int d)`) in Abbildung 2.3 ist durch Control Dependence Kanten (durchgehende Pfeile) mit den internen Statement Knoten (`int result = c + d` und `return result`), den Formal In Knoten (`c=c_in` und `d=d_in`) und dem Formal

Out Knoten (`result_out=result`) verbunden. Er wird von der aufrufenden Methode (`int added=add(a, b)`) über eine *Call Dependence Kante* (gepunkteter Pfeil) erreicht. Die Paare aus Actual In Knoten (`c_in=a` und `d_in=d`) und Formal In Knoten (`c=c_in` und `d=d_in`) werden durch *Parameter In Kanten* (gestrichelte Pfeile oben) verbunden, Actual Out Knoten (`added=result_out`) und Formal Out Knoten (`result_out=result`) durch eine *Parameter Out Kante* (gestrichelter Pfeil rechts). Bei den vier gestrichelten Kanten im Subgraphen unter `private int add(int c, int d)` handelt es sich um Data Dependence Kanten. Alle weiteren, nicht explizit genannten durchgezogenen Pfeile sind Control Dependence Kanten.

Um eine Klassenhierarchie darzustellen, wird ein Klassenabhängigkeitsgraph konstruiert. Für jede Klasse beinhaltet dieser einen *Class Entry Knoten*, welcher über *Data Member Kanten* mit den Statement Knoten der Attribute der Klasse und über *Class Membership Kanten* mit den Method Entry Knoten der Methoden der Klasse verbunden ist. Erbt eine Klasse von einer anderen, so sind diese durch eine *Class Dependence Kante* verbunden.

Schnittstellen werden mit einem Schnittstellenabhängigkeitsgraphen dargestellt. Für jede Schnittstelle beinhaltet dieser einen *Interface Entry Knoten*, der über *Abstract Member Kanten* mit den Method Entry Knoten der abstrakten Methoden der Schnittstelle verbunden ist. Die Method Entry Knoten der abstrakten Methoden sind mit denen der Methoden, die sie implementieren über *Implements Abstract Method Kanten* verbunden. Implementiert eine Klasse eine Schnittstelle, dann ist sie über eine *Implements Kante* mit ihr verbunden.

Pakete werden durch einen Paketabhängigkeitsgraphen repräsentiert. Ein *Package Entry Knoten* repräsentiert das Paket und wird über *Package Member Kanten* mit den Entry Knoten der enthaltenen Klassen und Schnittstellen verbunden.

2.2. Die Entwicklungsumgebung Eclipse

Eclipse[1] ist ein Werkzeug zur Entwicklung von Software, das eine Entwicklungsumgebung, auch Integrated Development Environment (IDE) und ein erweiterbares Plugin-System enthält. Eclipse wurde primär für die Erstellung von Java-Programmen konzipiert, wird aber mittlerweile wegen seiner Erweiterbarkeit auch für viele andere Entwicklungsaufgaben eingesetzt. Im Kontext des in dieser Arbeit entwickelten Plugins wird in diesem Abschnitt kurz auf die Eclipse *Rich Client Platform* (RCP) eingegangen, welche die Erweiterung durch Plugins ermöglicht.

2.2.1. Die Rich Client Platform

Als RCP basiert die Eclipse-IDE auf einem kleinen Laufzeitkernel, der für das Laden der einzelnen Plugins verantwortlich ist. Alle Eclipse-Funktionalitäten sind selbst auch als

[1] `http://www.eclipse.org/`

Plugins realisiert, sodass immer nur die benötigten Programmteile geladen werden müssen. Alle Erweiterungskomponenten interagieren mit dem Kernel und untereinander in der gleichen Art und Weise. Abbildung 2.4 zeigt die Architektur der Eclipse-IDE.

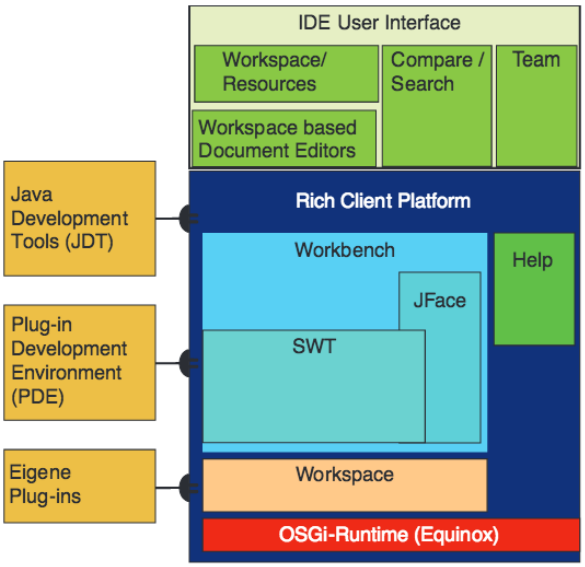

Abbildung 2.4. RCP Architektur der Eclipse-IDE [Steppan 2015]

Die in rot dargestellte Laufzeitumgebung für Plugins ist Equinox. Equinox basiert auf der Spezifikation der *Open Services Gateway initiative* (OSGi), und ermöglicht die Steuerung des Lebenszyklus der Plugins. So können Plugins während der Laufzeit installiert, ausgeschaltet oder ausgetauscht werden. Equinox verwaltet dabei die Abhängigkeiten zwischen den einzelnen Plugins und bildet das Fundament der Eclipse RCP.
Die hellblau dargestellte *Workbench* bietet eine erweiterbare Infrastruktur für die Darstellung von Oberflächenkomponenten wie Dialogen, Menüs oder Toolbars und koordiniert ihr Zusammenspiel in den verschiedenen Fenstern (*Parts*) einer Anwendung. Die Parts können in der Benutzeroberfläche verschoben, übereinander gelegt und beliebig positioniert werden. In dieser Arbeit werden mit *Views* und *Editoren* unterschiedliche Workbench-Parts entwickelt, aus denen sich der entwickelte Grapheditor insgesamt zusammen setzt. Kernbestandteile der Workbench sind das Standard Widget Toolkit (SWT) und das darauf aufbauende JFace-Toolkit, die Bibliotheken zur Implementierung der Oberflächenkomponenten bereitstellen.
Links sind in orange verschiedene Plugins dargestellt, die auf den beschriebenen Kernkomponenten aufbauen und sich zu Rich Client Applications (RCA) wie dem in dieser Arbeit entwickelten Grapheditor zusammensetzen. Die weiteren dargestellten Kompo-

nenten müssen im Rahmen dieser Arbeit nicht detaillierter betrachtet werden. [Steppan 2015]

2.3. Das Eclipse Modeling Framework

Das *Eclipse Modeling Framework* (EMF) ermöglicht das Modellieren von Sachverhalten, Objekten oder Funktionen und nutzt dabei die Möglichkeiten von Eclipse. Grundsätzlich verbindet das EMF wie in Abbildung 2.5 drei unterschiedliche Repräsentationen eines Modells:

Repräsentation

- in Form einer XML-Datei
- in Form von Java-Typen
- in Form eines UML-Diagramms

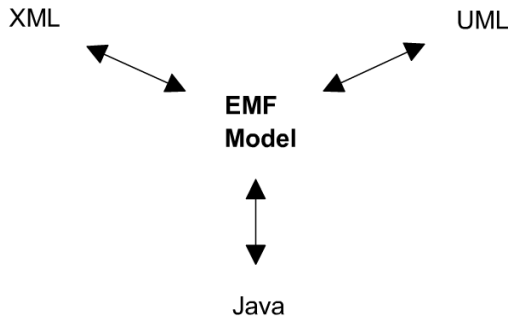

Abbildung 2.5. EMF vereint Java, XML und UML [Steinberg u. a. 2008]

Das EMF ermöglicht es, aus einem in einer beliebigen der drei genannten Formen vorliegenden Modell, die jeweils anderen zu generieren. [Steinberg u. a. 2008]

2.3.1. Das Ecore (Meta-) Modell

Nach Steinberg u. a. [2008] wird das Modell, das genutzt wird, um im EMF Modelle zu repräsentieren, Ecore genannt. Als Modell von Modellen spricht man daher vom *Ecore Metamodell*. Dieses ist seinerseits ebenfalls ein EMF Modell dessen grundlegende Struktur in Abbildung 2.6 dargestellt wird.

Es besteht nach Steinberg u. a. [2008] aus vier Ecore Klassen:

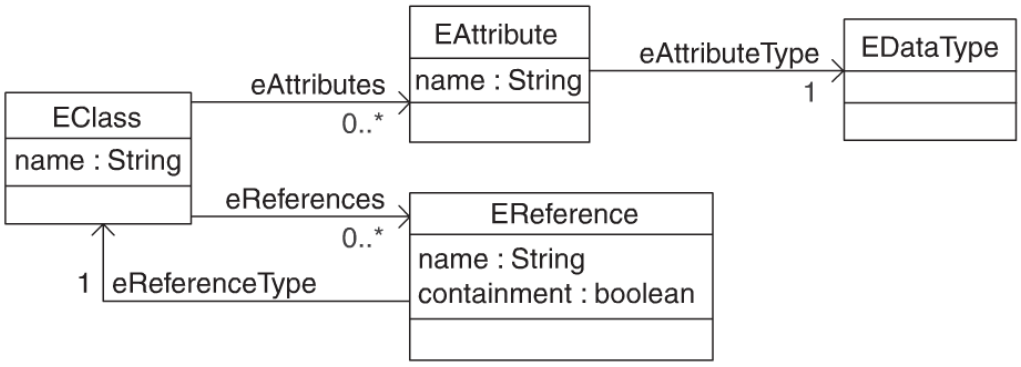

Abbildung 2.6. Vereinfachte Teilmenge des Ecore Metamodells [Steinberg u. a. 2008]

1. **EClass** repräsentiert eine modellierte Klasse, hat einen Namen und beliebig viele Attribute und Referenzen.

2. **EAttribute** repräsentiert ein Attribut einer Klasse, hat einen Namen und einen Datentyp.

3. **EReference** repräsentiert eine Assoziation zwischen der Klasse in der die EReference steht und der auf die sie zeigt. Sie hat einen Namen und ein Containment Flag, mit dem Beziehungen zwischen Teilen und dem Ganzen zu dem sie gehören, modelliert werden können.

4. **EDatatype** repräsentiert den Datentyp eines Attributs. Hier sind sowohl primitive Datentypen als auch komplexe, gegebenenfalls selbst definierte Datentypen möglich.

2.3.2. Code Generierung

Der Quellcode der aus einem Ecore Modell generierten Java-Typen folgt einem festen Schema: Für jede Klasse `MyEClass` des Modells wird zum einen ein Interface `MyClass` generiert und zum anderen eine Klasse `MyClassImpl`, die dieses Interface implementiert.
In dem zu einem Ecore Modell gehörenden *Ecore Generator Model* werden für die Generierung relevante Daten wie der Speicherort für die generieren Klassen gespeichert, die nicht im eigentlichen Ecore Modell enthalten sind.
Es kann zudem ein komplettes Eclipse Editor Plugin generiert werden, um Instanzen des erstellten Modells bearbeiten zu können. In einem Tree Editor können dann Objekte hinzugefügt, gelöscht, ausgeschnitten, kopiert und eingefügt werden. Die Attribute werden jeweils in einem Property Sheet der Standard Eclipse Properties View angezeigt und können dort editiert werden. [Steinberg u. a. 2008]

2.4. Das KIELER Projekt und KLighD

Kiel Integrated Environment for Layout Eclipse RichClient (KIELER[2]) ist der Name eines Forschungsprojekts der Arbeitsgruppe für Echtzeitsysteme und eingebettete Systeme an der Christian-Albrechts-Universität zu Kiel. Hauptziel von KIELER ist die Unterstützung und Entlastung von Programmierern beim grafischen, modellbasierten Entwurf komplexer Systeme.
Von besonderer Bedeutung für die Darstellung und Arbeit mit Graphen wie sie im Rahmen dieser Arbeit stattfinden wird, ist dabei die Subkomponente *KIELER Lightweight Diagrams* (KLighD[3]) welche im Rahmen der KIELER Architektur zum Bereich *Pragmatics* gezählt wird. Dieser umfasst Werkzeuge, die bei der praktischen Arbeit im Kontext modellbasierter Architektur helfen. KLighD legt den Fokus auf eine leichtgewichtige und temporäre grafische Repräsentation in Form eines Diagramms, das in Echtzeit aus einem gegebenen Modell generiert wird. Falls nicht explizit erwünscht, wird diese Repräsentation direkt wieder verworfen, wenn sie nicht mehr gebraucht wird. [Schneider u. a. 2013]

2.4.1. Basismodell und Grafikmodell

In dieser Arbeit wird klar zwischen dem zugrunde liegenden Modell eines Eigenschaftsgraphen und seiner Visualisierung als Diagramm unterschieden. Wir führen daher eine klare begriffliche Trennung zwischen *Basismodell* und *Grafikmodell* ein, da auch dem dargestellten Diagramm in KLighD wiederum ein Metamodell zugrunde liegt. Dieser Abschnitt geht auf die Unterschiede ein und macht deutlich, welche Komponenten zu welchem Modell gehören.

- Das **Basismodell** umfasst das zugrunde liegende Ecore Modell eines Eigenschaftsgraphen, auf welches in Kapitel 4 noch detaillierter eingegangen wird, sowie die daraus generierte Repräsentation als Java Code. Die wichtigsten Java Klassen für die Arbeit mit Eigenschaftsgraphen sind hier `Node` und `Edge`. In dieser Arbeit ist das Basismodell das Modell eines Eigenschaftsgraphen, es stellt also bereits einen Graphen dar, aber prinzipiell sind Modelle beliebiger Objekte als Basismodell möglich.

- Das **Grafikmodell** hingegen ist das Modell, das dem dargestellten Diagramm zugrunde liegt. Das Modell dieses Modells ist der *KGraph*. Der KGraph als Metamodell ist fester Bestandteil von KLighD und liegt allen mit KLighD dargestellten Graphen zugrunde. Er besteht u. a. aus den Klassen `KNode` und `KEdge`. Aus dem damit modellierten Grafikmodell erzeugt KLighD dann zur Laufzeit das dargestellte Diagramm. [Schneider u. a. 2012]

[2] `http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Home`
[3] `http://rtsys.informatik.uni-kiel.de/confluence/pages/viewpage.action?pageId=328115`

2.4.2. Die Synthesis

Die Synthesis ist eine Klasse innerhalb eines KLighD Projekts, die festlegt, wie eine Instanz des Basismodells in eine Grafikmodellinstanz transformiert wird. Das heißt, sie definiert, welche Elemente des Basismodells im Grafikmodell durch welche Elemente des KGraph repräsentiert werden. Da in dieser Arbeit bereits das Basismodell einen Graphen modelliert, werden Nodes und Edges des Basismodells auf KNodes beziehungsweise KEdges des Grafikmodells abgebildet.
Die Synthesis wird in KLighD in der Programmiersprache *Xtend*[4] implementiert, einem Java Dialekt, der zu Java Code kompiliert wird.

[4] https://eclipse.org/xtend/index.html

Kapitel 3

Übersicht über die Komponenten des Grapheditors

Dieses Kapitel gibt einen kurzen Überblick über die unterschiedlichen Komponenten des Grapheditors. Es verweist auf die jeweiligen Kapitel dieser Arbeit, in denen genauer auf jede einzelne Komponente und ihre Implementierung eingegangen wird.

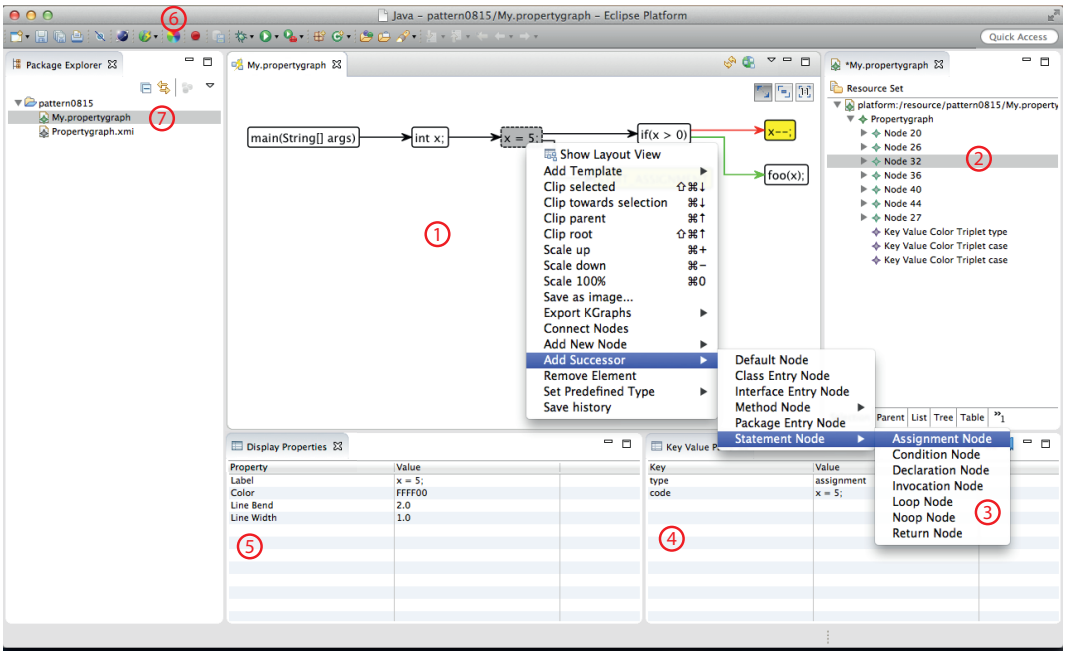

Abbildung 3.1. Übersicht über die Oberfläche des Plugins

Abbildung 3.1 zeigt die Oberfläche des Plugins mit folgenden Komponenten:

1. In der KLighD **Diagram View** auf die in Kapitel 6 eingegangen wird, wird der geöffnete Graph dargestellt.
2. Der **Tree Editor** zeigt die Instanz des Basismodells als Baumstruktur und wird in Kapitel

4 vorgestellt.

3. Der Graph kann über verschiedene **Kontextmenüs** mit der Maus bearbeitet werden, insbesondere können vordefinierte Graphelemente eingefügt werden. Auf die Editierbarkeit und die vordefinierten Elemente wird in 6.1 eingegangen.

4. In der **Key/Value View** werden die Schlüssel-Wert-Paare eines selektierten Graphelements angezeigt und können bearbeitet werden. Sie wird in 6.2 vorgestellt.

5. Die **Display Properties View** zeigt die für die Visualisierung wichtigen Attribute eines selektierten Graphelements und wird in 6.3 behandelt.

6. Über diese Schaltfläche wird das Dialogfenster für die **attributabhängige Farbauswahl** (Abb. 3.2) geöffnet. Dieses ermöglicht die Zuordnung von Schlüssel-Wert-Paaren zu bestimmten Farben und wird in 6.4 erläutert.

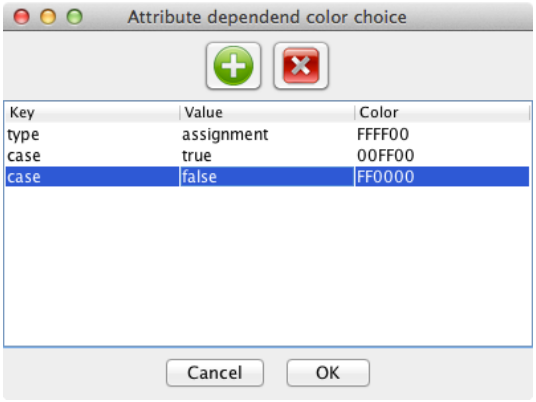

Abbildung 3.2. Dialog für attributabhängige Farbauswahl

7. Über den Package Explorer können angelegte Graphprojekte ausgewählt und geöffnet werden.

Kapitel 4

Architektur des Basismodells

In diesem Kapitel wird das Basismodell vorgestellt, welches dem später erzeugten visuellen Graphen zugrunde liegt. Es wird zunächst auf die grundlegenden Anforderungen an das Modell eingegangen, bevor die konkrete Implementierung mit dem EMF vorgestellt wird.

4.1. Anforderungen an das Basismodell

Da das im Rahmen dieser Arbeit entwickelte Plugin genutzt werden soll, um Systemabhängigkeitsgraphen darzustellen, liegt es nahe, dass das zugrunde liegende Modell genau diesen speziellen Graphen repräsentiert. Dieser verfügt über diverse in 2.1.2 vorgestellte Knoten- und Kantentypen, welche jeweils einzeln als eigene Komponente modelliert werden könnten.
Im Plugin sollen diese Typen allerdings, wie in 1.3 beschrieben, lediglich über spezifische Werte in der Schlüssel-Wert-Tabelle differenziert werden. Eine Modellierung als eigene Komponente, aus der gegebenenfalls für jeden Knoten- oder Kantentyp sogar eine Implementierung als eigener Datentyp resultieren würde, ist also nicht notwendig. Stattdessen repräsentiert in dieser Arbeit das Basismodell einen wie in 2.1.1 vorgestellten Propertygraphen. Dazu müssen Knoten und Kanten modelliert werden. Diese müssen jeweils über ein Label, sowie eine Menge von Schlüssel-Wert-Paaren verfügen. Außerdem wird Knoten und Kanten im Modell jeweils eine eigene Farbe gegeben. Hinzu kommen bei der im nächsten Abschnitt behandelten Implementierung zudem weitere Attribute, die die Konfiguration der Darstellung mit KLighD ermöglichen.

4.2. Umsetzung der Anforderungen

4.2.1. Elemente des Basismodells

In der konkreten Implementierung mit dem EMF besteht das Basismodell des für das Plugin verwendeten Propertygraphen aus den folgenden fünf Klassen mit ihren jeweiligen Attributen:

- **Propertygraph:** Wurzelklasse, die alle untergeordneten Klassen verbindet.
  - *nodes:* Liste von Referenzen auf die Knoten des Graphen.

- *keyValueColorTriplets:* Liste von Referenzen auf die Schlüssel-Wert-Farb-Tripel des Graphen.
- *nodeCounter:* Long-Zähler, um in der Historien-Funktion aus der Arbeit von Benekov [2015] eindeutige IDs für Knoten zu vergeben.
- *edgeCounter:* Long-Zähler, um in der Historien-Funktion aus der Arbeit von Benekov [2015] eindeutige IDs für Kanten zu vergeben.

- **GraphElement:** Elternklasse, die gemeinsame Attribute von Knoten und Kanten zusammenfasst.
  - *label:* String, der das Label des Graphelements repräsentiert.
  - *keyValuePairs:* Liste von Referenzen auf Schlüssel-Wert-Paare.
  - *color:* String, der einen Hexadezimal-RGB-Farbcode repräsentiert.
  - *lineWidth:* Float-Zahl, die die Strichstärke von Kanten und bei Knoten die Strichstärke der Umrandung definiert.
  - *ID:* Eindeutige ID für die Historien-Funktion aus der Arbeit von Benekov [2015].
- **Node:** Klasse für Knoten des Graphen.
  - *incomingEdges:* Liste von Referenzen auf eingehende Kanten.
  - *outgoingEdges:* Liste von Referenzen auf ausgehende Kanten.
  - *lineBend:* Float-Zahl, die die Randkrümmung bestimmt und damit festlegt, wie stark die Ecken des Knoten abgerundet sind.
- **Edge:** Klasse für Kanten des Graphen.
  - *sourceNode:* Referenz auf den Quellknoten der Kante.
  - *targetNode:* Referenz auf den Zielknoten der Kante.
- **KeyValueMap:** Datenstruktur um Schlüssel-Wert-Paare persistent speichern zu können.
  - *key:* String für den Schlüssel.
  - *value:* String für den Wert.
- **KeyValueColorTriplet:** Datenstruktur um Schlüssel-Wert-Farb-Tripel persistent speichern zu können.
  - *key:* String für den Schlüssel.
  - *value:* String für den Wert.
  - *color:* String, der einen Hexadezimal-RGB-Farbcode repräsentiert.

Abbildung 4.1 zeigt die Architektur des Basismodells. Es ist hervorzuheben, dass ein Propertygraph hier ausschließlich aus einer Menge von Knoten besteht. Diese können natürlich durch Kanten verbunden sein, eine Kante kann aber nicht ohne jeweils genau einen Quell- und Zielknoten existieren. In der Klasse `Propertygraph` gibt es keine direkte Referenz auf die Kanten des Graphen. Mit einer solchen Referenz könnten im in 4.2.2 behandelten Tree Editor Kanten ohne Quell- und Zielknoten erstellt werden.
Entscheidend hierfür ist das Containment Flag der Referenzen. In dieser Implementierung enthält der Propertygraph Knoten und Schlüssel-Wert-Farb-Tripel. Ein allgemeines Graphelement enthält Schlüssel-Wert-Paare. Schließlich enthalten Knoten zusätzlich noch die jeweils ausgehenden Kanten. Diese hierarchische Anordnung ermöglicht die Darstellung in einem Tree Editor.

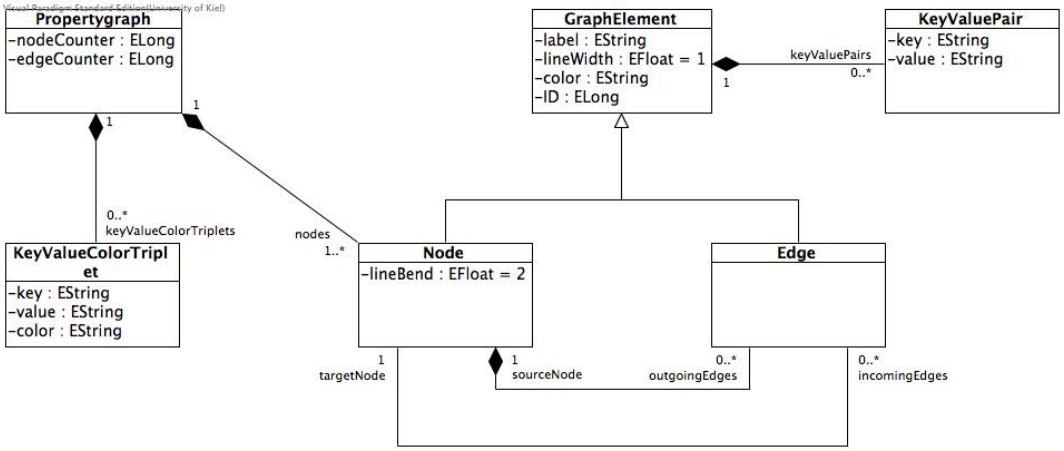

Abbildung 4.1. Abhängigkeiten im Graphmodell

Aus dem in diesem Abschnitt vorgestellten Ecore-basierten Basismodell wird über das zugehörige Generator Modell Java Code generiert. Dieser repräsentiert weiterhin das vorgestellte Modell und dient als Grundlage für die weitere Entwicklung des Plugins. Knoten, Kanten oder ein ganzer Graph können damit als Java Objekte aus dem generierten Code instanziiert werden.

4.2.2. Der EMF Tree Editor für Propertygraphen

Wie in 2.3 beschrieben, ermöglicht es das EMF, für ein Modell ein fertiges Tree Editor Plugin zu generieren. Abbildung 4.2 zeigt den für das Propertygraph Modell generierten Tree Editor. Darin können Knoten, Kanten, Schlüssel-Wert-Paare und Schlüssel-Wert-Farb-Tripel über ein Kontextmenü eingefügt oder entfernt werden. Zum Einfügen wird das Menü durch einen Rechtsklick auf ein Objekt der jeweiligen Containerklasse geöffnet und das

entsprechende Objekt im Untermenü „New Child“ausgewählt. Im gleichen Menü kann ein Objekt über den Menüpunkt „Delete“ entfernt werden.

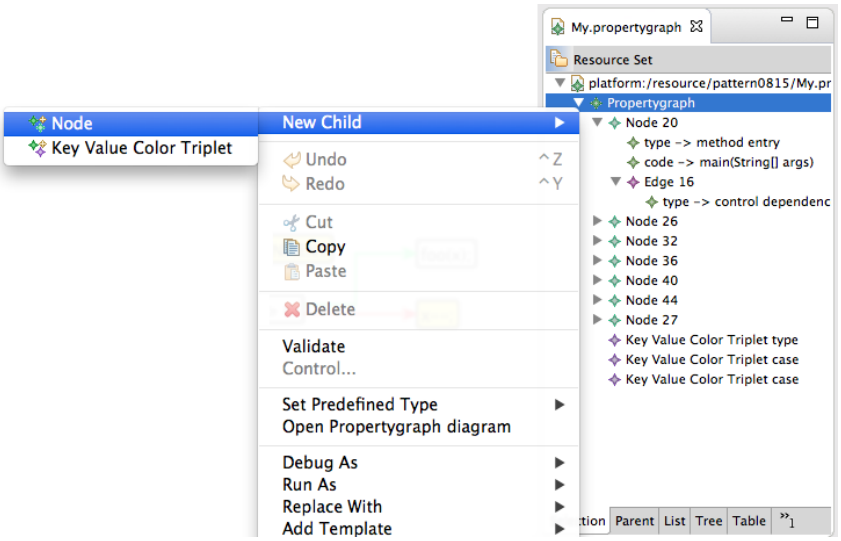

Abbildung 4.2. Generierter Tree Editor für Propertygraphen

Kapitel 5

Transformation vom Basismodell zum Grafikmodell

In 2.4 haben wir die Synthesis kurz eingeführt. In diesem Kapitel beschreiben wir die allgemeine Funktionsweise von Synthesis-Transformationen in KLighD im Detail. Anschließend gehen wir auf unsere konkrete Synthesis-Transformation ein. Wir beschreiben, wie Attribute des Basismodells ins Grafikmodell übertragen werden, wo sie direkt die Darstellung des Graphen beeinflussen.

5.1. Allgemeiner Transformationsprozess in KLighD

Abbildung 5.1 verdeutlicht noch einmal den Zusammenhang aller bisher behandelten Komponenten. Grundlage ist das im vorigen Kapitel behandelte, als Ecore-Modell vorliegende Basismodell aus dem eine Repräsentation als Java Quellcode generiert wurde. Mit dieser ist es möglich, eine Basismodellinstanz zu initialisieren. Die Synthesis wandelt diese nun in eine leichtgewichtige, flüchtige Grafikmodellinstanz, also eine Instanz des in den Grundlagen vorgestellten KGraph um. In der mit dem EMF generierten Tree View wird die Basismodellinstanz als Baum visualisiert. In der Diagram View, welche im folgenden Kapitel vorgestellt und zu einem grafischen Editor erweitert wird, wird grundsätzlich die Grafikmodellinstanz visualisiert. Um im grafischen Editor von Elementen des Grafikmodells auf die zugehörigen Elemente des Basismodells entgegen der Richtung der Synthesis zuzugreifen und diese zu aktualisieren, stellt KLighD entsprechende Funktionen zur Verfügung. Im Kern sieht KLighD jedoch vor allem die schnelle Umwandlung einer Modellinstanz in eine Instanz des KGraph und deren Darstellung als Diagramm vor. Die in Kapitel 6 entwickelte Key/Value und Display Properties View reagieren wie dargestellt auf Mausselektionen in Tree und Diagram View und aktualisieren gegebenenfalls die Basismodellinstanz. In Kapitel 6 wird außerdem darauf eingegangen, wie durch den Vergleich von Schlüssel-Wert-Paaren mit Schlüssel-Wert-Farb-Tripeln die Farben der Elemente in der Basismodellinstanz attributabhängig ausgewählt werden. Die Arbeitsschritte in der Diagram und der Key/Value View werden in der von Benekov [2015] entwickelten Bearbeitungshistorie aufgezeichnet.

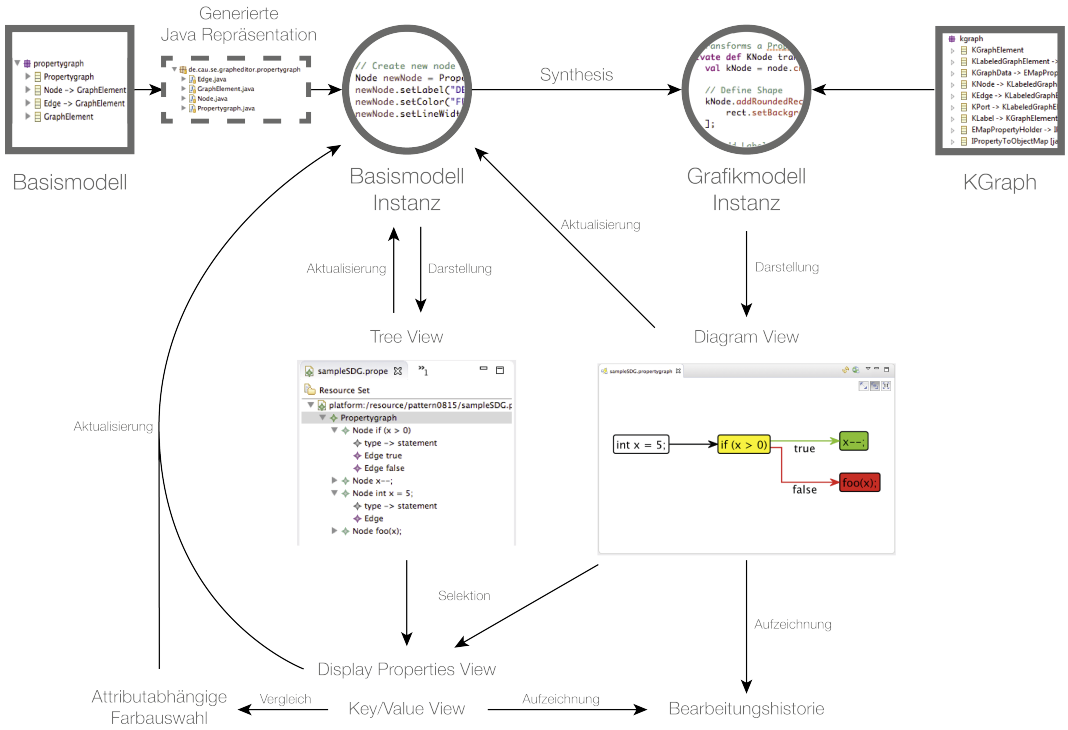

Abbildung 5.1. Allgemeiner Transformationsprozess in KLighD

5.2. Umsetzung der Synthesis

Das Grundgerüst der Synthesis wird beim Anlegen eines neuen KLighD Projekts generiert. Im Projektwizard wird auch schon das Basismodell angegeben. Die Synthesis enthält die Methode `KNode transform(Propertygraph model)`, der als Argument eine Instanz des Basismodells übergeben wird und die diese mit einem `KNode` assoziiert. Dieser `KNode` enthält damit den ganzen Graphen und ist der Rückgabewert der Methode.

5.2.1. Knoten und Kanten

Knoten und Kanten müssen zuvor jeweils auch mit `KNodes` und `KEdges` assoziiert werden. Im Folgenden werden die dafür durchgeführten Schritte beschrieben und anhand des Aktivitätsdiagramms in Abbildung 5.2 verdeutlicht. Für jeden Knoten des Modells wird aus der `transform`-Methode eine selbst definierte Methode aufgerufen, die die Knoten des Basismodells, also Objekte des Typs `Node` mit denen des Grafikmodells, also Objekten des Typs `KNode` assoziiert. Außerdem wird von dieser Methode für alle ausgehenden Kanten

des jeweiligen Knoten eine weitere Methode aufgerufen, die die Kanten des Basismodells, also Objekte des Typs `Edge` mit denen des Grafikmodells, also Objekten des Typs `KEdge` assoziiert.
Für eine ausgehende Kante, die in eine `KEdge` transformiert wird, muss jeweils geprüft werden, ob ihr Zielknoten bereits transformiert wurde, ob für ihn also bereits ein `KNode` existiert oder ob dieser noch erstellt werden muss. An dieser Stelle ist die Implementierung mit Xtend hilfreich, da Xtend für dieses Muster die Definition einer speziellen Methodensignatur anbietet. Damit ist es möglich, für die Zielknoten von Kanten die Transformationsmethode aufzurufen, welche direkt prüft, ob für den betreffenden Knoten schon ein `KNode` angelegt wurde. [Schulze 2015]
Über die KLighD Methode `associateWith` wird jede `KEdge` mit der zugehörigen `Edge`, jeder `KNode` mit dem zugehörigen `Node` und der Wurzelknoten des Diagramms mit dem anfangs übergebenen `Propertygraph` Objekt verknüpft. Um das Grafikmodell zu visualisieren, wurden für `KNodes` über die Methode `addRoundedRectangle` Kästen gezeichnet und für `KEdges` über die Methode `addPolyline` Linien, die dann mit `addHeadArrowDecorator` mit Pfeilspitzen versehen werden. In welchem Maße dabei das Aussehen noch über Attributwerte der Elemente des Basismodells modifiziert werden kann, klären die nächsten Abschnitte.

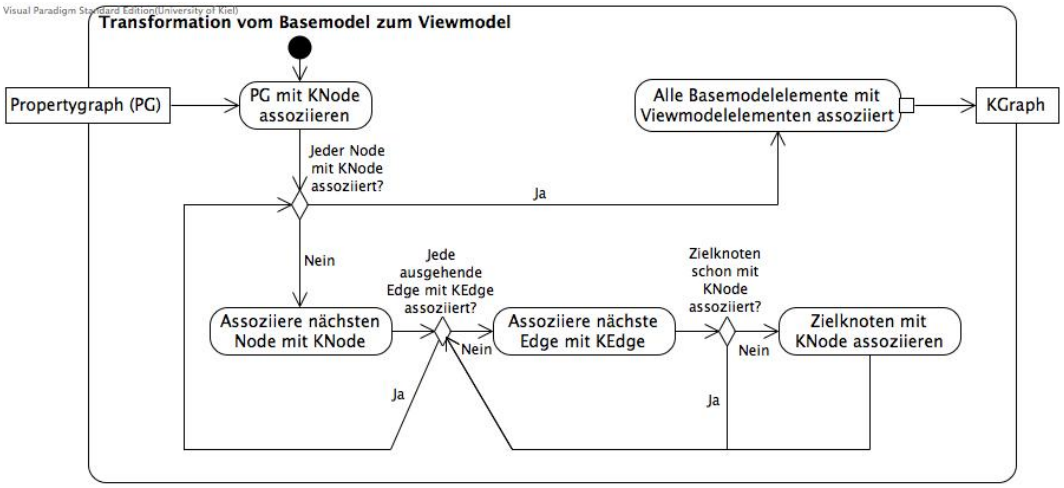

Abbildung 5.2. Aktivitäten bei der Transformation von Knoten und Kanten in der Synthesis

5.2.2. Linienform und Linienstärke

Das *lineWidth* Attribut, das dazu dient, die Linienstärke zu regulieren, wird in der Synthesis als Argument an die Methode `addRoundedRectangle` von `KNodes` beziehungsweise die Methode `addPolyline` von `KEdges` übergeben und beeinflusst so direkt die gezeichneten Kästen beziehungsweise Linien.

Das *lineBend* Attribut, das die Krümmung von Knotenumrandungen an den Ecken beschreibt, wird ebenfalls als Argument an die Methode `addRoundedRectangle` von `KNodes` weitergegeben. Theoretisch ermöglicht die Methode unterschiedliche Krümmungen in x- und y-Richtung. In dieser Arbeit wurde jedoch jeweils lineBend übergeben, sodass die Krümmung in x- und y-Richtung stets gleich ist. Abbildung 5.3 zeigt die unterschiedlichen Darstellungen für unterschiedliche Werte der Attribute lineWidth und lineBend. Im Knoten `if(x > 0)` wurden 1.0 beziehungsweise 2.0 verwendet, im Knoten `x--;` 0.5 beziehungsweise 8.0, im Knoten `foo(x);` 3.0 beziehungsweise 0.0, bei der Kante `false` 0.5 und bei der Kante `true` 3.0.

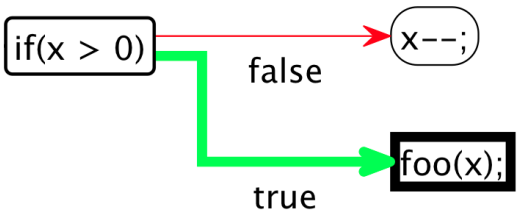

Abbildung 5.3. Unterschiedliche Linienstärken und Linienkrümmungen im Diagramm

5.2.3. Farben

Im Basismodell in der Klasse `GraphElement` werden Farben durch das Attribut *color* als String repräsentiert. In der Synthesis werden die ersten sechs Zeichen dieses Strings als Hexadezimalstring interpretiert, sodass je zwei Zeichen für die Farbwerte zwischen 0 und 255 von Rot, Grün und Blau des RGB-Farbraums stehen. So können Farben als standardisierter Hexadezimalcode eingegeben werden.
Im Grafikmodell werden die Farben mit den Methoden `setBackgroundColor(int r, int g, int b)` für Knoten und `setForegroundColor(int r, int g, int b)` für Kanten gesetzt. Bei Knoten wird so die Füllung gefärbt. Bei Kanten wird der Rand gefärbt, weil die Füllung nur in der Pfeilspitze erkennbar wäre.
Es ist also eine Konvertierung des Strings in Integer-Werte notwendig. Diese wird mit der statischen Methode `int parseInt(String s, int radix)` der Java `Integer`-Klasse umgesetzt. Als erstes Argument werden als Substring zwei Zeichen aus dem Farbcode übergeben. Durch Angabe von 16 als zweitem Argument wird der Substring als Hexadezimalzahl interpretiert.
Bei dem beschriebenen Verfahren können drei Arten von Exceptions ausgelöst werden, die behandelt werden müssen:

1. `NullPointerException`, falls dem Graphen ein neues Element hinzugefügt wird. Da beim Erstellen neuer Elemente nicht explizit die Farbe gesetzt wird, ist der Wert von color dann `null`. In diesem Fall wird der entsprechende Farbwert bei Knoten auf 255 und bei

Kanten auf 0 gesetzt. Der Hintergrund von Knoten wird somit weiß, Kanten werden schwarz.

2. `NumberFormatException`, falls der String andere Zeichen als 0, ... , 9, A, ... , F enthält. In diesem Fall wird der entsprechende Farbwert bei Knoten auf 255 und bei Kanten auf 0 gesetzt. Sind R, G und auch B Wert ungültig, wird der Hintergrund von Knoten somit wieder weiß, Kanten werden schwarz.

3. `StringIndexOutOfBoundsException`, falls ein kürzerer String als sechs Zeichen eingegeben wurde, sodass eine der substring-Methoden auf Stringindizes zugreift, die es nicht gibt. Auch in diesem Fall werden die Farbwerte wie zuvor auf weiß bzw. schwarz gesetzt.

Kapitel 6

Interaktionsmöglichkeiten mit dem Graphen

6.1. Editierbarkeit und Vordefinierte Graphelemente

Eine zentrale Anforderung an das entwickelte Plugin ist die Möglichkeit, den Graphen mit der Maus über Kontextmenüs bearbeiten zu können. Auf die Implementierung dieser Interaktion und die damit verbundenen Möglichkeiten vordefinierte Graphelemente einzufügen, wird in diesem Abschnitt eingegangen.

6.1.1. Darstellung des Graphen in der KLighD Diagram View

Das Ergebnis der Synthesis liegt der KLighD Diagram View vor, welche in Abbildung 6.1 zu sehen ist.

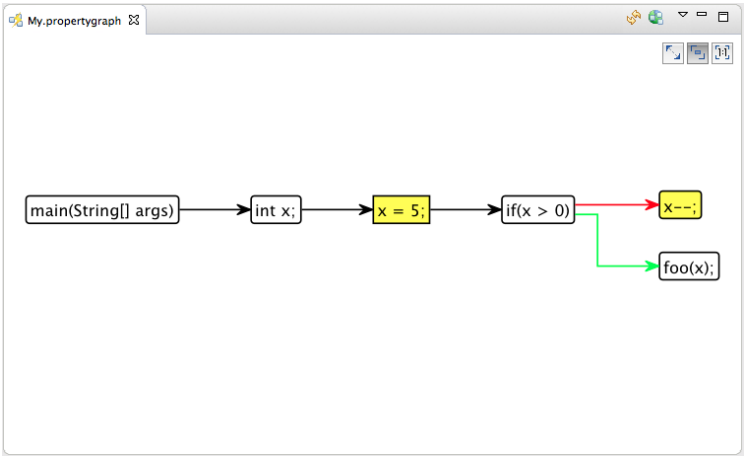

Abbildung 6.1. Die KLighD Diagram View

Hier kann mit dem Mausrad hinein und hinaus gezoomt werden. Bei gehaltener Maustaste kann der sichtbare Ausschnitt verschoben werden. Per Mausklick können einzelne und bei gehaltener Strg-Taste mehrere Elemente selektiert werden. Die Selektion

kann über Strg und + beziehungsweise - vergrößert und verkleinert werden. Ein Rechtsklick auf ein Element öffnet ein Kontextmenü. Wenn dieses für mehrere Elemente aufgerufen werden soll, muss das Letzte bereits mit der rechten Maustaste selektiert werden. Auf die Einträge in diesem Kontextmenü und die Funktionen, die dahinter stehen, wird im nächsten Abschnitt eingegangen.

6.1.2. Editierbarkeit des Graphen

In einem in der Diagram View per Rechtsklick auf ein Graphelement oder eine freie Fläche geöffneten Kontextmenü können folgende Optionen ausgewählt werden, um den Graphen zu bearbeiten:

- **Add New Node** fügt einen neuen, nicht mit Kanten verbundenen Knoten ein.
- **Add Successor** fügt nach dem selektierten Knoten einen neuen Knoten und eine Kante ein, die den selektierten Knoten und den neuen Nachfolger verbindet.
- **Connect Nodes** fügt zwischen dem ersten und dem zweiten selektierten Knoten eine neue Kante ein.
- **Remove Node** entfernt den selektierten Knoten und alle ein- und ausgehenden Kanten.
- **Set Predefined Type** lässt den Benutzer aus vordefinierten Knoten- beziehungsweise Kantentypen wählen, welcher Typ auf das selektierte Graphelement angewendet wird.

Bei den Menüeinträgen *Add New Node*, *Add Successor* und *Set Predefined Type* handelt es sich zunächst nur um Obermenüs, über deren Untermenüs dann aus den unterschiedlichen, vordefinierten Knoten- und Kantentypen des SDG gewählt werden kann (siehe Abbildung 1.2).

6.1.3. Vordefinierte Graphelemente

In den Grundlagen in Kapitel 2 wurde der SDG und seine Knoten- und Kantentypen vorgestellt. Um einzelne Anweisungen im Graphen besser differenzieren zu können, werden in dieser Arbeit für den Statementknoten unterschiedliche Untertypen definiert. Außerdem können Default Knoten und Kanten ohne vordefinierten Typ erstellt werden. Tabelle 6.1 listet alle in dieser Arbeit implementierten Elementtypen auf. Folgende Untertypen stehen für den Statement Knoten zur Verfügung:

- *Zuweisung* z.B `x = 5;`
- *Bedingung* z.B. `if(x > 0)`
- *Deklaration* z.B. `int x;`
- *Aufruf* z.B. `foo(x);`

- *Schleife* z.B. `while(x > 0)`
- *Rückgabe* z.B. `return x;`
- *Noop* soll am Ende eines geklammerten Anweisungsblocks, also z.B. if-Zweigen oder Schleifenkörpern, eingefügt werden, um einen einheitlichen Ausgangspunkt zu erhalten. In einem auf dieser Arbeit aufbauenden, in Kapitel 8 beschriebenen Projekt können so hierarchische Graphen mit aufklappbaren Knoten angelegt werden.

Tabelle 6.1. Unterschiedliche Knoten- und Kantentypen im Plugin

Knoten	**Kanten**
Default	Default
Anweisungsgraph	
Assignment Condition Declaration Invocation Loop Noop Return	Control Dependence Data Dependence
Methodenabhängigkeitsgraph	
Method Entry Actual In/Out Formal In/Out	Call Dependence Parameter In/Out
Klassenabhängigkeitsgraph	
Class Entry	Class Membership Class Dependence Data Member
Schnittstellenabhängigkeitsgraph	
Interface Entry	Abstract Member Implement Abstract Method Implements
Paketabhängigkeitsgraph	
Package Entry	Package Member

6.1.4. Unterschiedliche Elementtypen festlegen

Bei der Definition des Basismodells in Kapitel 4 wurde bereits darauf eingegangen, dass Knoten und Kanten jeweils lediglich der Datentyp `Node` beziehungsweise `Edge` zugrunde liegt und die unterschiedlichen Typen von Knoten und Kanten über spezifische Werte in der Key/Value View definiert werden.
Intern wird mit Enum Typen gespeichert, um welchen Elementtyp es sich handelt. So können die entsprechenden Werte in die Schlüssel-Wert-Tabelle des Elements eingetragen werden. Es ist natürlich auch möglich, den Typ eines Elements über die Schlüssel-Wert-Tabelle unter dem Schlüssel „type“ zu verändern, dort beliebige Werte einzutragen oder das Schlüssel-Wert-Paar komplett zu entfernen.

6.1.5. Umsetzung der Editierbarkeit in einem Kontextmenü

Die Menüeinträge des Kontextmenüs werden in der *plugin.xml* Datei des Plugins des grafischen Editors unter der Extension `org.eclipse.ui.menus` in einer *menuContribution* für Popup-Menüs (`popup:org.eclipse.ui.popup.any`) als `menu` für die Obermenüs und als `command` für die Einträge registriert. Unter der Extension `de.cau.cs.kieler.klighd.extensions` wird weiterhin für jeden der command-Einträge eine `Action` registriert, die jeweils in einer eigenen Klasse im Projekt implementiert ist.
Alle zu den einzelnen Kommandos gehörenden `Action`-Klassen implementieren das Interface `IAction`, dessen Methode `public ActionResult execute(final ActionContext context)` implementiert werden muss, sodass diese bei einem Klick auf den jeweiligen Menüeintrag ausgeführt wird. Um redundanten Code für die vielen Elementtypen zu vermeiden, wurden die Actionklassen `AddNewNodeAction`, `AddSuccessorAction`, `ConnectNodesAction`, `RemoveElementAction`, `SetPredefinedEdgeTypeAction` und `SetPredefinedNodeTypeAction` implementiert.
Bis auf `ConnectNodesAction` und `RemoveElementAction` handelt es sich jeweils um abstrakte Oberklassen, die nur den Code der `execute`-Methode vorgeben. Genaue Elementtypen werden aber erst mit Hilfe der genannten Enum Typen im Konstruktor der in Abbildung 6.2 dargestellten, konkreten Unterklassen definiert. Beim Verbinden von Knoten und Löschen von Elementen ist keine Differenzierung der Elementtypen nötig, was spezifische Unterklassen hier überflüssig macht.
Von der jeweiligen `execute`-Methode aus wird eine Methode aufgerufen, die letztendlich die Änderungen am Graphen ausführt. Diese wird über den KLighD `IModelModificationHandler` aufgerufen, der genutzt wird, um Änderungen am Modell vorzunehmen. In den Parametern wird der Graph, gegebenenfalls relevante Knoten (z.B die bei „Add Successor“/„Connect“/„Remove“ selektierten Knoten) und der Enum Typ eines neu erstellten oder modifizierten Knotens übergeben.

Die genannten, aus der `execute`-Methode aufgerufenen Methoden befinden sich in der Klasse `PropertygraphActionDefinitions`. Sie enthält folgende Methoden:

- `addNewNode`: Erstellt einen neuen Knoten, fügt ihn dem Graphen hinzu und ruft `setPredefinedNodeKVPairs` auf, um die passenden Schlüssel-Wert-Paare einzufügen.
- `addSuccessor`: Erstellt einen Nachfolgerknoten, eine Kante zu diesem Knoten, fügt den Knoten dem Graphen hinzu und ruft `setPredefinedNodeKVPairs` auf, um die passenden Schlüssel-Wert-Paare einzufügen. Wichtig ist hier, dass die Methode die Kante auch den aus-/eingehenden Kanten des Quell-/Zielknotens hinzufügt.
- `connectNodes`: Fügt zwischen den übergebenen Knoten eine Kante ein und fügt diese auch den aus-/eingehenden Kanten des Quell-/Zielknotens hinzu.
- `removeElement`: Entfernt bei Knoten alle ein- und ausgehenden Kanten inklusive der Referenzen auf sie in Vor- beziehungsweise Nachfolgerknoten, sowie den Knoten selbst

aus dem Graphen. Bei Kanten entfernt die Methode nur jeweils die Referenz im Quell- und Zielknoten.

- `setPredefinedEdgeKVPairs`: Setzt anhand des übergebenen Enum Typs für Kanten vornefinierte Schlüssel-Wert-Paare.
- `setPredefinedNodeKVPairs`: Setzt anhand des übergebenen Enum Typs für Knoten vornefinierte Schlüssel-Wert-Paare.

Die Klasse und ihre Methoden sowie die Assoziationen zu den aufrufenden `Action`-Klassen fasst Abbildung 6.2 zusammen.

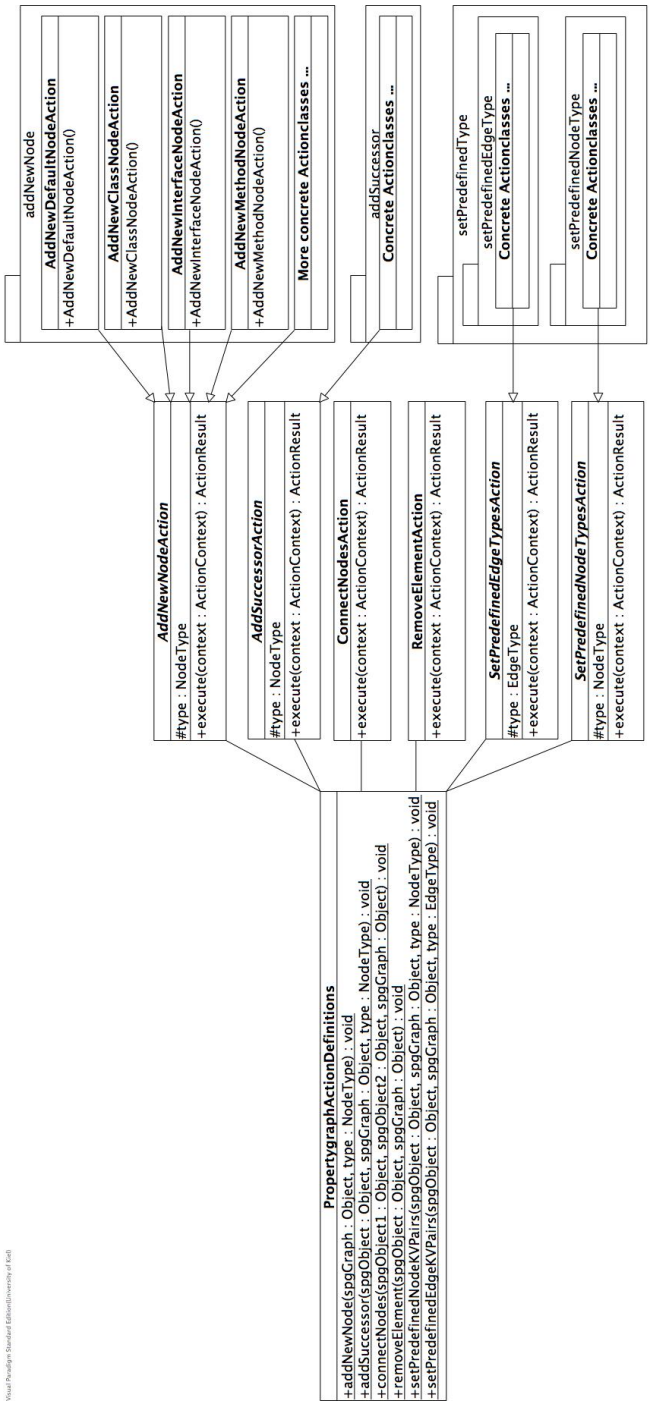

Abbildung 6.2. Klassendiagramm mit Definitionsklasse und Actionklassen des Kontextmenüs

6.2. Key/Value View

In Kapitel 4 wurden bereits die Schlüssel-Wert-Paare genannt, die Knoten und Kanten des bearbeiteten SDG haben können. Um diese Paare anzuzeigen und zu bearbeiten, wurde die Key/Value View entwickelt, welche in diesem Abschnitt vorgestellt wird.

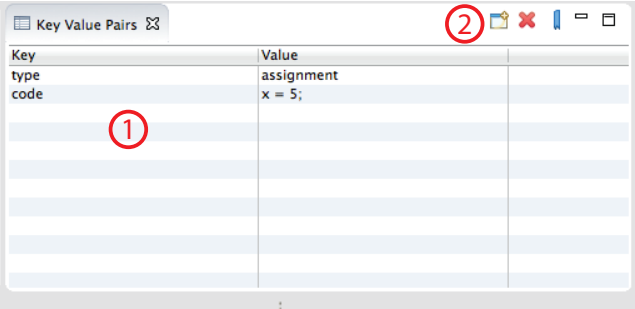

Abbildung 6.3. Key/Value View im Detail

Abbildung 6.3 zeigt die View im Detail. Sie besteht aus den beiden folgenden funktionalen Bereichen:

1. Einer **Tabelle**, die die Schlüssel-Wert-Paare des momentan selektierten Graphelements zeigt.

2. Einer Toolbar mit Schaltflächen um Schlüssel-Wert-Paare **hinzuzufügen**, zu **löschen** und den Wert als **Label festzulegen**.

Weiterhin wird in diesem Abschnitt erläutert, wie die View Daten mit dem Basismodell austauscht.

6.2.1. Umsetzung der Schlüssel-Wert-Tabelle

Die Tabelle zeigt die Schlüssel-Wert-Paare des momentan selektierten Graphelements. Wird ein Wert angeklickt, kann er geändert werden. Mit Betätigung der Enter-Taste wird der geänderte Wert dann bestätigt und in den Graphen übernommen.
Für die Implementierung der editierbaren Tabelle wurden die Klassen `org.eclipse.swt.widgets.Table`[1] und `org.eclipse.swt.custom.TableEditor`[2] verwendet. Da die Klasse `Table` allein nur Funktionen zum Anzeigen und Selektieren von Daten in einer Tabelle bietet, wurde ein `TableEditor` verwendet. Die zugehörige Eclipse Dokumentation bietet bereits

[1] `http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fswt%2Fwidgets%2FTable.html`

[2] `http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fswt%2Fcustom%2Fpackage-summary.html`

ein Quellcodemuster zur grundlegenden Implementierung des Editors an, welches für diese Arbeit genutzt wurde. Darin wird ein `TableEditor` in der GUI über einer Zelle der Tabelle positioniert und passt sich dann bei Veränderungen der Zellgröße an. Da der Inhalt einer Zelle nicht geändert werden kann, simuliert der darüberliegende `TableEditor` diese Funktion, indem er den bestehenden Zellinhalt anzeigt. Gegebenenfalls liest er einen neuen Zellinhalt als String ein und ändert den Inhalt der Tabelle an der entsprechenden Position selbst. Abbildung 6.4 zeigt den Unterschied zwischen der eigentlichen Tabelle und der Tabelle mit dem darüber gelegten `TableEditor`.

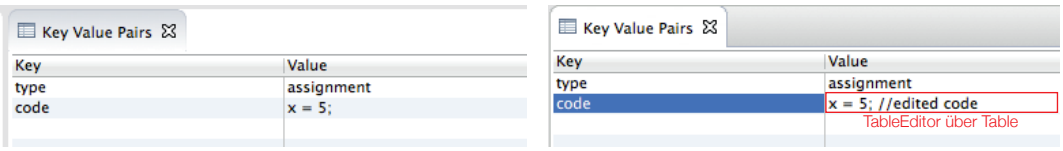

Abbildung 6.4. Unterschied zwischen Table und TableEditor

Dem `TableEditor` wird im Konstruktor ein `TableItem` übergeben. Dieses stellt eine komplette Zeile einer Tabelle dar und es besteht keine Möglichkeit, über den `TableEditor` die Position einer einzelnen selektierten Zelle in einer Zeile zu erfragen. Es wird immer die komplette Zeile selektiert. Um einzelne Zellen bearbeiten zu können, wird daher ein `TableEditor` an eine konstante Spaltennummer gebunden. Er verändert so je nach selektierter Zeile nur die Zellen in dieser Spalte. In dieser Arbeit wird ein Editor für die Wert-Spalte genutzt, es ist also nicht möglich, die Schlüssel in der Schlüssel-Spalte zu ändern, nachdem sie erstellt wurden.

6.2.2. Die Toolbar

Die Toolbar bietet folgende Funktionen:

- **Hinzufügen eines Schlüssel-Wert-Paares:** Über diese Schaltfläche kann ein neues Schlüssel-Wert-Paar hinzugefügt werden. Dabei öffnet sich zunächst ein Dialogfenster in dem ein Name für den zu erstellenden Schlüssel eingegeben werden muss. Danach kann der zugehörige Wert in der Tabelle angepasst werden.
  Technisch wird der Tabelle dabei ein `TableItem` hinzugefügt, welches an Position 0 den im Dialogfenster festgelegten Schlüssel enthält und an Position 1 zunächst den Wert „DEFAULT“.

- **Löschen eines Schlüssel-Wert-Paares:** Wird die Schaltfläche „Löschen“ betätigt, so wird das momentan selektierte Schlüssel-Wert-Paar aus der Tabelle entfernt.
  Dies wird umgesetzt, indem das selektierte `TableItem` entfernt wird. Um Fehler auszuschließen, wird jeweils geprüft, ob die Selektion nicht leer ist. In diesem Fall wäre der Index der selektierten Zeile -1.

▷ **Einen Wert als Label festlegen:** Die dritte Schaltfläche der Toolbar dient dazu, den Wert des selektierten Schlüssel-Wert-Paares als Label des zugehörigen Graphelements festzulegen. Diese Funktion ermöglicht es, die Visualisierung des Graphen an die Anforderungen des Nutzers anzupassen, ohne dass das Label für jedes Graphelement einzeln von Hand geschrieben werden muss.

6.2.3. Datenaustausch mit dem Graphen

Die Schlüssel-Wert-Tabelle tauscht Daten mit der den Graphen repräsentierenden Basismodellinstanz aus, damit der Zugriff auf die Elemente des bearbeiteten Graphen möglich ist. Beim Selektieren eines Knotens oder einer Kante in der Diagram View oder in der Tree View werden die zum Element gehörenden Schlüssel-Wert-Paare in der Tabelle angezeigt. Eclipse bietet dafür den *Selection Service*[3] an. Eine View in der Eclipse-Oberfläche (Workbench) stellt über einen so genannten *Selection Provider* eine Sammlung der momentan selektierten Objekte zur Verfügung. Andere Views können das Interface `org.eclipse.ui.ISelectionListener`[4] implementieren, welches die `selectionChanged`-Methode bereitstellt, die aufgerufen wird, wenn in einer beliebigen View ein Objekt selektiert wird.
Wie in Listing 6.1 deutlich wird, müssen in der `selectionChanged`-Methode die relevanten Workbench Parts manuell gefiltert werden. Tree View (`PropertygraphEditor` in Zeile 4) und Diagram View (`DiagramViewPart` in Zeile 10) liefern die selektierten Objekte jeweils als StructuredSelection, aus der das selektierte Element z.B. mit der Methode `getFirst()` ausgelesen werden kann.

```
1  public void selectionChanged(IWorkbenchPart part, ISelection sel) {
2
3          // Edit by selecting in tree view
4          if (part instanceof PropertygraphEditor
5                          && sel instanceof StructuredSelection) {
6                  ...
7          }
8
9          // Edit by selecting in diagram
10         else if (part instanceof DiagramViewPart
11                         && sel instanceof StructuredSelection) {
12                 ...
13         }
14 }
```

Listing 6.1. Filtern relevanter Workbench Parts

Während die Tree View für Knoten und Kanten Objekte des Typs `Node` beziehungsweise `Edge`, also Elemente des Basismodells zurück liefert, sind es bei der Diagram View Objekte

[3] `https://eclipse.org/articles/Article-WorkbenchSelections/article.html`

[4] `http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fswt%2Fevents%2FSelectionListener.html`

des Typs `KNode` beziehungsweise `KEdge`, also Elemente des Grafikmodells. Bei diesen muss wie in Zeile 11-12 von Listing 6.2 über entsprechende Methoden erst das entsprechende Element des Basismodells gefunden werden.

```
1         ...
2         else if (part instanceof DiagramViewPart
3                         && sel instanceof StructuredSelection) {
4                 table.removeAll();
5
6                 DiagramViewPart dvpart = (DiagramViewPart) part;
7                 StructuredSelection strucSel = (StructuredSelection) sel;
8
9                 if (strucSel.getFirstElement() instanceof KNode) {
10                        KNode kni = (KNode) strucSel.getFirstElement();
11                        Object sourceElem = dvpart.getViewer().getViewContext()
12                                        .getSourceElement(kni);
13
14                        if (sourceElem instanceof Node) {
15                                selectedElement = (Node) sourceElem;
16                                copyModelDataToTable();
17                        }
18                }
19        }
20        ...
```

Listing 6.2. Zu Grafikmodell-Element zugehöriges Basismodell-Element finden

Dieses Basismodell-Element wird in Zeile 15 in der globalen Variable `selectedElement` vom Typ `GraphElement` gespeichert, sodass die entsprechenden Attribute ausgelesen und mit der Methode `copyModelDataToTable()` in Zeile 16 in die Tabelle kopiert und dort editiert werden können.
Wurden sie editiert und zur Bestätigung die Enter-Taste gedrückt, werden die veränderten Werte mit der Methode `copyTableDataToModel()` zurück in das Basismodell-Element geschrieben.

6.3. Display Properties View

Neben den Schlüssel-Wert-Paaren besitzt jedes Graphelement wie in Kapitel 4 beschrieben noch weitere Attribute, über die das Aussehen des mit KLighD erzeugten Graphen modifiziert werden kann. Um diese Attribute anzeigen und bearbeiten zu können, wurde die in Abbildung 6.5 dargestellte Display Properties View entwickelt. Die technische Umsetzung gleicht zu großen Teilen der der Key/Value View. Auf signifikante Gemeinsamkeiten und Unterschiede wird in 6.3.2 genauer eingegangen.

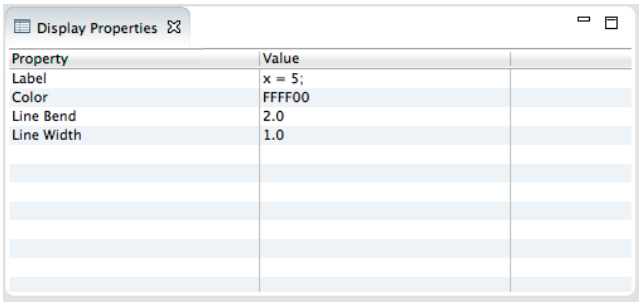

Abbildung 6.5. Display Properties View

6.3.1. Angezeigte Attribute

Wird ein Graphelement selektiert, zeigt die View die entsprechenden Attribute an und ermöglicht, deren Werte zu verändern. Die folgenden Attribute werden in der View angezeigt:

- **Label**: Die Zeichenkette, die unter Label eingetragen werden kann, wird im Grafikmodell als `KLabel` verwendet. Sie findet sich als Beschriftung an Knoten und Kanten in der Diagram View.
- **Color**: Hier kann als Hexadezimalstring eine RGB-Farbe für das entsprechende Element definiert werden, die dann, wie in Kapitel 5 beschrieben, in einzelne RGB Werte dekodiert wird. So wird die Farbe von Knoten und Kanten gesetzt.
- **Line Width**: Die hier eingetragene Float-Zahl bestimmt die Linienstärke der Umrandung von Knoten beziehungsweise die Linienstärke einer Kante.
- **Line Bend** legt bei Knoten im Diagramm die Krümmung an den Ecken fest. 0 sind rechtwinklige Ecken, höhere Werte führen zu immer stärker abgerundeten Ecken.

6.3.2. Abgrenzung zur Key/Value View

Die technische Umsetzung der Tabelle mit den Klassen `Table` und `TableEditor` gleicht der der Key/Value View. Auch das Selection Listener Interface wurde hier wieder verwendet, um auf Selektionen in der Diagram View oder im Tree Editor zur reagieren. In der Display Properties View werden Wertpaare angezeigt, die technisch auch in der Key/Value View angezeigt werden könnten.
Warum wurde also eine zusätzliche View implementiert? Zum einen bleiben die in 2.1.1 vorgestellten Schlüssel-Wert-Paare eines Eigenschaftsgraphen so stets klar von den in dieser Implementierung nur für die Visualisierung wichtigen Attributpaaren getrennt. Im Gegensatz zu den Paaren in der Key/Value View sollen in der Display Properties View auch keine Paare hinzugefügt oder gelöscht werden können. Zum anderen ermöglicht die

Benutzeroberfläche von Eclipse so eine flexible Anordnung, die der Nutzer frei bestimmen kann. Falls die Display Properties View nicht gebraucht wird, kann sie zugunsten der Übersicht geschlossen oder beispielsweise in einen Tab neben die Key/Value View gelegt werden.

6.4. Attributabhängige Farbauswahl

Der Grapheditor bietet dem Nutzer die Möglichkeit, für bestimmte Kombinationen von Schlüssel und Wert Farben festzulegen, sodass alle Graphelemente, deren Schlüssel-Wert-Tabelle genau diese Kombination enthält, in der definierten Farbe dargestellt werden.

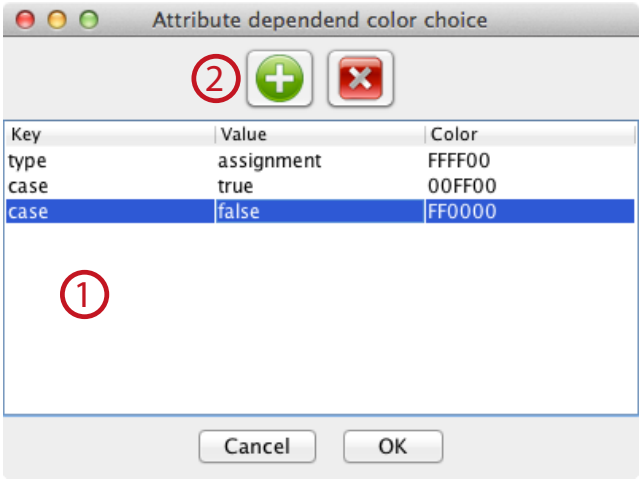

Abbildung 6.6. Farbauswahlfenster

Die Definition erfolgt über das in Abbildung 6.6 dargestellte Dialogfenster, welches folgende Komponenten enthält:

1. Eine **Tabelle**, die alle bereits definierten Schlüssel-Wert-Farb-Kombinationen (KVC-Tripel) auflistet.
2. Schaltflächen, um KVC-Tripel **hinzuzufügen** und zu **löschen**.

6.4.1. Das Dialogfenster

Die Tabelle im Dialogfenster listet alle im Basismodell gespeicherten KVC-Tripel auf. Im Gegensatz zur in 6.2 beschriebenen Tabelle können hier alle Werte direkt durch Anklicken in der Tabelle geändert werden. Änderungen werden jeweils durch Betätigung der Enter-Taste übernommen.

Das Dialogfenster wurde als `JFrame` mit Hilfe des Pakets `javax.swing` entwickelt und für die Implementierung der Tabelle wurde dazu die Klasse `javax.swing.JTable`[5] verwendet. Damit erstellte Tabellen sind anders als die Tabelle in 6.2 direkt durch Anklicken editierbar. Zur `JTable` gehört zudem ein entsprechendes Datenmodell, dessen Daten geändert werden können und letztendlich in der Tabelle angezeigt werden. In dieser Arbeit wurde das `DefaultTableModel` gewählt, die Tabellendaten liegen bei diesem Modell in der Datenstruktur `Vector` vor.
Oberhalb der Tabelle befinden sich zwei Schaltflächen:

- **Hinzufügen eines KVC-Tripels:** Über diese Schaltfläche kann ein weiteres Tripel hinzugefügt werden. Dabei wird der Tabelle zunächst das Tripel (DEFAULT, DEFAULT, DEFAULT) hinzugefügt. Diese Werte kann der Benutzer dann durch Anklicken editieren. Das Tripel wird zunächst nur der Tabelle beziehungsweise dem ihr zugrunde liegenden Datenmodell hinzugefügt, jedoch wird es noch nicht persistent im Graphen gespeichert, wie im folgenden Abschnitt 6.4.2 genauer erläutert wird.

- **Löschen eines KVC-Tripels:** Über diese Schaltfläche kann das selektierte Tripel gelöscht werden. Auch hierbei wird es zunächst nur aus dem Datenmodell der Tabelle entfernt, aber noch nicht aus den KVC-Tripeln des Graphen.

6.4.2. Datenaustausch mit dem Graphen

Anders als bei der in Abschnitt 6.2 beschriebenen Key/Value View muss kein Graphelement selektiert werden, um das Dialogfenster für die attributabhängige Farbauswahl zu öffnen. Wenn ein Graph geöffnet ist, soll auch das Dialogfenster geöffnet werden können. Daraus folgt allerdings, dass das Dialogfenster nicht wie die Key/Value View mit einem Listener Interface Selektionen überwachen kann, um Zugriff auf das Basismodell zu erhalten. Aus diesem Grund wird dem Dialogfenster eine Referenz auf den Graphen im Konstruktor übergeben.
Nach dem Betätigen der Schaltfläche zum Öffnen des Dialogfensters wird die `execute`-Methode einer in der plugin.xml Datei registrierten Handler Klasse aufgerufen. Von dort wird das Dialogfenster geöffnet, allerdings muss zunächst das Basismodell gefunden werden. Dazu wird unter den aktiven Views des geöffneten Eclipse Fensters nach der Diagram View gesucht. Diese hat eine Referenz auf die Basismodellinstanz, welche dann an den Konstruktor des Dialogfensters übergeben werden kann.
Da so im Dialogfenster eine Referenz auf die Basismodellinstanz vorliegt, kann auch auf die darin gespeicherten KVC-Tripel zugegriffen werden. Da die im Fenster angezeigte Tabelle allerdings auf dem `DefaultTableModel` basiert, müssen die Daten, die ausgetauscht werden sollen, jeweils in das Vektorformat konvertiert oder aus einem eine Tabellenzeile repräsentierenden Vektor ausgelesen werden.

[5] `http://docs.oracle.com/javase/7/docs/api/javax/swing/JTable.html`

Im Konstruktor des Dialogfensters müssen zunächst alle KVC-Tripel des Graphen in Vektoren gespeichert werden, um sie in die Tabelle einzufügen. Hat der Benutzer die Eingabe beendet und bestätigt diese durch die Schaltfläche „OK“, werden die Tripel aus dem Tabellenmodell wieder im KVC-Tripel Datentyp des Basismodells gespeichert. Dazu wird zunächst die im Graphen vorliegende Liste von Tripeln geleert, bevor die Tripel aus dem Dialogfenster wieder eingefügt werden. So ist keine Überprüfung notwendig, welche Tripel schon im Graphen vorhanden waren und nicht noch einmal eingefügt werden dürfen. Allerdings findet bisher auch keine explizite Überprüfung auf doppelte Tripel statt, das heißt, dass der Nutzer mehrere Tripel mit exakt gleichen Werten erstellen kann.
Bricht der Benutzer die Eingabe durch Betätigen der Schaltfläche „Cancel“ ab, dann werden die Daten des Dialogfensters nicht zurück in den Graphen geschrieben und das Dialogfenster wird inklusive der geänderten Tabellendaten verworfen.

Bisher wurde nur erklärt, wie die KVC-Tripel eines Graphen über die Benutzeroberfläche von Eclipse definiert und editiert werden können. Nun wird darauf eingegangen, wie sie auf jedes einzelne Element des Graphen angewendet werden, um automatisch bestimmte Farben für den angezeigten Graphen auszuwählen. Es wird hierbei bisher nicht zwischen Tripeln für Knoten und Kanten unterschieden.
Die dafür genutzten Methoden befinden sich in der Klasse `PropertygraphUtil` im Paket `de.cau.se.grapheditor.propertygraph.util` und werden sowohl aufgerufen, wenn über das Dialogfenster Änderungen an den KVC-Tripeln vorgenommen wurden, als auch, wenn in der Key/Value View Änderungen vorgenommen wurden und diese zurück in den Graphen geschrieben werden.
In der Implementierung wird über alle Knoten des Graphen und zu einem Knoten jeweils über seine ausgehenden Kanten iteriert. Für das jeweilige Element werden dann in einer weiteren Methode jeweils die Schlüssel-Wert-Paare mit den KVC-Tripeln verglichen und gegebenenfalls der Farbwert des Elements auf den des Tripels gesetzt.
Damit die Farbe eines Graphelements auf den Farbwert eines KVC-Tripels gesetzt wird, muss unter den Schlüssel-Wert-Paaren des Elements ein Paar sein, bei dem der Schlüssel und der Wert mit denen des Tripels übereinstimmen. Es kann passieren, dass für mehrere Schlüssel-Wert-Paare eines Graphelements jeweils ein passendes KVC-Tripel gefunden wird. Momentan werden alle Schlüssel-Wert-Paare mit allen KVC-Tripeln verglichen und bei einer Übereinstimmung die Farbe des Elements auf den im Tripel spezifizierten Wert gesetzt. Die Suche wird aber dennoch fortgesetzt, sodass später gefundene Übereinstimmungen mit anderen KVC-Tripeln zuvor gesetzte Farben überschreiben. In der aktuellen Implementierung wird also stets die Farbe des zuletzt passenden Tripels auf ein Graphelement angewandt.

Kapitel 7

Evaluierung

In diesem Kapitel wird anhand eines Anwendungseispiels evaluiert, inwiefern die in Kapitel 1 definierten Ziele in dieser Arbeit erfüllt wurden. Weiterhin wird noch einmal auf ausgewählte Punkte, die gegebenenfalls im Rahmen späterer Arbeiten weiterentwickelt werden können, eingegangen.

7.1. Anwendungsbeispiel

Um die einzelnen Ziele aus 1.3 zu evaluieren, wird in diesem Abschnitt mit dem entwickelten Grapheditor ein mögliches Candidatepattern erstellt. Als Vorlage für das Muster dient die in Listing 7.1 angegebene Funktion, die die Komponenten des übergebenen Arrays in einer Schleife aufsummiert. Grundsätzlich könnte dazu das Array in mehrere Teile zerlegt werden, deren jeweilige Komponenten zunächst parallel aufsummiert werden. Am Ende müssten dann noch die einzelnen Teilergebnisse aufsummiert werden.

```
1 public int sum(final int[] a) {
2         int sum = 0;
3
4         for(int i = 0; i < a.length(); i++) {
5                 sum = sum + a[i];
6         }
7
8         return sum;
9 }
```

Listing 7.1. Beispiel für parallelisierbaren Quellcode

Für die Methode wird nun auf den folgenden Seiten mit dem entwickelten Grapheditor der zugehörige Methodenabhängigkeitsgraph konstruiert. In Abbildung 7.1 wird zunächst ein neuer Method Entry Knoten mit dem Label `public int sum(final int[] a)` (Zeile 1) angelegt, bevor über den Menüpunkt *Add Successor* nach und nach die weiteren Knoten eingefügt werden.

Für `int sum = 0;` (Zeile 2) und `sum = sum + a[i];` (Zeile 5) werden dabei Assignment Knoten gewählt, für die Schleifendefinition `for(int i = 0; i < a.length(); i++)` (Zeile 4) wird ein Loop Knoten erstellt. Über die Funktion *Connect Nodes* wird wie in Abbildung 7.2

eine weitere Kante vom Schleifenkörper auf die Schleifendefinition eingefügt.

Als Endpunkt von Schleife und Methode werden wie in Kapitel 6 erklärt, mit * markierte Noop Knoten eingefügt. Für das Return-Statement `return sum;` (Zeile 8) wird ein Return Knoten eingefügt. Damit konnten wie in **Z1** gefordert alle Knoten über den grafischen Editor erstellt werden.
Um die Überprüfung der Schleifenbedingung im Graphen aufzunehmen, wird den Kanten, die vom Schleifenkörper zurück zur Definition beziehungsweise aus der Schleife heraus führen, wie in Abbildung 7.3 in der in **Z2** geforderten Schlüssel-Wert-Tabelle jeweils das Attribut *loop compare* hinzugefügt. Als Wert wird *true* eingetragen, wenn die Schleifenbedingung weiterhin erfüllt ist und *false*, wenn sie nicht mehr erfüllt ist. Die beiden Werte werden über die entsprechende Schaltfläche der Key/Value View auch als Label ins Diagramm übertragen.

Um eine deutlichere Kennzeichnung der gerade genannten Kanten vorzunehmen, wird wie in Abbildung 7.4 dargestellt, über den Dialog zur attributabhängigen Farbauswahl für die Schlüssel-Wert-Kombination *loop compare: true* die Farbe grün und für die Kombination *loop compare: false* die Farbe rot definiert, sodass die entsprechenden Kanten nun im Diagramm farbig dargestellt werden wie es in **Z4** gefordert war.
Schließlich können über die Funktion *Set Predefinend Type* den Kanten die entsprechenden Typen zugewiesen werden oder die Typen der Knoten nachträglich verändert werden, sodass auch **Z3** erfüllt wurde.

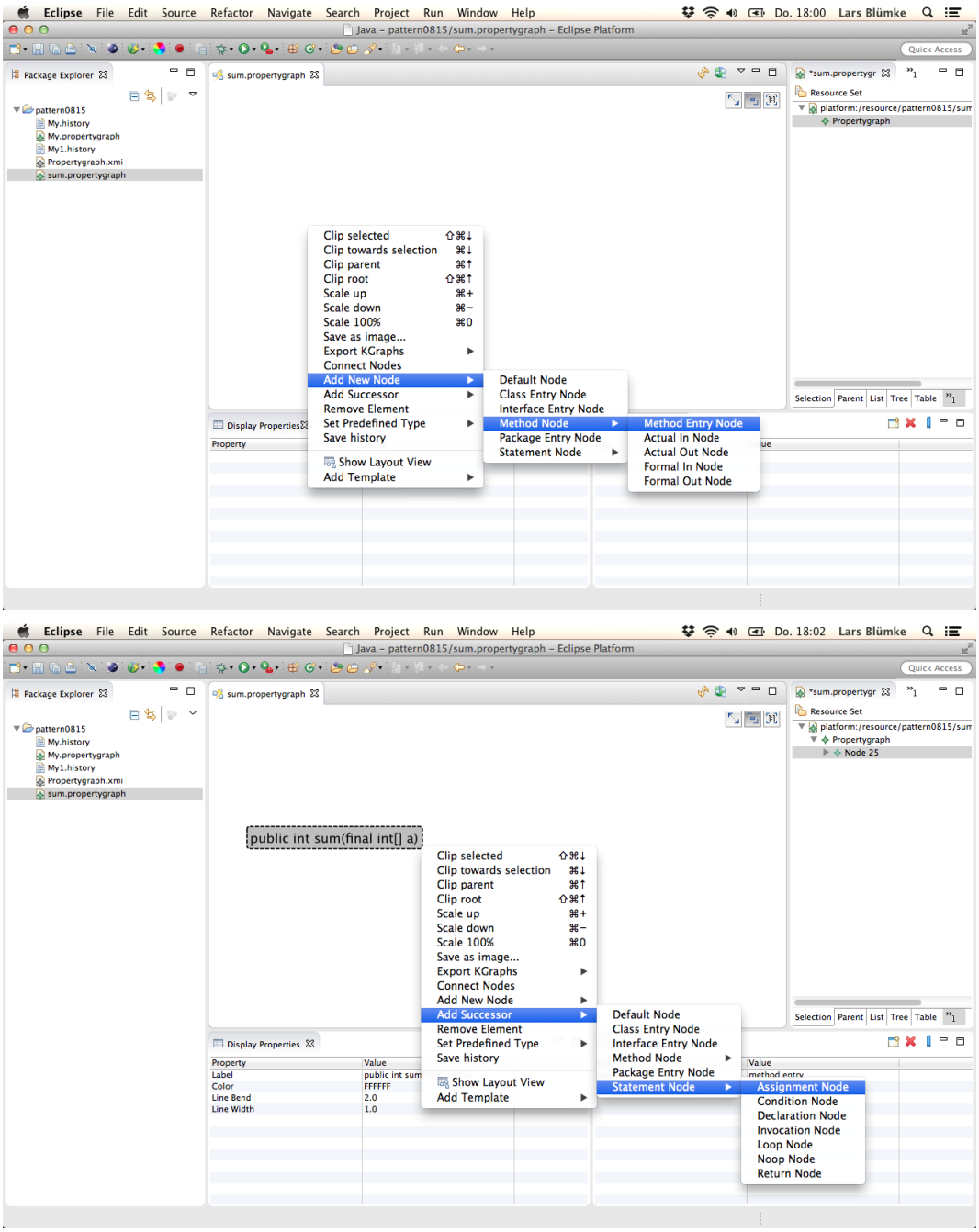

Abbildung 7.1. Erstellen der ersten Knoten des Graphmusters

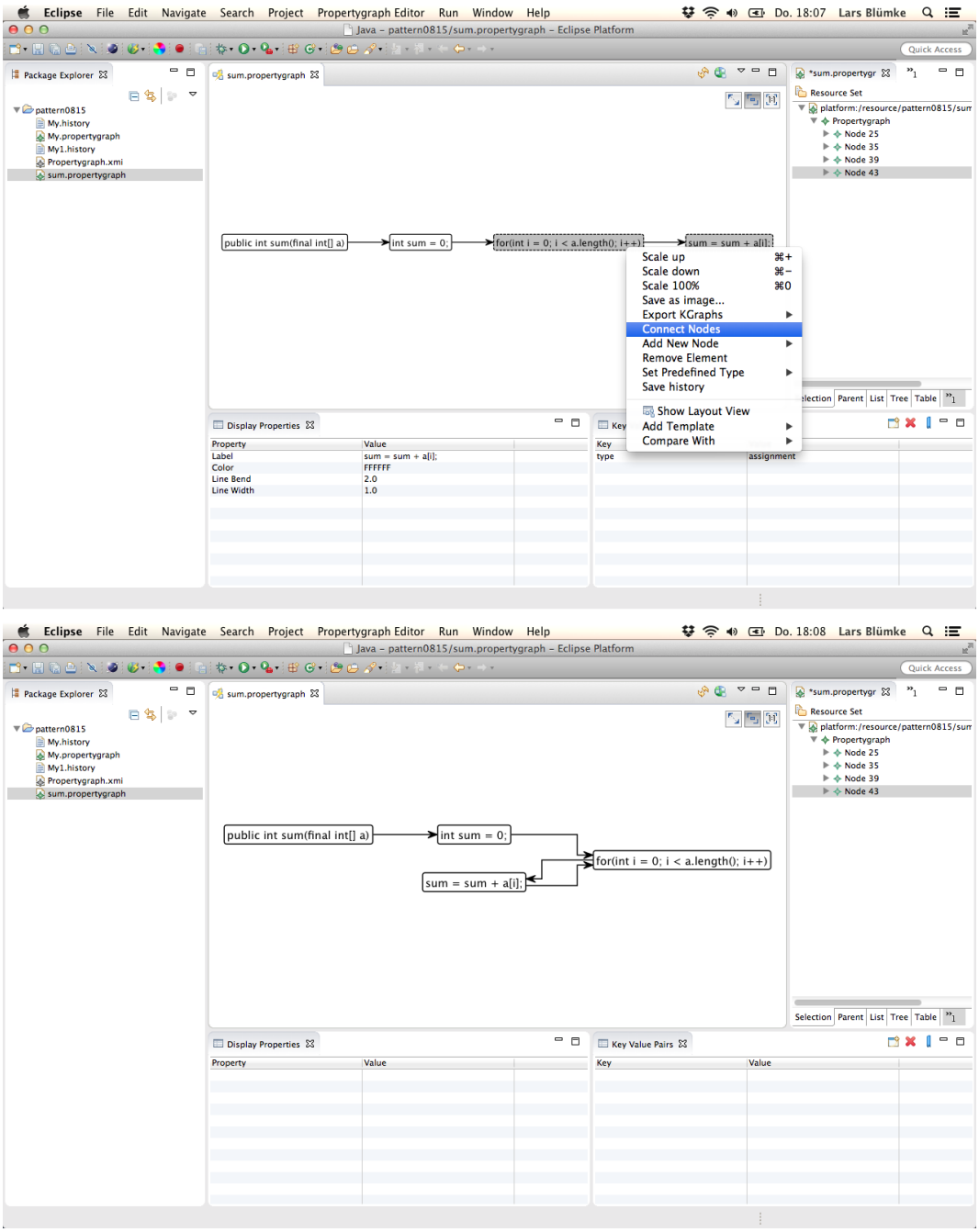

Abbildung 7.2. Verbinden von Schleifenkörper und -definition über die Funktion *Connect Nodes*

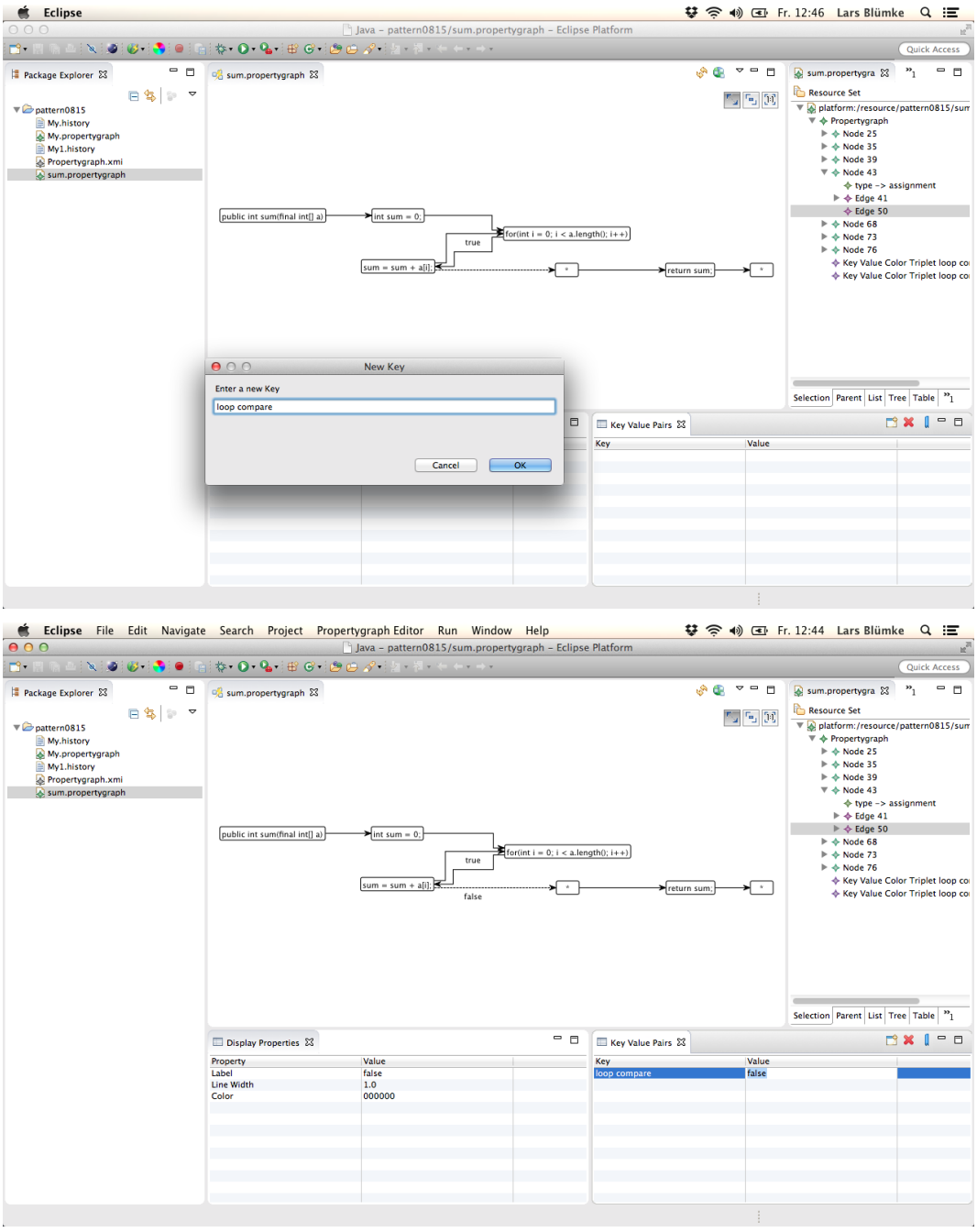

Abbildung 7.3. Definition von Schlüssel-Wert-Paaren

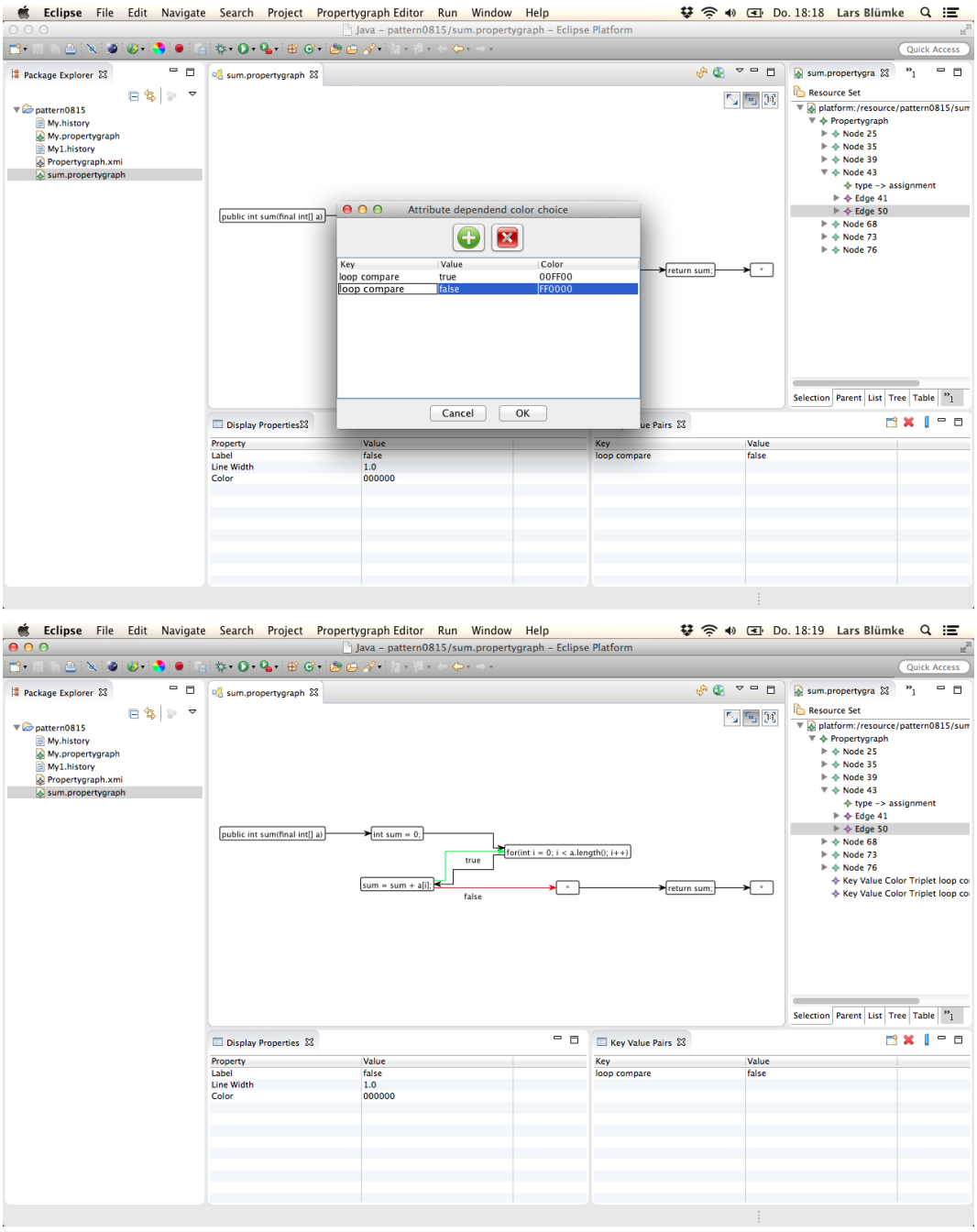

Abbildung 7.4. Zuweisung von Farben zu definierten Schlüssel-Wert-Paaren

7.2. Bewertung ausgewählter Komponenten und mögliche Weiterentwicklungen

7.2.1. Graphmodell

In Kapitel 4 wurde deutlich gemacht, dass es im genutzten Modell keine unterschiedlichen Datentypen für Knoten und Kanten gibt. Dadurch bleiben die unterschiedlichen Typen leicht änderbar, insbesondere ist kein explizites Casting und keine Konvertierung zwischen unterschiedlichen Elementtypen notwendig. Allerdings findet die Unterscheidung lediglich anhand der String Werte der Schlüssel-Wert-Paare statt, was beispielsweise durch Eingabefehler zu Verwechslungen führen könnte. Insgesamt werden dem Nutzer aber durch die gewählte Implementierung mehr Freiheiten gelassen den Graphen auch über die implementierte Definition von Walkinshaw u. a. [2003] hinaus nach den eigenen Ansprüchen zu erweitern.

7.2.2. EMF Tree Editor

Ob der über das EMF generierte Tree Editor in Weiterentwicklungen der Software noch Verwendung finden sollte, ist fraglich. Im Rahmen dieser Arbeit war er ein nützliches Werkzeug, um vor der Fertigstellung des grafischen Editors Graphen zu modifizieren, allerdings ist der grafische Editor nun deutlich funktionaler und besser auf die Arbeit mit dem zugrunde liegenden Graphmodell abgestimmt. Beim Löschen von Knoten über den generierten Tree Editor werden beispielsweise eingehende Kanten nicht mit entfernt, da diese durch die EMF Containment Zuordnung zum Vorgängerknoten gehören. So können Kanten mit nur einem Quellknoten und ohne Endknoten entstehen, was in der Implementierung des grafischen Editors komplett ausgeschlossen wurde.

7.2.3. Key/Value und Display Properties View

In den Tabellen der Key/Value und Display Properties View wäre gegebenenfalls weitere Arbeit zugunsten der Bedienbarkeit sinnvoll. Es könnte die momentane Implementierung in der Schlüssel einmalig angelegt werden und dann nicht mehr geändert werden können, so weiterentwickelt werden, dass wie im Dialogfenster der attributabhängigen Farbauswahl auch Schlüssel nachträglich änderbar sind.
Weiterhin trat bei der Entwicklung das Problem auf, dass vorgenommene Änderungen in der Key/Value und Display Properties View den Zustand der Modellinstanz in Eclipse nicht als geändert kennzeichnen, sodass ein manuelles Speichern nicht möglich ist. Die aktuelle Implementierung löst dieses Problem, indem im Quellcode festgelegt wurde, dass bei jeder Änderung automatisch die aktuelle Graphinstanz abgespeichert wird. In einer Weiterentwicklung könnte das Problem der nicht erkannten Änderungen gegebenenfalls durch die Benutzung des in 6.1.5 genutzen KLighD *IModelModificationHandlers* gelöst werden.

7.2.4. Attributabhängige Farbauswahl

Im Dialogfenster für die attributabhängige Farbauswahl wird momentan nicht zwischen Tripeln für Knoten und Kanten unterschieden. Eine solche Unterscheidung ist für die Funktionalität nicht notwendig, könnte aber bei einer großen Anzahl definierter Tripel in einer Weiterentwicklung die Übersicht erleichtern.
In der aktuellen Version ist es außerdem noch möglich, Tripel mit identischen Werten mehrfach zu definieren oder für die gleiche Schlüssel-Wert-Kombination unterschiedliche Farben festzulegen. Momentan wird beim Vergleich auf mögliche Übereinstimmungen mit den tatsächlichen Schlüssel-Wert-Paaren im Graphen über alle Tripel iteriert, sodass bei mehreren passenden Tripeln die Farbe des letzten passenden Tripels übernommen wird. Eine performantere Implementierung würde nach dem ersten passenden Tripel abbrechen und so die Farbe des ersten passenden Tripels übernehmen. Alternativ könnte bei Konflikten der Benutzer gefragt werden, welche Farbe gewählt werden soll.

Kapitel 8

Fazit und Ausblick

8.1. Fazit

Im Kontext des graphbasierten Parallelisierungsansatzes von Wulf [2014] wurde ein Grapheditor entwickelt, der das Erstellen von Eigenschaftsgraphen mit der Maus ermöglicht. Es wurde auf die Entwicklung des zugrunde liegenden Basismodells und die Transformation zum Grafikmodell mit KLighD eingegangen. Spezifischere Knoten- und Kantentypen von Systemabhängigkeitsgraphen wurden definiert, sodass Programme modelliert werden können. Statische Informationen aus dem Quellcode bestimmen dabei die Struktur des Graphen und dynamische Informationen können zusätzlich als Schlüssel-Wert-Paare im Graphen gespeichert werden. Ausgewählten Schlüssel-Wert-Kombinationen können Farben im Graphen zugeordnet werden. Die Funktionalität des Editors wurde schließlich anhand eines Anwendungsbeispiels evaluiert.

8.2. Ausblick

Zukünftige Arbeiten können den Grapheditor um weitere Funktionen im Kontext des graphbasierten Parallelisierungsansatzes von Wulf [2014] erweitern. Der Editor kann um eine Funktion erweitert werden, um einen aus gegebenem Quellcode generierten SDG importieren zu können. Es können Funktionen entwickelt werden, um definierte Candidatepatterns in einem gegebenen SDG zu erkennen. Aufbauend auf der Arbeit von Benekov [2015] kann dann eine Funktion entwickelt werden, um aufgezeichnete Transformationsschritte automatisiert auf erkannte Muster im SDG anzuwenden. Durch die Entwicklung einer Datenbankanbindung könnten erstellte Candidatepatterns und die zugehörigen Transformationsschritte in einer Datenbank gespeichert werden. Neben der Möglichkeit, einen Graphen mit der Maus zu erstellen, könnte ein textueller Editor entwickelt werden, um Graphmuster zeitsparend, textuell über eine entsprechende domänenspezifische Sprache zu definieren. Gegebenenfalls könnte hierzu auch direkt eine durch das EMF gegebene XML-Repräsentation des Basismodells editiert werden.
Durch die Implementierung mit KIELER und KLighD kann der Editor um hierarchische Knoten erweitert werden, sodass zum Beispiel ganze Schleifen oder Methoden im Graphen auf- und zugeklappt werden können.

Literaturverzeichnis

[Benekov 2015] Y. Benekov. Konzeption und Implementierung eines Eclipse-Plugins zur Erstellung und Verwaltung von Graphen. Bachelorarbeit. Christian-Albrechts-Universität zu Kiel, Institut für Informatik. 2015. (Siehe Seiten 2, 18, 21 und 49)

[Ottenstein und Ottenstein 1984] K. J. Ottenstein und L. M. Ottenstein. The program dependence graph in a software development environment. In: *SDE 1 Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*. New York, NY, USA, 1984. (Siehe Seite 6)

[Reps und Binkley 1990] S. H. T. Reps und D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12 (1990), Seiten 26–60. (Siehe Seite 6)

[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 (2010), Seiten 35–41. (Siehe Seite 5)

[Schneider u. a. 2012] C. Schneider, M. Spönemann und R. von Hanxleden. Transient View Generation in Eclipse. Technischer Bericht. Christian-Albrechts-Universität zu Kiel, Institut für Informatik, 2012. (Siehe Seite 12)

[Schneider u. a. 2013] C. Schneider, M. Spönemann und R. von Hanxleden. Just Model! – Putting Automatic Synthesis of Node-Link-Diagrams into Practice. In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'13)*. San Jose, CA, USA, 2013. (Siehe Seite 12)

[Schulze 2015] C. D. Schulze. KLighD Tutorial. Last visited August 27, 2015. 2015. URL: http://rtsys.informatik.uni-kiel.de/confluence/pages/viewpage.action?pageId=10751615. (Siehe Seite 23)

[Steinberg u. a. 2008] D. Steinberg, F. Budinsky, M. Paternostro und E. Merks. EMF: Eclipse Modeling Framework. Second Edition. Addison-Wesley Professional, 2008. (Siehe Seiten 10, 11)

[Steppan 2015] B. Steppan. Eclipse Rich Clients und Plug-ins: Modulare Desktop-Anwendungen mit Java entwickeln. Carl Hanser Verlag GmbH & Co. KG, 2015. (Siehe Seiten 9, 10)

[Walkinshaw u. a. 2003] N. Walkinshaw, M. Roper und M. Wood. The Java System Dependence Graph. In: *Third IEEE International Workshop on Source Code Analysis and Manipulation*. Amsterdam, Niederlande, 2003. (Siehe Seiten 6, 7 und 47)

[Wulf 2014] C. Wulf. Pattern-based detection and utilization of potential parallelism in software systems. In: *Proceedings of the Software Engineering*. Doctoral Symposium. Kiel, 2014. (Siehe Seiten 1, 2 und 49)