Graphische Visualisierung von Java-Variablen zur Laufzeit

Heiko Wissmann

Bachelor-Arbeit
eingereicht im Jahr 2013

Christian-Albrechts-Universität zu Kiel
Arbeitsgruppe Echtzeitsysteme und eingebettete Systeme

Betreut durch: Dipl.-Inf. Miro Spoenemann

Eidesstattliche Erklärung

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

Kiel,

_____

Zusammenfassung

Debuggen ist ein wichtiger Teil während der Entwicklung bzw. Verbesserung von Software. Allerdings kann dieser Teil auch sehr viel Zeit in Anspruch nehmen. Um diesen Zeitaufwand zu verringern, ist es wichtig schnell und effizient Fehler in Datenstrukturen und deren Inhalt zu finden. Eine graphische Visualisierung kann dabei helfen, allerdings nur, wenn der Benutzer selber festlegen kann, welche Informationen ausgewählt und wie diese dargestellt werden. Ziel dieses Projektes war es eine Schnittstelle für eine graphische Visualisierung von Java Variablen zur Laufzeit zu entwickeln. Durch Modelltransformationen hat der Benutzer die volle Kontrolle über Inhalt und Erscheinung der Visualisierung. Die Entwicklung einer generischen Transformation für Datenstrukturen, für die es keine spezielle Transformation gibt, und einer Transformation von Arrays war ebenfalls Bestandteil dieses Projekts. Zusätzlich wurden Transformationen zu häufig genutzten Datenstrukturen aus den Paketen `java.lang` und `java.util` entwickelt.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Das schnelle Finden und Beheben von Fehlern während der Entwicklung neuer Software oder der Erweiterung und Verbesserung schon bestehender Software ist sehr wichtig, um mehr Zeit in die Entwicklung neuer Konzepte stecken zu können. Um Fehler innerhalb genutzter Datenstrukturen zu erkennen, müssen diese zur Laufzeit eines Programms analysiert werden. Um Datenstrukturen zur Laufzeit analysieren zu können, müssen Informationen, die in diesen Strukturen enthalten sind, auf eine Art und Weise dargestellt bzw. visualisiert werden, mit der ein Entwickler schnell und effektiv arbeiten kann.

Ohne die Verwendung externer Systeme besteht diese Darstellung oder auch *Visualisierungen* nur aus einer Konsolenausgabe des Wertes einer Variablen durch Aufruf einer Ausgabemethode, z.B. `printf` (C), `println` (Pascal) oder `System.out.println` (Java), an einer Stelle im Quelltext, an der der Wert der Variablen von Bedeutung ist. Zusätzliche Programme zur Analyse waren nicht notwendig, allerdings ist diese Methode der Visualisierung bei komplexen Programmen sehr umständlich und Zeitaufwendig.

Es gibt Systeme, so genannte *Debugger*, die Entwicklern bei der Analyse durch verschiedenste Arten der Visualisierung helfen. Die ersten Debugger wurden noch über die Kommandozeile gesteuert und Informationen zu den Datenstrukturen wurden auf der Konsole als Text ausgegeben. Heutzutage haben diese Systeme oft eine graphische Benutzeroberfläche (*GUI*), mit der die Informationen dargestellt werden. Durch die GUI ist der Benutzer in der Lage die Daten ohne Eingabe von Befehlen in Textform zu analysieren. Debugger haben außerdem den Vorteil, dass kein weiterer Quelltext hinzugefügt und wieder entfernt werden muss, so dass kein weiterer Programmier- und Zeitaufwand entsteht. Die meisten Entwicklungsumgebungen haben bereits einen Debugger integriert.

1.1 Problemstellung

Eine häufig genutzte Form der Visualisierung für Datenstrukturen in einer GUI ist ein Baum, in dem die Werte abgelesen werden können. Die oberste Hierarchieebene beinhaltet eine Variable. Sollte diese Variable ein Objekt sein und weitere Variablen haben, so kann diese Ebene erweitert bzw reduziert werden, so dass Felder der Variablen analysiert werden können. Abbildung 1 zeigt die Darstellung einer `LinkedList` mit drei Elementen. Der Baum ist so weit erweitert, dass alle Elemente der Liste erkennbar sind.

Diese Darstellungsart weist allerdings Schwächen auf, die dazu führen, dass enthaltene Informationen nur schwer oder gar nicht zu finden sind. So kann das wiederholte Auftreten desselben Objektes bei komplexen Datenstrukturen nur mit Mühe erkannt werden, da dieses Objekt auf mehreren

1.1 Problemstellung

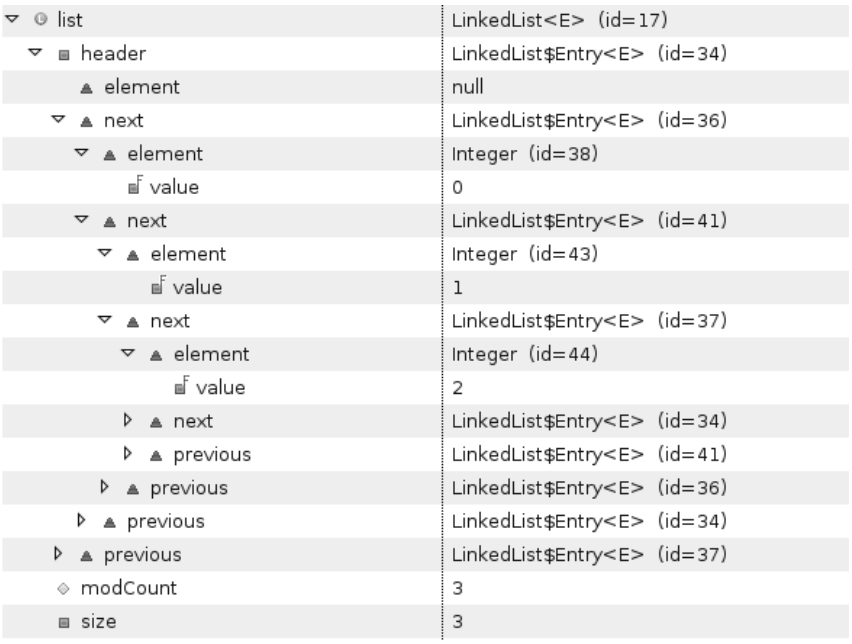


Abbildung 1. Darstellung einer `LinkedList` in Eclipse

Hierarchieebenen verwendet werden kann. Zudem enthält der Baum alle Daten ungefiltert, auch die, die für Entwickler irrelevant sind, wie z.B. Daten die für die Funktionalität von Datenstrukturen wichtig sind, wie etwa `loadFactor` oder `keySet` der Datenstruktur `HashMap`. Beide Variablen haben keinen Einfluss auf die Struktur der `HashMap` und werden, ausser bei der Entwicklung neuer Datenstrukturen ohne Bedeutung sein. Diese Schwächen können durch die Verwendung eines Graphen als Visualisierung, bei der der Benutzer festlegen kann, welche Informationen ausgewählt und auf welche Weise diese dargestellt werden, behoben werden. Die Graphen können je nach Datenstruktur und den Anforderungen des Benutzers verschiedene Informationen enthalten und heben diese hervor. Irrelevante Daten können komplett aus der Visualisierung entfernt werden, wodurch die Verständlichkeit der Visualisierung erhöht wird. So hat die Information welchen Wert die Variable `modCount` in der `LinkedList` in den meisten Fällen keinen Nutzen für den Entwickler.

Ziel dieses Projektes war die Entwicklung eines *Eclipse* basierten *Plug-ins*, mit dessen Hilfe Datenstrukturen, die im *Debugmodus* ausgewählt werden, zu einem Graph transformiert werden können, so dass Informationen, die in der Datenstruktur enthalten sind, visualisiert werden. Als Basis dieses Projektes dient das von der Arbeitsgruppe Echtzeitsysteme und eingebettete Systeme

entwickelte Plug-in *KLighD*, mit dessen Hilfe modellbasierte, leichtgewichtige Visualisierungen möglich sind. Zusätzlich wurden beispielhaft Transformationen zu häufig genutzten Datenstrukturen entwickelt.

1.2 Visualisierung

Nach Gershon et al. ist eine Visualisierung die Transformation von Daten, Informationen und Wissen in eine visuelle Form. Mit effektiven visuellen Schnittstellen können große Mengen an Daten schnell und effektiv analysiert werden um versteckte Charakteristiken, Muster und Tendenzen zu entdecken [2].

Diese grundlegende Definition kann je nach Anwendungsbereich der Visualisierungen verfeinert und erweitert werden. So ist die *Softwarevisualisierung* nach Price et al. die Verwendung von interaktiven Computergraphiken, Typographie, Graphikdesign, Animationen und Kinematographie zur Verbesserung der Schnittstelle zwischen Softwareentwickler oder Informatikstudenten und deren Programmen [11]. Myers teilt die Softwarevisualisierung in Teilvisualisierungen ein, je nachdem ob Code, Daten oder Algorithmen visualisiert werden oder ob die Visualisierung statisch oder dynamisch ist [8]. Eine statische Visualisierung stellt den aktuellen Zustand der Information dar. Eine statische Codevisualisierung ist z.B. die Anzeige des Quelltextes in einer Entwicklungsumgebung, die durch Syntaxhervorhebung die Lesbarkeit verbessert. Eine Form der dynamischen Visualisierung ist die Animation, durch die der Fokus auf die Änderung zwischen verschiedenen Zuständen gesetzt wird. Dies verbessert die Erkennbarkeit von fehlerhaften Zustandsänderungen. Bei der Codevisualisierung wird meist nicht die Änderung des Quelltextes visualisiert, sondern an welcher Stelle im Code sich die Ausführung des Programms befindet.

Neben der Klassifikation der Visualisierung sind die Fragen, welche Informationen visualisiert werden und auf welche Weise das geschieht, von elementarer Bedeutung. Verschiedene Formen der Visualisierung können Informationen, je nach Art der Information und Art der Fragestellung, die durch die Visualisierung beantwortet werden soll, besser oder schlechter verdeutlichen. Die Form des Baumes beispielsweise hat den Vorteil, dass eine Hierarchie klar erkennbar ist. Ein großer Nachteil besteht jedoch darin, dass sie keine Referenzen auf schon bestehende Knoten zulässt, so dass zwei Knoten, die dieselbe Information enthalten und auf verschiedenen Ebenen sind, nicht auf einen Knoten reduziert werden können. Somit können Informationen verloren gehen oder je nach Beschriftung nur schwer erkennbar sein. Die Form des Graphen lässt Referenzen auf schon bestehende Knoten zu, allerdings kann ein Graph mit zu vielen Knoten und Referenzen sehr schnell unübersichtlich werden.

Die in dieser Arbeit behandelte Visualisierung ist eine statische Datenvisualisierung in Form eines Graphen.

1.3 Debuggen

Jeder Programmierer schreibt Programme, die Fehler enthalten. Das erste Anzeichen dafür, dass ein Programm nicht korrekt ist, ist normalerweise ein nach außen sichtbares Symptom wie etwa eine falsche Ausgabe. Die Ursache dieses Symptoms ist ein fehlerhafter Ausgangszustand des Programms, auch Fehler genannt. Dieser Fehler wird meist von einem anderen Fehler ausgelöst. Diese Kette von Fehlern kann bis zu seinem Ursprung zurückverfolgt werden. Die Ursache des initialen Fehlers ist ein Defekt im Programm. Dieser Defekt wird *Bug* genannt [7].

Debuggen nennt man das Suchen und Beheben von Bugs. Es gibt Projekte bei denen das Debuggen bis zu 50 Prozent der Entwicklungszeit in Anspruch nimmt [4]. Um diese Zeit zu verkürzen ist es wichtig die Suche nach Bugs zu beschleunigen. Ein wichtiger Teil des Debuggens ist die Analyse der im Programm verwendeten Daten zur Laufzeit, um die Fehlerkette zurückzuverfolgen.

Software, mit deren Hilfe gedebuggt werden kann, nennt man Debugger. Debugger visualisieren Informationen, die dabei helfen, Fehler im Programm zu finden und zu beheben. Eclipse, die für dieses Projekt verwendete Entwicklungsumgebung, beinhaltet bereits einen Debugger. Ein Teil des Debuggers bildet eine Grundlage dieses Projekts.

1.4 Modell und Modelltransformation

Ein *Modell* kann nach Herbert Stachowiak durch drei Merkmale gekennzeichnet werden [12]. Nach dem Abbildungsmerkmal ist ein Modell eine Abbildung oder Repräsentation eines natürlichen oder eines künstlichen Originals, das auch ein Modell sein kann. Das Verkürzungsmerkmal verdeutlicht, dass Aspekte des Originals, die dem Modellschaffer bzw. Modellnutzer wichtig erscheinen, hervorgehoben werden und insbesondere unwichtige Details in den Hintergrund treten oder ganz verschwinden. Dass Modelle nicht eindeutig ihrem Original zugeordnet werden können, beschreibt das pragmatische Merkmal. Ein weiteres Merkmal besagt, dass Modelle nur für bestimmte Modellbetrachter, in einer bestimmten Zeit und unter Einschränkung auf bestimmte gedankliche oder tätliche Operationen das Original ersetzt. Ein Modell kann also durch die Fragen für wen, für wann und wozu charakterisiert werden.

Metamodelle sind Modelle, die die Struktur anderer Modelle beschreiben. Modelle lassen sich also gruppieren, wenn sie demselben Metamodell entsprechen.

Der Begriff *Modelltransformation* oder auch *Modell-zu-Modell-Transformation* stammt aus der modellgetriebenen Softwareentwicklung und wird als eine (automatisierte) Überführung von einem Quellmodell in ein Zielmodell verstanden, die nach einer Menge von Transformationsregeln durchgeführt wird. Diese Regeln beschreiben, wie Elemente aus dem Quellmodell in Elemente des Zielmodells transformiert werden [6]. Sie dient dazu wichtige Informationen des Quellmodells im Zielmodell hervorzuheben und irrelevante Daten in den Hintergrund rücken zu lassen.

In diesem Projekt dienen Modelltransformationen dazu, dem Benutzer die Entscheidung, welche Informationen wie dargestellt werden, zu überlassen. Die einzige Einschränkung liegt in der Wahl des Quell- und des Zielmodells. Das Quellmodell ist ein Modell, das die Struktur und den Inhalt einer Variablen beschreibt, und das Zielmodell ist ein Modell eines Graphen.

1.5 Aufbau dieser Arbeit

Im anschließenden Kapitel werden zunächst drei verwandte Arbeiten, die Ähnlichkeiten mit diesem Projekt aufweisen, vorgestellt und deren Vor- und Nachteile ausgearbeitet. Kapitel 3 befasst sich dann mit den in diesem Projekt verwendeten Technologien. In Kapitel 4 werden die Konzepte, die in diesem Projekt erarbeitet worden sind, und deren Implementierungen erläutert. In Kapitel 5 werden dann zunächst zwei generische und dann einige spezielle Transformationen vorgestellt. Zum Abschluss werden Erweiterungsmöglichkeiten diskutiert und ein Fazit gezogen.

2 Verwandte Arbeiten

Das Thema Softwarevisualisierung ist alles andere als neu. Schon 1948 verwenden Goldstine et al. Flowcharts zur Visualisierung von Programmen [3]. Myers hat versucht die Begriffe Programmvisualisierung und visuelle Programmierung einzugrenzen und sie zu klassifizieren [8]. Programmvisualisierung wurde in die Klassen Quelltext-, Daten- oder Algorithmusvisualisierung, also „was“, und statische oder dynamische Visualisierung, also „wie“, eingeteilt. Zwei Jahre später haben Price et al. diese Klassifikation um vier weitere Klassen erweitert, nämlich grundlegende Eigenschaften (z.B. Skalierbarkeit), Spezifikation (z.B. Anpassbarkeit), Interaktionen (z.B. Navigation) und Effektivität [11]. Diese Klassen werden für die Beurteilung von Visualisierungssystemen verwendet.

Neben den theoretischen Arbeiten existieren schon zahlreiche Projekte, die sich mit dem Thema Programm- und Datenvisualisierung beschäftigt haben. Diese haben verschiedene Zielgruppen und Anwendungsgebiete. Im folgenden werden drei verwandte Projekte vorgestellt, die Visualisierungen für Datenstrukturen bereitstellen. Bei keinem dieser Projekte kann der Benutzer vollständig bestimmen, welche und auf welche Weise Informationen von Datenstrukturen dargestellt werden.

2.1 LJV

Der *Lightweight Java Visualizer (LJV)* [5] ist eine Bibliothek zur graphischen Darstellung von Datenstrukturen und verwendet GraphViz[1] für die Grapherzeugung. Abbildung 2 zeigt eine häufig genutzte Datenstruktur, die `LinkedList`, gefüllt mit drei Werten vom Typ `Integer`.

Zielgruppe dieses Werkzeugs sind Informatikstudenten der ersten Semester, also unerfahrene Entwickler, die zunächst die Grundlagen der objektorientierten Entwicklung lernen müssen. Aus diesem Grund ist dieses Werkzeug sehr einfach einzubinden. Es muss lediglich eine Javaklasse eingebunden und eine statische Methode, mit dem darzustellenden Objekt und einem optionalen Kontext, mit dessen Hilfe Einstellungen vorgenommen werden können, aufgerufen werden. Da der vorhandene Code manipuliert werden muss, wodurch dieses Werkzeug nur für kleine Programme zu gebrauchen ist, werden die zu visualisierenden Informationen direkt aus dem übergebenen Objekt, ohne die Verwendung eines Debuggers, gewonnen.

Bei komplexeren Datenstrukturen werden nicht alle vorhandenen Informationen visualisiert und es ist nur begrenzt möglich zu entscheiden, welche Informationen dargestellt oder nicht dargestellt werden sollen, so dass die

[1] http://www.graphviz.org

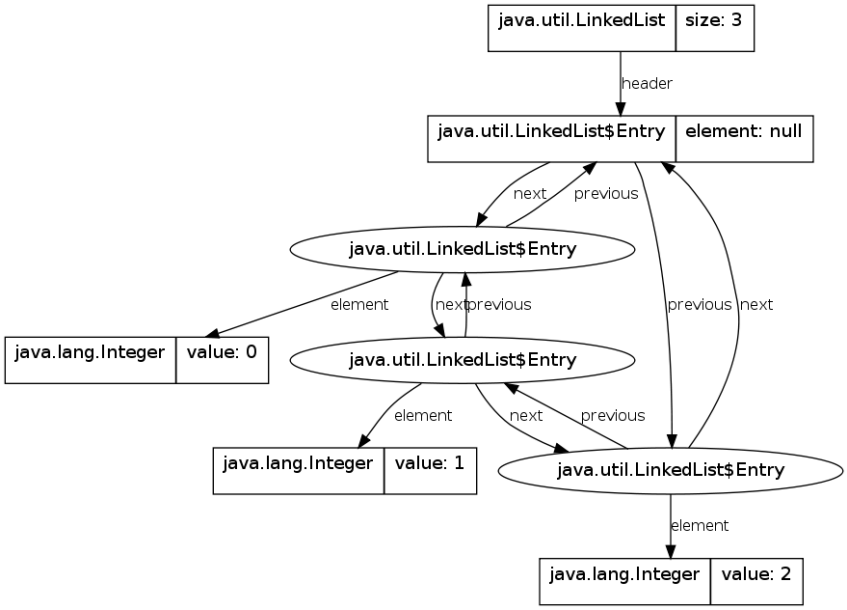


Abbildung 2. Darstellung einer `LinkedList` mit LJV

Visualisierung von komplexen Datenstrukturen schnell unübersichtlich werden kann. Abbildung 3 zeigt eine `WeakHashMap`, eine Standarddatenstruktur, bei der ein Wert (`value` Feld der `WeakHashMap`) einem Schlüssel (`referent` Feld der `WeakHashMap`) zugeordnet wird. Bei genauerer Betrachtung fällt auf, dass der Schlüssel nicht dargestellt wird, zudem enthält sie unter Umständen irrelevante Informationen. Eine Besonderheit bildet der Wert `null`, der, obwohl er ein Objekt repräsentiert, von LJV als primitiver Wert gehandhabt wird.

2.2 Javavis

Javavis [10] ist eine Programmvisualisierungssoftware, die zwei UML Diagramme, Objekt- und Sequenzdiagramme, zur Visualisierung und das *Java Debug Interface* verwendet, um laufende Java Programme zu kontrollieren und um Informationen, wie z.B. den Wert einer Variablen, zu erhalten. Es ist eine eigenständige Software und muss mit dem Java Programm als Parameter aufgerufen werden, der vorhandene Code des Programms muss also nicht manipuliert werden.

Wie bei LJV (Abschnitt 2.1) werden Datenstrukturen ungefiltert in Form eines Objektdiagramms dargestellt und können irrelevante Informationen enthalten, wodurch eine Hervorhebung relevanter Informationen nicht möglich

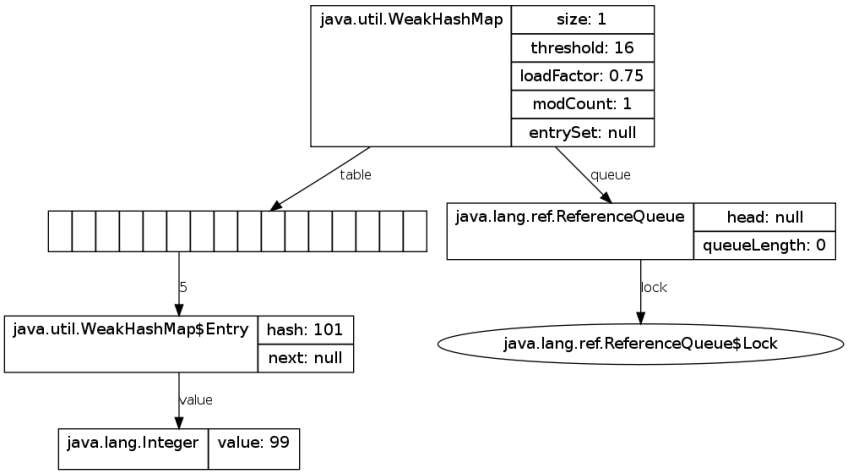


Abbildung 3. Darstellung einer WeakHashMap mit LJV

ist. Zusätzlich wird der Ablauf eines Programms in einem Sequenzdiagramm dargestellt. Abbildung 4 zeigt das Objektdiagramm während der Ausführung des Programms, welches eine einfach verkettete Liste mit Werten füllt.

Dank der implementierten Funktion, ein Java Programm Schritt für Schritt bzw. Methodenaufruf für Methodenaufruf zu durchlaufen, kann die Veränderung des Programmablaufs und des Inhalts von Datenstrukturen durch einen flüssigen Übergang zwischen den Zuständen visualisiert werden.

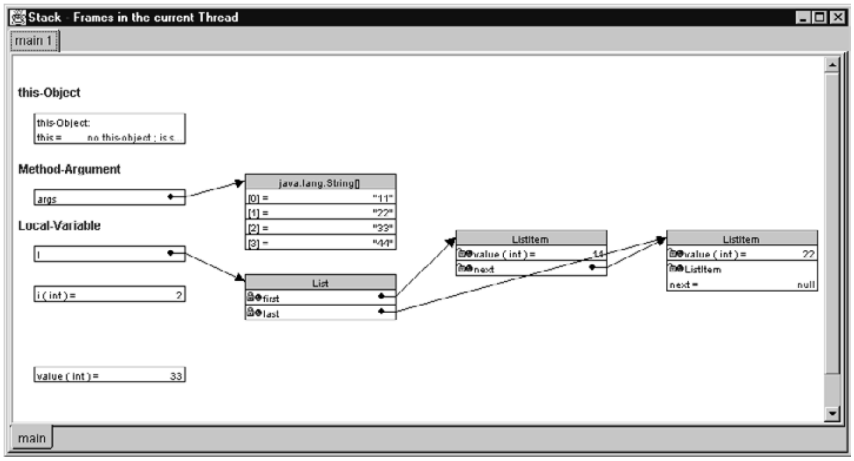


Abbildung 4. Objektdiagramm eines laufenden Java Programms in Javavis [10]

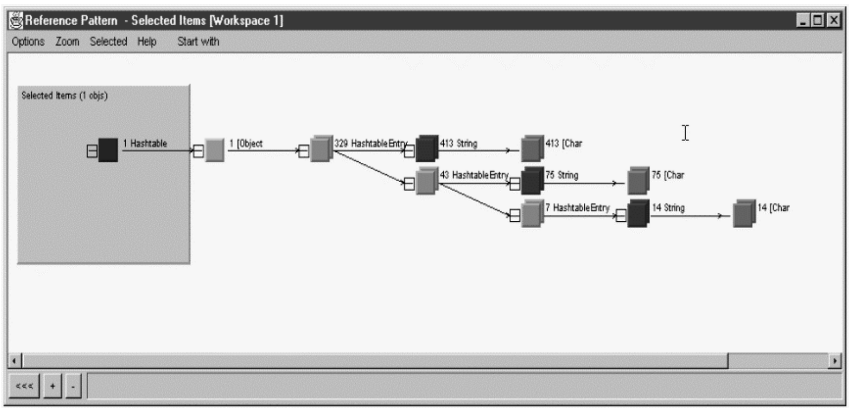


Abbildung 5. Visualisierung einer HashTable in Jinsight [1]

Anwendungsgebiet dieser Software ist auch bei Javavis die Lehre. Studenten sollen mit Hilfe der Software Abläufe im Programm und Veränderungen von Variablen schnell erkennen und verstehen können.

2.3 Jinsight

Jinsight [1] ist ein Werkzeug von IBM, das die Entwicklung bzw. Erweiterung von Software unterstützen soll. Genau wie mit Javavis (Unterkapitel 2.2) können Datenstrukturen und Programmabläufe visualisiert werden, zusätzlich kann die Resourcennutzung, also CPU- und Speicherauslastung, dargestellt werden. Damit kann ein Programm hinsichtlich Performance und Speicherlecks optimiert werden.

Für Datenstrukturen wird die so genannte *Reference Pattern View* verwendet. Anstatt jede Instanz innerhalb der Datenstruktur darzustellen, werden diese nach Typ und Referenz gruppiert. Dadurch wird die Komplexität des Baumes reduziert und die Struktur der Datenstruktur wird gegenüber dem Inhalt hervorgehoben. Der Benutzer hat zusätzlich durch einen Reduzierungs- bzw. Erweiterungsknopf an den einzelnen Gruppen die Möglichkeit die Komplexität des Baumes zu verringern oder zu vergrößern. Abbildung 5 zeigt eine Variable vom Typ HashTable mit über 1000 Elementen. Diese Art der Komplexitätsreduzierung sorgt für Übersicht auf Kosten von unter Umständen wichtigen Details.

3 Verwendete Technologien

In diesem Kapitel werden die wichtigsten Technologien vorgestellt, die in diesem Projekt zum Einsatz kamen.

3.1 Eclipse

Eclipse[2] ist bekannt als eine quelloffene integrierte Entwicklungsumgebung (IDE) für Java. Im Gegensatz zu vielen anderen Entwicklungsumgebungen ist Eclipse nicht auf Java beschränkt, sondern kann durch *Plug-ins* erweitert werden, so dass Eclipse für viele Programmier- und Modellierungssprachen wie z.B. C++, Unified Modeling Language (UML), Extensible Markup Language (XML) usw. verwendet werden kann.

3.1.1 Plug-in Mechanismus

Ein Plug-in ist die kleinste Einheit einer Funktion der Eclipse Plattform, die entwickelt und unabhängig ausgeliefert werden kann [9]. Üblicherweise sind kleine Werkzeuge einzelne Plug-ins wohingegen komplexere Werkzeuge ihr Funktionalität auf mehrere Plug-ins verteilen. Bis auf einen kleinen Kern, der

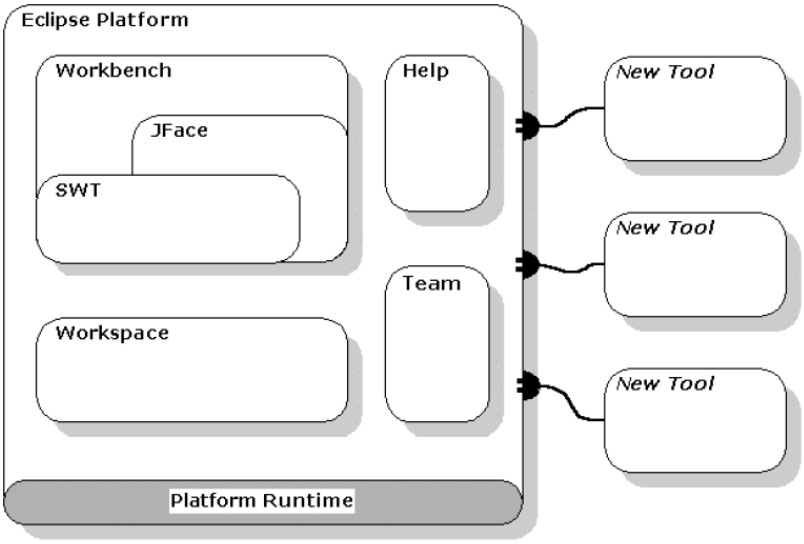


Abbildung 6. Plug-in Architektur von Eclipse [9]

[2]http://www.eclipse.org/org/

Platform Runtime, beruht die gesamte Funktionalität der Eclipse Plattform auf Plug-ins. Abbildung 6 zeigt die Plug-in Architektur von Eclipse.

Damit Plug-ins untereinander kommunizieren können, gibt es die *Extension Points* und die *Extensions*. Ein Extension Point spezifiziert welche Klassen implementiert und welche Optionen gesetzt werden können bzw müssen. Beim Hinzufügen einer Extension werden dann die verwendeten Klassen und Optionen festgelegt. Dieser Mechanismus funktioniert wie ein Puzzle. Extension Points entsprechen dabei den Einkerbungen und die Extensions den Ausbuchtungen, wobei es für eine Einkerbung mehrere Ausbuchtungen geben kann.

Für die GUI ist die sogenannte *Workbench* verantwortlich. Die Oberfläche besteht unter Anderem aus *Editoren*, mit dem der Inhalt von Dateien bearbeitet werden kann, und den *Views*, die zusätzliche Informationen darstellen, aber keine Veränderungen an Dateien zulassen (siehe Abbildung 7). Editoren und Views sind *Parts* eines Fensters der Workbench. Eine Standardview ist die *Variable View*, die die Struktur und den Inhalt von Variablen zur Laufzeit darstellt (siehe Abbildung 1. Für Editoren und Views gibt es auch Extension Points, so dass die Benutzeröberfläche erweitert werden kann.

3.1.2 Xtend

Xtend[3] ist eine Programmiersprache, die huntertprozentig kompatibel mit Java ist. Statt zu Byte-Code compilert zu werden, wird Java Code generiert. Viele Aspekte von Java werden durch Xtend verbessert. Unter anderem ist es möglich Klassen um Methoden zu erweitern ohne den Code der Klasse zu verändern. Diese Methoden werden *Extension Methoden* genannt. Daher hat Xtend auch seinen Namen. Extension Methoden beruhen auf einem syntaktischen Trick. Anstatt das Argument einer Methode beim Aufruf in Klammern zu setzten, wird das Argument so verwendet als wenn die Methode ein Teil des Argumentes ist. So sieht der normale Methodenaufruf

```
toFirstUper("hello")
```

in Xtend so aus

```
"hello".toFirstUper()
```

Dadurch wird in Xtend geschriebener Code sehr viel leserlicher und die Programme laufen genauso schnell, meist sogar schneller, gegenüber dem Java Equivalent. Die Xtend Bibliotek liefert eine sehr große Anzahl an vordefinierten Extension Methoden zu vorhandenen Java Klassen. Unter anderem den Operator =>, der es ermöglicht ein Objekt weiter zu verwenden, ohne dieses in einer Variablen speichern zu müssen.

[3] http://www.eclipse.org/xtend/documentation.html

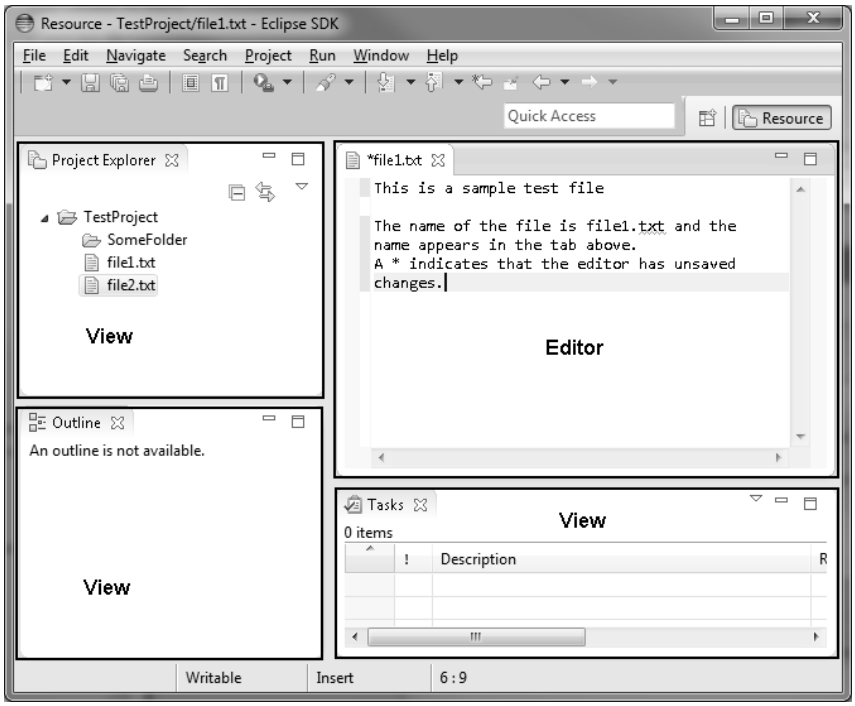


Abbildung 7. Benutzeroberfläche von Eclipse

```
val list = new LinkedList<Integer>();
list.add(1);
list.add(2);
```

wird zu

```
new LinkedList<Integer>() => [
        it.add(1)
        it.add(2)
]
```

`it` repräsentiert dabei das jeweilige Objekt in dem Beispiel also die Liste. Andere nützliche Methoden erweitern Klassen, die das Interface Iterable implementieren, z.B. Listen oder Arrays. Methoden wie `filter` oder `forEach` verwenden Funktionen als Parameter, die aus der funktionalen Programmierung bekannten Lambda Ausdrücke.

Der Programmabschnitt

```
list.filter[element | element % 2 == 0].forEach[
        element |
        println(element)
]
```

gibt alle Elemente von `list`, die einen geraden Wert haben, auf der Konsole aus. Sollte das Element nicht explizit benannt werden, repräsentiert `it` das Element. Sowohl explizite Variablentypen als auch Methodenklammern beim Aufruf von Methoden ohne Parameter sind optional. Dies kann die Lesbarkeit von weniger komplexem Quelltext verbessern, aber auch wegen fehlender Typisierung die Fehlersuche erschweren.

Alle in diesem Projekt entwickelten Modelltransformationen sind in Xtend geschrieben.

3.1.3 Java Development Tools Debug

Das *Java Development Tools* (JDT)[4] Projekt stellt Plug-ins zur Verfügung, durch die Eclipse zu einer Entwicklungsumgebung für Java und damit auch für Eclipse Plug-ins wird. Ein Teil dieses Projektes ist *JDT Debug*. JDT Debug implementiert das Interface, das durch die sprachenunabhängige Debugkomponente der Plattform (*Platform Debug*)[5] spezifiziert wird, für Java.

Abbildung 8 zeigt ein sehr reduziertes Klassendiagramm mit den verwendeten Interfaces von Platform Debug, zusätzlich sind für `IVariable` und `IValue` die konkreten Klassen von JDT im Diagramm enthalten.

Jeder Eintrag in der Variable View von Eclipse ist vom Typ `JDIVariable` und jede Variable hat einen Namen, einen Typ und einen Wert. Zusätzlich enthält eine `JDIVariable` weitere Informationen, z.B. die Sichtbarkeit einer Variablen, die in diesem Projekt allerdings nicht verwendet werden. Der Typ entspricht dabei dem Typ, der bei der Variablendefinition festgelegt wurde, und nicht dem Typ des konkreten Wertes, der in dieser Variablen gespeichert ist.

Alle Werte sind vom Typ `JDINullValue`, `JDIObjectValue`, `JDIPrimitiveValue` oder `JDIVoidValue`, welche die Klasse `JDIValue` erweitern. Neben dem Typ und einer Zeichenkette des gespeicherten Wertes, kann ein Wert wieder Variablen enthalten. Dies sind Felder der Klasse, die der Typ des Wertes ist.

In der Variable View ist diese Struktur in Form eines Baumes dargestellt. Die Wurzel ist der Eintrag auf der obersten Ebene. Die Variablen eines Wertes einer Variablen sind Folgeknoten. Primitive Werte, `null` oder Objekte, die keine Felder besitzen, sind die Blätter des Baumes. Es ist möglich, die Darstellung

[4] http://projects.eclipse.org/projects/eclipse.jdt.debug
[5] http://www.eclipse.org/eclipse/debug/index.php

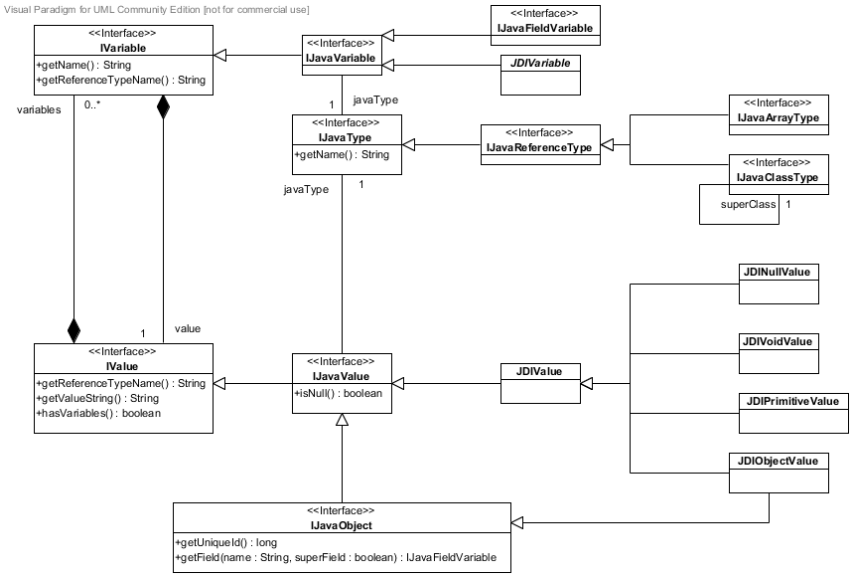


Abbildung 8. Klassendiagramm der verwendeten Elemente von Platform Debug und JDT

der Ebenen zu erweitern bzw. zu reduzieren, so dass die Komplexität der Darstellung begrenzt werden kann.

Die Verwendung der konkreten Klassen von JDT ist von Eclipse leider als `deprecated` also missbilligt eingestuft. Aus diesem Grund werden in diesem Projekt nur die jeweiligen Schnittstellen verwendet, die Zugriff auf die erforderlichen Daten gewähren. Dies ist für Variablen das Interface `IVariable` und für Werte das Interface `IValue`

Auch die Typen von Variablen und Werten haben ein eigenes Interface, durch das auf Informationen zu den Typen zugegriffen werden kann. `IJavaType` bildet dabei das Hauptinterface, das eine Methode zur Verfügung stellt, die den Namen des Typs zürückliefert. Das Zwischeninterface `IJavaReferenceType` liefert für dieses Projekt keine relevanten Informationen. Durch die beiden Interfaces `IJavaArrayType` und `IJavaClassType` lassen sich Arrays und Objekte unterscheiden. `IJavaClassType` bietet darüber hinaus noch eine Methode, die den Typ der Superklasse zurückgibt.

Das Quellmodell, der in diesem Projekt entwickelten Modelltransformationen, hat den Typ `IVariable`, ist also ein Modell einer Variablen.

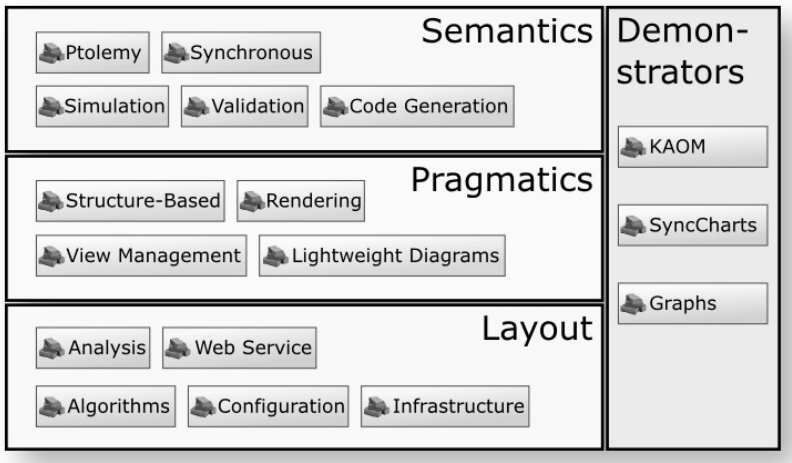


Abbildung 9. Teilbereiche von KIELER[6]

3.2 KIELER

Das *Kiel Integrated Environment for Layout Eclipse RichClient*, oder kurz *KIELER*, ist ein Forschungsprojekt rund um die Verbesserung von graphisch modellbasiertem Design von komplexen Systemen. Die grundlegende Idee dabei ist, ein automatisches Layout auf alle graphischen Komponenten von Diagrammen in der Modellierungsumgebung anzuwenden[6]. Das KIELER Projekt hat vier Bereiche (siehe Abbildung 9. Der Bereich Pragmatik befasst sich mit allen Aspekten, die die tägliche Arbeit von Modellierern im Bereich modellgetriebener Entwicklung beeinflusst.

3.2.1 KIELER Lightweight Diagrams

Basis dieses Projektes ist das *KIELER Lightweight Diagrams* Projekt[7], kurz *KLighD*, das Teil des KIELER Projekts und im Bereich Pragmatik angesiedelt ist. *KLighD* liefert eine vorübergehende leichtgewichtige Darstellung von Modell oder Teilmodellen, ohne die Einbindung von komplexen Bearbeitungsmechanismen, wie etwa graphische Editoren. Graphische oder textuelle Darstellungen werden aus gewählten Bruchstücken eines Grundmodells gebildet und verworfen, wenn sie nicht mehr verwendet werden. Teile des Modells

[6]http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Overview
[7]http://rtsys.informatik.uni-kiel.de/confluence/pages/viewpage.action?pageId=328115

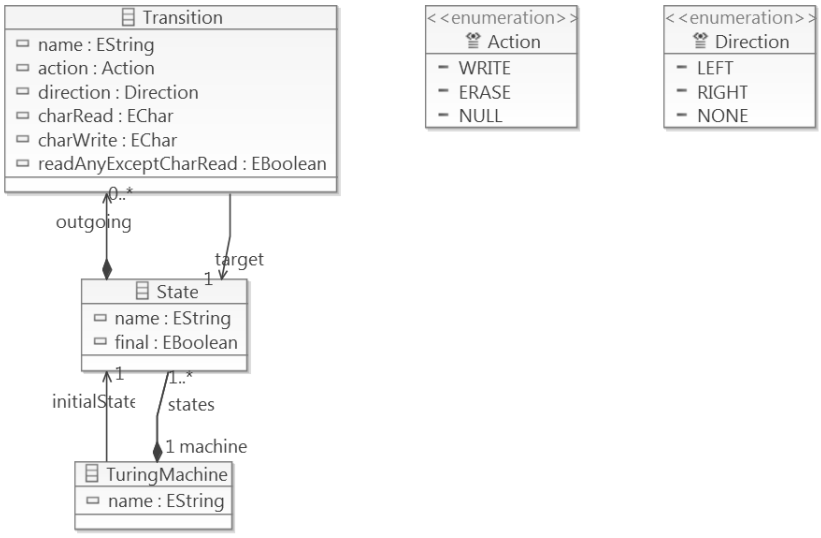


Abbildung 10. Modell einer Turingmaschine

werden also durch eine Modelltransformation in ein Teil der graphischen oder textuellen Darstellung umgewandelt.

KLighD ist ein Eclipse Plug-in und stellt einen Extension Point zur Verfügung, an dem Systhesen bzw. Transformationen registriert werden können. Neben dem Extension Point liefert das Plug-in auch die *Light-weight Diagram View*, in der die graphische oder textuelle Repräsentation dargestellt wird.

Abbildung 10 zeigt das Modell einer Turingmaschine, das im Vorfeld dieses Projektes entwickelt wurde. In der graphischen Repräsentation, die durch das Beispiel-Plug-in `de.cau.cs.kieler.klighd.examples` erzeugt wurde, sind lediglich die Struktur des Modells und die Namen von Elementen visualisiert (siehe Abbildung 11).

Die Extension muss die abstrakte Klasse `AbstractTransformation<S,T>` erweitern. Der generische Typ `S` entspricht dem Typ des Quellmodells und `T` dem Typ des Zielmodells, das in der View dargestellt wird. Wie in Abschnitt 3.1.3 erwähnt, ist `S` in diesem Projekt der Typ `IVariable`.

3.2.2 KGraph

Das Zielmodell der in diesem Projekt entwickelten Transformationen ist vom Typ *KGraph*, einem Modell eines Graphen. Das Metamodell eines KGraphen ist recht einfach aufgebaut (siehe Abbildung 12).

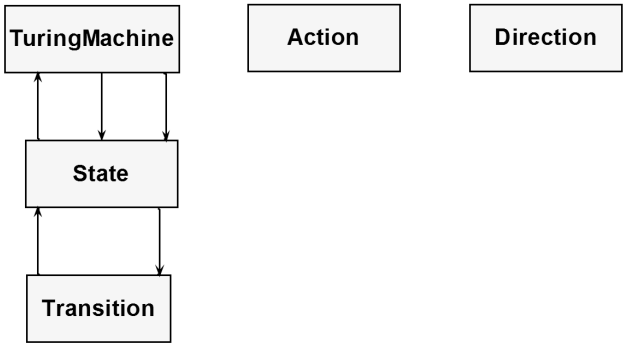


Abbildung 11. Repräsentation eines Modells einer Turingmaschine mit Hilfe von KLighD

Der gesamte Graph besteht aus einem einzelnen Knoten (`KNode`), der weitere Knoten enthalten kann. Dies entspricht dem Entwurfsmuster des Kompositums. Jeder Knoten kann als Quelle oder Ziel einer oder mehrerer Kanten (`KEdge`) dienen. Jedem Element des Graphen können Beschriftungen (`KLabel`) zugewiesen werden. Damit lässt sich die Struktur des Graphen beschreiben.

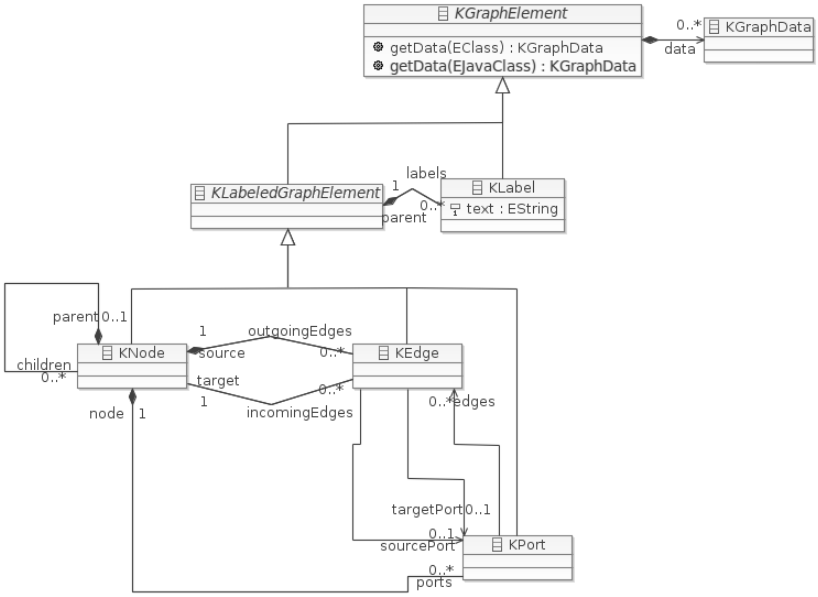


Abbildung 12. Metamodell eines KGraph

Jedes Element enthält noch eine Menge von zusätzlichen Daten (`KGraphData`), dazu gehören Informationen zur optischen Repräsentanz der Elemente (z.B. `KRectangle` oder `KText`) und Positionsdaten (`KShapeLayout` bzw. `KEdgeLayout`).

Da auch Transformationen für KLighD darauf ausgelegt sind in XTend geschrieben zu werden, werden viele hilfreiche Extension Methoden durch das Plug-in `de.cau.cs.kieler.core.krendering.extensions` bereitgestellt. Darunter sind Methoden zur Knotenerzeugung, sowie Methoden zur Veränderung der Eigenschaften von Graphelementen und Graphdaten.

Die Anordnung der Knoten des KGraphen in der View von KLighD übernimmt ein Layoutalgorithmus, der bei der Definition des Graphen mit Hilfe der Methode `addLayoutParam` hinzugefügt wurde.

4 Konzepte und Implementierung

Der grundlegende Ablauf der Visualisierung besteht aus mehreren Schritten (siehe Abbildung 13). Zunächst wählt der Benutzer die Variable in der Variable View aus, die Visualisiert werden soll. Durch die Auswahl wird die Light-weight Diagram View mit dem Namen „Variableërzeugt oder aktualisiert. Bei der Erzeugung bzw. Aktualisierung wird eine Modelltransformation (KLighD Transformation) mit der ausgewählten Variable als Quellmodell und einem Graphen als Zielmodell durchgeführt. Innerhalb dieser Transformation wird das Quellmodell an eine weitere Transformation , die für den konkreten Typ der Variablen registriert ist, oder an eine generische Transformation weitergeleitet (Debug Transformation). Während der Durchführung der Transformation können weitere registrierte bzw. generische Transformationen durchgeführt werden. Die Graphen, die die Ergebnisse der Transformationen sind, werden an die aufrufende Transformation zurückgegeben. Nachdem alle Transformationen durchgeführt wurden, wird der daraus resultierende Graph in der Light-weight Diagram View dargestellt und ein Layoutalgorithmus angewendet.

Die Wahl des Quellmodells ist durch den Typ der Einträge der Variable View vorbestimmt. Wenn ein Eintrag das konkrete Objekt bzw. der konkrete primitive Wert wäre oder beinhalten würde, könnten die Transformationen von KLighD direkt genutzt werden. Das Quellmodell hätte dann den Typ des Objektes bzw. primitiven Wertes und innerhalb der Transformation könnte direkt auf die Felder des Objektes zugegriffen werden. Allerdings haben die Einträge den Typ `IVariable` und sind Modelle der konkreten Variable. Es gibt Methoden, mit denen man alle Informationen zu den Variablen erhalten kann, aber es gibt keine, um die konkrete Variable zu erhalten. Der Umweg über die Interfaces des Debug Projektes und des JDT Debug Projektes muss genommen werden, um eine Visualisierung zu realisieren.

Im Folgenden werden einige Aspekte der Implementierung erläutert.

4.1 Selection Listener

Bei Auswahl eines Eintrags in der Variable View wird eine View geöffnet oder aktualisiert, wenn die View bereits geöffnet ist. Für diese Benutzerinteraktion bietet die Workbench von Eclipse einen Dienst (*Selection Service*), der dem Entwurfsmuster des Beobachters entspricht. Jedes Fenster der Workbench

Abbildung 13. Ablauf der Visualisierung

hat seine eigene Instanz des Selection Services. Jeder Part ist gleichzeitig ein *Selection Provider*, der dem Selection Service mitteilt, wenn in ihm etwas ausgewählt wurde oder ein anderer Part ausgewählt wurde. Der Selection Service leitet dann diese Auswahl an alle registrierten *Selection Listener* weiter (siehe Abbildung 14). Die Auswahl kann dabei verschiedene Typen haben (siehe Abbildung 15). Die Auswahl eines Eintrages in der Variable View ist vom Typ `IStructuredSelection` oder genauer vom Typ `ITreeSelection`. IStructuredSelection verweist auf eine Liste von Objekten. Bei ITreeSelection gibt es zu jedem Objekt zusätzlich Informationen zu dem Pfad von der Wurzel des Baumes bis hin zu dem Objekt. Jeder Pfad ist dabei eine Liste von Objekten. Da dieser Pfad für dieses Projekt irrelevant ist, wurde für die Auswahl der Typ `IStructuredSelection` verwendet. Die beiden Typen `ITextSelection` und `IMarkSelection` sind beide für Textmarkierungen vorgesehen.

Es war also lediglich notwendig einen Selection Listener zu implementieren und an dem Selection Service des aktuellen Fensters zu registrieren. Ein Selection Listener muss die Methode `selectionChanged(IWorkbenchPart part, ISelection selection)` des Interfaces `ISelectionListener` implementieren, die bei einer Auswahl aufgerufen wird.

4.2 Extension Point

Es wird angenommen, dass nur ein Eintrag der Variable View visualisiert werden soll, also wird lediglich das erste Objekt aus der Liste der ausgewählten Objekte verwendet. Nachdem geprüft wurde, ob die Auswahl vom Typ `IStructuredSelection` und der Typ des ersten Elements `IVariable` ist, wird

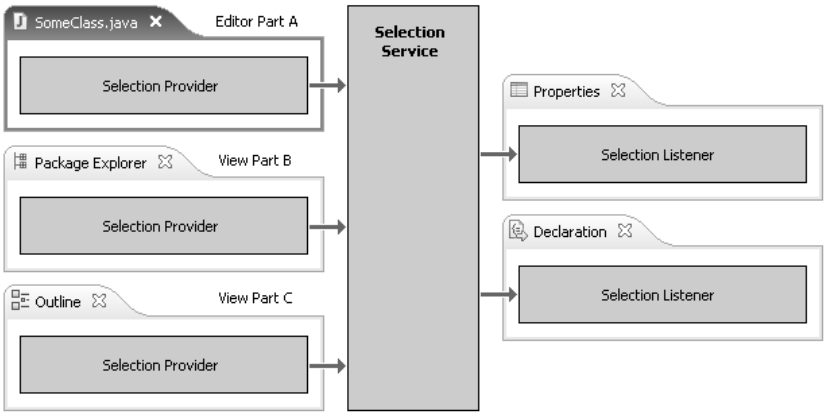


Abbildung 14. Selection Service von Eclipse

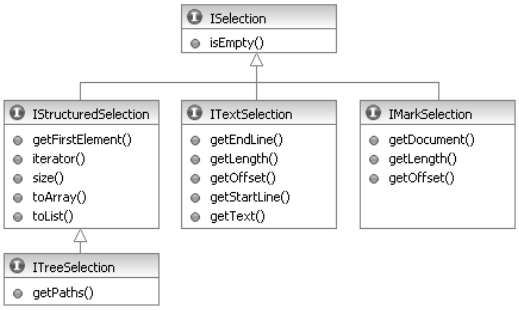


Abbildung 15. Klassendiagramm einer Auswahl

die Verbindung zur View von KLighD hergestellt. Diese View erhält den Namen `Variable`. Sollte keine View mit diesem Namen existieren wird eine neue erstellt, andernfalls wird die View aktualisiert, wenn sich die Auswahl geändert hat. Dadurch wird verhindert, dass dieselbe Visualisierung mehrfach hintereinander ausgeführt wird.

Beim Erstellen bzw. Aktualisieren der View wird die Methode `transform` der Klasse, die die abstrakte Klasse `AbstractTransformation<S,T>` erweitert und damit das Interface `ITransformation` für ein konkretes Quellmodell `S` und Zielmodell `T` implementiert, aufgerufen. Diese Klasse wird als Extension an dem Extension Point `de.cau.cs.kieler.klighd.modelTransformations` registriert. Die Auswahl der Transformation erfolgt über das konkrete Quell- und Zielmodell.

Der Benutzer hat die Möglichkeit, Transformationen zu bestimmten Datenstrukturen als Extension zu entwickeln und an einem Extension Point zu registrieren. Bei dieser Extension muss eine Klasse, die die abstrakte Klasse `AbstractDebugTransformation` erweitert, und die dazu gehörige Datenstruktur angegeben werden. Beim Start einer neuen Eclipse Instanz wird dann eine `HashMap` mit allen registrierten Transformationen gefüllt, wobei der Klassenname der Datenstruktur der Schlüssel und eine Instanz der Transformation der Wert ist.

Die Auswahl der Transformation erfolgt grundsätzlich über den Klassennamen. Sollte die Auswahl ein `Array` sein oder es keine Transformation für die Auswahl geben, wird eine Standardtransformationen für `Arrays` bzw. unbekannte Datenstrukturen zurückgegeben. Zunächst war die Auswahl der Transformation auf den Klassennamen der ausgewählten Variable beschränkt. Auch wenn Klassen zusätzliche Informationen enthalten können, enthalten sie dennoch alle Daten der Oberklasse, die visualisiert werden können. Durch die Methode `getSuperclass()` des Interfaces `IJavaClassType` kann auf den Namen der Oberklasse zugegriffen werden. Es wird so lange nach einer Transformati-

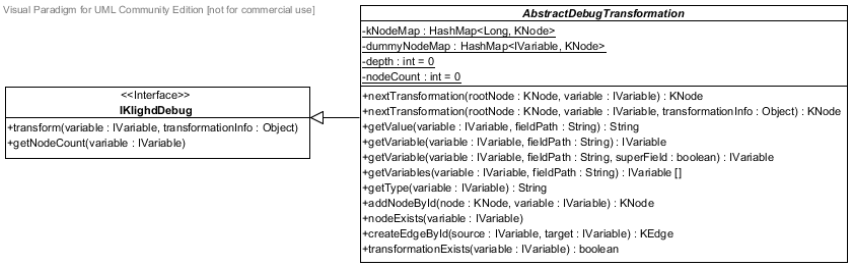


Abbildung 16. Klassendiagramm von AbstractDebugTransformation

on gesucht, bis eine gefunden wurde oder der aktuelle Typ keine Oberklasse mehr hat, also bis der Typ `java.lang.Object` ist.

Die zu erweiternde Klasse `AbstractDebugTransformation` enthält für die Transformation wichtige Methoden (siehe Abbildung 16), deren Konzepte und Implementierung im folgenden genauer beschrieben werden.

4.3 Variablen und Werte

Das Interface `IVariable` gewährt den Zugriff auf alle Informationen, die man zu Struktur und Inhalt einer Datenstruktur benötigt (siehe Abbildung 8). Konkrete Werte einer Variablen sind nur als Zeichenkette vorhanden und müssen in den konkreten Typ umgewandelt werden, wenn über die Visualisierung hinaus mit diesen gearbeitet wird. Die Methoden `getVariable`, `getValue`, `getVariables`, `getType` der Klasse `AbstractDebugTransformation` erleichtern den Umgang mit einer Variablen vom Typ `IVariable`. Da der deklarierte Typ der Variablen und der tatsächliche Typ des Wertes dieser Variablen unterschiedlich sein kann und der Typ der Variablen irrelevant ist, gibt die Methode `getType` den Typ des Wertes der Variablen als Zeichenkette zurück. Basis der anderen Methoden ist `getVariable`, die eine Variable, einen optionalen `boolean`-Wert und eine Liste von Feldnamen als Parameter erwartet. Sollte der `boolean`-Wert `true` sein, so wird angenommen, dass das letzte Feld der Liste in der Oberklasse zu finden ist. Es wird über die Liste der Namen iteriert und mit Hilfe der Methode `getField` des Interfaces `IJavaObject` die Variable mit dem aktuellen Namen zurückgegeben. In dieser Variablen wird dann nach dem nächsten Feld gesucht. Sollte in einem Iterationsschritt das jeweilige Feld nicht existieren oder der Name mehrdeutig sein, so wird `null` zurückgegeben. Die beiden Methoden `getValue` und `getVariables` ermitteln zunächst die Variable, dessen Wert bzw. Felder gesucht sind, und verwenden dann die entsprechende Methode des Interfaces `IValue`, wobei `getValue` die Zeichenkette des konkreten Wertes zurückgibt.

4.4 Assoziation von Variablen zu Knoten

Variablen können entweder Objekte, primitive Werte oder `null` sein. Da primitive Werte nicht referenziert werden können, also nicht eindeutig identifizierbar sind, ist jedem dieser Werte einen eigener Knoten zugeordnet, auch wenn sie den gleichen Wert haben sollten. Sollte eine Variable den Wert `null` haben, so wird diese wie ein Objekt behandelt. Damit Objekte in der Datenstruktur Knoten im Graphen zugeordnet werden können, müssen diese Objekte eindeutig identifizierbar sein. Dadurch kann geprüft werdenm ob schon ein Knoten zu diesem Objekt existiert und es kann verhindert werden, dass zu diesem Objekt ein neuer Knoten mit demselben Inhalt erzeugt wird. JDT Debug identifiziert jedes Objekt über eine einzigartige Nummer (unique ID), die die Methode `getUniqueId()` zurückliefert. Sollte das Objekt den Wert `null` haben, so ist die ID -1. Für primitive Werte wird die ID -2 verwendet. Diese Werte werden gesondert behandelt.

KLighD bietet die Möglichkeit erstellte Knoten mit einem Objekt zu verknüpfen, allerdings sind diese Informationen nur innerhalb einer Transformation vorhanden und können nicht in anderen Transformationen verwendet werden. Da aber während einer Transformation weitere Transformationen aufgerufen werden können, müssen diese Informationen über alle Transformationen hinweg erhalten bleiben. Dafür sorgt eine `HashMap`, die als statische Variable der Klasse `AbstractDebugTransformation` definiert ist. So kann jede Transformation auf die `HashMap` zugreifen. Als Schlüssel dient die ID des Objektes bzw. -1, wenn das Objekt den Wert `null` hat. Da primitive Werte nicht erneut referenziert werden, werden die zugeordneten Knoten auch nicht in der Map gespeichert, sondern nur mit der Variablen vom Typ `IVariable` verknüpft. Diese Verknüpfung besteht zwar nur innerhalb einer Transformation, aber eine Kante zu dieser Variablen wird auch in dieser Transformation erzeugt.

Grundsätzlich gibt es zu jedem Objekt nur einen Knoten, in dem die Informationen, die dieses Objekt enthält, dargestellt werden. Referenzen auf andere Variablen werden durch eine Kante zwischen den Variablen dargestellt. Wenn ein Objekt wiederholt auftritt, wird ein Knoten erzeugt, der selbst keinen Inhalt hat, sondern nur eine Referenz auf den dem Objekt zugeordneten Knoten. Dadurch bleibt die Struktur der Variablen erhalten und zu jedem Objekt gibt es nur ein Knoten, in dem Inhalt dargestellt wird. Damit man diese *Dummy Knoten* schnell erkennt, sind diese Knoten und die Kante zu dem Objekt rot. Abbildung 17 zeigt die Visualisierung einer `LinkedList`, bei der das erste und das zweite Element identisch sind. Dummy Knoten sind mit der Variablen vom Typ `IVariable`, die das Objekt repräsentiert, verknüpft. Diese Verknüpfung wird auch in einer statischen `HashMap` gespeichert.

Die Methode `addNodeById(KNode node, IVariable variable)` der Klasse `AbstractDebugTransformation` ist die einzige Methode, die ein Benutzer für

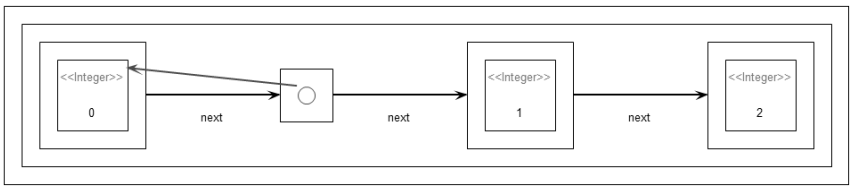


Abbildung 17. Visualisierung einer `LinkedList` mit erneut auftretenden Objekten

das Erzeugen und Hinzufügen neuer Knoten verwenden sollte. Ist bereits ein Knoten der Variablen zugeordnet, wird ein Dummy Knoten erzeugt, ansonsten wird ein neuer Knoten erzeugt, zu dem Knoten `node` hinzugefügt und gegebenenfalls eine Zuordnung zu der ID des Objektes in der `HashMap` gespeichert. Um Kanten zu erzeugen kann die Methode `createEdgeById(IVariable source,IVariable target)` verwendet werden. Sollte der Quell- (`source`) bzw. Zielvariablen (`target`) bereits einen Dummy Knoten zugeordnet sein, so wird dieser als Quell- bzw. Zielknoten der Kante verwendet. Repräsentiert `source` bzw. `target` einen primitiven Wert, so wird der Knoten, der dieser Variablen zugeordnet ist, verwendet. Ansonsten wird der Knoten, der mit der ID des Objektes verknüpft ist, als Quell- bzw. Zielknoten verwendet.

Nachdem die Transformation beendet wurde, werden alle statischen Variablen der Klasse `AbstractDebugTransformation` zurückgesetzt.

4.5 Rekursion

Da der Wert von Variablen wieder Variablen beinhalten kann, wird durch die Methode `nextTransformation` die Möglichkeit geschaffen, innerhalb einer Transformation weitere Transformationen aufzurufen. Sollte bereits ein Knoten mit der Variablen verknüpft sein und die Variable keinen primitiven Wert repräsentieren, so wird ein Dummy Knoten erzeugt und mit der Variablen verknüpft. Ansonsten wird die Transformation, die dem Typ der Variablen zugeordnet ist, durchgeführt. Die erzeugten Knoten werden dann dem Knoten `rootNode` hinzugefügt. Um sicherzustellen, dass ein Knoten mit der Variablen verknüpft wird, wird die Verknüpfung, falls dieser Variablen nicht schon ein Knoten zugeordnet ist, in der `HashMap` gespeichert. Um zusätzliche Informationen, die nicht in der Variablen gespeichert sind, zwischen den Transformationen austauschen zu können, kann ein Objekt an die nächste Transformation übergeben werden.

4.6 Begrenzung der Graphkomplexität

Ein großes Problem bei der Transformation von Datenstrukturen zu einem Graphen ist die Komplexität der Datenstruktur und damit des Graphen. Sollten zu viele Knoten oder Verschachtelungen sichtbar sein, wird der Graph sehr unübersichtlich und verliert meist seinen Nutzen. Zudem führt eine zu große Graphkomplexität dazu, dass der Layoutalgorithmus sehr lange braucht um die Position der einzelnen Elemente zu bestimmen. Der Benutzer kann über eine Seite in den Einstellungen von Eclipse festlegen, wieviele Knoten maximal angezeigt werden sollen und wie Tief eine Verschachtelung maximal sein soll.

Das Zählen der Rekursionstiefe ist recht einfach, sie wird vor dem Aufruf einer Untertransformation inkrementiert und nach der Ausführung dieser wieder dekrementiert. Problematisch beim Zählen der Knoten sind die zahlreichen Möglichkeiten Knoten zu erzeugen. KLighD bietet nicht die Möglichkeit einen Schwellwert für die Knotenanzahl anzugeben zudem haben die vorhandenen Layoutalgorithmen keine zeitliche Begrenzung, daher muss eine Methode implementiert werden, die die Anzahl der Knoten zurückliefert, die im aktuellen Graphen sichtbar sind. Dabei ist nur die Anzahl der Knoten wichtig, die in der jeweiligen Transformation erzeugt wurden, ohne eventuell aufgerufene Untertransformationen zu beachten.

Sollte die maximale Rekursionstiefe erreicht werden, so wird die Transformation für die nächste Tiefe nicht mehr ausgeführt. Stattdessen wird ein Standardknoten erzeugt, der den Namen und den Typ der Variablen enthält, die in der nächsten Transformation hätte visualisiert werden sollen (siehe Abbildung 19). Sollte die maximale Knotenzahl erreicht werden, so wird die aktuelle Transformation noch beendet, aber keine weitere mehr ausgeführt. Da die Graphen durch die Begrenzung nicht vollständig angezeigt werden und der Layoutalgorithmus sogar Fehler verursachen kann, wird beim Erreichen einer der Grenzen ein Informationsdialog geöffnet, so dass der Benutzer darüber informiert wird, dass eine Grenze erreicht wurde (siehe Abbildung 18).

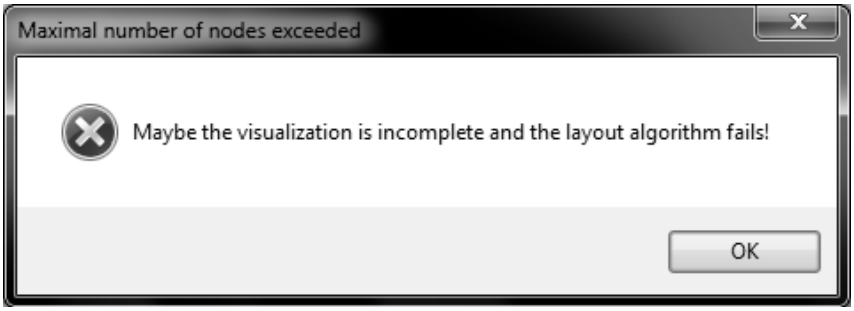


Abbildung 18. Dialog beim Erreichen der maximalen Knotenanzahl

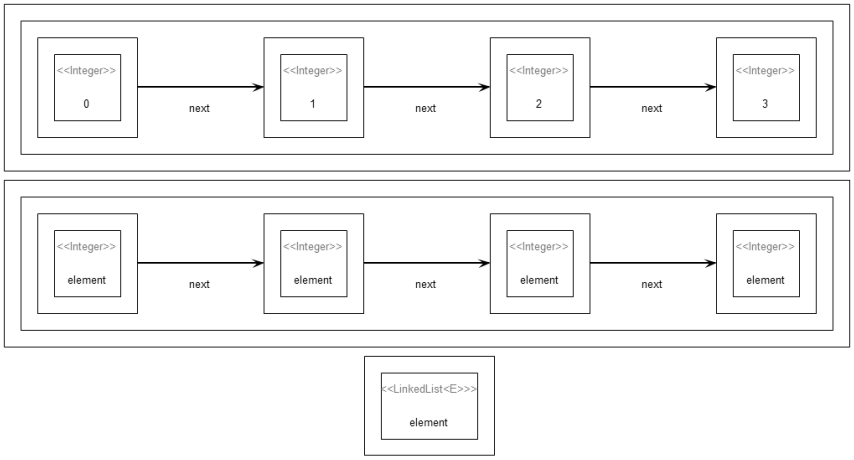


Abbildung 19. Drei Darstellungen derselben `LinkedList` bei verschienden Tiefen. Oben: Tiefe > 1, Mitte: Tiefe = 1, Unten: Tiefe = 0

5 Transformationen

Neben dem Interface und der Klasse mit Hilfsmethoden wurden auch einige Transformationen entwickelt, die einerseits eine generische Funktionsweise gewährleisten und andererseits zur Entwicklung von Konzepten beigetragen haben.

5.1 Standardtransformation

Sollte für eine Datenstruktur keine Transformation am Extension Point registriert sein, so wird eine Standardtransformation durchgeführt. Die Visualisierung beinhaltete zunächst nur den Typ und den Namen der Variablen. Es wurden also nur sehr wenige Informationen zu dieser Variablen visualisiert. Ein anderer Ansatz ist es alle Informationen darzustellen. Für jedes Objekt wird ein Standardknoten erzeugt, der von allen Feldern des Objektes, die einen primitiven oder `null` Wert haben, den Namen und den Wert beinhaltet. Zu allen Feldern, deren Wert ein Objekt ist, wird ein neuer Standardknoten erzeugt, der mit einer Kante mit dem Knoten, der mit dem Hauptobjekt verknüpft ist, verbunden wird (siehe Abbildung 20).

Nun stellt sich die Frage, wie Felder, für deren Typ eine Transformation registriert ist, eines Objektes, für deren Typ keine Transformation registriert ist, dargestellt werden. Es kann durchaus für die Übersicht von Vorteil sein, diese Transformationen auch bei der Standardtransformation zu verwenden. Aus diesem Grund gibt es die Möglichkeit drei verschiedene Methoden für die Standardvisualisierung auszuwählen. Die erste Variante (siehe Abbildung 20) stellt alle Knoten auf einer Ebene dar und wird flaches Layout genannt. Bei der zweiten Variante wird für jedes Feld eines Objektes, das ein Objekt als Wert hat, ein neuer Knoten erstellt, dessen Inhalt die Visualisierung dieses Objektes ist. Also ist der Knoten, der mit dem Objekt verknüpft ist, und je ein Knoten, der mit einem Objekt verknüpft ist, das in einem Feld des Hauptobjektes gespeichert ist, auf einer Ebene (siehe Abbildung 21). Durch diese verschachtelte Darstellung ist die Hierarchie, die auch schon in der Variable View existiert, klar erkennbar. Die dritte Darstellung ist eine Mischung aus der flachen und der hierarchischen Visualisierung. Sollte eine Transformation zu einem Typ eines Objektes existieren, so wird diese auch verwendet, ansonsten wird der Standardknoten auf der obersten Ebene der Visualisierung erzeugt (siehe Abbildung 22). Diese Darstellung ist am übersichtlichsten und ist deshalb auch voreingestellt. Der Benutzer kann über die Einstelltungsseite von Eclipse, auf der auch die Komplexität eingeschränkt werden kann (siehe Abschnitt 4.6), die Darstellungsart der Standardvisualisierung verändern.

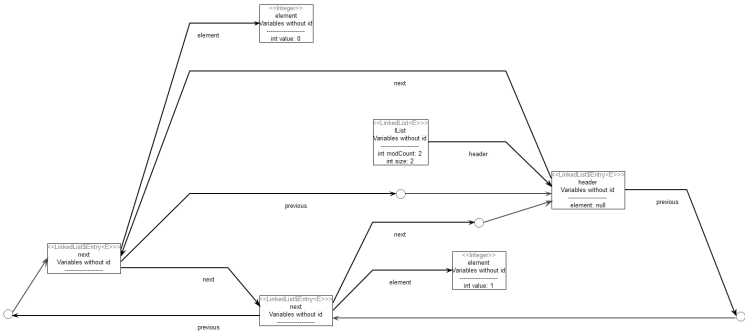


Abbildung 20. Flache Standardvisualisierung einer `LinkedList` mit zwei `Integer` Werten

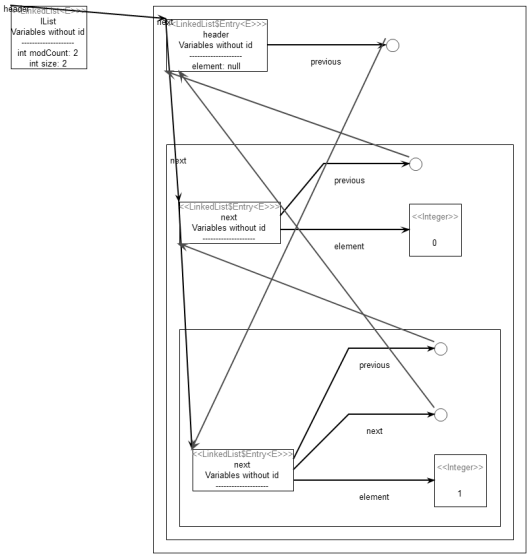


Abbildung 21. Verschachtelte Standardvisualisierung einer `LinkedList` mit zwei `Integer` Werten

5.2 Arraytransformation

Eine Transformation die unabhängig vom konkreten Typ eines Objektes ist, ist die Transformation eines Arrays. Ein Array ist eine Auflistung von Werten, diese Werte können primitive Werte, Objekte oder `null` sein. Außerdem können Arrays mehrere Dimensionen haben. Diese Dimensionen könnten durch Verschachtelung von Knoten visualisiert werden, eine übersichtlichere Mög-

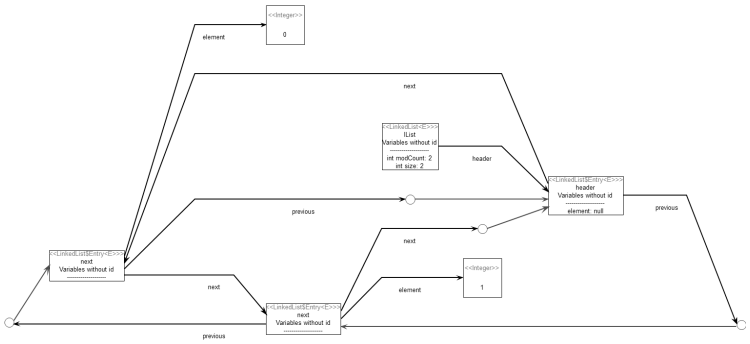


Abbildung 22. Gemischte Standardvisualisierung einer `LinkedList` mit zwei `Integer` Werten

lichkeit ist es allerdings, alle Dimensionen auf einer Ebene darzustellen, da für jede Dimension lediglich ein Knoten für jedes Element visualisiert werden muss (siehe Abbildung 23). Für die Transformation der konkreten Elemente des Arrays wird die Methode `nextTransformation` verwendet.

5.3 java.lang.*

Die am häufigsten genutzten Datenstrukturen sind im Java Paket `java.lang` zu finden. Diese Datenstrukturen sind die Objekt-orientierte Variante der primitiven Datentypen (`Boolean`, `Byte`, `Charakter`, `Double`, `Float`, `Integer`, `Long`,

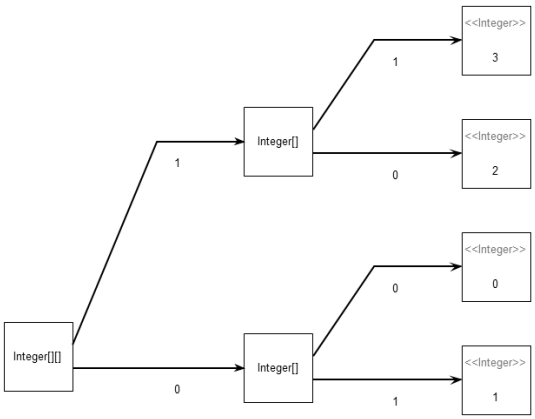


Abbildung 23. Standardvisualisierung eines zweidimensionalen Arrays mit `Integer` Werten

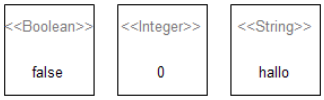


Abbildung 24. Visualisierung von drei Datenstrukturen aus dem Paket `java.lang`

`Short`). Das einzige Feld all dieser Datenstrukturen hat den Namen `value` und hat den primitiven Datentyp, den die Datenstruktur beschreibt. Die Visualisierung besteht aus einem Knoten, in dem der Typ und der Wert des Objekts dargestellt wird (siehe Abbildung 24). Da die Datenstrukturen, die eine Nummer beschreiben, alle die Klasse `java.lang.Number` erweitern, muss für diese nur eine Transformation registriert werden.

Eine weitere Datenstruktur im Paket `java.lang` ist `String`. Diese Struktur ist ein Array von Zeichen (`char`) und könnte auch so dargestellt werden. Es ist aber übersichtlicher ein `String` als Zeichenkette in einem Knoten darzustellen (siehe Abbildung 24). Also haben alle Klassen aus dem Paket `java.lang` dieselbe Darstellung.

5.4 java.util.*

Viele weitere oft verwendete Datenstrukturen sind in dem Paket `java.util`. Diese Datenstrukturen können in drei Kategorien eingeteilt werden, die im Folgenden beschrieben werden. Da diese Strukturen ausgereift sind, werden nur die Struktur und der Inhalt der konkreten Elemente und keine irrelevanten Informationen dargestellt.

5.4.1 Listen

Die Datenstrukturen `LinkedList` (Abbildung 25a), `ArrayList` (Abbildung 25b), `ArrayDeque` (Abbildung 25c), `PriorityQueue` (Abbildung 25d), `Stack` (Abbildung 25d) und `Vector` (Abbildung 25d) haben die Struktur einer Liste und können auf ähnliche Art und Weise visualisiert werden. Für jedes konkrete Element der Listen wird ein Knoten erzeugt, die durch Kanten miteinander verbunden sind, wodurch eine Reihenfolge dargestellt wird. Die Informationen über die konkreten Elemente erhält man je nach Klasse auf unterschiedliche Weise.

Bei allen außer der `LinkedList` liegen die Informationen in Form eines Arrays vor, über das iteriert wird. Die `LinkedList` besteht aus Einträgen, die ein Element, einen Folge- und ein Vorgängereintrag besitzen. Das Element des ersten Eintrags ist immer `null`, also beginnt die Liste mit dem Element des

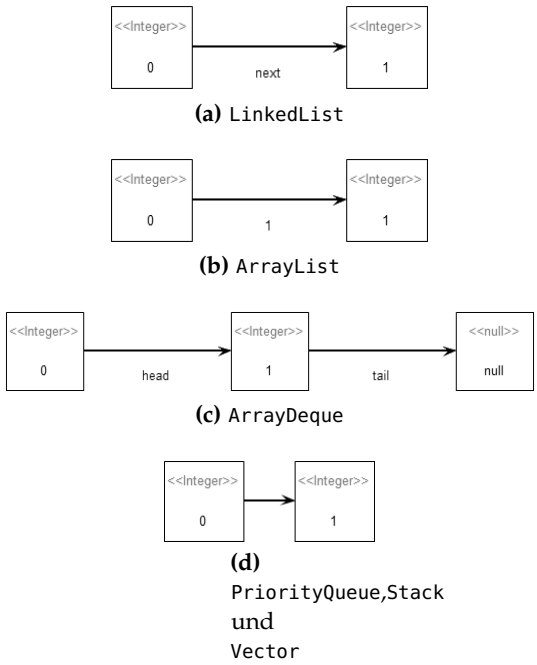


Abbildung 25. Visualisierung der Listen aus dem Paket `java.util`

ersten Folgeeintrags. Es wird über die Folgeeinträge iteriert und die Elemente dargestellt.

Einige Listen beinhalten zusätzliche Informationen, die auch Dargestellt werden. `ArrayDeque` hat ein `head` (erstes Element) und ein `tail` (letztes Element), das immer den Wert `null` hat. Eine `ArrayList` wird in Anlehnung an ein Array mit den Indizes der Elemente dargestellt. Um zu verdeutlichen, dass die `LinkedList` aus einer Reihe von Folgeeinträgen besteht, wird jede Kante zu einem Folgeelement mit `next` beschriftet.

5.4.2 Maps

Die Datenstrukturen `HashMap` (Abbildung 27a), `Hashtable` (Abbildung 27a), `IdentityHashMap` (Abbildung 27a), `WeakHashMap` (Abbildung 27a), `LinkedHashMap` (Abbildung 27b), `TreeMap` (Abbildung 27c) und `EnumMap` (Abbildung 27d) haben die Struktur einer Map. Es wird also ein Schlüssel mit einem Wert verknüpft. Wie bei den Listen sind auch bei den Maps die konkreten Schlüssel und Werte auf verschiedene Art und Weise in die Datenstruktur eingebettet.

Bei der `HashMap`, `Hashtable` und `WeakHashMap` sind die Informationen über den Schlüssel (`key` bei `HashMap` und `Hashtable`, `referent` bei `WeakHashMap`) und

den Wert (`value`) in einem Eintrag vorhanden, die in einem Array in der Map gespeichert sind. Es wird also über das Array iteriert und aus jedem Eintrag wird der Schlüssel und der Wert extrahiert und dargestellt.

Die `IdentityHashMap` hat die Besonderheit, dass sowohl die Schlüssel als auch die Werte in demselben Array gespeichert sind. Der Index eines Schlüssels ist stets gerade und der Wert ist das Element nach dem Schlüssel. Diese Eigenschaft hat allerdings nur Auswirkung auf die Transformation und nicht auf die Visualisierung.

Auch die `LinkedHashMap` besteht aus einem Array von Einträgen, diese sind aber im Gegensatz zu den anderen Maps miteinander verknüpft. Jeder Eintrag hat zusätzlich ein Feld namens `after`, dessen Wert der nächste Eintrag ist. Es wird also wie bei der `LinkedList` über die Folgeeinträge iteriert. Diese Verknüpfung wird durch eine Kante zwischen den Schlüsseln dargestellt.

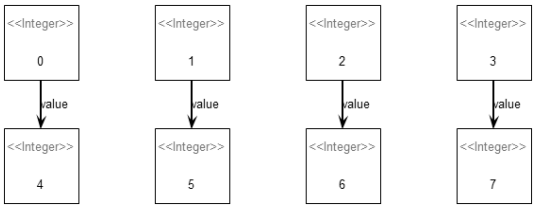
Neben dem Schlüssel und dem Wert enthält der initiale Eintrag `root` der `TreeMap` zwei Nachfolgeeinträge (`left` und `right`). Es wird zunächst über die linken Einträge und dann über die rechten Einträge iteriert und die Verknüpfung zwischen Schlüssel und Wert dargestellt. Zusätzlich ist jedes Schlüssel Wert Paar in einem Knoten, der den Eintrag repräsentiert, und die Referenzen zu den Folgeeinträgen sind mit `left` und `right` beschriftet. Da der Layoutalgorithmus für die Positionierung der Knoten verantwortlich ist, muss der linke und der rechte Eintrag nicht zwangsläufig in der richtigen Reihenfolge dargestellt werden.

Die `EnumMap` ist eine spezielle Map, bei der die Schlüssel Elemente einer Aufzählung (`Enum`) sind. Die Werte sind in einem Array gespeichert, bei dem der Index des Wertes dem Index des Schlüssels entspricht.

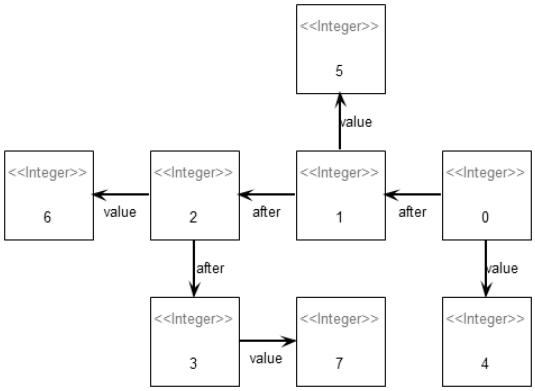
5.4.3 Sets

Die Klassen `HashSet`, `LinkedHashSet` und `TreeSet` sind Mengen und beruhen auf den jeweiligen Maps `HashMap`, `LinkedHashMap` und `TreeMap`. Im Gegensatz zu den Maps bestehen Sets allerdings nur aus den Schlüsseln ohne einen zugeordneten Wert. Die drei Sets haben ein Feld, dessen Wert die entsprechende Map ist, bei der der Wert vom Typ `Object` ist und lediglich für die Funktionalität der Map wichtig ist. Die Transformationen von Maps und Sets ähneln sich sehr. Bei den Sets muss kein Wert und damit auch keine Referenz zum Schlüssel dargestellt werden, ansonsten sind die Transformationen gleich.

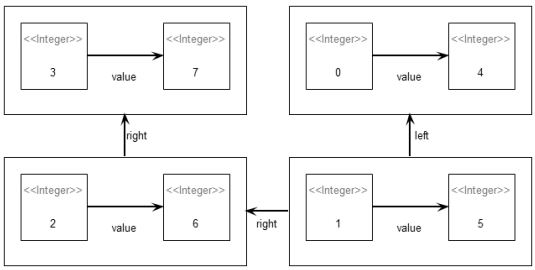
Ein Set, das nicht auf der entsprechenden Map beruht, ist `EnumSet`, das eine Menge von Elementen einer Aufzählung ist. `EnumSet` beinhaltet die Aufzählung und einen `Integer`-Wert, der die Elemente der Aufzählung spezifiziert, die in der Menge sind. In binärer Form werden die einzelnen Bits den Elementen zugeordnet. Wenn ein Bit gesetzt ist, so ist das jeweilige Element in der Menge.



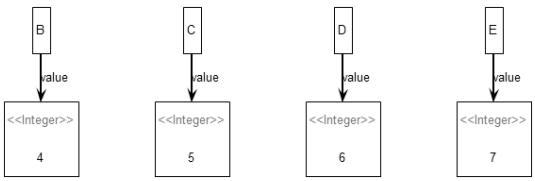
(a) `HashMap`, `Hashtable`, `IdentityHashMap` und `WeakHashMap`



(b) `LinkedHashMap`

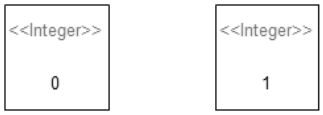


(c) `TreeMap`

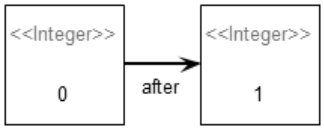


(d) `EnumMap`

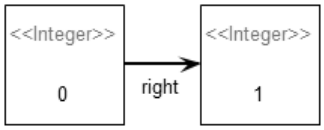
Abbildung 26. Visualisierung der Maps aus dem Paket `java.util`



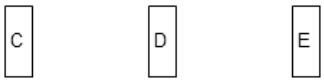
(a) `HashSet`



(b) `LinkedHashSet`



(c) `TreeSet`



(d) `EnumSet`

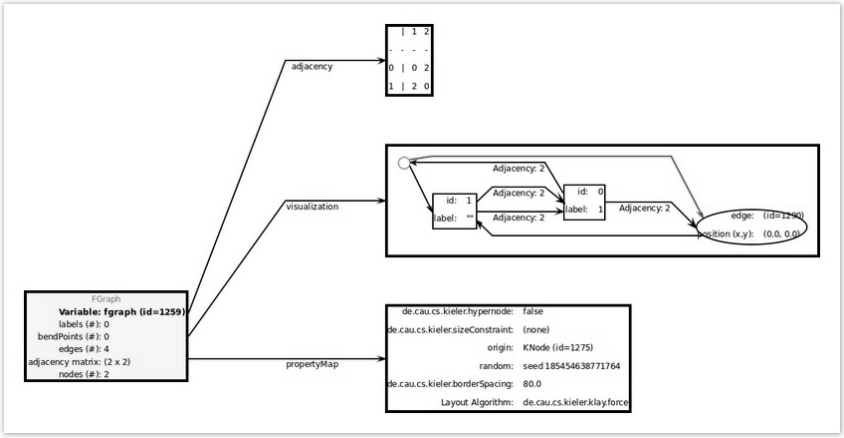
Abbildung 27. Visualisierung der Sets aus dem Paket `java.util`

6 Evaluation

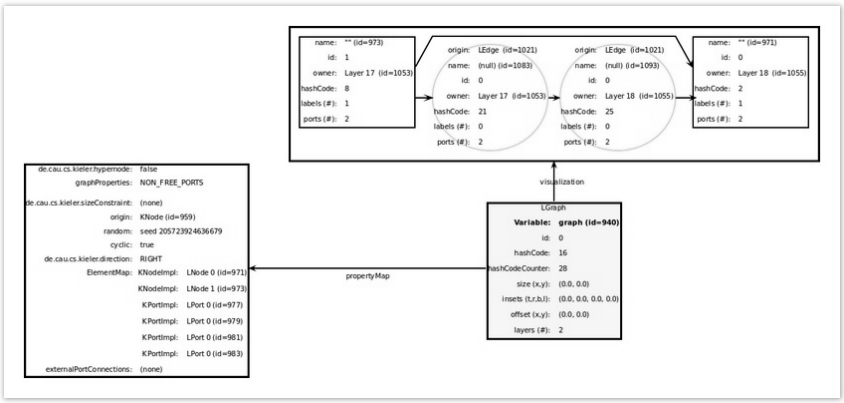
Die in diesem Projekt entwickelte Schnittstelle zur graphischen Visualisierung von Datenstrukturen wurde von Tibor Toepffer, einem weiteren Teilnehmer des Praktikums „Eclipse Modeling“ im Wintersemester 2012 der Arbeitsgruppe Echtzeitsysteme und eingebettete Systeme, verwendet um verschiedene Graphenstrukturen (`FGraph` (Abbildung 28a), `LGraph` (Abbildung 28b) und `PGraph` (Abbildung 28c)), die für die Entwicklung von Layoutalgorithmen wichtig sind, zu visualisieren. In der Variable View ist der Graph durch die Komplexität der Struktur gar nicht oder nur sehr schwer zu erkennen. Die Verwendung der Schnittstelle zur Entwicklung von Transformationen zu diesen Strukturen ermöglicht es einen schnellen und effizienten Überblick über den Graph zu gewinnen und die Analyse enthaltener Daten zu beschleunigen. Die Transformationen von Herrn Toeppfer wurden parallel zu den Hilfsmethoden der Schnittstelle entwickelt.

Die Visualisierungen folgen einer generellen Struktur. Sie bestehen aus einem Hauptknoten (eingefärbt), einem Knoten für die `propertyMap`, einem Knoten für die Visualisierung des Graphen und zusätzlichen Knoten für einige spezielle Informationen, die in den Graphenstrukturen gespeichert sind. Auf diese Weise wurden so viele Informationen wie möglich sehr übersichtlich dargestellt.

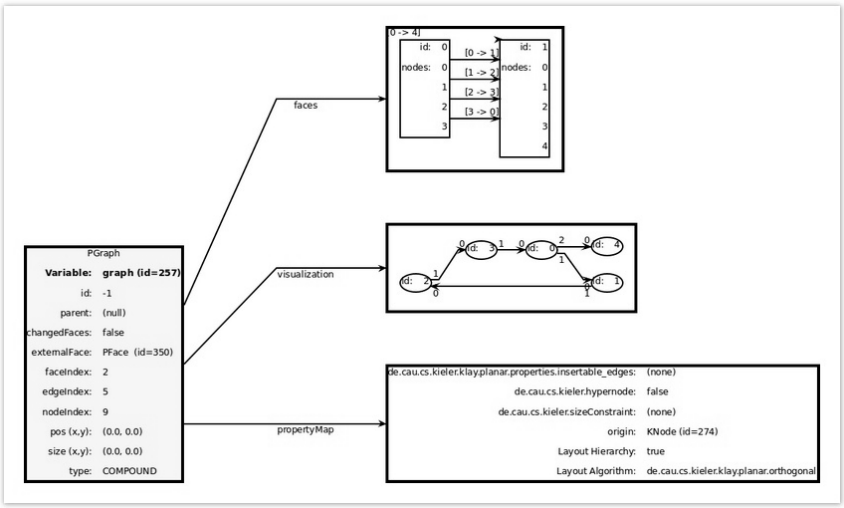
Hauptproblem bei der Entwicklung der Transformationen war die Auswahl der Informationen und wie diese dargestellt werden sollen. Die Registrierung der Transformation an den gegebenen Extension Point und die Verwendung der Hilfsmethoden rückten in den Hintergrund und waren nach kurzer Zeit eine Selbstverständlichkeit.



(a) `Graph`



(b) `LGraph`



(c) `PGraph`

Abbildung 28. Visualisierung von verschiedenen Graphenstrukturen

7 Ausblick und Fazit

Dieses Projekt hatte das Ziel die Entwicklung bzw. Verbesserung von Software durch eine Verringerung des Zeitaufwands, der beim Debuggen entsteht, zu unterstützen. Es basiert auf Eclipse und ist auf die Programmiersprache Java beschränkt. Da das Eclipse Plug-in KLighD die Möglichkeit bietet, modellbasierte, leichtgewichtige Visualisierungen zu erzeugen, bot es sich an das Modell einer Variablen in das Modell eines Graphen zu transformieren und darzustellen.

Das Projekt ist zwar auf Java beschränkt, kann allerdings auch für andere Programmiersprachen, die die Interfaces des sprachenunabhängigen Debug Projekts von Eclipse verwenden, erweitert werden. Einige Methoden der Klasse `AbstractDebugTransformation` nutzen Methoden von Interfaces des Java Development Tools Debug Projekts, die die sprachunabhängigen Interfaces implementieren. Diese Methoden müssten dahingehend angepasst werden, dass sie keine Methoden von sprachspezifischen Interfaces verwenden.

Eine weitere mögliche Erweiterung ist das Hinzufügen von Erweiterungs- bzw. Reduzierungsknöpfen innerhalb des Graphen, so kann die Graphkomplexität von vornherein minimiert werden, ohne dass Informationen verloren gehen und der Benutzer kann entscheiden, welche Objekte er genauer betrachten möchte. Knoten, deren Inhalt ein Objekt beschreibt, könnten reduziert dargestellt werden, so dass zunächst keine Transformation dieses Objektes vorgenommen wird, und durch eine Benutzeraktion erweitert werden, so dass die Transformation zu diesem Zeitpunkt durchgeführt wird. Dies hat außerdem den Vorteil, dass die Transformation und die Ausführung des Layoutalgorithmus weniger Zeit in Anspruch nimmt. Für diese Erweiterung müssten aber einige Änderungen an KLighD vorgenommen werden, weshalb es in diesem Projekt nicht mehr realisiert wurde.

Momentan wird der Zustand einer Datenstruktur zu einem Zeitpunkt dargestellt. Durch ein automatische Aktualisierung nach einem Ausführungsschritt des Debugmodus kann eine Art Animation realisiert werden, so dass Änderungen im Inhalt der Datenstruktur sehr schnell erkennbar sind. Dazu müsste eine Aktion nach der Ausführung des Schrittes ausgelöst werden, die die Aktualisierung ausführt.

Dieses Projekt hat gezeigt, dass die graphische Visualisierung von Datenstrukturen sehr viele Vorteile gegenüber der üblichen Darstellung in Form eines Baumes hat. Sie liefert einen schnellen Überblick über die Struktur und den Inhalt von Datenstrukturen. Dabei ist es wichtig, dass, je nach Anwendungsgebiet und Anwender, verschiedene Informationen wichtig bzw. irrelevant sind. Durch Modelltransformationen hat der Benutzer die volle Kontrolle über den Inhalt und die Erscheinung der Visualisierung. Die Transformationen sind recht einfach zu entwickeln und der Zeitaufwand der Entwicklung ist dem

Zeitaufwand der Analyse der Datenstruktur gegenüberzustellen. Schon bei wenig komplexen Datenstrukturen lohnt die Entwicklung einer Transformation, da diese auch wenig komplex sein wird. Je komplexer die Datenstruktur ist, desto wahrscheinlicher wird es, dass die graphische Visualisierung die Analyse beschleunigt.

Literatur

[1] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John M. Vlissides, and Jeaha Yang. Visualizing the Execution of Java Programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, London, UK, 2002. Springer-Verlag.

[2] Nahum Gershon, Stephen G Eick, and Stuart Card. Information visualization. *interactions*, 5(2):9–15, 1998.

[3] Herman Heine Goldstine and John Von Neumann. *Planning and coding of problems for an electronic computing instrument*. Institute for Advanced Study, Princeton University, New Jersey, 1948.

[4] John D. Gould. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, 7(2):151–182, 1975.

[5] John Hamer. A lightweight visualizer for Java. In *Proceedings of the Third Program Visualization Workshop, Research Report CS-RR-407, Department of Computer Science, University of Warwick*, pages 54–61, 2004.

[6] Heiko Kern, Stefan Kühne, and Daniel Fötsch. Merkmale und Werkzeugunterstützung für Modelltransformation im Kontext modellgetriebener Softwareentwicklung. In Klaus-Peter Fähnrich, Stefan Kühne, Andreas Speck, and Juia Wagner, editors, *Integration betrieblicher Informationssysteme: Problemanalysen und Lösungsansätze des Model-Driven Integration Engineering*, volume IV of *Leipziger Beiträge zur Informtik*, pages 94–104. Eigenverlag Leipziger Informatik-Verbund (LIV), Leipzig, Germany, September 2006.

[7] B. P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 135–144, New York, NY, USA, 1988. ACM.

[8] Brad A Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, 1990.

[9] Object Technology International, Inc. Eclipse Platform Technical Overview. *Technical Report*, http://www.eclipse.org/whitepapers/eclipse-overview.pdf, 2003.

[10] Rainer Oechsle and Thomas Schmitt. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug

Interface (JDI). In Stephan Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 176–190. Springer Berlin Heidelberg, 2002.

[11] B.A. Price, I.S. Small, and R.M. Baecker. A taxonomy of software visualization. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, volume ii, pages 597–606, 1992.

[12] H. Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, 1973.