FernUniversität in Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet Programmiersysteme
Prof. Dr. Friedrich Steimann

Analyse von Bezeichnerwechseln in Java-Programmen unter Berücksichtigung ihrer deklarierten und inferierten Typen

Abschlussarbeit im Studiengang
Master of Computer Science

vorgelegt von

Mark Thamm
Matrikel-Nr.: 6488773
An der alten Zuckerfabrik 8, 50321 Brühl

betreut durch

Prof. Dr. Friedrich Steimann

Brühl 2008

1 Abstract ... 4
2 Einleitung ... 5
2.1 Aufbau der Arbeit ... 6
2.2 Grundlagen und Begriffe ... 7
2.2.1 Deklarationselemente ... 7
2.2.2 Typbezeichner ... 7
2.2.3 Namen von Deklarationselementen ... 7
2.2.4 Zuweisungsgraphen ... 10
2.2.5 Protokoll, AccessSet und inferierter Typ ... 11
2.2.6 Darstellung der Zuweisungsgraphen ... 12
2.3 Beitrag der Arbeit ... 14
3 Problemstellung ... 16
3.1 Typwechsel ... 17
3.2 Inferierter Typ und Typwechsel ... 19
3.3 Inferierter Typ ohne Typwechsel ... 24
3.4 Bezeichnerwechsel ohne erkennbaren Grund ... 25
3.5 Zusammenfassung ... 26
4 Materialien und Methoden ... 28
4.1 Eclipse ... 29
4.1.1 Deklarationselemente finden ... 30
4.1.2 Die Benutzeroberfläche ... 32
4.2 Infertype ... 33
4.2.1 Typeconstraints ... 33
4.2.2 Übernommene Einschränkungen ... 37
4.3 Infername-Plug-In ... 38
4.3.1 Aufbau ... 38
4.3.2 Modeltransformation ... 40
4.3.2.1 Variablenzuweisungen ... 41
4.3.2.2 Zuweisungen von Methodenrückgaben ... 41
4.3.2.3 Zuweisungen an Parameterdeklaration ... 42
4.3.2.4 Algorithmus ... 43
4.3.3 Benutzung ... 44
5 Beispiele ... 47
5.1 Standardmuster ... 47

5.2 Refaktorisierungsmuster 53
5.3 Gegenmuster 65
6 Diskussion 69
7 Ausblick 72
8 Schlussbetrachtung 78
9 Anhang 79
9.1 Inhalt der beiliegenden CD 79
9.2 Installationsanleitung 79
9.3 Bedienung 80
9.4 Sourcecode 82
10 Literaturverzeichnis 83
Erklärung 86

1 Abstract

Variablen zeichnen sich in typisierten Programmiersprachen durch zweierlei aus: einen Typen und einen Bezeichner. Da jedes Programm in der Regel weit weniger unterschiedliche Typen als Bezeichner enthält, kommt dem Variablennamen eine speziellere Bedeutung zu als dem Typbezeichner. Tatsächlich bezeichnet der Variablenname üblicherweise auch die lokale Funktion der Variablen oder ihren erwartbaren Wert – er bezieht sich im Unterschied zum Typbezeichner also auf den Inhalt einer Variablen.

Variablen werden in Programmtexten einander zugewiesen, etwa über lokale Variablen oder Methodenparameter, so dass der Wert von Variable zu Variable wandert und im Laufe solcher Zuweisungen durch unterschiedliche Bezeichner auch unterschiedlich benannt werden kann. Insofern sich die Bedeutung des Variablennamen auf den Wert bezieht, ändert sich mit den Bezeichnern sehr wahrscheinlich auch der Wert. Eine Folge von Bezeichnern innerhalb solcher Zuweisungen kann man daher auch als Nutzungsprofil des Typen verstehen, mit dem die einander zugewiesenen Variablen deklariert sind. In der vorliegenden Arbeit werden diese Nutzungsprofile zunächst nach typischen Mustern untersucht, um herauszufinden, ob abweichende Fälle möglicherweise Strukturmängel aufweisen, die entweder durch eine standardisierte Namensgebung oder durch eine adäquatere Typisierung behoben werden können.

2 Einleitung

Eine wichtige Vorraussetzung für die Verständlichkeit von Programmtexten sind gut gewählte Variablen- und Methodennamen. Gut gewählt heißt, der Bezeichner einer Variablen oder Methode sagt bereits etwas über ihre Funktion aus.

Variablen können anderen Variablen zugewiesen werden, lokale Variablen etwa Methodenparametern, Methodenrückgabewerte wiederum lokalen oder globalen Variablen etc., so dass der Wert durch verschiedene Variablen weitergereicht wird. Gut gewählt heißt in diesem Zusammenhang auch, der gefundene Name sollte solange weiterverwendet werden, wie der Wert innerhalb der Zuweisungen noch dieselbe Bedeutung bzw. Aufgabe hat. Anders ausgedrückt: Wechselt ein Wert im Zuge seiner Verwendung laufend seine Bezeichnung, indem von Variablenzuweisung zu Variablenzuweisung ein anderer Bezeichner gewählt wird, ohne dass sich an der Bedeutung des Wertes etwas geändert hat, dann verschlechtert dies die Lesbarkeit des Programmtextes. Werden beispielsweise Konfigurationsdaten vom äußeren Rand einer Anwendung bis zu der die Konfigurationsdaten verarbeitenden Programmstelle über etliche Parameterdeklarationen einmal als `configuration`, dann als `confData` und ein anderes Mal als `mandantValues` weitergereicht, so erschwert dies für jeden Leser des Programmtextes die Identifikation der Variablen mit ihrer gemeinsamen Aufgabe – bzw. mit den Objekten der Anwendung. Begründbar wäre hingegen beispielsweise eine Zuweisung an eine Variable mit dem Namen `mandantConfiguration`, insofern der Verwendungskontext dieser Variablen eben in der Wiedergabe mandantspezifischen Konfigurationsdaten besteht.

In der vorliegenden Arbeit wird die Namensvergabe für Variablen innerhalb solcher Zuweisungen untersucht. Dabei soll überprüft werden, ob sich die Einführung weiterer Namen innerhalb einer Kette von Zuweisungen begründen lässt, oder ob man es mit einer refaktorisierungsbedürftigen Vielfalt von Bezeichnern zu tun hat. Für diese Überprüfung werden außerdem mögliche Typwechsel zwischen den zugewiesenen Variablen berücksichtigt. Die Zuweisung zweier Variablen `a` und `b`, mit den Typen `A` und `B`, wobei `A` eine Oberklasse von `B` ist, kann auch einen Wechsel der Bezeichner begründen. Darüber hinaus wird überprüft, ob hinter einem Bezeichnerwechsel ein erkennbar anderes Nutzungsverhalten des Typen liegt. Hierzu wird verglichen, ob die unterschiedlich bezeichneten Referenzen auch unterschiedliche oder zumindest unterschiedlich viele Methoden aus der Menge der Methoden, die der deklarierte Typ definiert, verwenden. Untersuchungsge-

genstand der vorliegenden Arbeit sind also Bezeichner- und Typwechsel sowie die Änderungen in der Nutzung des Typen innerhalb von Zuweisungsketten.
Ob ein einzelner Bezeichner die Aufgabe einer Variablen gut oder schlecht zur Sprache bringt, ist dabei nicht von Interesse. In der Entscheidung, ob `configuration`, `confData` oder `mandantValues` die treffendste Bezeichnung für die jeweilige Variable ist, lässt sich der Softwareentwickler auf programmatischem Weg nicht unterstützen, wohl aber bei der Frage, ob alle drei notwendig sind.

2.1 Aufbau der Arbeit

Die vorliegende Arbeit ist in fünf Bereiche unterteilt.

i. Zunächst wird die in dieser Arbeit notwendige Begrifflichkeit eingeführt. Dies geschieht in Kapitel 2.2 „Grundlagen und Begriffe“.
ii. In Kapitel 3 „Problemstellung“ werden die Thesen entwickelt, die der Arbeit zugrunde liegen und deren Anwendbarkeit im vierten Abschnitt überprüft wird. Diese Thesen beschreiben und begründen erwartete Zusammenhänge zwischen Bezeichner- und Typwechsel.
iii. Im Rahmen der Arbeit wurde ein Analysewerkzeug erstellt, dessen Anwendung, Struktur und Grundlage in Kapitel 4 „Materialien und Methoden“ beschrieben wird.
iv. Schließlich werden die zuvor entwickelten Thesen unter Zuhilfenahme des erstellten Analysewerkzeugs an realen Beispielen erprobt. Kapitel 5 „Beispiele“ beschreibt dabei die Ergebnisse dieser Anwendung und unterscheidet zwischen der Identifikation von erwarteten Mustern und solchen Fällen, in denen die Analyse zur Verdeutlichung von Strukturschwächen führt. Um auch die Grenzen der hier entwickelten Thesen deutlich zu machen, werden in einem weiteren Kapitel Codebeispiele vorgeführt, die mit den hier entwickelten Thesen nicht oder nur ungenügend analysierbar sind.
v. Kapitel 6 „Diskussion“, 7 „Ausblick“ und 8 „Schlussbetrachtung“ beenden die Untersuchung mit einer kritischen Beurteilung der Ergebnisse, der Skizze für ein Refaktorisierungswerkzeug als Fortsetzung der vorliegenden Arbeit sowie einem abschließenden Resümee.

2.2 Grundlagen und Begriffe

2.2.1 Deklarationselemente

Variablen, Methodenparameter und Methoden mit Rückgabewerten sind allesamt Programmelemente, die in typisierten Programmiersprachen aus einem Bezeichner sowie einer Typannotation bestehen. Sie alle können einen Wert im Werteraum des deklarierten Typen enthalten bzw. zurückgeben und werden im Folgenden zusammenfassend als Deklarationselemente bezeichnet.

2.2.2 Typbezeichner

Mit dem Typbezeichner soll üblicherweise auf ein Element des statischen Typmodells verwiesen werden, das dem Programm zugrunde liegt – sei es ein für die Software speziell erstelltes oder aber der „Alltagswelt" entlehntes. Bei letzteren haben wir es meist mit einem dem Typ zugrunde liegenden Gattungsbegriff zu tun, bei dem ein Softwareentwickler mit Recht auf ein Vorverständnis bei Lesern seines Codetextes ausgehen kann – `Person`, `Adresse`, `Kunde` wären Beispiele dieser Art.

2.2.3 Namen von Deklarationselementen

Im Unterschied zu Typbezeichnern haben die Namen von Deklarationselementen in der Regel keine so programmweit eindeutige Bedeutung. Einige der grundlegenden Bedeutungsmöglichkeiten werden im Folgenden aufgeführt.

Im einfachsten Fall gibt ein solcher Name die Funktion des Deklarationselements wieder – z.B. wenn ein Variablenname den lokalen Gebrauch des verwendeten Typen zum Ausdruck bringt. Das Deklarationselement `int i` gehört zu den prominentesten Beispielen dieser Gattung – insofern i traditionsgemäß für Laufindex steht.

Nicht wirklich gut zu trennen von diesem Gebrauch ist die mehr oder weniger ungekürzte Wiedergabe des Typbezeichners. Mit `Person p` oder `Person person` beispielsweise hat das Deklarationselement tatsächlich die Funktion einer nicht näher eingeschränkten Instanz[1] des zugrunde liegenden Typen. Insbesondere wenn öffentliche Methoden eine programmweite Operation definieren, die auf Instanzen eines bestimmten Typen durchgeführt werden kann, bringen die Bezeichner der Methodenparameter ihrerseits meist keine weiteren Angaben zur Funktion des Deklarationselementes mehr zum Ausdruck. Hier

[1] Objekt und Instanz werden im Folgenden synonym gebraucht.

werden oft die Typbezeichner ganz oder teilweise in den Namen der Deklarationselemente übernommen. Typische Beispiele dafür sind etwa:

- `dispose(Application application).`
- `public void inputChanged(Viewer viewer, ...,`
- `public void start(BundleContext context) ...`

Dieser Fall liegt vor allem dann vor, wenn das Deklarationselement innerhalb eines Interfaces definiert wurde, wo seine Bedeutung aus keinem unmittelbar vorliegenden Programmfluss heraus verstanden werden kann.

Neben der Funktion kann der Name eines Deklarationselementes aber auch den Zustand annotieren, in dem sich ein Objekt, das durch das Deklarationselement vertreten wird, an gegebener Stelle befindet. Beispielsweise ist der Name `authorizedUser` in einem Programmkontext vorstellbar, der eine Instanz vom Typ `User` bezeichnet, die eine erfolgreiche Autorisierung durchlaufen hat. Vor der Autorisierung wurden Deklarationselemente, die diesen Wert enthielten, mit `user` bezeichnet und in weiterverarbeitenden Methoden im Anschluss ebenfalls. Mit dem zusätzlichen Namensbestandteil `authorized` sagt der Programmtext etwas über den Zustand der Userinstanz nach der erfolgten Autorisierung aus.

Oft bezeichnet der Name auch eine Rolle, die ein Objekt in einem bestimmten Programmkontext spielt. Beispielsweise würde ein Deklarationselement `Employee chief` im Konstruktor einer Klasse `Employee` der hier übergebenen Instanz von Employee die Rolle `chief` zuweisen. Hierbei handelt es sich eigentlich um einen Aspekt des Typen – nämlich den, für andere Instanzen von `Employee` die Rolle `chief` spielen zu können –, der aber durch den Namen des Deklarationselementes zum Ausdruck gebracht wird. Es kann auch vorkommen, dass diese Fähigkeit des Typen zusätzlich durch ein Interface ausgedrückt wurde, das Supertyp von `Employee` ist, aber nur über die Zugriffsmethoden aus `Employee` verfügt, die für die Rolle `chief` benötigt werden. Mit Steimann[2] wird dabei davon ausgegangen, dass sich eine solche Rolle mindestens durch die folgenden Eigenschaften auszeichnet:

- Die Instanz wechselt nur vorübergehend vom Ausgangstyp zum Typen der Rolle – sofern denn überhaupt ein eigener Typ für die Rolle existiert. Davon unbeeinträchtigt bleibt das Objekt weiterhin auch eine Instanz des Ausgangstypen. Beispielsweise bleibt die Instanz von `Employee`, die anderen Instanzen von `Employee` als

[2] Vgl. hierzu: [Steimann 2000] (v.a.: S.1-66, S.115ff., S.169ff.) sowie [Steimann 2001].

chief zugewiesen wurde, weiterhin eine Instanz von Employee, und kann als solche behandelt werden. So kann sie in einem anderen Programmteil Adress- oder Kontodaten entgegennehmen – eine allgemeine Fähigkeit von Employee-Instanzen –, oder sie bekommt ihrerseits eine Instanz von Employee als chief zugewiesen, eine Fähigkeit, die eindeutig nicht aus der Rolle chief selbst stammen kann.

- Mit diesem Beispielszenario ist bereits eine weitere Eigenschaft von Rollen angesprochen, die sie zudem von Zuständen (s.o.) unterscheidet. Eine solche Rolle existiert immer nur in Beziehung zu anderen Typen – im Beispiel ist es eine Instanz von Employee, die für eine andere Instanz von Employee die Rolle des chief spielt. Rollen qualifizieren Assoziationen zu anderen Typen. Genau genommen sind Rollen deshalb selbst auch keine eigenständigen Entitäten, sondern Aspekte im Verhältnis zu anderen Typen. Reflektiert ein Zustand immer das Verhältnis zu sich selbst (authorizedUser, unsavedProject, unmodifiableModel, etc.), so reflektiert eine Rolle immer das Verhältnis des Typen zu anderen Typen (chief, child, etc.).
- Dieser Charakter einer Rolle, nämlich ein Verhältnis von Typ A zu Typ B als Teilaspekt von Typ A zu spezifizieren, entspricht in der objektorientierten Programmierung eher dem Konzept der Interfaces als dem von Klassen.[3] Als Typisierung von Rollen sind demnach eher Interfaces zu erwarten, insofern überhaupt eine spezielle Typisierung der Rolle vorliegt. Möglicherweise wird die Rolle nur durch den Namen angedeutet und verfügt über keinen eigenen Typen.
- Es kann mehrere Rollen für ein Objekt ggf. auch gleichzeitig geben.

Die Bedeutungsvielfalt der Namen eines Deklarationselements hängt natürlich stark von den Verwendungsmöglichkeiten eines Typen ab. Ein StringBuffer etwa hat einen recht eingeschränkten und klar definierbaren Verwendungszweck. Entsprechend sollte man erwarten, dass Deklarationselemente vom Typ StringBuffer immer entweder den Typbezeichner ganz oder teilweise übernehmen oder aber in ihrem Bezeichner den lokalen Verwendungszweck zum Ausdruck bringen – etwa StringBuffer htmlText. Bei einem Typ mit Namen Person ist hingegen mit allen genannten Bedeutungsmöglichkeiten zu rechnen.

[3] Vgl. hierzu [Steimann 2000] S.170ff.

Alle diese Herleitungen von Bedeutungen für Bezeichner unterstellen eine restlos standardisierte Namensgebung von Seiten des Softwareentwicklers, die tatsächlich aber in keinem Programmtext durchgehend vorausgesetzt werden kann. Es kommt in der vorliegenden Arbeit allerdings nicht darauf an, wie eine Namensgebung zustande gekommen ist, sondern ob sie im Ergebnis rational begründbar ist. Ist eine Benennung ohne die genannten Motive entstanden, kann sie im Sinne der angeführten Kriterien dennoch als refaktorisierungsbedürftig oder begründbar qualifiziert werden.

2.2.4 Zuweisungsgraphen

Ein Deklarationselement kann anderen, typkompatiblen Deklarationselementen zugewiesen werden, so dass der „ursprüngliche" Wert zwischen den Deklarationselementen gewissermaßen weitergereicht wird. Eine beliebige Menge solcher aufeinander zugewiesenen Deklarationselemente wird im Folgenden Zuweisungsgraph genannt. Dabei handelt es sich um einen in Zuweisungsrichtung gerichteten Graphen, der Zyklen enthalten und in dem jeder Knoten beliebig viele aus- sowie eingehende Kanten haben kann – kompakter ausgedrückt: ein gerichteter zyklischer Graph[4].

Entlang eines solchen Zuweisungsgraphen kann ein bestimmter Wert eine Folge von unterschiedlichen Bezeichnern erhalten, wenn man alle von ihm aus erreichbaren Knoten (genauer: die Bezeichner der Deklarationselemente, die ja Knoten des Zuweisungsgraphen sind) verfolgt. In dem folgenden Codeausschnitt

```
IrgendeinTyp a = null;
IrgendeinTyp b = a;
```

wird beispielsweise die Variable `IrgendeinTyp a` an `IrgendeinTyp b` zugewiesen, womit entlang des entstandenen Zuweisungsgraphen ein Bezeichnerwechsel von `a` nach `b` für einen Wert im Wertebereich von `IrgendeinTyp` stattfindet.

Ein Zyklus würde bereits durch die zusätzliche Codezeile

```
a = b;
```

entstehen. In diesem Fall gäbe es eine Zuweisung von `a` nach `b` und von `b` nach `a`.[5]

---

[4] Zu den Definitionen, siehe z.B.: [Diestel 2006], S.30ff

[5] Ein etwas realistischeres Quellcodeszenario wäre der klassische Wertetausch von zwei Variablen mit einer temporären Hilfsvariable:

```
temp = a;
a = b;
b = temp;
```

2.2.5 Protokoll, AccessSet und inferierter Typ

Ein beliebiger Typ *T* deklariert eine Menge von Membern wie Methoden und Felder, die Steimann als $\mu(T)$ bezeichnet.[6] Eine Teilmenge davon sind alle von anderen Typen erreichbaren und nicht statischen Member, die mit Steimann[7] im Folgenden als sein Protokoll bezeichnet werden. Dieses ist dort wie folgt definiert.

$\pi(T) := \{m \in \mu(T) \mid m$ ist eine öffentliche, nicht-statische Methode$\}$

Jedes Deklarationselement hat neben seinem deklarierten Typen einen inferierten Typen. Definiert wird der inferierte Typ `I` einer Referenz `a` vom Typ `T` als die kleinstmögliche Menge an Membern – das ist $\mu(I)$, auf die von `a` zugegriffen wird, vereinigt mit den entsprechenden Mengen aller Referenzen, denen `a` zugewiesen ist. Zur Ermittlung von $\mu(I)$ wird $\iota(a)$ als Funktion von `a` beschrieben, deren Algorithmus in der Sammlung aller Zugriffe aus $\mu(T)$ bei Vorwärtsverfolgung aller direkten und indirekten Zuweisungen von `a` besteht.

Da das dieser Arbeit zugrunde liegende Refaktorisierungswerkzeug Infertype inferierte Typen in Form von Interfaces erzeugen können soll und in Interfaces per definitionem nur öffentliche, nicht statische Methoden deklariert werden können, werden alle Fälle, in denen das Ergebnis von $\iota(a)$ entweder Felder oder statische sowie nicht öffentliche Methoden enthält, außer Acht gelassen. Der inferierte Typ wird also nur für die Fälle ermittelt, in denen $\iota(a) \subseteq \pi(T)$ gilt, bzw. in denen das benötigte Protokoll eine Teilmenge des Protokolls von `T` ist.

Das (vom inferierten Typen) benötigte Protokoll wird auch als das AccessSet bezeichnet.[8] Für den inferierten Typ ist auch die Bezeichnung maximal verallgemeinerter Typ gebräuchlich.

[6] Vgl. [Steimann 2007].

[7] Vgl. ebenda

[8] Vgl. hierzu [Steimann / Mayer 2007]

2.2.6 Darstellung der Zuweisungsgraphen

Die in Kapitel 3 „Problemstellung“ aufgezeigten Fragestellungen und die sich daraus möglicherweise ergebende Refaktorisierung lassen sich anhand einer geeigneten Darstellung des Zuweisungsgraphen untersuchen. Hier soll daher zunächst eine solche Darstellung beschrieben und exemplarisch vorgeführt werden.

In den Zuweisungsgraphen repräsentieren die Knoten die Deklarationselemente und die Kanten die Zuweisungen. Die Knoten werden beschriftet mit dem Bezeichner und dem Typ des dargestellten Deklarationselements. Zusätzlich wird durch die folgenden Abkürzungen (sowie unterschiedlicher Farbgebung) dargestellt, um was für eine Form von Deklarationselement es sich handelt – nämlich:

L: lokale Variable

F: Feldvariable

R: Methoden

P: Methodenparameter.

Das Verhältnis von Protokoll und AccessSet wird ebenfalls in der folgenden Verhältnisdarstellung [AccessSet/ Protokoll] in jedem Knoten eingetragen. Da in Java jeder Typ das Protokoll der Klasse `Object` erbt und daher auch jeder inferierte Typ zwangsläufig mindestens über das Protokoll von Objekt verfügt, bleibt der Zugriff auf Methoden der Klasse `Object` unberücksichtigt. Korrespondierend dazu werden diese Methoden auch nicht in der angegebenen Protokollgröße mitgezählt. Deklarationselemente vom Typ `Object` haben also eine Protokollgröße von null, und Protokollzugriffe wie etwa auf die Methode `getClass()` werden nicht im AccessSet aufgenommen.

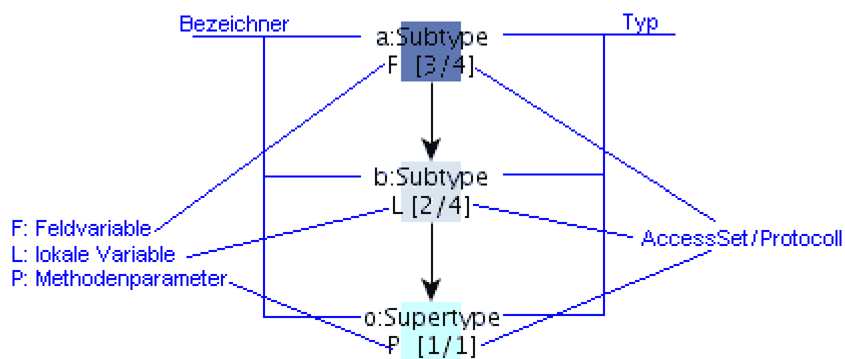
Das folgende Beispiel soll die Graphdarstellung veranschaulichen. Der Programmtext mit

```
Subtype a = null;
void foo(){
      Subtype b = a;
      a.m1();
      b.m2();
      method(b);
}

void method(Supertype o){
      o.m4();
}
```

den Typen Subtype und Supertype, die insgesamt über insgesamt vier Methoden verfügt,

```
public class Subtype extends Supertype{
            public void m1() {    }
            public void m2() {    }
            public void m3() {    }
}

public class Supertype {
            public void m4() {    }
}
```

würde z.B. wie folgt dargestellt:



Der Graph zeigt die drei Deklarationselemente mit Namen a, b, o, sowie deren jeweilige Typen ausgehend vom Deklarationselement a. Das Deklarationselement a ist eine Feldvariable, b eine lokale Variable und o ein Parameter, jeweils angezeigt durch F, L und P. Alle drei Deklarationselemente rufen genau eine Methode auf – die ersten beiden eine Methode des Subtypen, o hingegen die Methode des Supertypen. Da alle Methodenaufrufe von Deklarationselementen, die auf ein Deklarationselement zugewiesen sind, ebenfalls mit in das AccessSet aufgenommen werden,[9] enthält das AccessSet von a drei Methoden – den eigenen Aufruf sowie zusätzlich die Aufrufe von b und o –, b zwei und o nur noch seinen eigenen. Das Protokoll besteht im Falle des Typen Subtype aus den drei Methoden m1(), m2(), m3() sowie der geerbten Methoden m4() und für den Typen Su-

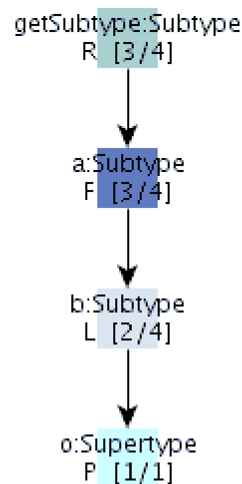
[9] Vgl. hierzu Kapitel 2.2.5 „Protokoll, AccessSet und inferierter Typ“.

pertype aus der einen Methode m4(). Damit ergeben sich die dargestellten Verhältnisse [3/4],[2/4] und [1/1].

Erweitert man den Beispielcode um eine Methode, die einen Subtype zurückgibt, und ändert die erste Zeile so, dass diese Methode auf das Deklarationselement a zuweist,

```
Subtype a = getSubtype();
...
Subtype getSubtype(){
    return new Subtype();
}
```

dann erweitert sich auch der Graph ausgehend von dem Deklarationselement getSubtype() um eben dieses Programmelement.



2.3 Beitrag der Arbeit

In dieser Arbeit wird untersucht, inwieweit sich Zuweisungsgraphen dazu verwenden lassen, Codestrukturen besser beurteilen zu können. Dabei werden zum einen die Interpretationsmöglichkeiten im Zusammenhang mit Zuweisungsgraphen eruiert, indem verschiedene Thesen über erwartbare Zusammenhänge zwischen Typ- und Bezeichnerwechseln sowie den Wechsel des inferierten Typen aufgestellt und begründet werden. Zum anderen wird demonstriert, wie diese Zusammenhänge praktisch sichtbar gemacht werden können, indem anhand eines eigens dafür entwickelten Eclipse-Plug-Ins Zuweisungsgraphen anhand von realen Programmkorpora erzeugt und mit den entwickelten Thesen interpretiert werden. Die vorliegende Arbeit kann in diesem Sinne auch als Machbarkeitsstudie eines geplanten und in Kapitel 7 „Ausblick“ skizzierten Refaktorisierungswerkzeuges ge-

lesen werden, das Zuweisungsgraphen als alternative Codedarstellung für Refaktorisierungsvorgänge anbietet.

3 Problemstellung

Aus der Benennung einzelner Deklarationselemente lässt sich für sich betrachtet keinerlei Information gewinnen. Verfolgt man aber eine Instanz über einen Zuweisungsgraphen hinweg, erhält man gegebenenfalls Zusatzinformationen aus dem Kontext ihrer Verwendung. Erfährt ein Objekt entlang des Zuweisungsgraphen einen oder mehrere Bezeichnerwechsel, so stellt sich die Frage:

a) Hat der Bezeichnerwechsel eine Bedeutung, das heißt: Bringt er womöglich einen zusätzlichen Aspekt des Typen – beispielsweise den, eine Rolle spielen zu können[10] – zum Ausdruck?

Oder wenn nicht:

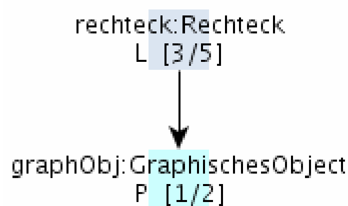
b) Ist er gewissermaßen bedeutungslos und aus Nachlässigkeit entstanden, das heißt: Verringert eine womöglich unnötige Vielzahl der Bezeichner für einen Wert lediglich die Lesbarkeit des Programmtextes?

Zunächst gilt es also zu prüfen, ob die unter a) gestellte Frage positiv beantwortet werden kann. Das heißt, für den vorliegenden Bezeichnerwechsel lässt sich eine der unter Kapitel 2.2.3 „Namen von Deklarationselementen“ angeführten Begründungen finden. Erst wenn sich keine Begründung finden lässt, stellt sich die Frage, ob nicht eine Vereinheitlichung der Namensgebung an der betreffenden Stelle eine sinnvolle Refaktorisierung wäre.
Als Indizien für einen „bedeutungsvollen“ Bezeichnerwechsel kann die Betrachtung der Typen und inferierten Typen herangezogen werden. Nicht nur der Name, sondern auch der Typ sowie der inferierte Typ können sich innerhalb eines solchen Zuweisungsgraphen im Rahmen der Typhierarchie ändern.

[10] Vgl. hierzu Kapitel 2.2.3 „Namen von Deklarationselementen“.

3.1 Typwechsel

Typwechsel innerhalb von Zuweisungsgraphen entstehen, indem ein Deklarationselement `a` einem Deklarationselement `b` zugewiesen wird, wobei `b` mit einem Supertyp des Typen von `a` deklariert ist. Aus technischen Gründen[11] sind Typwechsel innerhalb von Zuweisungen im Rahmen dieser Arbeit auf Zuweisungen an Supertypen beschränkt.
Häufig wechselt der Bezeichner von Variablen bei Typwechseln, indem er die Typbezeichnung ganz oder teilweise übernimmt. Dies gilt vor allem, wenn ein Deklarationselement auf Methodenparameter zugewiesen wird, die ganz allgemeine Operationen auf den Supertypen des Deklarationselements ausführen. Angenommen, ein Zeichenprogramm erzeugt eine Referenz `rechteck` vom Typ `Rechteck`, das seinerseits Subtyp eines alle geometrischen Objekte des Programms umfassenden Supertypen namens `GraphischesObject` ist. Nach der Erzeugung anhand von Benutzereingaben wird die Referenz `rechteck` einer Methode `paint(GraphischesObject graphObj)` übergeben. Es ergäbe sich dann der folgende Zuweisungsgraph.



In Methoden wie der Beispielmethode `paint()`, die eine Operation auf einem Supertypen definiert, sollen die Instanzen nicht eingeschränkt werden, weshalb oft der Typbezeichner auch als Name des Methodenparameters dient. Namensentwicklungen in Zuweisungsgraphen mit Typwechsel und Bezeichnerwechsel, in denen der Bezeichner ganz oder teilweise den Typnamen des Obertypen übernimmt, weil die Funktion des Deklarationselementes gegenüber den spezielleren Fähigkeiten der Subtypen abstrahiert, sollen im Folgenden Generalisierung genannt werden.[12]

---

[11] Die Arbeit übernimmt hier die Einschränkungen aus Infertype. Da Infertype keine Downcasts verfolgt, können auch keine Zuweisungen an Subtypen in den Zuweisungsgraphen übernommen werden. Siehe auch: Kapitel 4.2.2 „Übernommene Einschränkungen“.

[12] Aufgrund der unter Kapitel 4.2.2 „Übernommene Einschränkungen“ genannten Einschränkung, nämlich dass Zuweisungen nach Downcasts nicht weiterverfolgt werden, bleiben Zuweisungen von Ober- an Subtypen in dieser Arbeit unberücksichtigt. Somit muss auf die gegenläufigen Entwicklungen zu Generalisierung – wie Spezialisierung und Konkretisierung – sowie deren Unterscheidung nicht näher eingegangen werden.

Legt man die unter Kapitel 2.2.3 „Namen von Deklarationselementen“ dargestellte Idee des Bezeichners, gegebenenfalls mitsamt Typ, als Rolle zugrunde, dann kann es aber auch sein, dass ein solcher Namenswechsel mit Typwechsel die Zuweisung einer Rolle darstellt. Angenommen, in dem oben eingeführten Beispiel soll die Instanz von Rechteck mit einem anderen graphischen Objekt verbunden und dann gezeichnet werden. Zu diesem Zweck haben alle graphischen Objekte mit flächiger Ausdehnung eine Methode `connect()`, die für eine übergebene Zielfigur den geographisch nahesten Verbindungspunkt zum eigenen Außenrand wiedergibt. Da dies für Dreiecke, Kreise und Rechtecke etc. unterschiedlich berechnet wird, gibt es ein Interface `Connectable`, das graphische Objekte implementiert, die über diese Fähigkeit in Form der Methode `conntect(Connectable target)` verfügen. Im Beispielfall würde das Rechteck vor dem Aufruf von paint() zunächst seine Position durch Aufruf von `connect()` auf allen benachbarten Objekten ermitteln. Zu diesem Zweck würde es zeitweilig die Rolle `Connectable` einnehmen.

Ob nun Generalisierung oder Rollenzuweisung, in beiden Fällen sollten Typ- und Bezeichnerwechsel an derselben Stelle auftreten – also durch dieselbe Zuweisung verursacht werden. Für Zuweisungsgraphen mit Typ- und Bezeichnerwechsel stellt sich also die Frage:

c) Tritt ein Bezeichnerwechsel an der Stelle des Typwechsels auf?

Lässt sich die Frage bejahen, so kann der Bezeichnerwechsel im Sinne der eingangs gestellten Frage a) als begründet gelten – vermutlich handelt es sich um eine Rollenzuweisung oder eine Generalisierung.

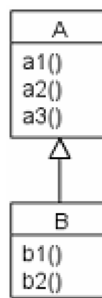
Liegen in einem Zuweisungsgraphen mit Typ- und Bezeichnerwechsel die beiden Wechsel hingegen nicht übereinander, dann lassen sich die folgenden beiden Fälle unterscheiden:

d) Tritt ein Bezeichnerwechsel erst nach einem Typwechsel auf, dann bleibt zu prüfen: Erhöht sich die Programmverständlichkeit, wenn der neue Bezeichner bereits ab dem Typwechsel eingeführt würde?
e) Tritt hingegen der Typwechsel später auf, so stellen sich zwei Fragen:
    a. Ist es möglich, dass der allgemeinere Typ schon frühzeitiger verwendet werden kann, als dies im vorliegenden Programmtext geschieht?
    b. Sollte der neue Bezeichner später eingeführt werden?

Im Zusammenhang dieser Fragen führt eine Betrachtung des Wechsels des inferierten Typen weiter.

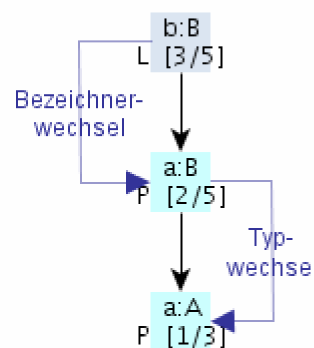
3.2 Inferierter Typ und Typwechsel

Im vorigen Kapitel wurde die Frage gestellt, ob die Tatsache, dass Bezeichner- und Typwechsel nicht übereinander liegen, ein Hinweis darauf ist, dass der Bezeichnerwechsel in die Richtung des Typwechsels verschoben oder der Typwechsel früher eingeführt werden sollte. Zur Illustration des in diesem Kapitel betrachteten Ausgangsproblems sei ein Beispielszenario eingeführt, in dem die Bezeichner `a` und `b` für Deklarationselemente vom Typ `A` oder `B`, mit `B` als Subtyp von `A`, existieren. Das Diagramm zeigt die vorhandenen Protokolle:



Der folgende Beispielcode veranschaulicht den Fall eines Bezeichnerwechsels vor einem Typwechsel:

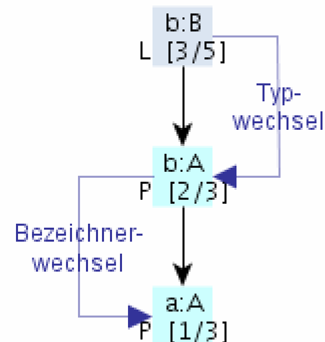
```
void test() {
 B b = null;
 b.b1();
 m1(b);
}

void m1(B a) {
 a.a1();
 m2(a);
}

void m2(A a) {
 a.a2();
}
```



Beispiel I: Bezeichner- vor Typwechsel

Werden Bezeichner und Typ in der Methode `m1()` von Beispielcode `A` vertauscht, ergibt sich ein Beispiel für einen Bezeichnerwechsel nach dem Typwechsel.

```
void test() {
 B b = null;
 b.b1();
 m1(b);
}

void m1(A b) {
 b.a1();
 m2(b);
}

void m2(A a) {
 a.a2();
}
```



Beispiel II: Bezeichner- nach Typwechsel

Mit dem Wechsel des inferierten Typen lässt sich klären, ob sich der allgemeinere Typ bereits früher einführen lässt, als das tatsächlich der Fall ist.

Ein solcher Wechsel bedeutet, dass sich ein Deklarationselement und sein Nachfolger innerhalb des Zuweisungsgraphen im Umfang ihres AccessSets unterscheiden. Da das AccessSet der einzelnen Deklarationselemente in Zuweisungsrichtung nur gleich bleiben oder sich verringern kann, bedeutet ein Wechsel des inferierten Typen immer, dass das AccessSet eines Deklarationselementes eine echte Teilmenge des AccessSets seines Nachfolgers ist – andernfalls hätte der inferierte Typ nicht gewechselt. Da der inferierte Typ aus der Vereinigung aller AccessSets entlang der Zuweisungen erzeugt wird, kann sich diese Menge in Zuweisungsrichtung nur verringern oder gleich bleiben.[13] Der Vergleich zwischen den AccessSets eines Deklarationselementes mit seinem Nachfolger enthält also nur die Fälle:

- Beide AccessSets enthalten dieselbe Anzahl Methoden.
- Das AccessSet des Vorgängers ist größer, bzw. enthält mehr Methoden.

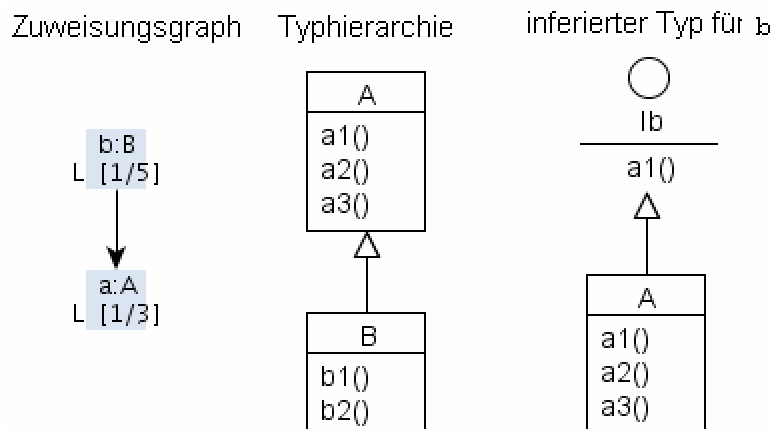
[13] Vgl. Kapitel: 2.2.5 „Protokoll, AccessSet und inferierter Typ“.

Der allgemeinere Typ kann ohne Verletzung der Typkorrektheit immer dann eingeführt werden, wenn der inferierte Typ des Vorgängers ein Supertyp des deklarierten Typs des Nachfolgers ist.[14]

Das trifft im einfachsten Fall immer dann zu, wenn der Vorgänger eines Deklarationselements denselben inferierten Typ hat wie der Nachfolger, der bereits allgemeiner typisiert ist. Hier ließe sich der allgemeinere Typ grundsätzlich früher einführen, da auch der Vorgänger kein größeres Protokoll des Typen benötigt als der Nachfolger. Ein solcher Fall liegt dem folgenden Beispielcode zugrunde:

```
A a = null;
B b = null;
a.a1();
b.a1();
a = b;
```

mit



Das AccessSet der beiden Deklarationselemente `a` und `b` enthält lediglich die Methode `m1()` aus dem Protokoll des Typen `A` – `b` ließe sich also genau wie `a` mit `A` deklarieren. Dieser Fall trifft immer zu, wenn die Größe beider AccessSets identisch ist, da aufgrund der Ermittlung des AccessSets[15] in beiden dieselben Methoden enthalten sein müssen.

---

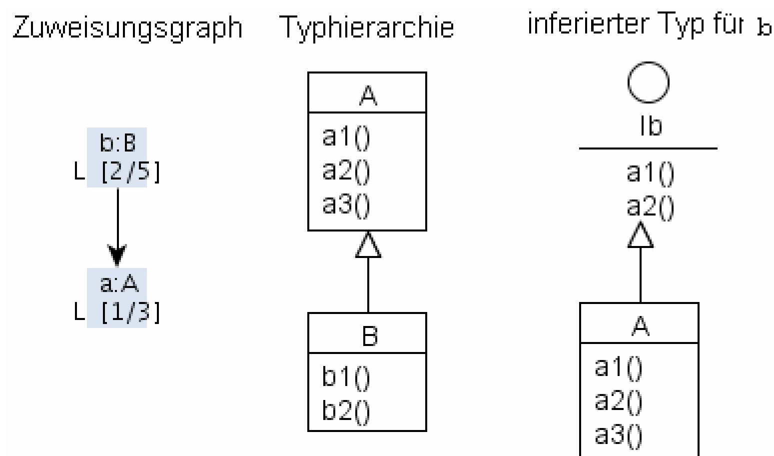
[14] Das gilt natürlich auch für den Spezialfall, dass das AccessSet des Vorgängers alle Methoden aus dem Protokoll des Typen des Nachfolgers benutzt. In diesem Fall ist der inferierte Typ des Vorgängers eher als typidentisch mit dem deklarierten Typ des Nachfolgers zu bezeichnen.

[15] Vgl. Kapitel 2.2.5 "Protokoll, AccessSet und inferierter Typ".

Ist das AccessSet des Vorgängers jedoch größer, so muss überprüft werden, ob der inferierte Typ ein Supertyp des deklarierten Typs des Nachfolgers ist. Ist das der Fall, ließe sich auch hier der allgemeinere Typ des Nachfolgers bereits für den Vorgänger einführen. Im vorigen Beispielszenario wird dies durch den Aufruf einer weiteren Methode aus dem Protokoll von `A` auf der Referenz `b` demonstriert.

```
A a = null;
B b = null;
a.a1();
b.a2();
a = b;
```

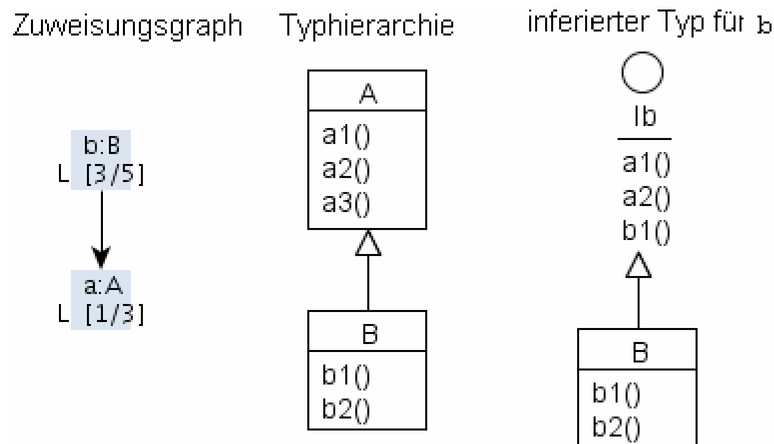
In diesem Fall ist der inferierte Typ von `b` ein Supertyp von `A`, und `b` könnte ebenso mit `A` deklariert werden, obwohl das AccessSet von `b` größer ist als das von `a`, wie der Zuweisungsgraph zeigt.



Enthält das AccessSet hingegen Methoden aus dem Protokoll eines anderen Typen als dem deklarierten Typen des Nachfolgers, so muss die Typisierung beibehalten werden. Fügt man im Beispielfall einen Methodenaufruf von `b1()` auf der Referenz `b` hinzu, dann ist der inferierte Typ von `b` nicht mehr Supertyp von `A`, sondern nur noch von `B`. Die Referenz `b` muss in diesem Fall ihre Typisierung beibehalten.

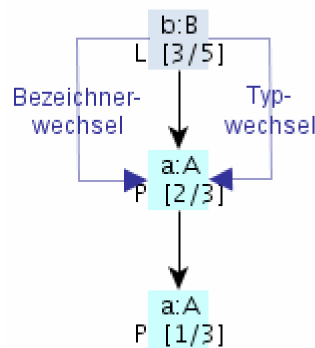
```
A a = null;
B b = null;
a.a1();
b.a2();
b.b1();
a = b;
```

In der Darstellung des inferierten Typ `Ib` für `b` sieht man, dass `Ib` ein Supertyp von `B` ist.



An dieser Stelle sei klargestellt, dass `Ib` nur für den vorliegenden Fall ein Supertyp von `B` ist. Bei einer längeren Typhierarchie mit drei Vererbungshierarchien kann der Typ des Nachfolgers mit dem allgemeinsten und sein Vorgänger mit dem speziellsten Typ deklariert sein und der inferierte Typ des Vorgängers könnte ein Supertyp des Typen der mittleren Vererbungsebene sein. Der inferierte Typ muss also nicht notwendigerweise ein Supertyp der deklarierten Typen innerhalb des Zuweisungsgraphen sein.
Entscheidend für die Einführbarkeit des allgemeineren Typen zu einem früheren Zeitpunkt ist die Prüfung, ob der inferierte Typ Supertyp des deklarierten Typen des Nachfolgers ist.
Ist das nicht der Fall, sollte überprüft werden, ob nicht der Bezeichnerwechsel weiter in Richtung Typwechsel verschoben werden kann. Das hätte zur Folge, dass der Bezeichner an der Stelle des Bezeichnerwechsels den Bezeichner des Vorgängers übernähme.

Anhand der Einführbarkeit des allgemeineren Typen sollte entschieden werden, ob entweder der Typwechsel früher oder der Bezeichnerwechsel später stattfinden kann, mit dem Ziel, Typ- und Bezeichnerwechsel wieder gleichzeitig stattfinden zu lassen.
Analysiert man die Eingangsbeispiele I und II anhand der hier eingeführten Methodik, so lassen sich folgende Änderungen vorschlagen:
Beispiel I: Das AccessSet des inferierten Typen des Methodenparameters von `m1()` enthält die Methoden `a1()` und `a2()` und ist damit Supertyp des deklarierten Typen des Nachfolgers, nämlich `A`. Es könnte also bereits für diesen Methodenparameter der allgemeinere Typ des Nachfolgers übernommen werden, womit Typ- und Bezeichnerwechsel übereinander lägen. Der folgende Zuweisungsgraph zeigt den Zustand nach dieser Änderung.

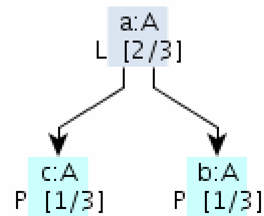


Beispiel II: In diesem Fall ist der Methodenparameter von `m1()` bereits mit dem allgemeineren Typ deklariert, führt aber den Bezeichner des Vorgängers fort. Hier sollte überlegt werden, ob der Bezeichner des Nachfolgers übernommen werden kann. Der resultierende Graph wäre derselbe wie für Beispiel I dargestellt.

3.3 Inferierter Typ ohne Typwechsel

Auch in Graphen oder Teilgraphen ohne Typwechsel, an denen nur ein Namenswechsel mit dem Wechsel des inferierten Typen übereinander liegt, kann überprüft werden, ob hier vielleicht eine Rollenzuweisung oder ein Zustandswechsel vorliegt. Im Falle einer Rollenzuweisung bietet sich möglicherweise die Einführung des Inferierten Typen in die Typhierarchie an. Das heißt: Hier böte sich die Ausführung des Infertype-Refactoring an. Der Name des zusätzlichen Typen wäre dann der Name des Deklarationselementes, das den Namenswechsel verursacht hat – denn dies sollte in aller Regel die zusätzliche Rolle treffend bezeichnen.
Ebenso analysierbar ist, ob es sich bei einer Reihe von Deklarationselementen mit unterschiedlichen Bezeichnern, aber gleichem Typ tatsächlich um verschiedene Typen mit verschiedenen Aufgaben handelt, die lediglich in einem Typen zusammengefasst wurden.
Ist ein Deklarationselement direkt mehreren Deklarationselementen zugewiesen, dann können die AccessSets dieser zugewiesenen Deklarationselemente untereinander disjunkt sein. Im folgenden Beispiel sind die AccessSets von `b` und `c` gleich groß aber disjunkt, da das eine AccessSet die Methode `a1()` und das andere `a2()` enthält.

```
A a = null;
m1(a);
m2(a);

void m1(A b) {
      b.a1();
}

void m2(A c) {
      c.a2();
}
```



Beispiel III: disjunkte AccessSets

In einem solchen Fall lässt sich prüfen, ob die Einführung mindestens einer der beiden inferierten Typen der beiden Referenzen `a` und `b` die Tatsache, dass der Typ `A` offenbar unterschiedliche Aufgaben wahrnimmt, besser zum Ausdruck bringt. Gegebenenfalls könnte man sich an dieser Stelle sogar fragen, ob nicht die Aufteilung in verschiedene Klassen sinnvoll wäre, ob also etwa ein „bad smell“-Befund im Sinne der „Large Class“ vorliegt.[16] Dabei handelt es sich um eine Klasse mit zu vielen Aufgaben, die besser in mehrere Klassen aufgeteilt würde.

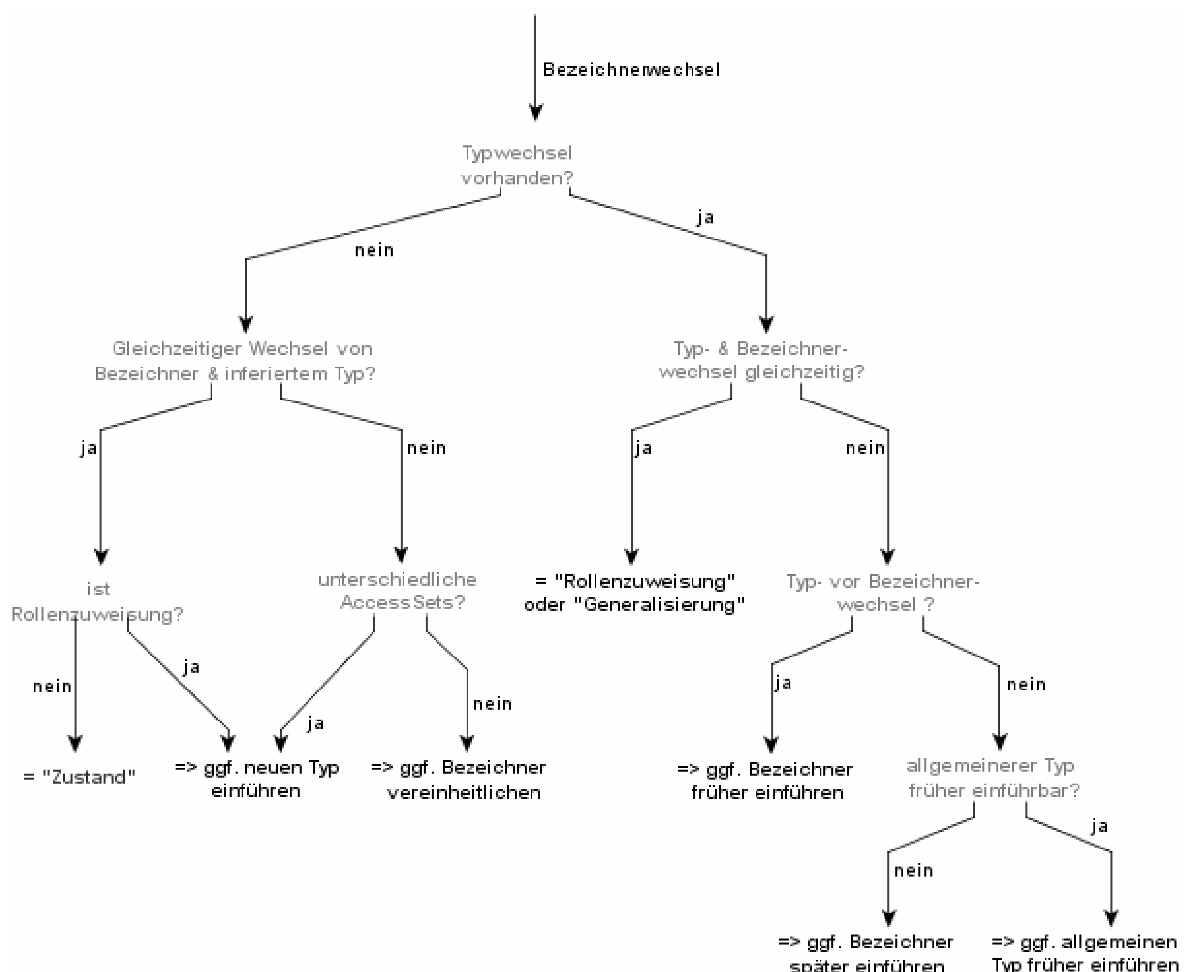
3.4 Bezeichnerwechsel ohne erkennbaren Grund

Insofern sich schließlich keines der Szenarien als zutreffend erweist, liegt es nahe anzunehmen, dass die in Kapitel 3 „Problemstellung“ unter b) gestellte Frage, nämlich ob eine unnötige Vielzahl an Bezeichnern vorliegt, bejaht werden kann. Dies ist vor allem bei Bezeichnerwechseln ohne einen Typwechsel und mit identischen bis sehr ähnlichen AccessSets der Fall, wo der neue Bezeichner sehr wahrscheinlich keine neue Bedeutung einführt. In solchen Fällen ist eine Refaktorisierung zur Vereinheitlichung der Bezeichner vorzuschlagen.

[16] [Fowler u. a. 1999], S.78ff.

3.5 Zusammenfassung

Die dargelegten Überprüfungen der verschiedenen Eigenschaften von Zuweisungsgraphen hinsichtlich ihrer Bezeichner-, Typ- und inferierten Typwechsel werden zur besseren Übersicht anhand der folgenden Baumdarstellung illustriert. Dabei stellen die Knoten Entscheidungen, mögliche Aktionen oder Analyseergebnisse dar. Als Analyseergebnis wird die Begründung eines Bezeichnerwechsels als Generalisierung, Rollenzuweisung oder Zustand bezeichnet. Diese Ergebnisse stehen in Anführungszeichen. Aktionen sind mit „=> ggf." eingeführt und weisen auf eine Refaktorisierungsmöglichkeit des analysierten Programmtextes hin.



Bei diesem Graphen handelt es sich um eine Veranschaulichung der einzelnen Herleitungen verschiedener Eigenschaften von Zuweisungsgraphen, wie sie in diesem Kapitel beschrieben wurden und wie sie der Interpretation realer Zuweisungsgraphen im Kapitel 5 „Beispiele" zugrunde gelegt werden. Es handelt sich nicht um die Darstellung eines Algo-

rithmus, der, auf einen Programmtext angewandt, zu zwangsläufigen Refaktorisierungsergebnissen führt.

4 Materialien und Methoden

Das zentrale Werkzeug der vorliegenden Arbeit ist ein in deren Rahmen entwickeltes Analysetool, mit dessen Hilfe Zuweisungsgraphen visualisiert werden können. Die wesentlichen Anforderungen an dieses Werkzeug waren dabei:

- die Ermittlung von Zuweisungsgraphen innerhalb beliebiger Programmtexte,
- eine möglichst übersichtliche Darstellung der Zuweisungsgraphen und
- die Möglichkeit, innerhalb des Programmtextes, der dem Graphen zugrunde liegt, zu navigieren.

Dieses Werkzeug – das in dieser Arbeit den Titel Infername-Plug-In trägt – wurde allein für den Zweck entwickelt, Zuweisungsgraphen zu veranschaulichen, um auf diese Weise Annahmen über Zusammenhänge zwischen den Wechseln von Typen, inferierten Typen und Bezeichnern zu überprüfen. Der Einsatzzweck des Infername-Plug-Ins reicht also nicht über den Dienst an der vorliegenden Arbeit hinaus und beabsichtigt daher auch nicht, in puncto Benutzerfreundlichkeit, Stabilität und Performanz eine Qualität für den professionellen Einsatz zu erreichen. Was grundsätzlich die Ermittlung von Zuweisungsgraphen und deren Darstellung innerhalb von Eclipse anbelangt, so kommt dem Plug-In außerdem der Charakter einer Machbarkeitsstudie zu, insofern in Kapitel 7 „Ausblick" dieser Arbeit für die Entwicklung eines neuen Refaktorisierungswerkzeuges geworben werden soll, das diese Techniken einsetzt.

Die Erzeugung von Zuweisungsgraphen setzt die Möglichkeit voraus, Deklarationselemente innerhalb von beliebigen Programmtexten als solche identifizieren zu können. Mit der Entwicklungsumgebung Eclipse steht bereits ein Werkzeug als Open-Source-Framework zur Verfügung, das mit einem so genannten Abstract-Syntax-Tree[17] über eine Darstellungsmöglichkeit von Programmtexten in Form einer Baumstruktur verfügt und somit die Identifikation von Deklarationselementen ermöglicht. Das vorliegende Analysetool verwendet Eclipse darüber hinaus als UI-Framework zur Präsentation seiner Ergebnisse und wurde als so genanntes Eclipse-Plug-In realisiert.

Ein weiteres Werkzeug, auf dem das Infername-Plug-In aufbaut, ist das Eclipse-Plug-In Infertype[18]. Dieses Plug-In erzeugt mit den erwähnten Möglichkeiten von Eclipse bereits Zuweisungsgraphen – allerdings mit etwas anderer Zielrichtung und daher auch in ande-

---

[17] im Folgenden abgekürzt als AST

[18] http://www.intoj.org/index.php/Infer_Type

rer Form. Trotzdem lassen sich diese Graphen transformieren und so als Datenmodell der Zuweisungsgraphen in dem entwickelten Analysetool verwenden.
Da diese beiden Werkzeuge grundlegend für die vorliegende Arbeit sind, werden sie in den folgenden beiden Kapiteln kurz eingeführt.

4.1 Eclipse

Für das Verständnis dieser Arbeit genügt eine grobe Vorstellung von Eclipse als seinerseits programmierbares Programmierwerkzeug, das sich in den für die Arbeit relevanten Aspekten auf die im Folgenden skizzierte Weise erweitern lässt. Dieses Kapitel unternimmt daher auch nicht den Versuch einer Einführung oder gar einer vollständigen Beschreibung von Eclipse, sondern skizziert nur die Möglichkeiten, auf die im Rahmen dieser Arbeit zurückgegriffen wurde.
Eclipse ist vor allem ein auf Java-Technologien basierendes UI-Framework, das besonders auf die Entwicklung von Entwicklungsumgebungen für Programmiersprachen ausgerichtet ist.[19] Eine der herausragendsten Eigenschaften von Eclipse ist seine flexible Erweiterbarkeit durch die so genannte Plug-In-Technologie[20], der die gesamte Architektur konsequent untergeordnet ist. Plug-Ins sind Programme, die Eclipse sowohl als Framework als auch als Laufzeitumgebung verwenden und so die ursprüngliche Plattform um ihr eigenes Anwendungsziel erweitern. Dabei kann jedes Plug-In anderen Plug-Ins Funktionen zur Verfügung stellen, muss dies aber nicht. Diese Erweiterbarkeit der Plattform erfolgt innerhalb von Eclipse nach exakt spezifizierten Regeln der Plug-In-Entwicklung, die zu beschreiben hier nicht der Ort ist.[21] Dank dieser Erweiterbarkeit dient Eclipse einer ständig wachsenden Zahl von Programmen mit unterschiedlichsten Anwendungszielen als Framework. Das macht es im Grunde unmöglich, Eclipse aus Sicht seiner Anwendungs-

---

[19] Das unter der so genannten „Eclipse Public Licence“ stehende Programm ist unter http://www.eclipse.org/ frei erhältlich.

[20] Seit der Version 3.0 von Eclipse und der Umsetzung des OSGI-Standards müsste man eigentlich von Bundles sprechen. In der Eclipse-Welt hat sich der Begriff Plug-In dennoch bereits vor dieser Version durchgesetzt und wird auch heute noch verwendet, da Plug-Ins und Bundles im Wesentlichen denselben Zweck und eine weitgehend gleiche Umsetzungsstrategie haben. Daher wird auch in der vorliegenden Arbeit von Plug-Ins gesprochen.

[21] Die Fülle der Literatur zum Thema Plug-In-Entwicklung ist längst unüberschaubar. Als Standardwerke sind zu nennen: [Gamma / Beck 2004] sowie [Clayberg / Rubel 2006]. Diese beiden Werke beschäftigen sich eingehend mit der Entwicklung von Plug-Ins für Eclipse. Aufgrund der kurzlebigen Aktualität von Druckerzeugnissen gegenüber der rasanten Weiterentwicklung von Eclipse bleibt ein Blick auf die offizielle Homepage – www.eclipse.org – der beste Literaturhinweis. Hier finden sich zahlreiche Artikel sowie Buchbesprechungen.

möglichkeiten zu charakterisieren. Die grundlegendste Eigenschaft ist die Erweiterbarkeit, daher lässt sich Eclipse vielleicht am treffendsten als Ablaufplattform und Framework für seine eigenen Erweiterungen in Form von Plug-Ins bezeichnen.

4.1.1 Deklarationselemente finden

Sogar seine bekannteste Verwendung, die Nutzung als Entwicklungsumgebung für die Programmiersprache Java, nutzt Eclipse mit dem Java-Development-Tooling[22] als Ablaufplattform für Plug-Ins, insofern das JDT selbst als Plug-In realisiert wurde. Mit dem JDT wird Eclipse um die Möglichkeit erweitert, Java-Quellcode zu erzeugen, zu analysieren, zu manipulieren und kompilieren. Das JDT stellt auch den bereits erwähnten AST zur Verfügung.[23] Dazu wird eine so genannte Kompilationseinheit – eine Java-Datei, die eine Java-Klasse beschreibt – gelesen und in eine Baumstruktur übersetzt, in der für jeden Ausdruck innerhalb des gelesenen Programmtextes ein dafür vorgesehener Knotentyp angelegt wird. Im Ergebnis enthält ein solcher AST Knoten für alle Import- und Cast-Anweisungen, alle Kommentare und Kontrollstrukturen wie If-Else-Blöcke, Methoden- und Variablendeklarationen sowie alle anderen Arten von Ausdrücken, die in Java möglich sind. Ein solcher AST lässt sich nun vergleichsweise komfortabel mit einem so genannten AST-Visitor nach bestimmten Elementen durchsuchen. Dazu enthält das JDT die abstrakte Klasse `ASTVisitor`, die für jeden Knotentyp über eine entsprechende visit-Methode verfügt, die aufgerufen wird, wenn ein Knoten des entsprechenden Typen besucht[24] wird. Überschreibt man nun in einer Spezialisierung dieser abstrakten Klasse in geeigneter Weise die Methoden, die beim Besuch der Knoten von Feldern, Methoden und Variablendeklarationen aufgerufen werden, so lässt sich ein Visitor zur Identifikation von Deklarationselementen innerhalb einer Java-Klasse realisieren.

---

[22] Im Folgenden als JDT abgekürzt

[23] Für den Zugriff auf den Abstract Syntax Tree in Eclipse siehe: [Kuhn / Thomann 2006]

[24] Die Begriffe ‚Visitor' und ‚besuchen' entspringen dem Umfeld des Visitor-Patterns. Dieses Architekturmuster unterscheidet zwischen der Implementation der Baum-Traversion auf der einen und dem Programmteil – dem so genannten Visitor, der kodiert, was ein Knotenbesuch zur Folge hat – auf der anderen Seite. Ein solcher Aufbau hat den Vorteil, dass insbesondere die Implementation der Traversion für unterschiedlichste Visitorexemplare wieder verwendet werden kann. Für eine detaillierte Darstellung dieses Entwurfsmusters (in der deutschen Ausgabe „Besucher" genannt) siehe [Gamma u. a. 2001].

Der folgende Visitor

```
public class DEVisitor extends ASTVisitor {
…
public void endVisit(FieldDeclaration node) {
      for (int i = 0; i < node.fragments().size(); i++) {
            VariableDeclarationFragment fragment =
            (VariableDeclarationFragment)fragments.get(i);
                  IVariableBinding binding =
                        fragment.resolveBinding();
                  ITypeBinding type = binding.getType();
                  String name = binding.getName();
       }
}
…
```

würde beispielsweise den Typ sowie den Namen von Deklarationselementen innerhalb aller Felddeklarationen einer Klasse identifizieren.[25] Ein solcher Visitor muss nur mit dem vom JDT zur Verfügung gestellten ASTParser auf die CompilationsUnit, die geparst werden soll, wie folgt angewendet werden:

```
      ASTParser parser = ASTParser.newParser(AST.JLS3);
      parser.setSource(unit);
      parser.setResolveBindings(true);
      CompilationUnit compilationUnit = (CompilationUnit)

      parser.createAST(null);
      DEVisitor devis = new DEVisitor();
      compilationUnit.accept(devis);
```

Anschließend werden die Felddeklarationen der Java-Klasse, die in `compilationUnit` enthalten sind, auf die beschriebene Weise ausgelesen.

Auf die hier skizzierte Art lässt sich die eingangs erwähnte Anforderung, Deklarationselemente in Programmtexten zu identifizieren, lösen. Um einen Zuweisungsgraphen zu er-

---

[25] Eine Felddeklaration besteht deshalb aus mehreren VariableDeclarationFragments, weil ein Feld auch z.B. wie folgt definiert sein kann: `private int a,b,c;`. In diesem Fall besteht die Felddeklaration aus drei Fragmenten. In allen anderen Fällen – wie etwa: `StringBuffer buffer;` – ist die Anzahl der Fragmente eins.

zeugen, müssen nun zusätzlich die Zuweisungen zwischen den Deklarationselementen erkannt werden. Auch dafür liefert der ASTVisitor eine visit()-Methode, die eine AST-Node vom Typ `Assignment` bekommt, die das linke und rechte Element der Zuweisung enthält.
Diese Aufgabe – ein Deklarationselement sowie dessen Zuweisungen zu ermitteln – wurde in dieser Arbeit unter Rückgriff auf das Infertype-Plug-In gelöst,[26] das seinerseits die hier vorgestellten Methoden des JDT zur Ermittlung der Zuweisungen verwendet.

4.1.2 Die Benutzeroberfläche

Mit seinem Standard-Widget-Toolkit verfügt Eclipse über eine eigene Sammlung von Bibliotheken zur Programmierung von graphischen Benutzerschnittstellen. Diese funktionieren ähnlich wie die in der Java-Welt bekannten UI-Frameworks Abstract-Window-Toolkit oder Swing und werden üblicherweise zur Entwicklung von Benutzerschnittstellen für Plug-Ins verwendet.
Die Benutzeroberfläche von Eclipse besteht grundsätzlich aus einer Anzahl kleinerer Fenster, die gleichzeitig unterschiedlichste Inhalte darstellen können. Benutzt man Eclipse zum Beispiel als Entwicklungsumgebung für Java, so stehen innerhalb der verfügbaren Fenster einige typische Funktionen bereit, wie etwa die Baumdarstellung der Package-Hierarchie, einen Sourcecode-Editor, eine Übersicht über die Importe, Felder und Methoden der aktuell geöffneten Java-Klasse und etliches mehr. Diese Oberfläche wird von dem oben erwähnten JDT bereitgestellt. Interessant im Rahmen dieser Arbeit ist nur, dass Plug-Ins die vorhandene Benutzeroberfläche durch weitere Funktionen innerhalb beliebiger Fenster erweitern können, insofern diese ihre Erweiterbarkeit vorgesehen haben. Mithilfe des Eventsystems von Eclipse – bestehend aus den so genannten Actions – können die einzelnen Anwendungsbestandteile miteinander kommunizieren. Beispielsweise ist es möglich, innerhalb eines Plug-Ins auf eine Textselektion innerhalb des Sourcecode-Editors aus dem JDT – also einem anderen Plug-In – zu reagieren, um einen beliebigen Aspekt dieser Selektion in einem anderen Fenster darzustellen. Das dieser Arbeit zugrunde liegende Infername-Plug-In macht genau das. Nach der Selektion eines Deklarationselementes innerhalb des Sourcecode-Editors des JDT und dem Aufruf von „Infername" wird der Zuweisungsgraph in einem durch das Infername-Plug-In zur Verfügung gestellten Fenster dargestellt.

---

[26] Vgl. Kapitel: 4.3 Infername-Plug-In.

4.2 Infertype

Das dem Infername-Plugin zugrunde liegende Infertype-Plug-In erzeugt so genannte Constraintgraphen, aus denen das Infername-Plugin die Zuweisungsgraphen extrahiert. Das folgende Kapitel beschreibt das Infertype-Plug-In und führt in die Typeconstraints ein. In einem weiteren Kapitel werden die aus dem zugrunde liegenden Plug-In übernommenen Einschränkungen dargelegt.

4.2.1 Typeconstraints

Unter der Bezeichnung Infertype versteht die vorliegende Arbeit vornehmlich die Version 3 des am Lehrstuhl von Herrn Prof. Friedrich Steimann, Fernuniversität Hagen, entwickelten Refactoring-Plug-Ins,[27] das dem hier entwickelten Infername-Plug-In als Grundlage zur Ermittlung des Zuweisungsgraphen dient.

Vorraussetzung dieser Entwicklung war ein von Steimann[28] unter dieser Bezeichnung vorgeschlagenes Refactoring, das den Typen eines beliebigen Deklarationselementes durch dessen maximal verallgemeinerten Typen[29] ersetzt. Steimann beschreibt dort einen Algorithmus zur Einführung eines inferierten – also gegebenenfalls noch nicht vorhandenen – Typen in das bestehende Typsystem. Der Algorithmus zur Berechnung des inferierten Typen für ein Deklarationselemente `a` ist die in Kapitel „2.2.5 Protokoll, AccessSet und inferierter Typ“ eingeführte Funktion ι(`a`).

Diese Funktion sowie der Algorithmus zur Einführung des maximal verallgemeinerten Typen wurde in einem Plug-In realisiert, das auf die oben skizzierte Art den AST zur Identifikation der Deklarationselemente nutzte, daraus die geforderten direkten und indirekten Verweise ermittelte und schließlich unter Verwendung des Steimannschen Algorithmus den inferierten Typen in das bestehende Typsystem einführte.[30]

Die aus der Einführung der Generics resultierenden Änderungen am Typsystem von Java ändern die Voraussetzung vor allem für den Algorithmus von ι(`a`). Wird beispielsweise ein Deklarationselement `a` einer mit dem Typen `T` typparametrisierten Liste hinzugefügt, so kann dieses Listenelement zu einem späteren Zeitpunkt ohne eine explizite Typumwand-

---

[27] In [Kegel 2007] wird dieses Plug-In beschrieben.

[28] Siehe: [Steimann 2007].

[29] Zur Begrifflichkeit siehe: Kapitel „2.2.5 Protokoll, AccessSet und inferierter Typ“.

[30] Eine Zusammenfassung aller drei bisherigen Implementationen von Infer-Type-Plug-Ins findet sich unter: http://www.intoj.org/index.php/Infer_Type.

lung der Liste wieder entnommen und einem zweiten Deklarationselement b zugewiesen werden.

```
private List<T> list = new ArrayList<T>();
void foo() {
      T a = new T();
      list.add(a);
      T b = list.get(0);
      b.m1();
}

 class T {
     public void m1(){};
      public void m2(){};
 }
```

Die Methode m1(), die hier auf der Referenz b aufgerufen wird, gehört ebenfalls in das Protokoll des maximal verallgemeinerten Typen der Referenz a, und das, obwohl b und a einander nicht zugewiesen sind. Hier ist also erkennbar, dass die Typparametrisierbarkeit der Liste zusätzliche Typbedingungen impliziert, die vor der Javaversion 1.5 nicht existierten. In der genericsfreien Programmversion wäre die Referenz a nicht implizit, sondern explizit durch eine entsprechende durch den Java-Compiler erzwungene Typumwandlung nach der Entnahme aus der Liste typisiert worden.

Es gibt eine ganze Reihe ähnlicher Probleme im Zusammenhang mit generischen Typen, die ausführlich in [Kegel 2007][31] besprochen und hier nicht wiederholt werden. Im Ergebnis führten diese Probleme zu einer Reimplementierung und Ablösung von Infertype in der Version 2 zu der Version 3, die dieser Arbeit zugrunde liegt. Dabei wurde die Definition von ι(a) dahingehend modifiziert, dass neben den direkten und indirekten Zuweisungen nun im Falle von Typparametern auch Typbeziehungen weiterverfolgt werden, was über die bloße Zuweisung von Deklarationselementen hinausgeht.[32]

[31] Siehe dort v.a. die Kapitel 3.3 und 4.3.

[32] Vgl. hierzu [Kegel 2007], S.41ff.

Grundlegend in der aktuellen Version ist die Umstellung von Infertype auf die Verwendung so genannter Typ-Constraints.[33] Dabei werden die impliziten Typbedingungen, die den Sprachkonstrukten des Programmtextes zugrunde liegen, ermittelt und in Typrelationen ausgedrückt. Ein einfaches Beispiel für eine implizite Typbedingung ist das folgende:

```
t.m1();
```

Hier setzt der Methodenaufruf `m1()` auf einer Variablen `t` voraus, dass die Variable `t` mindestens mit dem Subtypen desjenigen Typen deklariert sein muss, dessen Protokoll die Methode `m1()` enthält. Dabei werden die verwendeten Sprachkonstrukte – wie Variable `t` oder Methodenaufruf `m1()` – als Constraintvariablen bezeichnet, deren Wert – man spricht hier üblicherweise von Belegung – mit dem impliziten Typ des Ausdrucks belegt ist. Im vorliegenden Beispiel wäre das

a) der Typ von `t`
b) der Typ, dessen Protokoll `m1()` enthält.

In [Kegel 2007][34] wird für alle möglichen Varianten von Constraintvariablen eine an [Tip u. a. 2003] und [Fuhrer u. a. 2005] angelehnte formale Schreibweise eingeführt, in der die beiden oben umgangssprachlich formulierten Constraintvariablen wie folgt notiert werden würden:

a) [t]
b) Decl(m1())

Diese formale Notationsweise macht die Angabe von Typbedingungen sehr kompakt. Bei Kegel werden dazu insgesamt nur sieben verschiedenen Typen von Constraintvariablen angegeben.[35]

Relevant sind nur solche Constraintvariablen, die in einer der beiden folgenden möglichen Typbeziehungen zu einander stehen:

1. $T_1 \leq T_2$: T_1 ist Subtyp von T_2 oder identisch mit T_2
2. $T_1 \equiv T_2$: T_1 ist identisch mit T_2

In der formalen Schreibweise würde die implizite Typbedingung des Ausdrucks `t.m1()` also folgendermaßen ausgedrückt:

$[t] \leq Decl(m1())$

---

[33] Das Thema wird in [Kegel 2007] umfangreich v.a. in den Kapiteln 3 und 4 erläutert. Typconstraints sind ebenso Grundlage der Eclipse-Refactorings „Extract Interface“ und „Use Supertype Where Possible“. Vgl. hierzu auch: [Tip u. a. 2003] und [Nielson u. a. 2005].

[34] [Kegel 2007], S.21f.

[35] ebenda

Alle Sprachkonstrukte mit impliziter Typbedingung lassen sich auf diese Weise als notwendige Typrelation (="Constraint") zwischen zwei Constraintvariablen ausdrücken.[36]
Für Typparameter wird bei [Kegel 2007] ein zusätzlicher Typ von Constraintvariablen – die Elementvariable – eingeführt, der die Belegung eines Typparameters mit einem Typargument repräsentiert.[37] Auf diese Weise können implizite Typbedingungen im Zusammenhang mit Typparametern wie im oben angeführten Beispiel der typparametrisierten Liste angegeben werden. Mit der formalen Notation

Elem(<tatsächliches Typargument>, <Typparameter>) [38]

lässt sich das Beispiel vereinfacht wie folgt notieren:

[a] ≤ Elem(List<T>,T) ≤ [b] ≤ [Decl(m1())]

Umgangssprachlich bedeutet das: Der Typ der Referenz `a` muss mindestens Subtyp des Typarguments der Listenelemente sein (sonst könnten sie dort nicht hinzugefügt werden), die ihrerseits mindestens Subtyp des deklarierten Typs der Referenz `b` sein müssen (sonst könnte das 0-te Listenelement nicht `b` zugeordnet werden), und `b` muss mindestens Subtyp des Typen sein, der die Methode `m1()` definiert (sonst könnte sie nicht auf der Referenz `b` aufgerufen werden). Auf diese Weise ist der implizite Typzusammenhang zwischen den Referenzen `a` und `b` ausgedrückt.[39]
Infertype erzeugt nun für das Deklarationselement, dessen Typ durch den maximal verallgemeinerten Typen ersetzt werden soll, alle Constraintvariablen mitsamt Constraints, um den kleinsten Typen zu finden, der eine gültige Belegung der Constraintvariablen ergibt. Für diese Aufgaben verfügt Infertype für alle möglichen Arten von Constraintvariablen über einen entsprechenden Typen.[40] Nachdem alle von der Auswahl eines Deklarationselementes betroffenen Kompilationseinheiten ermittelt wurden, werden diese mittels eines speziellen ASTVisitors[41] geparst, der die AST-Knoten visitiert, um dabei für die vorgefundenen Ausdrücke Constraints und Constraintvariablen zu erzeugen. Für den Besuch eines Zuweisungsknotens wird beispielsweise immer eine Subtypconstraint angelegt, weil die Zuweisung zweier Deklarationselemente `a=b;` voraussetzt, dass `b` mindestens Sub-

---

[36] In [Kegel 2007], S.22 sind alle Varianten tabellarisch angegeben.

[37] [Kegel 2007], S.26ff.

[38] Siehe [Kegel 2007], S.21.

[39] Das Beispiel enthält keine Constraints zum Typparameter der Liste, sondern beschränkt sich aus Gründen der Vereinfachung auf den Zusammenhang zwischen `a` und `b`.

[40] Tatsächlich erweitert und verwendet Infertype die Menge der Constraintvariablen des Refactoringframeworks aus dem JDT. Siehe hierzu [Kegel 2007], S.54ff.

[41] Vgl. hierzu Kapitel: 4.1.1 Deklarationselemente finden.

typ von a ist – oder formal ausgedrückt: $[b] \leq [a]$. Der folgende Quellcode[42] aus Infertype zeigt diesen Schritt.

```
@Override
public final void endVisit(final Assignment node) {
   ...
   setConstraintVariable(node, ancestor);
   if (ancestor != null && descendant != null) {
       model.createSubtypeConstraint(descendant,
                                              ancestor);
   }
}
```

Alle Constraints und Constraintvariablen, die für ein selektiertes Deklarationselement erstellt wurden, erzeugen insgesamt einen Constraintgraphen, aus dem in der vorliegenden Arbeit der Zuweisungsgraph extrahiert wird.[43]

4.2.2 Übernommene Einschränkungen

Mit der Verwendung des Constraintgraphen als Grundlage des Zuweisungsgraphen innerhalb des Analyse-Plugin werden einige Einschränkungen übernommen, die im Folgenden dargestellt werden.

1. Da es das Ziel von Infertype ist, für das ausgewählte Deklarationselement einen neuen Typen in Form eines Interfaces einzusetzen, wird die Erzeugung des Constraintgraphen abgebrochen, sobald sie auf einen Zugriff über eine öffentliche oder geschützte (protected) Feldvariable oder eine geschützte Methode trifft, denn alle diese Zugriffsvarianten sind in Interfaces nicht definierbar.[44]
2. Stößt Infertype auf Deklarationselemente, deren Typen es nicht ändern kann, weil sie zum Beispiel nur binär in Form einer Bibliothek vorliegen, wird die Erzeugung des Constraintgraphen ebenfalls unterbunden.[45]
3. Nach einem Downcast werden keine weiteren Zuweisungen von Deklarationselementen verfolgt.[46]

---

[42] Der Quellcode ist dem entsprechenden ASTVisitor org.intoJ.inferType3.internal.constraintModel.creation. InferTypeConstraintsCreator entnommen.

[43] Vgl. hierzu Kapitel 4.3 Infername-Plug-In.

[44] [Kegel 2007], S.41, S.47, S.51f. und S.82

[45] [Kegel 2007], S.51f.

[46] [Kegel 2007], S.69, S.51f. , S.79 und S.82

4. Deklarationselemente, die Collections oder Listen hinzugefügt werden, werden nicht weiterverfolgt. Zwar wird im Falle von typparametrisierten Listen ein möglicher Methodenaufruf auf einem Element der Liste noch verfolgt, aber eine Zuweisungskette über die jeweilige add() oder put() -Methode der Liste selbst wird nicht erstellt.[47]

Da das Infername-Plug-In den Constraintgraphen zur Erzeugung des Zuweisungsgraphen benötigt, bricht auch das Infername-Plug-In in den ersten beiden Fällen mit demselben Hinweistext ab, den auch das Infertype-Plug-In zur Erläuterung an den Benutzer ausgibt. Die beiden zuletzt genannten Punkte verkleinern die Zuweisungsgraphen in einem für das Infername-Plug-In sachlich nicht notwendigen Grund. Dies muss einstweilen als technisch bedingte Einschränkung hingenommen werden.

4.3 Infername-Plug-In

Das folgende Kapitel beschreibt die Funktionsweise und Benutzung des für die vorliegende Arbeit erstellten Infername-Plug-Ins, das auf den beiden zuvor beschriebenen Technologien Eclipse und Infertype aufbaut.

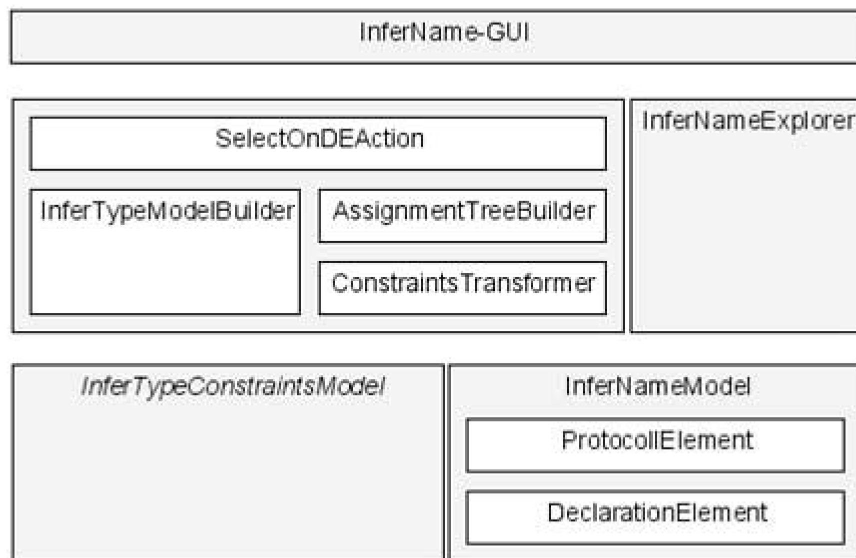
4.3.1 Aufbau

Die Aufgabe des Plug-Ins besteht darin, aus dem Constraintgraphen des Infertype-Plug-Ins den Zuweisungsgraph zu extrahieren und innerhalb von Eclipse darzustellen. Hier sollen nun die zentralen Komponenten des Infername-Plug-Ins mit ihren jeweiligen Rollen anhand eines vereinfachenden dreischichtigen Architekturmodell wie folgt dargestellt werden.[48]

---

[47] [Kegel 2007], S.51f.

[48] Das Kapitel richtet hauptsächlich sich an Leser, die an dem auf der beiliegenden CD befindlichen Quellcode des Plug-Ins interessiert sind. Vgl. auch den Anhang.



Architekturmodell von Infername

Das Architekturmodell zeigt auf der untersten Ebene die beiden für Infername relevanten Modelle `InferTypeConstraintsModel` und `InferNameModel`. Auf der mittleren Ebene sind links die vier Komponenten für die Modeltransformation und rechts die Komponente für Navigation innerhalb des `InferNameModel` angebracht. Der obere Balken `InferName-GUI` steht stellvertretend für alle weiteren Klassen, Ressourcen und Konfigurationen, die zur Funktionalität der Benutzeroberfläche des InferName-Plug-In beitragen, aber keine wesentliche Rolle innerhalb der Architektur spielen.

Eine der Hauptaufgaben ist die Modeltransformation von `InferTypeConstraintsModel` nach `InferNameModel`. Das Model von InferName besteht dabei lediglich aus den Klassen `DeclarationElement` und `ProtocollElement`, die die zugewiesenen Deklarationselemente bzw. die Methoden des zugehörigen AccessSets darstellen. Die Modeltransformation übernehmen die Klasse `InferTypeModelBuilder`, die durch Zugriff auf die entsprechenden Klassen von Infertype einen Constraintgraphen zurückliefert, und die Klasse `AssignmentTreeBuilder`, die ein Constraintmodel aus Infertype entgegen nimmt und mit Hilfe des `ConstraintsTransformer` daraus ein `InferNameModel` erzeugt. Diese Transformation wird angestoßen, indem der Benutzer über eine Textselektion im Sourcecodeeditor des JDT im Kontextmenü den Eintrag „Infername“ auswählt. Dieses Ereignis nimmt die Action `SelectOnDEAction` entgegen, die aus dem selektierten Programmtext eine ASTNode erzeugt und an den `InferTypeModelBuilder` weiterleitet.

`InferNameExplorer` ist die zentrale Controller-Klasse des InferNamePlug-Ins. Diese arbeitet auf dem vorliegenden Exemplar eines `InferNameModel` und ermöglicht die Darstellung und Navigation innerhalb des Zuweisungsgraphen in einem Fenster von Eclipse.

4.3.2 Modeltransformation

Einen Zuweisungsgraphen aus einem Constraintgraphen von Infertype zu extrahieren bedeutet im Wesentlichen, den Constraintgraphen um all jene Knoten zu reduzieren, die Typen von Constraintvariablen enthalten, die für einen Zuweisungsgraphen nicht relevant sind. Relevant sind dabei nur solche Constraintvariablen, die Programmausdrücke vom Typ Variablen-, Feld- oder Parameterdeklaration sowie Methodenrückgaben – also Deklarationselemente – repräsentieren. Alle anderen Typen von Constraintvariablen, wie etwa die in Kapitel 4.2.1 „Typeconstraints" vorgestellte Elementvariable, sowie die zu ihnen führende Typrelation sind nicht Bestandteil des Zuweisungsgraphen. Von allen 23 entweder durch das JDT zur Verfügung gestellten oder Infertype eingeführten Constraintvariablen[49] sind dies konkret die Typen `VariableVariable2` für Variablendeklarationen, `ReturnTypeVariable2` für Methodenrückgaben und `ParameterTypeVariable2` für Parameterdeklarationen. Diese entsprechen in der formalen Schreibweise der schon kennen gelernten Notation [t], für den Typen eines Deklarationselementes, der Notation [M], für den Rückgabetypen einer Methode M und [Param(i,M)] für den Typen des i-ten formalen Parameters der Methode M. Neben den bereits aufgeführten Constraintvariablen Decl(M) und Elem(E,T) gibt es in Infertype noch eine Reihe weiterer, die aber alle auf implementierungstechnische Gründe zurückgehen.[50]

Da Infertype im Falle von Zuweisungen zwischen Deklarationselementen Subtyp- und Gleichheitsconstraints zwischen diesen drei genannten Constraintvariablen anlegt, können diese Typrelationen auch als Zuweisungen gelesen werden. Dabei im Zusammenhang mit den drei relevanten Constraintvariablen zwischen drei Fällen zu unterscheiden, die in den folgenden drei Kapiteln dargelegt werden. Zur Illustration des von Infertype erzeugten Constraintsystems wird die Funktion „Constraint Graph Export" des Infertype-Plug-Ins verwendet. Innerhalb dieser Graphen werden Subtypconstraints als Pfeile und Typgleichheitsconstraints als gestrichelte Linien dargestellt. In den jeweiligen Knoten stehen die von Infertype verwendete Constraintvariable sowie deren Ausgangsbelegung.

---

[49] Infertype verwendet die Constraintvariablen des JDT, führt aber auch zusätzliche ein. Vgl. [Kegel 2007], S.57.

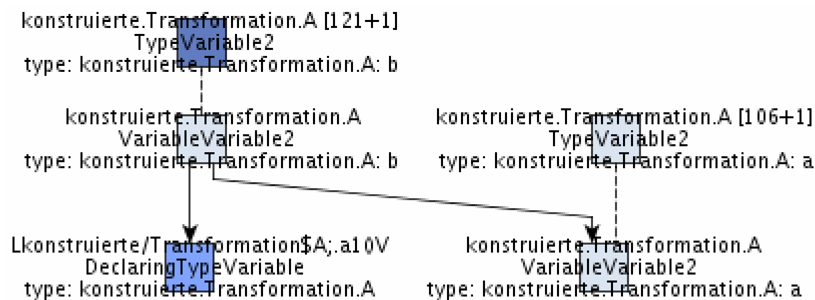
[50] Siehe [Kegel 2007], S.21ff.

4.3.2.1 Variablenzuweisungen

Für den folgenden Beispielcode

```
A a  = null;
A b = null;
b.a1();
a = b;
```

erzeugt Infertype für das Deklarationselement `b` den folgenden Constraintgraphen:



Dabei werden die beiden Variablendeklarationen durch die beiden Constraintvariablen vom Typ `VariableVariable2` und der Methodenaufruf `a1()` auf der Referenz `b` durch den Typ `DeclaringTypeVariable` repräsentiert. Letzterer gibt den zugehörigen Typ des verwendeten Protokolls an und entspricht in seiner Funktion dem, was in Kapitel 4.2.1 „Typeconstraints" mit der formalen Schreibweise Decl(a1()) eingeführt wurde. In der formalen Notationsweise würde die zu erwartende Typrelation $[b] \leq Decl(a1())$ und $[b] \leq [a]$. lauten. Eine Eigenart des Constraintgraphen von Infertype ist es nun, jeder `VariableVariable2` eine `TypeVariable2` über einen Gleichheitsconstraint zuzuordnen.[51] Reduziert man den Constraintgraphen nun auf die darin enthaltenen relevanten Constraintvariablen – im vorliegenden Fall nur die beiden vom Typ `VariableVariable2` –, so ergibt sich der gewünschte Zuweisungsgraph, bestehend aus der Zuweisung `b:A` →`a:A`.

4.3.2.2 Zuweisungen von Methodenrückgaben

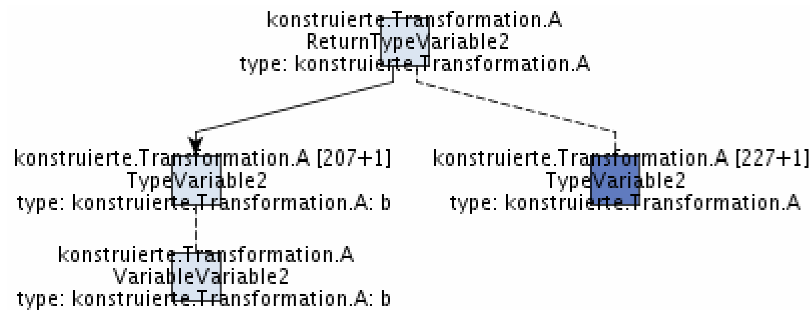
Ein ganz ähnliches Szenario ergibt sich im Falle von Methodenrückgaben, für die die zuweisungsgraphrelevante Constraintvariable `ReturnTypeVariable2` verwendet wird. Der folgende Beispielcode mit einer Methodenrückgabe

---

[51] Diese Zuweisung hat rein implementierungstechnische Hintergründe und soll hier nicht weiter beschäftigen. Begründet wird diese Konstruktion in [Kegel], S.60.

```
A b = gibA();
...
A gibA() {
      return new A();
}
```

erzeugt für das Deklarationselement `gibA` den folgenden Constraintgraphen:



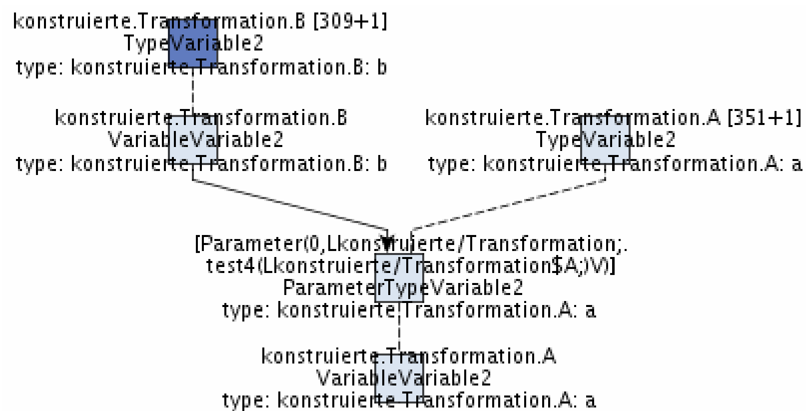
In formaler Notation ausgedrückt: [gibA()]≤[b]. Hier enthält der Constraintgraph von Infertype zunächst einen Subtypconstraint zur `TypeVariable2`, gefolgt von einem Gleichheitsconstraint zu der interessierenden `VariableVariable2` für die Referenz `b`, der die Methode `gibA()` zugewiesen ist. Im Falle einer `ReturnTypeVariable2` als Ausgangsknoten muss grundsätzlich der Gleichheitsconstraint über die `TypeVariable2` hinweg verfolgt werden, um zur zugewiesenen Referenz zu gelangen. Im Beispiel führt also die rot dargestellte Verfolgung der Constraintvariablen zum gewünschten Zuweisungsgraphen für das obige Codebeispiel `gibA:A`→ `b:A`.

4.3.2.3 Zuweisungen an Parameterdeklaration

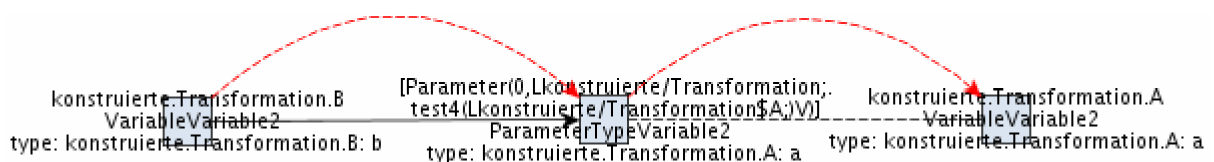
Für Zuweisungen an Parameterdeklaration muss ein Subtypconstraint zu einer `ParameterTypeVariable2` sowie ein Gleichheitsconstraint zu der zugehörigen `VariableVariable2` verfolgt werden. Der Constraintvariable vom Typ `ParameterTypeVariable2` wird hier nur die Information entnommen, dass es sich um eine Zuweisung an oder von einem Methodenparameter handelt. Für den folgenden Beispielcode

```
B b = null;
test4(b);
void test4(A a) { … }
```

ergibt sich für das Deklarationselement `b` der folgende Constraintgraph:



In der formalen Notation würde die Typrelation wie folgt ausgedrückt: [b] ≤[Param(1,test4(A a))]. Auch in diesem Fall werden zunächst alle Gleichheitsconstraints zu TypeVariable2 weggelassen. Eine weitere Eigenart von Infertype ist hier, den Methodenparameter durch einen Subtypconstraint zu einer Constraintvariablen vom Typ `ParameterTypeVariable2` und einem Gleichheitsconstraint zu einer Constraintvariablen vom Typ `Variable2Variable2` auszudrücken. In diesem Fall führt diese Form der Verfolgung zu dem gewünschten Zuweisungsgraph für den obigen Quellcode `b:B →a:A`.. Der `ParameterTypeVariable2` ist dabei lediglich die Information entnommen worden, dass es sich um einen Parameter handelt.



4.3.2.4 Algorithmus

Nachdem die drei unterschiedlichen Navigationsarten im Falle von Variablendeklarationen, Methodenrückgaben und Parameterdeklarationen auf die oben beschriebene Weise geklärt sind, ist der Algorithmus zur Erzeugung eines Zuweisungsgraphen aus einem gegebenen Constraintgraphen vergleichsweise leicht zu beschreiben. Er besteht aus zwei Schritten, wobei der erste alle Constraintvariablen und der zweite alle Constraints durchläuft. Neben diesen beiden Iterationen benötigt der Algorithmus außerdem eine Liste von

Schlüssel-Wert-Paaren, in der ein erzeugtes Deklarationselement als Wert sowie die zugehörige Constraintvariable als Schlüssel eingetragen wird:

Schritt 1: Iteriere über alle Constraintvariablen

- Handelt es sich um eine Constraintvariable vom Typ `VariableVariable2` oder `ReturnTypeVariable2` oder `ParameterTypeVariable2`, dann erzeuge daraus ein Exemplar von `DeclarationElement` als Modelrepräsentant eines Deklarationselementes innerhalb des `InferNameModel` und füge dies der Liste aller erzeugten Deklarationselemente unter Angabe der verwendeten Contraintvariable hinzu.[52] Dabei lassen sich alle Eigenschaften eines `DeclarationElement`es – wie Name, Typ und AccessSet – mit Hilfe des InferTypeConstraintsModel aus einer Constraintvariablen ermitteln.

Schritt 2: Iteriere über alle Constraints[53]

- Entnehme dem Constraint die linke Constraintvariable – das ist im Falle einer Zuweisung die Ausgangsseite – und prüfe in der Liste von Deklarationselementen, ob für diese Constraintvariable ein Exemplar vom Typ `DeclarationElement` angelegt wurde. Falls ja, suche mit der Constraintvariablen auf der rechten Seite nach einem zugewiesenen Deklarationselement, ebenfalls in dieser Liste. Um die Constraintvariable zu finden, unter der das Deklarationselement gegebenenfalls in der Liste gespeichert wurde, muss die in den Kapiteln 4.3.2.1 bis 4.3.2.3 verwendete Graphverfolgung in Abhängigkeit vom Typ der Constraintvariablen verwendet werden.
- Konnte ein Deklarationselement gefunden werden, füge dies dem Deklarationselemente für die linke Seite als Zuweisung hinzu.

4.3.3 Benutzung

Diese Kapitel beschreibt knapp die Anwendung des Infername-Plug-Ins. Eine Installationsanleitung sowie eine Beschreibung der konkreten Anwendung anhand einer Beispielklasse befinden sich im Anhang der Arbeit.

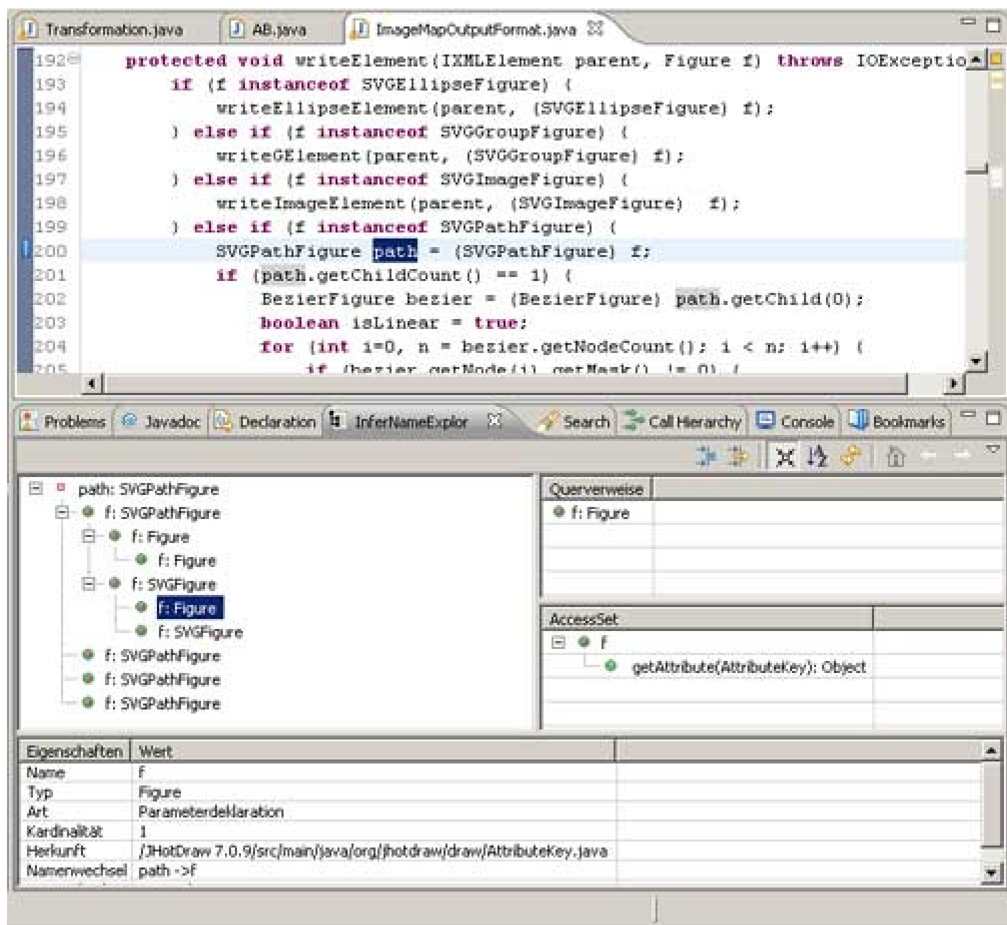
---

[52] Im Falle von `ParameterTypeVariable2` wird die über den Gleichheitsconstraint erreichbare `VariableVariable2` zusammen mit dem Deklarationselement abgespeichert. Vgl. hierzu Kapitel 4.3.2.3 „Zuweisungen an Parameterdeklaration".

[53] Genau genommen reichen alle Subtypconstraints, weil für Zuweisungen von Deklarationselementen immer Subtypconstraints erzeugt werden. Siehe Codebeispiel aus dem InferTypeConstraintsCreator auf S.37.

Das Infername-Plug-In wird wie Infertype über das Kontextmenü einer Textselektion innerhalb des Sourcecodeeditors von Eclipse aufgerufen. Nachdem das Infername-Plug-In den Zuweisungsgraphen ermittelt hat, wird dieser in einem eigenen Fenster als Baum dargestellt. Dabei stellen die Knoten des Baumes die Deklarationselemente und die Kanten die Zuweisungen dar. Der Doppelklick auf einem beliebigen Deklarationselement des Zuweisungsgraphen führt zur Programmtextstelle innerhalb des Sourcecodeeditors, die das Deklarationselement enthält. In einem weiteren Fenster wird das jeweilige AccessSet des selektierten Deklarationselementes angezeigt, das über einen Doppelklick auf einer dargestellten Methode ebenfalls zur Deklarationsstelle innerhalb des Sourcecodeeditors verlinkt. Zusätzlich werden einige Eigenschaften des selektierten Deklarationselementes – beispielsweise, ob es sich um einen Parameter, eine lokale Variable, ein Feld oder eine Methodenrückgabe handelt – in einem weiteren Fenster angezeigt.

Da es sich bei den Zuweisungsgraphen nicht um Bäume handelt, insofern alle Knoten mehreren Vaterknoten zugeordnet sein können, musste ein weiteres Fenster eingeführt werden, in dem alle zugewiesenen Deklarationselemente eines selektierten Deklarationselementes dargestellt werden, die in der Baumdarstellung schon über andere Zuweisungen dargestellt wurden. Ein Doppelklick in diesem Fenster führt zu einer Selektion des Deklarationselementes innerhalb der Baumdarstellung und zugleich zur Selektion der Programmtextstelle innerhalb des Sourcecodeeditors.



Screenshot des Infername-Plug-Ins

Aufgrund der geschilderten Nachteile dieser Darstellung wurde zusätzlich ein Export des Graphen in ein Dateiformat implementiert, das mit der frei verfügbaren Anwendung yEd[54] in einer adäquaten Graphendarstellung mit beliebigem Layout dargestellt werden kann. Alle Darstellungen von Zuweisungsgraphen innerhalb dieser Arbeit sind mit dieser Funktion erstellt worden. Der Export wird über das Kontextmenü der Baumdarstellung gestartet. Die Baumdarstellung selbst behält die Funktion der Navigation zwischen Zuweisungsgraph und Programmtextstellen.

---

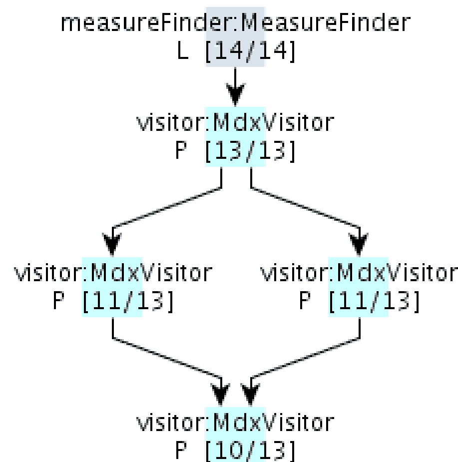
[54] http://www.yworks.com/en/index.html

5 Beispiele

Im Folgenden sollen exemplarisch eine Reihe Muster wiedergegeben werden, wie sie sich in der oben[55] beschriebenen Darstellungsform üblicherweise in Programmtexten finden lassen. Dabei sollen die gefundenen Beispiele unterteilt werden in solche, die die genannten Charakteristika deutlich machen, aber keinerlei Refaktorisierungsbedarf erkennen lassen, und solche, die sich anhand der genannten Überprüfungen als refaktorisierungsbedürftig erweisen. Erstere werden im Kapitel 5.1 „Standardmuster“ und letztere im Kapitel 5.2 „Refaktorisierungsmuster“ besprochen. Im Kapitel 5.3 „Gegenmuster“ sollen schließlich Beispiele angesprochen werden, die aufgrund der in dieser Arbeit besprochenen „Mustererkennung“ unerkannt oder gar falsch erkannt bleiben.
Als Grundlage wurden Programmtexte von Opensource-Projekten sowie von Projekten aus dem Arbeitsumfeld des Autors verwendet. Letztere wurden aus Gründen der Geheimhaltung verfremdet.

5.1 Standardmuster

Das erste Beispiel stammt aus dem Programmtext von `Mondrian`[56] und beginnt mit einer lokalen Variablen namens `measureFinder` aus der Klasse `RolapCube`.



In diesem Graphen gibt es insgesamt zwei Bezeichner (`measureFinder`, `visitor`) sowie zwei Typen (`MeasureFinder`, `MdxVisitor`). Während es sich bei dem Typen `MdxVisitor` um ein Interface handelt, ist `MeasureFinder` eine der Implementierungen

[55] Siehe Kapitel 3 „Problemstellung“.

[56] Mondrian ist ein Opensourceprojekt. Es handelt sich um einen in Java geschriebenen OLAP-Server – siehe: http://mondrian.pentaho.org/index.php.

dieser Schnittstelle. Das Deklarationselement `measureFinder` wird in der Klasse `RolapCube` angelegt und als konkretes Visitor-Exemplar einer Instanz der Klasse `Query` als Parameter übergeben, die dem Parsen eines so genannten MDX-Statement[57] dient. An dieser Stelle sind verschiedene Subklassen des Typs `QueryPart` dafür verantwortlich, das Statement in seine syntaktischen Bestandteile zu zerlegen – von denen jedes durch das vorliegende Exemplar vom Typ `MeasureFinder` visitiert wird. Alle vier Deklarationselemente der Form Methodenparameter werden in unterschiedlichen Klassen auf dieselbe Weise deklariert:

```
public Object accept(MdxVisitor visitor)
```

Alle diese Methoden akzeptieren also alle Typen von `MdxVisitoren`, während das Deklarationselement zu Beginn des Graphen einen konkreten Visitor verwendet. Soviel zum Inhalt des vorliegenden Szenarios, nun geht es um die Eigenschaften des Zuweisungsgraphen.

Der Graph beinhaltet neben dem Bezeichnerwechsel auch einen Typwechsel, der genau an der Stelle erfolgt, an der auch der Namenswechsel geschieht. Laut Annahme deutet der Bezeichnerwechsel mit gleichzeitigem Typwechsel auf die Zuweisung einer Rolle oder auf eine Generalisierung hin. Für ersteres spricht hier:

- `MdxVisitor` ist ein Interface, `MeasureFinder` eine Klasse – das spricht für eine Rolle im Falle von `MdxVisitor`.
- `MdxVisitor` ist schon aufgrund des zugrunde liegenden Visitor-Patterns als Rolle definiert – insofern hier unter anderem `MeasureFinder`-Instanzen in ihrer Rolle als `MdxVisitoren` von anderen Instanzen verwendet werden.
- Das Objekt der Klasse `MeasureFinder` nimmt nur vorübergehend die Rolle des `MdxVisitor(s)` ein und zwar in seiner Beziehung zu verschiedenen Subtypen[58] von `QueryPart`, die Teile des MDX-Statements repräsentieren. Dies geschieht durch die Zuweisungen beginnend mit `queryExp.accep(…)` ....

```
MeasureFinder measureFinder =
        new MeasureFinder(this, baseCube, measuresLevel);
queryExp.accept(measureFinder);
modifiedMeasureList.addAll(measureFinder.
                                    getMeasuresFound());
```

[57] MDX ist eine sql-ähnliche Datenbankabfragesprache für OLAP-Datenbanken.

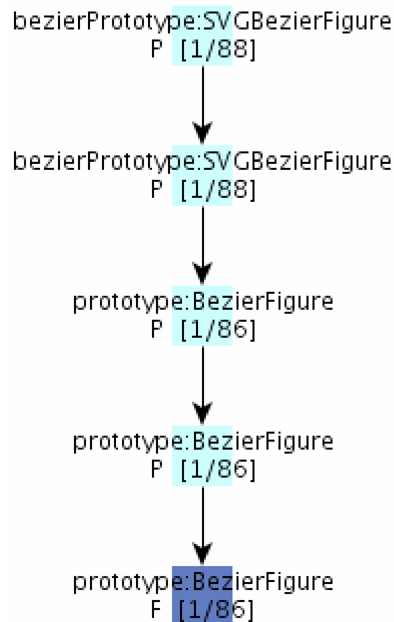
[58] nämlich Formula, Query, QueryAxis, Exp aus dem Package mondrian.olap.type

```
...
```

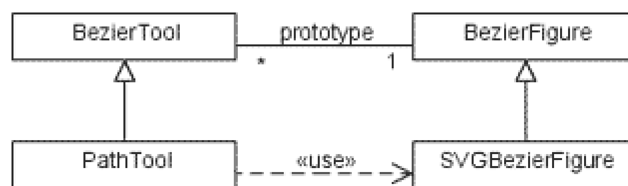
Anschließend wird mit `getMeasuresFound()` eindeutig das Protokoll des Typen `MeasureFinder` verwendet, da `MdxVisitor` über diese Methode gar nicht verfügt – das heißt, hier wurde die Rolle wieder verlassen.

Der Graph weist insgesamt vier inferierte Typen auf. Während die drei mit `visitor` benannten Deklarationselemente einen inferierten Typen haben, deren AccessSet Teil des Protokolls des Typen `MdxVisitor` sind, besteht das AccessSet von `measureFinder` mit der zusätzlichen Methode `getMeasuresFound()` aus genau dem Protokoll von `MeasureFinder`. Das bedeutet, für `measureFinder` ließe sich kein gemeinsamer Obertyp zusammen mit den anderen Deklarationselementen finden – `measureFinder` hat tatsächlich eine „umfangreichere Aufgabe" als seine Rolle als Visitor auszufüllen.
Damit lässt sich formulieren: `measureFinder` ist eine Instanz der Klasse `MeasureFinder`, die die Rolle eines `MdxVisitors` spielen können. Mit der Zuweisung von `measureFinder` zu `visitor` nimmt das Objekt die Rolle des `MdxVisitors` an. Der Bezeichnerwechsel begründet sich hier also durch eine Rollenzuweisung. Der zusätzliche Bezeichner führt somit zu einer Konkretisierung der Bedeutung der Deklarationselemente und erhöht eher die Verständlichkeit, als sie zu verschlechtern.

Im nächsten Beispiel wird eine Kette von Methodenparameterdeklarationen zuletzt auf eine Feldvariable namens `prototype` zugewiesen. Das Beispiel stammt aus dem Opensource-Projekt JHotDraw[59]



Der Graph besteht aus den Bezeichnern `bezierPrototype` und `prototype` sowie aus den Typen `SVGBezierFigure` und `BezierFigure`.

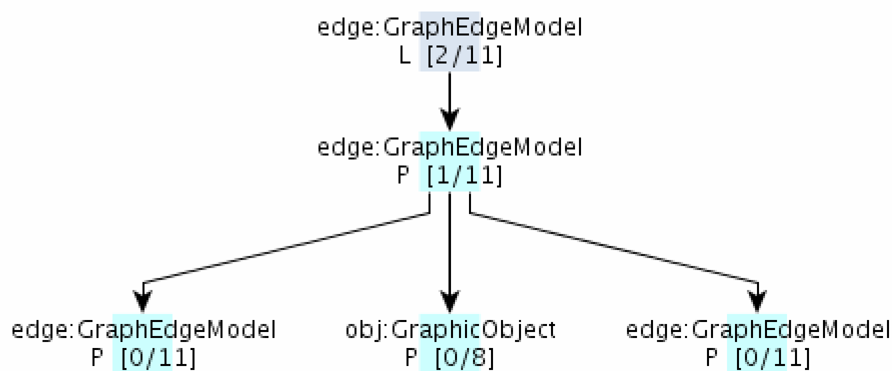


Während die beiden Deklarationselemente mit Namen `bezierPrototype` in der Klasse `PathTool` deklariert werden, werden alle mit `prototype` bezeichneten in der Klasse `BezierTool` definiert. `PathTool` selbst ist eine Spezialisierung von `BezierTool` und definiert in seinem Kontext auch eine speziellere Aufgabe für die Deklarationselemente innerhalb der gemeinsamen Typhierarchie von `BezierFigure` und `SVGBezierFigure`. Auch bei diesem Graph ist gut erkennbar, dass Bezeichner- und Typwechsel übereinander liegen. Das AccessSet aller Deklarationselemente besteht lediglich aus der in `BezierFigure` definierten Methode `clone()` – ein Wechsel des inferierten Typen liegt hier also nicht vor. Anders als bei dem vorhergehenden Zuweisungsgraphen handelt es sich hier aber nicht um eine Rollenzuweisung, sondern einfach um eine Generalisierung. In

[59] JHotDraw ist ein GUI-Framework für Graphikprogramme: http://sourceforge.net/projects/jhotdraw.

der allgemeineren Klasse `BezierTool` wird das Feld vom Typ `BezierFigure` definiert, das in der spezielleren Klasse `PathTool` verwendet wird, allerdings in der ebenfalls spezielleren Form `SVGBezierFigure`. Auch hier scheint die Namensgebung sinnvoll gewählt. Allerdings lässt sich die Generalisierung, die durch die Bezeichner zum Ausdruck kommt, hier offenbar eher auf die Verwendung innerhalb der Klassen `PathTool` und `BezierTool` als auf die deklarierten Typen zurückführen, denn der Namensbestandteil `bezier` ist nur in den Deklarationselementen enthalten, die in der Klasse deklariert sind, die auch `Beziertool` heißt.

Als nächstes soll ein Zuweisungsgraph aus dem Projekt Sinaxe[60] behandelt werden. Diese besteht aus den Typen `GraphEdgeModel` und `GraphicObject` sowie den Deklarationselementen `edge` und `obj`.



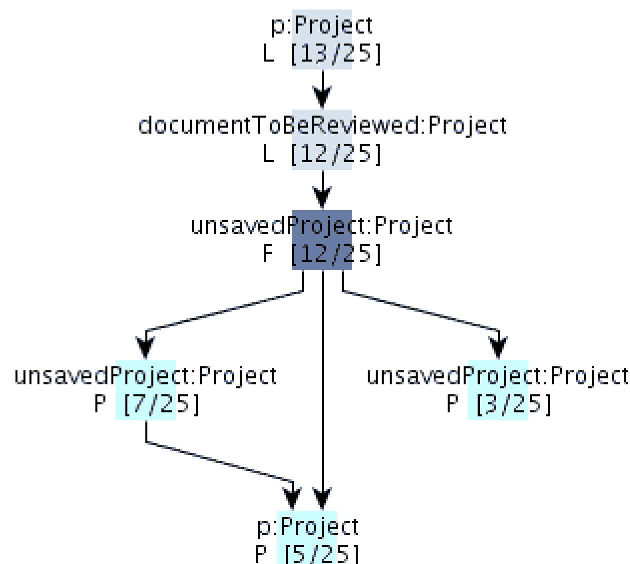
Sinaxe ist ein Framework, das graphisch darstellbare Komponenten bereitstellt, die XML-Datenstrukturen repräsentieren. Seine Hauptaufgabe besteht im Mapping von Struktureigenschaften und den typischen Anforderungen einer Benutzeroberfläche, solche Strukturen beispielsweise in Baumform darzustellen. Dementsprechend gibt es unter anderem die Typen `GraphEdgeModel` und `GraphNodeModel`, beides Subtypen von `GraphicObject`. Auch bei diesem Zuweisungsgraph liegen der Bezeichnerwechsel sowie der Typwechsel übereinander. Es handelt sich wieder um eine Generalisierung, diesmal mit der eingangs erwähnten typischen Übernahme von Namensbestandteilen aus dem Typbezeichner in die Namen der Deklarationselemente.[61]

---

[60] http://sinaxe.org/

[61] Vgl. hierzu Kapitel 2.2.3 „Namen von Deklarationselementen“.

Auch das folgende Beispiel stammt wieder aus dem Projekt JHotDraw. Diesmal erfolgt zwar ein Bezeichner-, aber kein Typwechsel.



Der Code behandelt das Beenden einer mit dem GUI-Framework erstellten Applikation. Hier geht es um die Behandlung gegebenenfalls noch nicht gespeicherter Projekte. Der Graph enthält die Bezeichner `documentToBeReviewed`, `unsavedProject` und `p` sowie den Typen `org.jhotdraw.app.Project`. Die Deklarationselemente mit dem Bezeichner `unsavedProject` befinden sich alle in der Klasse `ExitAction`, ebenso wie die lokale Variable `documentToBeReviewed`.

Bei der lokalen Variablen zu Beginn des Zuweisungsgraphen handelt es sich um eine Schleifenvariable, mit der über eine Liste von Projekten iteriert wird – `p` steht hier allgemein für den Typen `Project`.

```
for (Project p : app.projects()) {
      ...
```

Das Deklarationselement namens `p` am Ende des Graphen wird in `org.jhotdraw.app.Application` definiert

```
public void dispose(Project p).
```

und auch das bezieht sich allgemein auf Referenzen des Typs Project. Die anderen beiden Deklarationselemente beziehen sich auf die Prüfung noch nicht gespeicherter Projekte innerhalb der aktuellen Instanz von `Application` – und unterscheiden zwischen ungespeicherten Projekten und ungespeicherten Projekten, die außerdem dem Benutzer nochmals zur Speicherung vorgeführt werden sollen.

In dieser Kette von Bezeichnerwechseln (p → documentToBeReviewed → unsavedProject → p) wechselt die Instanz von Project ihren Zustand, bzw. jede Project-Instanz aus der Liste wechselt vom Status toBeReviewed in den Status unsaved und schließlich wieder nach Project ohne jede Statusangabe. Dieser Bezeichnerwechsel bringt, wie in Kapitel „2.2.3 Namen von Deklarationselementen“ dargelegt, einen Zustand des Objekts zum Ausdruck. Darin unterscheidet sich dieser Fall gegenüber dem Rollenwechsel zu Beginn dieses Kapitels[62], wo das Objekt gegenüber anderen Instanzen in der Form eines bestimmten (Rollen)Typen (dem MdxVisitor) auftrat. Während ein bestimmter Zustand, in dem sich ein Objekt im Verhältnis zu den Möglichkeiten seines Typen befindet, durch den Bezeichner lediglich annotiert wird, kann bei der Rolle das Verhältnis zu anderen Typen auf genau die Teilmenge des Protokolls reduziert werden, die dem Verhalten der Rolle entspricht. Dieser Umstand macht es auch nur bei der Rolle sinnvoll, zusätzliche Typen einzuführen, weil nur hier das Verhältnis zu anderen Typen – das zur Verfügung gestellte Protokoll – genauer spezifiziert werden soll.

Auch wenn dieser Zuweisungsgraph vergleichsweise viele Bezeichner aufweist, erscheint die Namenswahl sinnvoll. Alle Bezeichnerwechsel sind begründbar und erleichtern das Programmverstehen durch zusätzliche Kontextinformation, verglichen etwa mit einem Graphen, in dem jedes Deklarationselement mit p bezeichnet wäre.

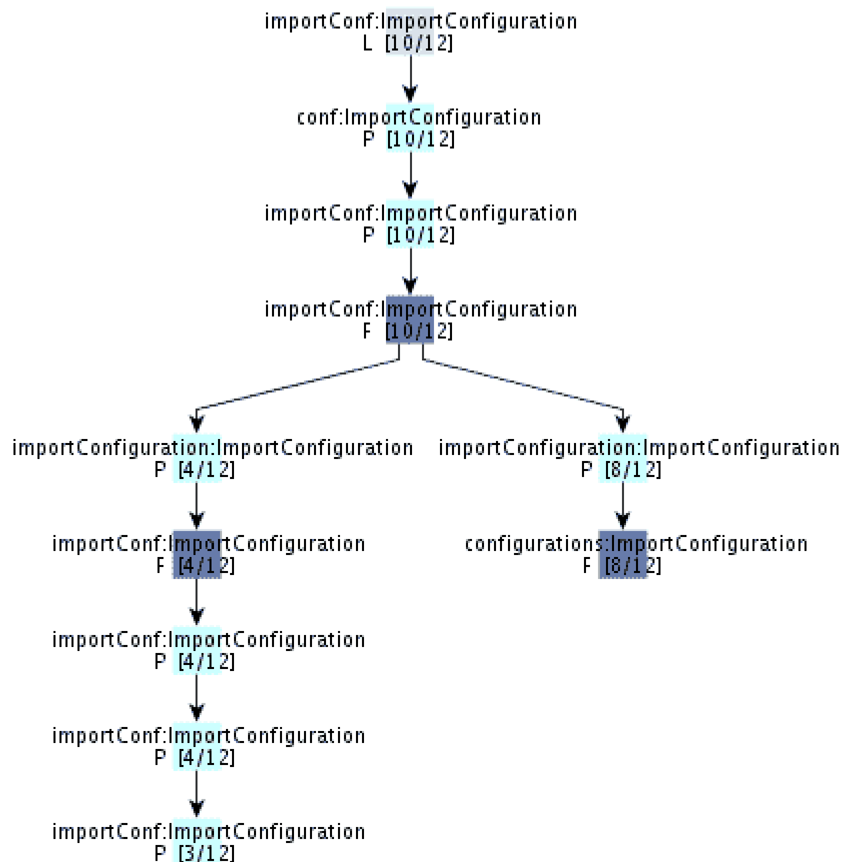
5.2 Refaktorisierungsmuster

Das erste Beispiel stammt aus einer Anwendung, die ein webbasiertes Berichtswesen zur Verfügung stellt. Der dem Zuweisungsgraph zugrunde liegende Programmtext behandelt den Start eines Datenimports, der über einen Datencontainer vom Typ ImportConfiguration parametrisiert wird. Die unmittelbar der ausführbaren Main-Methode folgende Methode, die das zweite Deklarationselement conf deklariert, hat die folgende Signatur:

```
public static void run(..., ImportConfiguration conf)
```

Diese Methode wird von verschiedenen ausführbaren Klassen, die über eine main-Methode verfügen, verwendet, die die abstrakte Klasse ImportConfiguration ihren jeweiligen Bedürfnissen entsprechend durch Subtyping erweitern und so den Import mit einer jeweils angepassten Konfiguration starten.

[62] Vgl. das erste Beispiel mit den Typen MeasureFinder und MdxVisitor.



Der Zuweisungsgraph entsteht, indem die Konfigurationsdaten vom äußeren Rand der Anwendung bis hin zu den Programmteilen, die diese Daten tatsächlich verarbeiten, weitergereicht werden. Er ist durchgehend mit `ImportConfiguration` typisiert und enthält die vier Bezeichner `importConf`, `conf`, `importConfiguration` und `configurations` und insgesamt sechs Bezeichnerwechsel. Abgesehen von den beiden Bezeichnerwechseln von `importConf` nach `importConfiguration` bleibt bei allen anderen der inferierte Typ vor wie nach dem Wechsel gleich. Auch auf die beiden Bezeichnerwechsel mit Wechsel des inferierten Typs folgen umgehend Bezeichnerwechsel mit gleich bleibendem inferierten Typ – in der linken Verzweigung kehrt die Teilfolge von Bezeichnerwechseln sogar zur Ausgangsbezeichnung zurück (`importConf → importConfiguration → importConf`). Beide Deklarationselemente mit Namen `importConfiguration` haben zudem untereinander einen unterschiedlichen inferierten Typen, obwohl sie gleich heißen – was ebenso darauf hinweist, dass ihre Einführung des zusätzlichen Bezeichners `importConfiguration` gegenüber den bereits eingeführten Bezeichnern vermutlich keinerlei zusätzliche Bedeutung mit sich bringt.

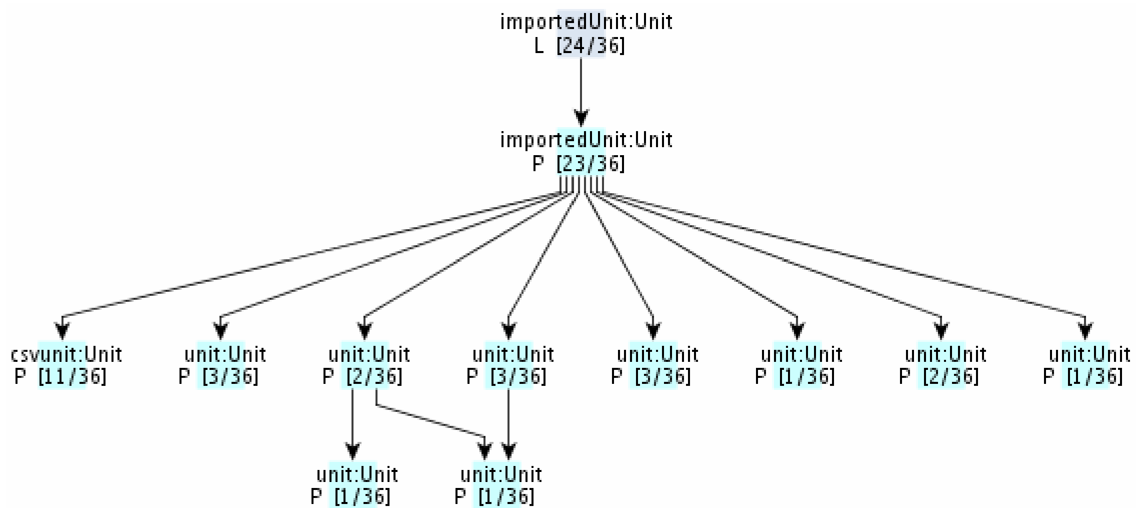
Zusammenfassend kann man feststellen: Keiner der vorhandenen Bezeichnerwechsel ist begründbar. Tatsächlich ließen sich alle Deklarationselemente durchgehend nach dem am häufigsten benutzten Bezeichner benennen – das wäre `importConf`.[63] Teilt man den Graphen in drei lineare Teilketten und besucht die Knoten des Baums in Zuweisungsrichtung, so wie es ein Algorithmus mache würde, so ließe sich an allen Knoten, in denen ein neuer Bezeichner eingeführt wird, ohne dass ein Wechsel des inferierten Typs (oder gar ein Typwechsel) stattfindet, ein „Unbegründet"-Eintrag vornehmen. Ausnahmen sind dabei die Übergänge in den linken und rechten Teilgraphen. Dabei könnte sich der linke Bezeichner noch als unbegründet herausstellen, wenn der Graph zusätzlich gegen die Zuweisungsrichtung gelesen würde. Damit fiele auf, dass der Bezeichner `importConf` viermal verwendet wird, um dann bei gleichem inferiertem Typ auf `importConfiguration` zu wechseln. Das Ergebnis dieser schematischen Vorgehensweise mit Einführung des häufigsten Bezeichners wäre ebenfalls eine durchgehende Benennung mit Ausnahme des rechten Teilgraphen, der entweder nach dem Bezeichner `importConfiguration` oder `configurations` benannt wird.
Ein durchgehender Bezeichner würde die Lesbarkeit des vorliegenden Codes verbessern. Wenn man berücksichtigt, dass die Deklarationselemente des Zuweisungsgraphen in insgesamt sieben verschiedenen Klassen, also sehr wahrscheinlich in sehr verschiedenen Programmteilen, deklariert werden, dann wäre es hilfreich, anhand eines durchgehenden Bezeichners direkt die Identität der Daten ablesen zu können.

Im Folgenden handelt es sich um einen Anwendungsteil, der Stammdaten für Verkaufsflächen – so genannte `Units` – aus Textdateien einliest. Diese Textdatei enthält pro Zeile semikolonsepariert alle Daten einer `Unit` und hat die Dateiendung .csv[64], was in die Namensgebung der Deklarationselemente eingegangen ist. Eine `Unit` repräsentiert dabei eine aktuell vorhandene Verkaufsfläche, die ihrerseits in einer hierarchischen Organisationsstruktur bestehend aus „Group", „Region" und „Headquarter" eingegliedert ist. Aus diesem Zusammenhang zeigt der vorliegende Graph ausgehend von einer lokalen Variablen namens `importedUnit` eine Reihe von Methodenparametern mit den Namen `im-`

[63] Leider ist hier der häufigste Bezeichner nicht der beste – importConfiguration würde den Programmtext unmissverständlicher lesbarer machen.

[64] csv = "*Comma-Separated Values*". Es handelt sich um einen bestimmten Aufbau einer Textdatei zur Speicherung einfach strukturierter Daten.

portedUnit, csvunit und unit, sowie keinerlei Typwechsel – alle Deklarationselemente sind vom Typ Unit.



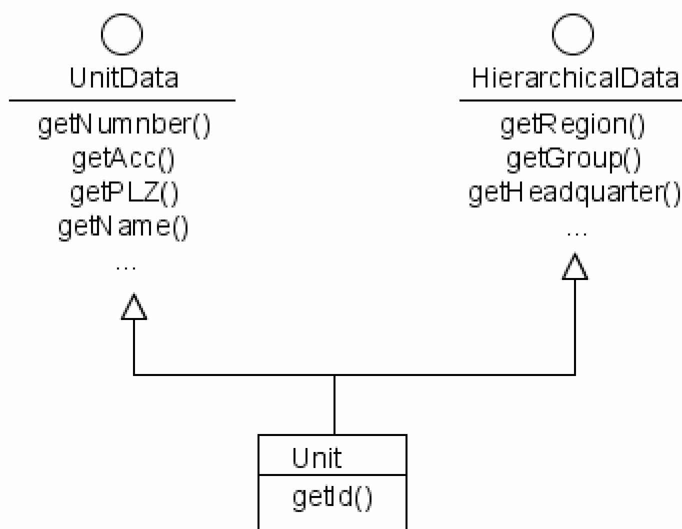
Der dem Graphen zugrunde liegende Programmtext hat die Aufgabe, aus den importierten Daten sowohl eine Unit-Entität als auch die zugehörige Organisationsstruktur zu erzeugen. Dies geschieht durch den Import der erwähnten Textdatei. Die einzelnen Zeilen dieser Datei enthalten jeweils alle Informationen für eine Unit, als da wären: Adresse, Name des Leiters, Kontoverbindung etc.. Neben den Daten, die sich auf eine Unit-Entität beziehen, gibt jede Zeile zusätzlich an, welchen organisatorischen Einheiten – Beispiele hierfür wären: „Group", „Region" und „Headquarter" – die Unit zugeordnet ist. Aus diesen Angaben erzeugt die Anwendung nicht nur die Unit-Instanzen, sondern auch die zugehörige Hierarchiestruktur der Unternehmerorganisation. Der Typ Unit enthält also Zugriffsmethoden für die einzelne Verkaufsfläche sowie für Daten hierarchisch höherer Organisationseinheiten, und genau diese Doppelfunktion kommt auch in der Namensgebung der Deklarationselemente zum Ausdruck. Während das Deklarationselement namens importedUnit noch das vollständige Importergebnis enthält, handelt es sich bei der Methode mit dem Parameter csvunit um die Methode, die die eigentlichen Unit-Daten liest. Alle anderen Methoden mit Parameter unit sind Methoden, die im wesentlichen Hierarchieinformationen verarbeiten. Die Doppelfunktion des Typen Unit hinsichtlich seiner Aufgabe liegt also zwischen den Bezeichnern csvunit und unit. Das zeigt sich besonders daran, dass – sieht man von der gemeinsamen Zugriffsmethode auf die Id des Objektes ab – das gemeinsame AccessSet aller Instanzen mit Namen unit und das AccessSet von csvunit disjunkt sind. Zum Vergleich seien hier die AccessSets von

csvunit einerseits und dem vereinigten AccessSet aller mit unit bezeichneten Referenzen aufgeführt.[65]

unit (zusammengefasst)	csvunit
getId()	getId()
getRegionManager ()	getPLZ()
getRegion()	getName()
getStart()	getNumber()
getEnd()	getStreet()
getGroupManager()	getAcc()
getGroup()	getAcc2()
getHeadquarter()	getState()
getCategory()	getCountry()
getEnd2()	getCC()
isAdministrated()	getDateOfChange()

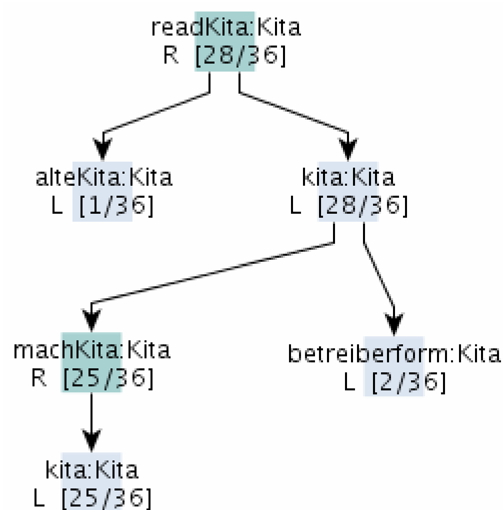
Im vorliegenden Programmtext bietet sich also an, die unterschiedliche Funktion, die hier Unit-Instanzen haben, auch durch unterschiedliche Typen zum Ausdruck zu bringen. Dies geschähe am einfachsten durch Einführung eines zusätzlichen Typen CsvUnit, bestehend aus den Methoden, auf die das Deklarationselement csvunit zugreift. Diese Refaktorisierung wäre leicht mit einer Ausführung von Infertype auf dem Deklarationselement csvunit durchzuführen. Eine weitergehende Refaktorisierung bestünde darin, den unterschiedlichen Rollen, die die importierten Unit-Daten hier gegenüber der Erzeugung sowohl von Unit-Instanzen als auch höheren Organisationsinstanzen spielen, durch die Einführung unterschiedlicher Typen Rechnung zu tragen. Angestrebt wäre dann etwa die folgende Typhierarchie,

[65] Da sich diese Information nicht aus dem dargestellten Graphen ergibt, wird in Kapitel 7 „Ausblick“ für die zusätzliche Darstellung der Schnittmengengröße zweier AccessSets plädiert.



die das Programmverstehen der betreffenden Codestelle deutlich erleichtert. Auch diese Refaktorisierung lässt sich mit Infertype ausführen, wenn man bei der Ausführung von Infertype auf einem der Deklarationselemente namens `unit` dem neuen Typen alle Methoden aus den AccessSets der mit `unit` bezeichneten Deklarationselemente hinzufügt.[66]

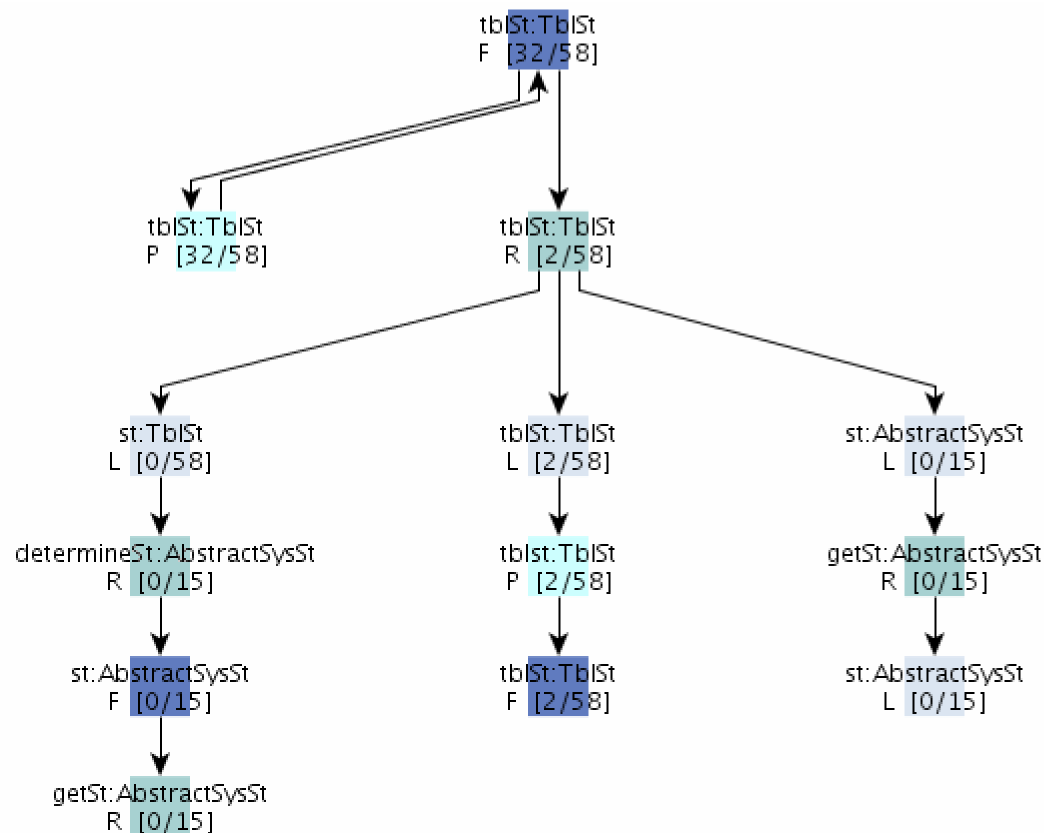
Das nächste Beispiel hat Ähnlichkeiten mit dem vorangegangen, insofern es ebenfalls aus dem Umfeld eines Stammdatenimports kommt. Diesmal aber werden nicht Verkaufsflächen, sondern Kindertagesstätten importiert. Der folgende Graph beginnt mit der `Kita`-Exemplare erzeugenden Methode `readKita`. Alle Deklarationselemente sind mit dem Typen `Kita` deklariert und heißen – vor allem, wenn man die Methode `machKita` hinzuzieht – mehrheitlich `Kita`.



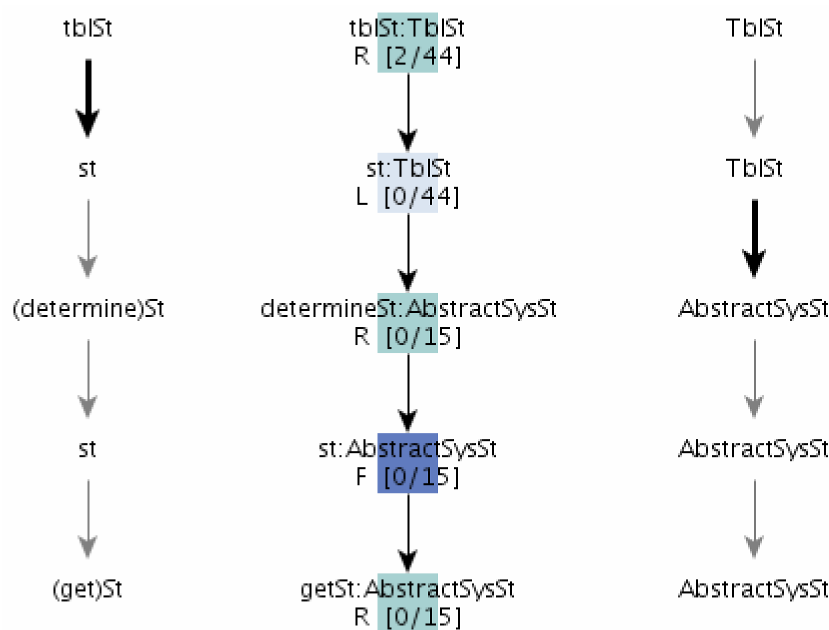
[66] Hierbei handelt es sich um ein Feature von Infertype – siehe: [Kegel 2007], S.9 f.

Auffällig ist, dass der Bezeichner `betreiberform` mit allen anderen verwendeten Bezeichnern keinerlei Ähnlichkeit hat, insofern hier „Kita“ überhaupt kein Namensbestandteil ist. Sieht man von der Referenz `alteKita` ab, die immerhin noch einen ähnlichen Bezeichner besitzt, dann fällt zudem auf, dass der inferierte Typ deutlich kleiner ist als bei den anderen Deklarationselementen. Tatsächlich handelt es sich bei Betreiberform auch begrifflich um eine Entlehnung aus der Fachdomäne, insofern darunter eine bestimmte Rechtsform der Kindertagesstädte verstanden wird. Jedes Kitaexemplar lässt sich nach einer dieser Rechtsformen kategorisieren (es wird nur zwischen privat und öffentlich unterschieden); und diese Kategorisierung ist genau an den beiden Eigenschaften unterscheidbar, die auf der Referenz `betreiberform` aufgerufen werden. An dieser Stelle wäre die Einführung eines Typs Betreiberform sinnvoll und entspräche der Begrifflichkeit der zugrunde liegenden Fachwelt.

Das nächste Beispiel stammt aus dem Umfeld einer Webanwendung, die den Klickpfad eines beliebigen Benutzers verfolgen muss. Dies geschieht über zwei unterschiedliche Klassen, `TblSt` und `SysSt`, die den gemeinsamen Obertypen `AbstractSysSt` haben. Beide haben die Aufgabe, sich navigationsrelevante URL-Parameter userspezifisch zu merken, so dass das bisherige Benutzerverhalten bekannt bleibt. Da die Anwendung im Wesentlichen aus der Navigation und einer Reihe tabellarischer Berichte besteht, gibt es für diese zwei Typen von Nutzerverhalten auch zwei Typen, die sich den Status merken. Der Zuweisungsgraph besteht aus den Typen `AbstractSysSt` und `TblSt` sowie den Bezeichnern `st`, `tblSt`, `getSt`() und `determineSt`.



Überwiegend sind die Deklarationselemente mit dem Typ `TblSt` als `tblSt` und die mit dem Typen `AbstractSysSt` als `st`, bzw. `getSt` im Falle von Methoden, bezeichnet worden. Eine Ausnahme bildet hier der Methodename `determineSt` vom Typ `AbstractSysSt`. Der mittlere sowie der rechte Teilgraph entsprechen den beschriebenen Erwartungen. Der mittlere Teilgraph weist keinerlei Typwechsel auf und hat folgerichtig auch keinen Bezeichnerwechsel. Beim rechten Teilgraphen fallen Typ- und Bezeichnerwechsel übereinander. Sieht man im linken Teilgraphen über die Ausnahme des Methodennamen `determineSt` hinweg und versteht `determine` und `get` als Synonyme, dann enthält er einen Bezeichner- und einen Typwechsel, die nicht übereinander liegen. An dieser Stelle wechselt der Bezeichner vor dem Typ, und in diesem Fall stellt sich wie beschrieben die Frage, ob der Bezeichnerwechsel später oder der Typwechsel früher stattfinden kann.



Wie am AccessSet bereits zu erkennen, lässt sich in diesem Fall der allgemeinere Typ früher einführen. Alle Deklarationselemente vom Typ `AbstractSysSt` haben ein AccessSet von 0, wohingegen alle Deklarationselemente vom Typ `TblSt` mindestens ein AccessSet von 2 haben. Für das betreffende Deklarationselement nach dem Bezeichnerwechsel kann also bereits der Typ `AbstractSysSt` eingeführt werden, genau wie sein Bezeichner das schon andeutet.

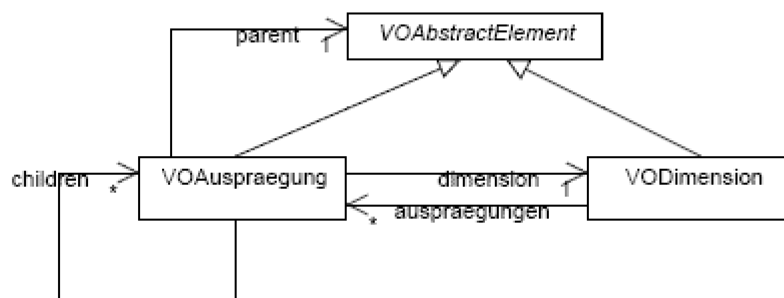
Der Refaktorisierungseffekt ist hier nicht besonders groß. Sieht man sich den Programmtext der Methode `determineSt()` an,

```
...
if (isTable()) {
      TblForm tblForm =TblForm.getInstance(request);
      TblSt st = tblForm.tblSt();
      return st;
}
return SysSt.getInstance(request);
...
```

dann fällt auf, dass das unterstrichene Deklarationselement, das verantwortlich für den „verfrühten" Bezeichnerwechsel ist, sogar wegfallen könnte, wenn die Methode direkt den Rückgabewert von `tblForm.tblSt()` zurückgäbe. Vielleicht weist es aber auf generellere Architekturschwächen hin – so ist die Behandlung der beiden Subtypen `TblSt` und `SysSt` offensichtlich nicht ganz so symmetrisch, wie man es erwarten würde. Indem der

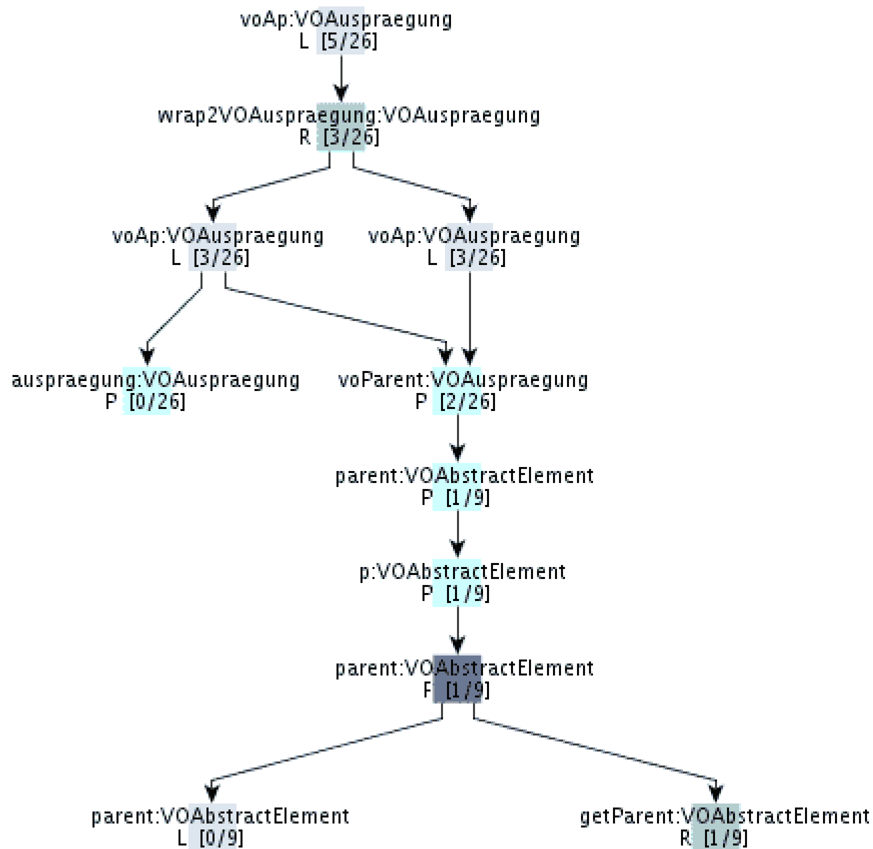
Name der Methode `determineSt()` mit State einen gemeinsamen Namensbestandteil der Typen `TblSt` und `SysSt` aufnimmt und so von deren Unterschieden abstrahiert, suggeriert er, dass Exemplare beider Typen auf die gleiche Art mit dem übergebenen `Request` initialisiert werden können. Tatsächlich aber sieht man, dass im Falle von `TblSt` dafür auf ein Objekt vom Typ `TblForm` zurückgegriffen werden muss, das im vorliegenden Fall ein Subtyp aus einer Fremdbibliothek ist.

Die Deklarationselemente des folgenden Beispielgraphen stammen aus einem Anwendungsteil, in dem mit Hilfe der Typen `VOAbstractElement`, `VODimension` und `VOAuspraegung` rekursive Baumstrukturen aufgebaut werden. Eine Vereinfachung der Typhierarchie zeigt das folgende Klassendiagramm.



Dimensionen sind dabei stets nur Wurzelelemente und Ausprägungen deren Kinder, die wiederum Ausprägungen als Kinder enthalten können. Beide Typen haben einen gemeinsamen abstrakten Obertypen, der die Grundeigenschaften eines solchen hierarchischen Elements definiert.

Der Beispielgraph enthält die Typen `VOAuspraegung` und `VOAbstractElement` sowie sieben unterschiedliche Bezeichner,



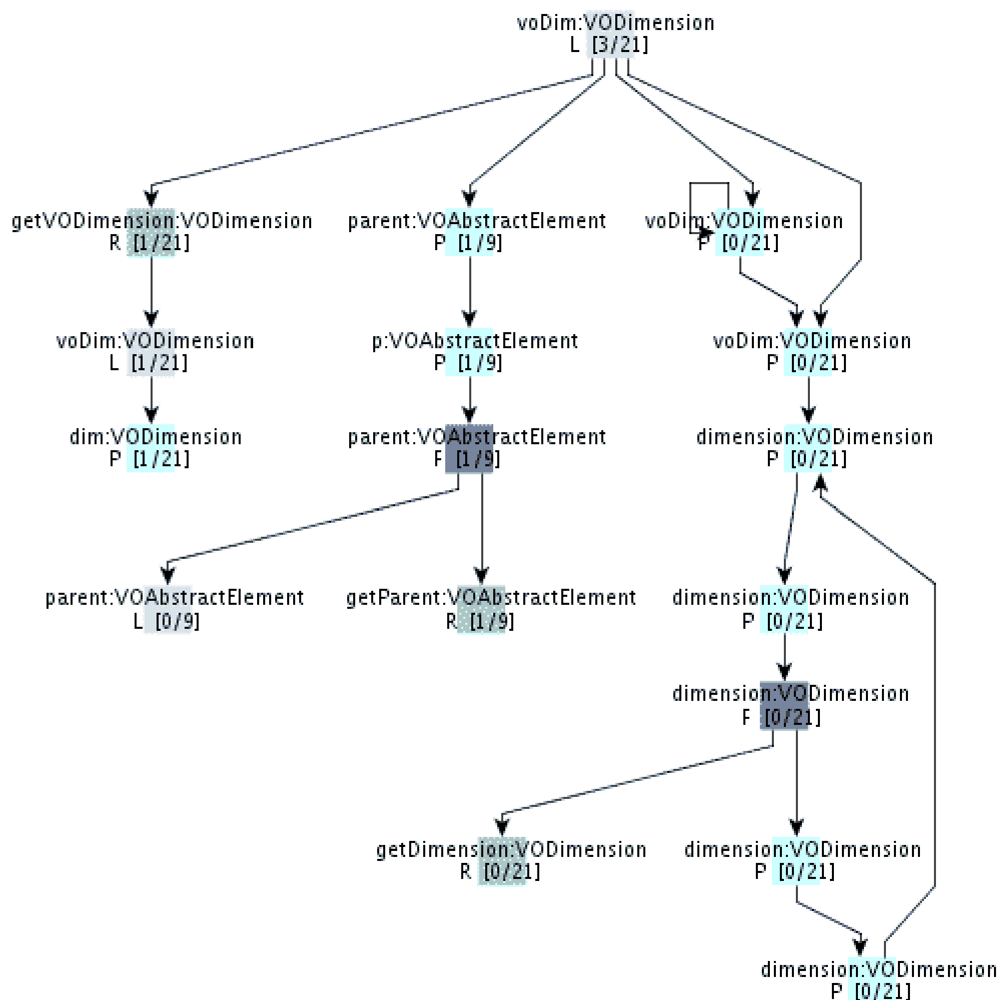
welche semantisch in die zwei Gruppen eingeteilt werden können, die entweder `parent` oder `auspraegung` bedeuten.

parent	auspraegung
voParent	voAp
parent	wrap2VOAuspraegung
p	auspraegung
getParent	

Diese Vereinheitlichung der Bezeichner in zwei Gruppen ergibt sich aus keinerlei bisher genannter Systematik, sondern allein aus der Kenntnis der entsprechenden Programmtextstelle. Lediglich das Deklarationselement `p` mit jeweils `parent` sowohl als Nachfolger wie als Vorgänger bei durchgehend gleichem inferierten Typ kann auf die dargestellte schematische Weise aufgefunden und bedenkenlos nach `parent` umbenannt werden. Versteht man die Bezeichner dieser Gruppen jeweils synonym, so fällt der Bezeichnerwechsel von `voAp` nach `voParent` von der dritten zur vierten Ebene des Zuweisungsgraphen auf, da der Wechsel der Typen `VOAuspraegung` nach `VOAbstractElement` erst in der nächsten Ebene stattfindet. Das heißt: Hier findet der Bezeichnerwechsel – damit ist ein Wechsel von einer Synonymgruppe zur anderen, also von `auspraegung`

nach parent, gemeint – vor dem Typwechsel statt. Bei der Überprüfung des AccessSets des Deklarationselementes voParent fällt auf, dass die zusätzliche Methode gegenüber dem folgenden Deklarationselement eine Methode aus dem Protokoll des Typen VO-Auspraegung ist. Das Deklarationselement voParent kann also nicht bereits mit dem allgemeineren Typen deklariert werden, und das bedeutet, dass es mit einem der Bezeichner für auspraegung benannt werden sollte.

Diese Namensänderung würde das Verständnis der betreffenden Codestelle durchaus erleichtern, zumal parent als Bezeichnung für Deklarationselemente mit dem Typen VOAbstractElement zur projektüblichen Nomenklatur gehört. Dies bestätigt der letzte Zuweisungsgraph für das Beispiel VODimension/VOAbstractElement, der zugleich als ein letztes Beispielmuster für das Auffinden unnötiger Bezeichnerwechsel dargestellt werden soll.



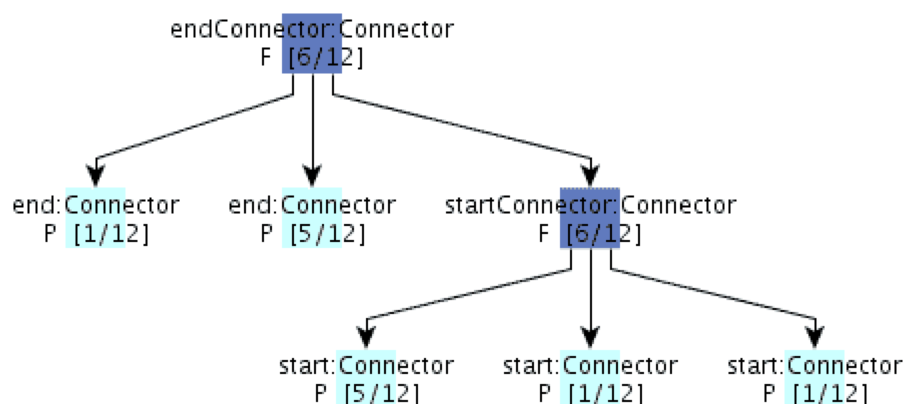
Ausgehend von dem Wurzelelement voDim enthält der Graph ebenfalls sieben unterschiedliche Bezeichner sowie die Typen VODimension und VOAbstractElement. Be-

trachtet man die beiden rechten Zuweisungen ausgehend vom Wurzelelement als rechten Teilgraphen und die beiden anderen Zuweisungen jeweils als mittleren und linken Teilgraphen, so fällt auf, dass jeder Teilgraphen durchgehend denselben inferierten Typen hat.[67] Alle diese Teilgraphen könnten demnach hinsichtlich ihrer Bezeichner vereinheitlicht werden. Während der mittlere durchgehend `parent` bzw. `getParent` enthielte, würden die beiden äußeren Teilgraphen durchgehend `dimension` bzw `getDimension` als Bezeichner enthalten, da `dimension` von allen Namensvarianten für Deklarationselemente vom Typ `VODimension` die häufigste ist.

5.3 Gegenmuster

Im Folgenden sollen noch drei Beispiele von Zuweisungsgraphen gezeigt werden, die zwar typische Szenarien darstellen, in denen aber die in Kapitel 3 „Problemstellung" dargelegte Schematisierung der Bedeutung von Bezeichnerwechseln eher zu Fehlannahmen führt.

Will man zwei Figuren innerhalb von JHotDraw miteinander verbinden, so muss ermittelt werden, welches die Verbindungspunkte einer Verbindung zwischen den beiden Figuren ist. Dazu dienen in JHotDraw Exemplare vom Typ `Connector`. In der Klasse `ConnectionTool` ist eine Feldvariable `endConnector` vom Typ `Connector` deklariert, für die sich der folgende Zuweisungsgraph genieren lässt.



Die Zuweisung von `endConnector` auf `startConnector` geschieht in der Methode `mouseReleased()`, durch die folgende Codezeile

[67] Ausnahme ist hier nur das letzte Deklarationselement namens `parent` im mittleren Teilgraphen. Diese Abweichung ist für die Argumentation jedoch unerheblich

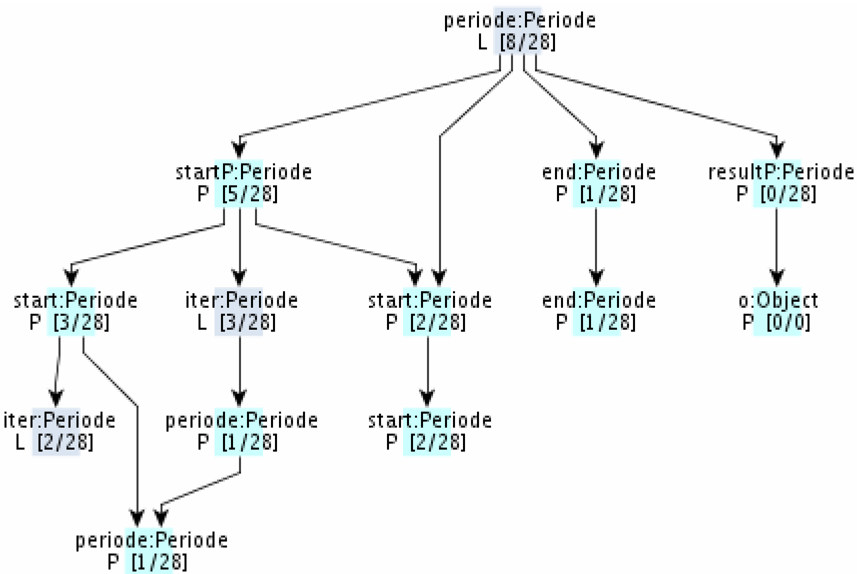
```
startConnector = endConnector = null;
```

und hat eher die Aufgabe, beiden Deklarationselementen `null` zuzuweisen. Nach den Ausführungen in Kapitel 3.4 „Bezeichnerwechsel ohne erkennbaren Grund“ würde insbesondere die Zuweisung zwischen `endConnector` und `startConnector` bei gleich bleibendem AccessSet als refaktorisierungsbedürftig erscheinen. Da zwei Feldvariablen innerhalb einer Klasse aber unterschiedlich heißen müssen, kann man solche Szenarien, in denen die Eindeutigkeit innerhalb des Namensraumes unterschiedliche Bezeichner erzwingt, aus der Betrachtung herausnehmen. Im vorliegenden Fall würden sie jedenfalls zu Fehlannahmen führen. Der Bezeichnerwechsel in diesem Beispiel ist also keineswegs refaktorisierungsbedürftig.

Das nächste Beispiel ist ähnlich gelagert und handelt ebenfalls von notwendig verschiedenen Bezeichnern innerhalb eines Namensraumes – in diesem Fall von Paramterbezeichnern innerhalb derselben Methodensignatur. Dabei enthält die folgende Methodensignatur

```
private SimpleResult computePerioden(
              …,
              Periode resultP,
              Periode startP,
              Periode end) throws … {
```

mehrere Parameter vom Typ `Periode`. Das Programm enthält einen Ausführungspfad, bei dem dasselbe Deklarationselement vom Typ `Periode` auf alle drei Parameter vom Typ `Periode` zugewiesen wird – tatsächlich können hier inhaltlich Start-, End- und Ergebniszeitpunkt für eine Anfrage identisch sein.
Der Zuweisungsgraph enthält zwangsläufig unterschiedliche Bezeichner bei gleichen Typen innerhalb derselben Methodensignaturen. Nachdem weitere Methoden mit mehreren Parametern vom Typ `Periode` aufgerufen wurden, sieht der Graph hier wie folgt aus:

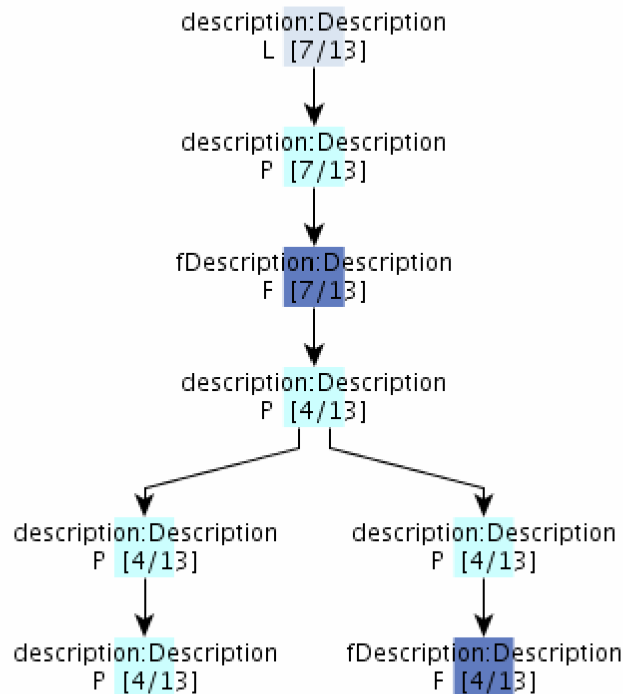


Gegebenenfalls könnte man sich aber dennoch fragen, ob bei stark unterschiedlichem bis disjunktem AccessSet und unterschiedlichem Bezeichner die Einführung der inferierten Typen nicht die Typhierarchie von Periode sinnvoll erweitern könnte.

Nicht immer setzt sich ein Bezeichner allein aus einer Art Inhaltsangabe des enthaltenen Objekts zusammen. Der durchaus verbreitete Einsatz von Präfixen wie f für Feldvariablen ist semantisch kein Bestandteil des Namens, sondern rührt aus einem rein programmiertechnischen Aspekt. So zeigt ein Beispiel aus dem Projekt JUnit[68] aus der Klasse `ErrorReportingRequest` für das Deklarationselement `description` vom Typ `Description` den folgenden Zuweisungsgraph:[69]

[68] http://www.junit.org/

[69] Der Graph wurde aus Gründen der Übersichtlichkeit gekürzt, indem Methodenrückgaben aus der Zuweisungsverfolgung ausgenommen wurden. Diese Möglichkeit ist in Kapitel 7 „Ausblick" als Feature eines künftigen Refactoringwerkzeuges mit aufgenommen.



In diesem Fall liegen streng genommen keine Bezeichnerwechsel vor. Es ist daher nicht sinnvoll, diesen Graphen im Hinblick auf Refaktorisierungsmöglichkeiten zu überprüfen, auch wenn hier Bezeichner von `description` nach `fDescription` und zurück wechseln, ohne dass sich der inferierte Typ ändert.

An dieser Stelle sei kurz auf ein Problem mit der unter der Bezeichnung „ungarische Notation" geläufige Codierung des Datentyps im Namen einer Variablen eingegangen. Diese Notationsweise verhindert natürlich generell, dass ein Bezeichnerwechsel vor oder hinter einem Typwechsel liegt, insofern konventionsgemäß mit der Typänderung auch eine Präfixänderung einherginge. Diese Variante[70] der Ungarischen Notation muss für die vorliegende Untersuchung, falls erkennbar, ignoriert, das heißt, aus den Namensbestandteilen herausgefiltert werden. Denn die schlichte Übernahme des Datentyps in den Namen berührt die Bedeutung des Deklarationselementes nicht, verschlechtert aber die Erkennbarkeit einer durchgehenden Bedeutung innerhalb des Zuweisungsgraphen.

---

[70] Die Ungarische Notation existiert in mehreren Varianten, von denen die hier angesprochene sicher die am meisten verbreitete ist. Andere Varianten, insbesondere wohl die ursprünglich von Charles Simonyi entwickelte, codieren in der Variablen eher die Funktion bzw. den Verwendungszweck der Variablen. Dieses Verständnis steht dem der vorliegenden Arbeit näher, insofern die verschiedenen Bedeutungen in den Namen der Variablen aufgenommen werden. Vgl.: [Simonyi 1999]

6 Diskussion

Diese Arbeit ist im Umfeld des intoJ-Projektes[71] entstanden, das die Technik der Typinferenz für Refaktorisierungswerkzeuge fruchtbar zu machen versucht. In diesem Zusammenhang sind einige Arbeiten und Werkzeuge entstanden, wie [Kegel / Steimann 2007], [Steimann 2007], [Steimann / Mayer 2006], [Steimann / Mayer 2007] und [Steimann 2007], um nur einige zu nennen. In all diesen Arbeiten wird die Möglichkeit ausgelotet, durch Einführung verallgemeinerter Typen in Form von Interfaces die Kopplung der Typen innerhalb bestehender Programme zu verringern. Die Ermittlung dieser Typen basiert dabei immer auf der in dieser Arbeit ebenfalls vorgestellten Protokollberechnung aus Infertype[72]. Während das Protokoll eines inferierten Typen algorithmisch ermittelt werden kann, hängt die Beurteilung, ob die Einführung eines Typen an einer gegebenen Stelle eine sinnvolle Entkopplung oder eine unnötige Steigerung der Typkomplexität darstellt, von einer Fülle von Kontextinformationen ab, die letztlich jede Chance auf Automatisierbarkeit dieser Entscheidung vereiteln. Aus diesem Grund handeln die meisten Arbeiten, die Infertype als Refaktorisierungsalgorithmus innerhalb verschiedenster Werkzeuge einsetzen, auch von den Möglichkeiten, Teile dieser Kontextinformation auf geeignete Weise zu visualisieren – [Bach 2007], [Fiedler 2007] und [Steimann / Mayer 2007] sind Beispiele dieser Art. Mit kontextueller Distanz wird bei Steimann[73] zudem ein Maß eingeführt, mit dem programmweit das Verhältnis zwischen tatsächlich verwendeten und deklarierten Protokollen gemessen werden kann. Nach empirischer Anwendung solcher Metriken auf größere Programmkorpora lässt sich beurteilen, ob dieses Verhältnis in einem gegebenen Programm über- oder unterdurchschnittlich ist – auch das sind Informationen, die im Softwareentwicklungsprozess zusätzlich als Kontextinformation zur Verfügung stehen, aber dem Entwickler keine Entscheidung abnehmen können.

Handeln alle diese Arbeiten von den Refaktorisierungsmöglichkeiten innerhalb einer gegebenen Typhierarchie, so fokussiert die vorliegende Arbeit eher die Werte, die Typen annehmen können, insofern untersucht wird, was die Abfolge von Bezeichnern während des Lebenszyklus' eines Objekt über die Verwendung des Typen aussagen könnte. Dabei wurde unterstellt, dass die Instanz eines Typen oft nur Teilaspekte des deklarierten Typen

---

[71] http://www.intoj.org/index.php/Main_Page

[72] Je nach Entstehungsjahr der Arbeit ist diese Ermittlung mit einer früheren Version von Infertype 3 durchgeführt worden.

[73] Siehe [Steimann 2007].

verwendet und dass sich diese Spezialisierung bereits in der Bezeichnung der Referenz zum Ausdruck bringt – wie etwa bei einem Bezeichner `verantwortlicher` vom Typ `Benutzer`. Unter den genannten Arbeiten ist in dieser Hinsicht [Steimann / Mayer 2007] hervorzuheben, aus der das Werkzeug „Type Access Analyzer" hervorging,[74] mit einer besonders breiten Darstellung der Kontextinformationen aller möglichen und bereits vorhanden Typen, insofern in dieser Arbeit das Thema der vorliegenden als Ausblick bereits angeschnitten ist. Das Werkzeug, heißt es dort über zukünftige Arbeiten,[75] könnte durch eine Darstellung von Detailinformationen angereichert werden, in der der Zuweisungsgraph eines Deklarationselementes bestimmt wird. „*Such could be used to harmonize the naming of variables in an assignment chain or, in case of a variable name change beyond a certain point, serve as indication for the introduction of a new interface (reflecting the new role of the objects passed to that variable).*"[76]

Diese Idee zusammen mit den Ausführungen von Steimann über die Interpretation von Interfaces als Rollen in [Steimann 2000] und [Steimann 2001] sowie die unterschiedlichen Arten des Gebrauchs von Interfaces in [Steimann / Mayer 2005] bilden den Hintergrund für die Fragestellungen der vorliegenden Arbeit, in der Abhängigkeiten zwischen Typen und Bezeichnern von Referenzen untersucht wurden. Ausgegangen wurde dabei von der oben zitierten Grundidee, dass die Bezeichner von Deklarationselementen Angaben darüber machen, wie der Typ verwendet wird.

Die dargelegten Beispiele haben die Möglichkeiten und Grenzen dieses Ansatzes zeigen können. Die Interpretation der Zuweisungsgraphen anhand der in Kapitel 3 „Problemstellung" eingeführten Ideen führt, wie in Kapitel 5 „Beispiele" demonstriert, tatsächlich zur Verdeutlichung von Refaktorisierungsmöglichkeiten innerhalb von Programmtexten. An dieser Stelle ist aber auch deutlich geworden, dass die Fundstellen in der Regel nicht ohne über den Zuweisungsgraph hinausgehende Kontextkenntnisse interpretiert werden können. So wurden beispielsweise die Kenntnis der gleichen Bedeutung von verschiedenen Bezeichnern ("`voParent`" = "`parent`" = "`p`" etc.) sowie letztlich domänenspezifische Informationen (`betreiberform` ist eine Entität ohne vorliegende Typentsprechnung) verwendet.

Eine weitere Schwäche des vorliegenden Ansatzes besteht darin, dass die in Kapitel 3 „Problemstellung" eingeführten Zusammenhänge zwischen Bezeichnerwechseln von De-

[74] http://www.intoj.org/index.php/TAA

[75] Siehe [Steimann / Mayer 2007], Kapitel 6 „Future Work".

[76] Ebenda

klarationselementen innerhalb von Zuweisungsgraphen einerseits und deren deklarierten wie inferierten Typen andererseits naturgemäß nicht alle möglichen Szenarien abdecken, die programmierbar sind. Ob eines der beschriebenen Muster wie Generalisierung, Rolle oder andere auf eine Programmstelle zutrifft, ob daher die dazugehörigen Erwartungen an die Struktur tatsächlich angebracht sind, bleibt eine Entscheidung des Betrachters, also wiederum gebunden an die zur Verfügung stehenden Kontextinformation während des Softwareentwicklungsprozesses.

Auf Grund der genannten Einschränkungen lässt sich daher resümieren, dass die Darstellung von Zuweisungsgraphen eine sinnvolle Bereicherung der Kontextinformation im Entwicklungsprozess ist, da sie sich gegebenenfalls auf die beschriebene Weise interpretieren lassen und somit Refaktorisierungsmöglichkeiten verdeutlichen können. Im folgenden Kapitel wird daher auch vor allem für die Fortführung der hiesigen Untersuchung in Form eines Refaktorisierungswerkzeuges plädiert, das Zuweisungsgraphen als zusätzliche Sichtweise auf Programmstrukturen einführt.

7 Ausblick

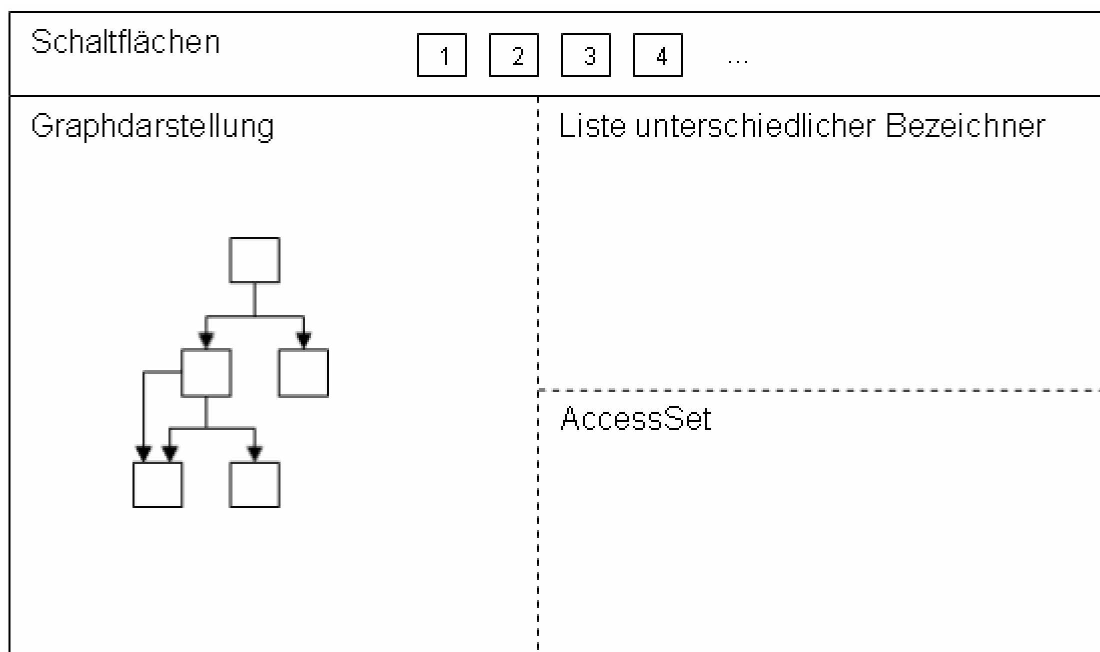
Die vorliegende Arbeit bietet mehrere Möglichkeiten, weiterentwickelt zu werden. Im Folgenden soll auf zwei Möglichkeiten eingegangen werden. Im ersten Fall handelt es sich um eine Komfortverbesserung innerhalb der aktuellen Implementierung von Infertype. In einem weiteren Abschnitt soll beschrieben werden, wie die Ergebnisse dieser Arbeit in Form eines neuen Refaktorisierungswerkzeuges verwendet werden könnten. Am Schluss des Kapitels wird auf Probleme eingegangen, die bei der Implementierung eines solchen Werkzeugs mit der gegenwärtigen Version von Infertype auftreten würden.

Eine einfache und zugleich nahe liegende Idee, die sich aus dieser Arbeit ergibt, besteht in einer Hilfe bei der Wahl eines Typbezeichners für den neuen inferierten Typen bei der Benutzung des InferType-Refactorings. Diese Hilfe bestünde aus einer Vorschlagsliste aller unterschiedlicher Bezeichner, die der Zuweisungsgraph des selektierten Deklarationselementes enthält und die zu keiner Namensraumverletzung führen. An dieser Stelle sollte nicht nur die Einführbarkeit der Namen als Typbezeichner überprüft werden, sondern einzelne Namen auch angepasst werden. Die vorgeschlagenen Bezeichner sollten konventionsgemäß mit einem Großbuchstaben beginnen und Methodennamen sollten, insofern erkennbar, mindestens um die einleitenden Tätigkeitsbegriffe wie „get", „set", „extract", „do", „mach", „make" etc. gekürzt werden. Möglicherweise sollte man die Namen von Methoden gänzlich aus der Vorschlagsliste heraushalten. Diese Komforterweiterung von Infertype könnte auf dem ohnehin von Infertype selbst bereits erzeugten Constraintgraphen aufsetzen und unter Anwendung des hier vorgestellten Algorithmus den Zuweisungsgraph ermitteln. Die zu erwartenden Performanzverluste sind – da der Constraintgraph schon existiert – nicht allzu hoch.

Wie in Kapitel 6 „Diskussion" bereits ausgeführt, erlaubt die Anwendung der hier vorgestellten Interpretationsmuster für den Wechsel von Bezeichnern, Typen und inferierten Typen innerhalb von Zuweisungsgraphen allein noch keine definitive Aussage über Art und Notwendigkeit der Refaktorisierung einer bestimmten Codestelle. Dazu sind meist Rückgriffe auf Kontextinformationen notwendig, die sich nicht aus den Zuweisungsgraphen ergeben. In der Regel lässt sich also mit Hilfe der Zuweisungsgraphen ein Refaktorisierungsbedarf verdeutlichen, aber nicht restlos begründen. Eine sinnvolle Fortsetzung der vorliegenden Arbeit wäre daher die Entwicklung eines Werkzeugs, das mit der Darstellung von Zuweisungsgraphen einen alternativen Codeeditor einführt. So könnten die hier erarbeiteten Analysemethoden toolgestützt auf jede beliebige Codestelle angewen-

det, möglicher Refaktorisierungesbedarf dabei verdeutlicht und im Bedarfsfall direkt über die Graphdarstellung korrigiert werden. Die mögliche Funktionalität eines solchen Werkzeugs soll nun ausgehend von einer erläuterten UI-Skizze konkretisiert werden. Wie in Kapitel 4 "Materialien und Methoden" bereits erwähnt, kann das im Rahmen dieser Arbeit entwickelte Infername-Plug-In einen ersten Eindruck über die Machbarkeit der Ermittlung und Darstellung von Zuweisungsgraphen vermitteln.

Es wird davon ausgegangen, dass sich die Benutzeroberfläche des geplanten Plug-Ins in einem eigenen Eclipsefenster – in der eclipseeigenen Begrifflichkeit View genannt – darstellt, um so sichtbar mit dem Sourcecodeeditor verlinkt werden zu können. Die folgende UI-Skizze stellt das Plug-In-Fenster dar, wobei jedes beschriftete Rechteck einen eigenen Bereich darstellt. Die gestrichelten Linien deuten an, dass die Bereiche an dieser Stelle innerhalb der zur Verfügung stehenden Gesamtfläche des Fensters größenveränderbar sein sollten.



Graphdarstellung: Hier wird der Zuweisungsgraph, so wie in Kapitel 2.2.6 „Darstellung der Zuweisungsgraphen" beschrieben, dargestellt.[77] Im Unterschied zur aktuellen Infername-Implementierung handelt es sich hierbei nicht um Baumdarstellungen, weil die Zuweisungsgraphen aus den genannten Gründen keine Bäume sind,[78] sondern um die Darstellung gerichteter Graphen, die auf Benutzereingaben wie Mausklicks reagieren.

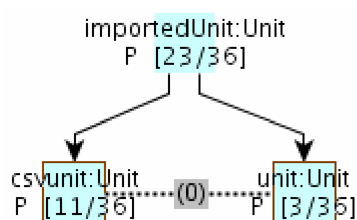
---

[77] Diese könnte unter Rückgriff auf die für Eclipse zur Verfügung stehenden Frameworks GEF (http://www.eclipse.org/gef) und GMF (http://www.eclipse.org/gmf) realisiert werden.

[78] Siehe Kapitel 4.3.3 „Benutzung".

Da Zuweisungsgraphen unter Umständen sehr groß werden können, sind einige intuitiv bedienbare Möglichkeiten vorzusehen, mit denen der Benutzer versuchen kann, die Graphdarstellung geeignet zu komprimieren. Denkbar wäre die Möglichkeit, temporär die Weiterverfolgung von Methodenrückgaben auszuschalten. Es könnte interessant sein, die Namensgebung von Methoden auszublenden, insofern sie in der Praxis ohnehin etwas anderen Regeln folgt als die Namensvergabe für Variablen, Felder oder Parameter. Damit der Benutzer den Graphen schnell auf den ihn interessierenden Teil verkleinern kann, sollte ein Kontextmenü über jedem dargestellten Deklarationselement ermöglichen, den Graphen bis zu oder ab diesem Deklarationselement auszublenden. Ein weitere Möglichkeit wäre, dass der Doppelklick auf einer Kante die Darstellung des zugewiesenen Deklarationselementes in den Mittelpunkt rückt.

Mit dem in Kapitel 3.3 „Inferierter Typ ohne Typwechsel“ eingeführten Vergleich zweier AccessSets von Deklarationselementen mit dem gleichen Vaterelement konnte unter anderem im Falle der Deklarationselemente `csvunit` und `unit` der Verdacht erhärtet werden, dass unterschiedliche Bezeichner mit disjunktem AccessSet darauf verweisen, dass der deklarierte Typ in zwei Typen aufgeteilt werden kann. Da diese Mengenbeziehung von AccessSets untereinander aus der Graphendarstellung nicht hervorgeht, sollte eine Möglichkeit vorgesehen werden, einen solchen Vergleich anzuzeigen. Denkbar wäre zum Beispiel, dass der Benutzer innerhalb des Graphen gleichzeitig zwei unterschiedliche Deklarationselemente selektieren kann, um sich die Anzahl der in beiden AccessSets gemeinsam vorkommenden Methoden darstellen zu lassen. Eine Darstellung wie die folgende



würde dann darauf hinweisen, dass die Schnittmenge der AccessSets von `csvunit` und `unit` die Größe 0 hat.

Das Kontextmenü eines Deklarationselementes sollte zudem die folgenden Refaktorisierungsmöglichkeiten anbieten:

- „Rename“: Ändert den Bezeichner des Deklarationselementes. Das Refactoring ist Bestandteil des JDT und behandelt bereits Namensraumkonflikte.

- „Generalize Declared Type“: Auch dieses Refactoring ist Bestandteil des JDT. Es kann genutzt werden, um, wenn möglich, einen allgemeineren Typen zur Deklaration des Deklarationselementes zu verwenden. Anders als bei Infertype muss dieser allgemeinere Typ aber bereits vorliegen.
- „Infertype“: Refactoring von [Kegel 2007], mit dem ein maximal verallgemeinerter Typ berechnet und eingeführt werden kann.

Liste unterschiedlicher Bezeichner: Hier wird eine Liste aller im aktuell dargestellten Zuweisungsgraphen vorkommenden Bezeichner angegeben, wobei auch mehrfach vorkommende Bezeichner unter Angabe ihrer Häufigkeit nur einmal angezeigt werden. Auch ausgehend von dieser Liste könnte das Rename-Refactoring aufgerufen werden, das dann bei häufiger vorkommenden Bezeichnern wiederholt ausgeführt werden muss. Optional könnten zu den Bezeichnern die Typisierungen mit angegeben werden. Aus der Ergebnismenge würden dann gleiche Paare, bestehend aus Bezeichner und Typangabe, durch eine Häufigkeitsangabe ersetzt, wenn sie wiederholt auftreten.

AccessSet: Dieser Ausschnitt stellt die Methoden des AccessSets eines aktuell selektierten Deklarationselementes in Form einer Liste dar. Diese Darstellung hat rein informativen Charakter und sollte sich in der Darstellung an der gewohnten Outlineview des JDT orientieren. Ein Doppelklick auf der Darstellung einer Methode sollte innerhalb des Sourcecodeeditors an die Programmtextstelle verzweigen, in der die Methode definiert ist.

Schaltflächen: Hier sollten die Schaltflächen angebracht sein, die Aktionen beinhalten, die die gesamte Graphdarstellung betreffen. Dazu gehört das oben erwähnte Ausschalten der Verfolgung von Deklarationselementen, die Methodenrückgaben darstellen oder die Möglichkeit, verzweigungsfreie Teilstrecken innerhalb des Graphen zusammenzufassen, wenn die enthaltenen Deklarationselemente gleich typisiert und benannt sind.

In zwei weiteren Schaltflächen sollten auf geeignete Weise einerseits Typwechsel und andererseits Bezeichnerwechsel hervorgehoben werden. Dies könnte durch unterschiedliche Kantendarstellung geschehen, so dass beide Ereignisse an einer Kante dargestellt werden können – etwa durch unterschiedliche Pfeilspitzen. An dieser Stelle sind weitere Schaltflächen denkbar. Zum Beispiel könnten Hervorhebungen von Typ- und Bezeichnerwechsel wieder entfernt werden, wo sie übereinander liegen, denn wie in dieser Arbeit ausgeführt, handelt es sich in solchen Fällen meist um begründbare Bezeichnerwechsel.

Zuletzt sei noch auf eine Reihe von Problemen eingegangen, die aus der zweckfremden Verwendung von Infertype resultieren, nämlich Constraintgraphen als Grundlage für Zuweisungsgraphen zu benutzen. Die aus dem Infertype-Plug-In übernommenen Einschränkungen bezüglich der Verfolgung von Zuweisungen[79] sind für die vorliegende Arbeit noch hinnehmbar, für ein zukünftiges Refaktorisierungstool wie oben skizziert aber kaum zu begründen. Dazu gehört zum Beispiel, dass Infertype die Erzeugung des Constraintgraphen grundsätzlich unterlässt, sobald Typen aus externen Bibliotheken involviert sind. Es ist aus Sicht von Infertype nachvollziehbar, dass ein Deklarationselement wie `String bezeichner` nicht betrachtet wird, weil der Typ String nicht modifiziert werden kann. Aus Sicht des skizzierten Werkzeuges ist das kein Grund, nicht den Zuweisungsgraphen zu ermitteln, da die Bezeichner sehr wohl änderbar sind und gegebenenfalls entlang der Zuweisungen angepasst werden sollten. Ebenso unveränderbar sind Zuweisungen an Parameter, deren Deklaration in Typen einer externen Bibliothek vorliegen. Auch Feldzugriffe von Deklarationselementen verhindern derzeit die Erzeugung des Zuweisungsgraphen, da Infertype einen solchen Feldzugriff nicht innerhalb eines neu einzuführenden Interface deklarieren kann. Kurz: Jedes Mal, wenn Infertype mit Blick auf die Veränderbarkeit der Typhierarchie die Erzeugung des Constraintgraphen mit gutem Grund unterlässt, ist das noch lange kein hinreichender Grund, auch die Erzeugung von Zuweisungsgraphen zu unterbinden, da die Änderbarkeit der Bezeichner nicht an dieselben Bedingungen geknüpft ist. Ein ähnliches Problem gilt für eine weitere aus Infertype übernommene Einschränkung, nämlich dass Zuweisungen nach Downcasts nicht weiterverfolgt werden.[80] Ein Werkzeug, das die unterschiedlichen Aufgaben und Zustände eines gegebenen Objektes anzeigt, so wie sie sich durch seine unterschiedlichen Bezeichner entlang seines Lebenszyklus' darstellen, sollte ein Objekt nach einem Downcast nicht aus dem Blick verlieren.

Infertype erweist sich letztlich also als Grundlage für die Ermittlung von Zuweisungsgraphen nur bedingt geeignet, insofern schon die Erzeugung des Constraintmodels an Infertype-spezifische Bedingungen gebunden bleibt. Im Falle von Downcasts stellt sich zusätzlich die Frage, ob ein auf Basis von Infertype erzeugter Zuweisungsgraph nicht gar zu klein ist. Voraussetzung für ein künftiges Refaktorisierungswerkzeug der hier beschriebenen Art wäre also eine Weiterentwicklung von Infertype, die die Erzeugung des Constraintsystems vollkommen von dem Zweck entkoppelt, maximal verallgemeinerte

---

[79] Vgl. hierzu Kapitel 4.2.2 „Übernommene Einschränkungen".

[80] Vgl. ebenda.

Typen zu ermitteln. Insofern dann die Reichhaltigkeit der erzeugbaren Zuweisungsgraphen steigt, da beispielsweise auch die Verwendung von Standardtypen wie `String` oder `int` dargestellt werden könnte, werden vielleicht auch neue, über diese Arbeit hinausgehende Zusammenhänge zwischen der Verwendung von Bezeichnern und Typen erkennbar. Die Idee eines Programmierwerkzeuges, das Zuweisungsgraphen als Darstellungsalternative für Codestrukturen anbietet, ist jedenfalls prinzipiell offen, auch sukzessive weitere Abhängigkeiten zwischen Bezeichner, Typ und inferiertem Typ darzustellen.

8 Schlussbetrachtung

Ziel der vorliegenden Arbeit war es, mit Hilfe von Zuweisungsgraphen ein Nutzungsprofil der eingesetzten Typen zu erstellen und dieses hinsichtlich einer standardisierteren Namensgebung oder adäquateren Typisierung zu prüfen. Dazu mussten Zuweisungsgraphen zunächst ermittelt und sichtbar gemacht werden. Anschließend wurden begründbare Mustererwartungen entwickelt, unter deren Maßgabe reale Zuweisungsgraphen interpretiert werden konnten. Im Ergebnis hat sich gezeigt, dass solche Muster gefunden werden können und dass sich in Programmtexten, die davon abweichen, tatsächlich Refaktorisierungsmöglichkeiten anbieten. Es hat sich aber auch gezeigt, dass die letztendliche Beurteilung über den Refaktorisierungsbedarf nicht allein auf die hier vorgestellte Technik zurückgeführt werden kann, sondern nicht ohne Rückgriff auf Kontextinformationen auskommt. Aus diesem Grund wurde am Ende auch hauptsächlich für die Fortsetzung dieser Arbeit durch Erstellung eines Programmierwerkzeuges plädiert, das die hier gewonnenen Kenntnisse als zusätzliche Kontextinformation so aufbereitet, dass sie den Refaktorisierungsprozess unterstützen. Bei der Umsetzung dieses Werkzeuges kann das für die vorliegende Arbeit entwickelte Plug-In als Machbarkeitsstudie verstanden werden.

Typinferenz ist ursprünglich eine Technik, mit deren Hilfe Compiler oder Laufzeitumgebungen in Sprachen, die eine statische Typisierung nicht vorschreiben oder nicht gestatten, dennoch die typkorrekte Verwendung von Variablen prüfen können.[81] Mit Infertype wurde diese Technik in einen ganz anderen Zusammenhang, nämlich den der Refaktorisierung, gestellt. Die vorliegende Arbeit hat diesen Fokus noch einmal verschoben, insofern sie nun die inferierten Typen zur Beurteilung der Namensgebung von Deklarationselementen heranzieht. Dabei ist die Idee durchaus nahe liegend. Denn während Typen in der Regel im Hinblick auf ihre programmweite Gültigkeit entworfen werden, entstehen Bezeichner im Zusammenhang einer vergleichsweise lokalen Verwendung eines Typen. Genau in diesem Sinne sind der inferierte Typ und der Bezeichner artverwandt, insofern beide Aspekte der lokalen Verwendung eines global angelegten Typen darstellen. So konnte sich zeigen, dass die Wechsel von Bezeichner und inferiertem Typ stark korrespondieren, bzw. dass ein Bezeichnerwechsel ohne einen zugehörigen Wechsel des inferierten Typen in der Regel unbegründet ist – also auf eine zu vermeidende Bezeichnervielfalt hinweist.

---

[81] Einen Überblick liefern [Bauer / Höllerer], S.127ff.

9 Anhang

9.1 Inhalt der beiliegenden CD

Um die Funktionsweise des für diese Arbeit entwickelten Infername-Plug-In nachvollziehen zu können, befindet sich das Plug-In als jar-Datei auf einer beiliegenden CD. Außerdem enthält die CD die verwendete Version des Infertype-Plug-In sowie die verwendete Version 3.3 von Eclipse. Die CD enthält außerdem den Sourcecode des Plug-Ins, um gegebenenfalls für die Entwicklung künftiger Plug-Ins ganz oder teilweise zur Verfügung zu stehen. Im Folgenden eine Übersicht über die Inhalte der CD:

Verzeichnis	Inhalt
dist	Infername- sowie das verwendete Infertype Plug-In als jar-Datei.
source	Der Sourcecode des Infername-Plug-In
samples	Eine Beispielklasse, die einige unterschiedliche Anwendungsmöglichkeiten von Infername demonstriert
eclipse	Eine Installation von Eclipse 3.3 mit „Infer Type“ und Infername für Windows.

9.2 Installationsanleitung

Die im Verzeichnis eclipse abgelegte Version von Eclipse 3.3 enthält bereits die benötigten Plug-Ins, so dass ein Kopieren des Verzeichnisses in ein beliebiges Verzeichnis auf dem eigenen Rechner als Installation ausreicht. Das verwendete Eclipse ist eine Version für Windows und wurde auf Windows-XP mit der Javaversion 1.6.0.5 getestet.

Um Infername in einer anderen Eclipseinstallation zu verwenden, müssen die beiden unter dist abgelegten jar-Dateien in das plugin-Verzeichnis der entsprechenden Eclipse-Installation kopiert werden.

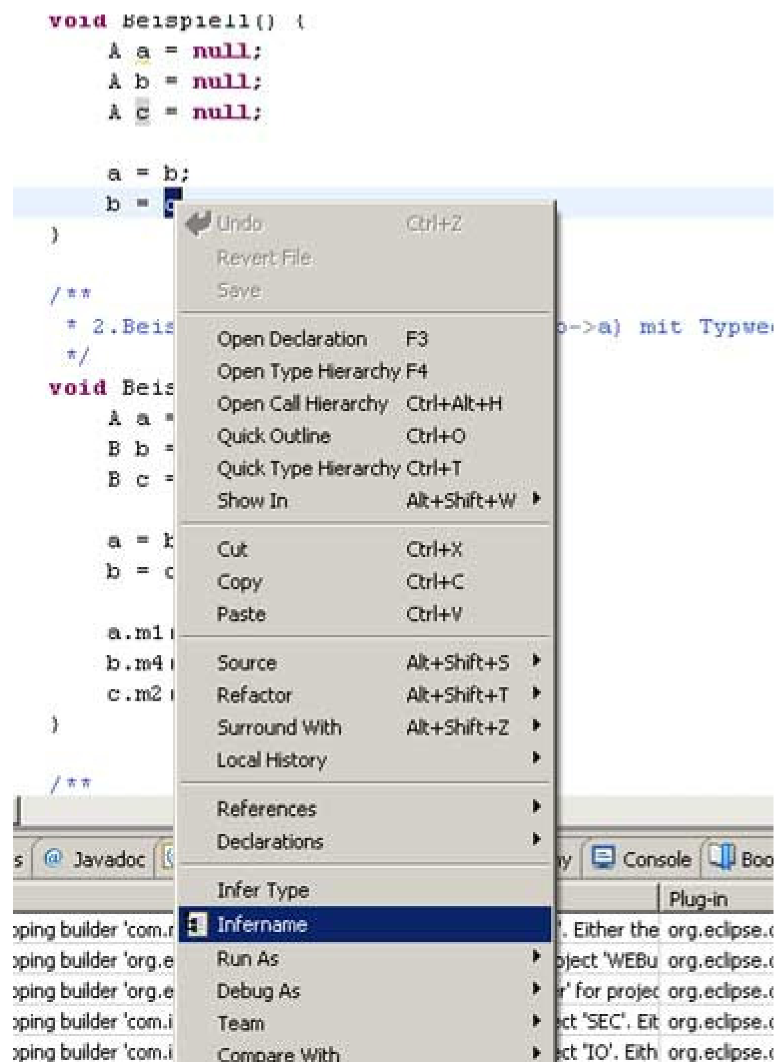
Zur Darstellung der Zuweisungsgraphen, wie sie in der Arbeit vorzufinden sind, wurde außerdem das frei verfügbare Programm yEd verwendet, das unter www.yworks.de erhältlich ist. In der vorliegenden Arbeit kommt die Version 2.4.2.2 von yEd zum Einsatz.

Das Kopieren von Dateien von CD auf eine Festplatte führt bei manchen Windowsversionen dazu, dass die Zieldateien schreibgeschützt abgelegt werden. In solchen Fällen muss der Schreibschutz entfernt werden, da Eclipse sonst gegebenenfalls keinen Workspace

anlegen kann und außerdem die Beispielklasse sowie der Sourcecode nicht bearbeitbar sind.

9.3 Bedienung

Als Beispiel kann die unter samples/samples abgelegte Klasse TestInfername.java benutzt werden. Diese sollte in einem Javaprojekt innerhalb von Eclipse im package /samples abgelegt werden. Diese Beispielklasse enthält die Beispiele 1 bis 5, die verwendet werden, indem in jedem Beispiel das Deklarationselement mit Namen c (oder c() im Beispiel 5) selektiert wird. Über die Verwendung der rechten Maustaste wird im Kontextmenü der Eintrag „Infername“ ausgewählt.



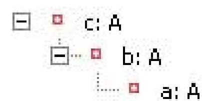
Nachdem das Modell erzeugt wurde, zeigt sich das Ergebnis in einem eigenen Infername-View. Dieser View kann zuvor über das Eclipsemenü windows/show view/other und dann InferName/InferNameExplorer ausgewählt werden, sollte sich aber automatisch öffnen,

wenn er noch nicht geöffnet ist. Der View zeigt die unter Kapitel 7 „Ausblick“ beschriebene Oberfläche und Funktionalität.

Für das erste Beispiel mit dem folgenden Beispielcode

```
17    void Beispiel1() {
18            A a = null;
19            A b = null;
20            A c = null;
21
22            a = b;
23            b = c;
24    }
```

ergibt sich die folgende TreeView-Darstellung:



Der Treeview zeigt die Zuweisung des Deklarationselementes c auf b und zuletzt auf a. Das entspricht den Zuweisungen der Zeilen 22 und 23. Neben den Namen der Deklarationselemente werden nach dem Doppelpunkt auch die Typen dargestellt, die in Zeile 18 bis 20 deklariert sind.

Auch die Treeview-Darstellung verfügt über ein Kontextmenü, das über die Bedienung der rechten Maustaste aufgerufen werden kann. Hier ist unter anderem der Eintrag „exportieren“ auswählbar, mit dem sich der Graph in eine gml-Datei exportieren lässt,

die mit dem erwähnten Programm namens yEd dargestellt werden kann.

In der Toolbar gibt es einige weitere Schaltflächen, mit denen Methodenrückgaben ausgeschlossen, alle gleichnamige DEs zusammengefasst werden können, sowie die Sortierung geändert oder der Graph expandiert oder geschlossen werden kann. Elemente der Treeview-Darstellung sowie der Fenster „Querverweise“ und „AccessSet“ reagieren auf

einen Doppelklick mit der Maus, indem sie die korrespondieren Code- bzw. Treeviewstelle anzeigen.

9.4 Sourcecode

Auf der CD befindet sich im Verzeichnis *source* eine Kopie des Eclipseprojektes, mit dem das Infername-Plug-In erstellt wurde. Der Code verdankt etliche Lösungsideen anderen Projekten aus dem intoJ-Umfeld[82], vor allem den Projekten „Infer Type“ und „Type Access Analyze“. Ersteres vor allem bezüglich des Umgangs mit dem Sourcecodeeditor des JDT sowie den AST-Nodes, letzteres für den Algorithmus zur Erzeugung von Zuweisungsgraphen. Wie bereits erwähnt, hatte das Plug-In allein die Aufgabe, für die vorliegende Arbeit die Zuweisungsgraphen zu ermitteln und darzustellen, an eine weiter reichende Veröffentlichung wurde nicht gedacht. Der Sourcecode wird hiermit mitgeliefert, um Folgeprojekten als Machbarkeitsstudie oder Ideengeber zu dienen.

[82] Siehe: http://www.intoj.org/index.php/Main_Page

10 Literaturverzeichnis

[Bach 2007]
Bach, Markus: Design und Implementierung eines Eclipse-Plugins zur Anzeige von möglichen Typgeneralisierungen im Quelltext, Masterarbeit Fernuniversität Hagen 2007.

[Bauer / Höllerer]
Bauer, Bernhard / Höllerer, Riitta: Übersetzung objektorientierter Programmiersprachen: Konzepte, abstrakte Maschinen und Praktikum 'Java-Compiler', München 1998.

[Clayberg / Rubel 2006]
Clayberg, Eric / Rubel, Dan: Eclipse: building commercial-quality plug-ins, München 2006.

[Diestel 2006]
Diestel, Reinhard: Graphentheorie, 3. Auflage, Heidelberg 2006.

[Fiedler 2007]
Fiedler, Frank: Ein Eclipse-Framework zur automatischen Bestimmung nützlicher Interfaces in Java-Programmen, Masterarbeit Fernuniversität Hagen 2007.

[Fowler u. a. 1999]
Fowler, Martin / Beck, Kent / Brant, John / Opdy, William: Refactoring. Improving the Design of Existing Code, München 1999.

[Fuhrer u. a. 2005]
Fuhrer, Robert / Tip, Frank / Kieżun, Adam / Dolby, Julian / Keller, Markus: Efficiently refactoring Java applications to use generic libraries, in: ECOOP 2005 – Object-Oriented Programming, 19th European Conference, Glasgow 2005, S. 71-96.

[Gamma u. a. 2001]
Gamma, Erich / Helm, Richard / Johnson, Ralph / Vlissides, John: Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software, München 2001.

[Gamma / Beck 2004]
Gamma, Erich / Beck, Kent: Eclipse Erweitern – Prinzipien, Patterns und Plug-Ins, München 2004.

[Kegel 2007]
Kegel, Hannes: Constraint-basierte Typinferenz für Java 5, Diplomarbeit Fernuniversität Hagen 2007.

[Kegel / Steimann 2007]
Kegel, Hannes / Steimann, Friedrich: ITcore: A type inference package for refactoring tools, in: Dig, Danny / Cebulla, Michael (Hrg.): 1st Workshop on Refactoring Tools (WRT'07). Proceedings, Berlin 2007, S.7-8.

[Kuhn / Thomann 2006]
Kuhn, Thomas / Thomann, Olivier: Abstract Syntax Tree, in: http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html, 2006.

[Nielson u. a. 2005]
Nielson, Flemming / Nielson, Hanne R. / Hankin, Chris: Principles of Program Analysis, Heidelberg 2005.

[Simonyi 1999]
Charles Simonyi, Hungarian Notation, in: http://msdn.microsoft.com/en-us/library/aa260976(VS.60).aspx, November 1999

[Steimann 2000]
Steimann, Friedrich: Formale Modellierung mit Rollen, Habilitationsschrift Universität Hannover, Hannover 2000.

[Steimann 2001]
Steimann, Friedrich: Role = Interface: a merger of concepts, in: Journal of Object-Oriented Programming 14:4 (2001), S. 23-32.

[Steimann 2007]
Steimann, Friedrich: The Infertype Refactoring and its Use for Interface-Based Programming, in: Journal of Object Technology 6:2, Special Issue OOPS Track at SAC 2006 (2007), S. 67-89.

[Steimann / Mayer 2005]
Steimann, Friedrich / Mayer, Philipp: Patterns of interface-based programming, in: Journal of Object Technology 4:5 (2005), S. 75-94.

[Steimann / Mayer 2006]
Steimann, Friedrich / Mayer, Philipp / Meißner, Andreas: Decoupling classes with inferred interfaces, in: Proceedings of ACM Symposium on Applied Computing (2006), S. 1404-1408.

[Steimann / Mayer 2007]
Steimann, Friedrich / Mayer, Philip: Type Access Analysis: Towards informed interface design, in: TOOLS Europe 2007: Technology of Object-Oriented Languages and Systems (2007), S. 147–164.

[Tip u. a. 2003]
Tip, Frank / Kieżun, Adam / Bäumer, Dirk: Refactoring for generalization using type Constraints, in: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) (2003), S. 13-26.

Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst, nur die angegebenen Quellen und Hilfsmittel verwendet und Zitate als solche kenntlich gemacht habe.

Brühl, den ______________________________

Mark Thamm ______________________________