Studienarbeit

Berufsakademie Stuttgart - Außenstelle Horb
Staatliche Studienakademie

Fachrichtung Informationstechnik

Integration der Java-Typinferenz in eine Programmierumgebung

Juni 2005

Eingereicht von:

Markus Melzer
Waldhörnlestraße 7

72072 Tübingen

Firma:

Handy Tech Elektronik GmbH
Brunnenstraße 10

72160 Horb-Nordstetten

Betreuer:

Prof. Dr. Martin Plümicke

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Horb, den 17. Juni 2005

Zusammenfassung

Inhalt dieser Studienarbeit ist die Integration der Typinferenz in die Entwicklungsumgebung Eclipse. Die Typinferenz ermöglicht das Berechnen von Typannahmen in Java, ohne dass ein explizieter Typ angegeben werden muss. Dabei kann es vorkommen, dass mehrere Möglichkeiten zur Auswahl stehen. Dass der Anwender diese Auswahl treffen kann, ist auch Aufgabe dieser Studienarbeit.
Als erstes wird nach nötigen Grundlagen auf die Datenstruktur eingegangen, die nötig ist, damit die Typinferenz zum einen angezeigt, zum anderen der Anwender darauf arbeiten kann.
Des weiteren wird auf die konkrete Implementation des Eclipse-Plugins eingegangen, die für die Anzeige in Eclipse notwendig sind und die Aktionen des Anwenders entgegen nimmt.

Abstract

Contents of this study are the integration of the type inference into the development environment Eclipse. The type inference makes the calculation possible of type assumptions in Java, without a explizieter declaration of types. With the fact it can occur that several possible types can be selected. That the user can select one of the severeal types, is also task of this study. When first dealt with necessary bases, the data structure is discussed, which is necessary that the type inference can become on the one hand displayed, on the other hand the user can work on it.
The-further with the concrete implementation of the Eclipse Plugins one deals, which is note agile for display in Eclipse and which sreceive actions of the user.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Motivation

Bei vielen Programmiersprachen, so auch bei Java, ist es nötig, dass die Variablen und Methoden explizit einem Typ zugeordnet werden müssen. Nun kann die Typvergabe unter Java für den Programmierer schnell komplex und unüberschaubar werden. Durch die Einführung generischer Typen in der JDK 1.5 ist die Komplexität der Typendeklaration noch mehr gestiegen.

Für einen Java-Programmierer wäre es also hilfreich, wenn das Zuordnen der Typinformation die IDE bzw. der Compiler übernehmen könnte. Solche Unterstützungen sind von Programmiersprachen wie z.B. SML, PHP oder Phyton bekannt, bei denen keine expliziete Typangabe nötig ist.

1.2 Typinferenz

Unter Typinferenz versteht man das Folgern eines Typen, der nicht explizit angegeben ist. Durch die von Dr. Martin Plümicke unter (Plü04) spezifizierten Algorithmen ist es möglich, dass man die verschieden möglichen Typen bei Java berechnen kann. Die Implementierung dieses Algorithmus wird in der Studienarbeit „Typinferenz in Java“ (Bäu05) behandelt und ist Grundlage dieser Studienarbeit.

1.3 Projektgeschichte

Grundlage für dieses Projekt ist ein Java-Compiler, der vom Jahrgang TIT2000 an der BA-Horb in den Vorlesungen „Informatik 4“ und „Software Engineering“ entwickelt und implementiert wurde. Dieser Compiler wurde dann von

den beiden Studienarbeiten (Rei03) und (Haa04) um die generischen Typen erweitert. Die Studienarbeit (Ott04) behandelte den Unifikationsalgorithmus, der Voraussetztung für die Studienarbeit (Bäu05) ist, welche den Typinferenzalgorithmus implementiert .

1.4 Aufgabenstellung

Aufgabe dieser Studienarbeit ist es, in eine bestehende Java-IDE die Typinferenz zu integrieren. Damit soll es möglich sein, ohne expliziete Angabe der Typen Java-Code programmieren zu können.

Aufgabe dieser Integration soll auch sein, dass, wenn mehere Typen für eine Methode oder Variable möglich sind, diese anzuzeigen. Durch Auswahl kann der Programmierer dann einen Typ festlegen und somit kann der Compiliervorgang abgeschlossen werden.

Als IDE wurde Eclipse ausgewählt, weil diese durch ihr eigenes Framework gute Möglichkeiten bietet, eigene Plugins zu erstellen. Desweiteren ist Eclipse eine etablierte IDE für die Java-Entwicklung.

Kapitel 2

Theoretische Grundlagen

2.1 Generische Datentypen

2.1.1 Allgemein

Generische Datentypen wurden zum einen für die Implementation dieses Themas der Studienarbeit verwendet, zum anderen sind sie Bestandteil der Typinferenz, wo sie als gültige Typen berechnet und somit dem Benutzer zur Auswahl gegeben werden können. Deshalb werden die generischen Datentypen kurz vorgestellt´.

Ein wichtiges Prinzip in Programmiersprachen - vor allem in objektorientierten Programmiersprachen - ist es, die Datenstruktur und die Algorithmen, die auf einer solchen Datenstruktur angewendet werdem, voneinander zu trennen. Wenn dieses Prinzip verletzt wird, werden nicht zusammengehörige Konzepte vermischt oder es werden unnötige Abhänigkeiten geschaffen. Dazu wird die generische Programmierung verwendet.

Die Unterstützung der generischen Programmierung ermöglicht das Programmieren mit Typen als Parameter. Dieses wird besonders bei den Collection- oder Containerklassen gewünscht.

Es gibt zwei Arten der Realisierung von generischen Datentypen. In der Programmiersprache C++ wurde die *Kopierende Sicht* umgesetzt. Bei dieser Umsetzung wird die generische Definition des Typs als Platzhalter verwendet. Dabei wird - wie bei einem Makro - bei der Instanziierung der Klasse dieser Platzhalter durch den konkreten Typ ersetzt. So wird Code für jede Klasse erzeugt, die einen anderen Typ als Platzhalter verwendet.

Die zweite Realisierung ist die *generische Sicht*. Dadurch wird, im Gegensatz zur *Kopierende Sicht* nur einmal Code für diese Klasse erzeugt. Dies ist nur dann möglich, wenn die Typen den gleichen Speicherbedarf haben. Dies ist der Fall, wenn mit Referanzen gearbeitet wird. Bei Java Generics wurde

genau diese zweite Realisierung verwendet.

Im Folgenden wird auf Java Generic etwas näher eingegangen.

2.1.2 Java Generics

Vor Java 1.5 gab es in Java noch keine generischen Datentypen. Das folgende Beispiel zeigt auf, wie unter Java 1.4 ein Stack verwendet wurde:

```
// Java 1.4
Stack s = new Stack();
s.push(new Integer(1));
Integer i1 = (Integer)s.pop();  // Cast erforderlich
s.push("abc");                  // ist möglich
Interger i2 = (Integer)s.pop(); // Laufzeitfehler !!!
```

Durch die Einführung der Java Generics wird schon beim Deklarieren des Stacks angegeben, welche Arten von Typen er beinhaltet. Somit sind auch die Casts überflüssig, die zum einen Schreibarbeit für den Programmierer erspart und den Quellcode etwas übersichtlicher gestalten lässt. Die Anzahl von Programmierfehler und somit Testaufwand verringert sich. Die Typsicherheit ist somit höher, da schon zur Compilierzeit falsche Typen erkannt werden.

Das folgende Beispiel zeigt die Verwendung des Stacks in Java 1.5:

```
// Java 1.5
Stack<Integer> s = new Stack<Integer>();
s.push(new Integer(1));    // gleich wie bei Java 1.4
Integer i1 = s.pop();      // Cast nicht mehr erforderlich
s.push("abc");             // Compilerfehler !!!
```

2.2 SWT und JFace

2.2.1 Allgemein

In diesem Abschnitt wird auf das Grafiksystem der GUI eingegangen. Eclipse benutzt als Grafikapi SWT bzw. JFace. Dieses ist ein Alternative zu dem in der Javawelt weit verbreiteten AWT / Swing.

Im Folgenden wird auf die Konzepte der beiden Arten eingegangen.

2.2.2 AWT und Swing

In jedem Betriebssystem gibt es andere Grafikelemente für die GUI. Da Java plattformunabhängigen Code erzeugt, muss das GUI-Verhalten unter einen Hut gebracht werden. Dabei ist man bei AWT (Abstract Window Toolkit) den Weg gegangen, dass man die gemeinsame Schnittmenge aller Betriebssysteme nimmt und darauf dann seine eigenen Grafikelemte selber zusammenbaut.

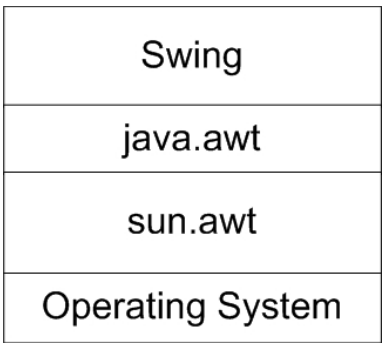

Abbildung 2.1: Schichten von AWT/Swing

Dabei setzt das java.awt auf das in C geschriebene sub.awt auf, welches die Schnittstelle zum darunterliegdenden Betriebssystem ist. Swing ist ein Framework, dass auf AWT aufbaut und so den Aufbau von GUI-Oberflächen ermöglicht.

Bis zur JDK 1.4 war die Grafikbibliothek komplett im Java implementiert. Ab der Version 1.4 gibt es manche „nativen“ Komponenten, bei denen Swing mit manchen Elementen des Betriebssystems korreliert.

2.2.3 SWT und JFace

Im Gegensatzt zu AWT ist SWT (Standard Widget Toolkit) eine plattformunabhängige Schnittstelle zu den GUI-Elementen des Betriebssystems. SWT reicht die verschiedenen Systemaufrufe an das Windowing-System weiter. Dazu verwendet es das JNI (Java Native Interface), welches ermöglicht, aus Java heraus C-Programme aufzurufen. SWT ist fast komplett in Jave implementiert, außer ein paar wenigen nativen Bibliotheken.

Da SWT quasi direkt auf dem Betriebssystem aufsetzt, ist das Look&Feel und das Ansprechverhalten wie in nativen Anwendungen des Betriebssystems. AWT und Swing sind etwas träger in der Anwendung, da sie viele Elemente der GUI selber implementiert haben und somit langsamer ablaufen

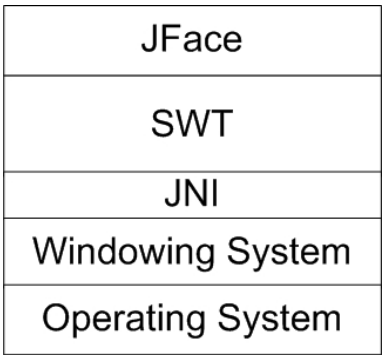

Abbildung 2.2: Schichten von SWT/JFace

JFace setzt direkt auf SWT auf und ist komplett in Java geschrieben. Somit hat es genauso das Verhalten wie native Anwendungen des Betriebssystems und nutzt die Vorzüge von Java.

2.3 Eclipse

2.3.1 Was ist Eclipse

Eclipse wurde von IBM entwickelt und wurde als 40-Millionen-Dollar Geschenk an die Open-Source-Geimeinde verschenkt. Die erste Version erschien im November 2001.

Eclipse wurde nicht als einfache IDE entwickelt. Damit man damit Java-Programme entwickeln kann, ist eine spezielle Anwendungsmöglichkeit von Eclipse. Inzwischen werden noch mehrere Sprachen wie C/C++ oder PHP unterstützt.

Eclipse besitzt eine Pluginstruktur, die es an verschiedensten Anwendungen anpassbar macht. So ist auch die Anwendung, dass ein Java-Programm entwickelt werden kann, ein Plugin in Eclipse.

Eclipse stellt die bereits erwähnten SWT- und JFace-Bibliotheken zur Verfügung und ist auch in diesen implementiert. Neben diesen Basisbiblitheken stellt Eclipse ein umfangreiches Framework für Java-Applikationen zur Verfügung. Es stehen höhere Klassen für z.B. Editoren, Viewer, Resourcenverwaltung, verschiedene Assistenten und Wizard für das Bauen von eigenene Applikationen zur Verfügung.

Ab Version 3 von Eclipse steht die Rich Client Plattform (RCP) zur Verfügung. Diese ermöglicht es, komplett eigene Anwendungen auf der Basis des Eclipsframwork zu entwickeln. Der Unterschied zwischen einen Plugin und des RCP ist der, dass der RPC auf das minimaleste Sytem von Eclipse auf-

setzt und somit die Verwendung des Eclipse-Workspaces und der Workspace-Ressourcen nicht mehr nötig ist. Für diese Studienarbeit spielt RPC keine Rolle.

2.3.2 Plugins - Erweiterung von Eclipse

Wie bereits erwähnt besteht Eclipse aus einer relativ kleinen Kernapplikation. Die eigentliche Funktionalität der konkreten Anwendung erwicken die Plugins. Dieses hat den Vorteil, dass man Eclipse sehr vielsietig benutzen kann, aber auch konkrete Features (also Plugins) entfernen kann und somit auf einem geringeren Niveau seine Applikation aufbauen kann.

Architektur

Der kleine Kern von Eclipse hat keine andere Aufgabe, als Plugins ablaufen zu lassen. Ein solches Plugin besteht meistens aus einem Jar-Archiv (das eigentliche Plugin) und zusätzlichen Dateien wie Bilder und Hilfetexten. Was nicht fehlen darf, ist das Plugin-Manifest in Form einer XML-Datei, das die Integration des Plugins in Eclipse beschreibt.

Ein Kernkozept von Eclipse sind die Extension Points. Über diese Erweiterungspunkte hängt sich das Plugin an die Plattform von Eclipse ein. In dem Plugin-Manifest werden die Erweiterungspunkte, die ein Plugin bereitstellt, und auch die Erweiterungspunkte, an die sich das Plugin hängt angegeben. Ein Beispiel ist, dass ein Plugin sich erst nach dem Öffenen einer `.cpp`-Datei aktiveiert. D.h. erst nachdem eine Datei mit der Endung .cpp geöffnet wurde, startet das Plugin, wenn es so in der plugin.xml definiert wurde.

Kapitel 3

Datenstruktur

3.1 Allgemein

In diesem Kapitel wird auf die interne Datenstruktur des Plugins eingegangen. Als erstes werden die Anforderung an die Datenstruktur beschrieben. Daraufhin wird die Datenstruktur, welche die eigentliche Typrekonstruktion (siehe (Bäu05)) anlegt und auch weitergegeben wird, analysiert. Schließlich wird die neu erforderliche Datenstruktur erläutert und wie sie implementiert ist.

3.2 Konzeption

3.2.1 Anforderungen

Für den Anwender ist es sinnvoll, dass er seine Auswahl, in diesem Fall zwischen Methode oder Variable, trifft und daraufhin den Typ angibt, den er haben möchte. Dabei sollte, wenn nur eine Möglichkeit besteht, auch nur diese eine Möglichkeit zur Auswahl stehen oder diese gleich vom Assistentem (dem Plugin) ausgewält werden. Des weiteren sollten auch, wenn manche Typen nicht mehr möglich sind, diese auch für den Anwender ausgeblendet werden.

Ein weitere Punkt ist, dass es für den Typrekonstruktionsalgorithmus nicht immer möglich ist, einen Typ festzustellen. Deshalb muss es möglich sein, dieses zu Erkennen und dem Anwender zu überlassen, was er für einen Typ haben möchte. Hier soll dem Benutzer die generischen Typparameter der Klasse zur Auswahl gestellt werden.

Am Ende muss, wenn alle noch unklaren Typen ausgwählt worden sind, die Weiterverarbeitung der Codegenerierung möglich sein. Dafür müssen die

Typen, die im Syntaxbaum noch keinen konkreten Typen zugewiesen bekommen haben, einen zugewiesen bekommen.

3.2.2 Datenstruktur der Typrekonstruktion

Der Typrekonstruktionsalgorithmus benutzt die im Folgenden analysierte Datenstruktur und gibt diese zur weiteren Verarbeitung weiter. Im Foldenden wird diese Datenstruktur erläutert.

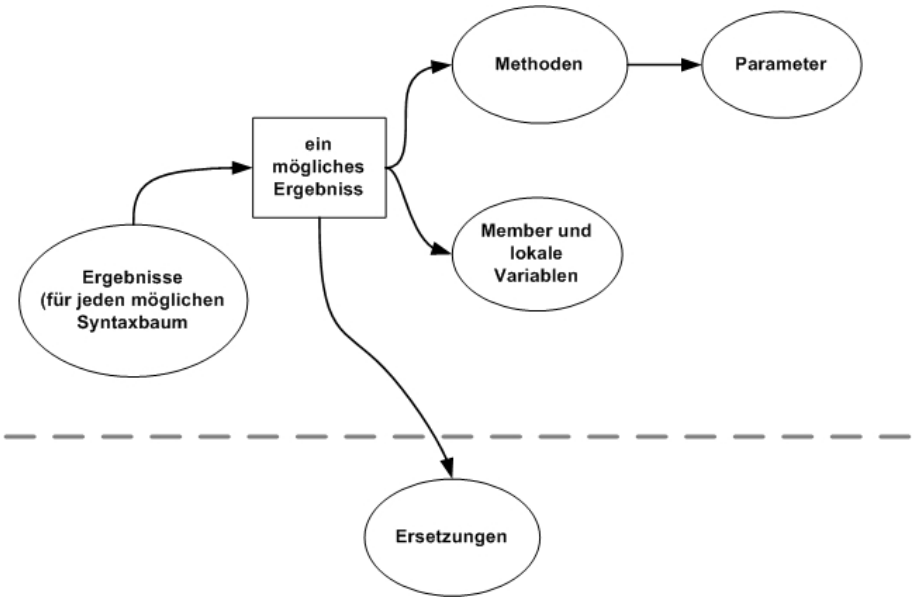

Abbildung 3.1: logische Datenstruktur der Typrekonstruktion

In der Abbildung 3.1 ist die logische Struktur dargestellt. Sie stellt sich wie folgt zusammen:

Als **Ergebniss** der Typrekonstruktion bekommt man eine Menge von Ergebnissen. Jedes dieser Ergebnisse repräsentiert einen gültigen Syntaxbaum - also eine mögliche Kombination von konkreten Typen. Sobald mindestens ein Typ nicht genau bestimmt werden konnte, d.h. es wurden mehrere mögliche Typen für einen Identifier gefunden, gibt es mehrere solcher Syntaxbäume.

Bei den Ergebnissen wurde jedoch noch nicht berücksichtigt, dass einem Identifier noch kein Typ zugeordnet werden konnte. Das geschieht z.B. dann, wenn keine Rückschlüsse auf diesen Identifier gezogen werden konnte (z.B. wenn eine Variable deklariert wird und sonst nirgends verwendet wird). Dieser stehen als *typlose Variable* im Syntaxbaum und wird keinem konkreten Typ zugeordnet. Für die Codegenerierung muss jedoch ein gültiger Typ zugeordnet werden.

Ein **einzelnes Ergebniss** entspricht - wie bereits erwähnt - einem gültigen Syntaxbaum. Dieses eine Eregebnis beinhaltet *Methoden*, *locale und Membervariablen* und die eigentlichen *Ersetzungen* als logische Teile.

Jede einzele Methode in der Menge der **Methoden** wird der Klasse zugeordnet, welcher sie angehört. Desweiteren besitzt diese eine *Typannahme*, die sie als Rückgabewert besitzt. Dieser Methode werden **Parameter** zugeordnet. Jeder dieser Parameter besitzt ebenso eine gültige *Typannahme*. Bedingt durch den Typrekonstruktionsalgorithmus können auch Methoden von anderen, nicht im geparsten Quellcode vorhanden, Klassen vorhanden sein. Diese müssen ignoriert werden.

In der Menge der **Member und lokalen Variablen** werden alle Variablen aller Klassen zugeordnet. Dabei wird, wie bei den Methoden, jede Variable eine Klasser, und jeder localen Variable, die Methode und der Block, in dem sie deklariert wird, zugeordnet. Diese Variablen besitzen auch eine *Typannahme*. Genauso wie bei den Methoden, können Variablen von anderen Klassen vorhanden sein, die ebenso ignoriert werden müssen.

Die letzte logische Einheit *eines Ergebnisses* ist die **Ersetzung**. In dieser sind die Verknüpfungen zwischen Typloser Variable im Syntaxbaum zur Typannahme abgespeichert. Diese gelten für dieses eine Ergebnis - also einem Syntaxbaum. Allerdings fehlen - wie bereits erwähnt - die Variablen, für die keine Annahme berechnet werden konnte.

3.2.3 Neue Datenstruktur

Gründe für die neue Datenstruktur

Da die Datenstruktur, die die Typrekonstruktion zurückgibt, sich nach dem Gesichtspunkt der Typen aufbaut, und nicht nach den eigentlchen Identifier (Variablen) entstehen folgene Probleme:

Zum einen besteht keine direkte Beziehung zwischen der eigentlichen Ersetzung und dem entsprechenden Identifier mehr. D.h. es müssen alle Ergebnisse zusammen untersucht werden, wo die einzelnen Idetifier sind, welchem konkreten Typ sie zugeordnet werden und wie sie voneinander abhängen.

Eine weiteres Erschwernis ist, dass Methoden, Member- und locale Variable keinerlei Hirarchie in der Datenstruktur haben. Ihnen wird eine Klasse bzw. eine Klasse und eine Methode zugeordnet. Aber möchte man z.B. alle Methoden einer Klasse haben, muss man alle Methoden durchgehen und jede überprüfen, ob sie zu dieser gesuchten Klasse gehören.

Da für jede Auswahl durch den Benutzer sich duch Abhängigkeiten alles verändern kann, muss jedes Mal mit großem Aufwand alles neu durchsucht werden und die noch gültigen Beziehungen gefunden werden.

Aufbau

Wegen den genannten Gründen wurde zu Gunsten einer neuen Datenstruktur entschieden. Der Inhalt dieser neuen Datenstruktur wird aus der Datenstruktur der Typrekonstruktion aufgebaut. Somit muss nur einmal die „alte“ Datenstruktur durchlaufen werden und danach kann auf die neue - dem Aufgabenbereich angepasste Datenstruktur - zugegriffen werden. Diese neue Datenstruktur ist wie folgt aufgebaut:

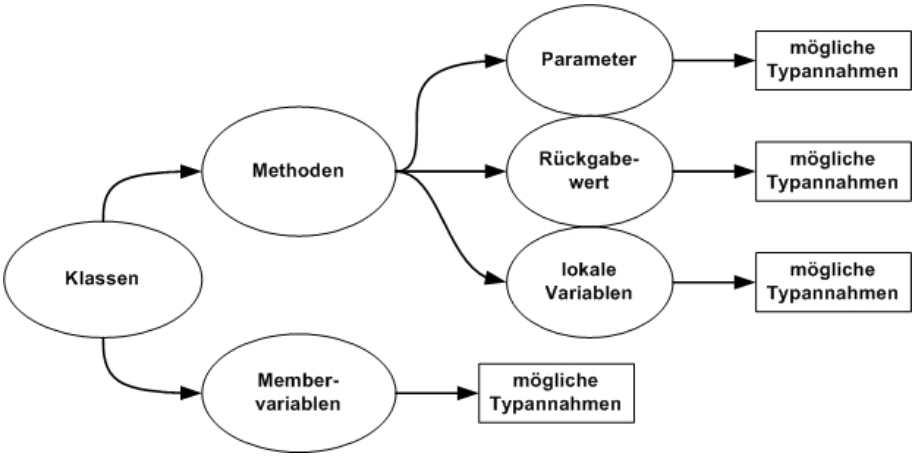

Abbildung 3.2: neue logische Datenstruktur

Diese Datenstruktur ist hirarchisch aufgebaut. Die oberste Ebene entspricht alle relevanten **Klassen**. Diesen Klassen sind die dazugehörigen **Methoden** und **Membervariablen** untergeordnet. Den Methoden selber sind die **Parameter**, der **Rückgabewert** und auch die **lokalen Variablen** untergeordnet. So kann jedes Identifier (Variable bzw. Methode) über die Hirarchie gefunden werden.

Den Parametern, dem Rückgabewert, den lokalen Variablen und den Membervariablen wird eine Menge von Typen zugeordnet, die für den jeweiligen Idetifier möglich sind. Dabei muss die Abhängigkeit zwischen den einzelnen Ergebnissen erhalten bleiben. So ist gewährleistet, dass bei der Auswahl einen Typen durch den Anwender alle nicht mehr möglichen Typannahmen ausgeblendet werden.

Die einzelen **Typannahmen** müssen so organisiert sein, dass wenn einem Identifier keinen Type zugeordnet werden konnte, die generischen Typparameter der Klasse genommen werden und diese dann beim späteren Erzeugen des Syntaxbaumes auch dort ersetzt werden.

3.2.4 Ersetzung von Typlosen Variablen

Damit ein gültiger Syntaxbaum erzeugt und somit dann auch Bytecode erzeugt werden kann, müssen alle typlosen Variablen durch die entsprechenden Typen ersetzt werden.

Wenn der Typrekonstruktionsalgorithmus eine Typannahme berechnen konnte, ist für diese typlose Variable bereits eine Ersetzung im Syntaxbaum enthalten. Wenn durch den Anwender alle noch nicht eindeutigen Typannahmen konkretisiert werden konnten, kommt nur noch ein Syntaxbaum in Frage. Für diesen Syntaxbaum können dann die konkreten Typen ersetzt werden.

Kann aber einer typlosen Variablen keine Typannahme (durch den Typrekonstruktionsalgorithmus) zugeordnet werden, muss die Auswahl des Anwenders noch in den Ersetztungen ergänzt werden, um einen gültigen Syntaxbaum zu erhalten.

3.3 Realisierung

3.3.1 Klassendiagramm

Nachdem jetzt die logische Struktur der Datenstruktur erläutert wurde, wird nun die Implementation näher erläutert. In Abbildung 3.1 wird grob die Klassenstruktur dargestellt. Dabei wurde kein Wert darauf gelegt, die komplette Klassen mit allen ihren Komponenten darzustellen, sondern nur die wichtigen Zusammenhänge darzustellen.

3.3.2 Implementierung

Die Klassen für die Datenstruktur kann man in drei Bereiche unterteilen: In Klassen für die *Hirarchie*, für die *möglichen Typen* und für die *Verwaltung*, welche Typanahmen überhaupt noch möglich und nicht mehr gültig sind.

Hirarchie

Die Klasse `JavDataStruct` ist die Schnittstelle zur Datenstruktur. Sie stellt alle Methoden zur Verfügung, welche zum Arbeiten mit der Datenstruktur nötig sind. Diese Klasse implementiert eine Hashmap, die alle Klassen beinhaltet, die in der Datenstruktur vorhanden sind. Als Schlüssel wird der Klassenname und als Wert die einzelne Klasse - gekapselt in der Klasse `JavClassIdent` - gespeichert.

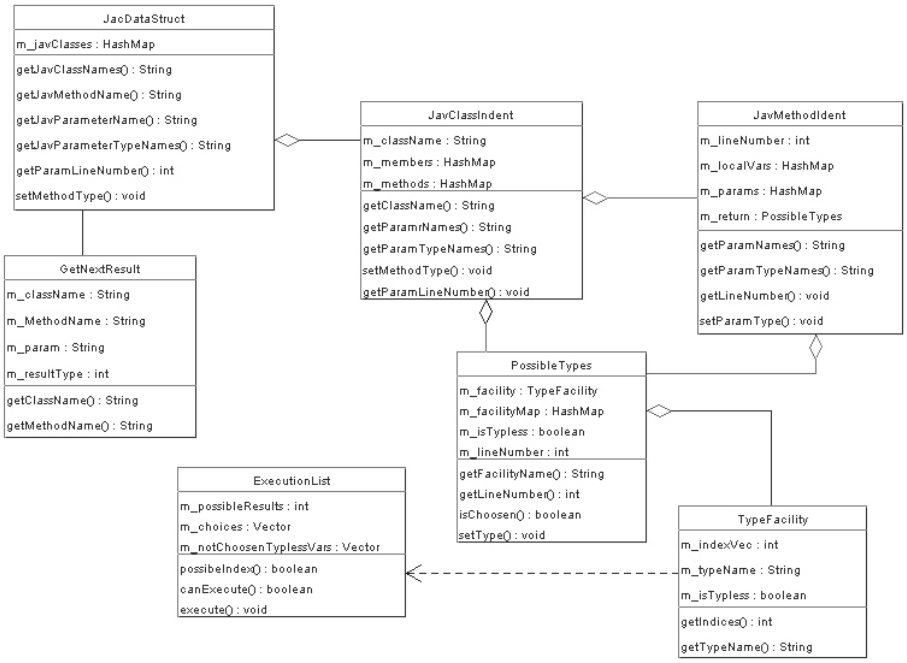

Abbildung 3.3: Klassendiagramm (sehr vereinfacht)

Jede einzelne Klasse besitzt zwei Hashmaps, die zum einen die Methoden und zum anderen die Membervariablen abspeichert. Als Schlüssel dient jeweils der Methoden- bzw. Membervariablenname.

Die Methoden wiederum sind in der Klasse `JavMethodIdent` gekapselt. Diese Klasse besitzt zwei Hashmaps, die lokalen Variablen in der Methode und die Parameter der Methode abspeichern.

Sowohl die Werte der letzten beiden Hashmaps, wie das Member `m_return` der Klasse `JavMethodIdent` und der Wert der Hashmap in der Klasse `JavClassIdent`, beinhalten die Klasse `PossibleTypes`, welche die möglichen Typannahmen kapselt.

Wie ein Aufruf z.B. von allen Parametern einer Methoden von einer Klasse abläuft, zeigt Abbildung 3.1. Die Anfrage z.B. der Parameternamen einer Methode, wird an die Klasse `JavMethodIdent` gegeben. Diese deligiert es weiter an die untergeordneten Klasse `JavClassIdent`. Diese deligiert wiederum an `JavMethodIdent`, die diese dann ausführt. Mit dem gleichen Prinzip funktionieren die anderen Methodenaufrufe auf die Datenstruktur.

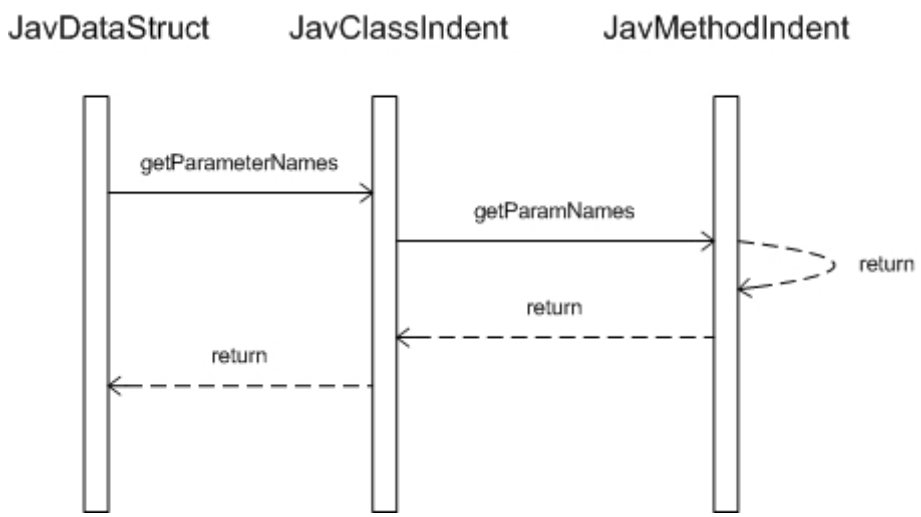

Abbildung 3.4: Aufruf für Parameternamen einer Methode

Mögliche Typen

Wie bereits erwähnt stehen die Typannahmen in der Klasse `PossibleTypes`. Diese beinhaltet alle möglichen Typen, die für den einen Identifier möglich sind. Jede mögliche Typannahme ist wiederum in der Klasse `TypeFacility` gekapselt. Diese Klassen sind in einer Hashmap gespeichert, die als Schlüssel den Typnamen und als Wert genau diese Klasse weiß.

Eine `PossibleTypes`-Klasse kann dann in Verbinding mit der Klasse `ExecutionList` zusammen feststellen, welche Typen noch möglich sind und welche nicht. Dieses ist möglich, da in jedem Typannahme (`TypeFacility`) die möglichen Syntaxbäume als Index mit abspeichert werden. Wird ein Typ ausgewählt, speichert er sich zusätzlich in der Klasse `ExecutionList`. Diese stellt dann mit den mit abgespeichertet Indices der Syntaxbäume, welche Typen noch gültig sind und welche nicht mehr. Beim Vergleich der noch gültigen Syntaxbäumen mit denen vom der aktuell betroffenen Typannahmen, kann die Gültigkeit festgestellt werden.

Da es jetzt möglich ist, festzustellen, welche Typannahmen noch möglich sind, gibt die Methode `getJavParamTypes()` alle Möglichen Typen eines Parameters einer Methode zurück. Ist nur eine Typ möglich, wird diese genommen.

Suchen nächsten Identifier

Da der Anwender nicht alle Identifier durchgehen möchte, bei denen unter Umständen die meisten Typannahmen schon eindeutig sind, ist eine Suche

nach dem nächsten, nicht eindeutigen Identifier wünschenswert.

Damit dieses möglich ist, sucht die Methode `findNextToChoose()` in der Klasse `JavDataStruct` alle ihre Unterhirarchien durch, ob ein Identifier noch nicht eindeutig festgelegt ist. Wird ein Identifier gefunden, gibt diese Methode die Klasse `GetNextResult` zurück. In dieser Klasse sind dann alle nötigen Information wie z.B. Klassenname, Methodennamen und Parametername gespeichert. Somit kann die Anwendung den entsprechenden Identifier direkt anzeigen.

Aufbau der Datenstruktur

Damit überhaupt das Arbeiten auf der Datenstruktur möglich ist, muss diese erstellt und befüllt werden. Dazu ist die Methode `Initialize()` verantwortlich. Diese geht das Ergebniss der Typrekonstruktion durch, also jeden einzelnen Syntaxbaum, und füllt nach und nach die hirarchische Struktur.

Wird für einem Identifier in einem weiterem Syntaxbaum ein bereits zugeweisen Typannahme gefunden, wird dieser Typ nicht nochmal hinzugefügt, sondern bei dem bereits existenten Annahme die Nummer des Syntaxbaumes hinzugefügt. So ist eine eindeutige Zuordnung von Typannahme zu einem gültigen Sytntaxbaum gewährleistet und es können Rückschlüsse auf die noch gültigen Typannahmen durch die Indizes der Syntaxbäume geschlossen werden.

Für typlose Variablen, die keinem Typen zugeordnet werden können, werden als Typannahmen die generischen Parameter der Klasse angenommen. Dafür wird jedoch die Instance der Klasse `PossibleTypes` anders initialisiert. Diese braucht jetzt Informationen, damit nachher diese typlosen Variablen im Syntaxbaum ersetzt werden können. Diese ist sonst nicht nötig, da diese Ersetzungen bereits in dem Ergebniss der Typrekonstruktion für jeden gültigen Syntaxbaum vorhanden sind.

Fertiger Syntaxbaum

Wurden alle Typannahmen ausgewählt, d.h. es gibt nur noch eindeutig zugordnete Typen für jeden Identifier, kann jetzt der Syntaxbaum erstellt werden. Dies wird dann festgestellt, wenn nur noch ein gegebener Syntaxbaum gültig ist und alle typlosen Variablen einen generischen Typen zugewiesen bekommen haben.

Um den Syntaxbaum zu erstellen, müssen erst alle typlosen Variablen zu den Ersetzungen hinzugefügt werden. Da ja nur noch ein Syntaxbaum gültig ist, wird dieses Ergebniss aus der Typrekonstruktion genommen und die einzelnen Eretzungen der typlosen Variablen dort hinzugefügt.

Danach kann auf diesen Syntaxbaum ein `execute()` aufgerufen werden, welches dann die Typannahmen durch die tatsächlichen Typen ersetzt. Somit hat man einen gültigen Syntaxbaum, der zur weiteren Verarbeitung benutzt werden kann.

Kapitel 4

Graphic User Interface

4.1 Aufgabe der GUI

Nachdem in Eclipse eine Datei mit der Endung .jav geöffnet wird, soll das Plugin aktiviert werden. Die Endung `.jav` wurde deswegen gewählt, dass es nicht zu Problemen mit den normalen `.java`-Dateien kommt, da es wegen den eventuel fehlenden Typdeklarationenen nicht 100% mit der Java-Syntax kompatibel ist.

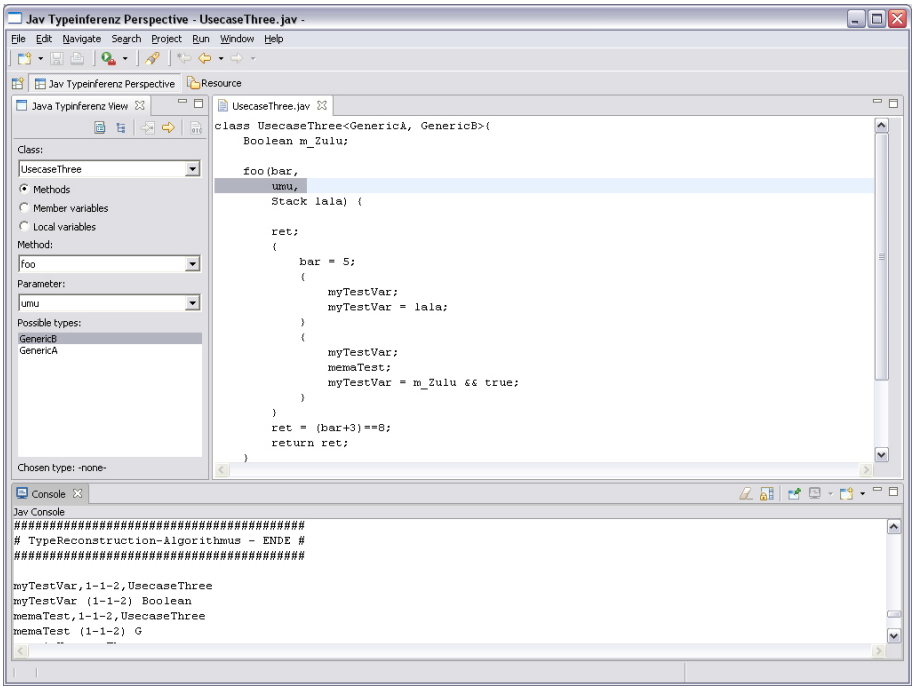

Abbildung 4.1: fertige GUI

Der Anwender soll in der GUI die möglichen Identifier auswählen können. Für den jeweiligen Identifier kann der Anwender dann eine Typannahme treffen, wenn nicht schon durch den Typrekonstruktionsalgorithmus eine eindeutige Annahme getroffen werden konnte oder andere Annahmen ausgeschlossen werden konnten.

Desweiteren soll über die GUI, außer dem Öffnen und Speichern einer `.jav`-Datei auch das Editieren einer solchen Datei und die komplette Codeerzeugung vom parsen bis zum codegenerieren möglich sein. Dazu soll die API der Typreferenz (siehe (Bäu05)) integriert werden.

Die zusätzliche Anzeige in Eclipse von Ausgaben, die bei der Codeerzeugung auf die Standartconsole geschrieben werden, sollen ebenso in einem Fenster angezeigt werden. Ausgaben sind z.B. Fehlermeldungen und Debuggausgaben.

In den folgenden Abschnitten wird erst auf die Konzeption der GUI eingegangen und daraufhin auf die konkrete Implementation.

4.2 Konzeption

4.2.1 Zustand der GUI

Wie bereits erwähnt muss für den Anwender vom Öffnen von Dateien bis hin zur Codegenerierung alles anwendbar sein. Dazu muss die GUI in immer einen anderen Zustand gebracht werden. Man kann schließlich kein Dokument parsen, wenn es noch nicht geöffnet bzw. ausgewält wurde.

Abbildung 4.2 zeigt die verschiedenen Zustände, in der das Plugin sich befinden kann. Dabei werden nur die Hauptzustände betrachtet. Für die Ansicht und Auswahl der verschiedenen Typannahmen der Identifier werden noch mehr Zustände gebraucht, aber darauf wird später eingegangen. Hier werden nur globalere Zustände betrachtet.

Diese Zustände sind für die einzelnen GUI-Elemente erforderlich. Jedes GUI-Element kennt den aktuellen Zustand des Plugins und kann sich somit an die Gegebenheiten anpassen, d.h. es deaktiviert sich, zeigt Informationen an oder kann etwas ausführen. Im Folgenden werden nun die einzelnen Zustände näher erläutert:

Kein .jav Dokument geöffnet

Bei diesem Zustand ist noch kein `.jav`-Dokument geöffnet und somit kann das Plugin noch nichts machen und ist somit deaktiviert.

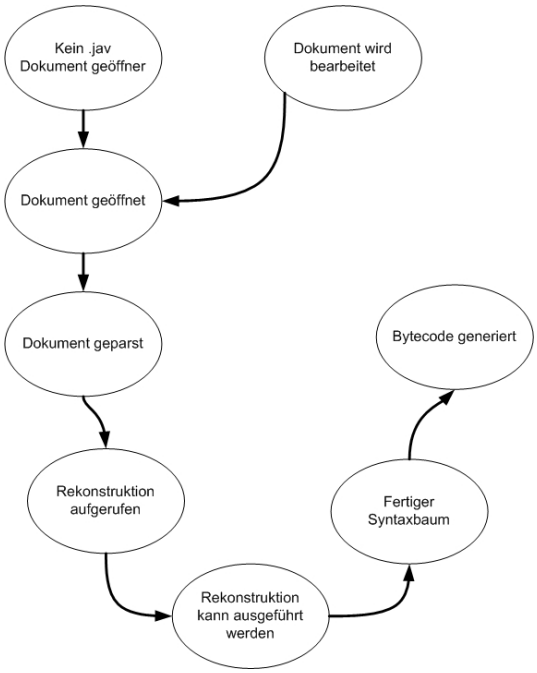

Abbildung 4.2: Hauptzustände der GUI

Dokument geöffnet

Wenn eine `.jav`-Datei im Editor geöffnet wird, wird dieser Zustand erreicht. Zum einen können jetzt die im Editor üblichen Aktionen durchgeführt werden (z.B. Text editieren), als auch, wenn das Dokument im aktuellen Zustand gespeichert ist, geparst werden. Die ist der ersten Schritt der Codeerzeugung.

Dokument geändert

Wird im Editor das aktuelle `.jav`-Dokument geändert, so wird dieser Zustand erreicht. Das Editieren dieser Datei ist grundsätzlich zu jeder Zeit möglich. Jetzt darf kein Schritt mehr für die Codegenerierung gemacht werden. Erst wenn das `.jav`-Dokument gespeichert wurde, kann weiter gemacht werden. Es muss jedoch die Codegenerierung wieder von vorne begonnen werden, d.h. das Dokument muss erst wieder geparst werden.

Dokument geparst

Dieser Zustand wird erreicht, wenn dass Dokument erfolgreich geparst wurde und damit eine Weiterverarbeitung für die Codeerzeugung möglich ist.

Rekonstruktion aufgerufen

Sobald der Typrekonstruktionsalgorithmus erfolgreich angewandt wurde, ist dieser Zustand erreicht. Dabei gibt es noch Identifier, die noch nicht eindeutig sind. Sonst wird gleich der nächste Zustand erreicht.

Syntaxbaum kann eretzt werden

Dieser Zustand ist erreicht, wenn allen typlosen Variablen im Syntaxbaum genau eine Typanname zugeordnet wurde. Somit kann die Codeerzeugung fortgesetzt werden.

Syntaxbaum wurde ersetzt

Dieser Zustand wird erreicht, wenn alle typlosen Variablen im Syntaxbaum durch ihre Annahme ersetzt worden sind.

Bytecode wurde erzeugt

Wenn schließlich der Bytecode des .jav-Dokuments erzeugt wurde, ist dieser Zustand erreicht.

4.2.2 Logik der Typauswahl

Der Anwender möchte die Identifier, die in der besprochenen Datenstruktur liegen, auswählen können. Um das zu verwirklichen können, muss es auch eine Art Statemachine geben, damit die Auswahl der Identifier möglich ist.

Da die Daten hirachisch vorliegen, muss der Anwender sich durch die Hirarchie durcharbeiten. So wissen die einzelnen Grafikelemente, die bei der Typauswhl betroffen sind, dass wenn das Element, das in der Hirarchie vor ihnen liegt, geändert wird, sie sich aktuallisieren müssen. Sie müssen sich nicht aktallisieren, wenn ein Element, das in der Hirarchie danach kommt, sich ändert. Da es drei Arten von Identifiern gibt (Attribute, Methodenparameter und Rückgabewert und lokale Variablen) und sich die Hirarchie für diese in manchen Punkten unterscheidet, muss für jede Art die Logik angepasst werden.

Da in der Datenstruktur für jeden Identifier die jeweilige Zeile gespeichert ist, kann, wenn der Anwender einen konkreten Identifier ausgewählt hat, diese Zeile im Editor markiert werden.

Durch die in der Datenstruktur implementierte Suchfunktion für den nächsten nicht eindeutigen Identifier, können die Felder, die für die Aus-

wahl des Identifier ausgefüllt werden, automatisch gesetzt werden. Somit ist das direkte Anspringen eines Identifiers ebenso möglich.

4.3 Realisierung

4.3.1 Allgemein

Da auch das ganze Eclipse-Framework objektorientiert aufgebaut ist, ist es möglich, die einzelnen Aufgaben der GUI in Module zu zerlegen. Auch durch die Mächtigkeit des Eclipse-Framworkes können für viele Elemente, wie z.B. der Editor, auf bereits im Framework implementierte Klassen zurückgegriffen werden.

Im Folgenden werden die Realisierungen der einzelnen Bereiche der GUI näher erläutert.

4.3.2 Verwaltung der einzelnen Bereiche

Da jeder Bereich (Editor, Anischt, ...) für sich alleine steht, braucht man eine zentrale Instanz, über die die einzelenen Bereiche Informationen austauschen können. Diese Aufgabe übernimmt die Klasse `PluginMediator`. Eine schematische Darstellung ist in Abbildung 4.3 dargestellt.

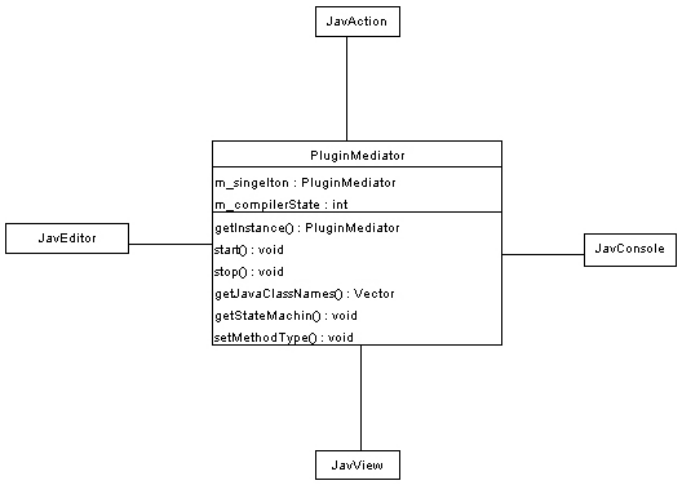

Abbildung 4.3: Klassendiagramm PluginMediator (vereinfacht)

Diese Klasse ist nach dem Patterntyp *Singeltion* aufgebaut. D.h. es existiert immer nur eine Instanz dieser Klasse. So ist gewährleistet, dass alle Elemente des Plugins immer auf die selbe Klasse zugreifen.

Diese Klasse ist auch der Haupteinstiegspunkt des Eclipsplugin. D.h. diese Klasse wird als erstes beim Start des Plugins aufgerufen.

Hauptaufgabe der Klasse `PluginMediator` ist, dass alle einzelnen Module miteinander komunizieren können. Das funktionier so, dass beteiligte Klassen dort registrieren können. Zum anderen stellt sie Methode bereit, über die die einzelnen Module z.B. auf die Datenstruktur oder auf die Funktionen der API für die Typrekonstruktionsalgorithmus (Parsen, Rekonstruktion, Codegenreierueung) zugreifen können.

4.3.3 Aktionen

Unter Aktionen versteht man, dass man z.B. über das Menü oder einem Symbol eine Aktion durchführen kann. In diesem Plugin sind die Aktionen die einzelnen Schritte für die Codeerzeugung. Die einzelnen Schritte sind *Parsen*, *Typrekonstruktion*, *Suchen nach dem nächsten noch unbestimmten Identifier*, *Sytnaxbaum ersetzten* und *Bytecodeerzeugung*.

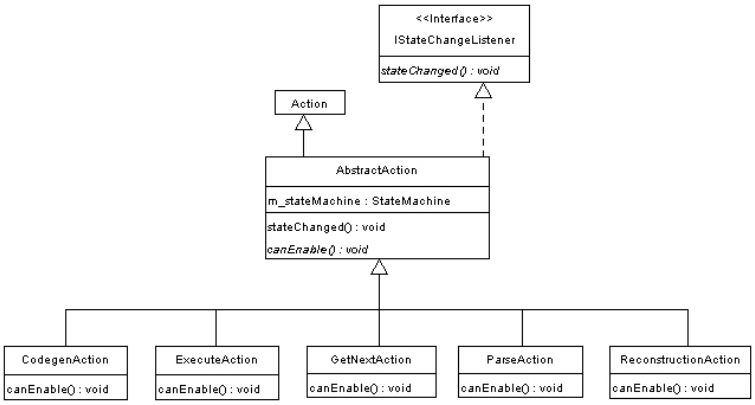

Abbildung 4.4: Klassendiagramm der Aktionen

Abbildung 4.4 zeigt die Implementation der Aktionen. Ist eine Aktion aktiviert und wird sie aufgerufen, kann sie so die gewollte Aufgabe erledigen.

4.3.4 Ansicht

Über die Ansicht soll es den Anwender möglich sein, dass einem Identifier ein Typ zugeordnet werden kann. Die Auswahl des Identifiers erfolgt hirrachisch (siehe Datenstruktur).

Als Elemnte der Ansicht wurden Comboboxen, Labels und eine Liste verwendet. In den Comboboxen kann man z.B. die verschiedenen Klassen, eine Methode oder einen Parameter auswählen, um an den gewünschten Identifier

zu gelangen. Die Labels werden für die Beschriftung der Comboboxen und für die Anzeige des Types des gerade ausgewählten Identifiers verwendet. In der Liste werden alle noch möglichen Typannahmen angezeigt.

Das Aktuallisieren der einzelnen GUI-Elementen geschieht durch das Interface `IViewChangeListener`, das von jedem Element implementiert wurde. Ändert sich eine Element, wird die Methode `viewChanged` aufgerufen und alle, die das Interface implementieren wissen, dass sich was geändert hat.

4.3.5 Editor

Für Editoren gibt es im Eclipsframework schon fertig implementierte Klassen. Diese haben schon sämtliche Funktionalität, die ein Editor braucht, wie Bearbeiten von Texten und Arbeiten mit der Zwischenablage.

Für die konkrete Implementation (`JavEditor`) wurde hinzugefügt, dass wenn diese Klasse instaziiert wird, eine Referenz in der Klasse `PluginMediator` gespeichert wird, damit andere Module des Plugins auf diesen Editor zugreifen können.

Außerdem muss in der `plugin.xml` noch der Extension Point definiert werden, damit beim Öffnen einer `.jav`-Datei auch der neue Editor gestartet wird und nicht der Standarteditor von Eclipse.

Des weiteren wurde die Methode `markLine()` hinzugefügt, dass eine bestimmte Zeile markiert werden soll. Dies wird gebraucht, um die Zeile, in der sich gerade in der Ansicht ausgewählte Identifier befindet, hervorzuheben.

4.3.6 Konsole

Da der benutzte Java-Compiler und die Typrekonstruktion Informationen auf die Standartausgabe rausschreiben, wurde die Console eingefügt.

Das Eclipsframework bietet Konsolenklassen, die in einem Fenster einzelne Zeilen ausgeben können. Dabei wurde die Standartausgabe so umgebogen, dass sie die Informationen nicht mehr in der Konsole des Betriebssystems ausgeben, sondern in dem Fenster in Eclipse.

Die Konsole ist auch für die Entwicklung des Plugins hilfreich, da auf ihr komfortabel Debuggausgaben gemacht werden konnten, wenn diese einfach auf die Standartausgabe geschrieben werden.

4.3.7 Perspektive

Eclips verwaltet verschiedene Perspektiven. Unter einer Perspektive versteht man die Anordnung der einzelnen Ansichten und Editoren in Eclipse, die beliebig angeordnet werden können.

Damit man nicht immer die Ansicht neu ordnen muss, wurde eine Perspektive definiert, in der die Anordnung für die in diesem Anwendungsfall wichtigen Fenster bestimmt. Diese Perspektive muss ebenfalls in der `plugin.xml` als Extension Point deklariert werden, damit der Anwender in Eclipse diese Perspektive auswählen kann.

Kapitel 5

Ausblick

5.1 Rückblick

Das Plugin für Eclipse wurde im Rahmen dieser Studienarbeit implementiert. Zum einen wurde die API (siege (Bäu05) für die Erzeugug des Bytecodes und die Typeberechnung integriert. Um den Anwender die Identifier und die durch die Berechnung nicht eindeutigen Typannahem zur Auswahl zu geben, wurde eine neue Datenstruktur implementiert, die dieses einfacher ermöglicht. Für die Anzeige der Auswahlmöglichkeitet wurden die entsprechenden Ansichten implementiert.

Dieses Plugin wurde für die Usecases, die der Typrekonstruktionsalgorithmus beherrscht, getestet.

5.2 Ausblick

Es wäre wünschenswert, wenn die Auswahl der Identifier nicht nur über das Auswahlmenü möglich wäre, sondern auch über den Quellcode im Editor auszuwählen wäre. Zwar wurde im späten Verlauf dieser Studienarbeit jedem Identifier eine Zeile, in die er steht, mitgeliefert. Diese Zeile bezieht sich jedoch nur auf die Zeile, in der er deklariert wird. Es müssten also Erweiterungen getroffen werden, in der die genaue Position des Identifier im Quellcode, und zwar überall, wo er vor kommt (und nicht nur wo er Deklariert wird) festzustellen ist, damit man rückwärts von der Stelle im Quellcode auf den Identifier und seine Typannahmen kommen kann.

Es kostet viele Resourcen, die Datenstruktur, die der Typrekonstruktionsalgorithmus zurückgibt, in die neue Datenstruktur umzusetzten. Eine einzige Datenstruktur wäre die bessere Möglichkeit. Dazu muss aber abgelärt werden, ob diese beiden Datenstrukturen, die unterschiedlichen Anforderungen

genügen müssen, in einee einzigen gebündelt werden können.

Literaturverzeichnis

[Bäu05] Bäuerle, Jörg: *Typinferenz in Java.* BA-Horb, 2005

[Dau04] Daum, Berthold: *Java-Entwicklung mit Eclipse 3 (Anwendungen, Plugins und Rich Clients).* 2., überarbeitete und erweiterte Auflage. Heidelberg : dpunkt.verlag GmbH, 2004

[ecl05] *Eclipse 3.0.1 Documentation*, Juni 2005. – http://help.eclipse.org/help30

[Haa04] Haas, Markus: *Weiterentwicklung der Java-Codegenerierung zur Ausführung von parametrisierten Datentypen.* BA-Horb, 2004

[jav05] *JavaTM 2 Platform Standard Edition 5.0 API Specification*, Juni 2005. – http://java.sun.com/j2se/1.5.0/docs/api/

[Ott04] Ott, Thomas: *Typinferenz in Java.* BA-Horb, 2004

[Plü04] Plümicke, Dr. M.: *Type Inference in Gereric Java.* 2004

[Rei03] Reichenbach, Felix: *Erweiterung der semantischen Analyse des Java-Compilers.* BA-Horb, 2003