S T U D I E N A R B E I T

Berufsakademie Stuttgart – Außenstelle Horb-
Staatliche Studienakademie

Studiengang Informationstechnik
TIT2003

Einbindung von Java – Typinferenz in eine integrierte Entwicklungsumgebung

Juni / 2006

Eingereicht von:
Thomas Hornberger
Eugen-Nägele-Straße 20
72250 Freudenstadt

Firma:
Häfele GmbH & Co KG
Adolf-Häfele-Str. 1
72202 Nagold

Betreuer:
Prof. Dr. Martin Plümicke

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Horb, den 15. Juni 2006

_____

Thomas Hornberger
TIT2003, BA Stuttgart - Außenstelle Horb a. N.

Abstract

In a previous student research project, there has been developed a first version of an Eclipse-plug-in for Java with type inference.
The present student research project deals with the development and implementation of a concept to improve the existing plug-in. The major task was to chart all important type information of the compiler in a clear and traceable way. At the same time the compiler has been advanced by two other fellow students. The progress of their work should also harmonize with the plug-in.

Zusammenfassung

In einer Vorgängerstudienarbeit wurde eine erste Version eines Eclipse-Plugins für Java mit Typinferenz entwickelt.
Diese Studienarbeit befasst sich mit der Entwicklung und Implementierung eines Konzepts zur Erweiterung des bestehenden Plugins. Die Hauptaufgabe der Arbeit bestand darin, alle wichtigen Typinformation des Compilers entsprechend anschaulich und für Benutzer nachvollziehbar grafisch darzustellen. Zeitgleich wurde auch der Compiler durch zwei weitere Studienarbeiten weiterentwickelt. Der hier erzielte Fortschritt sollte auch stets mit dem Plugin in Einklang gebracht werden.

Inhalt

Ehrenwörtliche Erklärung **2**

Abstract **3**

Verzeichnis von Abbildungen und Quellcodebeispielen **6**

Abkürzungen **7**

1. Einleitung **8**
1.1. Entstehung der Studienarbeit 8
1.2. Aufbau der Arbeit 8

2. Vermittlung von Grundlagen **9**
2.1. Die Rich Client Platform in Eclipse 9
2.1.1. Motivation und Abgrenzung 9
2.1.2. Plugins und die RCP 10
2.1.3. Architektur 10
2.2. Plugins für die Eclipse-Workbench entwickeln 11
2.2.1. Die Klasse Plugin 11
2.2.2. Die Klasse Preferences 11
2.2.3. Die Klasse Platform 11
2.2.4. Die Klasse Resource 11
2.2.5. Die Konfigurationsdatei plugin.xml 12

3. Analyse und Anforderungen **13**
3.1. Organisation in Eclipse 13
3.2. Kompiliervorgang 13
3.3. Ergebnisanzeige der Typrekonstruktion 13
3.4. Typauswahl bei nicht eindeutiger Typrekonstruktion 14
3.5. Fehlerhandling 14

4. Design **15**
4.1. Allgemein 15
4.2. Error-Handling 16
4.3. Sonderfunktionen 17
4.4. Zusammenspiel der Komponenten 18

5. Implementierung **19**
5.1. Vorwort 19
5.2. Änderungen am CompilerCore 19
5.2.1. Parser und Scanner 19
5.2.2. Skeleton 19
5.2.3. Offsets einer Typannahme 20
5.2.4. IItemWithOffset 20
5.3. Plugin.xml 20
5.4. ShortCutAction 22
5.5. ContextAction 22
5.6. Outline 23
5.7. OutlineSection 23
5.8. Datastruct 24
5.9. JavEditor 24
5.9.1. Einleitung 24
5.9.2. makeTooltip() bzw. makeWarning() und makeError() 25
5.9.3. refreshTooltips() 26
5.9.4. getAdapter 26

5.9.5. editorContextMenuAboutToShow 26
5.10. Console 27

6. Schlussbetrachtung 28
6.1. Auswirkungen der Studienarbeit 28
6.2. Persönliche Erkenntnisse 28
6.3. Ausblick 29

7. Quellenverzeichnis 30
7.1. Internet 30
7.2. Literatur 30

Anlagen 31

A Klassendiagramme 31

A.1 typeInferenz 31

A.2 actions 32

A.3 console 33

A.4 helpers 33

A.5 editors 33

A.6 datastruct 36

Verzeichnis von Abbildungen und Quellcodebeispielen

Abbildung 1: Beispiel von Erweiterungspunkten in der RPC 10
Abbildung 2: Eclipse-Architektur 10
Abbildung 3: Beispiel einer plugin.xml 12
Abbildung 4: Grundsätzliche Funktionalität des Plugins 15
Abbildung 5: Reaktion im Fehlerfall, Beispiel 1 16
Abbildung 6: Reaktion im Fehlerfall, Beispiel 2 16
Abbildung 7: Typauswahl bei uneindeutiger Typrekonstruktion 17
Abbildung 8: Zusammenspiel der Komponenten 18
Abbildung 9: Code-Beispiel für Typinferenz 20
Abbildung 10: Implementierung des Erweiterungspunktes Highlight-Annotation 21
Abbildung 11: Starten der Typrekonstruktion 22
Abbildung 12: Aktualisierung der Outline 23
Abbildung 13: Dokument und Annotationsmodell 25
Abbildung 14: Listing von makeTooltip 25

Abkürzungen

SWT: Das Standard Widget Toolkit ist eine Bibliothek von IBM für grafische Oberflächen mit Java.

AWT: Das Abstract Window Toolkit ist Bestandteil der Java Foundation Classes (JFC) und stellt eine Standard-API zur Erzeugung und Darstellung einer plattformunabhängigen grafischen Benutzerschnittstelle (GUI) für Java-Programme dar.

Swing: Bei Swing handelt es sich um eine API zum Programmieren von grafischen Benutzeroberflächen. Swing wurde von Sun Microsystems für Java entwickelt, in welches es seit der Java-Version 1.2 (1998) integriert ist.

IDE: Eine integrierte Entwicklungsumgebung ist eine Anwendung zur Entwicklung von Software.

RCP: Rich-Client-Applikation. Siehe 2.1.2.

1. Einleitung

1.1. Entstehung der Studienarbeit

Im Rahmen eines Software-Projektes eines Vorgängerkurses wurde einmal ein erster Prototyp eines Java-Compilers erstellt. Dieser wurde seither durch etliche Studienarbeiten immer wieder verbessert und erweitert. Vor allen Dingen aber wurde die Typinferenz in den Compiler implementiert.

Um den Compiler auch sinnvoll über das Eclipse-Framework nutzen zu können, wurde dann in einer weiteren Vorgänger-Studienarbeit ein Eclipse-Plugin entwickelt. Da jedoch noch mehr Potential in dem Plugin steckt gilt es nun, dieses zu analysieren und zu erweitern.

1.2. Aufbau der Arbeit

Als Einführung in das Thema „Typinferenz in Java - IDE“ werden im zweiten Kapitel dieser Arbeit die allgemeinen Grundzüge der Plugin-Entwicklung beschrieben. Im nächsten Abschnitt wird der bisherige Stand des Plugins analysiert und gewünschte Anforderungen definiert. Der vierte Abschnitt beschreibt das grafische Aussehen des Plugins, sowie die Handhabung der GUI. Im fünften Hauptpunkt geht es um das eigentliche Coding des Plugins, also wie die Software implementiert ist bevor schließlich im sechsten Kapitel eine Schlussbemerkung diese Studienarbeit abschließt.

2. Vermittlung von Grundlagen

2.1. Die Rich Client Platform in Eclipse

2.1.1. Motivation und Abgrenzung

Die Rich Client Platform (RCP) ist das wohl wichtigste neue Feature in Eclipse 3. Wenigstens sehen dies laut Umfragen sehr viele Eclipse-Benutzer so. So basieren denn auch schon verschiedene Klienten von IBMs neuer Lotus-Version auf der Eclipse RCP. Grob gesagt erlaubt es die RCP, auf der Basis von Eclipse eine weite Klasse von Anwendungen zu erstellen und dabei die Eclipse-Plattform als Framework zu nutzen.

Unter einem Rich Client versteht man Software, die applikationsspezifische Funktionalität direkt beim Klienten (z.B. Desktop oder mobile Plattform) implementiert. Das Gegenteil ist nicht der Poor Client, sondern der Thin Client, bei dem applikationsspezifische Funktionen vom Server aus gesteuert werden. Ein typisches Beispiel für einen Thin Client ist ein Webbrowser. Die applikationsspezifischen Funktionen werden hier mit Hilfe von Webseiten realisiert, die ja vom Server abgerufen werden.

Mit Eclipse 3 wurde dann die Eclipse-Workbench einer Radikalkur unterzogen und die Workspace-spezifischen Teile abgetrennt. Die generischen Workbench-Teile sind dabei im Plugin org.eclipse.ui_3.0.0 verblieben, die Workspace-abhängigen Teile sind in das neue Plugin org.eclipse.ui.ide_3.0.0 abgewandert.

Eclipse bietet nun also vier Optionen für die Applikationsentwicklungen mit Java:

- Klassische Applikationsentwicklung mit AWT und Swing. Hier werden keinerlei Eclipse-Komponenten genutzt.
- Applikationsentwicklung nur mit SWT und JFace. Diese Möglichkeit sollte gewählt werden, wenn keine weiteren Features wie z.B. ein Hilfesystem benötigt werden.
- Applikationsentwicklung unter dem Eclipse IDE. Diese Möglichkeit kommt in Betracht, wenn die Applikation wie ein Studio aufgebaut sein soll, also ein geschlossenes Workspace-Konzept hat.
- Applikationsentwicklung unter der RCP.

2.1.2. Plugins und die RCP

Auch unter der RCP erfolgt die Implementierung von Anwendungsfunktionalität selbstverständlich in Form von Plugins. Wie alle Plugins muss auch ein Plugin, das eine Rich-Client-Applikation implementiert, über eine Manifest-Datei plugin.xml verfügen. Die Manifest-Datei muss allerdings Erweiterungen für die Erweiterungspunkte org.eclipse.core.runtime.applications und org.eclipse.ui. perspectives definieren. Mindestens eine Perspektive ist erforderlich, denn eine RCP-Applikation kann sich nicht auf die Ressourcenperspektive stützen, die normalerweise vom Eclipse IDE bereitgestellt wird. Die Erweiterung für den Erweiterungspunkt org.eclipse.core.runtime.applications muss im run-Element die Klasse spezifizieren, welche die Applikation repräsentiert. Hier ist ein Beispiel für die beiden Erweiterungspunkte:

```
<extension id="RcpApplication"
                    point="org.eclipse.core.runtime.applications">
  <application>
    <run class="com.RcpApplication">
    </run>
  </application>
</extension>
<extension point="org.eclipse.ui.perspectives">
  <perspective id="com.RcpPerspective">
                        name="Perspective1"
                        class="com.RcpPerspective">
  </perspective>
</extension>
```

Abbildung 1: Beispiel von Erweiterungspunkten in der RPC

2.1.3. Architektur

Die Eclipse-Architektur soll mit folgendem Schaubild und Vergleich mit der Vorgängerversion erläutert werden:

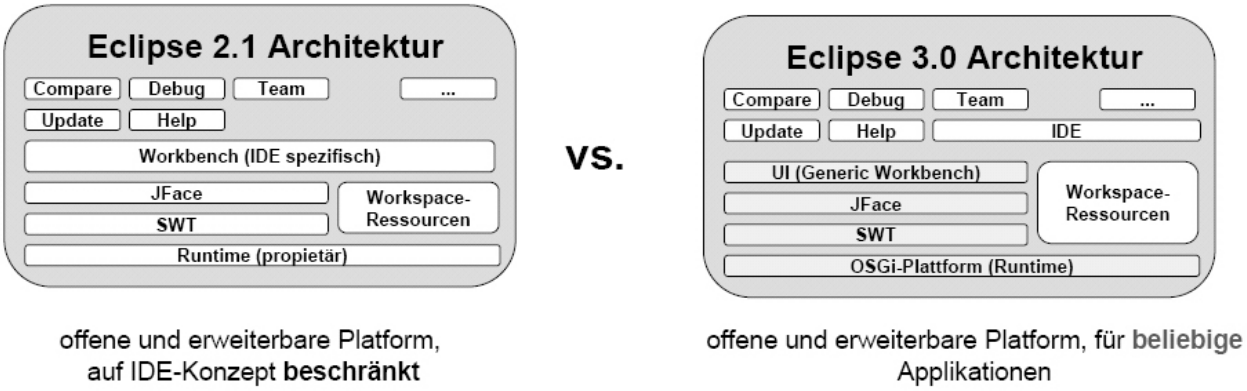


Abbildung 2: Eclipse-Architektur

2.2. Plugins für die Eclipse-Workbench entwickeln

2.2.1. Die Klasse Plugin

Die Implementierung eines Eclipse-Plugins fängt in der Regel mit der Implementierung einer Unterklasse der abstrakten Klasse Plugin an. Dabei wird man den Konstruktor implementieren. Da von dieser Klasse nur eine einzige Instanz benötigt wird (Singleton), wird man diese Instanz in Konstruktor erzeugen und in einer statischen Variable abspeichern. Über diese Instanz erhält man dann Zugriff auf die Ressourcen der Plattform. Im Konstruktor wird man im Wesentlichen eventuell benutzte Ressourcenbündel laden.

Die Superklasse stellt hauptsächlich Plugin-Methoden für die Ressourcenverwaltung auf niedriger Ebene als auch Methoden für die Verwaltung von Präferenzen zur Verfügung.

2.2.2. Die Klasse Preferences

Die Klasse implementiert einen persistenten Speicher für Präferenzen, wobei jeweils eine Instanz dieser Klasse alle Präferenzen eines Plugins enthält. Jede einzelne Präferenz besteht aus einem Namen-Wert-Paar. Der Name ist eine eindeutige, nicht-leere Zeichenkette, der Wert ist eine Variable vom Typ boolean, double, float, int, long oder String. Die Änderung von Präferenzen erzeugt Ereignisse vom Typ Preferences.PropertyChangeEvent.

2.2.3. Die Klasse Platform

Von zentraler Bedeutung ist die Klasse Platform. Sie beinhaltet nur statische Methoden. Sie verwaltet alle installierten Plugins, kümmert sich um die Zugriffsautorisierung zu Eclipse-Ressourcen und verwaltet das Eclipse-Protokoll. Insbesondere erfahren wie mittels der Methode getLocation() den Ort des Workspace-Wurzelverzeichnisses. Mit getCommandLineArgs() erfährt man die Kommandozeilen-optionen, die beim Start der Eclipse-Platform angegeben waren. Mit getPlugin() kann man ein Plugin anhand seiner Identifikation in der Plugin-Registratur suchen.

2.2.4. Die Klasse Resource

Ressourcen des Eclipse-Workspace werden durch das Interface IResource beschrieben. Dieses Interface ist zusammen mit der Klasse Resource, die davon abgeleitet ist, im Package org.eclipse.core.resources enthalten.

Da jede Ressource im Workspace auch gleichzeitig eine Ressource im Dateisystem des jeweiligen Betriebssystems ist, hat jede Ressource auch zwei Adressen. Die Adresse im Workspace findet man mit der Methode getFullPath() heraus, die Adresse im Dateisystem mit getLocation().

2.2.5. Die Konfigurationsdatei plugin.xml

Jedes Eclipse-Plug-in beschreibt in der Datei plugin.xml welche Erweiterungspunkte es bereitstellt, und in welche es sich einklinkt. Man nennt plugin.xml auch das Plug-In-Manifest. Es ist eine XML-Datei, in der alle Informationen über das Plug-In, seine Datei (runtime), die benötigten Bibliotheken (requires) und die eigenen offen gelegten Schnittstellen (extension) hinterlegt sind. Daneben beschreibt die Manifest-Datei , wie sich das Plug-in in die Platform integrieren soll. In dieser Datei wird spezifiziert, welche anderen Plug-ins benutzt werden und an welcher Stelle die Eclipse-Platform erweitert wird.

Somit ist das Wissen über Abhängigkeiten zwischen Eclipse-Plug-ins vollkommen explizit. Neben diesen expliziten Abhängigkeiten von anderen Plug-ins kann es auch noch implizite Voraussetzungen durch Plug-ins geben, die nicht durch Eclipse überprüft werden können. Hier liegt es im Verantwortungsbereich des Plugin-Entwicklers, diese Elemente bei Auslieferung ebenfalls mitzuliefern, um so die Einheit des Plug-ins zu gewährleisten.
Ein Beispiel einer plugin.xml ist in folgender Abbildung dargestellt:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
   <extension id="application"
         point="org.eclipse.core.runtime.applications">
      <application>
         <run
               class="com.Application">
         </run>
      </application>
   </extension>
   <extension id="product"
         point="org.eclipse.core.runtime.products">
      <product application="com.application"
            name="Application">
         <property
               name="aboutText"
          </product>
   </extension>
   <extension point="org.eclipse.ui.perspectives">
      <perspective
            name=" Perspective"
            class="com.Perspective"
            id="com.perspective">
      </perspective>
   </extension>
</plugin>
```

Abbildung 3: Beispiel einer plugin.xml

3. Analyse und Anforderungen

3.1. Organisation in Eclipse

Die GUI besitzt momentan im Eclipse eine eigene Perspektive, sowie ein eigenes View-Fenster. Um nun den Compiler mit Typinferenz zu benutzen muss man immer in die Typinferenz-Perspektive wechseln und hier arbeiten.

Da es sich aber bei der Benutzung des Compilers um normales Java-Programmieren handelt, nur eben mit Typinferenz, wäre es erwünscht, wie gewohnt in der Java-Perspektive arbeiten zu können. Somit wären auch zusätzliche Features wie Syntax-Highlighting und Content-Assistant möglich. Die Java-Perspektive sollte lediglich durch einen Button ergänzt werden, mit dem es möglich ist, Java-Quellecode mit dem Compiler mit Typinferenz kompilieren zu lassen.

3.2. Kompiliervorgang

Der Kompiliervorgang ist in vier Stufen unterteilt. Zuerst wird der Parsebaum anhand der Grammatik erstellt und die lexikalische Analyse durchgeführt. Anschließend wird die Typrekonstruktion gemacht, mit dem Ziel, alle unbekannten Typen anhand eines Algorithmus zu berechnen.

Nun folgt der Typersetzungsteil. Hier werden alle Typen, die nicht vom Typrekonstruktionsalgorithmus zurückberechnet werden konnten, ersetzt. Vorzugsweise wählt diese Typen dann der Endanwender selbst aus.
Zum Schluss wird der nun der entsprechende Bytecode für den kompletten Syntaxbaum generiert.

In der GUI sind diese vier Schritte getrennt dargestellt und müssen auch einzeln vom Anwender angestoßen werden. Nun sollte sich der Vorgang des Kompilierens für den Anwender jedoch als Blackbox darstellen. Somit wäre es wünschenswert, wenn eine komplette Kompilierung nur eine Aktion seitens des Users benötigen würde.

3.3. Ergebnisanzeige der Typrekonstruktion

Der Typrekonstruktionsalgorithmus berechnet Typen von Variablen, mit welchen dann im Compiler intern weitergearbeitet wird, ohne dass diese Typen irgendwo im Quellcode direkt ersichtlich sind. Nun möchte man als Anwender natürlich gerne wissen, welche Typen denn der Algorithmus für welche Variablen berechnet hat. Dies wird in der GUI über eine Listbox angezeigt.

Hierfür muss man allerdings erst in einer Combo-Box die gewünschte Klasse auswählen. Dann muss man die Art der Variablen anhand von Radio-Buttons selektieren. Anschließend muss man Methode und Variable in zwei weiteren Combo-Boxen auswählen.

Der Weg um an eine Information über den berechneten Typ einer Variablen zu kommen, gestaltet sich also sehr mühsam über eine Navigation über Combo-Boxen. Hier wäre es sicher sinnvoll die Combo-Boxen durch eine allgemeine Übersicht über alle Klassen, Methoden und Variablen sowie deren Typen zu ersetzen.

Hierfür könnte man die bereits aus der Eclipse Java-Perspektive bekannte Outline verwenden, welche nichts anderes beinhaltet, wie alle Methoden und Parameter einer Klasse. Weiterhin sollte man die Information über den berechneten Typ auch direkt im Quellcode ersehen können. Dies könnte vorzugsweise über einen Tooltip geschehen, der angezeigt wird, wenn der Anwender mit der Maus über eine Variable fährt.

3.4. Typauswahl bei nicht eindeutiger Typrekonstruktion

Falls der Typrekonstruktionsalgorithmus bei seinen Berechnungen einer Variablen zu mehreren möglichen Lösungen kommt, muss sich der Anwender für eine dieser Lösungen entscheiden.

Grundsätzlich erhält der Anwender keine Information darüber, wie viele Variablen genau der Typrekonstruktionsalgorithmus nicht berechnen konnte. Er muss sich durch die in 3.3 beschriebenen Combo-Boxen durchklicken und den Variablen einen Typ zuweisen.

Wünschenswert wäre hier eine komplette Auflistung aller Variablen mit nicht eindeutigem Typ. Idealerweise könnte dies auch durch eine direkte Markierung der entsprechenden Variable im Quellcode geschehen, die man dann beispielsweise mit einem Warn-Symbol realisieren könnte.

Die Typauswahl sollte auch direkten Bezug zur Variablen haben.
Erwünscht wäre hier eine unmittelbare Auswahl im Quellcode, zum Beispiel bei einem Klick mit der rechten Maustaste auf die Variable.

3.5. Fehlerhandling

Sollte irgendwo während des Kompiliervorgangs ein Fehler auftreten, so wird lediglich eine Textbox mit dem Inhalt der Quelltext-Exception ausgegeben. Es wird nicht zwischen verschiedenen Arten von Fehlern unterschieden und die Fehlermeldung enthält nur Debug-Informationen aus der Java Virtual Machine. Somit ist für den Anwender nicht klar, was nun beim Kompilieren falsch gelaufen ist, oder was er ändern muss, um den Fehler eventuell zu beheben.

Erstrebenswert wären detaillierte Auskünfte über den aufgetretenen Fehler. So sollte der Anwender wissen, in welcher Phase des Kompilierens der Fehler auftrat. Also zum Beispiel ein Syntaxfehler oder ein Fehler bei der Typrekontruktion.
Weiterhin sollte der genaue Ort der Fehlerursache angezeigt werden. Ideal wäre hier eine direkte rote Markierung der Fehlerstelle im Quelltext.
Die angezeigten Fehlerinformationen sollten auch mehr Aussagekraft über die genaue Fehlerart besitzen. Die Informationen könnten als Tooltip erscheinen, wenn der User mit der Maus über die rote Markierung fährt.

4. Design

4.1. Allgemein

Im Folgenden soll die grundsätzliche Funktionalität des Plugins anhand eines Bildes mit entsprechender Beschreibung der einzelnen Punkte erläutert werden. Das Bild beschreibt den Zustand nach erfolgreicher Kompilierung einer Klasse:

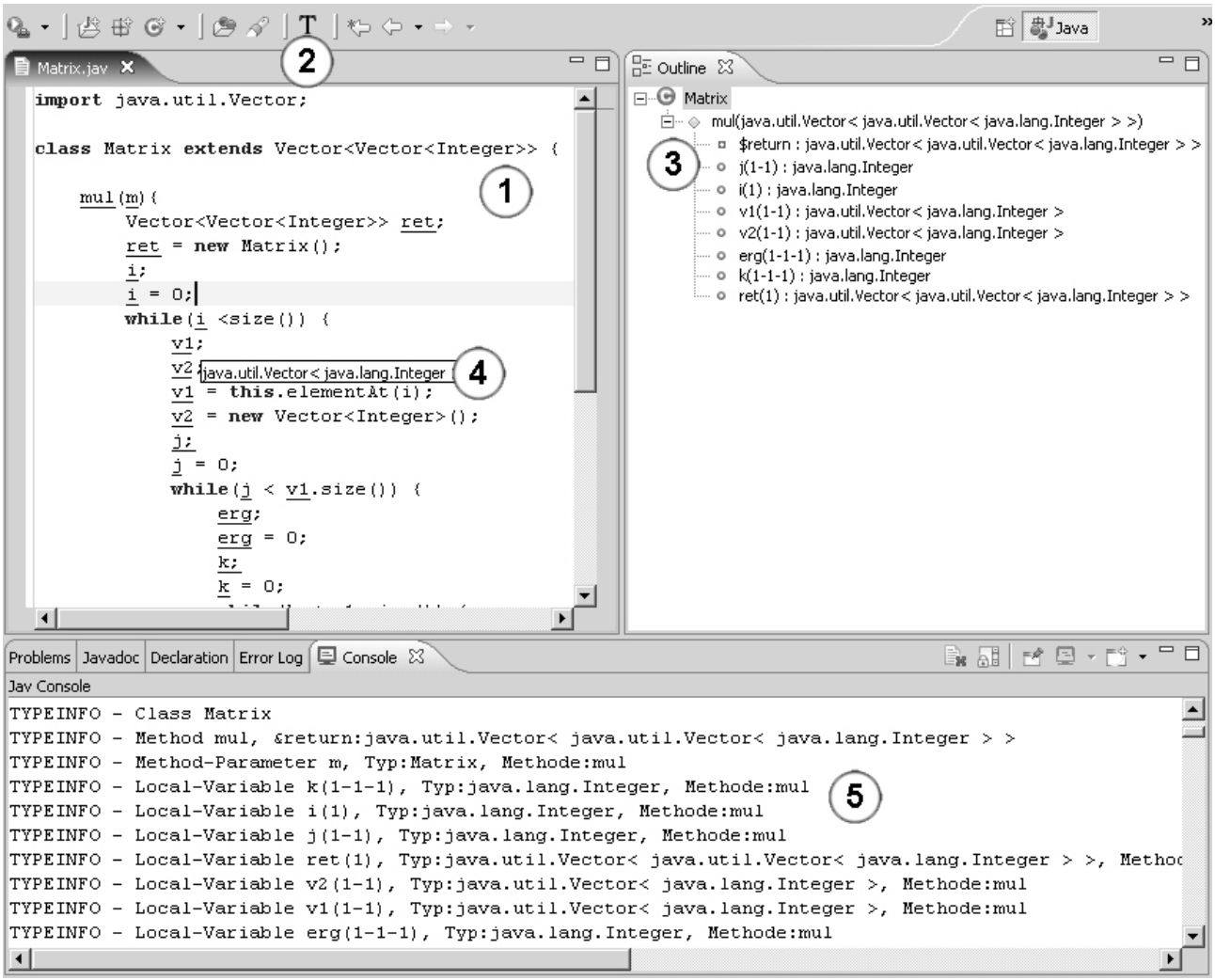


Abbildung 4: Grundsätzliche Funktionalität des Plugins

1. Hier steht der Java-Quelltext ohne Typangaben
2. Über diesen Button wird der Quelltext kompiliert und die Typen rekonstruiert
3. Übersichtsliste über alle Klassen, sowie deren Methoden und Variablen mit Angabe des berechneten Typs
4. Durch Überfahren einer Variable mit der Maus kann der berechnete Typ per Tooltip abgerufen werden
5. In der Konsole werden gefundene Variablen, sowie evtl. auch Fehler angezeigt

4.2. Error-Handling

Folgende zwei Beispiele zeigen, wie das Plugin im Fehlerfall reagiert:

1. Hier wurde eine Klammer am Ende des Statements vergessen
2. Hier schlug der Typrekonstruktionsalgorithmus fehl, da er für eine Variable zwei nicht vereinbare Angaben gefunden hat

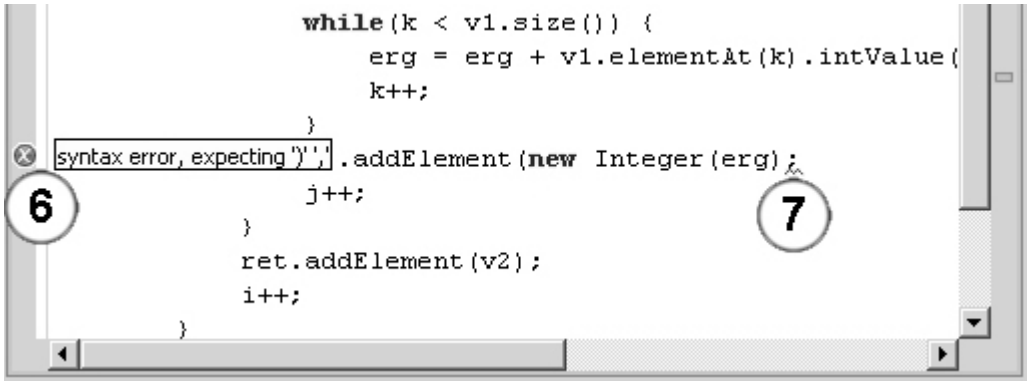


Abbildung 5: Reaktion im Fehlerfall, Beispiel 1

6 Das Error-Symbol markiert eine Zeile, in der ein Fehler aufgetreten ist
Es erscheint der Hinweis, das eine Klammer erwartet wird

7 Die genaue Fehlerstelle wird rot unterstrichen

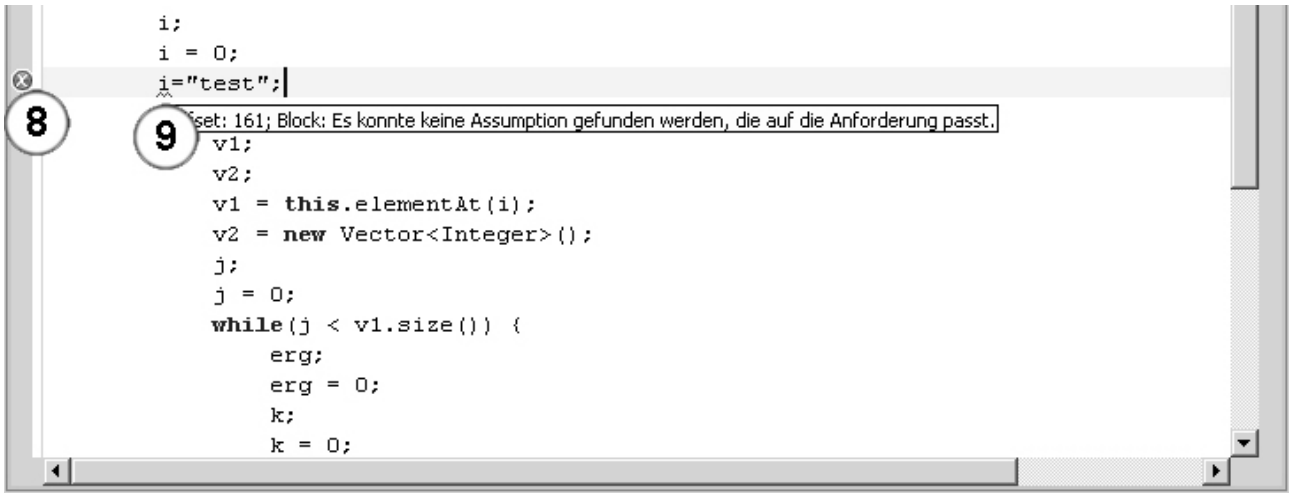


Abbildung 6: Reaktion im Fehlerfall, Beispiel 2

8 Das Error-Symbol markiert eine Zeile, in der ein Fehler aufgetreten ist

9 Die genaue Fehlerstelle wird rot unterstrichen

4.3. Sonderfunktionen

Der Compiler berechnet unbekannte Typen anhand deren Verwendung. Es gibt jedoch Fälle, in denen die Typrekonstruktion nicht eindeutig ist. Hier ist der Anwender selbst gefordert, den gewünschten Typ über ein Kontextmenü auszuwählen. Dem Anwender werden alle Typen, die für diesen Fall in Frage kommen, in einer Auswahlliste angezeigt. Ein solches Szenario ist in folgendem Bild dargestellt:

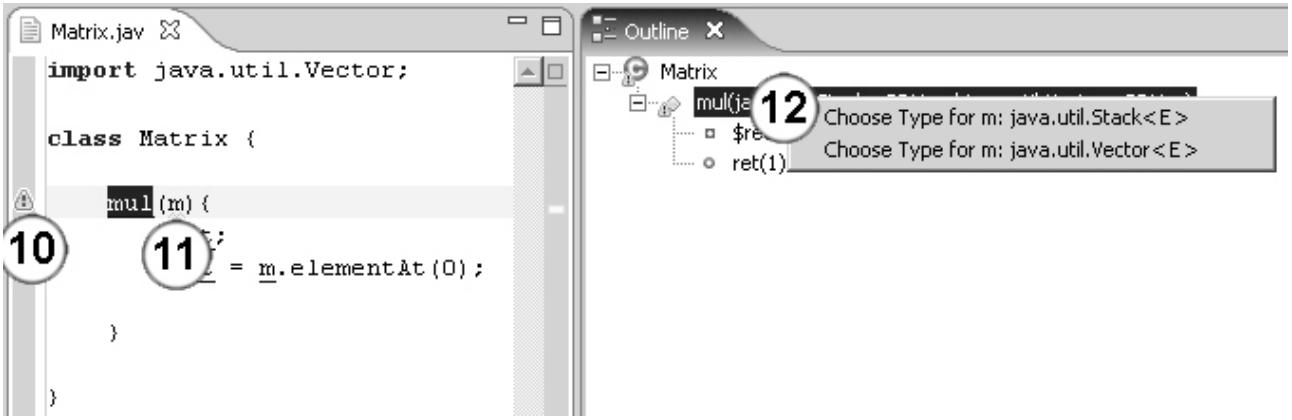


Abbildung 7: Typauswahl bei uneindeutiger Typrekonstruktion

10 Das Warn-Symbol markiert eine Zeile, in der eine Unklarheit besteht
Hier mit dem Hinweis Possible Types: java.util.Stack< E > / java.util.Vector< E >

11 Die Variable, welche keinen eindeutigen Typ besitzt, wird gelb unterstrichen

12 Über ein Kontextmenü kann der Anwender hier der Variablen einen konkreten Typ aus einer vorgegebene Auswahlliste zuordnen

4.4. Zusammenspiel der Komponenten

Folgendes Schaubild erläutert den groben Ablauf eines fehlerfreien Kompiliervorgangs und welchen Teil dabei die einzelnen Elemente des Plugins spielen:

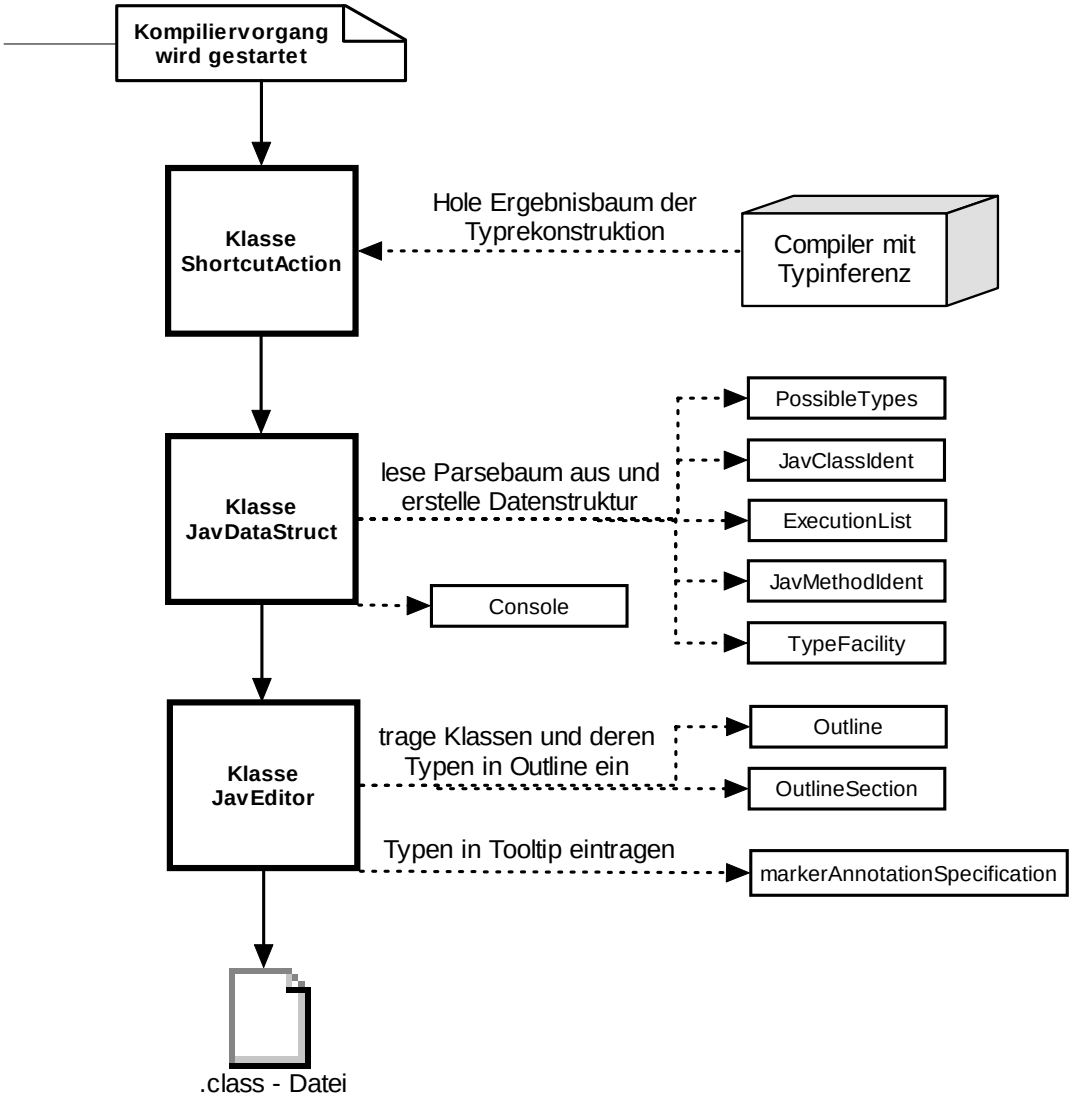


Abbildung 8: Zusammenspiel der Komponenten

5. Implementierung

5.1. Vorwort

In den folgenden Abschnitten wird die Implementierung der einzelnen Komponenten der GUI beschrieben. Hier wird lediglich auf die Teile des Plugins eingegangen, die neu hinzugekommen sind oder die sich, gegenüber der Vorgängerversion, grundlegend geändert haben.
Der erste Abschnitt befasst sich mit Änderungen am eigentlichen Compiler.

5.2. Änderungen am CompilerCore

5.2.1. Parser und Scanner

Der Scanner wurde um die Funktionalität von Kommentaren in Java-Quellcode erweitert. So gibt es nun neben dem standardmäßigen Status yylex noch commentblock für mehrzeilige und commentsingleline für einzeilige Kommentare. Wenn sich der Scanner in einem der beiden Kommentarstatus befindet, werden alle Tokens einfach weggeworfen.

Weiterhin wurde die Unterstützung des Zeichenzählens im Scanner implementiert. Dies ist erforderlich um nachher in der GUI die exakte Position von zum Beispiel einer Variablen im Quellcode zu kennen. Hierzu wurde die Klasse Token um ein Attribut Offset erweitert, welches der Scanner bei Anlage eines Tokens im Konstruktor setzt.

In der Grammatik wurde bei allen Identifiern die Methode setOffset() hinzugefügt. Sie weist einer Variablen ihre genaue Position im Quelltext zu.

5.2.2. Skeleton

Wenn der JLex-Scanner einen Token findet, der nicht in der Menge der akzeptablen Tokens ist, wirft er eine yyException und bricht ab. Diese Exception gibt lediglich eine Fehlernachricht in der alle akzeptablen Tokens stehen auf der Konsole aus. Leider kann man daraus nicht schlussfolgern, welcher Token jetzt genau diesen Fehler verursacht hat. Die ist hauptsächlich für die GUI-Darstellung wichtig, denn hier möchte man ja den fehlerhaften Codeteil zum Beispiel rot markieren.

Also wurde im skeleton-file die Klasse yyException um ein Attribut Token erweitert, welches den fehlerverursachenden Token enthalten soll. Diesen Token kann man nun bei Abfangen der Exception im catch-Teil auswerten. Weiterhin wurde die Methode yyError() umgebaut, so dass sie den Fehlertext als Exception-Message abspeichert und nicht nur auf der Konsole ausgibt. Somit kann dieser auch in der GUI angezeigt werden.

5.2.3. Offsets einer Typannahme

Die Klasse CTypeAssumtion wurde um ein Attribut Offset erweitert, wo die Position der Deklaration der Variablen steht, sowie ein Attribut OffsetVector<Integer> wo alle Positionen enthalten sind, in denen die Typannahme verwendet wird. Ein Beispiel:

```
1: simple(){
2:   i;
3:   i=5+5;
4:   return i;
5: }
```

Abbildung 9: Code-Beispiel für Typinferenz

In Zeile 2 wird vom Parser wie in 5.2.1 beschrieben eine TypeAssumtion angelegt und das Offset gesetzt. In Zeile 3 und 4 stößt der Parser zwar auf eine Variable i, hat aber zu diesem Zeitpunkt noch keinerlei Bezug zu der Assumption zu der die Variable eigentlich gehört. Er findet nur einen Identifier vor und wirft diesen als Token nach oben zur nächsten Regel. Ziel ist aber, in dem Attribut OffsetVector<Integer> einer Assumption alle Vorkommen dieser Variablen zu speichern.
Es ist deshalb notwendig, nach Abschluss des Parsevorgangs den kompletten Parsebaum noch einmal zu durchwandern und für alle verwendeten Variablen deren entsprechende Offsets zu setzen. Hierfür wurde in der Klasse Class eine statische Methode addOffsetsToAssumption() implementiert. Aufgerufen wird diese Methode in der TRProg()-Methode der Klasse Class für Methoden und Objektvariablen, sowie in der TRStatement()-Methode einer LocalVarDecl, also einer lokalen Variablenanlage.

Der Methode addOffsetsToAssumption() werden die Assumption, sowie der Block der Deklaration dieser Assumption übergeben. Die Methode durchsucht dann den ganzen Block nach weiteren Vorkommen dieser Variablen. Bei Klassenvariablen wird die ganze Klasse dursucht. Der Algorithmus läuft rekursiv ab, indem er, wenn er einen Block findet wieder sich selbst aufruft. Findet er ein Statement oder eine Expression ruft er die jeweils zugehörige wieder rekursive Hilfsmethode addOffsetsToStatement() oder addOffsetsToExpression() auf.

5.2.4. IItemWithOffset

Das Interface IItemWithOffset beinhaltet die Methoden getOffset() und getVariableLength() mit welchen Stelle und Länge einer Variablen abgefragt werden können. Da die Klassen Statement, UsedID und Type dieses implementieren ist sichergestellt, das bei allen Variablen ein Offsetzugriff erfolgen kann.

5.3. Plugin.xml

Die plugin.xml wurde durch einen Erweiterungspunkt für das Actionset des Compilers ergänzt, so dass der Button Kompilieren in der Eclipse Java-Perspektive erscheint. Erweiterungspunkte werden wie in 2.1.2 beschrieben implementiert.

Hinzugefügt wurde auch ein Eintrag für die Outline, sodass der Zugriff auf diese seitens des Plugins möglich ist.
Nun gibt es noch 3 Einträge, für Anntotations. Diese sind für Markierungen direkt im Quelltext verantwortlich:

- Highlight-Annotation: Zuständig für die Anzeige eines normalen Tooltips. Wurde so umgesetzt, dass der Typ einer Variablen angezeigt wird, wenn man mit der Maus darüber fährt.
- Error-Annotation: Zeigt an einer bestimmten Textzeile ein rotes Error-Symbol an und unterstreicht die genaue Textstelle rot.
- Warning-Annotation: Zeigt an einer bestimmten Textzeile ein gelbes Warn-Symbol an und unterstreicht die genaue Textstelle gelb.

Als Beispiel folgt der Erweiterungspunkt der Highlight-Annotation:

```
<extension
   point="org.eclipse.ui.editors.markerAnnotationSpecification">
   <specification
     annotationType="typeInferenz.highlightannotation"
     verticalRulerPreferenceKey="linked.focus.verticalruler"
     textPreferenceKey="linked.focus.text"
     colorPreferenceKey="linked.focus.color"
     highlightPreferenceKey="linked.focus.highlight"
     textPreferenceValue="true"
     textStylePreferenceValue="UNDERLINE"
     overviewRulerPreferenceKey="linked.focus.overviewruler"
     presentationLayer="4"
     highlightPreferenceValue="false"
     label="Typeinference"
     symbolicIcon="info"
     colorPreferenceValue="0,0,128"
     verticalRulerPreferenceValue="false"
     overviewRulerPreferenceValue="false"
     textStylePreferenceKey="linked.focus.text.style">
    </specification>
   </extension>
```

Abbildung 10: Implementierung des Erweiterungspunktes Highlight-Annotation

Des weiteren wurde die Teile für die Typinferenz-Perspektive und View entfernt.

5.4. ShortCutAction

Stellt die Aktion dar, die ausgeführt wird, wenn der Anwender einen Kompiliervorgang anstößt. Die Methode run() führt nacheinander die Schritte Parsen, Typerekonstruktion, Execution und Bytecodegeneration aus. Im Fehlerfall wird jeweils eine entsprechende Routine abgearbeitet.

Hier der Teil der Typerekonstruktion:

```
Statics.print("Type reconstruction...",Statics.CompilerStatusInfo);
try {
   Vector<CTypeReconstructionResult> resultSet =
  m_compilerApi.typeReconstruction();

  PluginMediator.getInstance().setReconstructionResult(resultSet);

  PluginMediator PM = PluginMediator.getInstance();
  PluginMediator.getInstance().getActiveEditor().temp_status=2;
             Statics.print("OK",Statics.CompilerStatusInfo);
             if(PM.getActiveEditor().makeTooltips(true))
             {
                  return;
             }
        }
catch (CTypeReconstructionException e)
  {

PluginMediator.getInstance().getActiveEditor().temp_status=1;
for(IItemWithOffset itemOffset : e.getHighestProblemSources())
             {

   PluginMediator.getInstance().getActiveEditor().makeError(e.getMessa
ge(),itemOffset.getOffset(),itemOffset.getVariableLength());
             }
  }
```

Abbildung 11: Starten der Typrekonstruktion

5.5. ContextAction

Die Klasse ContextAction repräsentiert die Aktion, die ausgeführt wird, wenn der Anwender einer Variable einen konkreten Typ zuweist, da dieser nicht eindeutig berechnet werden konnte. Jede Aktion enthält den zugehörigen Klassennamen der Variable, evtl. Methodenname, sowie Variablenname und den Typ der Variablen, der gesetzt wird, wenn man diese Aktion ausführt.

5.6. Outline

Die Outline enthält eine Übersichtsliste aller Klassen, Methoden und Variablen, sowie deren Typen. Sie ist quasi ein Container für mehrere OutlineSections, die im folgendem Abschnitt beschrieben werden.

Bei jeder Aktualisierung oder Änderung der Outline werden alle Einträge durchlaufen und das jeweilige Kontextmenü für jeden Eintrag neu gesetzt. Das geschieht über die selectionChanged()-Methode durch folgenden Code:

```
public void selectionChanged(SelectionChangedEvent event)
  {
     try{
     menuMgr.removeAll();
     menuMgr.add(new
     Separator(IWorkbenchActionConstants.MB_ADDITIONS));
     menuMgr.add(new Separator(IWorkbenchActionConstants.MB_ADDITIONS)
     IStructuredSelection selection = (IStructuredSelection)
    event.getSelection();
    OutlineSection outlinesection =
    (OutlineSection)selection.getFirstElement();
    if(outlinesection.m_ActionVector!=null){
      for(ContextAction contextaction :
        outlinesection.m_ActionVector)
        {
          contextaction.setText("Choose Type for
          "+contextaction.getVariablenname()+":
          "+contextaction.getTyp());
          contextaction.ActualOutlineSection=outlinesection;
          menuMgr.add(contextaction);
      }}}
    catch(NullPointerException NPE){}
  }
```

Abbildung 12: Aktualisierung der Outline

5.7. OutlineSection

Diese Komponente stellt einen Gliederungseintrag in der Outline dar. Sie besitzt die gängigen Methoden zur Verwaltung einer Baumstruktur, also add(), remove(), getParent, setParent, size() etc.

Erweitert wurde sie durch die Methoden getIconNumber() und setIconNumber() zum setzen der Icons für zum Beispiel eine Klasse oder Interface.
Wichtig ist auch hier die Methode getOffset(), welche angibt, an welcher Stelle im Quellcode sich dieser Eintrag befindet.
Weiterhin implementiert ist das Attribut m_ActionVector. In diesem werden sämtliche ContextActions (siehe 5.5) gesammelt, die für diesen Eintrag gültig sind. Gemeint sind die Variablen für die mehrere Typen zur Auswahl stehen. Diese zur Auswahl stehenden Typen werden in den Container m_ActionVector getan und können somit bei Aufruf des Kontextmenüs einer OutlineSection angezeigt und aufgerufen werden.

5.8. Datastruct

Die Komponente wurde durch Umstrukturierungen wesentlich verändert. Der Grundansatz gegenüber einer getrennten Datenstruktur von GUI und Compiler wurde jedoch beibehalten, da sich nur mit einem solchem Aufbau einer Datenstruktur die Anforderungen an die GUI erfüllen lassen(vgl. Studienarbeit Markus Melzer).

Folgende Erweiterungen des Kerncompilers wurden zusätzlich noch ergänzt:

- überladene Methoden
- Interfaces
- Umstellung der BaseTypeAssumption auf die JRE
- generische Methoden

5.9. JavEditor

5.9.1. Einleitung

Der Editor ist zentrales Element für die grafische Darstellung. Er implementiert die Methoden für Fehler- und Warnanzeige, sowie die Tooltips, Typauswahl und Hervorhebung der Variablen.

Zudem verwaltet er den Status des aktuellen Kompiliervorgangs. Hier gibt es fünf Zustände:

1. Anfangszustand: Die Klasse ist noch völlig unkompiliert.
2. Fehlerzustand: Während des Kompilierens trat ein Fehler auf und muss vom Anwender behoben werden.
3. Warnzustand: Es existieren nicht eindeutige Typen, welchen vom Anwender ein Typ zugewiesen werden muss.
4. Bereit zur Codegenerierung: Alle Typen wurden korrekt berechnet oder ersetzt. Die Bytecodegenerierung kann erfolgen.
5. Fertig: Die Klasse wurde erfolgreich kompiliert.

In den folgenden Abschnitten werden die wichtigsten Methoden des Editors, sowie deren Aufgabe beschrieben.

5.9.2. makeTooltip() bzw. makeWarning() und makeError()

Diese Methoden implementieren das Setzen eines Tooltips oder einer Warning.
Der in einem Editor dargestellte Inhalt besteht aus zwei Teilen: Neben dem Text, auf den über das IDocument-Interface zugegriffen wird, enthält das Annotationsmodell eine Menge von Annotationen, die dem Text zugeordnet sind. Eine Annotation ist ein Stück zusätzliche Information zu einem Text an einer bestimmten Position.

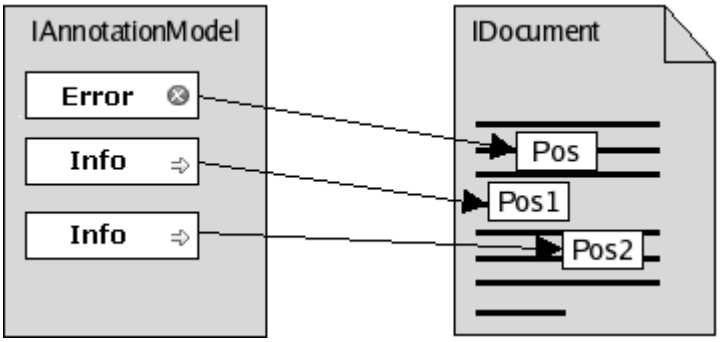


Abbildung 13: Dokument und Annotationsmodell

Eine Position repräsentiert eine Stelle (Offset und Länge) in einem Dokument. Ist eine Position bei einem IDocument registriert, wird sie bei Dokumentänderungen automatisch aktualisiert. Eine Position wird beispielsweise nach hinten verschoben, wenn vor ihr Text eingefügt wurde.

Der Editor stellt Annotationen je nach Typ unterschiedlich dar. Sie können als Ikonen am linken Editorrand, als Farbmarkierungen in der Übersichtsleiste (rechts des Editors) oder direkt im Text als Wellenlinie oder durch eine spezielle Hintergrundfarbe angezeigt werden.

Um über Selektionsänderungen (einschließlich Cursorbewegungen) informiert zu werden, registriert sich der Updater beim IPostSelectionProvider des Editors. Dieser Provider sendet seine SelectionEvents erst aus, wenn der Benutzer die Selektion für eine kurze Zeit unverändert gelassen hat. Dies hat den Vorteil, dass schnell wechselnde Selektionen nicht empfangen werden. Erhält der Updater eine Selektionsänderung, so extrahiert er das aktuelle Wort aus dem Text.

Auszug aus makeTooltip():

```
IAnnotationModel myIAnnotationModel =
myISourceViewer.getAnnotationModel();
        Annotation annotation= new
Annotation("typeInferenz.highlightannotation", false, DerTooltip);
        vectorTooltips.addElement(annotation);
        Position position= new Position(von, bis);
        myIAnnotationModel.addAnnotation(annotation, position);
...
```

Abbildung 14: Listing von makeTooltip

5.9.3. refreshTooltips()

Diese Methode wird aufgerufen, nachdem der Typrekonstruktionsalgorithmus durchgelaufen ist. Vorraussetzung ist , dass hier und vorher beim Parsen kein Fehler auftrat. Alle Variablen in allen Klassen und Interfaces werden durch Schleifen durchlaufen. Für jede Variable wird ein Tooltip über die Methode makeTooltip() mit entsprechendem Typ erzeugt.

Bei den Schleifendurchgängen für die Tooltips wird gleichzeitig auch die Outline mit entsprechenden Typinformationen erzeugt. Für jede Variable wird hier gleich überprüft, ob mehrere Typen zur Auswahl stehen und dann ein Kontextmenü in der entsprechenden OutlineSection(siehe 5.7) erzeugt. In diesem Fall wird auch eine Warning an der entsprechenden Stelle gesetzt und der Compilerstatus auf 3 gesetzt(siehe 5.9.1).

5.9.4. getAdapter

Das Eclipse-Rahmenwerk bietet eine Gliederungsansicht an, welche immer die zum aktiven Editor gehörende Gliederung anzeigt, genannt Outline. Die Verbindung zwischen einem Editor und der Gliederungsansicht wird über einen Adapter im Editor hergestellt. Wenn ein neuer Editor geöffnet wird, fragt die Gliederungsansicht mit editor.getAdapter(IContentOutlinePage.class) nach einer Gliederungsseite und zeigt diese an.

5.9.5. editorContextMenuAboutToShow

Methode wird aufgerufen, wenn der Benutzer mit der rechten Maustaste auf eine Quelltextstelle klickt mit dem Ziel eine Typauswahl für eine Variable treffen zu wollen.

Um herauszufinden, welche Variable der Anwender nun mit seinem Klick ansprechen will, wird eine weitere Hilfsmethode checkMultipleTypes()
benutzt. Diese durchsucht eine Hashmap, in der alle Variablen und deren Offsets gespeichert sind. Diese Offsets werden nun mit dem Offset verglichen, an welche Stelle der Anwender im Quelltext geklickt hat. Wenn dieses Offset im Bereich einer Variablen liegt, also Startoffset bis Startoffset plus Anzahl Buchstaben der Variablen, dann wird überprüft, ob diese Variable nicht eindeutig ist und eine Menge von Typen für diese Variable in Frage kommen. In diesem Fall werden diese Typen als ContextAction(siehe 5.5) zu dem System-Kontextmenü von Eclipse hinzugefügt.

5.10. Console

Auf der Konsole werden dem Anwender hilfreiche Information zum Plugin angezeigt. Um die Ausgabe anschaulich zu gliedern wurde hier für jede Art von Ausgabe eine Methode implementiert, ähnlich dem log4j.

Für folgende Ausgabetypen gibt es jeweils eine Methode.
Die Ausgaben der jeweiligen Informationen sind natürlich auch an den entsprechenden Stellen im Quellcode implementiert:

- MainInfo: Gibt allgemeine Statusangaben zum Plugin aus, z.B. Plugin gestartet.
- ExceptionErrors: Zuständig für die Ausgabe eines Fehlers.
- CompilerStatusInfo: Gibt an, in welcher Phase sich der Compiler gerade befindet, bzw. welche er gerade betritt oder verlässt.
- VariablenAnlage: Für jede Variable im Parsebaum wird hier eine Meldung mit Name und entsprechendem Typ der Variablen angezeigt.
- DoubleTypeWarning: Falls es nicht eindeutige Variablen gibt, werden diese hierdurch ausgegeben.

6. Schlussbetrachtung

6.1. Auswirkungen der Studienarbeit

Größte Verbesserungen der GUI wurden in den Punkten Fehlerhandling und Benutzerfreundlichkeit gemacht, was ja auch im Mittelpunkt der Studienarbeit stehen sollte. So werden Parse- oder Typinferenzfehler direkt im Quellcode angezeigt. Weiterhin erscheint ein Warn-Symbol bei nicht eindeutigen Typen. Die Auswahl dieser Typen kann ebenso direkt im Quellcode erfolgen.

Der Grundaufbau der GUI wurde komplett geändert. So besitzt das Plugin keine eigene Perspektive und View mehr, sondern wird über die Java-Perspektive in Eclipse gesteuert, welche lediglich durch einen Button zum Kompilieren ergänzt wird. Weiterhin wurde der Kompiliervorgang zusammengefasst, so dass für den Anwender nur noch eine Aktion erforderlich ist.

Die grundsätzliche Anzeige der rekonstruierten Typen wurde wesentlich übersichtlicher gestaltet. So werden alle Klassen, Methoden und Variablen , sowie deren Typen in einer Liste, genannt Outline, am rechten Bildrand angezeigt. Weiterhin kann jederzeit der Typ einer Variablen direkt im Quellcode per Tooltip abgerufen werden.

6.2. Persönliche Erkenntnisse

„Ein Compiler ist eine Applikation, welche Programme, die manchmal laufen, in Programme verwandelt, die niemals laufen.“

Die Studienarbeit um den Compilerbau stellte sich komplexer heraus, als zunächst angenommen. Hauptgrund hierfür ist, dass um eine ordentliche GUI-Umsetzung des Compilers erstellen zu können, ein doch sehr tiefer Einstieg in den Compiler Kern erforderlich ist. Eine Einarbeitung in zwei Projekte, GUI und Compiler, war also erforderlich.

Mit zunehmendem Verständnis des Compilers und der Java-Generics konnte schlussendlich dann aber sogar Sinn und Zweck von Java mit Typinferenz erkannt werden, was mir zu Beginn noch Schwierigkeiten machte.

Ingesamt war das Forschungsprojekt Typinferenz in jedem Fall eine Erfahrung, die man im Nachhinein nicht missen will.

6.3. Ausblick

Ein wichtiger Punkt, der in der Studienarbeit nicht realisiert wurde, ist die Speicherung einmal gemachter Typauswahlen. So muss man, wenn es in einer Klasse nicht eindeutige Typen gibt (z.B Vector und Stack), nach jedem Kompiliervorgang diese erneut bestimmen. Besser wäre es, wenn die Auswahl z.B. in einem xml-File gespeichert werden würde und dieses beim Kompilieren einer Klasse automatisch ausgelesen würde.

Nice to have wäre auch noch, wenn der Editor ein Content-Assistant-Menü bereitstellen würde. Gemeint ist das aus Eclipse bekannte Menü, wenn man von einer Klasse beim Programmieren durch betätigen der Taste „.“ alle Methoden dieser Klasse angezeigt bekommt.

Die Icons in der Outline könnten auch noch erweitert werden, so dass auch zum Beispiel Attribute wie public und private grafisch zum Ausdruck kommen.

7. Quellenverzeichnis

7.1. Internet

- http://www.wikipedia.org
- http://www.eclipse.org
- http://www.sun.com
- http://www.cetic.be/internal.php3?id_article=225
- http://www.koders.com (javaeditor.java)
- http://www.sigs.de/publications/js/2003/06/frenzel_JS_06_03.pdf
- http://www.tutorials.de/forum/c-c-c-java-tutorials/186300-eclipse-editor-plugin-mit-sourcecode-highlightning-1-a.html
- http://www.eclipse-magazin.de/itr/online_artikel/psecom,id,647,nodeid,230.html

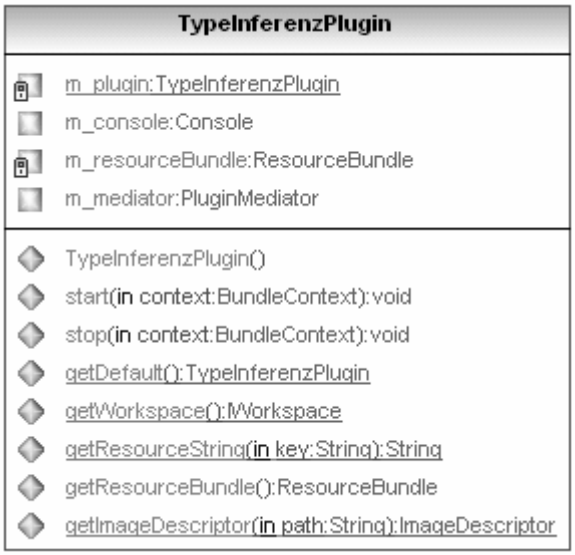
7.2. Literatur

- [Bäu05] Jörg Bäuerle. Typinferenz in Java
- [Mel05] Markus Melzer. Integration der Java-Typinferenz in eine Programmierumgebung
- [Haa04] Markus Haas. Weiterentwicklung der Java-Codegenerierung zur Ausführung von parametrisierten Datentypen
- [Ott04] Thomas Ott. Typinferenz in Java.
- [Dau05] Berthold Daum. Java-Entwicklung mit Eclipse 3
- [Bur05] Ed Burnette. Eclipse IDE kurz und gut.

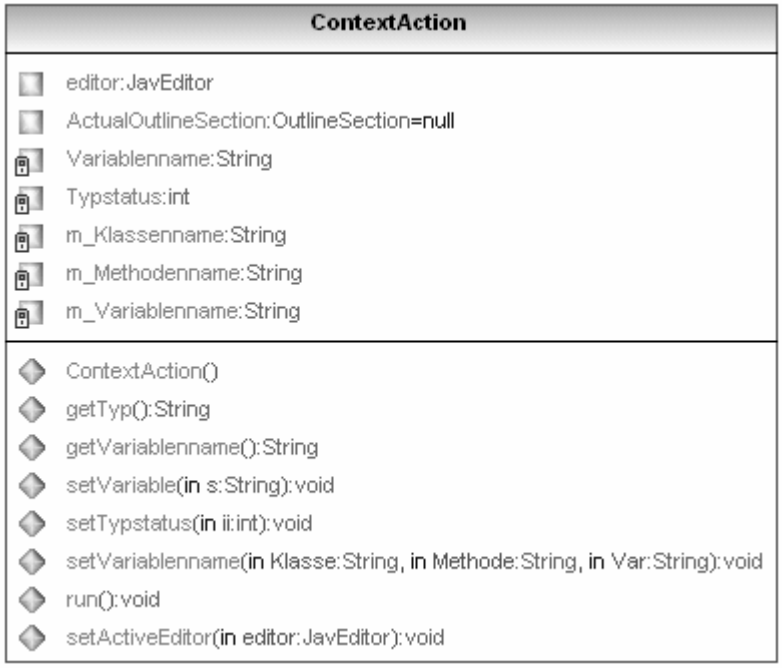
Anlagen

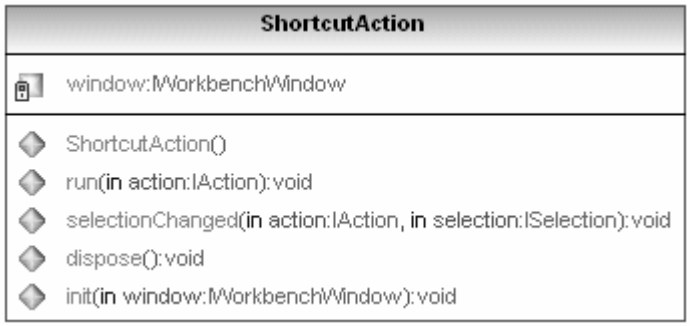
A Klassendiagramme

A.1 typeInferenz

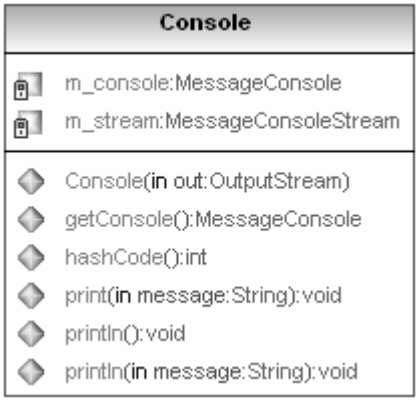

TypeInferenzPlugin
m_plugin:TypeInferenzPlugin
m_console:Console
m_resourceBundle:ResourceBundle
m_mediator:PluginMediator
TypeInferenzPlugin()
start(in context:BundleContext):void
stop(in context:BundleContext):void
getDefault():TypeInferenzPlugin
getWorkspace():IWorkspace
getResourceString(in key:String):String
getResourceBundle():ResourceBundle
getImageDescriptor(in path:String):ImageDescriptor

A.2 actions

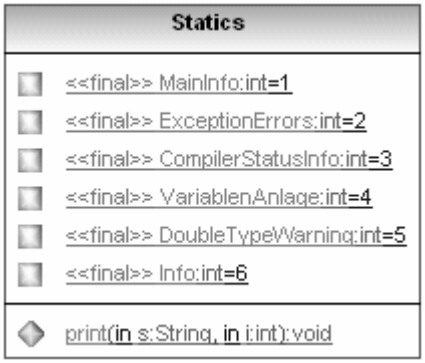

ContextAction
editor:JavEditor
ActualOutlineSection:OutlineSection=null
Variablenname:String
Typstatus:int
m_Klassenname:String
m_Methodenname:String
m_Variablenname:String
ContextAction()
getTyp():String
getVariablenname():String
setVariable(in s:String):void
setTypstatus(in ii:int):void
setVariablenname(in Klasse:String, in Methode:String, in Var:String):void
run():void
setActiveEditor(in editor:JavEditor):void


ShortcutAction
window:IWorkbenchWindow
ShortcutAction()
run(in action:IAction):void
selectionChanged(in action:IAction, in selection:ISelection):void
dispose():void
init(in window:IWorkbenchWindow):void

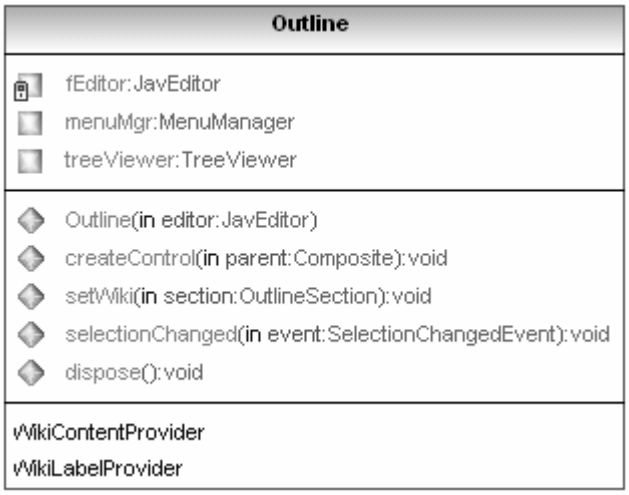
A.3 console



A.4 helpers



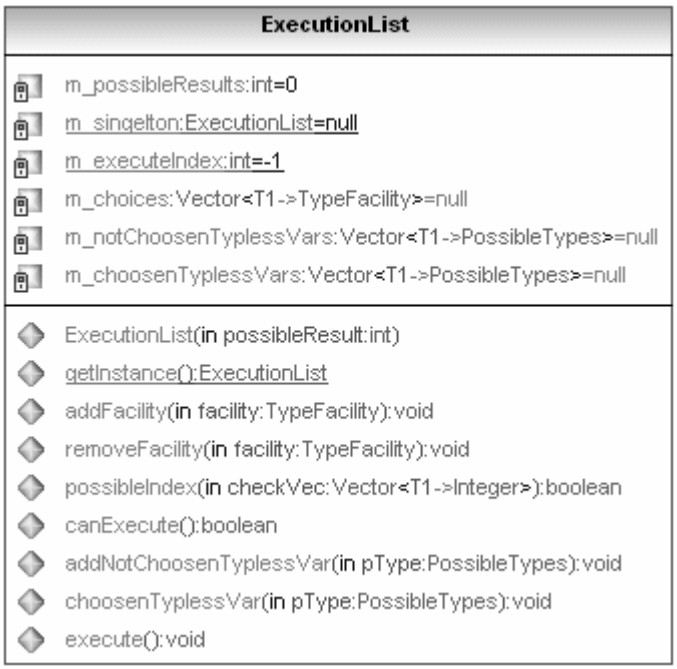
A.5 editors

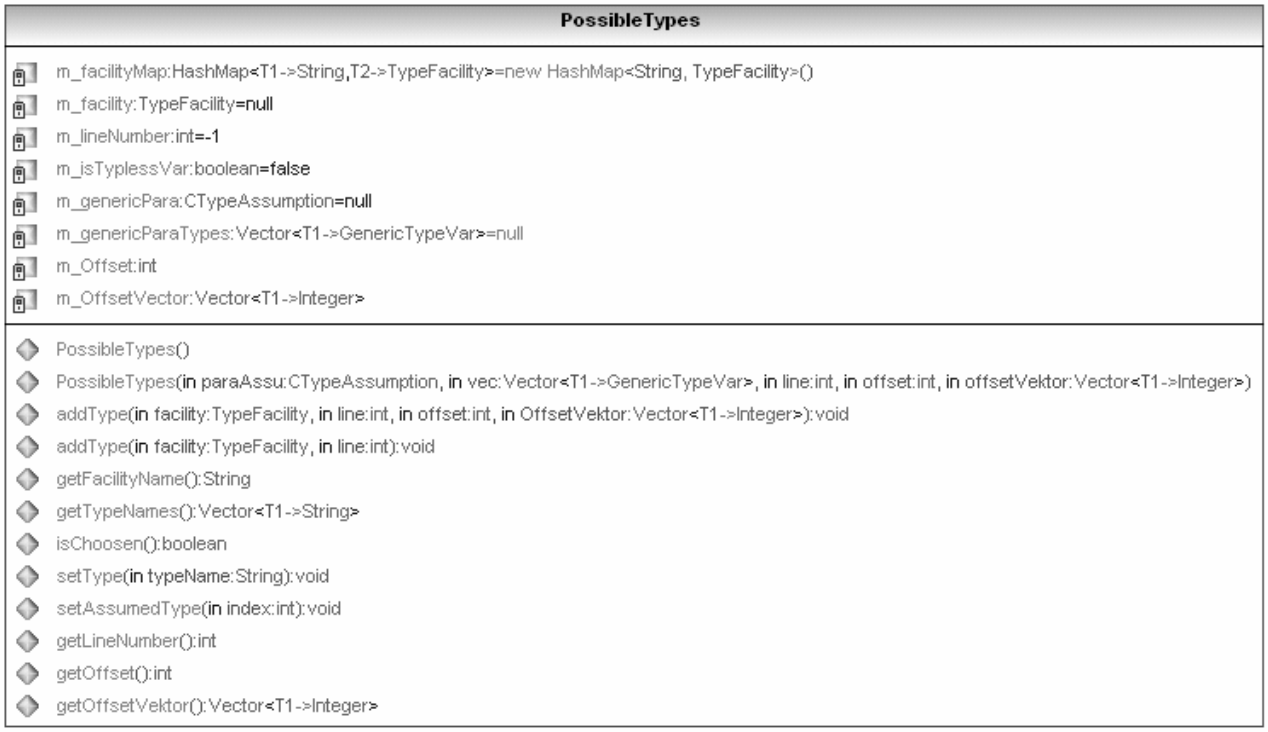


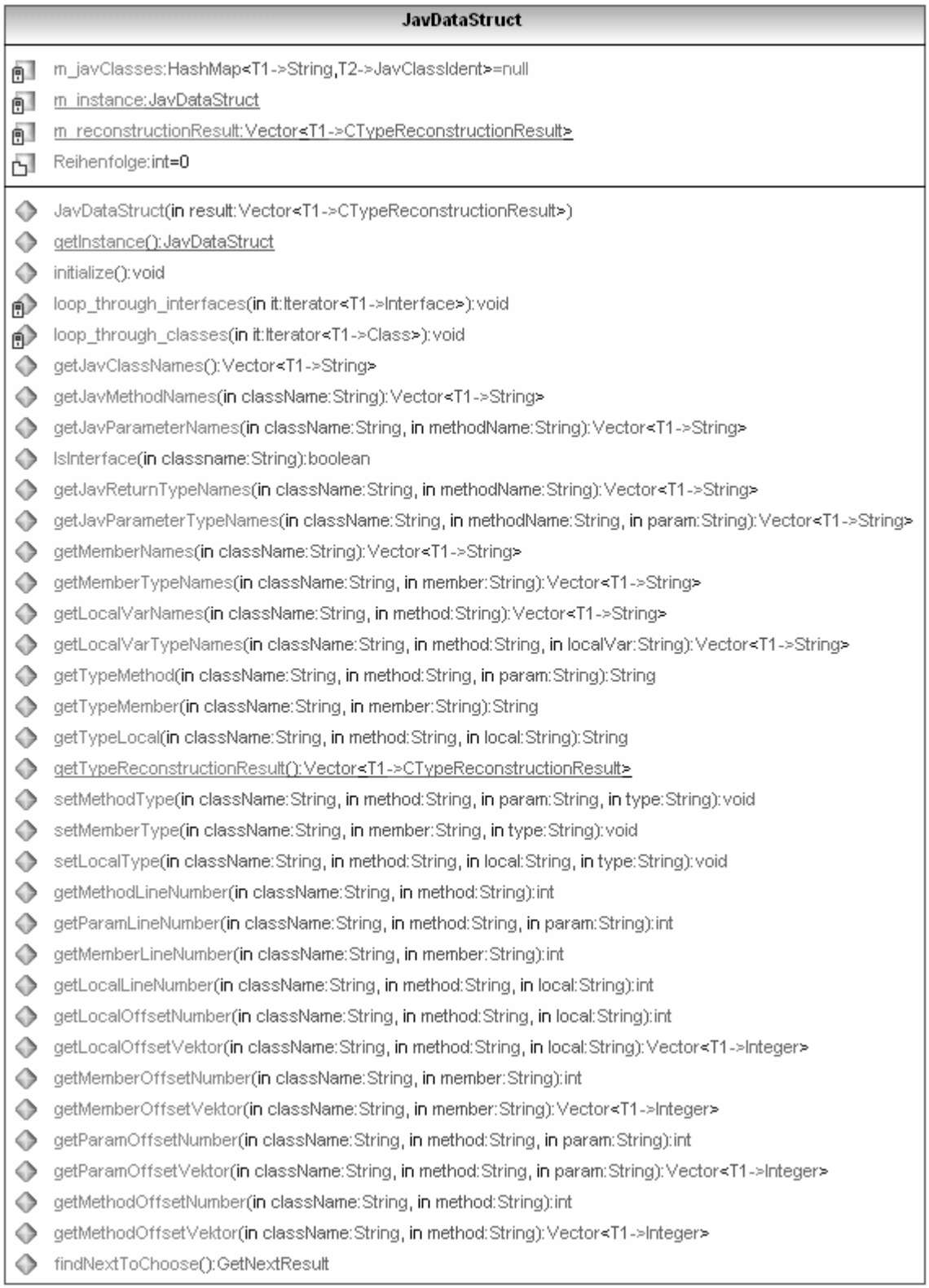

OutlineSection
NO_CHILDREN:OutlineSection[*]=new OutlineSection[0]
fParent:OutlineSection
m_ActionVector:Vector<T1->ContextAction>=new Vector<ContextAction>()
fName:String
fOffset:int
fLength:int
fheaderLevel:int
recipeElements:ArrayList<T1->OutlineSection>
IconNumber:int
OutlineSection(in parent:OutlineSection, in name:String, in level:int, in offset:int, in length:int, in ikonnummer:int)
setIconNumber(in ii:int):void
getIconNumber():int
getChildren():Object[*]
getName():String
setName(in s:String):void
getOffset():int
getLength():int
add(in index:int, in element:OutlineSection):void
add(in o:OutlineSection):boolean
get(in index:int):OutlineSection
size():int
setLength(in length:int):void
setoffset(in offset:int):void
getHeaderLevel():int
setHeaderLevel(in fheaderLevel:int):void
getParent():OutlineSection
setParent(in parent:OutlineSection):void

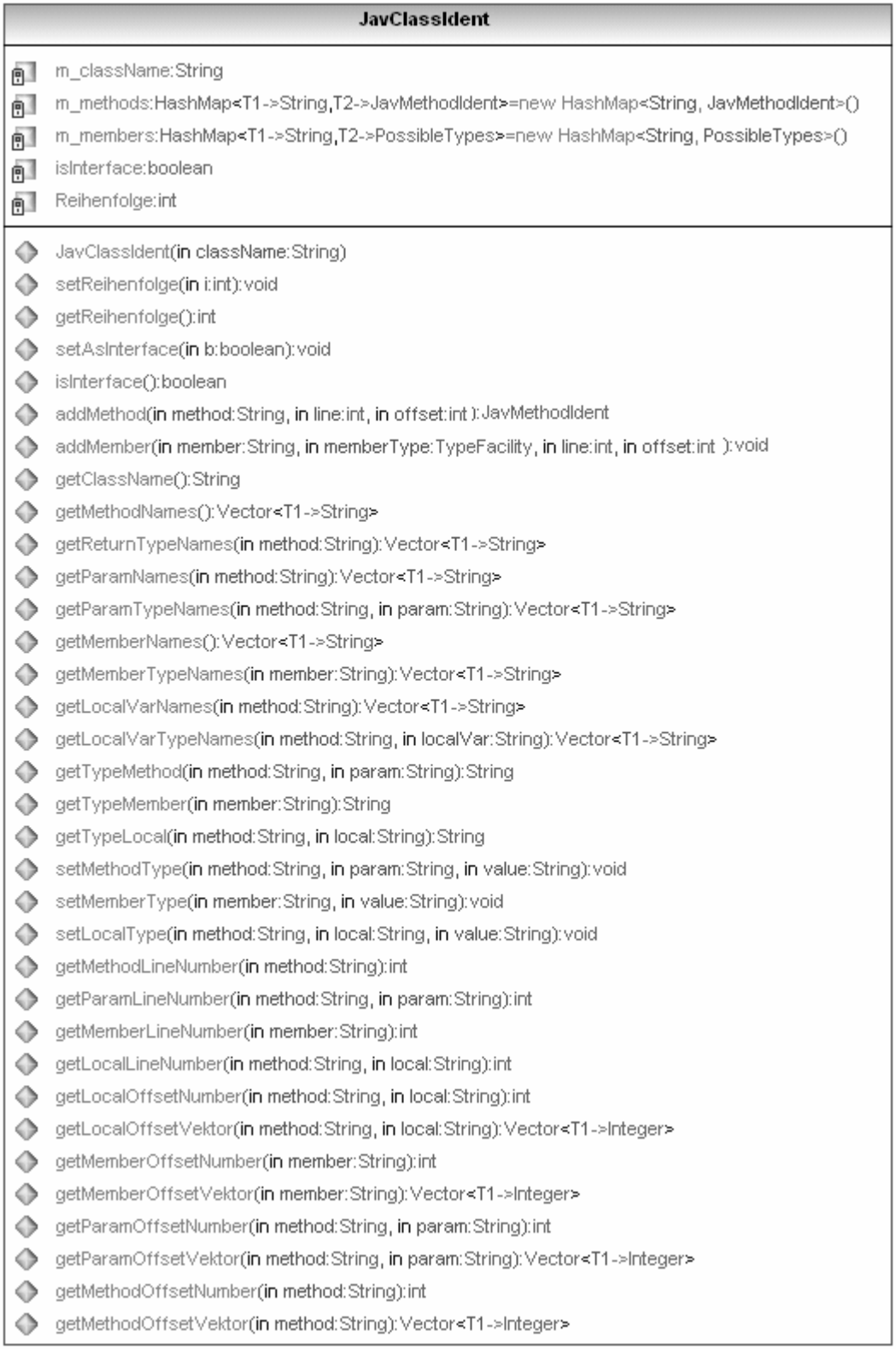

JavEditor
m_mediator:PluginMediator
m_dirtyFlag:boolean
fProjectionSupport:ProjectionSupport
myISourceViewer:ISourceViewer
vectorTooltips:Vector<T1->Annotation>=new Vector<Annotation>()
fSection:OutlineSection
fOutlinePage:Outline
alleOffsets:Hashtable<T1->Integer,T2->String>=new Hashtable<Integer,String>()
m_Typnamen:Vector<T1->String>=new Vector<String>()
m_Klassenname:String
m_Methodenname:String
m_Variablenname:String
temp_status:int=0
CommandButton:IAction
JavEditor()
dispose():void
initializeEditor():void
isDirty():boolean
getContent():String
createSourceViewer(in parent:Composite, in ruler:IVerticalRuler, in styles:int):ISourceViewer
createPartControl(in parent:Composite):void
deleteAllTooltips():void
makeTooltip(in DerTooltip:String, in von:int, in bis:int):void
makeWarning(in DieWarning:String, in von:int, in bis:int):void
makeError(in DerError:String, in von:int, in bis:int):void
fuelleHashtable(in Variable:String, in Offset:int, in Laenge:int):void
getRealMethodname(in s:String):String
getClassOffset(in isinterface:boolean, in cname:String):int
getClassLength(in Klassenname:String):int
makeTooltips(in macheDieOutline:boolean):boolean
macheOutline(in macheDieOutline:boolean):void
getAdapter(in required:Class):Object
outlinePageClosed():void
editorContextMenuAboutToShow(in parentMenu:IMenuManager):void
checkMultipleTypes(in Variable:String, in Offeset:int):int
checkOffset(in offsetDerVariablen:int, in offsetVektorDerVariablen:Vector<T1->Integer>):boolean
createActions():void

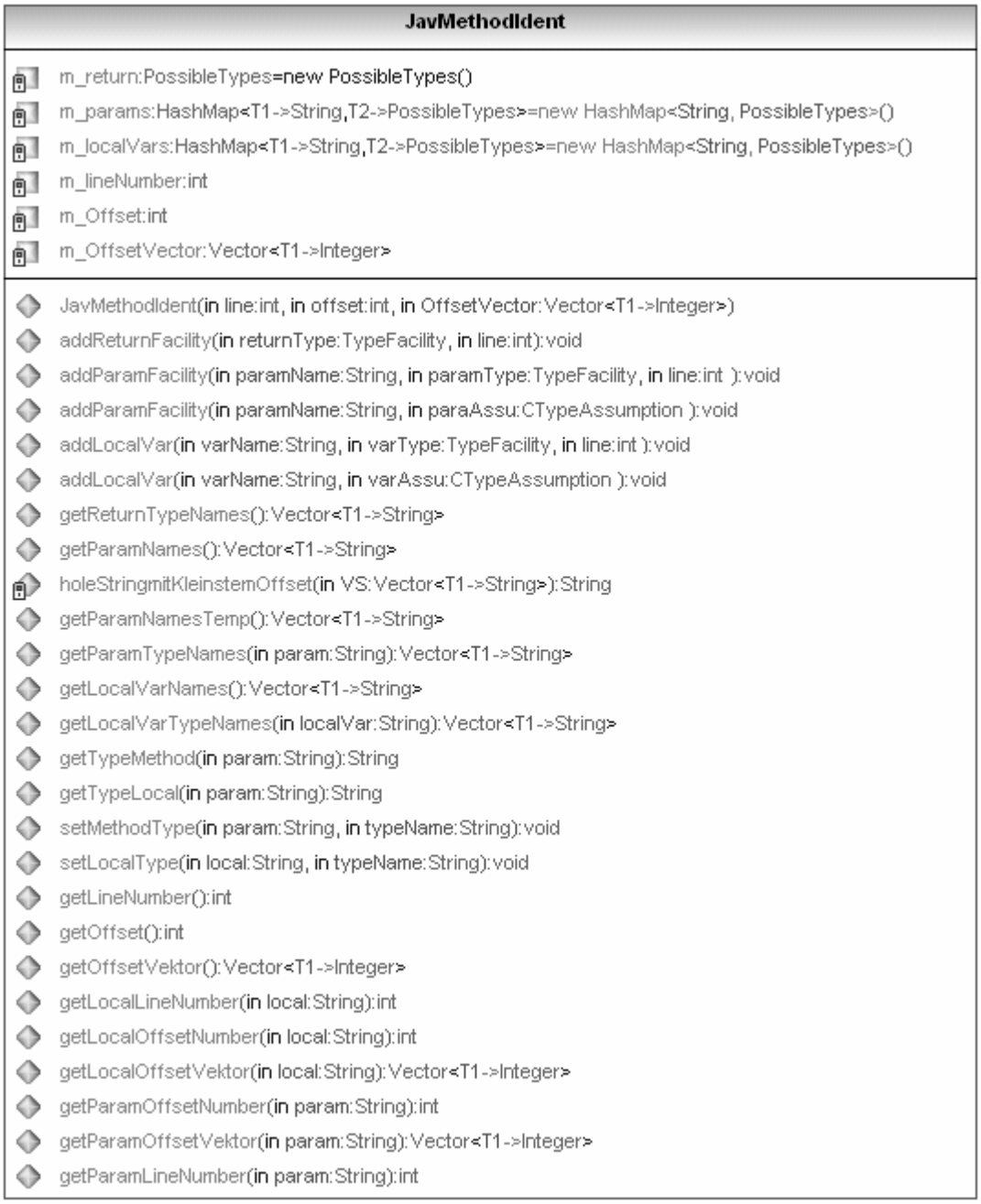
A.6 datastruct

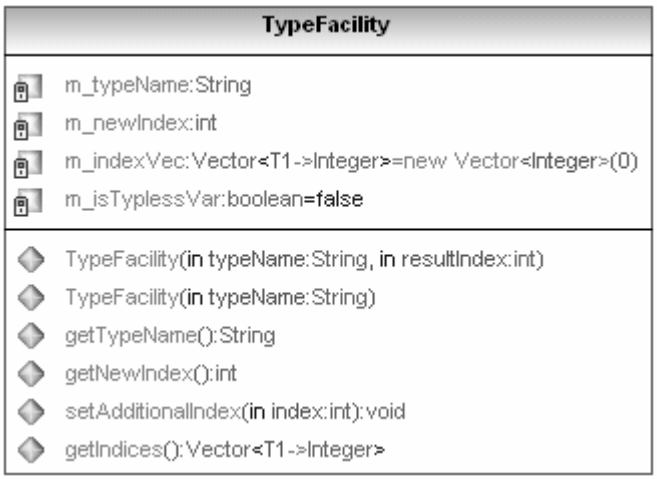


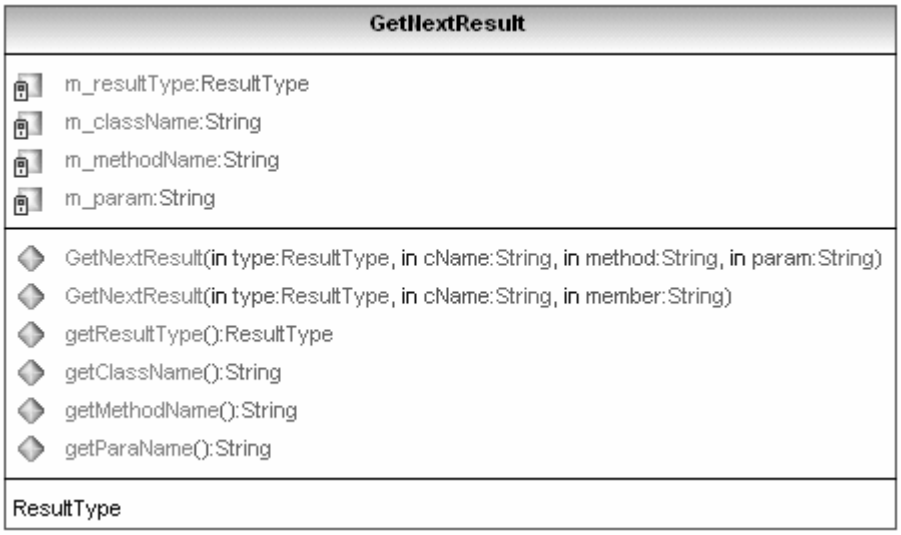



JavDataStruct
m_javClasses:HashMap<T1->String,T2->JavClassIdent>=null
m_instance:JavDataStruct
m_reconstructionResult:Vector<T1->CTypeReconstructionResult>
Reihenfolge:int=0
JavDataStruct(in result:Vector<T1->CTypeReconstructionResult>)
getInstance():JavDataStruct
initialize():void
loop_through_interfaces(in it:Iterator<T1->Interface>):void
loop_through_classes(in it:Iterator<T1->Class>):void
getJavClassNames():Vector<T1->String>
getJavMethodNames(in className:String):Vector<T1->String>
getJavParameterNames(in className:String, in methodName:String):Vector<T1->String>
IsInterface(in classname:String):boolean
getJavReturnTypeNames(in className:String, in methodName:String):Vector<T1->String>
getJavParameterTypeNames(in className:String, in methodName:String, in param:String):Vector<T1->String>
getMemberNames(in className:String):Vector<T1->String>
getMemberTypeNames(in className:String, in member:String):Vector<T1->String>
getLocalVarNames(in className:String, in method:String):Vector<T1->String>
getLocalVarTypeNames(in className:String, in method:String, in localVar:String):Vector<T1->String>
getTypeMethod(in className:String, in method:String, in param:String):String
getTypeMember(in className:String, in member:String):String
getTypeLocal(in className:String, in method:String, in local:String):String
getTypeReconstructionResult():Vector<T1->CTypeReconstructionResult>
setMethodType(in className:String, in method:String, in param:String, in type:String):void
setMemberType(in className:String, in member:String, in type:String):void
setLocalType(in className:String, in method:String, in local:String, in type:String):void
getMethodLineNumber(in className:String, in method:String):int
getParamLineNumber(in className:String, in method:String, in param:String):int
getMemberLineNumber(in className:String, in member:String):int
getLocalLineNumber(in className:String, in method:String, in local:String):int
getLocalOffsetNumber(in className:String, in method:String, in local:String):int
getLocalOffsetVektor(in className:String, in method:String, in local:String):Vector<T1->Integer>
getMemberOffsetNumber(in className:String, in member:String):int
getMemberOffsetVektor(in className:String, in member:String):Vector<T1->Integer>
getParamOffsetNumber(in className:String, in method:String, in param:String):int
getParamOffsetVektor(in className:String, in method:String, in param:String):Vector<T1->Integer>
getMethodOffsetNumber(in className:String, in method:String):int
getMethodOffsetVektor(in className:String, in method:String):Vector<T1->Integer>
findNextToChoose():GetNextResult


JavClassIdent
m_className:String
m_methods:HashMap<T1->String,T2->JavMethodIdent>=new HashMap<String, JavMethodIdent>()
m_members:HashMap<T1->String,T2->PossibleTypes>=new HashMap<String, PossibleTypes>()
isInterface:boolean
Reihenfolge:int
JavClassIdent(in className:String)
setReihenfolge(in i:int):void
getReihenfolge():int
setAsInterface(in b:boolean):void
isInterface():boolean
addMethod(in method:String, in line:int, in offset:int):JavMethodIdent
addMember(in member:String, in memberType:TypeFacility, in line:int, in offset:int):void
getClassName():String
getMethodNames():Vector<T1->String>
getReturnTypeNames(in method:String):Vector<T1->String>
getParamNames(in method:String):Vector<T1->String>
getParamTypeNames(in method:String, in param:String):Vector<T1->String>
getMemberNames():Vector<T1->String>
getMemberTypeNames(in member:String):Vector<T1->String>
getLocalVarNames(in method:String):Vector<T1->String>
getLocalVarTypeNames(in method:String, in localVar:String):Vector<T1->String>
getTypeMethod(in method:String, in param:String):String
getTypeMember(in member:String):String
getTypeLocal(in method:String, in local:String):String
setMethodType(in method:String, in param:String, in value:String):void
setMemberType(in member:String, in value:String):void
setLocalType(in method:String, in local:String, in value:String):void
getMethodLineNumber(in method:String):int
getParamLineNumber(in method:String, in param:String):int
getMemberLineNumber(in member:String):int
getLocalLineNumber(in method:String, in local:String):int
getLocalOffsetNumber(in method:String, in local:String):int
getLocalOffsetVektor(in method:String, in local:String):Vector<T1->Integer>
getMemberOffsetNumber(in member:String):int
getMemberOffsetVektor(in member:String):Vector<T1->Integer>
getParamOffsetNumber(in method:String, in param:String):int
getParamOffsetVektor(in method:String, in param:String):Vector<T1->Integer>
getMethodOffsetNumber(in method:String):int
getMethodOffsetVektor(in method:String):Vector<T1->Integer>


JavMethodIdent
m_return:PossibleTypes=new PossibleTypes()
m_params:HashMap<T1->String,T2->PossibleTypes>=new HashMap<String, PossibleTypes>()
m_localVars:HashMap<T1->String,T2->PossibleTypes>=new HashMap<String, PossibleTypes>()
m_lineNumber:int
m_Offset:int
m_OffsetVector:Vector<T1->Integer>
JavMethodIdent(in line:int, in offset:int, in OffsetVector:Vector<T1->Integer>)
addReturnFacility(in returnType:TypeFacility, in line:int):void
addParamFacility(in paramName:String, in paramType:TypeFacility, in line:int):void
addParamFacility(in paramName:String, in paraAssu:CTypeAssumption):void
addLocalVar(in varName:String, in varType:TypeFacility, in line:int):void
addLocalVar(in varName:String, in varAssu:CTypeAssumption):void
getReturnTypeNames():Vector<T1->String>
getParamNames():Vector<T1->String>
holeStringmitKleinstemOffset(in VS:Vector<T1->String>):String
getParamNamesTemp():Vector<T1->String>
getParamTypeNames(in param:String):Vector<T1->String>
getLocalVarNames():Vector<T1->String>
getLocalVarTypeNames(in localVar:String):Vector<T1->String>
getTypeMethod(in param:String):String
getTypeLocal(in param:String):String
setMethodType(in param:String, in typeName:String):void
setLocalType(in local:String, in typeName:String):void
getLineNumber():int
getOffset():int
getOffsetVektor():Vector<T1->Integer>
getLocalLineNumber(in local:String):int
getLocalOffsetNumber(in local:String):int
getLocalOffsetVektor(in local:String):Vector<T1->Integer>
getParamOffsetNumber(in param:String):int
getParamOffsetVektor(in param:String):Vector<T1->Integer>
getParamLineNumber(in param:String):int


TypeFacility
m_typeName:String
m_newIndex:int
m_indexVec:Vector<T1->Integer>=new Vector<Integer>(0)
m_isTyplessVar:boolean=false
TypeFacility(in typeName:String, in resultIndex:int)
TypeFacility(in typeName:String)
getTypeName():String
getNewIndex():int
setAdditionalIndex(in index:int):void
getIndices():Vector<T1->Integer>


GetNextResult
m_resultType:ResultType
m_className:String
m_methodName:String
m_param:String
GetNextResult(in type:ResultType, in cName:String, in method:String, in param:String)
GetNextResult(in type:ResultType, in cName:String, in member:String)
getResultType():ResultType
getClassName():String
getMethodName():String
getParaName():String
ResultType