BERUFSAKADEMIE STUTTGART
University of Cooperative Education

Erweiterung des Eclipse Plugins zur Unterstützung der Arbeit mit dem Typinferenz Compiler

Studienarbeit

Eine Projekt-Dokumentation eingereicht für den Abschluss zum

Diplom-Informatiker (Berufsakademie)

im Studiengang Angewandte Informatik
an der Berufsakademie Stuttgart

von

Thorsten Hake und Christian Stresing

Juni 2008

Zeitraum	5. und 6. Semester
Kurs	TIT05AIA
Firma	IBM Deutschland GmbH Stuttgart
Betreuer	Timo Holzherr nero AG
Projektverantwortlicher	Prof. Dr. Plümicke Berufsakademie Horb

Selbständigkeitserklärung

Wir versichern hiermit, dass wir die vorliegende Arbeit mit dem Thema

Erweiterung des Eclipse Plugins zur Unterstützung der Arbeit mit demTypinferenz Compiler

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet haben.

Ort: _____

Datum: _____

Unterschriften: _____

Zusammenfassung

In der Programmiersprache Java ist dem Programmierer vorgegeben für eine Variable immer einen Typ zu definieren, sie ist typsicher. In Skriptsprachen wie zum Beispiel PHP ist dies nicht nötig, da es sich hierbei um eine typunsichere Sprache handelt.

Um in Java trotzdem die Möglichkeit zu haben keinen Typen angeben zu müssen, und trotzdem die durch Java garantierte Typsicherheit zu haben, wurde der Typinferenz-Algorithmus von Prof. Dr. Martin Plümicke entwickelt. Dieser wurde im Rahmen einiger Projekte in einem eigenen Compiler implementiert. Für eine benutzerfreundliche Verwendung des Compilers wurde darüber hinaus ein Eclipse Plugin geschrieben.

Die Weiterentwicklung und Verbesserung dieses Eclipse Plugins ist das Thema dieser Studienarbeit. Es wird dabei auf die Migration und Integration in der Eclipse Version 3.4 ebenso wie auf das Hinzufügen einer neuen Visualisierung von Typeinschränkungen eingegangen.

Abstract

In the programming language Java it is mandatory to declare types for variables. It is therefore called a type-safe programming language. Other languages such as PHP don't enforce to give variables' types. These languages are called type-unsafe programming languages.

Enabling Java programmers to declare variables without type declarations was one of the goals of Prof. Dr. Martin Plümicke's type-inference algorithm. It is done in a way that still ensures the type-safety of the Java programming language. This algorithm has been integrated in a compiler by several project. A eclipse plugin has been developed in order to have a user-friendly access to the features of the compiler.

The enhancement and further development of this eclipse plugin is subject of this student research project. Migration and Integration into a the new Eclipse Version 3.4 will be covered just as the new feature of visualization of type limitations.

Danksagung

An dieser Stelle möchten wir uns besonders bei Herrn Prof. Dr. Martin Plümicke bedanken, der uns bei Fragestellungen rund um die Struktur und Intention des Eclipse Plugins immer eine große Hilfe war. Die Diskussionen an verschiedenen Nachmittagen an der BA würden wir nicht missen wollen.

In diesem Sinne möchten wir uns auch bei unserem Mentor Herrn Timo Holzherr bedanken. Er war stets ein guter Ansprechpartner, wenn es um die Studienarbeit ging.

Beiden möchten wir für das Testen des Plugins danken. Die dadurch entdeckten Fehler haben uns bei der Stabilisierung des Plugins geholfen.

Inhaltsverzeichnis

1 Einführung 1
1.1 Typinferenz 1
1.2 Projektgeschichte 1
1.3 Aufgabenstellung 2
1.4 Struktur 2
2 Erläuterungen 4
2.1 Eclipse 4
2.2 Plugins 5
2.3 Extension Points 6
2.4 Features und Update-Sites 6
3 Migration von Eclipse 3.1 zu Eclipse 3.4 8
3.1 Wesentliche Framework Änderungen 8
3.1.1 Eclipse 3.1 zu 3.2 8
3.1.2 Eclipse 3.2 zu 3.3 9
3.1.3 Eclipse 3.3 zu 3.4 10
3.2 Analyse 11
3.2.1 API Änderungen 11
3.2.2 Funktionale Änderungen 12
3.3 Durchführung 13
3.4 Weitere Integration in das Eclipse Framework 16
4 Integration von log4j 18
4.1 Intention 18
4.2 Änderungen in der Architektur 19
5 Visualisierung von Einschränkungen bei der Typinferenz 21
5.1 Offset Berechnung 22
5.2 Ersetzungsstrategien 22
5.2.1 Lokale Variablen 22
5.2.2 Methodenparameter 23
5.2.3 Methodenrückgabewerte 23
5.3 Architekturänderungen 24
5.3.1 Änderung des Ablaufs des Typrekonstruktions-Aufrufs 25
6 Fazit 27
6.1 Rückblick 27
6.2 Ausblick 27
7 Referenzen 28
8 Illustrationsverzeichnis 29
9 Anhang 30
9.1 Beschreibung der Installation/Aktualisierung/Deinstallation des Eclipse Plugins 30
9.2 Überblick über Plugins, Features und Update-Sites 35

1 Einführung

1.1 Typinferenz

Der Begriff Typinferenz bezeichnet das Bestimmen eines vorher noch nicht bestimmten Typs durch die Analyse von beeinflussenden Faktoren. Der in [PLU05] definierte Typinferenz-Algorithmus kann dazu benutzt werden, in der Programmiersprache JAVA unbekannte Typen zu bestimmen. Hierdurch muss bei der Programmierung nicht jede Variable typisiert werden, da der Typ bei einigen Variablen vom Compiler selber inferiert werden kann.

1.2 Projektgeschichte

Der in dieser Studienarbeit im Mittelpunkt stehende Compiler wurde im Rahmen der Vorlesungen Informatik 4 und Software-Engineering vom Jahrgang TIT2000 an der Berufsakademie Horb entwickelt.

Im Rahmen von darauf aufsetzenden Studienarbeiten wurde dann mit den Studienarbeiten [REI05] und [HAA04] der Compiler um generische Typen erweitert. Der in [PLU05] beschriebene Typinferenz-Algorithmus wurde dann in der Studienarbeit [OTT04] und in [BAE05] im Compiler integriert. Gebundene generische Typen, Wildcards, Interfaces und Packages wurden in [HOL06] und [LUE07] dem Algorithmus hinzugefügt. Die Studienarbeiten [BUR07] und [SCH07] hatten zudem das Ziel, den Compiler Java 5.0 kompatibel zu machen.

Das Eclipse Plugin als komfortable Benutzeroberfläche zur Verwendung des Typinferenz Compilers wurde schon in zwei Studienarbeiten behandelt.

Die erste Studienarbeit zum Thema Eclipse Plugin [MEL05] beschäftigte sich mit der Integration des Compilers in Eclipse. Dies beinhaltete die Erstellung einer neuen Datenstruktur, welche es möglich machte, die vom Compiler gelieferten Informationen in einem geeigneten Format grafisch darzustellen. Eine GUI wurde auch in dieser Studienarbeit beschrieben, allerdings wurde die GUI während der zweiten Studienarbeit [HOM06] erneuert. Diese Studienarbeit beschäftigte sich mit dem Ziel das Plugin benutzerfreundlicher zu gestallten. Hierdurch kamen

viele nützliche Funktionen, wie zum Beispiel das Compiler Feedback, hinzu.

1.3 Aufgabenstellung

Die Aufgabenstellung für diesen Teil der Studienarbeit bestand daraus, das bereits existierende Eclipse Version 3.1 Plugin zur Nutzung des Type-Inferenz Compilers in Eclipse zu verbessern und auf die aktuelle Eclipse 3.3 Version zu migrieren.

Des Weiteren sollen durch das Plugin vorgenommene Beschränkungen bei der Typinferenz auch für den Benutzer kenntlich gemacht werden.

Vorhandene Fehler des Plugins sollten zudem in Absprache mit Herrn Plümicke ermittelt und ggf. korrigiert werden, um das Plugin stabiler zu machen.

1.4 Struktur

Auf einen rein theoretischen Teil wurde in dieser Studienarbeit verzichtet, da die Kapitel teilweise zu unterschiedliche Aspekte des Projektes beschreiben. Die für das jeweilige Kapitel notwendigen, theoretischen Grundlagen werden wenn nötig innerhalb des jeweiligen Kapitels erläutert.

Nachfolgend werden die für den Umgang mit Eclipse notwendigen Begriffe erklärt und die grundlegende Architektur von Eclipse dargestellt.

Auf die Erklärungen aufbauend wird im Kapitel 3 die Migration von Eclipse 3.1 auf Eclipse 3.3 respektive 3.4 beschrieben. Dies beinhaltet auch eine Analyse des bisher bestehenden Plugins auf eine Aufwärtskompatibilität. Außerdem wird in diesem Kapitel die Integration in das Eclipse Plugin Management Framework erläutert.

Informationen über die Integration von log4j als Logging Lösung sowohl für das Eclipse Plugin als auch für den Typinferenz Compiler können im Kapitel 4 gefunden werden.

Der funktionalen Erweiterung des Plugins im Form der Visualisierung der Typeinschränkung durch den Benutzer widmet sich das darauf folgende Kapitel. Die verschiedenen Fälle der Ersetzung und dabei auftretende Probleme werden hierbei erläutert.

Den Abschluss bildet der Rückblick über die in dieser Arbeit geschaffenen Verbesserungen und ein Ausblick auf noch anstehende Arbeiten und Anregungen.

Gefundene Fehler im Plugin selber werden in dieser schriftlichen Ausarbeitung keinen Platz einnehmen. Das Plugin wurde jedoch im Rahmen der praktischen Studienarbeit an vielen Stellen von bestehenden Fehlern befreit. Diese Fehler waren allerdings zu klein, als dass sie erwähnenswert seien.

2 Erläuterungen

2.1 Eclipse

Eclipse ist ein auf OSGi basierendes Anwendungsframework, welches durch Plugins genutzt und erweitert werden kann. Eine Sammlung von Plugins, die das Eclipse Framework als Basis nutzen, ist die Eclipse-IDE (Integrated Development Environment). Diese IDE bietet eine graphische Benutzeroberfläche zum Programmieren in verschiedenen Programmiersprachen. Sie integriert alle dafür nötigen Programme in einer einheitlichen Oberfläche (im Eclipse Umfeld Workbench genannt). Die Trennung zwischen dem Eclipse Framework und der Eclipse IDE ist seit der Version 3.0 unabdingbar, da das Eclipse Framework in Form einer Rich Client Platform (RCP) auch zu ganz anderen Zwecken genutzt werden kann. So bauen inzwischen viele kommerzielle Programme auf die Rich Client Platform auf [RCPCA].

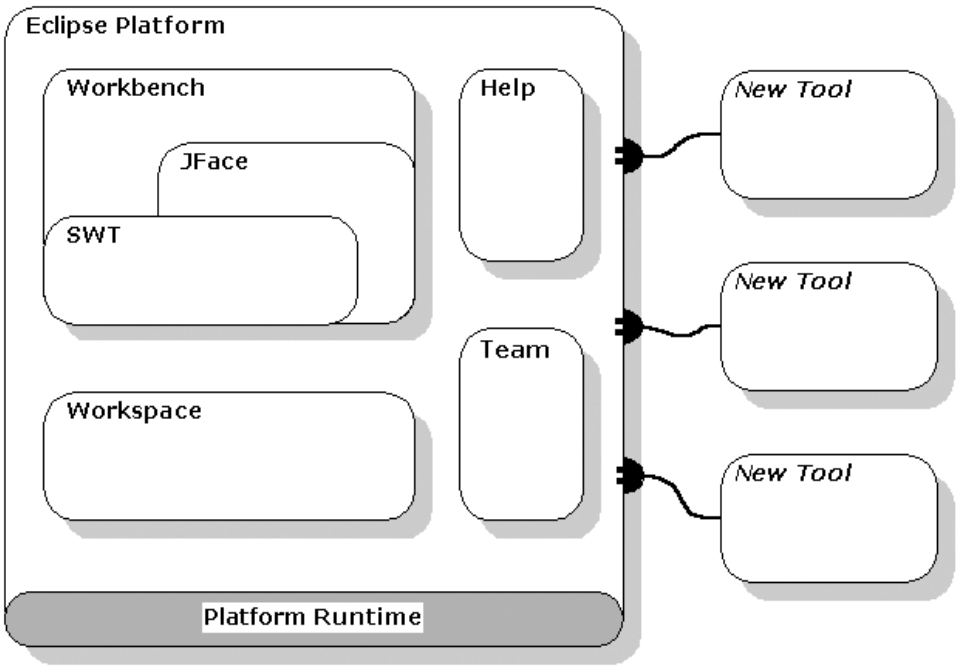


Illustration 1: Die Eclipse Plattform

Die durch die Abkopplung der Anwendung vom Framework gewonnene Flexibilität macht es möglich, auch die Eclipse-IDE beliebig durch eigene Plugins zu erweitern. Die Eclipse-IDE als Plattform für dieses Projekt ist geradezu die Musterlösung. Nicht nur die vielen, für eine Entwicklungsumgebung unabdingbaren Funktionen, welche von Haus aus integriert sind, sondern auch die Quelloffenheit von Eclipse sind optimal für dieses Projekt. Die Quelloffenheit lässt diesem Projekt viele Möglichkeiten offen, sich in Zukunft weiterentwickeln zu können.

Das Nutzen dieses Frameworks hat allerdings auch seine Schattenseiten. Durch die Abhängigkeit eines Plugins von den Funktionalitäten des Eclipse Frameworks kann es durchaus vorkommen, dass bei Versions-Änderungen des Eclipse Frameworks Plugins einer älteren Version nicht funktionieren. Dies muss bei jedem Plugin ausgiebig analysiert werden.

2.2 Plugins

Da Eclipse auf der OSGi Plattform Equinox aufsetzt, basiert Eclipse auch bei der Erweiterbarkeit auf Equinox. Hierbei wird die Definition eines OSGi Bundles erweitert zu der Definition eines Plugins [EH3.4].

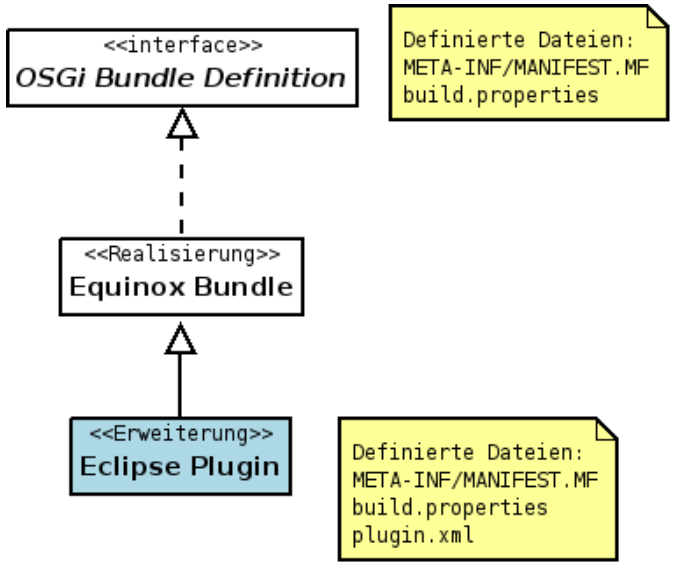


Illustration 2: Abhängigkeiten der Verschiedenen Bundle Definitionen

Ein OSGi Bundle muss im Allgemeinen und ein Plugin im Speziellen einer vorgegebene Struktur genügen. In der OSGi Definition ist festgeschrieben, dass ein Manifest existieren muss. In einem Manifest wird unter anderem der Name, die Version, eventuelle Abhängigkeiten von anderen Bundles und die beim Aktivieren des Bundles zu startende Klasse angegeben.

Um ein Plugin für das Eclipse Framework zu erstellen, können die von einem OSGi Bundle zur Verfügung gestellten Informationen ausreichen. Allerdings kann zusätzlich noch eine *plugin.xml* angelegt werden. In dieser kann unter anderem definiert werden, welche *Extension Points* genutzt werden.

2.3 Extension Points

Jedes Plugin kann Extension Points definieren und nutzen. Wie der Name schon sagt, handelt es sich hierbei um einen Erweiterungspunkt. Definiert ein Plugin *A* einen eigenen Extension Point, so können andere Plugins durch das Nutzen dieses Extension Points das Plugin *A* an diesem definierten Punkt erweitern. Diese nützliche Funktionalität wird eingesetzt, um eigene Erweiterungen zu bereits bestehenden Plugins hinzuzufügen. So kann man hierdurch auch mit einem eigenen Plugin Änderungen an der Eclipse Oberfläche vornehmen.

2.4 Features und Update-Sites

Eclipse nutzt zum Verwalten von Plugins eine zentrale Komponente, den Update Manager. Über diesen lassen sich Features installieren. Jedes Feature besteht aus einem oder mehreren Eclipse Plugins. Ein Feature bildet also quasi eine Aggregation von Plugins. Es beschreibt die Funktionen, die es durch die Plugins bereitstellt. Features sind auf Update-Sites zu finden. Eine Update-Site stellt eine Ansammlung von Features dar. Update-Sites können dabei sowohl lokal als auch per Internet erreichbar sein.

Im Update-Manager kann man Update-Sites verwalten und sie nach Features durchsuchen. Die Beschreibung eines Features sollte dem Benutzer eine weitere Abstraktionsebene über der Plugin-Ebene bieten. Hierdurch entsteht der Vorteil, dass die vom Feature bereitgestellte Funktionalität einfacher zu erkennen ist.

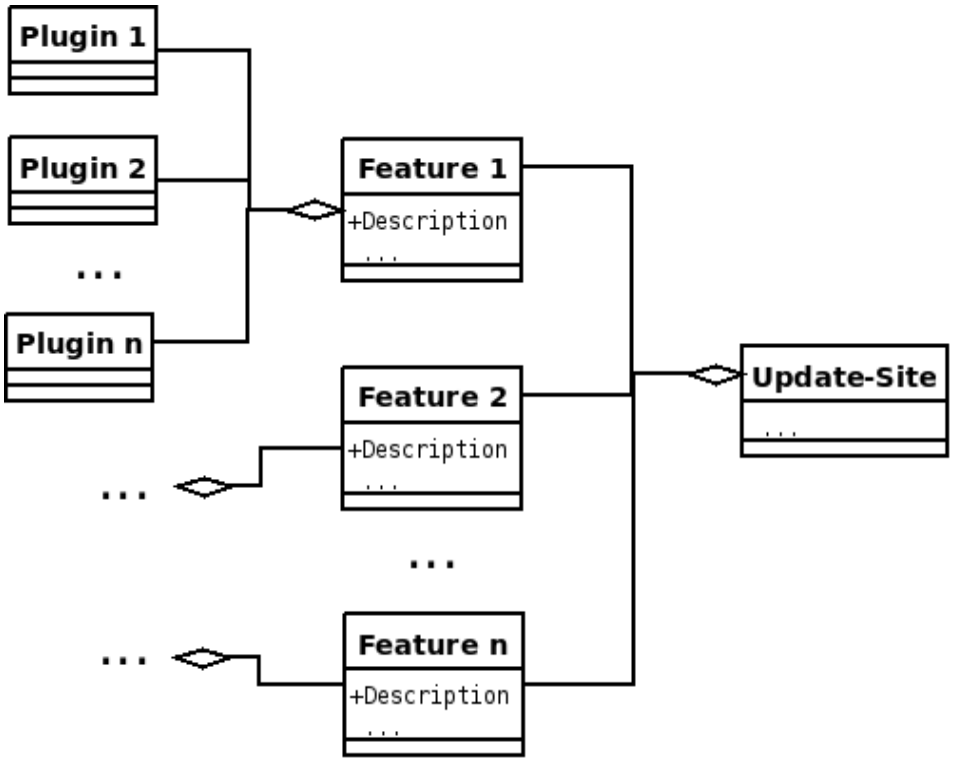


Illustration 3: Plugins, Features und Update-Sites

Durch diese modulare Architektur ist es relativ einfach, einem großen Nutzerkreis seine Plugins auf eine komfortable Art und Weise zur Verfügung zu stellen. Dem Benutzer wird außerdem sofort die Integration des Plugins in die Entwicklungsumgebung bewusst, da schon bei der Installation auf Methoden zurückgegriffen wird, welche bereits von der Plattform bekannt sind.

3 Migration von Eclipse 3.1 zu Eclipse 3.4

Dieses Kapitel beschreibt die Migration des vorhandenen Plugins zur Unterstützung der Arbeit mit dem Typ Inferenz Compilers von der Eclipse-IDE Version 3.1 zur Version 3.3 bzw. 3.4[1].

Hierbei werden zuerst alle wesentlichen Änderungen im Eclipse Framework vorgestellt. Es werden nur Änderungen in Betracht gezogen, welche große Auswirkungen auf die Plugin-Infrastruktur haben. Anschließend wird analysiert, wie sehr das Plugin von den Änderungen betroffen ist und wie hoch der Aufwand ist, das bestehende Plugin zu migrieren. Das Unterkapitel Durchführung beschreibt die deswegen vorgenommenen Änderungen am Plugin.

3.1 Wesentliche Framework Änderungen

Da zwischen der Eclipse Version 3.1 und der Eclipse 3.4 die Versionen 3.2 und 3.3 liegen, besteht dieses Kapitel aus drei Unterkapiteln. Jedes Kapitel behandelt die entstehenden Unterschiede bei dem betreffenden Versionswechsel.

3.1.1 Eclipse 3.1 zu 3.2

In Eclipse 3.2 wurde eine neue Processing Instruction für die *plugin.xml* eingeführt. Processing Instructions sind Anweisungen an den XML Parser. In Eclipse lautet die normale Processing Instruction für die *plugin.xml* seit Version 3.0 *<?eclipse version= 3.0 ?>*Da Eclipse 3.2 allerdings nun erlaubt, Extension Points auch in einem anderen Namespace als dem eigenen zu definieren, wurde für Anwendungenm die diese Funktion nutzen wollen, die Processing Instruction *<?eclipse version= 3.2 ?>*eingeführt.

Des Weiteren ist es nun möglich, die genaue Laufzeitumgebung des Plugins definieren zu können. So kann man festlegen, welche Version des Java Runtime Environments mindestens verfügbar sein muss. Diese Einschränkung ist in den meisten Fällen sehr sinnvoll, da der Benutzer bei Nichterfüllung der Mindestanforderungen eine aussagekräftige Fehlermeldung bekommen kann. Bisher konnten nicht erfüllte Mindestanforderungen nur zur Laufzeit erkannt

1 Zum Zeitpunkt des Verfassens war die Eclipse-IDE Version 3.4 noch nicht als Release veröffentlicht.

werden, und dies meistens nicht durch Laien.

Seit der Eclipse Version 3.2 kann nicht mehr einfach davon ausgegangen werden, dass sich eine Ressource auf einem normalen lokalen Dateisystem befindet. Damit ist es möglich, neue Arten von Dateisystemen in Eclipse einzubinden. Es wäre zum Beispiel die transparente Repräsentation eines Archivs (zip, tar, oder ähnliche) als ein normales Dateisystem möglich. Diese Umstellung hat allerdings nicht nur positive Folgen. Ältere Plugins könnten Methoden nutzen, welche versuchen, den lokalen Pfad zu einer Datei zurückzugeben. Ist dies der Fall, würde es zwangsläufig zu einem Fehler kommen, falls ein nicht lokales Dateisystem benutzt werden würde.

Mehr Informationen zu den Veränderungen die zu Komplikationen bei der Migration von Eclipse 3.1 auf Eclipse 3.2 führen sind auf [EM31-32] zu erfahren.

3.1.2 Eclipse 3.2 zu 3.3

Die Änderungen von Eclipse 3.2 zu Eclipse 3.3 waren wie bei 3.1. zu 3.2 zahlreich. Allerdings waren es eher die kleinen Änderungen, die diese Version in Hinsicht auf die Plugin-Programmierung geändert haben.

Die wohl wichtigste Änderung gab es bei der so genannten *Boot Delegation*. Die Boot Delegation entscheidet, welche Plugins wann geladen werden. Dies ist insofern wichtig, da die meisten Plugins andere Plugins als Abhängigkeiten definieren. Würden jetzt bei einem Plugin die Abhängigkeiten nicht vor dem eigentlichen Plugin geladen werden, käme es zu Fehlern. Die Reihenfolge änderte sich wie folgt:

1. importierte Packages
2. benötigte Bundles
3. das zu ladende Bundle und dessen Fragmente
4. Boot (parent) class loader

Vor Eclipse 3.3 wurde der Boot (parent) class loader immer als Erstes geladen. Wenn sich ein

Plugin nun auf diese Reihenfolge verlässt, kann es sein, dass benötigte Klassen nicht oder falsche Klassen geladen werden.

Weitere Informationen zu den Änderungen zwischen Version 3.2 und 3.3, welche die Kompatibilität eines Plugins beeinflussen, sind auf [EM32-33] zu finden.

3.1.3 Eclipse 3.3 zu 3.4

Als wohl größte Änderung ist die Einführung von *Eclipse Provisioning Platform (p2)* als Provisionierungssystem für Plugins in Eclipse 3.4 anzusehen. Es ersetzt den bisherigen Update Manager. In dieser Studienarbeit wird aber noch der Update Manager als Beispiel herangezogen, da Eclipse 3.4 zum jetzigen Zeitpunkt noch nicht in der endgültigen Version verfügbar ist. Die Verwendung von *p2* anstatt des Update Managers ist aber ziemlich gleich.

Das neue Provisionierungssystem bringt Änderungen in vielen Bereichen mit sich. Die logische Struktur einer Update-Site hat sich allerdings nicht geändert. Die Änderungen betreffen eher die manuellen Installationen von Plugins und den Betrieb mehrerer Eclipse Applikationen auf einem System.

Vor Eclipse 3.4 war eine Installation von Plugins ohne Benutzung des Update Managers durch einfaches Verschieben des Plugins in den *plugin* Ordner einer Eclipse Applikation möglich. Dieses Vorgehen ist zwar relativ einfach, jedoch eher unsystematisch und kann schnell zu einer falsch konfigurierten Eclipse Applikation führen. Eclipse 3.4 bietet für die Installation von Plugins ohne GUI einen separaten *dropin* Ordner an. Hier hinein können Plugins kopiert werden, um dann beim nächsten Start von Eclipse automatisch erfasst und installiert zu werden.

Wenn mehrere Eclipse Anwendungen auf einem System installiert und betrieben werden, hatte bisher jede Applikation alle Plugins in einem privaten *plugin* Ordner. Hierdurch kam es des Öfteren vor, dass die gleichen Plugins gleich mehrfach auf der Festplatte vorhanden waren. Mit *p2* können sich nun Eclipse Applikationen einen gemeinsamen *plugin* Ordner teilen. Dies reduziert den für mehrere Eclipse Applikationen benötigten Speicherplatz erheblich.

Eclipse 3.4 beinhaltet natürlich noch weitere Änderungen, diese sind aber für die Migration des

Plugins nicht von Bedeutung. Die zwei wichtigsten Dokumente hierzu sind [ENT34M6a] und [ENT34M7].

3.2 Analyse

Wie man sieht, hat sich zwischen Eclipse 3.1 und Eclipse 3.4 einiges verändert. Das vorhandene Plugin, welches kompatibel zu Eclipse 3.1 ist, muss deswegen auf eventuell vorhandene Inkompatibilitäten untersucht werden.

3.2.1 API Änderungen

Hierzu wurden zunächst die Abhängigkeiten des Plugins untersucht. Wenn bei einer dieser Abhängigkeiten eine Änderung in der API (Application Programming Interface) vorgekommen ist, müssen die betroffenen Komponenten abgeändert werden, da eine Änderung in der Schnittstelle vorgenommen wurde. Zur Hilfe der Analyse von Abhängigkeiten wurde ein Diagramm erstellt.

Illustration 4: Visualisierung der Abhängigkeiten des Plugins (GELB: Typinferenz Plugin, Orange: Directe Abhängigkeiten,

Aus diesem Diagramm können die direkten Abhängigkeiten des Plugins abgelesen werden. An dieser Stelle kann man gut sehen, dass das Plugin unter anderem auch Abhängigkeiten zum

Plugin *org.eclipse.core.resources* aufweist. In diesem Plugin wird der Zugriff auf verschiedene Ressourcen gesteuert, wie zum Beispiel das Dateisystem. Da dieser Zugriff sich allerdings zwischen Eclipse 3.1 und 3.2 geändert hat, müssen wir die Abhängigkeiten genauer betrachten.

Die Abhängigkeiten von den anderen Packages sind für die Migration eher als weniger kritisch einzustufen, da sich an deren API eher wenig geändert hat.

Analysiert man nun die Zugriffe auf Klassen des Plugins *org.eclipse.core.resources* genauer, so wird man feststellen, dass keine der veränderten Methodenaufrufe verwendet wurden. Es sind deswegen keine Fehler bei der Migration zu erwarten, die durch eine API Änderung hervorgerufen werden.

3.2.2 Funktionale Änderungen

Neben den API Änderungen musste das Plugin aber auch noch daraufhin untersucht werden, ob alle Features, welche es vom Eclipse Framework implizit genutzt hat, auch in den neuen Eclipse Versionen vorhanden sind. Ansonsten würde dem Benutzer nicht der gleiche Funktionsumfang wie in Eclipse 3.1 gewährleistet.

Diese Inkompatibilitäten fallen allerdings erst nach dem ersten erfolgreichen Start in Eclipse 3.4 auf. Da keine Probleme mit der API-Kompatibilität aufgetreten sind, muss an dieser Stelle nichts geändert werden. Das Plugin muss nur gegen die neuen Versionen der Plugin-Abhängigkeiten compiliert werden und schon ist es lauffähig.

Diese Art der Inkompatibilitäten ist meistens nicht in der Dokumentation von Eclipse aufgenommen, da sie durch das Nutzen von API-Aufrufen entstehen, welche nicht für andere Plugins gedacht waren aber trotzdem zur Verfügung standen.

Nach dem ersten Start mit Eclipse 3.3 kann man die funktionalen Änderungen speziell für dieses Plugin entdecken. Es fällt ziemlich schnell auf, dass das Syntaxhighlighting komplett verschwunden ist. Bisher wurde das Syntaxhighlighting implizit bei der Erstellung eines neuen Editor bereitgestellt. Nun muss jeder Editor das für ihn richtige Syntaxhighlighting selbst implementieren. Macht diese Änderung die Portierung von Plugins auf eine neue Version des

Eclipse Frameworks schwieriger, so lässt sie aber auf der anderen Seite Programmierern den Freiraum auch ein anderes, als das herkömmliche Syntaxhighlighting zu verwenden.

Neben dem fehlenden Syntaxhighlighting sind keine weiteren funktionellen Unterschiede zu erkennen.

3.3 Durchführung

Um in Eclipse Syntaxhighlighting für einen Editor zu integrieren, müssen bestimmte Regeln gefunden werden nach denen ein Wort, ein Absatz oder ein Zeichen in einer anderen Farbe hervorgehoben werden. In Java sind die wichtigsten Regeln folgende:

1. Einzeilige Kommentare fangen mit // an und enden am Ende der Zeile
2. Mehrzeilige Kommentare fangen entweder mit /* oder /** an und enden mit */
3. Zeichenketten (Strings) fangen mit an und enden mit
4. Ein Zeichen welches ein ' vorangestellt und ein ' angehängt ist.
5. Schlüsselwörter bestehen aus einem Wort. Ein Vorkommen eines Schlüsselwortes mit Präfix oder Suffix ist kein Schlüsselwort.

Es könnten jedoch noch mehr Syntaxhighlighting-Features wie etwa das kontextabhängige Hervorheben von Konstanten oder Attributen implementiert werden. So ist im normalen Java-Eclipse-Editor neben diesem Feature auch das Durchstreichen eines Aufrufs, von als *deprecated* deklarierte Methoden, vorhanden. Um dies zu implementieren, müsste allerdings eine umfangreiche Klassenstruktur angelegt werden, was allerdings nicht im Rahmen dieser Studienarbeit zu realisieren wäre.

Das folgende Klassendiagramm erläutert den Aufbau der, für das Syntaxhighlighting notwendigen, Struktur. Durch diese verwendete Struktur werden auch weitere, bisher implizit bereitgestellte, Funktionen (wie zum Beispiel die Annotations) nun explizit verwendet.

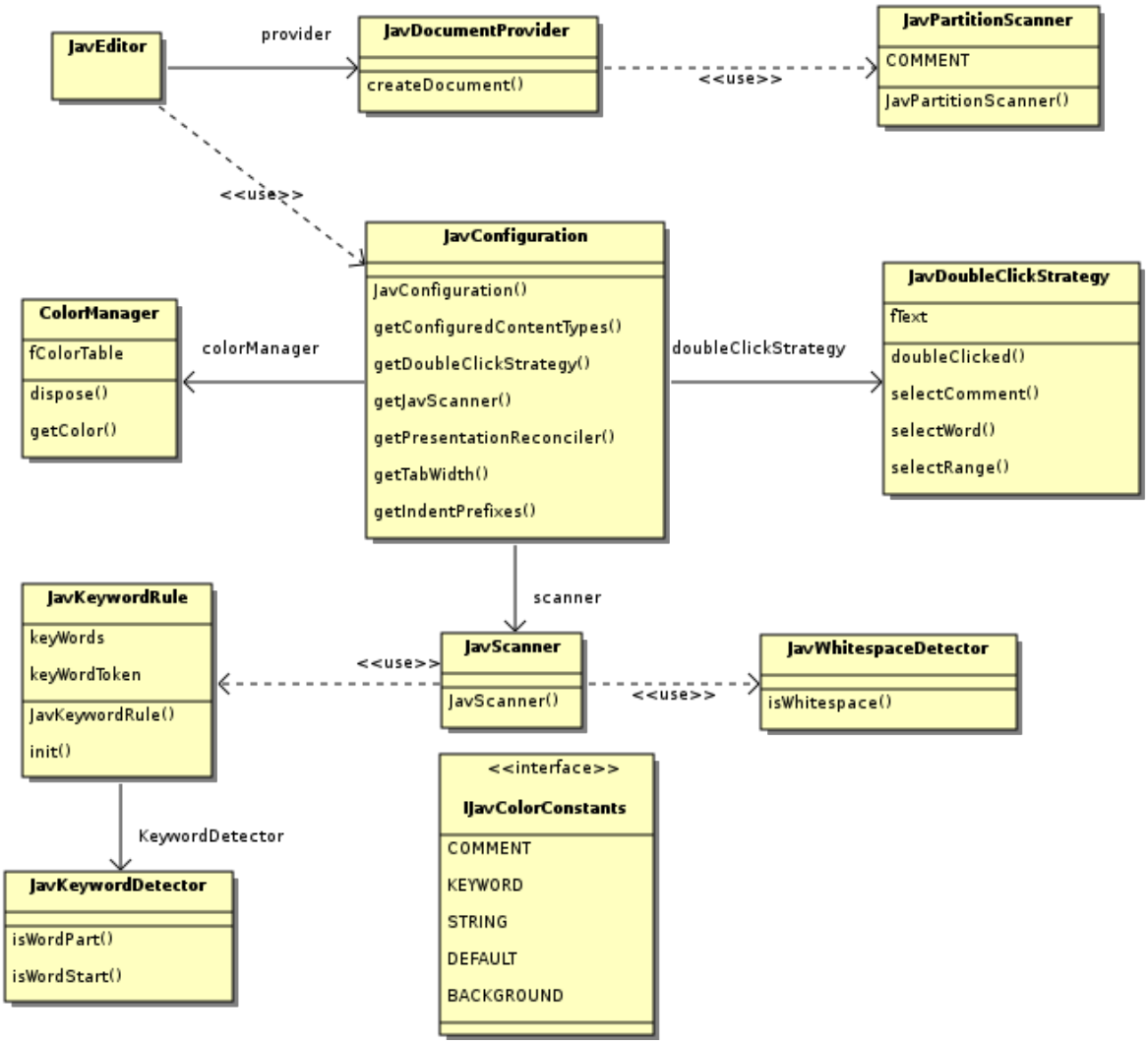


Illustration 5: Klassendiagramm zur Integrierung der Syntaxhighlighting Funktion

Im Mittelpunkt der Klassenstruktur steht die Klasse *JavConfiguration*, welche von *SourceViewerConfiguration* erbt. Diese Klasse stellt einen zentralen Verwaltungspunkt bereit für alle Funktionen, die die Ansicht des Editors betreffen.

Die Klasse *ColorManager* dient zum schonenden Umgang der Ressourcen der *Color* Objekten des Standard Widgeting Toolkit (SWT). Da das SWT, anders als andere Java Widgeting Toolkits, native Komponenten des Betriebssystems nutzt, verwendet es eine kompliziertere Ressourcenverwaltung. Java SWT Objekte, die nicht mehr gebraucht werden, können nicht

einfach nur vom Garbage Collector freigegeben werden, da sie meistens nur eine Java Repräsentation für die darunter liegenden Systemobjekte darstellen. Diese Abhängigkeit würde der Garbage Collector allerdings nicht entdecken und so nur den Java Teil der logischen Objekte freigeben. Um aber auch zu gewährleisten, dass die darunter liegenden nativen Strukturen bereinigt werden, muss bei jeder SWT Komponente die Methode dispose() aufgerufen werden. Die Verwaltung der Farben durch den *ColorManager* garantiert zum Einen, dass für jede Farbkombination nur ein SWT Objekt angelegt wird, und zum Anderen, dass alle über den *ColorManager* angelegten Objekte auch auf nativer Ebene freigegeben werden.

JavDoubleClickStrategy implementiert das Interface *ITextDoubleClickStrategy*. Diese Klasse bestimmt das Verhalten des Editors, wenn der Benutzer einen Doppelklick ausführt. Die vom Benutzer erwartete Reaktion ist hier meistens, dass ein oder mehrere Wörter markiert werden. Was markiert wird, entscheidet diese Klasse. *JavDoubleClickStrategy* implementiert hierbei, dass bei einem Doppelklick das angeklickte Wort markiert wird.

Nun kommen wir zur eigentlichen Implementierung des Syntaxhighlightings. Die von *RuleBasedScanner* erbende Klasse *JavScanner* dient zum Erkennen von hervorzuhebenden Strukturen im Quelltext. Hierbei orientiert sich die Klasse an Regeln (*Rule*), nach denen sie die Strukturen erkennen kann. Je nachdem ob eine Struktur einer Regel entspricht wird dann auf diesem Bereich eine, für diese Struktur geltende, Formatierung angewendet. Um Java Schlüsselwörter zu finden wurde hierfür, die von *WordRule* erbende Klasse, *JavKeywordRule* erzeugt, welche ausschließlich hierzu dient. Die Definitionen, in welcher Farbe eine bestimmte Art von Struktur angezeigt wird, sind im Interface *IJavColorConstants* hinterlegt.

Viele Java Strukturen können hierdurch richtig erkannt und hervorgehoben werden. Dies gilt allerdings nicht für Regeln, die andere Regeln außer Kraft setzen würden. Zu diesen Regeln gehören:

- Zeichenketten
- Einzeilige Kommentare
- Mehrzeilige Kommentare

Um in diesen Strukturen andere Regeln, wie etwa die *JavKeywordRule*, auszuschalten, muss mit Hilfe des *JavPartitionScanners* das Dokument in Partitionen unterteilt werden. In diesen Partitionen werden die normalen Regeln nicht angewendet. Auf eine Partition als Ganzes können Formatierungen durchgeführt werden. Die im Plugin vorgenommenen Formatierungen betreffen nur die Farbgebung der Kommentare und Zeichenketten. Die Farben sind, wie schon zuvor, in dem Interface *IJavColorConstants* definiert.

Die hier vorgestellte Methode zur Implementierung des Syntaxhighlighting ist der übliche Weg bei der Plugin-Programmierung für Eclipse. Sie bietet dem Programmierer weitreichende Möglichkeiten das Aussehen des Editors für den Benutzer mit relativ einfachen Mitteln zu verändern.

Die reine Migration auf Eclipse 3.4 ist nun abgeschlossen. In dem nächsten Unterkapitel wird die weitere Integration in das Eclipse Framework besprochen.

3.4 Weitere Integration in das Eclipse Framework

Um das Installieren, Aktualisieren und Deinstallieren von Eclipse Plugins so einfach wie möglich zu gestalten, wurde hierfür der Update Manager im Rahmen des Eclipse Frameworks erstellt. Er bietet einen zentralen Konfigurationspunkt für alle Aufgaben rund um das Management eines Eclipse Features.

Um diese Möglichkeiten zu nutzen und um dem Plugin eine bessere Integration in das Eclipse Framework zu schaffen, wurde die bestehende Projektstruktur erweitert und modifiziert. Hierzu war es nötig das Plugin in einem Feature zu verpacken. Das Feature hat den beschreibenden Namen *TypeInference* bekommen. Es wurde zusätzlich mit einer verständlichen Beschreibungen versehen, welche einem Benutzer der Zielgruppe den Nutzen des Plugins erklärt.

Um dieses Feature auch einfach zugänglich zu machen, wird es mit Hilfe einer Update-Site installierbar gemacht. Diese Update-Site kann beliebig viele Features beinhalten. Das Projekt benötigt allerdings nur das *TypeInference* Feature.

Diese beiden Projekte, zum einen das Feature *TypeInference* und zum anderen die Update-Site,

müssen eigenständige Eclipse Projekte sein, um vernünftig verwaltet werden zu können. Das Feature Projekt besteht außer dem Plugin, welches zuständig ist für die Typinferenz Funktionalität, auch aus der log4j Bibliothek, als Eclipse Plugin verpackt, und einem Plugin-Fragment, welches log4j konfiguriert. Näheres dazu im nächsten Kapitel und in der, im Anhang vorzufindenden, Zeichnung (Überblick über Plugins, Features und Update-Sites).

Die entstandene Update-Site kann verwendet werden, um einen einfachen Zugang zum TypeInference Feature für Eclipse zu bieten. Im Rahmen der Transformation in ein Open Source Projekt wäre es sinnvoll, die Update-Site auf einem Webserver zu hosten. Die Benutzer müssten dann nur die Adresse der Update-Site in Eclipse angeben, um das Feature installieren zu können.

Eine genaue Anleitung zur Installation, Aktualisierung und Deinstallation eines Features über eine Update-Site ist im Anhang beigefügt.

4 Integration von log4j

4.1 Intention

Das Logging spielt in Projekten der Software Entwicklung eine sehr große Rolle. Es unterstützt die Entwickler beim schnellen Finden von Fehlern und deren Korrektur. Außerdem bietet ein ausgefeiltes Logging-Konzept dem Benutzer eine Menge Informationen über eventuell entstandene Fehler und deren Ursachen. Hierbei ist es allerdings wichtig, dass der Benutzer nicht mit Logging-Informationen in verschiedenen Repräsentationen oder an logisch verschiedenen Speicherorten konfrontiert wird. Dies würde ein großes Verwirrungspotential mit sich bringen, da unter anderem die zeitliche Abfolge von Logging Einträgen nicht mehr unbedingt übersichtlich wäre.

Der Typinferenz Compiler nutzt die log4j Bibliothek zum Loggen von Meldungen. log4j ist die meist genutzte Logging Bibliothek im Java Umfeld.

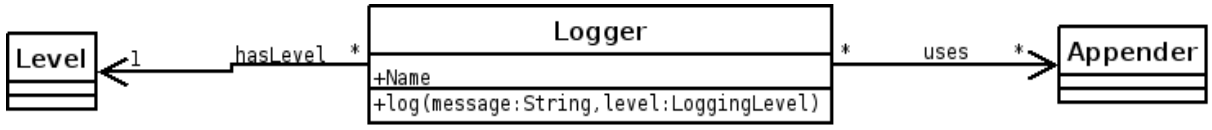


Illustration 6: Verhältnis der Log4j Komponenten

Log4j basiert auf einer einfachen Konfigurierbarkeit. Man kann verschiedene Typen von *Loggern* definieren. Ein Logger ist konfiguriert auf ein spezielles *Level*, das so genannte *Logging Level*. Logging Level bestimmen den Grad an Informationen, die beim Logging ausgegeben werden. Das Level *DEBUG* würde alle Ausgaben enthalten, das Level *FATAL* jedoch nur fatale Fehlermeldungen. Neben dem Level ist dem Logger auch eine beliebige Anzahl von Appendern zugewiesen. Ein Appender definiert wie die Logging-Informationen ausgegeben werden. Es gibt verschiedene Typen von Appendern. Der *FileAppender* würde die Logging Informationen zum Beispiel in eine Datei schreiben. Sowohl Level, als auch Appender kann man selber erweitern. Dieses Modell zeichnet die Flexibilität von log4j aus. Die Beliebtheit der Bibliothek über viele Jahre hinweg hat sie quasi zum de facto Standard für Logging unter Java gemacht.

Im Gegensatz zum Typinferenz Compiler benutzte das Eclipse Plugin eine selbst geschriebene

Logging Funktionalität. Diese Lösung machte es schwerer die Ausgaben vom Compiler (Log4J) und die des Plugins zu konsolidieren und dem Benutzer in angemessener Form zu präsentieren.

Eine Konsolidierung der beiden Logging Funktionalitäten ist in diesem Fall sinnvoll. Das Plugin musste auf Log4J als Logging Bibliothek umgestellt werden. Dies hat zudem den Vorteil eine zentrale Konfigurationsdatei für das Logging bereitstellen zu können.

4.2 Änderungen in der Architektur

Die Integration von Log4J als Logging Lösung bringt eine Änderung in der Plugin Architektur mit sich.

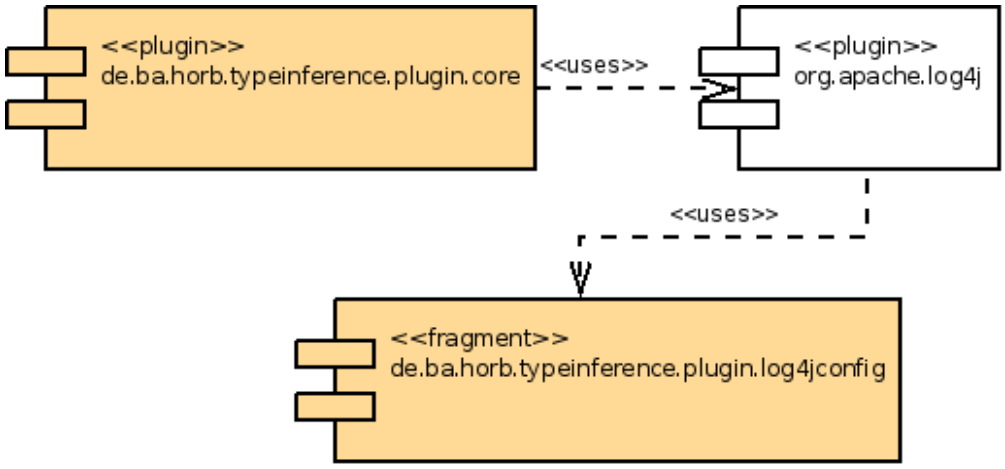


Illustration 7: Plugin Abhängigkeiten in Bezug auf Log4J

Als zusätzliche Abhängigkeit bekommt das Typinferenz Plugin (*de.ba.horb.typeinference.plugin.core*) das log4j Plugin, welches die benötigte log4j Bibliothek beinhaltet. Durch die Erstellung eines log4j Plugins ist die log4j Bibliothek für alle Plugins verfügbar.

Die Konfigurationsdatei wird vom Quelltext unabhängig aufbewahrt, so dass sie ohne das Plugin neu kompilieren zu müssen, abgeändert werden kann. Sie wird in einem so genannten *Fragment* integriert. Ein Fragment erweitert immer ein anderes Plugin um weitere Funktionalitäten. So erweitert das Plugin *de.ba.horb.typeinference.plugin.log4jconfig* das *org.apache.log4j* Plugin um die Konfigurationsdatei. Darüber hinaus befinden sich im besagten Fragment auch eine

Erweiterung der log4j Level Struktur um ein neues Logging Level, welches im Kontext des Plugins benötigt wird.

Durch diese locker gekoppelten Abhängigkeiten kann sowohl die log4j Bibliothek, als auch die Konfigurationsdatei ohne eine Veränderung des Typinferenz Plugins ausgetauscht werden.

5 Visualisierung von Einschränkungen bei der Typinferenz

Eine weitere Aufgabe dieser Studienarbeit ist die Visualisierung von Einschränkungen bei der Typinferenz. Die Einschränkungen werden durch den Benutzer über die grafische Schnittstelle des Eclipse Plugins vorgenommen.

Bisher war es bereits möglich bei der Inferenz mehrerer möglicher Typen einen Bestimmten auszuwählen. Diese Auswahl wurde dann auch im Compiler umgesetzt. Eine Rückmeldung an den Benutzer war allerdings nicht vorhanden und der Quelltext reflektierte die Einschränkung nicht.

Dies ist für den Benutzer nicht nur sehr irritierend, sondern verhindert auch das Übertragen der Einschränkung auf andere Editoren.

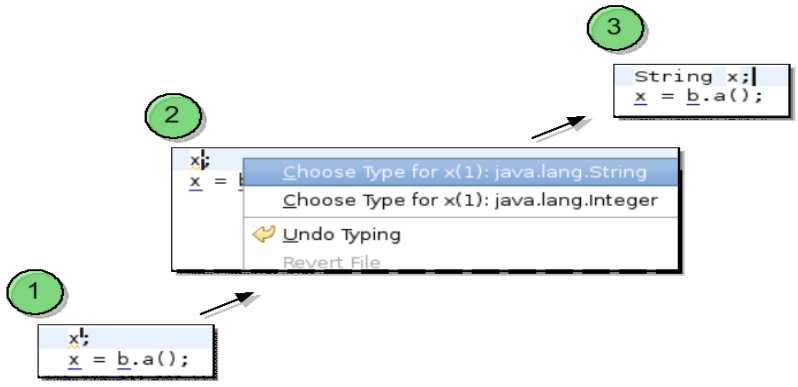


Illustration 8: Die drei Schritte zur Visualisierung:

1. Mehrere Typen werden angeboten.

2. Einen Typ auswählen.

3. Änderung wird im Quelltext übernommen

Um die Einschränkung geeignet zu Visualisieren und gleichzeitig Persistent zu machen, wurde der Ansatz gewählt, die Einschränkungen direkt in den Quelltext zu schreiben. Dadurch entspricht die Typdeklaration nach einer Beschränkung wieder der normalen Java Syntax.

5.1 Offset Berechnung

Eine wichtige Information, zur richtigen Bearbeitung einer Einschränkung, ist die Berechnung der richtigen Offsets für die Ersetzung. Unter Offset versteht man die Position eines Elements innerhalb einer Datei. Die Position wird in Anzahl der Zeichen vor dieser Position angegeben.

Auf Grund oftmals ungenauer Angaben über die Position eines Elements ist die Offset Berechnung ist meist nicht trivial. So liefert die bereits vorhandene Datenstruktur bei Methoden lediglich den Anfangs Offset des Methodennamens zurück. Für die Einsetzung eines Rückgabetyps einer Methode ist dies nicht genügend.

Die trivialste Art der Offset Berechnung ist bei lokalen Variablen zu erkennen. Hier ist der Ersetzungs Offset immer gleich dem Deklarations Offset.

Ein wenig prekärer ist die Lage bei den Methoden Offsets. Hier muss unterschieden werden zwischen Parameter-. Rückgabe- und Generics Deklarations Offsets.

5.2 Ersetzungsstrategien

Je nach Typ und Art der Variable müssen verschiedene Ersetzungen vorgenommen werden, um die Änderung im Quelltext sichtbar zu machen.

5.2.1 Lokale Variablen

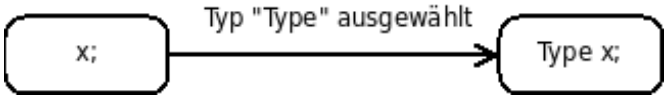
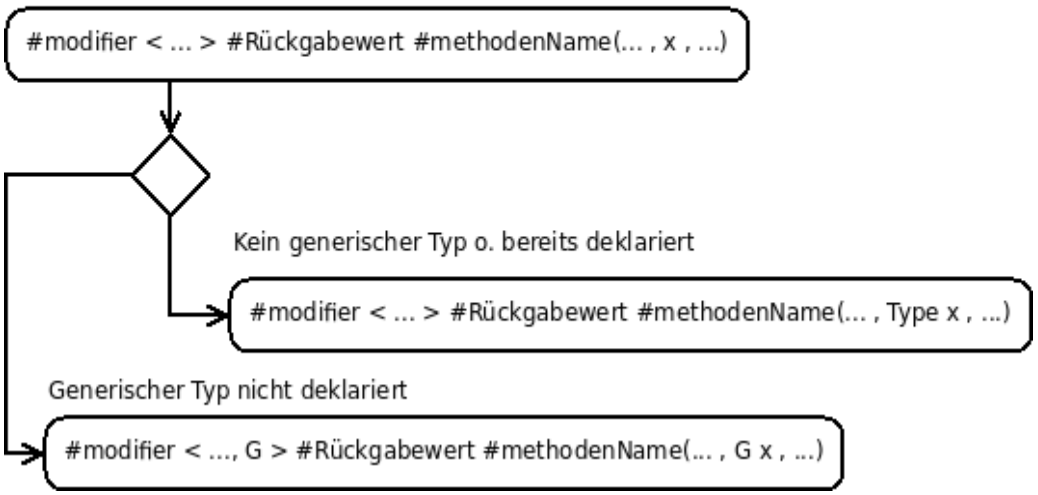


Illustration 9: Fälle der Ersetzung bei einer lokalen Variable

Die einfachste Ersetzung ist hierbei das Einsetzen von Typdeklarationen bei lokalen Variablen. Hier ist es in jedem Fall möglich die Typdeklaration vor der Deklaration der Variable einzusetzen.

5.2.2 Methodenparameter

Illustration 10: Fälle der Ersetzung bei Methodenparametern



Die Typdeklarationen von Methodenparameter können je nach Typ direkt vor dem Parameter eingesetzt werden. Dies ist bei den meisten Typen möglich. Handelt es sich allerdings bei dem Typ um einen generischen Typ, welcher bisher nicht existiert hat, so muss zusätzlich noch die Deklaration des generischen Typs in die Methodendeklaration aufgenommen werden.

5.2.3 Methodenrückgabewerte

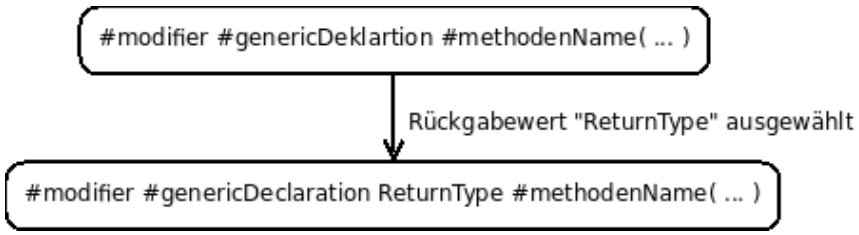


Illustration 11: Fälle der Ersetzung bei einem Methodenrückgabewert

Der Rückgabewert einer Methode ist direkt im Methodenkopf deklariert. Er ist erst einmal von keiner Variable im Methodenkörper abhängig. Einzig alleine an einem *return* Statement kann erkannt werden, welchen Rückgabetyp die Methode besitzt. Ist dieser von dem Benutzer

beschränkt worden, sollte der Typ in der Methodendeklaration eingesetzt werden. Hier ist die richtige Position in der Methodendeklaration zu finden an der der ausgewählte Typ eingesetzt werden kann.

5.3 Architekturänderungen

Das Hinzufügen dieser Visualisierung benötigt einige Änderungen an der Architektur des Plugins.

Den zentralen Punkt dieser Änderungen repräsentiert die Klasse *JavMethod*. Sie übernimmt die logische Repräsentation einer Methode im Quelltext. Die Klasse *JavMethod* ist nicht mit der Klasse *JavMethodIdent* zu verwechseln. *JavMethodIdent* ist eine Repräsentation der Typrekonstruierung einer Methode. *JavMethod* hingegen repräsentiert einzig und alleine die Methode, so wie sie im Quelltext vorkommt. Sie dient vor allem zur Ermittlung von Offsets, die für die Ersetzung notwendigen sind. Hierfür verfügt die Klasse über eine Referenz auf ein Objekt vom Typ *JavSyntaxRelatedScanner*. Die Klasse *JavSyntaxRelatedScanner* stellt Operationen zum Suchen von relevanten Zeichenketten in einem Java Quelltext bereit. Alle Operationen beachten dabei, dass Zeichenketten innerhalb von Kommentaren oder Strings nicht relevant sind.

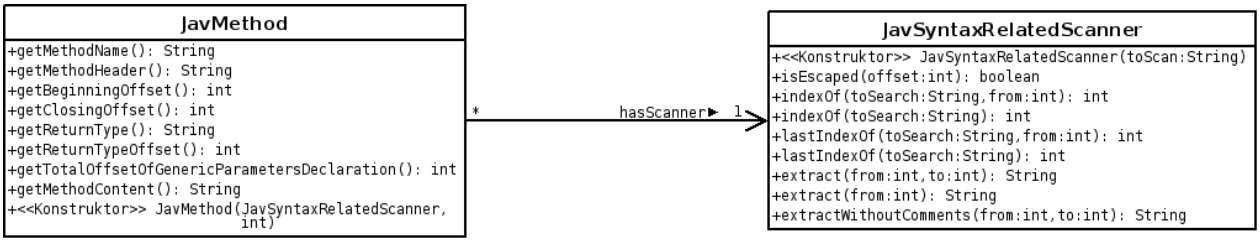


Illustration 12: Klassendiagramm von JavMethod und JavSyntaxRelatedScanner

Um die Visualisierung auszuführen wurde die Klasse *ContextAction* verändert. Sie wurde um Methoden erweitert, die an den gefundenen Offsets den entsprechenden Typ im Quelltext einsetzen.

5.3.1 Änderung des Ablaufs des Typrekonstruktions-Aufrufs

Um die Auswirkungen der Typ-Einschränkung durch den Benutzers visuell Repräsentieren zu

können, müssen diese erst berechnet werden. Jede durch den Benutzer vorgenommene Typ-Einschränkung bei einer Variable kann mehrere Einschränkungen bei anderen abhängigen Variablen nach sich ziehen. Diese Abhängigkeit wird an einem kleinen Beispiel erklärt.

```
public method(){
      a;
      b;
      b = a;
      a = new Vector<Integer>();
}
```

Illustration 13: Beispiel zu Abhängigkeiten zwischen noch nicht bestimmten Variablen

In dem in Illustration 13 vorgestellten Beispiel ist die Variable *b* von der Variable *a* abhängig. Der Typinferenz-Algorithmus gibt *Vector<Integer>*, *Vector<? extends Integer>* und *Vector<? super Integer>* als mögliche Typen für *a* zurück. Für *b* wurden durch die direkte Abhängigkeit von *a* die gleichen Typen bestimmt. Wenn nun *a* auf den Typ *Vector<? extends Integer>* eingeschränkt wird, muss auch die Menge der möglichen Typen von *b* auf *Vector<? extends Integer>* und *Vector<Integer>* eingeschränkt werden. Damit die, für den Benutzer zur Auswahl stehenden, Typen immer richtig bestimmt werden, muss der Typinferenz-Algorithmus nach einer Typ-Einschränkung durch den Benutzer aufgerufen werden. Hieraus ergibt sich eine Änderung im Ablaufdiagramm der Typinferenz durch das Eclipse Plugin.

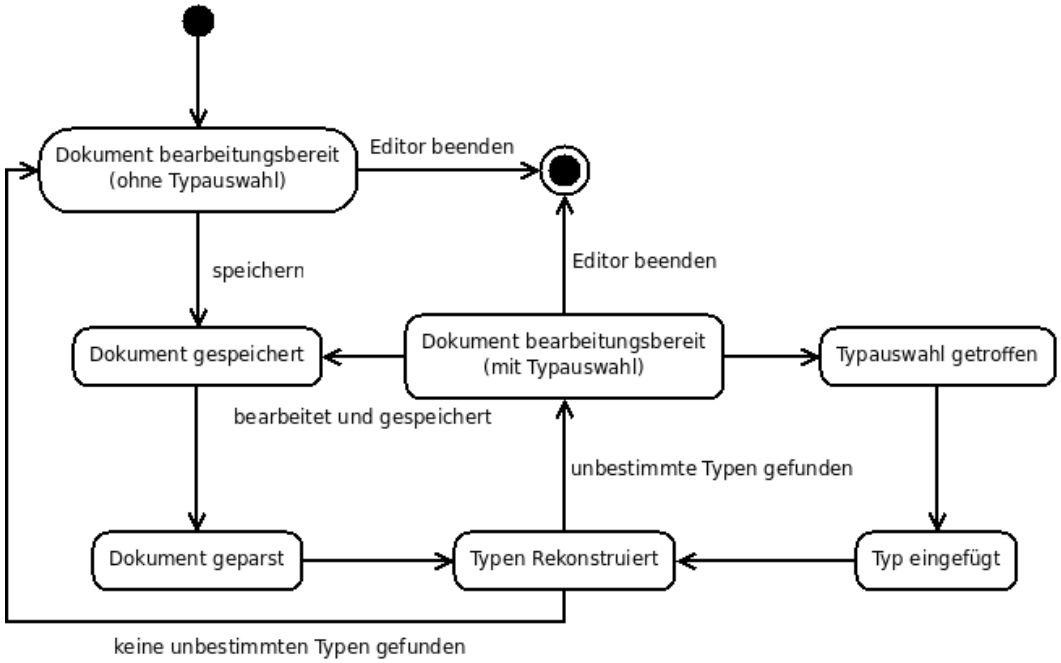


Illustration 14: Abstrakter Ablaufplan des Aufrufs der Typinferenz

Nach der Typ-Einschränkung einer Variablen wird der Typ sowohl im Quelltext, als auch in der logischen Repräsentation der Typ-Möglichkeiten für die Variable eingeschränkt. Dies ist nötig, um eine möglichst effiziente Ausführung des Typinferenz-Algorithmus zu gewährleisten. Würde die Einschränkung der Typen nur visuell für den Benutzer dargestellt, so müsste der komplette Quelltext neu geparsed und der Typinferenz-Algorithmus immer komplett durchlaufen werden. Dadurch würden Einbußen in der Geschwindigkeit der Ausführung entstehen, welches zu einer längeren Reaktionszeit führt. Bei einer grafischen Anwendungsoberfläche ist es allerdings sehr wichtig eine kurze Reaktionszeit zu bieten. Deswegen wird die Information der Auswahl der Typen redundant im Arbeitsspeicher gehalten. So muss der Quelltext bei einer Typ-Einschränkung nicht neu geparsed werden. Die Typ-Einschränkung wird direkt in die logische Repräsentation eingefügt und als Eingabe des Typinferenz-Algorithmus zusammen mit den, im vorherigen Aufruf errechneten, Abhängigkeiten gegeben. Dies spart nicht nur das Parsen des Quelltextes, sondern auch einige Iterationen im Typinferenz-Algorithmus.

6 Fazit

6.1 Rückblick

Diese Studienarbeit hat Einiges an der Architektur und Funktionalität des Eclipse Plugins verändert. Das Plugin wurde auf die Eclipse Version 3.4 migriert und ist damit bestens für die kommende Eclipse Version gerüstet. Die Integration in die Eclipse Update-Manager Struktur führt außerdem zu einem einfacheren Einstieg in die Benutzung des Plugins.

Als eine Konsequenz der geplanten Transformation in ein Open Source Projektes wurde das Logging des Plugins und des Typinferenz-Algorithmus auf eine gleiche Plattform gebracht. Die Wahl von log4j als Logging Plattform ist ein erster Schritt zu einem auf Standard-Bibliotheken basierenden und stabilen Quelltext.

Die Implementierung zusätzlicher Features, wie der Visualisierung der Typeinschränkung erweitert zudem noch die Funktionalität des Plugins. Dadurch wird es für den Benutzer einfacher sich mit dem Thema Typinferenz in Java zu beschäftigen.

6.2 Ausblick

Das aktuelle Eclipse Plugin ist äußerst nützlich, um Programme für den Typinferenz Compiler zu programmieren. Aber um eine komplette Integration in die Eclipse IDE zu gewährleisten fehlen noch einige Funktionalitäten. So wäre zum Beispiel der Start, einer vom Typinferenz compilierten, *.class* Datei in Eclipse ein weiterer Schritt in Richtung einer komfortablen und integrierten Entwicklungsumgebung.

Man sollte sich auch darüber Gedanken machen, inwiefern es vielleicht Sinn machen würde, den Typinferenz Compiler auf Basis bereits existierender Compiler Bibliotheken weiterzuführen. Hier wäre der Eclipse interne JDT Compiler sicher eine interessante Alternative. Dieser würde, vor allem auch durch die dadurch einfachere Implementierung von Debug Funktionalitäten, im Plugin sehr nützlich sein.

7 Referenzen

PLU05: Martin Plümicke, Type Inference in Generic Java, 2005

REI05: Felix Reichenbach, Erweiterung der semantischen Analyse des Java-Compilers, 2005

HAA04: Markus Haas, Weiterentwicklung der Java-Codegenerierung zur Ausführung von parametrisierten Datentypen., 2004

OTT04: Thomas Ott, Typinferenz in Java, 2004

BAE05: Joerg Baeuerle, Typinfernz in Java, 2005

HOL06: Timo Holzherr, Typinferenz in Java, 2006

LUE07: Arne Lüdtke, Java Typinferenz mit Wildcards, 2007

BUR07: Achim Burger, Erweiterte Typinferenz in Java 5.0, 2007

SCH07: Jürgen Schmiing, Migration einer Codegenerierungskomponente für einen Java-Compiler, 2007

MEL05: Markus Melzer, Integration der Java-Typinferenz in eine Programmierumgebung, 2005

HOM06: Thomas Homberger, Einbindung von Java Typinferenz in eine integrierte Entwicklungsumgebung, 2006

RCPCA: Eclipse Foundation, Commercial Rich client platform (RCP) applications, 2008, http://www.eclipse.org/community/rcpcp.php

EH3.4: Eclipse Foundation, Platform Plug-in Developer Guide, 2008, http://help.eclipse.org/stable/topic/org.eclipse.platform.doc.isv/guide/runtime_model_bundles.htm

EM31-32: Eclipse Foundation, Incompatibilities between Eclipse 3.1 and 3.2, http://help.eclipse.org/stable/topic/org.eclipse.platform.doc.isv/porting/3.2/incompatibilities.html

EM32-33: Eclipse Foundation, Incompatibilities between Eclipse 3.2 and 3.3, http://help.eclipse.org/stable/topic/org.eclipse.platform.doc.isv/porting/3.3/incompatibilities.html

ENT34M6a: Eclipse foundation, Eclipse 3.4 M6 News, 2008, http://download.eclipse.org/eclipse/downloads/drops/S-3.4M6a-200804091425/eclipse-news-M6.html

ENT34M7: Eclipse foundation, Eclipse 3.4 M7 News, 2008, http://download.eclipse.org/eclipse/downloads/drops/S-3.4M7-200805020100/eclipse-news-M7.html

8 Illustrationsverzeichnis

Illustration 1: Die Eclipse Plattform.....4
Illustration 2: Abhängigkeiten der Verschiedenen Bundle Definitionen.....5
Illustration 3: Plugins, Features und Update-Sites.....7
Illustration 4: Visualisierung der Abhängigkeiten des Plugins (GELB: Typinferenz Plugin, Orange: Directe Abhängigkeiten,.....11
Illustration 5: Klassendiagramm zur Integrierung der Syntaxhighlighting Funktion.....14
Illustration 6: Verhältnis der Log4j Komponenten.....18
Illustration 7: Plugin Abhängigkeiten in Bezug auf Log4J.....19
Illustration 8: Die drei Schritte zur Visualisierung: 21
Illustration 9: Fälle der Ersetzung bei einer lokalen Variable.....22
Illustration 10: Fälle der Ersetzung bei Methodenparametern.....23
Illustration 11: Fälle der Ersetzung bei einem Methodenrückgabewert.....23
Illustration 12: Klassendiagramm von JavMethod und JavSyntaxRelatedScanner.....24
Illustration 13: Beispiel zu Abhängigkeiten zwischen noch nicht bestimmten Variablen.....25
Illustration 14: Abstrakter Ablaufplan des Aufrufs der Typinferenz.....26
Illustration 15: Auswahl der Feature Update Möglichkeiten. "Search fot new features to install" wird gewählt.....30
Illustration 16: Liste der Update Sites. Diese kann von Eclipse Installation zu Eclipse Installation variieren.....31
Illustration 17: Dialog "Edit Locale Site" nachdem der Projekt Ordner ausgewählt wurde.....31
Illustration 18: Gefundene neue Komponenten. Der Titel variiert abhängig von dem vergebenen Namen der UpdateSite.....32
Illustration 19: Das License Agreement muss akzeptiert werden.....33
Illustration 20: Nur noch auf Finish beenden.....33
Illustration 21: Install / Update manager.....34
Illustration 22: Screenshot der Product Configuration mit geöffneten Kontextmenü des TypInference Plugins.....35
Illustration 23: Überblick über Plugins, Features und Update-Sites.....35

9 Anhang

9.1 Beschreibung der Installation/Aktualisierung/Deinstallation des Eclipse Plugins

Projekt aus dem Repository auschecken

Um das Plug-In in der aktuellen Version zu installieren, müssen die für die Installation notwendigen Dateien aus dem CVS geholt werden. Das Modul JavaCompilerEclipsePlugins/de.ba.horb.typeinference.plugin.updateSite beinhaltet diese. Um es mit Eclipse aus dem Repository zu holen begibt man sich in die CVS Repositoy Exploring Perspektive. Hier wählt man nun das Repository herbie..ba-horb.de aus und expandiert die HEAD Sektion. Nun sollte man JavaCompilerEclipsePlugins expandieren. Hier öfffnet man das Kontextmenü von de.ba.horb.typeinference.plugin.updateSite und wählt Check out As... aus. Den aufkommenden Dialog kann man mit Finish schließen.

Vor einem Update des Plugins muss das Projekt de.ba.horb.typeinference.plugin.updateSite auf den neusten Stand sein. Dies wird am einfachsten durch ein CVS update garantiert. Um dies zu machen ruft man im Kontextmenü des Projektes Team->Update auf.

Installation des Plug-Ins

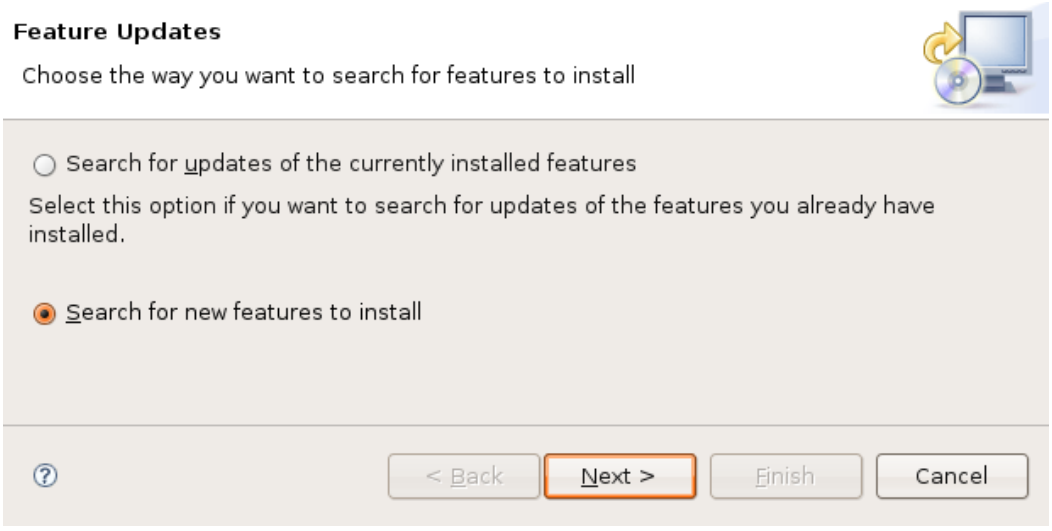


Illustration 15: Auswahl der Feature Update Möglichkeiten. "Search fot new features to install" wird gewählt.

In Eclipse werden Plug-Ins über *Help->Software Updates->Find and Install ...* installiert. So gehen wir auch bei diesem Plug-In vor. Die nachfolgende Reihe von Dialogen werden anhand von Screenshots beschrieben.

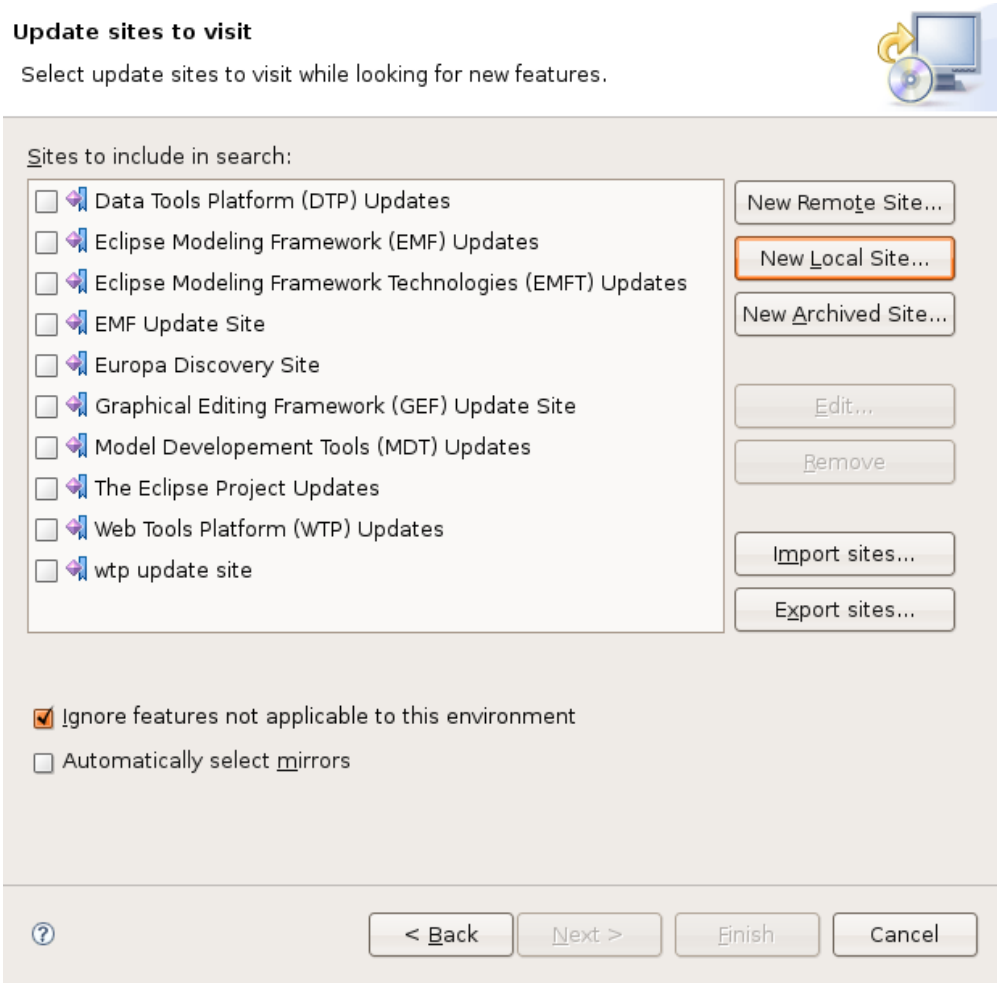


Illustration 16: Liste der Update Sites. Diese kann von Eclipse Installation zu Eclipse Installation variieren.

Um das Plug-In installieren zu können, müssen wir eine neue lokale Update Site hinzufügen (*New Local Site...*). Im aufkommenden Dialog muss der Ordner des aus dem CVS importierten Projektes angegeben werden. Er liegt im Verzeichnis des benutzten Workspaces.

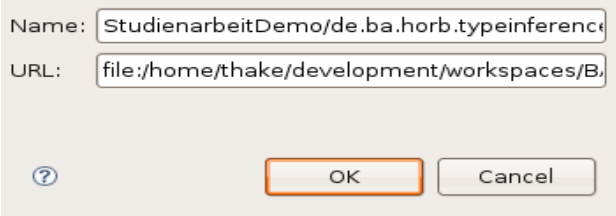


Illustration 17: Dialog "Edit Locale Site" nachdem der Projekt Ordner ausgewählt wurde.

Nun mit *OK* bestätigen und auf *Finish* klicken.

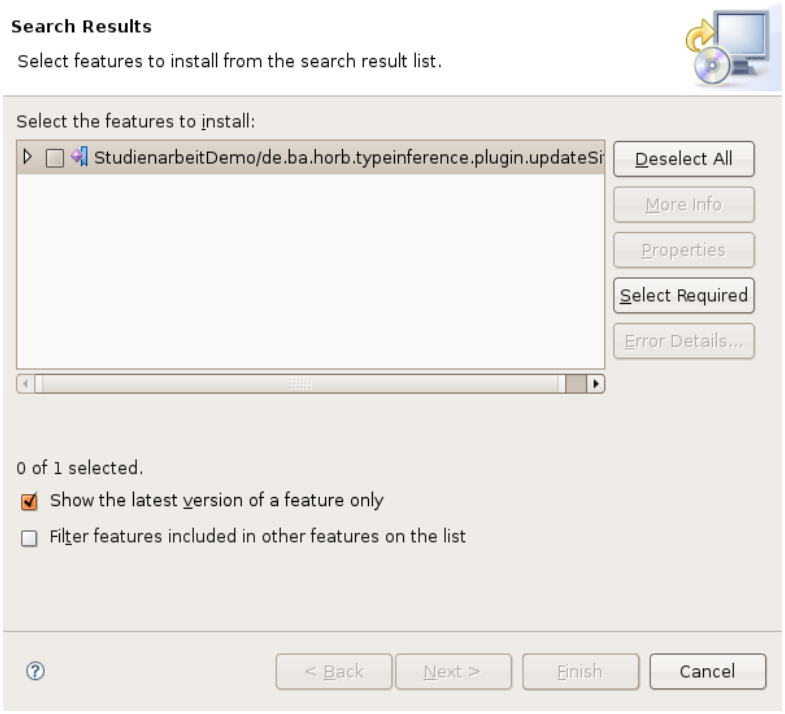


Illustration 18: Gefundene neue Komponenten. Der Titel variiert abhängig von dem vergebenen Namen der UpdateSite

Es sollte ein ähnliches Fenster erscheinen. Nun das einzig gefundene Feature selektieren und auf *Next* klicken.

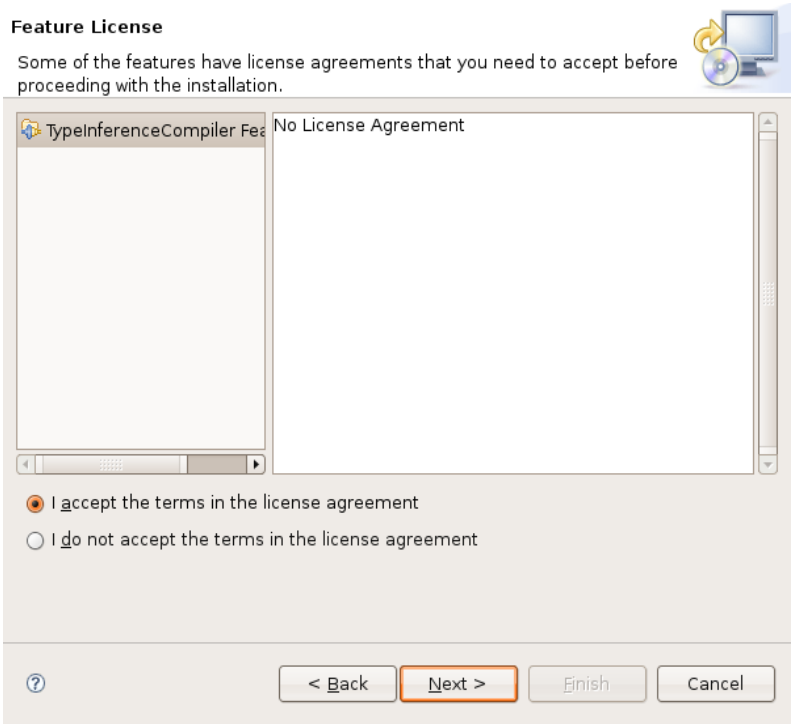


Illustration 19: Das License Agreement muss akzeptiert werden

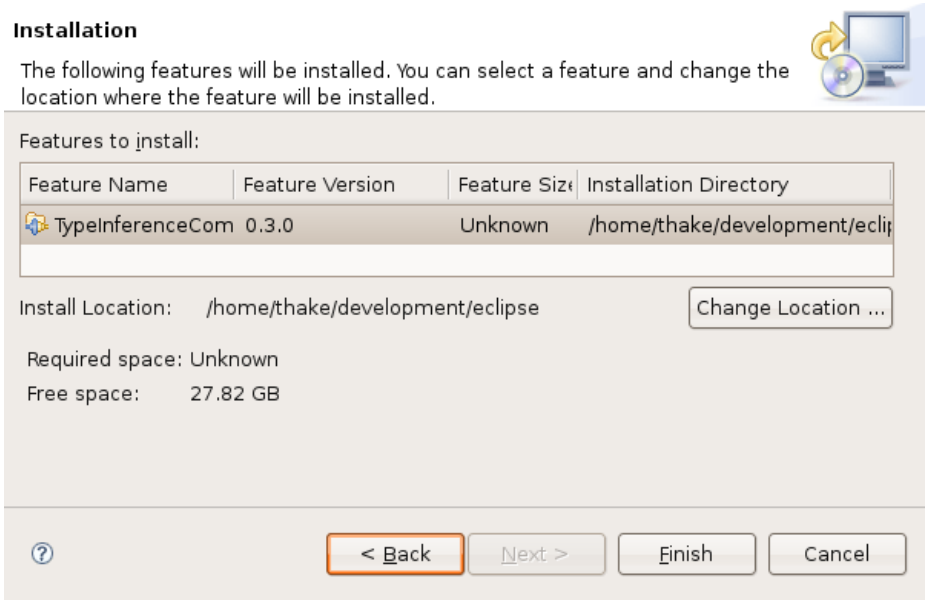


Illustration 20: Nur noch auf Finish beenden

Update des Plugins

Wenn das Plugin bereits installiert ist, aber eine neuere Version des Plugins zur Verfügung steht, kann das Plugin aktualisiert werden. Hierzu muss die Version des Projektes wie in Projekt aus dem Repository ausschecken beschrieben auf die neuste Version gebracht werden.

Ähnlich wie bei der Installation muss wieder der Install/Update Manager über *Help->Software Updates-> Find and install...* gestartet werden.

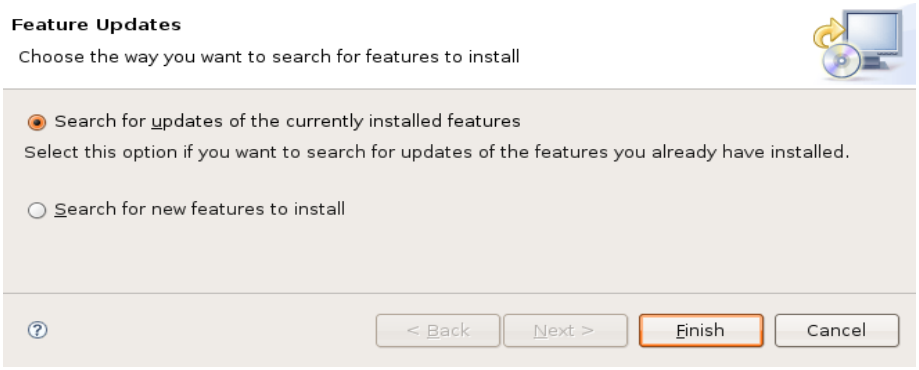


Illustration 21: Install / Update manager

Diesmal soll hier Search for updates of the currently installed features ausgewählt werden. Eclipse sucht nun automatisch nach Aktualisierungen der momentan installierten Plugins.

Die nachfolgenden Fenster sind ähnlich der von der Installation eines Plugins.

Deinstallation des Plugins

Um das Plugin zu deinstalliere genügen ein paar wenige Schritte.

Als erstes wird die *Product Configuration* aufgerufen. Diese findet man unter *Help->Manage Configuration*. Hier kann man alle installierten Plugins deaktivieren, aktivieren und deinstallieren.

Wenn man in der linken Liste nun das Kontextmenü des zu deinstallierenden Plugins öffnet und auf *Uninstall* klickt, wird das Plugin deinstalliert.

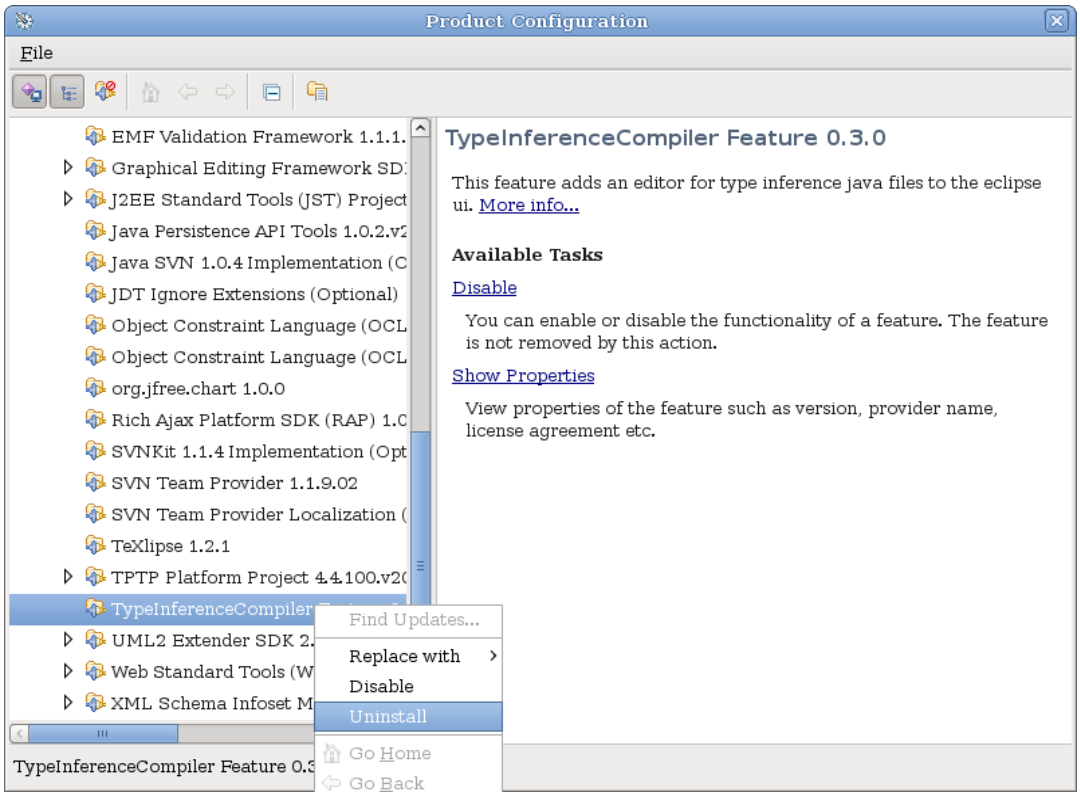


Illustration 22: Screenshot der Product Configuration mit geöffneten Kontextmenü des TypInference Plugins

9.2 Überblick über Plugins, Features und Update-Sites

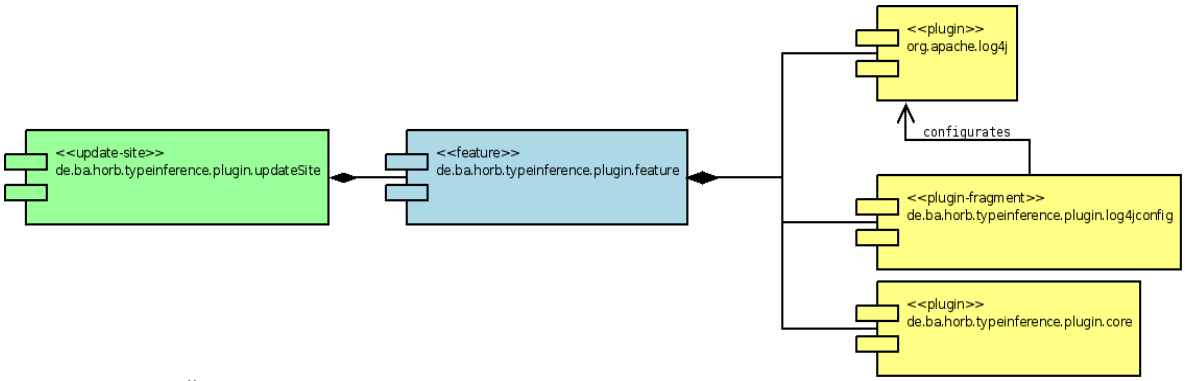


Illustration 23: Überblick über Plugins, Features und Update-Sites