Fern-Universität Hagen
Fachbereich Mathematik und Informatik
Lehrstuhl Programmiersysteme

Abschlussarbeit im Studiengang Master of Science im Fach Informatik

Sommersemester 2006

Design und Implementierung eines Eclipse-Plug-Ins zur Anzeige von möglichen Typgeneralisierungen im Quelltext

Autor:	Dipl.-Inf.(FH) Markus Bach
Matrikel-Nr.:	6931944
Adresse:	Schlossstraße 5
	91743 Unterschwaningen
Telefon:	09836 / 970323
E-Mail:	bach.markus@gmx.net
Betreuer:	Prof. Dr. Friedrich Steimann
	Dipl.-Inf. Florian Forster
Stand:	7. Februar 2007

Widmung

Für meine Eltern
Gisela und Werner Bach (†15. Februar 2005)

Inhalt

1. Einleitung **6**

2. Grundlagen der Typgeneralisierung **8**
2.1. Motivation und Ziele 8
2.2. Definition der Typinferenz 9
2.3. Typinferenzalgorithmen 10
2.4. Vorgehen und Tool-Unterstützung 11

3. Refactoring und Typgeneralisierung in Eclipse **14**
3.1. Refactoring 14
3.2. Typgeneralisierung 16
3.2.1. Generalize Declared Type Refactoring 16
3.2.2. Infer Type Refactoring 18

4. Das Declared Type Generalization Checker Plug-In **21**
4.1. Anforderungen und Vorgehen 21
4.2. Architektur des Plug-Ins 22
4.2.1. Quelltextanalyse mit Builder und AST-Visitor 23
4.2.2. Anbindung von Typinferenzalgorithmen 24
4.2.3. Erzeugung von Markierungen und Lösungshilfen 26
4.2.4. Konfiguration und Aufruf des Plug-Ins 27
4.3. Klassenstruktur und Packages 28
4.4. Implementierung 32
4.4.1. Hauptklasse und Manifest 32
4.4.2. Property-Page 35
4.4.3. Nature und Builder 37
4.4.4. Generalisierungsprüfungs-Interface und Adapter 40
4.4.5. Code-Prüfungs-Visitors 47

4.4.6. Problem-Marker . . . 49
4.4.7. Marker-Resolution . . . 50
4.4.8. Hilfsklassen . . . 52
4.5. Dokumentation . . . 53
4.5.1. Eclipse Online-Hilfe . . . 53
4.5.2. Quelltextdokumentation . . . 55
4.6. Deployment und Auslieferung . . . 56
4.6.1. Aufbau des Plug-In-Feature . . . 57
4.6.2. Installation per Update-Site . . . 58

5. Fallstudien beim Einsatz des Declared Type Generalization Checker Plug-Ins 60
5.1. Anwendungsszenarien . . . 60
5.2. Laufzeitverhalten . . . 62
5.3. Qualtität und Aussagekraft der Ergebnisse . . . 65
5.4. Bewertung der Fallstudie . . . 68

6. Fazit und Ausblick 70
6.1. Chancen und Risiken beim Einsatz . . . 70
6.2. Anbindung von Algorithmen zur Generalisierungsprüfung . . . 71
6.3. Parallele Prüfung mit mehreren Algorithmen . . . 72
6.4. Unterstützung der abgesetzten Prüfung . . . 73
6.5. Vorgaben für den Ausschluss von Deklarationselementen von der Prüfung 73

A. Die Eclipse-Plattform 76
A.1. Entwicklungsgeschichte . . . 76
A.2. Interne Struktur und Erweiterbarkeit . . . 77

B. Grundlagen der Plug-In-Entwicklung 80
B.1. Prinzipien . . . 80
B.2. Aufbau und Struktur von Plug-Ins . . . 83
B.2.1. Das Plug-In-Manifest . . . 83
B.2.2. Extensions und Extension-Points . . . 86
B.2.3. Abhängigkeiten zwischen Plug-Ins . . . 89
B.3. Plug-In-Development-Environment (PDE) . . . 91
B.4. Deployment von Plug-Ins . . . 93
B.5. Lebenszyklus . . . 95

C. Code-Audits beim Compiler-Lauf **98**
C.1. Strategie und Ziele 98
C.2. Grundlagen in Eclipse 100
C.2.1. Natures und Builders 100
C.2.2. Parsen des Abstract-Syntax-Tree (AST) 104
C.2.3. Problem-Marker 106
C.2.4. Marker-Resolution 108

D. Quelltexte des Plug-Ins **112**
D.1. Property-Page 112
D.2. Nature und Builder 117
D.2.1. Klasse DeclaredTypeGeneralizationCheckerNature 117
D.2.2. Klasse DeclaredTypeGeneralizationCheckerBuilder 118
D.3. Code-Prüfungs-Visitors 119
D.3.1. Klasse Checker 119
D.3.2. Klasse CheckerASTVisitor 124
D.3.3. Klasse CheckerResourceDeltaVisitor 126
D.4. Problem-Marker 127
D.5. Marker-Resolution 128
D.5.1. Klasse MarkerResolutionGenerator 128
D.5.2. Klasse GeneralizeDeclaredTypeMarkerResolution 130
D.5.3. Klasse InferTypeMarkerResolution 131
D.6. Hilfsklassen 132
D.6.1. Klasse Messages 132
D.6.2. Klasse ExtensionValidator 133

Abbildungsverzeichnis **135**

Tabellenverzeichnis **136**

Literaturverzeichnis **137**

Index **142**

Glossar **146**

Erklärung **150**

1. Einleitung

Landläufig versteht man unter Programmierung das algorithmische Lösen eines bestehenden Problems. Spinnt man diesen stark vereinfachten Gedanken weiter, so müsste das erklärte Ziel dieses Prozesses sein, das Problem schnell und effizient, d.h. mit möglichst geringem Aufwand, zu lösen. Das Ziel der Programmierung ist es aber nicht einfach nur, den augenscheinlich kürzesten Weg zum Ziel zu gehen, sondern weitere unerlässliche Aspekte wie u.a. Wartbarkeit, Erweiterbarkeit und Pflegbarkeit durch Mittel der Modularisierung umzusetzen. Umso wichtiger wird dieses Vorgehen, je größer Softwareprojekte werden, denn ohne klare innere Struktur sind große Projekte nicht handhabbar. Deshalb werden Softwareprojekte in Komponenten zerlegt, die gekapselt spezialisierte Dienste erbringen und über definierte Schnittstellen anderen Komponenten des Systems zur Verfügung stellen. Grundlage hierfür ist, dass die Komponenten in sich abgeschlossen sind, d.h., möglichst wenig Abhängigkeitsbeziehungen untereinander bestehen.
Bricht man den Begriff einer Komponente auf Klassenebene in einem objektorientierten Programm herunter, bedeutet dies, dass auch hier intensiv mit Schnittstellen, sprich Interfaces, gearbeitet werden muss. Verwendet man minimale Interfaces für Variablendeklaration, dient dies der Entkopplung von Klassen, was zu flexibleren Programmen führt.[1] Deshalb lautet auch eines der wichtigsten Mottos der objektorientierten Softwareentwicklung: 'Programmieren auf eine Schnittstelle hin, nicht auf eine Implementierung'.[2]
Ziel dieser Arbeit soll es deshalb sein, das Design und die Implementierung eines Plug-Ins für die Programmierumgebung Eclipse zu beschreiben, welches den Softwareentwickler unterstützt, oben genanntes Motto bei der täglichen Arbeit effizient anzuwenden. Hierbei bedeutet Effizienz auch Einfachheit. Entwicklern soll ein Werkzeug an die Hand gegeben werden, das weitgehend automatisiert während des Entwicklungsprozesses die Quelltexte analysiert und Hinweise auf zu konkret gewählte Typen

[1] nach: Steimann u. a. (2006)
[2] aus: Gamma u. a. (2001)

in Deklarationen gibt. Der Fokus liegt dabei darauf, den Entwickler während des Entwicklungsprozesses nicht nur auf die mögliche Verwendbarkeit von Interfaces und somit von verallgemeinerten Typen hinzuweisen, sondern ihn bei der Findung und Erstellung geeigneter Typen zu unterstützen. Zurückgegriffen wird hierbei auf bestehende Algorithmen und Softwarekomponenten im Rahmen der Eclipse-Plattform. Das Plug-In wird so offen gestaltet sein, dass jederzeit andere Algorithmen zur Typinferenz angebunden werden können. In der ersten Ausbaustufe des Plug-Ins sind Adapter für die Refactorings Generalize Declared Type und Infer Type implementiert.

Das erste Kapitel dient der Einführung in diese Arbeit und in die Problemstellung. Im zweiten Kapitel werden die Grundlagen der Typgeneralisierung und der Typinferenz erläutert. Im Kapitel 3 wird auf die Themen Refactoring und Typgeneralisierung in der Entwicklungsumgebung Eclipse eingegangen. In diesem Zuge werden die Refactorings Generalize Declared Type und Infer Type näher erläutert. Kapitel 4 beschreibt neben den gestellten Anforderungen und der architektonischen Umsetzung auch die Implementierung des Plug-Ins. Im darauf folgenden Kapitel 5 werden mehrere Anwendungsfälle für den Einsatz des Plug-Ins und sein Verhalten in Projekten verschiedener Größe beschrieben und diskutiert. Das abschließende Kapitel 6 stellt eine kritische Betrachtung der gewonnenen Erkenntnisse dar und ermöglicht einen Ausblick auf weitere Entwicklungen.

In den Anhängen A und B wird auf die Architektur der Eclipse-Plattform eingegangen und detailliert beschrieben, wie eigene Erweiterungen implementiert werden. Anhang C setzt die Einführung in die Erweiterung der Entwicklungsumgebung fort, jedoch werden die verwendeten Elemente für die Durchführung von Code-Audits als Grundlage für diese Arbeit allgemeiner erklärt. Diese Anhänge dienen dem tieferen Verständnis der Implementierung. Im Anhang D sind Auszüge aus den Quelltexten des Plug-Ins zu finden. Diese wurden aus Platzgründen an weniger relevanten Stellen gekürzt.

2. Grundlagen der Typgeneralisierung

Für die Entwicklung modularer Systeme ist es erforderlich, Klassenstrukturen zu entkoppeln. Erreicht wird dies, indem für Variablendeklarationen keine Klassen, sondern abstrakte, im Idealfall kontextabhängige, Interfaces verwendet werden. Dieses Vorgehen wird im Folgenden als Typgeneralisierung bezeichnet. Der bereits einleitend erwähnten Forderung 'Programmieren auf eine Schnittstelle hin, nicht auf eine Implementierung' wird dadurch genüge getan.

2.1. Motivation und Ziele

Obwohl oben genanntes Motto aus Gamma u. a. (2001) zum Einsatz von Interfaces breite Zustimmung findet und sein Beitrag zur Entwicklung lose gekoppelter Systeme unbestritten ist, zeigen Studien, dass nur etwa jede fünfte Variable ein Interface als Typ hat.[1] Einen Überblick über die Entwicklung zwischen den verschiedenen Versionen des Java-JDK gibt die Studie Steimann und Mayer (2005). Diese stellt fest, dass das Verhältnis von Klassen- zu Interfacedeklarationen von Variablen vor der Einführung von Java 2 noch bei 9 : 1 lag. Wenn man zusätzlich noch betrachtet, dass das Verhältnis von Anzahl der Klassen zu Anzahl der Interfaces im Java-JDK annähernd konstant blieb, stellt das erreichte Verhältnis von 5 : 1 eine enorme Verbesserung dar. Zurückzuführen ist die Verbesserung des Verhältnisses größtenteils auf die Einführung des Collections-Frameworks und des Swing-Frameworks. In Steimann und Mayer (2005) wird deshalb gefolgert, dass sich die Verwendung von Interfaces gefestigt hat, wenn auch hauptsächlich bei den Entwicklern des Java-JDK selbst. Die Gründe, warum sich Interfaces bei Variablendeklarationen in Java-Projekten noch nicht vollständig durchgesetzt haben, liegen oftmals darin begründet, dass es pragmatisch ist, zuerst den Kontext, in dem das Interface benutzt werden soll, zu pro-

[1] nach: Steimann u. a. (2003)

grammieren und dann das Interface aus dem im Kontext benutzten Protokoll zu erstellen.[2] Es ist offensichtlich, dass die Einführung des Interfaces somit in den Bereich des Refactorings verschoben wird. Das Refactoring muss nach der eigentlichen Programmierarbeit bewusst durchgeführt werden, um die angestrebte Entkopplung zu erreichen. Oftmals ist das Vorgehen aber so, dass einmal getesteter und lauffähiger Quelltext, aus Zeitmangel oder sonstigen Gründen, nur ungern wieder geändert wird. Erst mit dem Aufkommen von agilen Vorgehensmodellen in der Softwareentwicklung wurde Refactoring zum festen Bestandteil des Entwicklungsprozesses und somit auch zeitlich eingeplant.
Messbar ist die Verwendung und die Qualität von Interfaces mit Hilfe geeigneter Metriken. Dabei soll mit einer Metrik ein Aspekt in einem Programm in Zahlen ausgedrückt werden. Somit liefern Metriken, in gleicher Weise auf verschiedene Programme angewendet, vergleichbare Werte, mit deren Hilfe Abschätzungen und Aussagen getroffen werden können. In Steimann u. a. (2003) werden Metriken vorgestellt, mit denen sich Werte für die Allgemeinheit und die Popularität von Interfaces bestimmen lassen:

- Allgemeinheit: 'The more classes implementing an interface, the more general this interface may be assumed to be'
- Popularität: 'Popularity of an interface counts the number of variables declared with that interface as their type'

Weitere Metriken über die Verwendung von Interfaces werden in Mayer (2003) vorgestellt. Darüber hinaus werden die Ergebnisse der Anwendung dieser Metriken auf das Java-JDK präsentiert.

2.2. Definition der Typinferenz

Der Begriff Typinferenz lässt sich vom lateinischen Wort inferre[3] ableiten, demnach handelt es sich dabei wörtlich um Typableitung. Zu einer Klasse lassen sich alle Supertypen bestimmen, welche die gegebene Klasse erweitert, bzw. die sie implementiert. Exemplarisch sei in Abbildung 2.1 die Typhierarchie der Klasse *Vector* dargestellt. Klassen sind mit C, abstrakte Klassen mit C^A und Interfaces mit I gekennzeichnet.

[2] nach: Steimann u. a. (2003)
[3] deutsch: Deduktion, Folgerung oder Ableitung

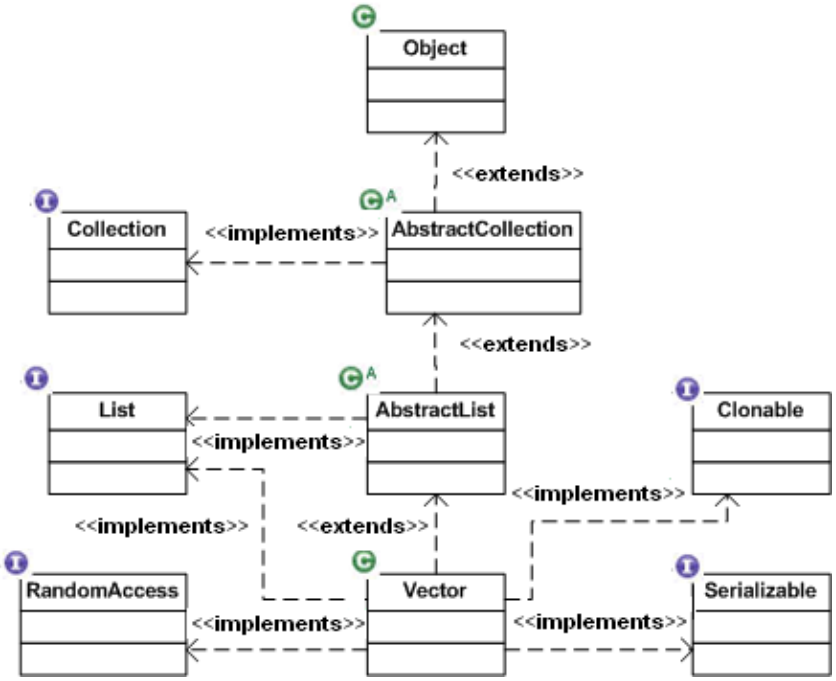

Abbildung 2.1.: Typhierarchie der Java-Klasse Vector

Mittels Typinferenz ließe sich für eine Deklaration eines Elements (Variable etc.) mit der Klasse *Vector* ein alternativer Typ berechnen, der nur die Methoden enthalten würde, die von dem Element unter Berücksichtigung der Typregeln bei Zuweisungen gebraucht werden. Ein solcher Typ ist maximal generalisiert, d.h. es gibt keinen Typ mit weniger Elementen, der ebenfalls in der Deklaration einsetzbar wäre. Der berechnete Typ kann bereits ein Element der Typhierarchie sein oder aber ein neuer Typ, der dann eingeführt werden könnte. Für eine formale Beschreibung der Typinferenz sei hier auf die Ausführungen in Steimann u. a. (2006) verwiesen.

2.3. Typinferenzalgorithmen

Die Algorithmen für die Typinferenz lassen sich in zwei Gruppen einteilen:

- zum einen in Algorithmen, die auf Type Constraints,
- und zum anderen in Algorithmen, die auf Datenflussanalyse basieren.

Beide Verfahren können sowohl mit statischer als auch dynamischer Programmanalyse angewendet werden. Für die Grundlagen der Programmanalyse sei hier auf Nielson u. a. (2005) verwiesen, außerdem beschreibt Nielson u. a. (2005) detailliert die Bildung von Type Constraints und die Durchführung der Datenflussanalyse.
Eine Beschreibung der Umsetzung des Typinferenz-Algorithmus beim Use Supertype Where Possible Refactoring von Eclipse, der auf Type Constraints setzt, findet sich in Tip u. a. (2003). Einen Überblick und eine Zusammenfassung über Typinferenz-Algorithmen gibt außerdem Steimann u. a. (2006).

2.4. Vorgehen und Tool-Unterstützung

Das folgende kurze Java-Beispiel zeigt die enge Kopplung von Klassen. Die Kopplung ergibt sich, da in den Klassen *StringContext* und *IntContext* jeweils eine Variable vom Typ *Formatter* deklariert wird. Beide Kontexte können deshalb alle Methoden aus *Formatter* aufrufen, obwohl sie in ihren Kontexten diese nicht benötigen. Darüber hinaus sind sie durch die Deklaration mit der Klasse direkt von der konkreten Implementierung dieser abhängig:

```
class Formatter {
    String formatString (String stringValue) { ... }
    String formatInt (int integerValue) { ... }
}
class StringContext {
    void doIt() {
        Formatter f = new Formatter ();
        String formattedString = f.formatString ("...");
    }
}
class IntContext {
    void doIt() {
        Formatter f = new Formatter ();
        String formattedInt = f.formatInt (...);
    }
}
```

Ein erster Schritt zur Entkopplung ist die Einführung der kontextspezifischen und minimalen Interfaces *IStringFormatter* und *IIntFormatter*, die von *Formatter* implementiert werden und nur die Methoden enthalten, die für den entsprechenden Kontext benötigt werden. In der Typhierarchie sind die Interfaces Supertypen von *Formatter* und können stattdessen in Variablemdeklarationen verwendet werden. Das obige Beispiel wurde im Folgenden entsprechend angepasst:

```
interface IStringFormatter {
    String formatString(String stringValue);
}
interface IIntFormatter {
    String formatInt(String integerValue);
}
class Formatter implements IStringFormatter, IIntFormatter {
    String formatString (String stringValue) { ... }
    String formatInt (int integerValue) { ... }
}
class StringContext {
    void doIt() {
        IStringFormatter f = new Formatter ();
        String formattedString = f.formatString ("...");
    }
}
class IntContext {
    void doIt() {
        IIntFormatter f = new Formatter ();
        String formattedInt = f.formatInt (...);
    }
}
```

Der Grad der Kopplung wurde durch Verwendung der Interfaces in den Variablendeklarationen reduziert. Eine gewisse Kopplung ist bei der Instanziierung der Variablen weiterhin gegeben, da hier die konkrete Klasse *Formatter* noch auftritt. Diese Kopplung ließe sich z.B. durch Verwendung von Dependency Injection[4] noch beheben.
Für das oben gezeigte Refactoring bieten die meisten Entwicklungsumgebungen ge-

[4]Für eine detaillierte Betrachtung sei auf Fowler (2004) verwiesen.

eignete Toolunterstüzung. In Eclipse leistet dies das Extract Interface[5] Refactoring, das für eine Klasse neue Interfaces erzeugen kann, die nur ausgewählte Methoden enthalten. Weiterhin stehen für die Typgeneralisierung noch die Refactorings Generalize Declared Type und Use Supertype Where Possible zur Verfügung. Diese unterstützen bei der Suche nach einem passenden Supertypen für eine Variablendeklaration bzw. führen die Ersetzung aller bisherigen Deklarationen geeignet durch. Ähnliche Refactorings finden sich auch in der Entwicklungsumgebung IntelliJ IDEA unter den Namen Extract Interface und Use Interface Where Possible. Außerdem helfen die Refactorings Pull Members Up und Push Members Down Interfaces minimal zu gestalten.[6]

'Obwohl diese Refactorings die Entwicklung lose gekoppelter Programme unterstützen, bleiben zwei wichtige Entscheidungen, nämlich die Auswahl des am besten passenden Supertyps – bei Use Supertype Where Possible und Generalize Type – und die Auswahl der Methoden, welche ein Interface enthält – bei Extract Interface –, im Verantwortungsbereich des Entwicklers.'[7]

[5]Für Details sei auf Tip u. a. (2003) verwiesen.
[6]siehe Produktpräsentation: JetBrains (2006)
[7]aus: Forster (2006)

3. Refactoring und Typgeneralisierung in Eclipse

Da das in dieser Arbeit vorgestellte Plug-In für die Eclipse-Plattform auf die Refactorings Generalize Declared Type und Infer Type aufsetzt, werden diese beiden Refactorings im folgenden Abschnitt näher betrachtet. Allerdings wird zunächst generell in das Thema Refactoring in Eclipse eingeführt, u.a. um die beiden verwendeten Refactorings einordnen zu können.

3.1. Refactoring

Refactoring ist nicht erst mit dem Aufkommen agiler Vorgehensmodelle in der Softwareentwicklung entstanden. Diese Modelle machen Refactoring gewissermaßen nur hoffähig. Zuvor war Refactoring in den strukturierten Vorgehensmodellen einigermaßen verpönt, denn 'das Design hat schließlich nach allen gängigen Ansätzen vor der Implementierung zu erfolgen, die Notwendigkeit ergibt sich aber ... aus der Praxis'[1]. Refactoring ist dabei ein probates Mittel, um der Software-Fäulnis zu entgegnen, und soll somit dazu beitragen, Wartbarkeit, Lesbarkeit und Struktur der Software zu verbessern bzw. wieder an die vorgegebene Architektur anzupassen. Die Notwendigkeit für Refactoring ergibt sich auch aus einer ständigen Verschiebung von Anforderungen. Statistische Erhebungen zeigen, dass die sogenannte Requirements drift im Mittel ein Prozent der Anforderungen pro Monat beträgt[2]. Refactoring ist somit im agilen und strukturierten Entwicklungsablauf unbestritten, deshalb beschäftigen sich auch die meisten Hersteller von Entwicklungsumgebungen mit diesem Thema und integrieren Refactorings in ihre Systeme[3]. Dies nicht nur aus dem Grund, dass viele Refactorings

[1] aus: Steimann u. a. (2005)
[2] nach: Steimann u. a. (2005)
[3] siehe Abschnitt 2.4

stereotype Tätigkeiten, wie z.B. Ändern eines Klassennamens in allen Variablendeklarationen, sind, sondern vielmehr, da Refactorings tiefgreifende Änderungen in der Softwarestruktur bedeuten können. Mit reiner Handarbeit sind diese Änderungen fehleranfällig oder könnten wieder der Bequemlichkeit des Programmierers zum Opfer fallen.
In Eclipse stehen dem Entwickler grundsätzlich drei Klassen von Refactorings zur Verfügung. Die erste Klasse beeinflusst die physische Struktur des Quelltextes, z.B. indem Klassen, Methoden oder Variablen umbenannt oder verschoben werden können. In der zweiten Klasse sind Refactorings zusammengefasst, die die Struktur auf Klassenebene beeinflussen, wie z.B. das Extrahieren eines Interfaces. Refactorings, die sich auf die Struktur innerhalb einer Klasse auswirken, bilden die dritte Klasse, ein Vetreter ist hier das Extrahieren einer Methode. In Tabelle 3.1 sind alle Refactorings, die Eclipse in der Version 3 standardmäßig anbietet, namentlich aufgeführt und der entsprechenden Klasse zugeteilt. Für eine detaillierte Beschreibung der Reafctorings sei auf Enns (2004) verwiesen, nur das Generalize Declared Type Refactoring wird im folgenden Abschnitt 3.2.1 im Rahmen der Typgeneralisierung genauer betrachtet. Eine Entwicklungsumgebung kann nicht alle bekannten Refactorings implementieren, deshalb stellt die Implementierung von Eclipse nur einen Auszug der wichtigsten und am häufigsten benutzten Refactorings dar. Als optionale Plug-Ins stehen noch weitere Refactorings zur Verfügung, wie z.B. das Infer Type Refactoring, das später in Abschnitt 3.2.2 noch genauer betrachtet wird.
Für eine weiterführende Diskussion über die Refactorings in Eclipse hinaus sei auf Steimann u. a. (2005) verwiesen. Ein umfassender Katalog mit Refactorings ist bei Fowler (2006) zu finden.

Refactorings die physische Struktur betreffend
Rename
Move
Change Method Signature
Convert Anonymous Class to Nested
Move Member Type to New File
Refactorings auf Klassenebene
Push Down
Pull Up
Extract Interface
Generalize Declared Type
Use Supertype Where Possible

Refactorings für die interne Struktur einer Klasse
Inline
Extract Method
Extract Local Variable
Extract Constant
Introduce Parameter
Introduce Factory
Encapsulate Field

Tabelle 3.1.: Refactorings in Eclipse

3.2. Typgeneralisierung

Für die Typgeneralisierung stehen in Eclipse die Refactorings Extract Interface, Generalize Declared Type und Use Supertype Where Possible zur Verfügung. Als Plug-In ist außerdem das Infer Type Refactoring installierbar.
In den folgenden Abschnitten wird auf die Refactorings Generalize Declared Type und Infer Type detailliert eingegangen und ihre Arbeitsweise beschrieben. Hierzu gehört insbesondere auch die Definition ihrer Vorbedingungen, die erfüllt sein müssen, damit garantiert ist, dass sich das refaktorierte Programm noch genauso verhält wie zuvor. Die Vorbedingungen beruhen auf komplexen Type Constraints[4], die nicht Teil dieser Arbeit sind; der Leser sei jedoch auf Tip u. a. (2003) verwiesen. Darüber hinaus werden die Grundlagen zu den eingangs genannten Eclipse-Refactorings auch in Tip u. a. (2003) beschrieben. Das Infer Type Refactoring wird ausführlich in Steimann u. a. (2006) behandelt.

3.2.1. Generalize Declared Type Refactoring

Das Generalize Declared Type Refactoring bestimmt für eine Deklaration mit einem Typ alle möglichen Supertypen des verwendeten Typs. Hierzu gehören nicht nur konkrete Klassen, sondern auch abstrakte Klassen und Interfaces. Hierbei wendet das Refactoring keine Typinferenz an, um neue Typen zu berechnen – es kann nur auf bereits bestehende Typen zurückgreifen. Das Refactoring kann auf die Deklarationen

[4] Allgemein zum Thema Type Constraints sei auf Nielson u. a. (2005) verwiesen.

von Feldern, lokalen Variablen, Methodenparametern und Rückgabewerten angewendet werden. Es müssen allerdings einige Vorbedingungen eingehalten werden, damit das Refactoring auch durchgeführt werden kann. So darf es sich bei der Deklaration nicht um

- eine Array-Deklaration,
- einen primitiven Typ (z.B. *int*) oder
- einen Aufzählungstyp *(enum)* handeln.

Außerdem darf als Typ

- keine innere Klasse oder
- ein parameterisierter Typ verwendet worden sein.

Darüber hinaus kann das Refactoring nicht angewendet werden bei

- überladenen Methoden oder
- bei Mehrfachdeklarationen von Variablen[5].

Diese Vorbedingungen werden beim Aufruf des Refactorings unmittelbar geprüft, so dass die Typinferenz im Fall einer nicht erfüllten Bedingung gar nicht erst ausgeführt wird. Als Ergebnis liefert das Refactoring eine Liste der für die Deklaration in Frage kommenden Supertypen, in der auch die Hierarchie der Typen erkennbar ist. In dieser Liste kann der Entwickler einen alternativen Typ auswählen, der ansatt des bisherigen Typs im Quelltext eingefügt wird. Nachfolgend wird das Refactoring auf das Programmbeispiel aus Abschnitt 2.4 angewendet. Für die Deklaration einer Variable vom Typ *String* ergibt sich die in Abbildung 3.1 gezeigte Supertyphierarchie. In dieser Supertyphierarchie werden grundsätzlich alle gefundenen Typen dargestellt, um einen Überblick zu ermöglichen. Typen, die nicht alternativ verwendet werden können, sind ausgegraut. Nachteilig bei diesem Refactoring ist, dass die Entscheidung, welcher allgemeinere Typ verwendet werden soll, allein dem Entwickler obliegt. Das Refactoring gibt nur in soweit Hilfestellung, als dass es unmögliche Typen ausgraut,

[5] z.B.: *String stringVariable1, stringVariable2;*

es macht darüber hinaus keine Vorschläge, welcher Typ am günstigsten erscheint. Allerdings ist auch nicht immer gewährleistet, dass ein alternativer Typ überhaupt zur Verfügung steht. Ist kein solcher Typ vorhanden, muss dies der Entwickler erkennen und gegebenenfalls zunächst mit dem Extract Interface Refactoring einen Typ schaffen.

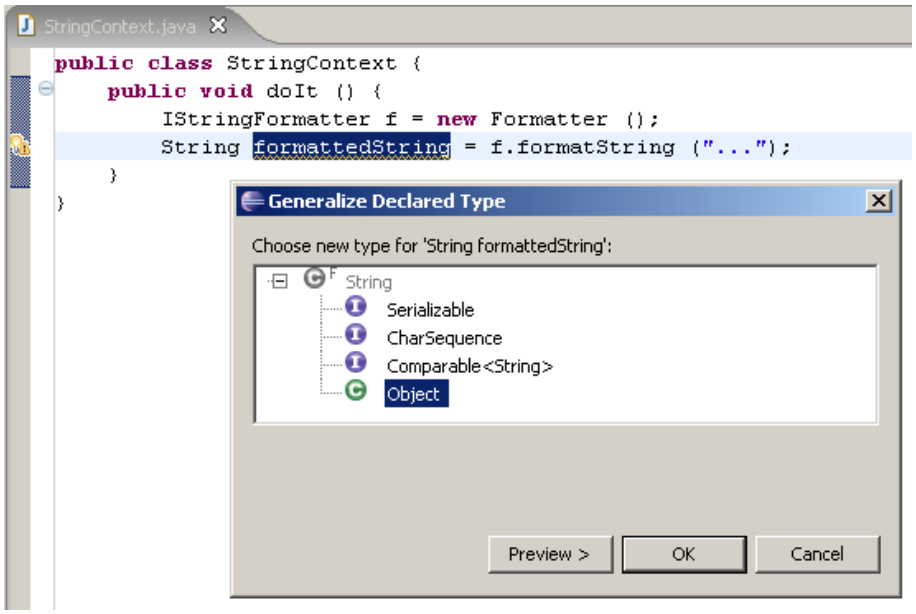

Abbildung 3.1.: Anwendung des Generalize Declared Type Refactorings

3.2.2. Infer Type Refactoring

Das Infer Type Refactoring ergänzt die Refactorings von Eclipse um die Funktionalität, für eine Referenz automatisch einen geeigneten Typ zu berechnen, der auch die Anforderung der Minimalität[6], die bereits in Kapitel 1 gefordert wurde, erfüllt. Infer Type nimmt dem Entwickler dabei die Entscheidungen ab, welcher alternative (evtl. neu hinzuzufügende) Typ am besten im gegebenen Kontext passen würde und welche Methoden er enthalten müsste, um minimal zu sein. Die theoretischen Grundlagen des Refactorings und des zugrundeliegenden Typinferenzalgorithmus werden in Steimann u. a. (2006) ausführlich behandelt und können dort nachgelesen werden. Die Ausführung von Infer Type ist auch an einige Vorbedingungen geknüpft. So ist Infer

[6]Ein Typ wird im folgenden minimal genannt, wenn er maximal generalisiert ist, d.h., wenn nach den Regeln des Typsystems von Java keine weiteren Elemente entfernt werden dürfen.

Type nicht auf jede Typdeklaration anwendbar, insbesondere nicht für Deklarationen mit Typen aus dem Java-JDK (z.B. *String*), keine primitiven Typen (z.B. *int*), Annotationen, Aufzählungstypen (*enum*), Arrays oder parameterisierte Typen. Außerdem ist Infer Type nicht anwendbar auf Deklarationen mit einer inneren Klasse als Typ.
Die Arbeitsweise wird verdeutlicht, indem im Folgenden das Refactoring auf das Programmbeispiel aus Abschnitt 2.4 angewendet wird, wie dies die folgende Abbildung 3.2 zeigt.

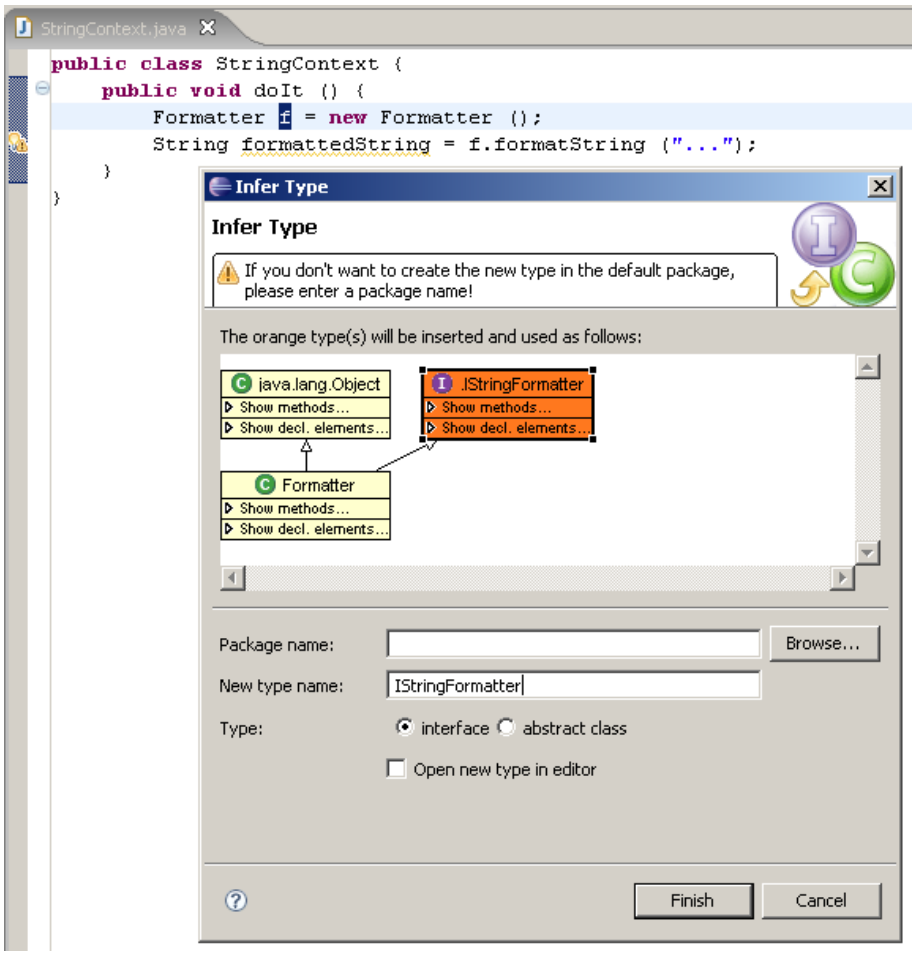

Abbildung 3.2.: Anwendung des Infer Type Refactorings

Im Gegensatz zum Generalize Declared Type Refactoring kann Infer Type auf die Deklaration der Variable vom Typ *Formatter* angewendet werden. Generalize Declared Type würde dafür kein brauchbares Ergebnis liefern, da der Typ *Formatter* nur vom generellen Typ *Object* abgeleitet ist. Generalize Declared Type kann nur auf beste-

hende Typen zurückgreifen, Infer Type kann einen neuen Typ berechnen und an den entsprechenden Stellen im Quelltext einfügen. Infer Type berechnet in diesem Fall für die Deklaration in der Klasse *StringContext* ein neues Interface, welches nur die Methoden die in diesem Kontext tatsächlich aufgerufen werden enthält. Neben der Änderung der Deklaration führt Infer Type das neue Interface in das entsprechende Package ein und ergänzt die *implements*-Anweisung für das Interface in der Klasse *Formatter*. Man beachte, dass Infer Type auch in wesentlich komplexeren Fällen minimale Interfaces berechnen kann; die Bestimmung kann dann allerdings ziemlich zeitaufwändig werden. Um das gleiche Ergebnis der Entkopplung wie im Beispiel aus Abschnitt 2.4 zu bekommen, muss Infer Type noch auf die Deklaration der Variable vom Typ *Formatter* in der Klasse *IntContext* angewendet werden. Hierbei ergibt sich nachfolgende neue Struktur. Das erzeugte Interface für den zweiten Kontext ist auch hier orange gekennzeichnet.

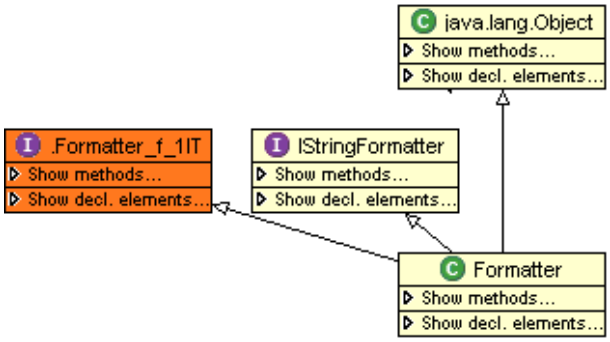

Abbildung 3.3.: Anwendung des Refactorings auf einen bereits bearbeiteten Typ

Ebenso wird in diesem Fall die Deklaration entsprechend an das neue Interface angepasst, das Interface selbst erzeugt und als weitere *implements*-Anweisung in der Klasse *Formatter* ergänzt.
Infer Type schlägt durch seine Arbeitsweise eine Brücke zwischen den bestehenden Eclipse Refactorings, indem es deren Funktionalitäten vereinigt und automatisiert. Bei den bisherigen Refactorings wurde dem Entwickler zwar die stereotype Handarbeit bei der Einführung eines allgemeineren Typs im Quelltext abgenommen, aber mit den wichtigen Entscheidungen über Eignung und Minimalität eines alternativen Typs blieb er auf sich gestellt. Dennoch nimmt ihm Infer Type eine Entscheidung auch nicht ab, nämlich auf welche Deklarationen es anzuwenden ist, um ein entkoppeltes System zu erhalten.

4. Das Declared Type Generalization Checker Plug-In

In den folgenden Abschnitten wird die konkrete Struktur und Implementierung des Plug-Ins beschrieben. In den Darstellungen wird konkret auf die Eigenheiten von Architektur und Umsetzung des Declared Type Generalization Checker Plug-Ins eingegangen. Bevor aber auf Implementierungsebene gewechselt wird, findet sich in diesem Kapitel zunächst die Definition der Anforderungen, die der Entwicklung vorausging.

4.1. Anforderungen und Vorgehen

Der vorliegende Abschnitt soll die Anforderungen beschreiben, die vor Projektbeginn an das Declared Type Generalization Checker Plug-In gestellt wurden. Anforderungen definieren dabei die Natur der späteren konkreten Umsetzung und geben eine Richtung für bestimmte Aspekte vor. Folgende Anforderungen wurden für das Projekt definiert:

- Das Projekt soll als Plug-In für die Entwicklungsumgebung Eclipse realisiert werden.
- Das Plug-In soll jede Variablendeklaration in einem Java-Projekt hinsichtlich der Allgemeinheit des verwendeten Typs prüfen und eine Warnung ausgeben, wenn der Typ verallgemeinert werden kann.
- Die Prüfung soll optional möglich sein, also durch den Entwickler zu- oder abschaltbar, da die Prüfungen unter Umständen sehr zeitintensiv sein können.
- Als Algorithmus für die Prüfung wird sowohl das Infer Type Plug-In als auch das Generalize Declared Type Refactoring verwendet.

- Die Anbindung der Prüf-Plug-Ins soll so erfolgen, dass auswählbar ist, welcher Algorithmus verwendet wird.
- Dabei soll die Anbindung flexibel genug sein und geeignete Schnittstellen bieten, um in Zukunft weitere Algorithmen anbinden zu können.
- Für gefundene, mögliche Typgeneralisierungen soll auch gleich eine Lösungshilfe in Form eines Quick-Fixes bereitgestellt werden. Dies bedeutet, dass auf den monierten Quelltext das entsprechende Refactoring angewendet werden kann, um die Typgeneralisierung zu erreichen.

Neben den Anforderungen, die vor Projektbeginn formuliert worden sind, ergeben sich weitere, implizite Anforderungen. Zum einen sind diese durch die gewählte Umgebung vorgegeben und sind indirekt aus den Prinzipien der Plug-In-Entwicklung in Anhang B.1 abgeleitet. Zum anderen ergeben sich Anforderungen aus allgemeingültigen Vorgaben oder Erkenntnissen der Softwareentwicklung. Deshalb wurde das Plug-In möglichst modular und logisch strukturiert aufgebaut. Außerdem wurden alle relevanten Teile kommentiert, um daraus automatisiert eine aussagekräftige Quelltextdokumentation erstellen zu können.

4.2. Architektur des Plug-Ins

Durch die Entscheidung, das Projekt als Plug-In für die Eclipse-Plattform zu realisieren, wurde für den größten Teil der Architektur bereits ein Rahmen festgelegt. Sie richtet sich deshalb nach den Vorgaben der Architektur der Eclipse-Plattform.[1] Dies erleichtert erheblich den Entwurf des Plug-Ins, da angestrebt wurde, wo immer es geht die Ressourcen zu nutzen, die durch die Plattform bereitgestellt werden. Am wichtigsten ist hierbei die Steuerung des Lebenszyklusses von Plug-Ins. Diese konnte beispielsweise auf reine Konfiguration reduziert werden, womit sichergestellt wurde, dass das Plug-In höchstwahrscheinlich auch in zukünftigen Versionen von Eclipse lauffähig bleiben wird. Die Architektur konnte nun darauf abzielen, sich auf die Basislogik der Quelltext-Prüfung zu konzentrieren, wobei hierfür auch auf etablierte Konzepte innerhalb der Plattform zurückgegriffen werden konnte; die folgenden Abschnitte beschreiben die zentralen Komponenten des Plug-Ins und ihr Zusammenspiel.

[1] Diese Vorgaben werden im Anhang B.1 als Prinzipien der Plug-In-Entwicklung definiert.

4.2.1. Quelltextanalyse mit Builder und AST-Visitor

Der naheliegendste Gedanke wäre, die Quelltextanalyse als Aktion zu implementieren, die über einen Menüeintrag angestoßen wird. Somit könnte der Enwtickler, der das Plug-In verwendet, jederzeit auf einfache Weise die Generalisierungsprüfung durchführen. Allerdings ergeben sich entscheidende Nachteile, da es innerhalb der Eclipse-Plattform für Menüaktionen keinen festen Rahmen gibt. Grundsätzlich könnte eine solche Aktion fast jede Operation ausführen, jedoch muss die Aktion dann viel Logik implementieren, die bestehende Konzepte der Plattform bereits bereitstellen. Deshalb wurde die Analyse der Quelltexte, die für die Prüfungen sämtlicher Typdeklarationen durchgeführt werden muss, als Builder realisiert. Das in Anhang C.2.1 vorgestellte Konzept des Builders ist prädestiniert für diesen Einsatz. Hierfür sprechen folgende Gründe:

- Builder sind Bestandteil der Eclipse-Plattform und somit ein etabliertes Konzept, das einfach erweitert werden kann.
- Builder sind dafür geschaffen worden, um Operationen auf Quelltexten auszuführen. Bekanntester Vertreter ist der Java-Builder, der die Quelltexte mit Hilfe des Java-Compilers übersetzt.
- Builder haben Zugriff auf alle Ressourcen eines Projektes und können sogar die Differenzen seit dem letzten Lauf des Builders bereitstellen.[2]
- Builder können automatisch oder optional bei Bedarf gestartet werden.
- Builder stellen die nötigen Schnittstellen zum Benutzer bereit, um Prüfungen durchführen, konfigurieren und die Ergebnisse anzeigen und gegebenenfalls zurücksetzen zu können.

Der Builder stellt aber nur die Infrastruktur für die Prüfung bereit. Die Analyse des Grades der Generalisierung führt der Typinferenzalgorithmus aus, der mit Typdeklarationen aus dem Quelltext parametriert werden muss. Um an die Typdeklarationen zu kommen, wird aus den Ressourcen, die der Builder liefert, ein Abstract-Syntax-Tree aufgebaut, der mit Hilfe eines AST-Visitors traversiert wird. Die in Anhang C.2.2 vorgestellte Implementierung des AST-Visitors in der Eclipse-Plattform wird

[2] Hierbei spricht man von einem inkrementellen Builder.

entsprechend erweitert, damit er Deklarationen von Variablen, Methodenparametern und Rückgabewerten von Methoden liefert. Diese drei Deklarationsarten können auf Generalisierung geprüft werden. Mit dem Konzept des AST-Visitors stellt die Plattform einen Dienst bereit, um auf einfache Weise Quelltexte untersuchen zu können. Die Implementierung im Plug-In beschränkt sich deshalb auf den Einsatz dieser Standardkomponente.

4.2.2. Anbindung von Typinferenzalgorithmen

Die Prüfung des Grades der Generalisierung einer Deklaration ist nicht Teil des Declared Type Generalization Checker Plug-Ins. Dessen Aufgabe ist vielmehr, die erforderlichen Ressourcen für die Prüfung bereitzustellen und diese dann anzustoßen. Deshalb muss eine Möglichkeit geschaffen werden, den Typinferenzalgorithmus, der aus einer externen Komponente stammt, sauber anzubinden. Hierbei ist natürlich eine lose Kopplung anzustreben. Nur so ist sichergestellt, dass zukünftig weitere Typinferenzalgorithmen dynamisch hinzugefügt werden können. Diese Anforderung lässt sich erfüllen, indem das Declared Type Generalization Checker Plug-In ein Interface bereitstellt, das von den Komponenten implementiert werden muss, welche einen Typinferenzalgorithmus anbieten. Die Abbildung 4.1 stellt dies in UML-Notation entsprechend dar.

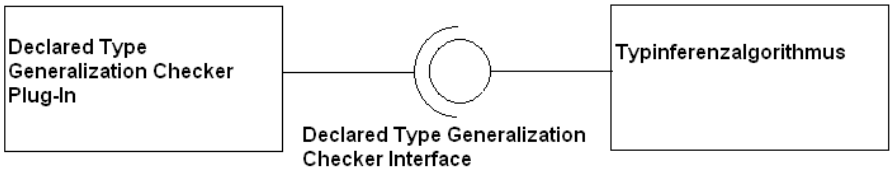

Abbildung 4.1.: Interface für Prüfalgorithmen

Es handelt sich dabei um ein anbietendes Interface, da die implementierende Klasse dem Aufrufer, also dem Declared Type Generalization Checker Plug-In, eine Dienstleistung anbietet, von der der Aufrufer profitiert.[3] Um die Implementierung des Interfaces konform mit den Richtlinien der Eclipse-Plattform zu halten, wird das Interface als Extension-Point deklariert. Somit wird der Regel der expliziten Erweiterung genüge getan.[4] Außerdem hat das Declared Type Generalization Checker Plug-In da-

[3] Für eine weiterführende Betrachtung von Interfacetypen sei auf Steimann u. a. (2005) verwiesen.
[4] siehe Anhang B.1

mit die Möglichkeit, über die Extension-Registry zu erfragen, von welchen Plug-Ins das Interface implementiert wird, und es kann die Implementierungen dem Benutzer in einem Dialog zur Auswahl anbieten.
In der ersten Version des Declared Type Generalization Checker Plug-Ins soll die Typinferenz mit den Algorithmen des Infer Type und des Generalize Declared Type Refactorings geprüft werden. Hierbei besteht die Schwierigkeit, dass, um die Logik für die Generalisierungsprüfung bereitzustellen, beide Plug-Ins das benötigte Interface des Declared Type Generalization Checker Plug-Ins implementieren müssen. Die Quelltexte beider Plug-Ins können aber nicht innerhalb dieses Projektes einfach entsprechend geändert werden; auch ist nicht zu erwarten, dass die Entwickler der Plug-Ins das Interface kurzfristig selbst implementieren. Deshalb ist klar, dass der Zugriff auf die Algorithmen doch in das Declared Type Generalization Checker Plug-In verlagert werden muss.
Beide Algorithmen direkt anzubinden wäre jedoch eine unsaubere Lösung. Außerdem wäre das Declared Type Generalization Checker Plug-In dann zu eng an beide Plug-Ins gebunden, was bedeuten würde, dass beide Plug-Ins stets geladen sein müssen, damit auch das Declared Type Generalization Checker Plug-In einsetzbar ist. Gerade das ist bei der offenen Struktur der Eclipse-Plattform nicht bei jeder Installation zwingend gegeben. Außerdem ist es sinnvoller, die Anbindung an diese beiden Plug-Ins nicht proprietär zu lösen, wenn für beliebige Plug-Ins bereits eine allgemeine Lösung mittels des Interfaces vorhanden ist.
In solch einer Konstallation bietet sich das Muster des Adapters an, das dafür konzipiert ist, syntaktisch nicht zusammenpassende Komponenten zu verbinden.[5] Die Implementierung des Adapters, wie sie sich hier wiederfindet, sieht vor, dass er Teil des Declared Type Generalization Checker Plug-Ins ist und ganz offiziell das vom Plug-In selbst bereitgestellte Interface mittels einer Extension implementiert. Durch diesen Kunstgriff stellt der Adapter, obwohl er Teil des Declared Type Generalization Checker Plug-Ins ist, die Funktionalität der Generalisierungsprüfung bereit. Damit der Adapter auch die Funktionalität bereitstellen kann, delegiert er die Aufrufe an das Infer Type bzw. Generalize Declared Type Refactoring. Die folgende Abbildung veranschaulicht die Funktion des Adapters:

[5] Für die theoretischen Grundlagen des Adapters sei auf Gamma u. a. (2001) verwiesen.

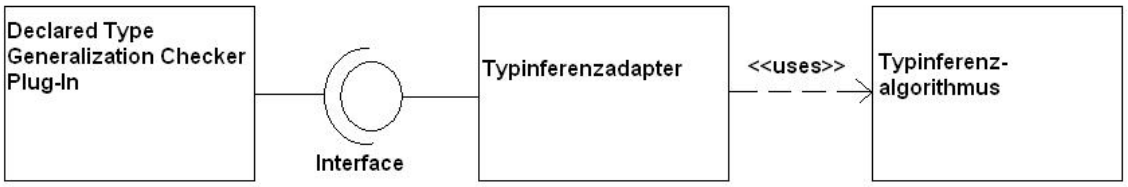

Abbildung 4.2.: Adapter zwischen Interface und Prüfalgorithmus

Im Declared Type Generalization Checker Plug-In existiert jeweils ein Adapter zum Infer Type und Generalize Declared Type Refactoring. So ist das Plug-In auch ohne eines der beiden anderen Plug-Ins einsetzbar. Problematisch ist an diesem Modell, dass sowohl das Infer Type Refactoring als auch das Generalize Declared Type Refactoring keinen Zugriff auf den Typinferenzalgorithmus als Extension-Point bereitstellen. Deshalb wird ganz bewusst die Regel der expliziten Erweiterung verletzt, indem der Adapter direkt auf die beiden Plug-Ins zugreift. Bei Änderungen an den Aufrufschnittstellen innerhalb der beiden Plug-Ins besteht die Gefahr, dass der Adapter nicht mehr funktioniert. Im Moment besteht keine andere Möglichkeit, dieses Problem sauber zu lösen, und etwaige Konsequenzen daraus müssen akzeptiert werden (wobei die erste Konsequenz daraus ist, dass bei Versionssprüngen der Refactoring-Plug-Ins darauf geachtet werden muss, die Adapter gegebenenfalls nachzuziehen).
Der Aufruf des Prüfalgorithmus erfolgt in der Weise, dass während der Initialisierungsphase des Declared Type Generalization Checker Plug-Ins von der Extension-Registry erfragt wird, welche Plug-Ins das exportierte Interface implementieren und welches dieser Plug-Ins vom Entwickler ausgewählt wurde, die Prüfung durchzuführen. Die Extension-Registry liefert auch den Pfad der Klasse mit der Implementierung des Interfaces. Über die Reflection-Mechanismen von Java lässt sich diese Klasse zur Laufzeit dynamisch laden und die Methoden zur Generalisierungsprüfung können vom Declared Type Generalization Checker Plug-In aufgerufen werden. Der Aufruf der Prüfung einer Deklaration erfolgt dann innerhalb der *visit(...)*-Methoden, die der AST-Visitor zur Bearbeitung des jeweiligen Deklarationselements aufruft.

4.2.3. Erzeugung von Markierungen und Lösungshilfen

Sofern die Generalisierungsprüfung von Variablendeklarationen das Ergebnis liefert, dass der verwendete Typ verallgemeinert werden kann, muss dies dem Entwickler natürlich auch mitgeteilt werden. Hierfür bietet sich das in Anhang C.2.3 beschrie-

bene Konzept der Problem-Marker an. Dieses erlaubt es, den Entwickler in seinem Quelltext auf Probleme oder, wie in diesem Fall, auf mögliche Typgeneralisierungen hinzuweisen. Die Anzeige des Hinweises fügt sich in die geläufige Darstellung ein, deshalb muss der Entwickler auch keine Ansichten umschalten oder über spezielle Dialoge die Prüfungsergebnisse abfragen. Außerdem sind die Ergebnisse so als Warnungen gestaltet, dass sie den normalen Entwicklungsprozess nicht aufhalten oder beeinträchtigen. Darüber hinaus greifen bei den Warnungen auch die Filter- und Sortierfunktionen der Eclipse-Oberfläche, damit der Entwickler gegebenenfalls die Hinweise ausblenden oder gezielt nach ihnen suchen kann. Das Setzen der Markierungen erfolgt in den Methoden, die der AST-Visitor aufruft. Hier wurde bereits die Prüfung einer Deklaration durchgeführt, deshalb stehen alle Informationen über die geprüfte Ressource[6] zur Verfügung und können gleich für das Setzen der Markierung verwendet werden.
Eng mit den Markierungen sind die Lösungsvorschläge zu den Markierungen, die in Anhang C.2.4 vorgestellten Quick-Fixes, verbunden. Das Declared Type Generalization Checker Plug-In stellt die Infrastruktur bereit, zu jeder gesetzten Markierung einer möglichen Typgeneralisierung einen Quick-Fix anzubieten. Allerdings ist dies optional, da nicht erwartet werden kann, dass jeder Inferenzalgorithmus, der angebunden wird, hierfür die nötige Logik bereitstellt.[7] Deshalb wird der Aufruf des Quick-Fixes auch über das Interface des Declared Type Generalization Checker Plug-Ins gekapselt. Dies wird erreicht, indem das Declared Type Generalization Checker Plug-In einen Generator für Quick-Fixes implementiert, der alle Aufrufe für die gesetzten Markierungen entgegennimmt und an den dynamisch geladenen Prüfalgorithmus delegiert. Dieser implementiert über das Interface eine Methode, über die erstmal nachgefragt werden kann, ob der Algorithmus einen Quick-Fix anbietet. Darüber hinaus implementiert er eine weitere Methode, die dann auch den Quick-Fix zurückliefert, sofern einer bereitgestellt wird.

4.2.4. Konfiguration und Aufruf des Plug-Ins

Die Generalisierungsprüfung sollte so steuerbar sein, dass sie der Entwickler nicht zwingend für alle Projekte durchführen muss, die er in seiner Entwicklungsumge-

[6] In diesem Fall sind dies die Bezeichner der Deklarationselemente von Methoden, Feldern und lokalen Variablen sowie deren Positionen im Quelltext.
[7] Sowohl Infer Type als auch Generalize Declared Type Refactoring stellen einen Quick-Fix in Form eines Aufrufs des Refactorings bereit.

bung bearbeitet. Vielmehr soll die Prüfung selektiv für die Projekte zugeschaltet werden können, bei denen es der Entwickler für sinnvoll erachtet. Deshalb wurde die Konfiguration des Declared Type Generalization Checker Plug-Ins als Eigenschaft eines Projektes gestaltet,[8] denn Eigenschaften lassen sich für jedes Projekt einzeln definieren.[9]
Im Dialog des Declared Type Generalization Checker Plug-Ins kann das Plug-In zunächst grundsätzlich für das Projekt aktiviert bzw. deaktiviert werden. Dies bewirkt, dass dem Projekt die Nature und der Builder hinzugefügt bzw. davon entfernt werden.[10] Darüber hinaus kann man in einer Auswahlliste, die alle verfügbaren Prüfalgorithmen enthält, den Algorithmus auswählen, mit dem die Prüfungen durchgeführt werden sollen. Diese Einstellungen bleiben auch nach dem Neustart der Eclipse-Plattform erhalten, können aber jederzeit vom Entwickler über die Dialoge wieder geändert werden.

4.3. Klassenstruktur und Packages

Für die Beschreibung der Struktur der Klassen und deren Zusammenwirken eignet sich am besten eine grafische Darstellungsform. Die Abbildung 4.3 zeigt die Klassen des Declared Type Generalization Checker Plug-Ins und die Zugriffe der Klassen untereinander. Die Zugriffe sind mit dem Stereotyp <<*uses*>> kenntlich gemacht, darüber hinaus ist in Lollipop-Notation angegeben, wenn Klassen Interfaces implementieren. Die Grafik genügt somit der UML-Notation für Klassendiagramme[11], allerdings wird in diesem Abschnitt darauf verzichtet, die einzelnen Klassen näher zu beschreiben. Dies erfolgt im nächsten Abschnitt, wo die genaue Implementierung diskutiert wird.
Die Klassen des Plug-Ins lassen sich zu Paketen zusammenfassen, die ihrer Zusammengehörigkeit bzgl. der Funktion und Aufgabe entsprechen. Diese Packages und ihre Beziehungen werden ebenfalls in UML-Notation in Abbildung 4.4 dargestellt. Wird ein Paket von einem anderen benötigt, so ist diese Abhängigkeit mit dem Stereotyp <<*importieren*>> gekennzeichnet. Um das Paketdiagramm nicht zu überladen,

[8] im Eclipse-Jargon: Property
[9] Die entsprechenden Dialoge lassen sich beispielsweise öffnen, indem man mit der rechten Maustaste in der Entwicklungsumgebung auf ein Projekt klickt und im Kontextmenü *Properties* auswählt.
[10] siehe Anhang C.2.1
[11] Für die genaue Definition der Notationselemente sei auf Jeckle u. a. (2004) verwiesen.

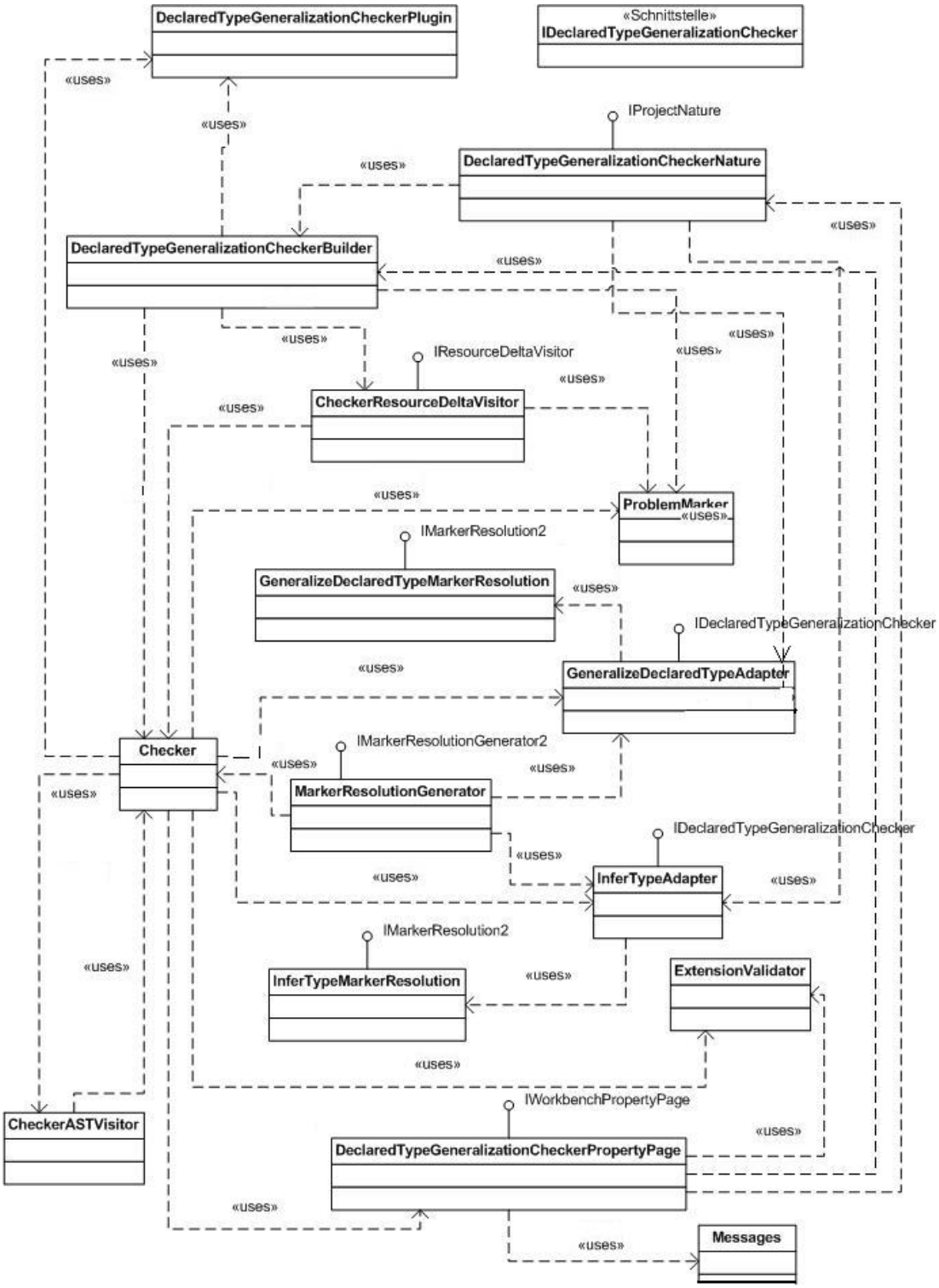

Abbildung 4.3.: Klassendiagramm des Plug-Ins in UML-Notation

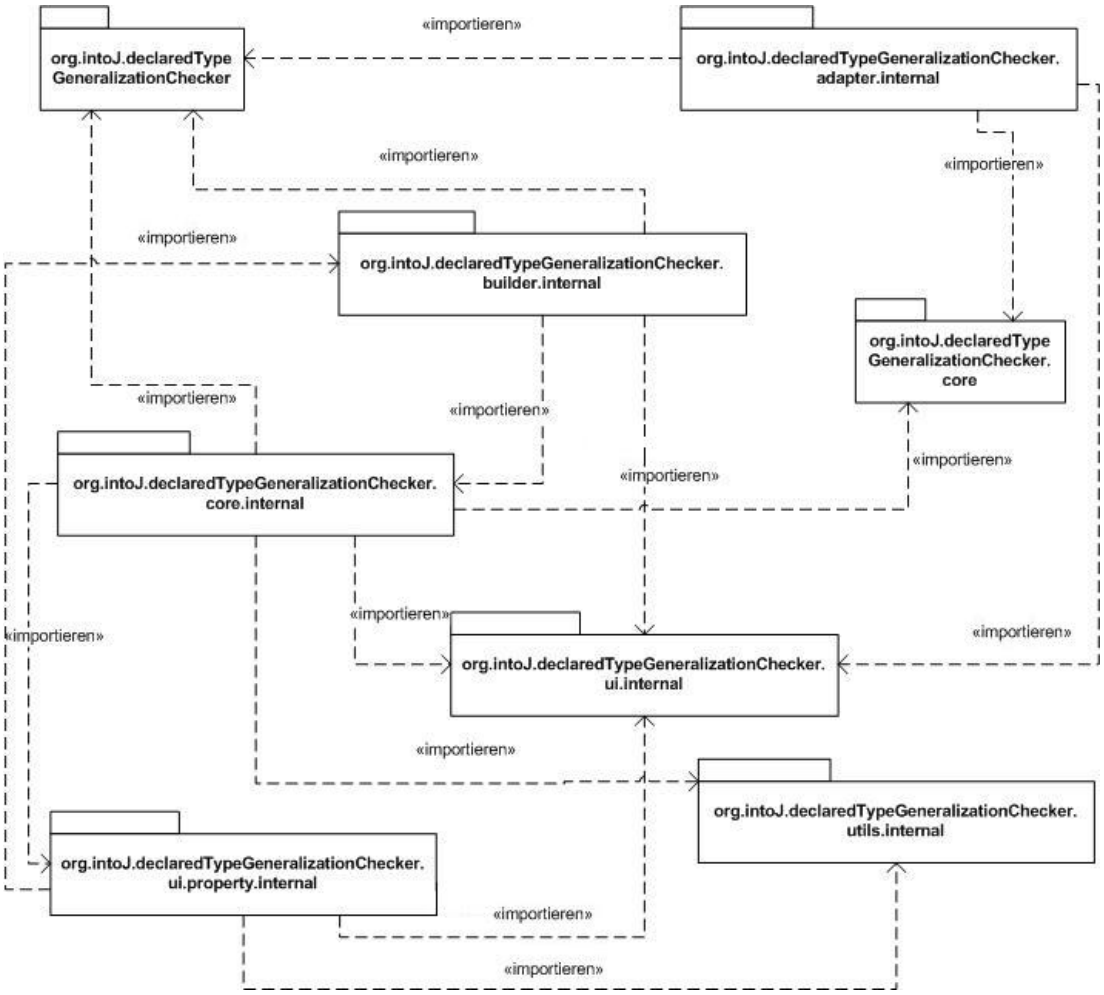

Abbildung 4.4.: Paketdiagramm der Plug-In-Struktur

wurde darauf verzichtet, den Paketen die zugehörigen Klassen zuzuweisen. Diese Zuordnung kann aber der Tabelle 4.1 entnommen werden.

Paket	**Klasse**
org.intoJ.declaredTypeGeneralization-Checker	DeclaredTypeGeneralizationChecker-Plugin
org.intoJ.declaredTypeGeneralization-Checker.adapter.internal	GeneralizeDeclaredTypeAdapter GeneralizeDeclaredTypeMarker-Resolution InferTypeAdapter InferTypeMarkerResolution
org.intoJ.declaredTypeGeneralization-Checker.builder.internal	DeclaredTypeGeneralizationChecker-Builder DeclaredTypeGeneralizationChecker-Nature
org.intoJ.declaredTypeGeneralization-Checker.core	IDeclaredTypeGeneralizationChecker-
org.intoJ.declaredTypeGeneralization-Checker.core.internal	Checker CheckerASTVisitor CheckerResourceDeltaVisitor MarkerResolutionGenerator ProblemMarker
org.intoJ.declaredTypeGeneralization-Checker.ui.internal	Messages
org.intoJ.declaredTypeGeneralization-Checker.ui.property.internal	DeclaredTypeGeneralizationChecker-PropertyPage
org.intoJ.declaredTypeGeneralization-Checker.utils.internal	ExtensionValidator

Tabelle 4.1.: Aufteilung der Klassen des Plug-Ins in Pakete

Die Namensvergabe und Einteilung der Pakete orientiert sich an den Vorgaben bzw. Standards der Eclipse-Plattform. Deshalb sind alle Klassen bzw. Interfaces, die über einen Extension-Point für den externen Zugriff freigegeben sind, zu separaten Paketen zusammengefasst. Pakete mit Elementen, die nicht freigegeben sind, tragen in ihrem Namen den Postfix *internal*.[12] Auf diese Weise soll auch auf Paketebene verdeutlicht werden, dass Zugriffe auf Interna von Plug-Ins zu vermeiden sind. Ansonsten entspricht ein Paketname der umgekehrten Domain-Notation, wie dies allgemein in Java üblich ist.

[12]Die einzige Ausnahme dieser Konvention stellt die Basisklasse des Plug-Ins dar, ihr Paketname trägt nie den Postfix.

4.4. Implementierung

Nachdem bisher auf architektonischer Ebene dargestellt wurde, in welchem Rahmen das Plug-In implementiert wurde und welche Konzepte (der Eclipse-Plattform sowie genereller Natur) Anwendung fanden, wird in den folgenden Abschnitten detailliert auf die tatsächliche Umsetzung eingegangen. Dabei geht es nicht nur um die Quelltexte, deshalb werden im Folgenden bzw. im Anhang D auch keine vollständigen Implementierungen gezeigt. Zu jeder Klasse werden jedoch Rümpfe präsentiert, aus denen man entnehmen kann, welche Elemente die Klassen enthalten. Detailliert wird nur auf interessante Aspekte eingegangen, deren Implementierung weiterführend diskutiert und dargestellt werden.
Die Aufteilung der Abschnitte entspricht nicht unbedingt den Paketen, aus denen sich das Plug-In zusammensetzt. Die Pakete wurden logisch weiter zusammengefasst, um Beziehungen und Abhängigkeiten klarer darzustellen. Darüber hinaus wird gegebenenfalls zusammen mit der Implementierung die jeweilige notwendige Konfiguration im Manifest des Plug-Ins beschrieben.

4.4.1. Hauptklasse und Manifest

Die Hauptklasse des Plug-Ins ist die Klasse *DeclaredTypeGeneralizationCheckerPlugin* im Package *org.intoJ.declaredTypeGeneralizationChecker*. Sie wird von der Eclipse-Plattform aufgerufen, wenn das Plug-In geladen werden soll, weshalb sie auch die Methoden für die Steuerung des Lebenszyklus enthält. Ansonsten enthält die Klasse nur noch die Funktionalität, über die statische Methode *getDefault()* Instanzen von sich selbst zu liefern, um somit den Zugriff auf die nicht-statischen Methoden ihrer Basisklasse *AbstractUIPlugin* zu erlauben. Die Klasse *AbstractUIPlugin* muss von jedem Plug-In erweitert werden, das eine grafische Oberfläche erzeugt, denn die Klasse enthält Programmlogik, um auf Konfigurationsparameter für Dialoge oder Grafiken zuzugreifen. Ihrerseits erweitert die Klasse selbst die ebenfalls abstrakte Klasse *Plugin*, die Grundfunktionalitäten wie z.B. Logging oder den Zugriff auf Basiskonfigurationen bereitstellt. Die Implementierung der Klasse *DeclaredTypeGeneralizationCheckerPlugin* sieht wie folgt aus:

```
public class DeclaredTypeGeneralizationCheckerPlugin extends \
  AbstractUIPlugin {
```

```
    private static DeclaredTypeGeneralizationCheckerPlugin plugin;

    public DeclaredTypeGeneralizationCheckerPlugin() {
        plugin = this;
    }
    public void start(BundleContext context) throws Exception {
        super.start(context);
    }
    public void stop(BundleContext context) throws Exception {
        super.stop(context);
        plugin = null;
    }
    public static DeclaredTypeGeneralizationCheckerPlugin getDefault() {
        return plugin;
    }
}
```

Für des Laden bzw. Beenden des Plug-Ins sind keine besonderen Operationen notwenig, deshalb rufen die Methoden *start(...)* bzw. *stop(...)* nur die Implementierungen der Methoden in der Basisklasse auf.
Die Grundkonfiguration des Plug-Ins erfolgt in der Datei *MANIFEST.MF* im Verzeichnis *META-INF*:

```
Bundle-ManifestVersion: 2
Bundle-Name: Declared Type Generalization Checker Plug-In
Bundle-SymbolicName: org.intoJ.declaredTypeGeneralizationChecker; \
  singleton:=true
Bundle-Version: 1.0.0
Bundle-ClassPath: org.intoJ/
Bundle-Activator: org.intoJ.declaredTypeGeneralizationChecker. \
  DeclaredTypeGeneralizationCheckerPlugin
Bundle-Vendor: intoJ Team, University of Hagen
Eclipse-LazyStart: true
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime,
 org.eclipse.core.resources,
 org.eclipse.jdt.core,
 org.eclipse.jdt.ui,
```

```
 org.eclipse.ltk.core.refactoring,
 org.eclipse.ui.ide,
 org.eclipse.jface.text,
 org.intoJ.inferType;resolution:=optional,
 org.eclipse.help
Export-Package: org.intoJ.declaredTypeGeneralizationChecker.core
Bundle-Localization: plugin
```

Das Manifest macht Angaben darüber, welcher Version der OSGi-Implementierung das Plug-In entspricht, um der Eclipse-Plattform mitzuteilen, wie das Plug-In gegebenenfalls zu behandeln ist. Darüber hinaus wird mit den Parametern *Bundle-Name*, *Bundle-SymbolicName*, *Bundle-Version* und *Bundle-Vendor* spezifiziert, um welches Bundle es sich handelt. Der zusätzliche Wert *singleton* des Parameters *Bundle-SymbolicName* legt fest, dass innerhalb der Plattform nur eine Instanz des Plug-Ins gestartet werden darf. Die zu startende Hauptklasse wird als Wert des Parameters *Bundle-Activator* angegeben. Der nicht OSGi-konforme Parameter *Eclipse-LazyStart* legt mit dem Wert *true* hierzu fest, dass das Plug-In beim ersten Aufruf des Plug-Ins zu starten ist. Den generell für den Start benötigten Classpath, unterhalb dessen nach Ressouren gesucht wird, setzt der Parameter *Bundle-ClassPath*. Zusätzlich macht das Manifest mit dem Parameter *Require-Bundle* Angaben darüber, welche Plug-Ins benötigt werden. Ist eines der Plug-Ins nicht verfügbar, wird das Plug-In selbst auch nicht gestartet. Dies kann jedoch umgangen werden, indem ein Plug-In mit dem zusätzlichen Wert *resolution:=optional* gekennzeichnet wird. Hier ist dies bei der Abhängigkeit vom Infer Type Refactoring der Fall. Da das Plug-In Funktionalität anderen Plug-Ins zur Verfügung, stellt wird das interne Package, das die Funktionalität enthält, als Wert des Parameters *Export-Package* genannt. Eine genauere Konfiguration des Extension-Points erfolgt dann aber in der Datei *plugin.xml*, worauf später noch genauer eingegangen wird. Durch den Wert des Parameters *Bundle-Localization* wird angegeben, dass in der Datei *plugin.properties*[13] Texte sind, auf die in den Plug-In-Konfigurationen zugegriffen werden kann. Dieses Herauslösen und zentrale Sammeln ist unter dem Gesichtspunkt der Internationalisierung eines Plug-Ins wichtig.[14]

[13] Der Wert besagt, dass sich die Datei auf der untersten Verzeichnisebene im Plug-In befindet und der Name mit *plugin* beginnt, die Endung *.properties* ist implizit.

[14] Die Datei enthält die Texte in der Form *<Schlüssel>=<Wert>*. Die Texte werden in den Konfigurationen referenziert, indem man, wo sie eingesetzt werden sollen, angibt *%<Schlüssel>*.

4.4.2. Property-Page

Das Declared Type Generalization Checker Plug-In implementiert eine Property-Page, auf der die Generalisierungsprüfung aktiviert und zusätzlich der Algorithumus, mit dem geprüft werden soll, festgelegt werden kann. Die Property-Page integriert sich in den Rahmen für Property-Pages, den die Eclipse-Plattform vorgibt. Deshalb enthält die Implementierung der Property-Page nur die Elemente, die auf der Page angezeigt werden, den Rest steuert die Plattform. Hierfür muss zunächst der Extension-Point *org.eclipse.ui.propertyPages* erweitert werden. Die hierfür nötige Konfiguration in der Datei *plugin.xml* ergibt sich wie folgt:

```
<extension point="org.eclipse.ui.propertyPages"
           name="%AbstractionCheckerPropertyPage">
    <page id="declaredTypeGeneralizationCheckerPropertyPage"
          class="org.intoJ.declaredTypeGeneralizationChecker.ui.property. \
                internal.DeclaredTypeGeneralizationCheckerPropertyPage"
          name="%AbstractionCheckerPropertyPage"
          objectClass="org.eclipse.core.resources.IProject"/>
</extension>
```

Die Deklaration der Extension enthält das Element *page*, in dem die Details zur Extension angegeben werden. Hierzu gehört insbesondere der Pfad zur Klasse, welche die Funktionalität der Page implementiert. Das Attribut *objectClass* regelt, für welche Ressourcen die Property-Page aufrufbar ist, in diesem Fall also für alle Projekte innerhalb der Entwicklungsumgebung. Für die Page muss neben einem eindeutigen Bezeichner im Attribut *id* auch noch ein lesbarer Name angeben werden. Der Wert des Attributs *name* besagt, dass der Text dafür aus der *.properties*-Datei des Plug-Ins stammt.[15] Für den Namen der Extension selbst kann der gleiche Wert verwendet werden, da die Extension nur eine Property-Page bereitstellt.
Die Klasse *DeclaredTypeGeneralizationCheckerPropertyPage*, die in der Deklaration der Extension angegeben wurde, befindet sich im Package *org.intoJ.declaredType GeneralizationChecker.ui.property.internal* und erweitert die abstrakte Klasse *PropertyPage* der Eclipse-Plattform. Der Aufbau der Klasse ist im Anhang D.1 zu finden. Die wichtigste Methode der Klasse ist *createContents(...)*; sie wird von der Eclipse-Plattform aufgerufen, wenn die Property-Page betreten wird. Die Methode erzeugt

[15] Hierzu sei auch auf den vorherigen Abschnitt verwiesen.

die Inhalte der Poperty-Page und liefert sie in Form eines Objekts vom Typ *Control* an die Plattform zurück. Die Methode ist obligatorisch, da sie als abstrakte Methode in der ebenfalls abstrakten Klasse *org.eclipse.ui.dialogs.PreferencePage* deklariert wird, von der sich die Klasse *org. eclipse.ui.dialogs.PropertyPage* ableitet, die ja Basisklasse von *DeclaredTypeGeneralizationCheckerPropertyPage* ist. Die Methoden *addSeparator(...)*, *addBuilderSwitch(...)* und *addCheckerCombo(...)* sind reine Hilfsmethoden, die von *createContents(...)* aufgerufen werden, um einen Zeilentrenner und die Steuerelemente für das Aktivieren der Generalisierungsprüfung und die Auswahl des Algorithmus zu erstellen. Ebenfalls eine Hilfsmethode ist *addInferTypeLink(...)*; sie erstellt auf der Benutzeroberfläche einen Hinweis und Link zum Download vom Infer Type Plug-In, sofern dieses nicht installiert ist. In der Klasse *PreferencePage* befindet sich auch die Implementierung einer Methode *performOk()*. Sie ist der Default-Handler, wenn auf der Property-Page die Schaltfläche *OK* gedrückt wird. In der Klasse *DeclaredTypeGeneralizationCheckerPropertyPage* wird die Methode deshalb überschrieben, denn dort bewirkt das Drücken auf *OK*, dass die Werte, die auf der Page eingestellt wurden, dauerhaft gespeichert werden und abhängig von der Auswahl dem Projekt die Nature und der Builder des Plug-Ins hinzugefügt bzw. diese entfernt werden. Die Einstellungen, die auf der Property-Page gemacht wurden, sollen natürlich auch dauerhaft erhalten bleiben. Das Speichern und Laden der Einstellungen übernehmen die Methoden *setPersitentCheckerSelection(...)* und *getPersitentCheckerSelection()*. Diese Methoden rufen ihrerseits die Lademechnismen der Eclipse-Plattform auf, somit muss sich um den Speicherort und die Speicherart nicht selbst gekümmert werden. Für die Daten muss aber ein eindeutiger Bezeichner gesetzt werden, damit sie wieder auffindbar sind. Dieser Bezeichner liegt im Feld *CHECKER_PROPERTY_KEY* vom Typ *QualifiedName*. Im Weiteren enthält die Klasse die Methode *getDefaultCheckerSelection()*, die stets den Bezeichner des Infer Type Refactorings zurückliefert, da dieses der vorausgewählte Algorithmus zur Generalisierungsprüfung ist. Die Methode *getProject()* gibt eine Referenz auf das Projekt zurück, für das die Property-Page aufgerufen wurde. Diese Referenz wird intern benötigt, da über sie dem Projekt die Nature hinzugefügt bzw. davon entfernt wird.

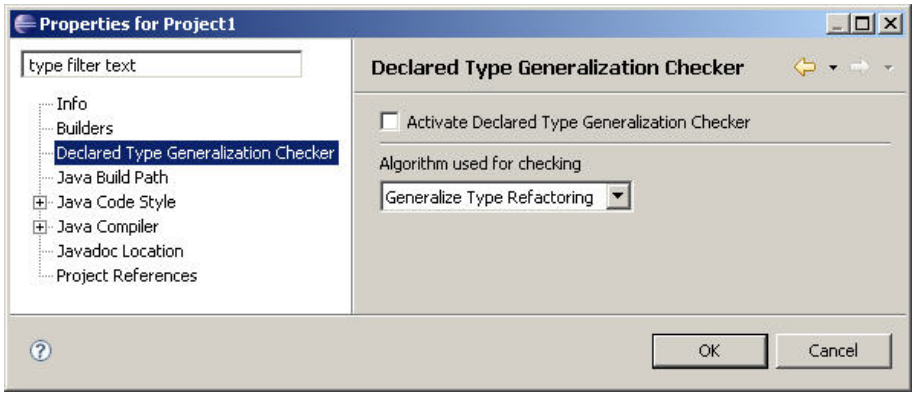

Abbildung 4.5.: Property-Page zur Steuerung des Plug-Ins

4.4.3. Nature und Builder

Zunächst wird in diesem Abschnitt auf die Realisierung der Nature eingegangen, da sie die Grundlage für den Builder ist und ihn mit einem Java-Projekt verknüpft. Zuerst sei deshalb die konkrete Konfiguration der Nature gezeigt; für einen allgemeinen Überblick wird auf Anhang C.2.1 verwiesen.

```
<extension id="declaredTypeGeneralizationCheckerNature"
           name="%NatureName"
           point="org.eclipse.core.resources.natures">
    <builder id="org.intoJ.declaredTypeGeneralizationChecker.declaredType \
      GeneralizationCheckerBuilder"/>
    <runtime>
        <run class="org.intoJ.declaredTypeGeneralizationChecker.builder. \
          internal.DeclaredTypeGeneralizationCheckerNature"/>
    </runtime>
</extension>
```

Die Erweiterung wird durch den eindeutigen Bezeichner *id* und den lesbaren Bezeichner *name* ausgezeichnet. Mit dem Element Builder gibt die Erweiterung bereits den Bezeichner des Builders bekannt, den sie verwaltet. Das Element *run* innerhalb des Elements *runtime* gibt im Attribut *class* den Namen Klasse der Nature an; die Implementierung von DeclaredTypeGeneralizationCheckerNature befindet sich im Anhang D.2.1. Diese Klasse muss das Interface *org.eclipse.core.resources.IProjectNature* implementieren, das die Methoden *configure()*, *deconfigure()*, *getProject()* und *set-*

Project(...) enthält. Der Bezeichner der Nature aus der Konfiguration wird in der Klasse im Feld *NATURE_ID* wiederholt. Hierüber wird die Nature auf Java-Ebene identifiziert. Die Nature enthält darüber hinaus eine Referenz auf das Projekt, zu dem sie zugewiesen ist. Die entsprechenden Accessor-Methoden stammen bereits aus dem Interface *IProjectNature*. Die Methode *configure()* verbindet zusätzlich noch den Builder mit dem Projekt. Der Builder implementiert zu diesem Zweck eine statische Methode[16] die einfach aufgerufen wird. Das Entfernen des Builders in der Methode *deconfigure()* erfolgt analog.
Die statischen Methoden *addNatureToProject(...)* und *removeNatureFromProject(...)* werden von der Property-Page aufgerufen, wenn die Nature zugewiesen bzw. entfernt werden soll. Beide Methoden führen zunächst einfache Prüfungen durch, ob die Nature dem Projekt überhaupt zugewiesen werden kann oder ihm schon bereits zugewiesen ist. Die Nature stellt generell eine Beschreibung eines Projektes[17] dar; sie wird ihm zugewiesen, indem alle Beschreibungen des Projekts als Array abgerufen werden. Diesem wird der Bezeichner der Nature hinzugefügt und die modifizierte Liste wird dem Projekt komplett zugewiesen. Mehr ist hierfür nicht zu tun. Das Entfernen der Nature erfolgt ähnlich: Vom Array der Projektbeschreibungen wird der Bezeichner der Nature entfernt und dann das Array beim Projekt gesetzt.
Für die Implementierung des Builders muss der Extension-Point *org.eclipse.core.resources.builders* erweitert werden; die konkrete Konfiguration des Builders in der Datei *plugin.xml* hat somit folgenden Aufbau:

```
<extension id="declaredTypeGeneralizationCheckerBuilder"
           name="%BuilderName"
           point="org.eclipse.core.resources.builders">
    <builder hasNature="true"
             isConfigurable="false">
        <run class="org.intoJ.declaredTypeGeneralizationChecker.builder. \
          internal.DeclaredTypeGeneralizationCheckerBuilder"/>
    </builder>
</extension>
```

Neben den obligatorischen Attributen *id*, *name* und *point* ist das Element Builder enthalten, das den Builder genauer spezifiziert. Das Attribut *hasNature* teilt der

[16]Die Methode *addBuilderToProject(...)*, wird weiter unten in diesem Abschnitt noch genauer beschrieben.
[17]Dies ist ein Objekt vom Typ *org.eclipse.core.resources.IProjectDescription*.

Plattform mit, dass dieser Builder von einer Nature verwaltet wird. *isConfigurable* steuert, ob der Entwickler in einem Standarddialog einstellen kann, welche Build-Anforderungen[18] dem Builder übergeben werden. Das Attribut ist jedoch hier auf *false* gesetzt, da der Builder grundsätzlich immer gestartet werden soll und selbst entscheidet, ob und wie zu reagieren ist. Das Attribut *class* des Elements *run* legt fest, welche Klasse den Builder bereitstellt und von der Eclipse-Plattform aufzurufen ist. Die Klasse *DeclaredTypeGeneralizationCheckerBuilder* erweitert die abstrakte Klasse *org.eclipse.core.resources.IncrementalProjectBuilder*; für die Implementierung sei auf Anhang D.2.2 verwiesen. Bevor auf die zentralen Methoden eingegangen wird, seien die statischen Methoden *addBuilderToProject(...)* und *removeBuilderFromProject(...)* erwähnt, die von der Nature aufgerufen werden, um den Builder einem Projekt hinzuzufügen bzw. ihn zu entfernen. Auf eine genaue Beschreibung der Implementierung dieser Methoden wird verzichtet, da sie ähnlich dem Hinzufügen und Entfernen von Natures ist[19], denn Builder werden auch als Projektbeschreibungen verwaltet. In diesem Zusammenhang ist auch die Hilfsmethode *hasBuilder(...)* zu nennen, die eine Prüfung enthält, ob der Builder dem Projekt aktuell hinzugefügt ist. Für die Verwaltung des Builders ist das Feld *BUILDER_ID* wichtig; über diesen Bezeichner wird der Builder referenziert.

Durch das Erweitern der abstrakten Klasse *IncrementalProjectBuilder* muss der Builder die abstrakte Methode *build(...)* überschreiben, die von der Eclipse-Plattform aufgerufen wird, wenn eine Build-Anforderung eintrifft. Mit dem Parameter *kind* wird dem Builder mitgeteilt, welche Build-Anforderung gestellt wurde. Die möglichen Build-Arten werden in Anhang C.2.1 beschrieben. Wenn feststeht, welche Art von Build vorliegt, ruft die Methode *build(...)* die entsprechenden privaten Methoden *fullBuild()* oder *incrementalBuild(...)* auf. Diese führen dann die Generalisierungsprüfung unter Zuhilfenahme einer Instanz der Klasse *Checker* durch. Auf die Details dieser Klasse wird später im Abschnitt 4.4.5 über Prüfungs-Visitors eingegangen.

Der Parameter *args* der Methode *build(...)* wird hier nicht benötigt, über ihn könnte die Plattform dem Builder einen Satz Parameter übergeben. Hingegen ist der Parameter *monitor* vom Typ *org.eclipse.core.runtime.IProgressMonitor* von größerer Bedeutung. Über dieses Objekt kann der Builder der Plattform den Bearbeitungsfortschritt mitteilen, der dem Entwickler als Fortschrittsbalken in der Oberfläche angezeigt wird. Der Builder bekommt darüber aber auch Aktionen des Benutzers

[18] Mögliche Anforderungen sind Clean-Build, manueller Build oder automatischer Build.
[19] Siehe hierzu den Abschnitt über Natures weiter oben

mitgeteilt, wenn dieser besispielsweise den Build abbrechen möchte. Der Parameter *monitor* wird an das Objekt vom Typ *Checker* weitergegeben, da es die Generalisierungsprüfung steuert und somit den aktuellen Status kennt.
Die Implementierung der Methode *clean(...)* löscht alle Markierungen, die im Java-Projekt zuvor gesetzt wurden.

4.4.4. Generalisierungsprüfungs-Interface und Adapter

Das Generalisierungsprüfungs-Interface stellt die Schnittstelle zwischen der Infrastruktur für die Prüfung und den Komponenten, welche die Prüfung eines Typs durchführen, dar. Die Infrastruktur sammelt aus den Java-Quelltexten alle nötigen Informationen zu den Typdeklarationen und verwertet das Ergebnis der Prüfung des Abstraktionsgrades. Das Interfaces ist Teil des Declared Type Generalization Checker Plug-Ins. Implementiert wird das Interface dann in den Plug-Ins, welche die Generalisierungsprüfung durchführen. Die Ausnahme stellen die Adapter zum Infer Type und Generalize Declared Type Refactoring dar, da die Adapter das Interface implementieren, aber dennoch Teil des Declared Type Generalization Checker Plug-Ins sind. Bevor jedoch genauer auf die Adapter eingegangen wird, soll zunächst das Interface selbst und seine Veröffentlichung als Extension-Point diskutiert werden. Das Interface ist wie folgt deklariert:

```
public interface IDeclaredTypeGeneralizationChecker {
    public static final boolean GENERALIZATION_OK = true;
    public static final boolean GENERALIZATION_POSSIBLE = false;

    public boolean checkType(ICompilationUnit unit, String selectedType,
      int selectedTypeStart, String selectedName, int selectedNameStart,
      IProgressMonitor monitor);
    public boolean hasResolution();
    public IMarkerResolution2 getResolution();
}
```

Die zentrale Methode für die Generalisierungsprüfung ist *checkType(...)*, sie wird im Declared Type Generalization Checker Plug-In aufgerufen, um eine zu prüfende Typdeklaration zu bearbeiten. Der Methode wird eine Referenz auf die zu prüfende Ressource in Form eines Objekts vom Typ *org.eclipse.jdt.core.ICompilationUnit* übergeben;

h ier stellt eine Ressource eine Java-Datei dar. Im Weiteren wird der Methode der Name des zu prüfenden Typs und die Startpostion des Namens in der Datei übergeben. Ebenso wird der Methode der Name der deklarierten Variable und deren Position übergeben. Somit kann der Prüfalgorithmus den Kontext und die Deklaration selbst genau in der Ressource lokalisieren. Zusätzlich wird noch eine Instanz vom Typ *org.eclipse.core.runtime.IProgressMonitor* übergeben, damit der Inferenzalgorithmus gegebenenfalls Statusinformationen an die Oberfläche durchschleusen kann. Als Rückgabewert dienen die Werte der Felder *GENERALIZATION_POSSIBLE* und *GENERALIZATION_OK*, abhängig davon, ob der geprüfte Typ verallgemeinert werden kann oder nicht.
Das Interface enthält zusätzlich die Methoden *hasResolution()* und *getResolution()*, die mit der Prüfung direkt nichts zu tun haben, aber über die das Declared Type Generalization Checker Plug-In vom Prüfalgorithmus zunächst erfragen kann, ob er zusätzlich zum Prüfungsergebnis auch eine Lösung zur Typgeneralisierung anbietet. Dies leistet die Methode *hasResolution()*. Liefert die Methode *true* zurück, muss der Prüfalgorithmus beim Aufruf von *hasResolution()* ein Objekt vom Typ *org.eclipse.ui.IMarkerResolution2* zurückgeben, das als Quick-Fix gestartet werden kann.
Damit das Interface von anderen Plug-Ins erweitert werden kann, muss es als Extension-Point deklariert werden. Dies sieht in der Datei *plugin.xml* folgendermaßen aus:

```
<extension-point id="declaredTypeGeneralizationCheckerInterface"
    name="%AbstractionCheckerInterface"
    schema="schema/declaredTypeGeneralizationCheckerInterface.exsd"/>
```

Neben dem eindeutigen Bezeichner *id* und dem lesbaren Text *name* enthält die Deklaration noch das Attribut *schema*, das den Pfad zur Schemadatei angibt, in der die Anforderungen an eine Erweiterung genauer spezifiziert sind:

```
<schema targetNamespace="org.intoJ.declaredTypeGeneralizationChecker">
    <element name="extension">
        <complexType>
            <sequence>
                <element ref="declaredTypeGeneralizationChecker"/>
            </sequence>
            <attribute name="point" type="string" use="required"/>
            <attribute name="id" type="string" use="required"/>
```

```
            <attribute name="name" type="string" use="required">
                <annotation>
                    <appInfo>
                        <meta.attribute translatable="true"/>
                    </appInfo>
                </annotation>
            </attribute>
        </complexType>
    </element>
    <element name="declaredTypeGeneralizationChecker">
        <complexType>
            <attribute name="class" type="string" use="required">
                <annotation>
                    <appInfo>
                        <meta.attribute kind="java" basedOn="org.intoJ. \
                          declaredTypeGeneralizationChecker.core. \
                          IDeclaredTypeGeneralizationChecker"/>
                    </appInfo>
                </annotation>
            </attribute>
        </complexType>
    </element>
</schema>
```

Das Schema besagt, dass in einer Extension-Deklaration das Element *extension* die Attribute *point*, *id* und *name* zwingend enthalten muss. Für *name* wurde noch definiert, dass es sich dabei um ein Attribut handelt, dessen Wert sprachabhängig sein kann und das Attribut gegebenenfalls eine Referenz auf eine Sprachdatei enthält.[20] Die Definition des Elements Extension enthält zusätzlich die Vorgabe, dass ein Element *declaredTypeGeneralizationChecker* als Kindelement folgen muss. Die Definition des Elements *declaredTypeGeneralizationChecker* gibt an, dass dieses das Attribut *class* enthält, wobei für *class* nur der Name einer Java-Klasse eingetragen werden darf, die das Interface *org.intoJ.declaredTypeGeneralizationChecker.core.IDeclared TypeGeneralizationChecker* implementiert.
Die gezeigte Schemadatei wurde stark gekürzt, zusätzlich enthält sie Beschreibungen und Beispiele im Klartext für die Erweiterung des Extension-Points, um diese zu

[20] Dieser Mechanismus wurde im Abschnitt 4.4.1 bereits beschrieben.

erleichtern. Der Aufbau solcher Erweiterungen wird im folgenden gezeigt, wenn die Adapter zu den Refactorings vorgestellt werden. Eine allgemeine Beschreibung von Extensions und Extension-Points befindet sich in Anhang B.2.2.
Die Adapter zeichnen sich dadurch aus, dass sie die Brücke zwischen dem Declared Type Generalization Checker Plug-In und den Refactorings schlagen. Hierzu müssen die Adapter den eigenen Extension-Point des Declared Type Generalization Checker Plug-Ins erweitern. Die Deklaration, die sich aus den obigen Vorgaben ergibt, hat für das Infer Type Refactoring folgenden Aufbau:

```
<extension id="inferTypeAdapter"
           name="%InferTypeAdapter"
           point="org.intoJ.declaredTypeGeneralizationChecker. \
             declaredTypeGeneralizationCheckerInterface">
    <declaredTypeGeneralizationChecker class="org.intoJ. \
      declaredTypeGeneralizationChecker.adapter.internal.InferTypeAdapter"/>
</extension>
```

Die Erweiterungsdeklarartion des Adapters zum Generalize Declared Type Refactoring erfolgt analog und wird deshalb nicht gezeigt. Die Implementierung dieses Adapters ergibt sich jedoch wie folgt:

```
public class GeneralizeTypeAdapter implements \
  IDeclaredTypeGeneralizationChecker {
    public boolean checkType(ICompilationUnit unit, String selectedType,
      int selectedTypeStart, String selectedName, int selectedNameStart,
      IProgressMonitor monitor) {
        try {
            ChangeTypeRefactoring refactoring = new ChangeTypeRefactoring(
              unit, selectedTypeStart, selectedType.length(), selectedType);
            if (refactoring.checkInitialConditions(monitor).getSeverity() \
              == RefactoringStatus.OK) {
                Collection validTypes = refactoring.computeValidTypes(monitor);
                if (validTypes.size() > 0) {
                    return GENERALIZATION_POSSIBLE;
                }
                else {
                    return GENERALIZATION_OK;
```

```
                }
            }
            else {
                return GENERALIZATION_OK;
            }
        }
        catch (CoreException e) {
            return GENERALIZATION_OK;
        }
    }
    public boolean hasResolution() {
        return true;
    }
    public IMarkerResolution2 getResolution() {
        return new GeneralizeTypeMarkerResolution();
    }
}
```

Die Methode *checkType(...)* greift direkt auf das Refactoring zu, indem sie zunächst eine neue Instanz von *org.eclipse.jdt.internal.corext.refactoring.structure.ChangeType Refactoring* erzeugt und den Konstruktor entsprechend mit den Informationen zum zu prüfenden Typ parametriert. Anschließend muss noch sichergestellt werden, dass alle Vorbedingungen an den Typ erfüllt sind.[21] Diese Funktionalität stellt die Methode *checkInitialConditions(...)* des Refactorings bereit und braucht nicht selbst implementiert zu werden. Im Weiteren wird die Methode *computeValidTypes(...)* aufgerufen, die eine Liste gültiger, alternativer Typen liefert. Enthält die Liste keine Elemente, existiert zum geprüften Typen kein allgemeinerer Typ (wobei das Refactoring nur bereits implementierte Typen betrachten kann – es macht keine Vorschläge, wie ein allgemeiner Typ aussehen könnte).
Die Methode *hasResolution()* liefert stets *true* zurück, da das Refactoring immer eine Lösung zur Generalisierung eines Typen anbieten kann, den es auch prüfen konnte. Die Implementierung der Lösung, die mit der Methode *getResolution()* abgerufen werden kann, wird von der Klasse *GeneralizeTypeMarkerResolution* implementiert, die ebenfalls Teil des Declared Type Generalization Checker Plug-Ins ist.
Das Infer Type Refactoring liefert im Gegensatz zum Generalize Declared Type Re-

[21] Eine Aufstellung der Vorbedingungen ist in Abschnitt 3.2.1 zu finden.

factoring andere Ergebnisse. Es schlägt nicht nur einen existierenden Typ vor, der geeignet ist, sondern liefert auch als Ergebnis zurück, dass ein allgemeinerer Typ geschaffen werden könnte. Im Folgenden wird die Implementierung des zugehörigen Adapters gezeigt, der diese Informationen vom Refactoring abgreifen kann:

```
public class InferTypeAdapter implements \
  IDeclaredTypeGeneralizationChecker {
    public boolean checkType(ICompilationUnit unit, String selectedType,
      int selectedTypeStart, String selectedName, int selectedNameStart,
      IProgressMonitor monitor) {
        ITextSelection textSelection = new TextSelection(selectedNameStart,
          selectedName.length());
        InferTypeAdapterEditorAction inferTypeAdapterAction = \
          new InferTypeAdapterEditorAction();
        inferTypeAdapterAction.selectionChanged(null, textSelection);

        try {
            SingleDeclarationElement declarationElement = \
              inferTypeAdapterAction.getSelectedDeclarationElement(
              textSelection, unit);
            try {
                if (! inferTypeAdapterAction.checkPrerequesites(
                  declarationElement)) {
                    return GENERALIZATION_OK;
                }
                else {
                    InferTypeRunnable inferTypeOperation = \
                      new InferTypeRunnable(declarationElement,
                      unit.getJavaProject(), 1000, new DESet_Key(), false);

                    try {
                        inferTypeOperation.run(monitor);
                    } catch (InferTypeNotExecutable e) {
                        return GENERALIZATION_OK;
                    } catch (InterruptedException e) {
                        return GENERALIZATION_OK;
                    }
```

```
                    if (inferTypeOperation.isInferType2Executable()) {
                        if (! inferTypeOperation.getDeclarationElement(). \
                          getNewDeclaredType().equals(inferTypeOperation. \
                          getDeclarationElement().getDeclaredType())) {
                            return GENERALIZATION_POSSIBLE;
                        }
                        else {
                            return GENERALIZATION_OK;
                        }
                    }
                    else {
                        return GENERALIZATION_OK;
                    }
                }
            }
            catch (NullPointerException e) {
                return GENERALIZATION_OK;
            }
        } catch (InternalCalculationException e) {
            return GENERALIZATION_OK;
        }
    }
    public boolean hasResolution() {
        return true;
    }
    public IMarkerResolution2 getResolution() {
        return new InferTypeMarkerResolution();
    }

    private class InferTypeAdapterEditorAction extends
      AbstractInferTypeAction {...}
}
```

Der Zugriff des Adapters auf das Infer Type Refactoring gestaltet sich nicht ganz so problemlos wie beim Generalize Declared Type Refactoring. Für die Prüfung muss dem Refactoring ein Objekt vom Typ *org.inoJ.inferType.model.model.Single DeclarationElement* übergeben werden. Allerdings ist die Methode, die dieses Ob-

jekt aus den verfügbaren Parametern erstellen kann, im Infer Type Plug-In *private* deklariert und kann nicht aufgerufen werden. Darüber hinaus sind noch weitere Methoden, die für die Prüfung der Vorbedinungen gebraucht werden, ebenfalls *private*. Deshalb wurde in den Adapter eine innere Klasse eingeführt, welche die entsprechende Klasse *org.inoJ.inferType.actions.AbstractInferTypeAction* des Infer Type Refactorings erweitert, um zumindest auf die Methoden zugreifen zu können, die dort als *protected* definiert wurden. Die innere Klasse *InferTypeAdapterEditorAction* enthält außerdem Kopien der *private* Methoden, die gebraucht werden. Die eigentliche Durchführung der Generalisierungsprüfung ist glücklicherweise wieder in der öffentlichen Klasse *org.inoJ.inferType.InferTypeRunnable* implementiert. Zunächst wird eine Instanz dieser Klasse erzeugt, für den Aufruf des Konstruktors wird wieder das Objekt vom Typ *SingleDeclarationElement* benötigt, das die zu prüfende Typdeklaration enthält. Anschließend wird das Refactoring mit dem Aufruf der Methode *run(...)* gestartet. Wenn das Refactoring problemlos ausgeführt werden konnte, erfolgt die Prüfung, ob der gelieferte Typ nicht dem bereits übergebenen entspricht. Dies hätte die Bedeutung, dass die Deklaration bereits allgemein genug ist, ansonsten hat Infer Type einen neuen, optimalen Typ berechnet.
Die Methoden *hasResolution()* und *getResolution()* sind analog denen des Adapters zum Generalize Declared Type Refactoring. Infer Type kann auch stets eine Lösung anbieten; deren Implementierung ist in der Klasse *InferTypeMarkerResolution* zu finden. Diese Klassen werden im Abschnitt 4.4.7 diskutiert.

4.4.5. Code-Prüfungs-Visitors

Im vorliegenden Abschnitt wird beschrieben, wie die Generalisierungsprüfung auf Deklarationselementen aufgerufen wird. Eine zentrale Rolle spielt hierbei die Klasse *Checker*, welche die Prüfung steuert und auch die Initialisierung der Prüfadapter vornimmt. Deshalb enthält die Klasse *DeclaredTypeGeneralizationCheckerBuilder* ein Objekt vom Typ *Checker*, das beim Aufruf der *build(...)*-Methode durch den Aufruf der Factory-Methode der Klasse *Checker* initialisiert wird. Der Factory wird eine Instanz des aktuell zu prüfenden Projektes, ein Objekt vom Typ *IProgressMonitor* und der Builder selbst übergeben. Im Falle der vollständigen Prüfung des Projekts, einem *FULL_BUILD*, ruft der Builder die Methode *performCheck()* der Klasse *Checker* auf. Bei einer inkrementellen Prüfung wird das *Checker*-Objekt über den Konstruk-

tor an den entsprechenden Visitor übergeben.[22] Auf den Visitor für die inkrementelle Prüfung wird weiter unten noch genauer eingegangen. Zunächst wird die Implementierung der Klasse *Checker* vorgestellt, anschließend der Visitor für die vollständige Prüfung. Der Quelltext der Klasse *Checker* ist im Anhang D.3.1 zu finden. Instanzen der Klasse werden, wie bereits erwähnt, mit der Factory-Methode *create(...)* erzeugt. Diese Methode ruft den als *private* deklarierten Konstruktor auf, der aus dem übergebenen Objekt vom Typ *org.eclipse.core.resources.IProject* ein Objekt vom Typ *org.eclipse. jdt.core.IJavaProject* erzeugt.[23] Der Konstruktor ruft auch die Methode *initExternalChecker()* auf, welche zunächst auf die vom Entwickler gemachte Auswahl zum Algorithmus, mit dem geprüft werden soll, zugreift und zur Auswahl gehörende Klasse, die die Prüfung implementiert, instanziiert.
Die Klasse *Checker* enthält zwei öffentliche, überladene Methoden *performCheck*. Die Variante ohne Parameter wird vom Builder aufgerufen und führt dann eine vollständige Prüfung durch, indem sie durch alle Kompilationseinheiten, also Java-Klassen, der Instanz vom Typ *IJavaProject* iteriert und die Methode *performCheck(...)* mit einer Kompilationseinheit als Parameter aufruft. Diese Methode initiiert dann die Prüfung der Kompilationseinheit, indem sie für diese den Abstract-Syntax-Tree erzeugt. Hierbei entsteht ein Objekt vom Typ *org.eclipse.jdt.core.dom.CompilationUnit*, dem über die Methode *accept(...)* die Implementierung eines Visitors übergeben wird. Der Visitor in der Klasse *CheckerASTVisitor* (siehe Anhang D.3.2) traversiert den AST und enthält *visit(...)*-Methoden für die Deklarationselemente *FieldDeclaration*, *MethodDeclaration* und *VariableDeclarationStatement*. Innerhalb dieser Methoden wird zunächst geprüft, ob die Prüfung durch den Benutzer abgebrochen wurde. Die Klasse *Checker* stellt hierfür die Methode *isCheckCancelled()* zur Verfügung. Wurde die Prüfung abgebrochen, liefern die *visit(...)*-Methoden den Rückgabewert *false* zurück, was bewirkt, dass der folgende Ast des AST nicht mehr durchwandert wird. Vom Builder wurde bis in den Visitor ein Parameter durchgereicht, der steuert, ob die besuchten Deklarationselemente geprüft werden. Sollen sie nicht geprüft werden, sammelt der Visitor nur die Anzahl der Deklarationselemente. Diese Information wird für die Berechnung der Fortschrittsanzeige gebraucht; dort muss zuerst eine entsprechende Anzahl Arbeitsschritte allokiert werden, bevor sie benutzt werden kann. Legt der Builder allerdings fest, dass die Prüfung durchgeführt werden soll, rufen die *vi-*

[22] Siehe hierzu den Abschnitt 4.4.3 über die Implementierung des Builders.
[23] Innerhalb der Eclipse-Plattform repräsentiert ein Objekt vom Typ *IProject* jede Art von Projekt, egal in welcher Sprache es geschrieben wurde. Um aber auf die Deklarationselemente eines Java-Quelltextes zugreifen zu können, muss es entsprechend transformiert werden.

sit(...)-Methoden für das gerade besuchte Deklarationselement die entsprechenden Methoden in der Klasse *Checker* auf, welche die Prüfung starten und das Ergebnis der Prüfung auswerten. Ergibt die Prüfung, dass ein Deklarationselement generalisiert werden kann, wird eine Markierung gesetzt. Auf die Problem-Marker wird aber erst im nächsten Abschnitt 4.4.6 genauer eingegangen. Für die Prüfung der Deklarationselemente implementiert die Klasse *Checker* die Methoden *handleField(...)*, *handleMethod(...)* und *handleVariable(...)*, die mit Informationen aus dem Deklarationselement die Prüfmethode *checkType(...)* des Interfaces, das ein Prüfalgorithmus implementieren muss, aufruft.
Die inkrementelle Prüfung eines Quelltextes erfolgt etwas anders. Bei seinem Aufruf bekommt der Builder von der Eclipse-Plattform ein Objekt vom Typ *org.eclipse.core.resources.IResourceDelta* übergeben, das in einer Baumstruktur Referenzen auf die Änderungen der Arbeitsumgebung seit dem letzten Build-Lauf enthält. Dieser Baum kann auch mit einem Visitor traversiert werden, der dem *IResourceDelta*-Objekt über dessen *accept(...)*-Methode im Builder übergeben wird. Dem Visitor in der Klasse *CheckerResourceDeltaVisitor* (siehe Anhang D.3.3) wird über den Konstruktor auch die Instanz der Klasse *Checker* übergeben, da diese ebenfalls die inkrementelle Prüfung kontrolliert. Die *visit(...)*-Methode prüft zunächst stets, ob die Änderung, die sie gerade besucht, eine Java-Datei ist. Für Java-Dateien wird ein Objekt vom Typ *org.eclipse.jdt.core.ICompilationUnit* erzeugt. Diese Kompilationseinheit wird anschließend geprüft, indem sie der Methode *performCheck(...)* der Klasse *Checker* übergeben wird.

4.4.6. Problem-Marker

Um Problem-Marker in der Arbeitsumgebung setzen zu können, muss zunächst der Extension-Point *org.eclipse.core.resources.markers* erweitert werden. Die konkrete Konfiguration in der Datei *plugin.xml* sieht folgendermaßen aus; für eine allgemeine Beschreibung sei auf Anhang C.2.3 verwiesen.

```
<extension id="declaredTypeGeneralizationCheckerProblemMarker"
           name="%AbstractionCheckerProblemMarker"
           point="org.eclipse.core.resources.markers">
    <super type="org.eclipse.core.resources.problemmarker"/>
    <super type="org.eclipse.core.resources.textmarker"/>
```

```
    <persistent value="true"/>
</extension>
```

Die Extension muss neben den üblichen Attributen für die eindeutigen Bezeichner und dem zu erweiternden Punkt die Elemente *super* und *persistent* enthalten. Mit dem Element *super* wird gesteuert, welcher Art die Markierungen sind. Hier wird jeweils mit dem Attribut *type* festgelegt, dass die Markierungen als *problemmarker* in der Liste der Fehler- und Warnmeldungen und zusätzlich als *textmarker* direkt im Quelltext erscheint. Das Element *persistent* gibt an, dass die Markierungen dauerhaft gespeichert werden sollen, d.h. auch nach einem Neustart der Eclipse-Plattform noch vorhanden sind.
Die Klasse *ProblemMarker* im Anhang D.4 bezieht sich beim Setzen der Markierungen auf diese Konfiguration, indem in der Methode *setMarker(...)* beim Erzeugen einer neuen Markierung der eindeutige Bezeichner *id* der Extension angegeben wird. Neben dieser Angabe werden bei der Markierung noch Angaben gemacht, an welchen Positionen im Quelltext die Markierung beginnen und enden soll. Außerdem wird der Schweregrad angegeben. Hierbei handelt es sich um eine Warnung. Der bei den Warnungen angezeigte Text wird auch in diesem Schritt an die Markierung gebunden, ebenso die Zeilenummer der Quelltextdatei, in der die Markierung erscheint. Die als *private* deklarierte Methode *setMarker(...)* wird von den öffentlichen Methoden *prepareFieldOrVariableMarker(...)*, *prepareReturnTypeMarker(...)* und *prepareMethodParameterMarker(...)* aufgerufen. Diese Methoden werden in der Klasse *Checker* aufgerufen, wenn eine Markierung zu setzen ist. Innerhalb dieser Methoden werden die zum Setzen der Markierung benötigten Daten gesammelt und der anzuzeigende Text aufbereitet, der aus einer übersetzbaren Sprachdatei stammt. Die hierfür benötigte Hilfsklasse *Messages* wird im Abschnitt 4.4.8 beschrieben.
Neben den Methoden zum Setzen der Markierungen enthält die Klasse *ProblemMarker* die statische Methode *deleteMarkers(...)*, mit der alle Markierungen in einem übergebenen Projekt gelöscht werden können.

4.4.7. Marker-Resolution

Ein Quick-Fix ist immer an eine bestimmte Art von Markierung gebunden. Dies wird auch in seiner Konfiguration deutlich, da im Element *markerResolutionGenerator* im Attribut *markerType* der eindeutige Bezeichner der hier gesetzten Mar-

kierung steht. Um einen Quick-Fix überhaupt implementieren zu können, muss der Extension-Point *org.eclipse.ui.ide.markerResolution* erweitert werden, die konkrete Konfiguration sieht wie folgt aus; eine allgemeine Abhandlung ist im Anhang C.2.4 zu finden.

```
<extension point="org.eclipse.ui.ide.markerResolution">
    <markerResolutionGenerator
      class="org.intoJ.declaredTypeGeneralizationChecker.core.internal. \
        MarkerResolutionGenerator"
      markerType="org.intoJ.declaredTypeGeneralizationChecker. \
        declaredTypeGeneralizationCheckerProblemMarker"/>
</extension>
```

Die Klasse, welche den Quick-Fix-Generator implementiert wird, im Element *markerResolutionGenerator*, allerdings im Attribut *class*, angegeben. Die Klasse *MarkerResolutionGenerator* implementiert das Interface *org.eclipse.ui.IMarkerResolution Generator2*; ihre Implementierung wird im Anhang D.5.1 gezeigt. Die Klasse *MarkerResolutionGenerator* delegiert die Aufrufe der Methoden *hasResolutions(...)* und *getResolutions(...)* weiter an die Implementierung der Prüfung in den entsprechenden Adaptern. Somit kann die Implementierung der Prüfung selbst die Quick-Fixes bereitstellen, wird aber nicht dazu gezwungen.[24]
Die Implementierung der Quick-Fixes ist Teil der Anbindung an die Prüfung. Die Prüfadapter erzeugen in der Methode *getResolution()* eine Instanz der jeweiligen Klassen, die den Quick-Fix letztlich implementieren. Für das Infer Type Refactoring befindet sich der Quick-Fix in der Klasse *InferTypeMarkerResolution* und für das Generalize Type Refactoring in der Klasse *GeneralizeDeclaredTypeMarkerResolution*. Beide Klassen sind im Anhang D.5.2 bzw. D.5.3 zu finden. Die Klassen müssen das Interface *org.eclipse.ui.IMarkerResolution2* implementieren. Neben den Methoden *getDescription()*, *getImage()* und *getLabel()*, welche die Texte und Bilder für die Anzeige des Quick-Fixes in der Oberfläche liefern, ist die Methode *run(...)* am wichtigsten. Im Fall des Declared Type Generalization Checker Plug-Ins ruft sie auf der aktuellen Markierung das Infer Type bzw. Generalize Declared Type Refactoring auf.

[24] Bei der Implementierung der Prüfung muss es sich nicht zwingend um ein Refactoring handeln, das einen Quick-Fix bereitstellen kann.

4.4.8. Hilfsklassen

Momentan sind alle Texte des Declared Type Generalization Checker Plug-Ins, die in der Benutzeroberfläche angezeigt werden, inklusive aller Fehlermeldungen, in englischer Sprache. Dies ist für den angesprochenen Benutzerkreis auch vollkommen ausreichend, da die meisten Entwickler des Englischen mächtig sein dürften. Außerdem sind ohnehin fast alle Plug-Ins für die Eclipse-Plattform nur in Englisch zu haben. Trotzdem sollten die Mechanismen, welche die Eclipse-Plattform für Lokalisierungen anbietet, nicht ganz unbeachtet bleiben, um sich die spätere Übersetzung nicht gleich zu verbauen. Für die Struktur des Plug-Ins bedeutet dies, dass alle Texte in einer zentralen Sprachdatei abgelegt sind und über einen eindeutigen Bezeichner in den Klassen des Plug-Ins referenziert werden.[25]
Den Zugriff auf die Sprachdatei stellt die Hilfsklasse *Messages* (siehe Anhang D.6.1) zur Verfügung. Hierfür implementiert sie die beiden überladenen, statischen Methoden *getString(...)*. In der einfachen Variante wird die Methode nur mit dem Bezeichner als Parameter aufgerufen und liefert den Text zurück. Der zweiten Variante kann zusätzlich noch ein Array mit *String*-Werten übergeben werden, die für Platzhalter in den Texten aus der Sprachdatei eingesetzt werden.[26] Der Aufbau der Sprachdatei *messages.properties*, die im Package *org.intoJ.declaredTypeGeneralizationChecker.ui.internal* liegt, sieht wie folgt aus:

```
<key>=<value>
```

Als Platzhalter innerhalb des Textes dient der Wert *%VALUE*, der von der zweiten Variante der Methode *getString(...)* ersetzt wird .
Die zweite Hilfsklasse *ExtensionValidator* (siehe Anhang D.6.2) kapselt statische Methoden, die Informationen über andere Plug-Ins aus der Extension-Registry abrufen:

getImplementingExtensions() liefert ein Array mit Referenzen auf die Plug-Ins zurück, die den Extension-Point des Declared Type Generalization Checker Plug-Ins erweitern.

isInferTypeAdapter(...) prüft, ob eine übergebene Referenz zu einem Plug-In, die des Infer Type Refactorings ist.

[25] Auch ohne die Absicht der Übersetzung ist es durchaus sinnvoll, die Texte zentral zu sammeln, anstatt sie über die Quelltexte zu verstreuen.
[26] Gebraucht wird dies beispielsweise, um den Namen eines Typs in der Mitte einer Meldung zu ergänzen.

isInferTypeLoaded() liefert zurück, ob das Infer Type Refactoring installiert und geladen ist.

getExtensionLabel(...) gibt den lesbaren Namen eines Plug-Ins zurück.

getExtensionById(...) liefert eine Referenz auf ein Plug-In mit dem übergebenen Bezeichner zurück.

getClassForInterfaceExtension(...) gibt den Pfad zu der Klasse eines Plug-Ins zurück, welche das Interface des Declared Type Generalization Checker Plug-Ins implementiert.

4.5. Dokumentation

Die Dokumentationen, die für das Declared Type Generalization Checker Plug-In erstellt wurden und ausgeliefert werden, bedienen zwei Zielgruppen. Zum einen unterstützt die Online-Hilfe den Entwickler, der das Plug-In benutzt, indem sie ihm Hilfestellung bei Installation und Anwendung gibt. Andererseits ist die Quelltextdokumentation für die Entwickler gedacht, die das Plug-In selbst weiterentwickeln oder Erweiterungen für das Plug-In implementieren.
Die folgenden Abschnitte beschreiben, mit welchen Hilfsmitteln die Dokumentationen erstellt wurden und wie sie sich in die Eclipse-Plattform einfügen.

4.5.1. Eclipse Online-Hilfe

Hilfeseiten für die Eclipse-Plattform sind einfache HTML-Dateien, in denen auch alle HTML-Stilelemente verwendet werden können.[27] Um der Eclipse-Plattform die Hilfeseiten bekannt zu machen, muss in der Datei *plugin.xml* die entsprechende Konfiguration für die Erweiterung des Extension-Points *org.eclipse.help.toc* eingetragen werden:

```
<extension point="org.eclipse.help.toc">
    <toc file="doc/eclipse/toc.xml"
         primary="true"/>
</extension>
```

[27] Gemeint sind Links, Tabellen, Überschriften, Aufzählungslisten, Bilder usw.

Mit dem Attribut *file* im Element *toc* wird der relative Pfad zur Datei mit dem Inhaltsverzeichnis angegeben. Das Attribut *primary* legt fest, dass das neue Inhaltsverzeichnis auf der ersten Ebene des gesamten Inhaltsverzeichnisses eingehängt werden soll.[28] Die Datei mit dem Inhaltsverzeichnis ist im XML-Format und hat folgenden Aufbau:

```
<?xml version="1.0" encoding="UTF-8"?>
<?NLS TYPE="org.eclipse.help.toc"?>
<toc label="Declared Type Generalization Checker"
     topic="doc/eclipse/html/generalizationChecker.html">
    <topic label="Overview"
           href="doc/eclipse/html/generalizationChecker.html#overview" />
    <topic label="User Guide"
           href="doc/eclipse/html/generalizationChecker.html#guide" />
    <topic label="Credits"
           href="doc/eclipse/html/generalizationChecker.html#credits" />
</toc>
```

Mit den Attributen *label* wird jeweils die Beschriftung des Eintrags im Inhaltsverzeichnis angegeben. Die Schachtelung der Einträge wird erreicht, indem das oberste Element *toc* Elemente *topic* enthält, die wiederum Unterelemente enthalten könnten. Das Element *toc* enthält im Attribut *topic* den relativen Pfad zur HTML-Datei mit der Hilfe. Die Attribute *href* der Elemente *topic* sind Verweise auf Unterpunkte innerhalb der HTML-Datei.
Die folgende Abbildung zeigt, wie sich die Hilfeseite des Declared Type Generalization Checker Plug-Ins in die bestehenden Hilfeseiten integriert:

[28] Es wäre auch möglich, das neue Inhaltsverzeichnis unter einem bereits bestehenden Baum einzuhängen.

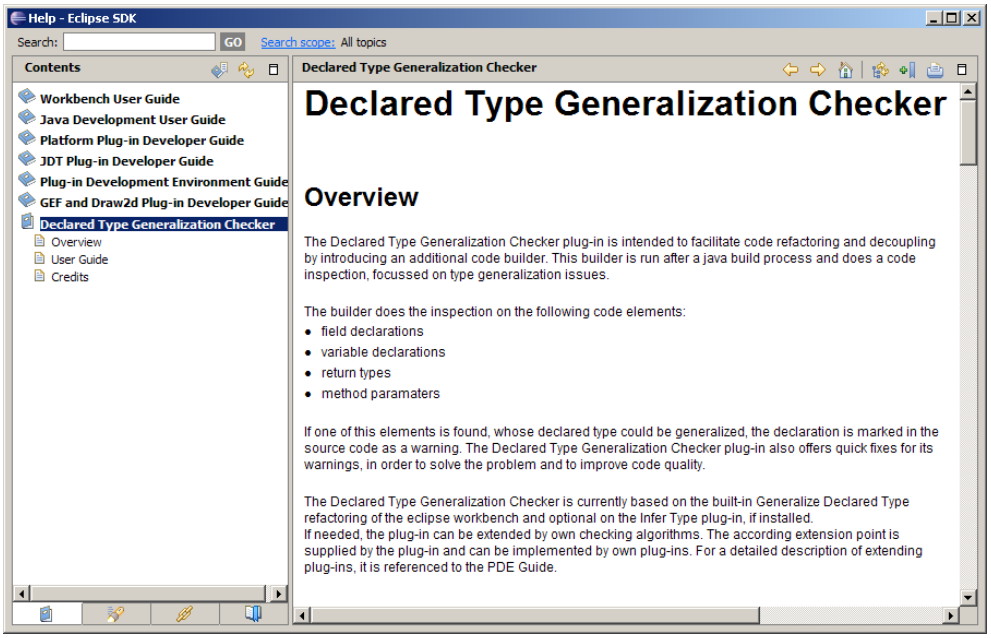

Abbildung 4.6.: Eclipse Online-Hilfe des Plug-Ins

4.5.2. Quelltextdokumentation

Um die Erweiterung oder Weiterentwicklung des Declared Type Generalization Checker Plug-Ins zu erleichtern, wurden alle Klassen, Methoden und Felder dokumentiert. Hierbei wurde auf den De-facto-Standard Javadoc[29] zurückgegriffen, um eine API-Dokumentation des Plug-Ins im HTML-Format zu erstellen. Die HTML-Dateien befinden sich innerhalb des Plug-In-Projektes im Verzeichnis *doc/api*, wobei die Startseite die Datei *index.html* ist. Die Dokumentation beschreibt für alle Klassen grundsätzlich deren Nutzen und Einsatz sowie zusätzlich für Methoden deren Parameter, Rückgabewerte und Exceptions, die geworfen werden können. Die Bedeutung von Feldern in Klassen wird ebenfalls beschrieben. Hierfür wurden die Kommentare im Javadoc-Format verfasst.[30] Da die Javadoc-Kommentare in den Quelltexten im Anhang C dieser Arbeit aus Platzgründen gekürzt wurden, sei hier als Beispiel der Kommentar der Methode *create(...)* in der Klasse *Checker* dargestellt:

[29] Die Homepage des Projekts ist unter http://java.sun.com/j2se/javadoc zu finden.
[30] Auf die entsprechenden Auszeichnungen wird hier nicht näher eingegangen, hierfür sei auf die Dokumentation von Javadoc auf der Homepage des Projektes verwiesen.

```
/**
* This is the factory method for creating a checker instance.
* @param project the project that should be checked
* @param monitor the progress monitor
* @param builder the builder that called the checker
* @return an instance of type Checker
* @throws CoreException
*/
```

Startet man Javadoc und lässt es über die Quelltexte des Declared Type Generalization Checker Plug-Ins laufen, so wird die Dokumentation erstellt. Ein Beispiel für die Dokumentation zeigt die folgende Abbildung:

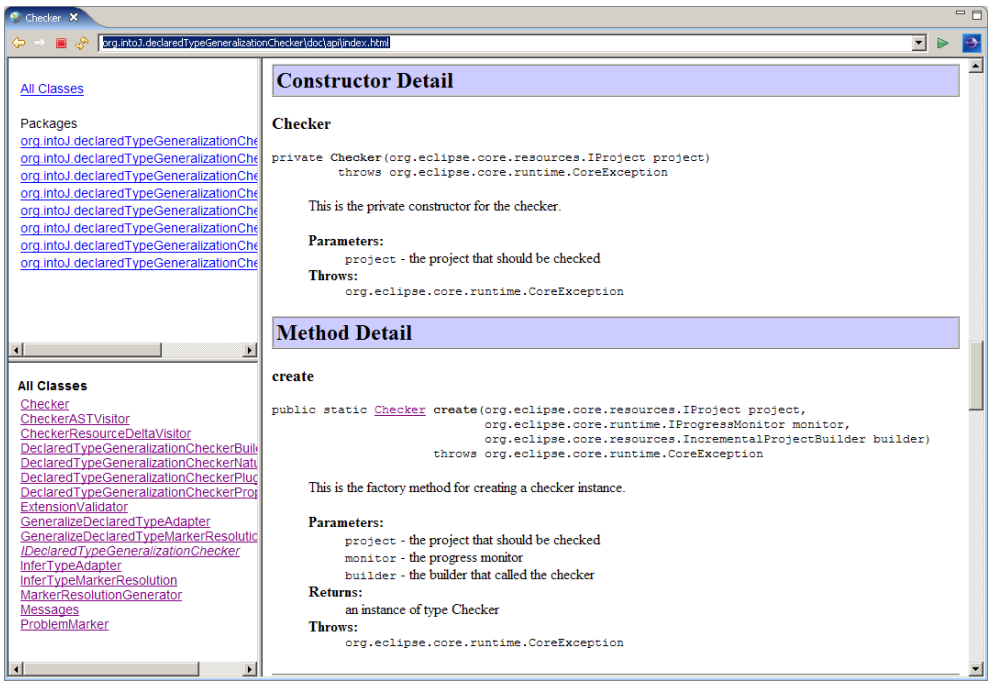

Abbildung 4.7.: Beispiel der API-Dokumentation mit Javadoc

4.6. Deployment und Auslieferung

Die Auslieferung und Installation des fertigen Plug-Ins wäre ohne weitere Schritte zwar möglich, aber umständlich und fehleranfällig. In den folgenden Abschnitten wird

deshalb beschrieben, wie das Plug-In zu einem Feature hinzugefügt wird, das über eine Update-Site einfach beim Anwender installierbar ist. Der interessierte Leser sei hierzu auch auf den Anhang B.4 verwiesen.

4.6.1. Aufbau des Plug-In-Feature

Das Feature wird als separates Projekt angelegt und enthält nur die Datei *feature.xml*. Diese Datei enthält alle nötigen Informationen, um das zugehörige Plug-In installieren zu können:

```
<feature id="org.intoJ.declaredTypeGeneralizationChecker.feature"
         label="Declared Type Generalization Checker Feature"
         version="1.0.0" provider-name="intoJ Team, University of Hagen">
    <description url="http://www.fernuni-hagen.de/ps/prjs/TGC/">
        This feature contains the Declared Type Generalization Checker.
    </description>
    <copyright>
        (c) 2007 by the intoJ Team, University of Hagen.
    </copyright>
    <license> ... </license>
    <url>
        <update label="Declared Type Generalization Checke Update Site"
                url="http://www.fernuni-hagen.de/ps/prjs/TGC/update/"/>
    </url>
    <requires>
        <import plugin="org.eclipse.ui"/>
        <import plugin="org.eclipse.core.runtime"/>
        <import plugin="org.eclipse.core.resources"/>
        <import plugin="org.eclipse.jdt.core"/>
        <import plugin="org.eclipse.jdt.ui"/>
        <import plugin="org.eclipse.ltk.core.refactoring"/>
        <import plugin="org.eclipse.ui.ide"/>
        <import plugin="org.eclipse.jface.text"/>
        <import plugin="org.eclipse.help"/>
    </requires>
    <plugin id="org.intoJ.declaredTypeGeneralizationChecker" version="1.0.0"/>
</feature>
```

Für das Feature selbst werden im Element *feature* neben einem eindeutigen Bezeichner im Attribut *id*, ein lesbarer Name im Attribut *label*, die Versionsnummer der Feature im Attribut *version* und der Name des Veröffentlichers im Attribut *providername* angegeben. Die Daten in den Elementen *description*, *copyright* und *license* werden während der Installation angezeigt. Dabei wird zur Beschreibung des Plug-Ins ein Link auf eine Internetseite angezeigt, auf welcher der Anwender die vollständige Beschreibung einsehen kann. Konfiguriert wird dies mit dem Attribut *url* im Element *description*. Der Inhalt des Elements *license* wird im Installationsdialog als Lizenzbestimmung des Plug-Ins angezeigt und muss explizit akzeptiert werden, um die Installation fortzusetzen. Das Element *url* enthält wiederum das Element *update*. In dessen Attributen *label* und *url* wird ein Beschreibungstext und die Adresse der Internetseite angegeben, auf der die Eclipse-Plattform automatisch nach Aktualisierungen für das Plug-In suchen kann.
Bevor das Plug-In überhaupt installiert wird, prüft der Installationsdialog, ob alle Abhängigkeiten des Declared Type Generalization Checker Plug-Ins zu anderen Plug-Ins erfüllt sind. Können nicht alle Abhängigkeiten gelöst werden, bricht die Installation mit einer entsprechenden Meldung ab. Dies hat den Vorteil, dass das Plug-In nicht vermeintlich korrekt installiert wird, sich aber dann nicht starten lässt. Die Abhängigkeiten werden im Element *requires* und dort jeweils in den Elementen *import* im Attribut *plugin* mit dem Bezeichner des abhänigen Plug-Ins angegeben.
Die Verbindung der Feature mit dem Plug-In wird im Element *plugin* konfiguriert. Im Attribut *id* ist der Bezeichner und im Attribut *version* die Versionsnummer des Declared Type Generalization Checker Plug-Ins eingetragen.

4.6.2. Installation per Update-Site

Die Update-Site des Declared Type Generalization Checker Plug-Ins wird ebenfalls als separates Projekt angelegt. Es enthält alle Inhalte, die später unter der spezifizierten Adresse auf einem Webserver abgelegt werden. Die zentrale Datei ist dabei *site.xml*. Diese Datei wird vom Installationsdialog der Eclipse-Plattform abgerufen und verarbeitet:

```
<site>
    <description url="http://www.fernuni-hagen.de/ps/prjs/TGC/update/">
        Declared Type Generalization Checker
```

```
    </description>
    <feature url="features/org.intoJ.declaredTypeGeneralizationChecker. \
              feature_1.0.0.jar"
             id="org.intoJ.declaredTypeGeneralizationChecker.feature"
             version="1.0.0"/>
</site>
```

Das Wurzelelement *site* enthält im Element *description* eine kurze Beschreibung des Feature, das über diese Seite installiert wird. Zusätzlich wird mit dem Attribut *url* die Adresse zur Update-Site spezifiziert. Der Bezug zum Feature wird mit dem Element *feature* hergestellt. Es enthält im Attribut *url* die relative Adresse des Archivs des Features, das installiert werden muss. Darüber hinaus wird im Attribut *id* der eindeutige Bezeichner des Feature und im Attribut *version* deren Version angegeben. Wird für das Projekt des Features ein Build angestoßen, wird im Wurzelverzeichnis des Projekts die Datei *index.html* erstellt. Sie dient dazu, einen Link zum Download der Feature anzuzeigen, falls die Adresse der Update-Site direkt mit einem Browser geöffnet wird. Während des Builds werden in den Verzeichnissen *features* und *plugins* die entsprechenden Archive aus den Projekten des Feature und des Declared Type Generalization Checker Plug-Ins erzeugt und dort abgelegt.
Der Ablauf der Installation eines Features wird im Anhang B.4 dargestellt.

5. Fallstudien beim Einsatz des Declared Type Generalization Checker Plug-Ins

Nachdem im vorherigen Kapitel die Realisierung des Declared Type Generalization Checker Plug-Ins detailliert beschrieben worden ist, wird in den folgenden Abschnitten darauf eingegangen, wie sich das Plug-In Im Einsatz unter realen Bedingungen verhält. Hierfür wird zunächst geschildert, in welchem Umfeld und unter welchen Voraussetzungen die Analyse durchgeführt wurde. Anschließend werden die Ergebnisse, die erzielt wurden, dargestellt und diskutiert, insbesondere unter den Gesichtspunkten der Laufzeit und ihrer Aussagekraft.
Durchgeführt wurden die Analysen auf einem aktuellen Notebook mit einem Dualcore-Prozessor mit 2,26 GHz Taktung und 1 GB Hauptspeicher. Die Eclipse-Plattform wurde so konfiguriert, dass sie zwischen 512 und 768 MB des Hauptspeichers anfordern kann.

5.1. Anwendungsszenarien

Die Analyse des Plug-Ins wurde an drei Projekten getestet. Im ersten Szenario wurde die Generalisierungsprüfung auf einem kommerziellen Projekt durchgeführt. Dieses Projekt ist eine klassische Desktop-Anwendung, die in Java nach den gängigen Designprinzipien mit einer mehrschichtigen Architektur konzipiert wurde. Es handelt sich dabei um eine Anwendung, die mit einer Datenbank über JDBC[1] kommuniziert und die Daten in einer Logik-Schicht aufbereitet. Die Darstellung erfolgt in einer gekapselten GUI-Komponente, die auf Swing[2] aufbaut. Neben der Logik-Schicht exis-

[1] JDBC ist eine API um Datenbankzugriffe zu kapseln und zu abstrahieren.
[2] Swing ist eine Java-API zur Programmierung von grafischen Oberflächen.

tieren Komponenten für den Datenimport und -export. Die Teile der Anwendung wurden auf acht Eclipse-Projekte aufgeteilt. Die folgende Tabelle vermittelt einen Überblick über Größe und Aufteilung der geprüften Quelltexte. Die Aufstellung soll nur einen quantitativen Eindruck der Software vermitteln, auf die Interna kann hier nicht näher eingegangen werden.

Projekt	**Anzahl Pakete**	**Anzahl Klassen**	**Anzahl Deklarationselemente**
dongle	3	9	301
database	9	53	1795
unit	2	6	416
importexport	8	63	1605
interface	3	15	324
view	25	211	8212
moduls	11	151	7515
basic	8	26	799

Tabelle 5.1.: Größe und Aufteilung des kommerziellen Projekts

Um die Ergebnisse der Analyse verifizierbar und vergleichbar zu machen, wurde das Plug-In zusätzlich an zwei öffentlich verfügbaren Projekten erprobt. Hierfür wurde das JUnit-Projekt[3] in der Version 3.8 und JHotDraw[4] in der Version 6.0 beta 1 ausgewählt.

Projekt	**Anzahl Pakete**	**Anzahl Klassen**	**Anzahl Deklarationselemente**
JUnit	12	93	1501
JHotDraw	29	482	7788

Tabelle 5.2.: Größe und Aufteilung von JUnit und JHotDraw

Die Generalisierungsprüfung wurde bei den Projekten, außer JHotDraw, jeweils mit den Algorithmen des Generalize Declared Type und des Infer Type Refactorings durchgeführt. JHotDraw kann nur mit dem Generalize Declared Type Refactoring geprüft werden, da die derzeit verfügbare Infer Type Implementierung bei diesem Projekt an ihre Grenzen stößt.
Bei den Analysen wurde zunächst das komplette Projekt geprüft. Anschließend wur-

[3]Details und Quelltexte sind auf http://www.junit.org verfügbar.
[4]Die URL der Homepage des Projekts mit weiteren Ausführungen lautet: http://www.jhotdraw.org

den kleine Teile der Projekte geändert, um auch Vergleichswerte für die inkrementelle Prüfung zu bekommen.

5.2. Laufzeitverhalten

Die Dauer der Prüfung hängt direkt von der Größe der Projekte ab. Da jedes Deklarationselement für sich geprüft werden muss, dauert die Prüfung insgesamt länger, je mehr Deklarationselemente zu prüfen sind. Es handelt sich aber um keinen linearen Anstieg, d.h., es kann nicht pauschal gesagt werden, dass die Prüfung eines Projekts mit doppelt sovielen Klassen auch doppelt so lange dauert. Darüber hinaus ist die Zeit für die Prüfung eines Deklarationselements nicht konstant. Die Prüfung komplexer Typen beansprucht wesentlich mehr Zeit als die von simplen Typen, da die komplette Typhierarchie aufgebaut und aus ihr geeignete Typen für den Kontext ausgewählt bzw. diese erst berechnet werden müssen.
Die folgende Tabelle zeigt die gemessenen Zeiten bei der Prüfung des jeweils kompletten Projekts in Abhängigkeit vom verwendeten Algorithmus:

Projekt	**Dauer in Minuten Generalize Declared Type**	**Dauer in Minuten Infer Type**
dongle	0,2	0,1
database	10,9	54,3
unit	2,1	5,0
importexport	9,0	35,5
interface	1,2	3,5
view	220,1	104,5
moduls	90,0	556,7
basic	2,75	10,8
JUnit	3,4	7,1
JHotDraw	30,6	-

Tabelle 5.3.: Vergleich der Laufzeiten zwischen beiden Algorithmen

Die folgende Abbildung stellt die Ergebnisse aus Tabelle 5.3 grafisch dar. Die Grafik verdeutlich auch den nicht-linearen Anstieg der Prüfungsdauer abhängig von der Projektgröße. Um die Grafik nicht zu verzerren, wurden die großen Projekte darin nicht berücksichtigt.

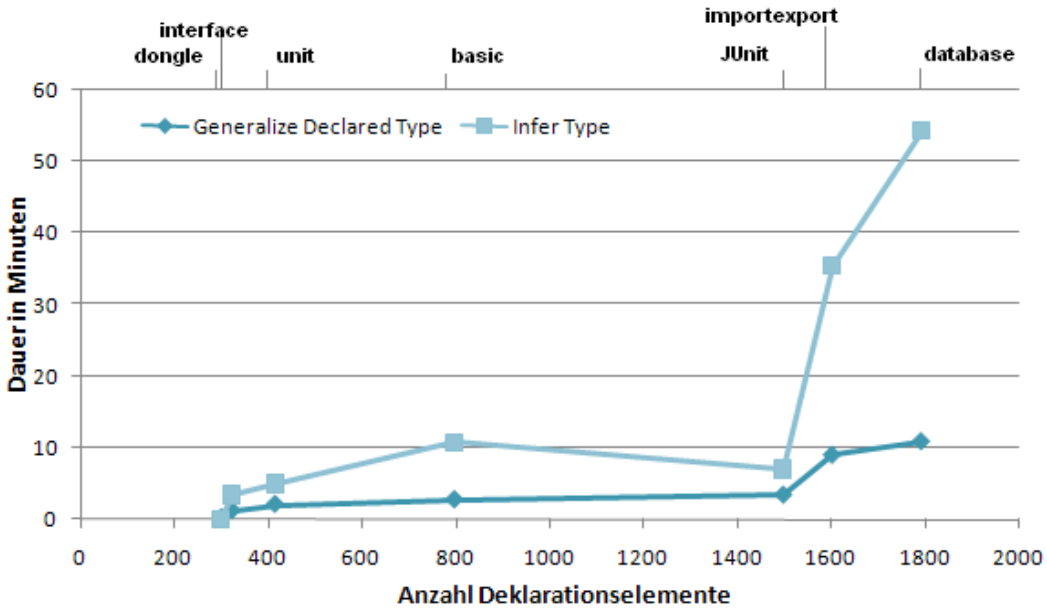

Abbildung 5.1.: Gegenüberstellung der Prüfergebnisse in Abhängigkeit von der Projektgröße

Der Vergleich der beiden Algorithmen zeigt, dass das Infer Type Refactoring meist wesentlich länger für die Generalisierungsprüfung braucht als das Generalize Declared Type Refactoring.[5] Dies lässt sich damit begründen, dass sich die Ziele, die beide Algorithmen jeweils verfolgen, stark unterscheiden. Während sich das Generalize Declared Type Refactoring damit zufrieden gibt, festzustellen, dass ein allgemeinerer Typ in der Supertyphierarchie verfügbar ist, geht das Infer Type Refactoring einen Schritt weiter. Insbesondere wenn kein allgemeiner Typ vorhanden ist, berechnet es einen neuen, minimalen Typ, was ungleich aufwändiger ist.
Die Ausnahme bei den Laufzeiten stellt hier das Projekt view dar. Seine Prüfung dauert mit dem Infer Type Refactoring nur knapp halb so lange wie mit dem Generalize Declared Type Refactoring. Begründen lässt sich dies mit der Struktur des Projektes. Es beinhaltet die GUI-Komponente der Software, die auf Swing basiert. Swing enthält intern schon viele Interfaces, die dementsprechend oft in Deklarartionen im Projekt vorkommen.[6] Diese Interfacetypen sind meist schon maximal generalisiert, weshalb bei der Prüfung mit dem Infer Type Refactoring die aufwändige Berechnung

[5] Ein genereller Faktor für den Unterschied der Laufzeiten lässt sich nicht bestimmen, wie sich zeigt, ist dies abhängig von den Eigenschaften der Projekte.
[6] Für Details und weitere Ausführungen sei auf Steimann und Mayer (2005) verwiesen.

minimaler Typen kürzer ausfällt. Darüber hinaus enthält das Projekt viele Deklarationen mit Basistypen, die nur vom Generalize Declared Type Refactoring behandelt werden können. Eine ähnliche Tendenz ist auch beim Projekt dongle zu erkennen, da es ebenfalls eine kleine Swing-GUI enthält; durch die Projektgröße ist die Tendenz aber nicht so stark ausgeprägt.
Einen Ausreißer bei der Laufzeit bemerkt man beim Projekt moduls. Seine Prüfung dauert mit dem Infer Type Refactoring mit Abstand am längsten, obwohl es sogar etwas kleiner ist als das Projekt view. Die Begründung hierfür liegt darin, dass es sich um ein monolithisches Projekt handelt, bei dem kaum Interfaces verwendet werden, deshalb muss das Infer Type Refactoring für die meisten Deklarationen minimale Typen erstmal berechnen.
Aus der Tabelle 5.3 lässt sich generell ableiten, dass die vollständige Prüfung eines Projektes zeitintensiv ist. Deshalb unterstützt das Declared Type Generalization Checker Plug-In den Modus der inkrementellen Prüfung, bei der nur die Differenz seit der letzten Prüfung erneut einer Prüfung unterzogen wird.
Aus der Erfahrung des Entwicklungsalltags lässt sich sagen, dass die Zeitspannen und auch die Änderungen zwischen Buildläufen ziemlich klein sind und sich auf wenige Minuten und kurze Codeabschnitte beschränken. Selbst wenn zwischen den Buildläufen etwa zehn Klassen angelegt und ausprogrammiert werden,[7] zeigt Tabelle 5.3 unter Rückblick auf Tabelle 5.1 für das Projekt dongle (mit insgesamt nur 9 Klassen), dass die Prüfaufwände hierfür bei einer vollständigen Prüfung moderat bleiben.
Die folgende Tabelle 5.4 zeigt das Verhalten des Plug-Ins, wenn bei beispielhaft ausgewählten Projekten jeweils Änderungen an fünf Klassen vorgenommen wurden. Es werden wiederum beide Algorithmen gegenübergestellt.

Projekt	**Dauer in Minuten Generalize Declared Type**	**Dauer in Minuten Infer Type**
database	0,80	3,03
view	7,48	0,58
moduls	2,87	11,48
JUnit	0,23	0,50
JHotDraw	1,70	-

Tabelle 5.4.: Vergleich der Laufzeiten bei inkrementeller Prüfung

[7] Das ist zwar bereits sehr unwahrscheinlich, soll aber dennoch Großen Programmierern nicht abgesprochen werden.

Wie ein Vergleich zu den Laufzeiten aus Tabelle 5.3 zeigt, sind die Laufzeiten bei inkrementeller Prüfung wesentlich kürzer. Es wird aber deutlich, wie stark die Dauer von der Komplexität und vom Umfang des Gesamtprojektes abhängt und kaum von der Anzahl der zu prüfenden Klassen.

5.3. Qualtität und Aussagekraft der Ergebnisse

Die Qualität der Ergebnisse, welche die Generalisierungsprüfung liefert, lässt sich am besten daran messen, wie viele Typdeklarationen in den Projekten gefunden wurden, die verallgemeinert werden können. Dies stellt die folgende Tabelle für beide Algorithmen dar.

Projekt	**Generalize Declared Type**	**Infer Type**
dongle	26	4
database	569	804
unit	165	114
importexport	454	657
interface	122	115
view	1867	1682
moduls	2623	3846
basic	369	495
JUnit	205	315
JHotDraw	544	-

Tabelle 5.5.: Gegenüberstellung der Anzahl gefundener generalisierbarer Deklarationen

Die Aufstellung zeigt, dass das Infer Type Refactoring meist mehr generalisierbare Deklarationselemente findet als das Generalize Declared Type Refactoring. Dies liegt daran, dass wie im Abschnitt 5.2 bereits erwähnt das Infer Type Refactoring neue minimale Typen berechnet, anstatt nur bestehende auszuwählen. Rein deshalb würde man aber eine noch größere Abweichung der Ergebnisse erwarten. Jedoch fällt die Abweichung bei den meisten Projekten geringer aus, da beide Refactorings leicht abweichende Voraussetzungen bzgl. der prüfbaren Deklarationen haben.[8] Der größte

[8]siehe hierzu auch die Abschnitte 3.2.1 und 3.2.2

Unterschied liegt aber darin, dass das Generalize Declared Type Refactoring auch Basistypen prüfen kann, die sehr häufig in Deklarationen vorkommen und auch meistens generalisiert werden können; und sei es nur zum allgemeinsten Typ *Object*. Deshalb zeigt sich in Tabelle 5.5 bei Projekten, die viele Deklarationen mit Basistypen enthalten, dass das Generalize Declared Type Refactoring sogar mehr generalisierbare Deklarationen findet als das Infer Type Refactoring. Dies ist bei den Projekten dongle, unit, view und moduls zu beobachten. Bei den Projekten dongle und view kommt noch hinzu, dass – wie bereits in Abschnitt 5.2 erwähnt – häufig Interfaces verwendet werden und eine weitere Generalisierung somit oft nicht notwendig bzw. möglich ist. Eine Aussage zum Nutzen der automatisierten Generalisierungsprüfung liefert die Tabelle 5.6, in der prozentual der Anteil der generalisierbaren Deklarationen der einzelnen Projekte gezeigt wird.

Projekt	**Generalize Declared Type**	**Infer Type**
dongle	8,6 %	1,3 %
database	31,7 %	44,8 %
unit	39,7 %	27,4 %
importexport	28,3 %	40,9 %
interface	37,7 %	35,5 %
view	22,7 %	20,5 %
moduls	34,9 %	51,2 %
basic	46,2 %	62,0 %
JUnit	13,7 %	21,0 %
JHotDraw	7,0 %	-

Tabelle 5.6.: Prozentualer Anteil der generalisierbaren Typdeklarationen

Es wird deutlich, dass in den meisten Projekten viele Deklarationen generalisiert werden können; im Einzelfall sogar über 50%. Es gibt aber auch auffallend positive Beispiele, bei denen nur wenig Bedarf an weiterer Generalisierung besteht. Es lässt sich dadurch auch ein direkter Zusammenhang zur Laufzeit der Prüfung erkennen: Je mehr Deklarationen eines Projketes bereits maximal generalisiert sind, umso kürzer fällt die Prüfung des Projektes aus. Obwohl JHotDraw eines der größten Projekte im Vergleich ist, lässt es sich – wie in Tabelle 5.3 gezeigt – mit moderatem Zeitaufwand einer kompletten Prüfung unterziehen. Ähnliches gilt auch für JUnit, womit sich auch der Knick Abbildung in 5.1 erklären lässt.
Als Beispiel für die unterschiedlichen Ansätze beider Refactorings bei den gemachten

Vorschlägen zur Generalisierung, sei das Ergebnis der Prüfung einer Typdeklaration aus dem Projekt moduls gezeigt. Die geprüfte Deklaration sieht wie folgt aus:

```
AsciiFileLoaderTask afTask = new AsciiFileLoaderTask();
```

Das Generalize Declared Type Refactoring macht für die Generalisierung folgenden Vorschlag:

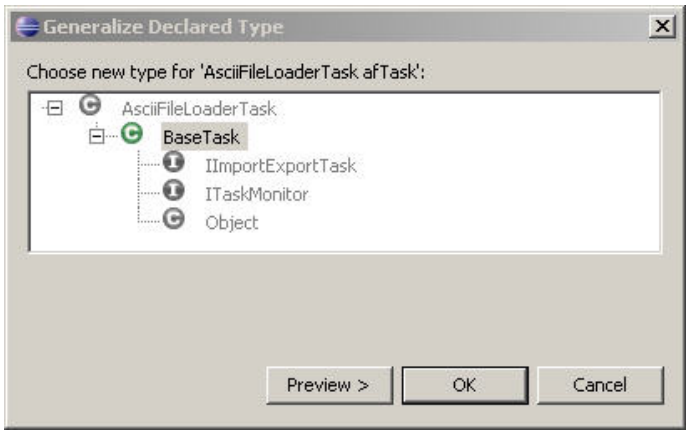

Abbildung 5.2.: Typgeneralisierungsvorschlag des Generalize Declared Type Refactorings

Für die Deklaration könnte somit auch die Klasse *BaseTask* verwendet werden. Der Vorschlag des Infer Type Refactorings sieht jedoch vor, für die Klasse *AsciiFileLoaderTask* ein neues Interface einzuführen und dieses dann in der Deklaration zu verwenden, was nicht nur die Typgeneralisierung verbessern würde, sondern darüber hinaus eine noch stärkere Entkopplung der Klassen bedeutet.

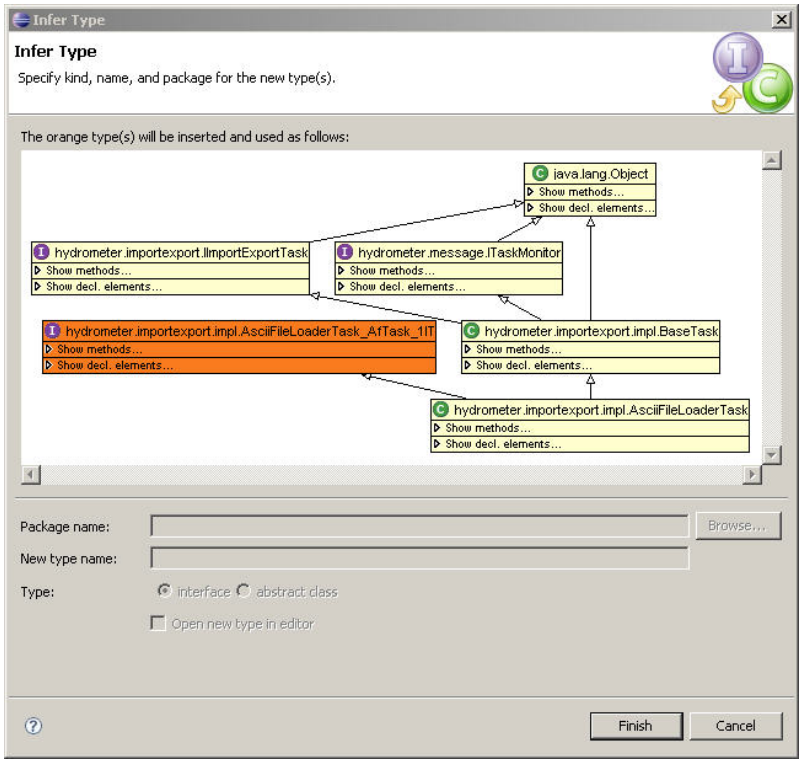

Abbildung 5.3.: Typgeneralisierungsvorschlag des Infer Type Refactorings

5.4. Bewertung der Fallstudie

Die gemessenen Zahlenwerte erlauben natürlich keine generelle Aussage darüber, wie gut oder wie schlecht die Idee des Einsatzes von allgemeinen Typen bei Deklarationen in der Softwareentwicklung Einzug gehalten hat. Es ist aber eine Tendenz klar erkennbar: In durchschnittlich einem Drittel der Typdeklarationen werden Klassen als Typ verwendet, wo allgemeinere Typen verwendet werden könnten.
Aus den Messungen ist auch erkennbar, dass nicht alle Softwareprojekte in gleichem Maße zu konkrete Typdeklarationen enthalten. Die Projekte JHotDraw, JUnit und view schneiden vergleichsweise gut ab. Das Projekt dongle schneidet bei der Analyse ebenfalls gut ab, es handelt sich aber auch um das kleinste Projekt im Vergleich und ist deshalb nicht unbedingt repräsentativ.
Die Eigenheiten der anderen Projekte werden in diesem Zusammenhang nicht weiter vertieft. Sie dienen nur dazu, aufzuzeigen, dass Handlungsbedarf besteht, Entwickler

bei der täglichen Arbeit zu unterstützen, indem ihnen eine Möglichkeit gegeben wird, Aspekte der Typgeneralisierung immer wieder automatisiert prüfen zu lassen.
Die Fallstudien haben ebenfalls wichtige Erkenntnisse in Bezug auf die Laufzeiten der Prüfungen geliefert. Letztlich kann anhand derer entschieden werden, welcher Einsatz des Plug-Ins sinnvoll ist. Der Nutzen des guten Vorsatzes, auf Typgeneralisierung achten zu wollen, wird rasch verringert, wenn das gewählte Hilfsmittel den Arbeitsfluss behindert und verzögert. In den Fallstudien wurde deutlich, dass eine komplette Prüfung eines Projekts umso zeitintensiver ist, je größer ein Projekt wird und somit die Zahl der zu prüfenden Deklarationselemente und die Komplexität zunimmt.
Akzeptable Laufzeiten sind bei Projekten mit bis etwa 100 Klassen möglich, bei denen eine vollständige Prüfung etwa zehn Minuten dauert. Solche Prüfungen können hin und wieder gestartet werden und liefern relativ rasche Ergebnisse. Bei größeren Projekten sollte eine vollständige Prüfung genauer bedacht sein, da sie durchaus bis zu einer Stunde dauern kann. Abhängig von der Projektart und der Qualität der zu prüfenden Programme können für eine Prüfung aber auch mehrere Stunden gebraucht werden – dies lässt sich im normalen Arbeitsprozess nicht mehr unterbringen. Es empfiehlt sich dann, die Prüfungen über Nacht oder abgesetzt auf einem separaten Rechner durchzuführen. Die Häufigkeit, in der man den Aufwand betreibt, sollte sich am Stellenwert der Typgeneralisierung orientieren, der ihr im jeweiligen Projekt zugestanden wird.
Inkrementelle Prüfungen eignen sich auch nur bei Projekten erstgenannter Größe. Hier sind die Laufzeiten noch im Bereich unter einer Miunte. Bei größeren Projekten muss man bedenken, dass die Prüfzeit jedesmal anfällt, wenn ein Projekt zum Test übersetzt wird. Gerade dies erfolgt häufig im täglichen Entwicklungsgeschehen.
Als Schlussfolgerung aus der Fallstudie lässt sich feststellen, dass das Declared Type Generalization Checker Plug-In nur bei kleineren Projekten in den Entwicklungsprozess direkt eingebunden werden kann, um fortlaufend auf mögliche Generalisierungen hinzuweisen. Bei größeren Projekten muss die Prüfung abgesetzt erfolgen. Dies bedeutet, dass der Declared Type Generalization Checker zu einem definierten Zeitpunkt dem Projekt hinzugefügt und die Prüfung dann außerhalb der regulären Entwicklungszeit gestartet wird. Nach Abschluss der Prüfung müssen die Ergebnisse bewertet und gegebenenfalls eingearbeitet werden, bevor der Declared Type Generalization Checker wieder vom Projekt entfernt wird und die Entwicklungsarbeit fortgesetzt werden kann.

6. Fazit und Ausblick

Nachdem in den vorhergehenden Kapiteln die Entwicklung des Generalize Declared Type Checker Plug-Ins und auch erste Fallstudien zum Einsatz beschrieben wurden, folgt in diesem Kapitel eine kritische Bewertung des Plug-Ins, bei der die Chancen, aber auch die Risiken beleuchtet werden. Abschließend wird darauf eingegangen, welche zukünftigen Erweiterungen bereits jetzt denkbar sind und welche Aspekte dabei bedacht werden müssen.

6.1. Chancen und Risiken beim Einsatz

Der Vorteil der automatisierten Generalisierungsprüfung als Builder liegt darin, dass die Prüfung umfassender und konsequenter durchgeführt wird als dies manuell möglich wäre. Die bisher beschriebenen Refactorings Generalize Declared Type und Infer Type müssen bewusst auf Deklarationselemente ausgeführt werden. Dabei ergibt sich die Gefahr, dass Deklarationselemente übersehen werden oder manche Prüfungen schlicht der Bequemlichlkeit zum Opfer fallen. Gerade die automatische Prüfung aller Deklarationselemente in Verbindung mit der Anzeige der Prüfungsergebnisse direkt im Quelltext wirkt diesem entgegen. Der Entwickler kann unkompliziert die Quelltexte prüfen und kann auf die generalisierbaren Deklarationen unmittelbar das Refactoring anwenden. Dieses Vorgehen sollte mittelfristig das Bewusstsein von Entwicklern für die Verwendung von allgemeineren Typen schärfen, was bedeuten würde, dass grundsätzlich schon bei der Entwicklung auf allgemeine Typen geachtet würde. Am effektivsten lässt sich das Declared Type Generalization Checker Plug-In bei kleineren Projekten einsetzen, da hier unmittelbar bei der Entwicklung mögliche Generalisierungen aufgezeigt werden. Außerdem hält sich dann die Anzahl der vorgeschlagenen Generalisierungen in Grenzen. Größere Projekte enthalten auch viele Deklarationselemente, wodurch sich je nach Codequalität viele mögliche Generalisierungen ergeben. Dies birgt die Gefahr, dass der Entwickler sich mit Warnungen

überschüttet fühlt, insbesondere wenn die Prüfläufe, aus zeitlichen Gründen, abgesetzt von der Entwicklung stattfinden. Da sich die Implementierung der Warnungen aber an die Eclipse-Vorgaben hält, können die Meldungen gefiltert oder ausgeblendet werden. Im Gegensatz zu Compilerfehlern kann das Programm trotz der Warnungen ausgeführt werden. Demnach können die Generalisierungen der Deklarationen auch schrittweise durchgeführt werden. Jedoch besteht die Gefahr, dass sich die Generalisierungsvorschläge anhäufen und nachgezogen werden müssen. Hier fehlt dann, wie aber grundsätzlich bei abgesetzten Prüfläufen, der unmittelbare Bezug zum Entwickelten, da die monierten Quelltexte bereits vor einiger Zeit geschrieben wurden.
Bei der Verwendung des Plug-Ins muss man sich auch bewusst machen, welche Ziele die Refactorings haben, auf die für die Prüfung zurückgegriffen wird. Die Entscheidung, welches eingesetzt wird, legt letztlich das Ergebnis fest, in welchem Maße die Typdeklarationen verallgemeinert werden. Da das Infer Type Refactoring Typen für Deklarationen berechnen kann und somit neue Interfaces einführt, liefert es in Bezug auf die Entkopplung von Klassen gute Ergebnisse. Es macht aber keinerlei Vorschläge für Basistypen von Java, da es diese nicht verarbeiten kann. Hingegen macht das Generalize Declared Type Refactoring auch Vorschläge für Basistypen. Dies führt bei Quelltexten, die mit dem Generalize Declared Type Refactoring geprüft worden sind, zu einer Vielzahl von Warnungen.
Obwohl das Plug-In die Prüfung automatisiert und die Möglichkeit bereitstellt, die Refactorings aufzurufen, ist die Bearbeitung der Warnungen mühsame Handarbeit. Insbesondere wenn viele Deklarationen mit dem gleichen Typ moniert werden, muss für jede Deklaration einzeln das Refactoring ausgeführt werden, obwohl das Ergebnis meist bekannt ist. Unterstützung an dieser Stelle bietet das Refactoring Use Supertype Where Possible, das auf einen Typ angewendet versucht, in allen Deklarationen den Supertyp einzusetzen.

6.2. Anbindung von Algorithmen zur Generalisierungsprüfung

Mit den Refactorings Generalize Declared Type und Infer Type sind bereits zwei gute Plug-Ins für die Generalisierungsprüfung angebunden. Da das Declared Type Generalization Checker Plug-In durch das Generalisierungsprüfungs-Interface und dessen Veröffentlichung als Extension-Point erweiterbar gestaltet wurde, ist damit zu rech-

nen, dass davon zukünftig auch Gebrauch gemacht wird. Aktuell liegen allerdings noch keine konkreten Refactorings als Implementierung vor, die angebunden werden könnten.
Bei der Implementierung weiterer Anbindungen spielt ein performanter Zugriff auf den Algorithmus zur Generalisierungsprüfung eine wichtige Rolle. Dies hat sich gerade bei der Anbindung des Infer Type Refactorings deutlich gezeigt. Die Erkenntnis hierbei war, dass für die Anbindung die Meldung, ob ein Typ generalisiert werden kann oder nicht, reicht. Alles andere darüber hinaus findet zunächst keine Anwendung. Der Vorschlag alternativer Typen muss erneut berechnet werden, wenn der Entwickler das Refactoring explizit für eine monierte Deklaration startet. Diese vollständige Berechung bewirkt während der Prüfung nur eine unnötig lange Laufzeit. Deshalb sollte auch darauf geachtet werden, dass die Funktionalität der Generalisierungsprüfung von Refactorings explizit und gut gekapselt veröffentlicht wird. So ist zu erwarten, dass sich zukünftig bessere Laufzeiten bei der Prüfung aller Deklarationen erreichen lassen, was auch bei größeren Projekten zügige Prüfungen erlaubt.

6.3. Parallele Prüfung mit mehreren Algorithmen

Es wurde bereits deutlich, dass bei einer Prüfung der Typgeneralisierung die Ergebnisse von den Eigenschaften des zur Prüfung verwendeten Algorithmus abhängen. Für den Entwickler könnte es deshalb hilfreich sein, sich bei der Prüfung nicht nur auf einen Algorithmus festlegen zu müssen, sondern parallel mit weiteren Algorithmen prüfen zu können. Dies hätte den Vorteil, dass gleichzeitg mehr Aspekte untersucht werden können und direkt vergleichbare Ergebnisse vorliegen. Zwar kann der Entwickler bereits jetzt den Algorithmus, der für die Prüfung verwendet wird, einstellen, aber um ein Projekt mit einem anderen Algorithmus prüfen zu lassen, muss die Prüfung nochmals für das ganze Projekt erfolgen. Dabei gehen die Ergebnisse der vorherigen Prüfung verloren, womit auch kein direkter Vergleich möglich und das Vorgehen umständlich und unter Umständen zeitaufwändig ist.
Momentan ist eine Implementierung der parallelen Prüfung nicht sinnvoll. Dadurch würden die Laufzeiten für Prüfungen selbst bei kleinen Projekten stark zunehmen. Der ausschlaggebende Faktor ist hierbei die Prüfung innerhalb der Refactorings, da diese abhängig von den Zugriffsmöglichkeiten und Interna der angesprochenen Refactorings ist. Hierzu sei aber auf den vorherigen Abschnitt 6.2 verwiesen.

6.4. Unterstützung der abgesetzten Prüfung

In Abschnitt 5.4 wurde festgestellt, dass für größere Projekte die Prüfung des Grades der Generalisierung abgesetzt von der Entwicklung erfolgen sollte. Bisher bietet das Declared Type Generalization Checker Plug-In keine vollständig ausreichende Unterstützung hierfür an. Das Problem ist in erster Linie die Lebensdauer der Markierungen von generalisierbaren Deklarationselementen. Wurde der Build beispielswiese über Nacht angestoßen, müsste der Builder am nächsten Morgen vom Projekt entfernt werden, um ohne Verzögerungen durch die aktivierte Generalisierungsprüfung arbeiten zu können. Wird jedoch das Declared Type Generalization Checker Plug-In vom Projekt entfernt, werden auch alle Markierungen gelöscht; was dem regulären Verhalten von Buildern entspricht. Somit müssten die Ergebnisse unmittelbar nach der Prüfung verwertet werden, bevor der Builder entfernt wird und mit der regulären Entwicklungsarbeit begonnen werden kann. Dies mag unter Umständen vertretbar sein, da die abgesetzte Prüfung mit dem Ziel gestartet wird, die generalisierbaren Deklarationselemente auch zu bearbeiten. Mehr Komfort und auch Integration in den Entwicklungsablauf würde bringen, wenn der Builder einfach nur deaktiviert werden könnte, anstatt ihn komplett entfernen zu müssen. Die Property-Page des Plug-Ins müsste um eine entsprechende Auswahl erweitert werden. Die Auswahl würde steuern, dass der Builder nur nicht auf Build-Anweisungen reagiert und somit keine Verzögerungen durch die Prüfung verursacht. So würden die Markierungen bestehen bleiben und könnten über die Filter der Darstellung von Warnungen weiterhin bei Bedarf aus- und eingeblendet werden. Der Entwickler kann somit jederzeit auf die Prüfungsergebnisse zugreifen und die Generalisierungen durchführen; er ist nicht genötigt, dies unmittelbar nach der Prüfung zu tun.

6.5. Vorgaben für den Ausschluss von Deklarationselementen von der Prüfung

Die aktuelle Implementierung des Declared Type Generalization Checker Plug-Ins führt die Generalisierungsprüfung grundsätzlich für alle Deklarationselemente durch, die den Vorbedingungen der beiden angebundenen Refactorings entsprechen.[1] In einigen Fällen mag das aber gar nicht gewünscht sein bzw. der verwendete Typ ist

[1] Die Vorbedingungen werden in den Abschnitten 3.2.1 bzw. 3.2.2 definiert.

bewusst so gewollt, obwohl er nicht maximal generalisiert ist. In diesen Fällen dauert die Prüfung nur unnötig lange, da manche Ergebnisse keine Relevanz haben. Eine Übersicht, in welchen Fällen auf maximale Generalisierung verzichtet werden kann, gibt Steimann u. a. (2003).

Um dem Rechung zu tragen, sind zwei Ansätze für die Weiterentwicklung des Plug-Ins denkbar. Zum einen könnte die Konfiguration des Declared Type Generalization Checker Plug-Ins auf der Property Page so erweitert werden, dass Deklarationselemente bzw. Typen von vornherein von der Prüfung ausgeschlossen werden können. Der Entwickler hätte die prüfbaren Deklarationselemente[2] zur Auswahl und könnte bestimmen, dass beispielsweise Rückgabewerte von Methoden grundsätzlich nicht zu prüfen sind. Er hätte aber auch alle Typen des Projektes zur Auswahl, um definieren zu können, dass z.B. Deklarationen vom Typ String nicht zu prüfen sind.[3] Hilfreich sind auch Kombinationen aus beiden Parametern in der Art, dass nicht nur generell Deklarationselemente oder Typen ausgeblendet werden können, sondern beispielsweise nur Felddeklarationen mit dem Typ String. Zusätzlich sollte auch noch die Einstellung kombinierbar sein, dass *private* deklarierte Elemente ausgeschlossen werden können.

Der andere Ansatz geht davon aus, dass einzelne Deklarationen nicht weiter generalisiert werden sollen, die sich aber nicht mit einem, wie oben beschriebenen, generellen Filter ausblenden lassen. Solche Deklarationen auszublenden hätte nicht nur den Vorteil, dass sich die Laufzeit der Prüfung verkürzen würde, sondern auch, dass die zugehörige Markierung gelöscht werden könnte und nicht als Leiche – die ohnehin nicht abgearbeitet wird, da bereits ausreichend generalisiert – in den Listen oder im Quelltext bestehen bleibt. Um Deklarationen als ausreichend generalisiert zu markieren, könnte eine entsprechende, zu definierende Annotationen verwendet werden.[4] Stößt die Generalisierungsprüfung, bei der Abarbeitung der Deklarationselemente, im Quelltext auf eine solche Annotation, wird das so markierte Deklarationselement ungeprüft übersprungen. Um das Einfügen dieser Annotation zu erleichtern und zu automatisieren, könnte dies als alternativer Quick-Fix für Markierungen der Generalisierungsprüfung implementiert werden. Der Entwickler hat dann für eine Markierung die Wahl, entweder das Refactoring zu starten, um die Deklaration zu generalisieren, oder die Annotation zu setzen, um die Deklaration zu kennzeichnen und zukünftig

[2] siehe hierzu Abschnitt 4.4.5

[3] Hier wäre auch eine Gruppierung von Typen denkbar, so dass zusätzlich z.B. eine Gruppe Basistypen zur Auswahl stünde, damit nicht jeder Typ einzeln selektiert werden muss.

[4] Ein Vorschlag für die Annotation ist: *@Sic*

von der Prüfung auszuschließen.

A. Die Eclipse-Plattform

Die folgenden Abschnitte sollen in die Eclipse-Workbench einführen, deshalb wird zunächst ein Überblick über die Plattform selbst und ihre Entstehung gegeben. Vertiefend wird anschließend der Aufbau, gerade in Bezug auf Erweiterbarkeit, behandelt.

A.1. Entwicklungsgeschichte

Die Eclipse-Plattform wird als Open-Source-Projekt von einer breiten Entwicklergemeinde vorangetrieben. Die Basisplattform kann dabei von jedem über Plug-Ins erweitert werden. Die Plattform selbst bzw. die Hauptprojekte werden von registrierten Committern betreut. Somit wird eine stetige Weiterentwicklung auf hohem Niveau garantiert.
Verantwortlich für die Plattform in Bezug auf strategische und architektonische Ausrichtung sowie Versionsverwaltung und Freigabe ist die Eclipse-Foundation. Die Eclipse-Foundation ist der 2004 gegründete und rechtlich eigenständige Nachfolger des Eclipse-Konsortiums, das weitestgehend von IBM geführt wurde. Die Foundation wird von gewählten Direktoren geleitet, diese Direktoren sind Repräsentanten von strategischen Anwendern und Entwicklern. Jedoch werden noch heute die meisten der Basisentwickler von IBM bezahlt. Die führende Stellung von IBM liegt in den Anfängen von Eclipse begründet, da Eclipse der Nachfolger der Entwicklungsumgebung Visual Age for Java 4.0 von IBM ist. Am 7. November 2001 veröffentlichte IBM die Quelltexte seiner Entwicklungsumgebung; dies war zugleich die Geburtsstunde von Eclipse 1.0. Bereits im Juni 2002 folgte Eclipse in der Version 2.0, die bis zur Version 2.1 rein als erweiterbare Entwicklungsumgebung konzipiert war. Seit dem Erscheinen der Version 3.0 im Juni 2004 ist Eclipse eine vollwertige Rich-Client-Plattform. Dies bedeutet, dass Eclipse nur noch einen Anwendungskern bereitstellt, der sich um den Lebenszyklus der Plug-In-Komponenten kümmert und die

vollständige Anwender-Funktionalität aus den Komponenten stammt. Somit lassen sich aufbauend auf der Eclipse-Plattform nicht nur Entwicklungsumgebungen, sondern vielfältige Anwendungen entwicklen.

A.2. Interne Struktur und Erweiterbarkeit

Bis zur Version 2.1 hatte der Eclipse-Kern einen proprietären Aufbau und enthielt neben den Funktionen für die Verwaltung von Plug-Ins auch bereits Strukturen, die für eine IDE gebraucht wurden. Auf der Rich-Client-Plattform, ab Version 3.0, können beispielsweise sowohl Content-Management-Systeme als auch Programme zur Verwaltung von Aktienkursen entwickelt werden.[1]
Die nötige Änderung der Architektur hätte große Umstrukturierungen am proprietären Eclipse-Kern erfordert, da viele geforderten Features nicht ohne weiteres realisierbar waren. Deshalb hielt man Ausschau nach einer standardisierten Lösung und entschied sich letztendlich für eine OSGi-konforme Implementierung. Der OSGi-Standard[2] für Komponentenmodelle stellte viele der benötigten Dienste bereits fertig zur Verfügung.
Der Eclipse-Kern fungiert nun als OSGi-Server und implementiert den Standard vollständig bzw. erweitert ihn sogar. Die Erweiterung sieht dabei vor, dass dynamische Komponenten selbst wieder durch Komponenten erweitert werden können. Diese Erweiterbarkeit war bereits in der Version 2.0 Grundlage des Plug-In-Konzepts von Eclipse. Wenn man nach einer Alternative zu bestehenden OSGi-Implementierungen sucht, kann dieser Dienst auch standalone als reine OSGi-Umgebung genutzt werden.[3]
In der OSGi-Welt spricht man allerdings nicht mehr von Plug-Ins, sondern von Bundles. Ein Bundle besteht dabei aus deklarativen Meta-Daten, die das Bundle beschreiben, sowie dem eigentlichen Code bzw. den weiteren Ressourcen wie z.B. Grafiken oder Texten. Ein Bundle unterscheidet sich somit rein architektonisch nicht von einem Plug-In. In der Eclipse-Welt spricht man aber landläufig weiterhin von Plug-Ins, obwohl es sich ab der Version 3.0 um OSGi-Bundles handelt.
Bei der Umstellung des Eclipse-Kerns wurde das Ziel verfolgt, abwärtskompatibel zu Plug-Ins der Version 2 zu bleiben, damit diese ohne Änderung weiter verwendet

[1] Eine Übersicht ist zu finden auf: http://www.eclipse.org/community/rcpos.php
[2] siehe: OSGi-Alliance (2006)
[3] Bekannte weitere OSGi-Implementierungen sind Knopflerfish und Oscar.

werden können. Da aber der Eclipse-Kern nur OSGi-konforme Bundles verarbeiten kann, musste eine entsprechende Strategie überlegt werden. Um bei der neuen Implementierung nicht durch zu viele Kompromisse behindert zu werden, die eine direkte Lauffähigkeit von alten Plug-Ins erfordert hätte, ließ man die Abwärtskompatibilität für die neue Runtime völlig außer Acht. Erreicht wird die Kompatibilität jedoch durch einen Compatibility-Layer, der die Ablaufsteuerung der alten Plug-Ins übernimmt und die Plug-Ins an den neuen Kern nur durchreicht. Der Compatibility-Layer nimmt hierzu nötige Änderungen an den Plug-Ins automatisch vor. Eclipse erkennt alte Plug-Ins zunächst am Fehlen der Manifest-Datei, deshalb wird das Plug-In nicht gegen die neue Runtime im Package *org.eclipse.core.runtime* gelinkt, sondern gegen *org.eclipse.core.runtime.compatibility*, die den Compatibility-Layer enthält. Dieser transformiert anschließend die Meta-Daten aus der *plugin.xml*-Datei in ein OSGi-konformes Manifest und bringt das Plug-In als Bundle zur Ausführung.
Die OSGi-Implementierung kümmert sich nicht nur um Kompatibilität zwischen Version 2 und Version 3 Plug-Ins. Die Runtime ist auch in der Lage, verschiedene Versionen des gleichen Plug-Ins parallel ablaufen zu lassen. Möglich ist dies, da jedes Bundle einen eigenen Classloader hat, der nur dieses Bundle kennt und dafür zuständig ist, alle Teile dieses Bundles korrekt zu laden. Deshalb hat jedes Bundle auch seinen eigenen Classpath und der Classloader hat keinen Zugriff auf einen systemweiten Classpath. Abhängigkeiten zwischen Bundles werden durch einen Delegationsmechnismus gelöst. Wenn das aktuelle Bundle Ressourcen eines anderen benötigt, wird der Classloader des benötigten Bundles angestoßen. Dies erfolgt auch rekursiv, so dass zur Laufzeit ein Bundle nur geladen wird, wenn alle übergeordneten Ressourcen geladen sind. Dieser Mechanismus erlaubt es auch, dass viele Bundles ohne Neustart der Anwendung zur Laufzeit gestartet werden können.
In der folgenden Abbildung A.1 findet sich eine Übersicht über die Komponenten der Eclipse-Plattform. Abgegrenzt sind die Komponenten, die als Teil der Rich-Client-Plattform von jeder Art Anwendung benötigt werden bzw. generisch genug sind, um Basisdienste anzubieten. Teil der Rich-Client-Plattform sind die Bundles, die den Eclipse- bzw. OSGi-Kern beinhalten. Außerdem gehören hierzu die Bundles, die für Benutzeroberflächen und Dialoge benötigt werden, insbesondere die Dienste SWT und JFace.[4] Diese Bundles werden als Generic Workbench bezeichnet. Mit der Umstellung auf Eclipse 3 wurden dort alle Komponenten gebündelt, die für Oberflächen benötigt werden, jedoch nicht spezifisch für eine IDE sind.

[4]Für eine detallierte Beschreibung sei auf Daum (2005) verwiesen.

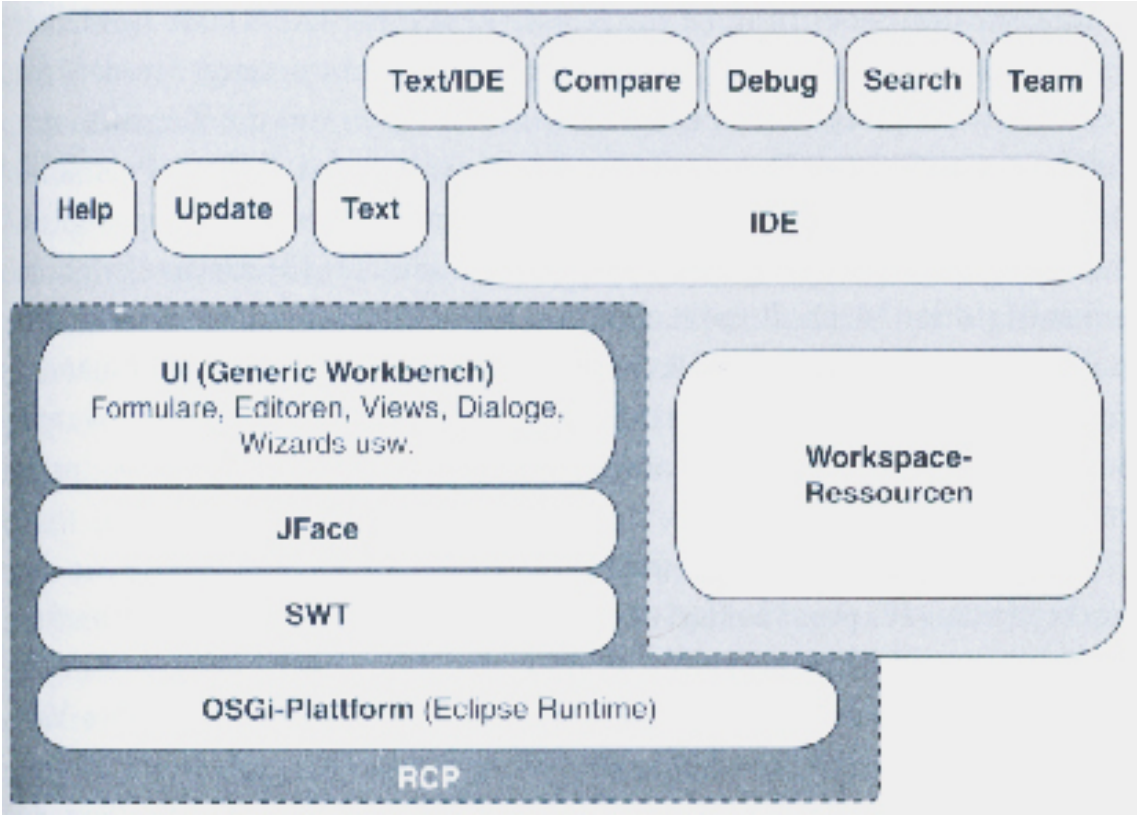

Abbildung A.1.: Architektur der Eclipse-Plattform
aus: Daum (2005)

Die restlichen Komponenten der Eclipse-Plattform, beispielsweise das mächtige Hilfesystem, können zwar von jeder Rich-Client-Anwendung verwendet werden, aber sie sind nicht Teil des minimalen Kerns, der stets komplett mit ausgeliefert wird.

B. Grundlagen der Plug-In-Entwicklung

‘Eclipse ist ein ehrgeiziges Unternehmen. Es stellt eine Plattform bereit, die unterschiedlichen Tools eine Zusammenarbeit ermöglicht, oftmals in einer Weise, die sich die Tool-Autoren anfangs gar nicht ausgemalt haben. Um dieses Ziel Wirklichkeit werden zu lassen, verwenden wir einen offenen, leistungsorientierten, kooperativen Entwicklungsprozess – qualitativ hochwertige Beiträge werden von jedermann angenommen. Denken wir an die Erweiterungsregel ‘Alles ist eine Erweiterung’. Und mit vielen Erweiterungen sind die Möglichkeiten schier endlos.’[1]
In diesem Sinne wird im folgenden Kapitel die Philosophie und der Aufbau von Plug-Ins für Eclipse beschrieben. Darüber hinaus wird auch auf das Deployment und den Lebenszyklus eingegangen.

B.1. Prinzipien

Bei der Programmierung von Plug-Ins für Eclipse sollte man sich stets vor Augen führen, dass man niemals für sich alleine auf der grünen Wiese hantiert. Damit ist gemeint, dass sich schon rein durch die Struktur der Plattform immer Abhängigkeiten und Wechselwirkungen zwischen Plug-Ins ergeben. Außerdem arbeitet kein geschlossener Entwicklerkreis am Eclipse-Projekt, man kann nie sicher sein, in welcher Weise sich andere Komponenten weiterentwickeln oder die eigene benutzt oder wiederum erweitert wird. Deshalb ist es wichtig, einige Regeln einzuhalten. Glücklicherweise handelt es sich nur um eine handvoll Regeln, jedoch sollten diese umso penibler eingehalten werden, um eine reibungslose Integration in die Plattform zu ermöglichen und unerwünschte Seiteneffekte zu verhindern, die Anwender oder Entwickler irritie-

[1] aus: Gamma und Beck (2004)

ren.
In Gamma und Beck (2004) werden Regeln, die zu beachten sind, detailliert erklärt. Sie sind danach gruppiert, für welchen Entwicklerkreis die Regeln relevant sind. Gamma und Beck (2004) unterscheiden

- den Erweiterer, also den Entwickler, der eigene Plug-Ins schreibt und somit fremde erweitert,
- den Enabler, der eigene Funktionalität veröffentlicht und für andere erweiterbar macht
- und den Veröffentlicher, der Plug-Ins verbreitet oder vertreibt.

Im Folgenden werden die wichtigsten Regeln definiert, die für diese Arbeit Relevanz haben. Für einen vollständigen Überblick sei auf Gamma und Beck (2004) verwiesen.

Erweiterungsregel: Das ganze System besteht aus Erweiterungen.[2]

Regel der gemeinsamen Nutzung: Plug-Ins werden immer zu einer bestehenden Umgebung hinzugefügt, sie ersetzen auch keine bestehenden Plug-Ins.

Konformitätsregel: Implementiert ein Plug-In eine Schnittstelle, muss es sich an das Protokoll der Schnittstelle halten und darf auch keine andere als die erwartete Funktionalität bereitstellen.

Schichtenregel: Funktionalität darf in den Klassen nicht bunt gemischt werden, sondern muss den Schichten der Plattform entsprechen. So ist beispielsweise Kernfunktionalität von der Benutzeroberfläche zu trennen.

Lazy-Loading-Regel: Plug-Ins werden beim Start der Anwendung nicht automatisch geladen, sondern beim ersten Zugriff. Deshalb sollten Zugriffe auf andere Plug-Ins erst gemacht werden, wenn sie gebraucht werden. Auf eine frühe Initialisierungsphase, die alle benötigten Plug-Ins aufruft, sollte verzichtet werden.

Sichere Plattform-Regel: Stellt ein Plug-In Schnittstellen bereit, muss das anbietende Plug-In auch die fehlerhafte Verwendung abfangen.

[2]Es handelt sich weniger um eine Regel, sondern mehr um einen Hinweis auf die Architektur der Plattform.

Regel der expliziten Erweiterung: Ein Plug-In definiert explizit, an welchen Stellen Funktionalität von außen zugegriffen werden kann, alle anderen Stellen sind tabu.

Stabilitätsregel: Das Protokoll einmal veröffentlichter Schnittstellen darf nicht mehr verändert werden.

Benutzerkontinuität: Einstellungen, die Benutzer vornehmen, sollten über Sitzungen hinweg persistent gespeichert werden.

Neben den Regeln, die die Entwickler der Eclipse aufgestellt haben, existieren noch weitere, die aus der Praxis der Plug-In-Entwicklung stammen. Buck (2006) kennt folgende Prinzipien:

- Auf Benutzeraktionen muss immer eine Reaktion erfolgen. Ist die Aktion nicht ausführbar, muss eine aussagekräftige Fehlermeldung angezeigt werden.
- Aufgaben, die nebenläufig im Hintergrund erledigt werden, müssen dem Benutzer den Fortschritt anzeigen und müssen gegebenenfalls durch den Benutzer abgebrochen werden können.
- Für die Fehlersuche sollten Log-Mechanismen implementiert werden, wobei auf die plattforminternen Ressourcen zurückgegriffen werden sollte, d.h., der Eclipse-Log-Mechanismus und die Eclipse-Log-Datei ist zu verwenden.
- Persistente Konfigurationen von Plug-Ins, d.h Einstellungen auf Property- oder Preference-Pages, sollen auch über die Eclipse-Mechanismen gespeichert werden.
- Texte in der Benutzeroberfläche werden in einer zentralen Sprachdatei gesammelt, um eine spätere Übersetzung zu ermöglichen.

Auch für die Gestaltung der Benutzeroberflächen existieren Vorgaben, die einzuhalten sind.[3] Nur so kann eine Anwendung, die von zahllosen Plug-In-Entwicklern gestaltet wird, mit einem einheitlichen Erscheinungsbild entstehen. Ohne die Vorgaben würde die Oberfläche bald chaotische Zustände annehmen, da kein durchgängiges Bedienkonzept mehr gegeben wäre. In den Oberflächen-Richtlinien finden sich Anweisungen

[3] siehe: Edgar u. a. (2004)

über die Verwendung von Farben oder zur Gestaltung von Schaltflächen. Außerdem wird geregelt, wie Dialoge und Menüs angeordnet und aufgebaut sein müssen.

B.2. Aufbau und Struktur von Plug-Ins

Seit die Version 3.0 von Eclipse veröffentlicht wurde, handelt es sich bei Plug-Ins genaugenommen um OSGi-konforme Bundles. Die folgenden Abschnitte sollen einen Überblick verschaffen, wie Bundles aufgebaut sind, über welche Schnittstellen sie in die Eclipse-Architektur eingebettet werden und wie Bundles miteinander interagieren. Die generelle Architektur von Eclipse wurde in Abschnitt A.2 dargestellt, darüber hinaus sei hierzu auf Daum (2005) und Lippert (2006) verwiesen.

B.2.1. Das Plug-In-Manifest

Grob gesagt enthält das Manifest die Konfiguration eines Plug-Ins. Dabei ist nicht die Konfiguration gemeint, die ein Plug-In für seine Aufgabe braucht und die gegebenenfalls vom Benutzer zur Laufzeit angepasst werden kann. Gemeint ist die Konfiguration, die der Eclipse-Kern braucht, um das Plug-In laden, starten und in die Infrastruktur einbinden zu können.
In der Version 2 von Eclipse befand sich die Konfiguration in einer einzigen Datei, die mit dem Namen *plugin.xml* im Wurzelverzeichnis des Plug-Ins liegen musste. Seit der Version 3.0 ist die Konfiguration auf zwei Dateien aufgeteilt. Hierbei handelt es sich um einen Tribut an die OSGi-Konformität des Eclipse-Kerns. Gemäß der OSGi-Spezifikation müssen die Informationen, die der Kern zum Laden eines Bundles braucht, in der Datei *MANIFEST.MF* im Verzeichnis *META-INF* liegen. Die Parameter der Datei *MANIFEST.MF* wurden aus der bisherigen *plugin.xml* herausgelöst, diese enthält nun nur noch die Konfigurationen, die typisch für ein Eclipse-Plug-In sind.
Die Datei *MANIFEST.MF* liegt nicht wie die *plugin.xml*-Datei im XML-Format vor. Es handelt sich um eine einfach strukturierte Textdatei; in ihr befinden sich zeilenweise die Konfigurationsparameter, mit folgendem Aufbau:

```
<Parametername>: <Parameterwert>[,<Parameterwert>]
```

In OSGi-Alliance (2006) definiert der Standard eine Vielzahl von möglichen Parametern, jedoch finden nicht alle in Eclipse-Manifesten Anwendung. Der OSGi-Standard erlaubt ausdrücklich, dass eigene implementierungsabhängige Parameter definiert werden können. Hiervon wird bei Eclipse-Manifesten Gebrauch gemacht. In der nachfolgenden Aufstellung finden sich die Parameter, die in Eclipse-Bundles verwendet werden. Spezifische Parameter für Eclipse wurden als solche kenntlich gemacht, indem der Parametername mit *ECLIPSE* beginnt.

Bundle-ManifestVersion: Angabe, auf welcher Version der OSGi-Spezifikation das Manifest beruht

Bundle-Name: Name des Bundles; sollte ein lesbares Format haben

Bundle-SymbolicName: eindeutiger Name des Bundles, meist im Format einer umgekehrten Domain, z.B. *org.intoJ.plugIn*

Bundle-Version: Versionsnummer des Bundles

Bundle-ClassPath: Angabe des Classpathes zum Laden des Bundles

Bundle-Activator: Pfad zur Klasse, die als erstes geladen werden muss, um das Bundle zu starten

Bundle-Vendor: Angabe zum Veröffentlicher eines Bundles, z.B. Firmenname

Bundle-Localization: Pfad zur Datei, die allgemeine Texte enthält, die in verschiedenen Sprachversionen in der Benutzeroberfläche angezeigt werden

Require-Bundle: Liste der Bundles, die vom aktuellen Bundle benötigt werden, damit es ausführbar ist

Export-Package: Liste der Schnittstellen, die das Bundle anderen öffentlich zur Verfügung stellt

Import-Package: Liste mit Komponenten, die vom Bundle explizit importiert werden, aber keine offiziellen Schnittstellen sind

Bundle-RequiredExecutionEnvironment: Liste mit Laufzeitumgebung, in denen das Bundle lauffähig ist, z.B. *J2SE-1.5*

Eclipse-LazyStart: Angabe, ob das Bundle beim Start von Eclipse geladen werden muss oder ob es beim ersten Zugriff geladen werden kann

Eclipse-PlatformFilter: Angabe, in welchen Umgebungen das Bundle lauffähig ist; z.B. erlaubt der Parameterwert *osgi.os=win32* den Start nur auf einem Windows-Betriebssystem

Eclipse-RegisterBuddy: Liste mit Bundles, denen das Bundle erlaubt, auf alle Ressourcen zuzugreifen

Eclipse-BuddyPolicy: Angabe, ob das Bundle auf befreundete Bundles generell zugreifen soll

Eclipse-ExtensibleAPI: Angabe, ob das Bundle durch abhängige Bundles erweitert werden darf

Im Gegensatz hierzu wird die Datei *plugin.xml* im XML-Format gespeichert. Wurzelelement ist der Tag *plugin*; dieser enthält jedoch nur zwei Kindelemente, nämlich *extension* und *extension-point*. Daraus wird ersichtlich, wozu diese Datei dient: Sie enthält die detaillierte Konfiguration, über welche Schnittstellen das Plug-In mit den anderen Plug-Ins im System kommuniziert. Diese Einstellungen sind reine Interna der Eclipse, die so auf keinem anderen OSGi-System Anwendung finden würden. Somit ist auch die Aufteilung auf zwei Manifest-Dateien gerechtfertigt.
Das Element *extension* dient zur Konfiguration der Schnittstellen anderer Plug-Ins, auf die das Plug-In zugreift. Im Gegensatz hierzu regelt *extension-point* die Veröffentlichung eigener Schnittstellen. Innerhalb der Elemente befinden sich abhängig von der verwendeten Schnittstelle weitere Elemente zur Konfiguration. Beiden Elementen ist jedoch gemein, dass sie die Attribute *id* und *name* enthalten müssen. *id* enthält dabei eine eindeutige Kennung der Schnittstelle, über die sie im System auffindbar ist. Hingegen enthält *name* einen lesbaren Bezeichner der Schnittstelle. Zusätzlich muss das Element *extension* noch das Attribut *point* enthalten. Mit diesem wird in umgekehrter Domain-Notation angegeben, auf welche Schnittstelle in welchem Plug-In zugegriffen wird.
Ein Beispiel für den Aufbau der *plugin.xml* ist im Abschnitt B.2.2 zu finden, wo auf Extensions und Extension-Points und ihre Bedeutung innerhalb der Eclipse genauer eingegangen wird.

B.2.2. Extensions und Extension-Points

Als Extension bezeichnet man in Eclipse jeden Zugriff eines Plug-Ins auf andere Plug-Ins, deren Funktionalität somit erweitert wird. Unter erweitern ist in der Eclipse-Plattform jedoch mehr zu verstehen, als man zunächst denken würde. Im gewohnten objektorientierten Sprachgebrauch versteht man darunter nur das Ableiten einer bestehenden Klasse. In Eclipse kann es die Implementierung eines Interfaces, das Benutzen von Methoden, aber auch die Ableitung von Klassen bedeuten. Da die Eclipse-Plattform, abgesehen vom Kern, nur aus Plug-Ins besteht, müssen sogar vermeintliche Basisdienste wie das Einhängen eines neuen Menüpunktes über den Extension-Mechanismus laufen.
Damit innerhalb der Plattform kein Wildwuchs an Extensions entsteht, der die Stabilität gefährden würde, sollte nur auf Funktionalität zugegriffen werden, die explizit für die Verwendung von außen über Extension-Points freigegeben wurde. Dieses Vorgehen ist konform zur Regel der expliziten Erweiterung aus Abschnitt B.1. Es ist allerdings möglich, aber nicht ratsam, diese Regel zu unterwandern, da Plug-Ins letztlich nur Klassenkonstrukte in Java sind. Allerdings läuft man dann Gefahr, wenn Klassen in einer anderen Version des erweiterten Plug-Ins verändert werden, das eigene Plug-In nicht mehr lauffähig ist. Durch die Stabilitätsregel werden nur Änderungen an bereits veröffentlichten Schnittstellen untersagt. Interna von Plug-Ins sind von der Regel nicht betroffen und können dies auch nicht sein, denn sonst wäre eine Weiterentwicklung eines Plug-Ins kaum möglich.
Im Folgenden soll das Zusammenspiel von Extensions und Extension-Points am Beispiel eines Menüeintrages demonstriert werden. Um den Eintrag anzulegen, muss der Extension-Point *org.eclipse.ui.actionSets* erweitert werden:

```
<extension id="org.intoJ.plugIn.extension1"
           name="Erweiterung für einen Menüpunkt"
           point="org.eclipse.ui.actionSets">
    <actionSet id="org.intoJ.plugIn.actionSet1"
               label="ActionSet 1 für Menüpunkt">
        <action class="org.intoJ.plugIn.actions.MenuAction"
                icon="icons/plugIn.gif"
                id="org.intoJ.plugIn.actionSet1.action1"
                label="Menütext für diese Action"
                menubarPath="about/org.intoJ.plugIn"
```

```
                tooltip="Tooltip für den Menüeintrag"/>
    </actionSet>
</extension>
```

Die Erweiterung des Extension-Point *org.eclipse.ui.actionSets* erfordert zusätzliche Parameter, die über das Manifest definiert werden. Deshalb enthält das Element *extension* ein Element *actionSet*, das auch eindeutig über *id* und *label* gekennzeichnet werden muss. Dies ist erforderlich, da *extension* mehrere Elemente *actionSet* enthalten kann, die unabhängig voneinander sein können. Ein ActionSet kann wiederum mehrere Aktionen enthalten, die mit dem Element *action* definiert werden. Das Attribut *class* gibt an, welche Klasse die Funktionalität bereitstellt, denn der Menüeintrag wird mit einem Delegate-Mechanismus realisiert. Durch die Erweiterung wird die Klasse bekanntgegeben und später, wenn das Menü aufgebaut wird, aufgerufen. Um das Protokoll einzuhalten, muss die Klasse zwingend das Interface *org.eclipse.ui.IWorkbenchWindowActionDelegate* implementieren. Das Element *action* wird durch das Attribut *id* eindeutig identifiziert. Die Attribute *icon*, *label* und *tooltip* steuern die Ansicht im Menü, indem sie angeben, welches Bild, welcher Text und welcher Kurzhilfetext erscheinen. Mit dem Attribut *menubarPath* legt man die Stelle fest, wo der neue Menüeintrag eingehängt werden soll.
Die Definition eines Extension-Points zerfällt hingegen in zwei Teile. Der erste Teil ist der Eintrag im Manifest des Plug-Ins, das die Funktionalität bereitstellt. Der Extension-Point wird mit einem eindeutigen Bezeichner (das Attribut *id*) und einem lesbaren Namen (das Attribut *name*) versehen. Zusätzlich trägt der Extension-Point einen Verweis auf eine ausgelagerte Schema-Datei:

```
<extension-point id="actionSets"
                 name="%ExtPoint.actionSets"
                 schema="schema/actionSets.exsd"/>
```

Der zweite Teil der Extension-Point-Definition, das Schema, enthält Informationen darüber, wie die Extension zum Extension-Point genau auszusehen hat. Die Elemente *element* legen fest, welche Tags die Definition der Extension enthalten muss und welche Attribute erwartet werden:

```
<schema targetNamespace="org.eclipse.ui">
    <annotation>
```

```
        <appInfo>
            <meta.schema plugin="org.eclipse.ui"
                         id="actionSets"
                         name="%ExtPoint.actionSets"/>
        </appInfo>
    </annotation>
    <element name="extension">
        <complexType>
            <attribute name="point" type="string"/>
            <attribute name="id"    type="string"/>
            <attribute name="name"  type="string"/>
        </complexType>
    </element>
    <element name="action">
        <complexType>
            <attribute name="class" type="string">
                <annotation>
                    <appInfo>
                        <meta.attribute kind="java"
                                        basedOn="org.eclipse.ui. \
                                        IWorkbenchWindowActionDelegate"/>
                    </appInfo>
                </annotation>
            </attribute>
        </complexType>
    </element>
</schema>
```

Das Schema besagt, dass für die Erweiterung von *org.eclipse.ui.actionSets* die Elemente *extension* und *action* erwartet werden, wobei *extension* die Attribute *id*, *name* und *point* enthält. Während andere optional sind, benennt die Definition des Elements *action* das Attribut *class* als zwingend erforderlich und legt für den Wert des Elements fest, dass es den Namen einer Java-Klasse enthalten muss, die das Interface *IWorkbenchWindowActionDelegate* implementiert.

B.2.3. Abhängigkeiten zwischen Plug-Ins

Da innerhalb der Eclipse-Plattform kein allgemeiner Classpath existiert, sondern jedes Plug-In über einen eigenen verfügt, müssen Plug-Ins, auf die zugegriffen werden soll, explizit bekannt gegeben werden. Diese Abhängigkeitsbeziehungen werden im Manifest beschrieben. Die Anweisung *Require-Bundle* in der Datei *MANIFEST.MF* wird hierzu gebraucht.
Unabhängig davon, welche Funktion ein Plug-In hat, bindet es immer die Bundles *org.eclipse.core.runtime* und *org.eclipse.ui* ein. Sie enthalten Basisdienste der Plattform, die schon zum Start eines Plug-Ins gebraucht werden. Grundsätzlich lassen sich alle Bundles der Plattform, auch solche von Fremdanbietern, einbinden. Allerdings wird der Quelltext im Bundle nicht immer mit ausgeliefert. Bei solchen Bundles ist man auf eine aussagekräftige Beschreibung der Extensions-Points angewiesen.
Bundles werden dynamisch gebunden und sind deshalb darauf angewiesen, dass alle verwendeten Ressouren in der Plattform vorhanden sind. Allerdings kann man im Manifest definieren, dass Abhängigkeiten optional sind. Der entsprechende Eintrag im Manifest ergibt sich wie folgt:

```
Require-Bundle: org.eclipse.ui.ide;resolution:=optional
```

Das Plug-In würde nun auch ohne das Bundle *org.eclipse.ui.ide* gestartet werden. Allerdings würde ein Zugriff auf die Ressourcen bei Fehlen dieses Bundles einen Laufzeitfehler verursachen. Dieser muss entweder abgefangen werden, oder es kann vor dem Zugriff geprüft werden, ob das Bundle tatsächlich vorhanden ist.
Informationen über die Eclipse-Plattform und die geladenen Plug-Ins lassen sich von der zentralen Klasse Plattform aus dem Bundle *org.eclipse.core.runtime* beziehen. Über statische Methoden können Daten der Laufzeitumgebung[4] abgefragt werden, u.a. auch die Erweiterungen. Das folgende Beispiel soll die Abfrage von Extension-Points und Extensions verdeutlichen:

```
// Der Extension-Registry sind alle Extension-Points bekannt, die
// im System deklariert worden sind
IExtensionRegistry registry = Platform.getExtensionRegistry();
// Die Methode getExtensionPoints() liefert ein Array aller
// Extension-Points zurück
```

[4] z.B. Betriebssystem, Prozessor-Architektur, Installationspfad usw.

```
IExtensionPoint[] points = registry.getExtensionPoints();
// Die Extension-Points lassen sich aber auch gezielt über ihre
// eindeutige Id abfragen
IExtensionPoint point =
    registry.getExtensionPoint("org.intoJ.extensionPoint");
// Extension-Points geben auch Auskunft darüber von welchen Plug-Ins
// sie über Extensions erweitert werden
IExtension[] extensions = point.getExtensions();
```

Bedingt durch die dynamische Komposition der Plug-In-Abhängigkeiten ist neben der Vergabe von eindeutigen Bezeichnern für Extensions und Extension-Points auch die zusätzliche Vergabe von lesbaren Namen wichtig. Dies erleichtert dem Entwickler das Auffinden der passenden Elemente.
Aus der Extension-Registry können noch keine Informationen bezogen werden, ob ein Plug-In generell im System vorhanden ist. Hierfür ist die Klasse *Bundle* zuständig. Eine Instanz dieser Klasse liefert wiederum die Klasse Plattform. Das folgende Beispiel zeigt den Zugriff auf Laufzeitinformationen von Bundles:

```
// Über die eindeutige Id wird ein Bundle referenziert
Bundle bundle = Platform.getBundle("org.intoJ.plugIn");
// Wird null zurückgeliefert ist das Bundle nicht vorhanden
// ansonsten kann der Zustand abgefragt werden in dem sich das Bundle
// befindet
if (bundle != null) {
    switch (bundle.getState()) {
        // Das Bundle wurde deinstalliert und ist nicht lauffähig
        case Bundle.UNINSTALLED: break;
        // Das Bundle wurde installiert, aber seine Referenzen
        //wurden noch nicht aufgelöst, es ist nicht lauffähig
        case Bundle.INSTALLED: break;
        // Die Referenzen wurden aufgelöst und das Bundle kann
        // gestartet werden
        case Bundle.RESOLVED: bundle.start();
                              break;
        // Das Bundle startet gerade
        case Bundle.STARTING: break;
        // Das Bundle wird gerade gestoppt
```

```
        case Bundle.STOPPING: break;
        // Das Bundle läuft gerade und könnte gestoppt werden
        case Bundle.ACTIVE: bundle.stop();
                            break;
    }
}
```

B.3. Plug-In-Development-Environment (PDE)

Eclipse bringt gewissermaßen das Werkzeug, um erweitert zu werden, gleich mit. Das Plug-In-Development-Environment ist allerdings keine separate Anwendung, sondern eine Sammlung von Werkzeugen innerhalb der Eclipse-Arbeitsumgebung. Die PDE muss nicht einmal explizit gestartet werden. Der erste Schritt, um ein Plug-In zu erstellen, ist, ein neues Projekt über das Menü *File - New - Project...* zu erzeugen. Im Dialog aus Abbildung B.1 kann als Projekt-Typ ein Eclipse-Plug-In ausgewählt werden.

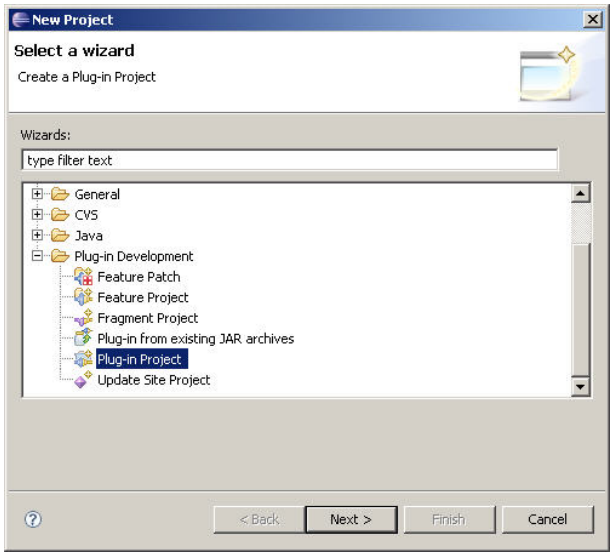

Abbildung B.1.: Anlegen eines neuen Plug-In-Projekts

In den weiteren Dialogen werden Einstellungen für das Plug-In abgefragt, aufgrund derer in der Arbeitsumgebung das Projekt angelegt wird. Alle Dateien und Verzeichnisse, die für Plug-Ins grundsätzlich gebraucht werden, erstellt die PDE ebenfalls in diesem Zuge. Die Entwicklung eines Plug-Ins unterscheidet sich im wesentlichen

nicht von der Umsetzung anderer Java-Projekte und erfolgt deshalb in der gewohnten Arbeitsumgebung. Die PDE unterstützt den Entwickler bei Konfiguration und Deployment des Plug-Ins. Sie stellt einen grafischen Editor für das Manifest und die *plugin.xml*-Datei bereit, über den alle Einstellungen in Auswahllisten und Dialogen gemacht werden können. Der Editor wird gestartet, wenn eine dieser beiden Dateien geöffnet wird. Durch den Editor wird man aber nicht vollkommen bevormundet, sondern er erlaubt das direkte Bearbeiten der Dateien und übernimmt diese Änderungen in die Dialoge. Abbildung B.2 zeigt den Dialog für die Angaben zu Plug-In-Name, Version etc.

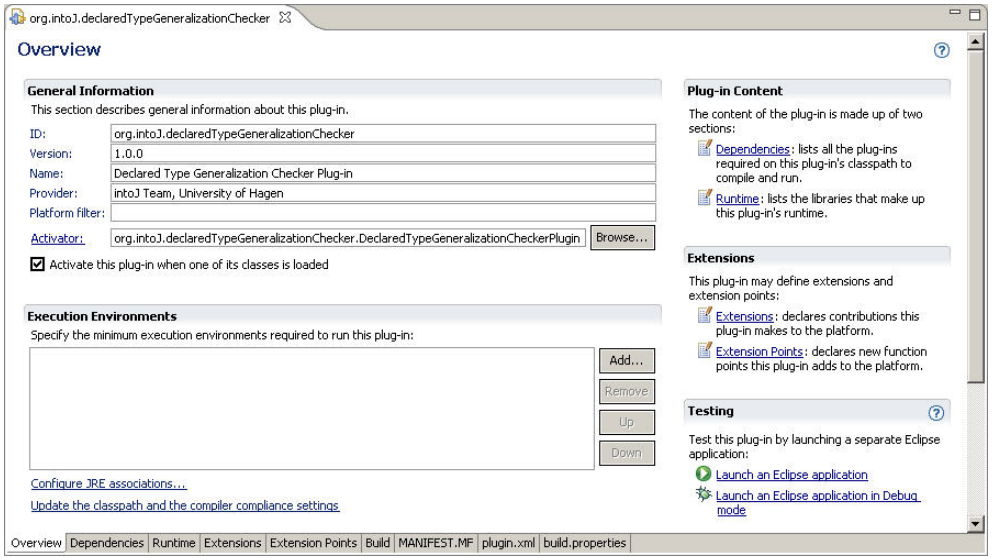

Abbildung B.2.: PDE-Editor für Manifest- und plugin.xml-Datei

Auch bei der Plug-In-Entwicklung ist ständiges Ausprobieren und Testen des gerade Entwickelten notwendig. Die PDE hilft dem Entwickler, indem sie ermöglicht, eine unabhängige Eclipse-Instanz mit dem aktuellen Entwicklungsstand zu starten. In dieser kann das Plug-In getestet werden, ohne dass es zunächst aufwändig deployed werden muss. Dabei steht für die Test-Instanz der Debugger von Eclipse zur Verfügung, mit dem alle Aktionen des Plug-Ins, aber auch der kompletten Eclipse-Instanz inspiziert werden können. Debug- oder Trace-Ausgaben der Test-Instanz werden in der Konsole der Entwicklungsumgebung angezeigt. Der Start der Test-Instanz kann beim Aufruf konfiguriert werden. Beispielsweise ist einstellbar, mit welchen Plug-Ins die Test-Instanz gestartet wird, welche JVM-Einstellungen verwendet werden oder für welche Plug-Ins Tracing aktiviert werden soll; siehe hierfür Abbildung B.3.

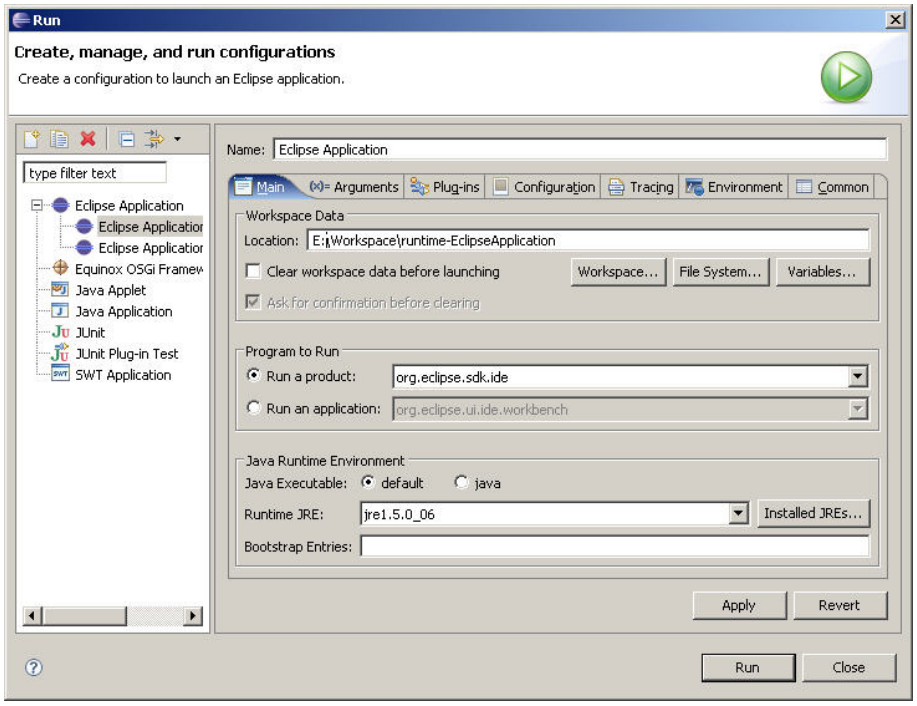

Abbildung B.3.: Konfiguration der Plug-In-Testumgebung

B.4. Deployment von Plug-Ins

Sobald man ein Plug-In fertig entwickelt und getestet hat, möchte man es natürlich auch vertreiben oder der Öffentlichkeit zur Nutzung zugänglich machen. Der Build-Prozess der PDE unterstützt den Entwickler dabei soweit, dass am Ende eine fertige und ausführbare JAR-Datei entsteht.

An diesem Punkt haben die Väter der Eclipse aber weitergedacht und die Frage gestellt, ob eine Software wirklich immer komplett fertig ist, wenn man mit der Auslieferung beginnt. Die Antwort sei dem Leser selbst überlassen – jedenfalls steht bei der Entwicklung von Plug-Ins das Konzept der Fragmente zur Verfügung. Dabei handelt es sich um leichtgewichtige Plug-Ins, die ein bestehendes Plug-In ergänzen können. Fragmente werden dafür verwendet, Sprachpakete, die zum Zeitpunkt der Auslieferung noch nicht verfügbar waren, nachzuliefern, um plattformabhängigen Code bereitzustellen oder um Updates im Sinne von Bugfixes zu installieren. Durch Fragmente wird ein Plug-In vollkommen transparent auf Binärebene ergänzt, ohne dass das Plug-In vom Entwickler neu gebaut und von den Anwendern neu installiert werden muss. Sobald Eclipse beim Startvorgang auf ein Fragment stößt, wird es

zusätzlich zum zugehörigen Plug-In geladen. Dabei können zu einem Plug-In beliebig viele Fragmente gehören. Die PDE beinhaltet speziell für Fragmente einen eigenen Projekt-Typ.
Eine große Stärke der Eclipse-Plattform ist ihre Modularität und die Aufteilung sämtlicher Funktionalität in kleine Einheiten. Die Kehrseite der Medaille ist aber, dass dadurch eine Vielzahl von Plug-Ins entsteht. Aus diesem Grund, stellen Gamma und Beck (2004) fest, ist es wichtig, 'dass die Menge der Plug-Ins überschaubar bleibt. Andernfalls endet eine Eclipse-Installation als Plug-In-Suppe.' Um dieser Entwicklung entgegenzuwirken, wartet Eclipse mit der Definition von Features auf. Mehrere Plug-Ins können zu einem Feature zusammengefasst werden und der Benutzer installiert dann nur noch das Feature. Welche Plug-Ins das Feature mit installiert, ist für ihn nicht mehr relevant. Tatsächlich ist die komplette Eclipse-Plattform selbst zu einer Handvoll Features zusammengefasst. Diese Features sind die Plattform selbst, die PDE, die Rich-Client-Plattform, und die Entwicklungsumgebungen JDT[5] und SDK[6].
Ein Feature kann ebenfalls als Eclipse-Projekt erstellt werden. Die Komposition eines Features gestaltet sich dabei recht einfach. Dialoggestützt müssen nur die Plug-Ins ausgewählt werden, die zum Feature gehören (siehe Abbildung B.4). In dem Feature

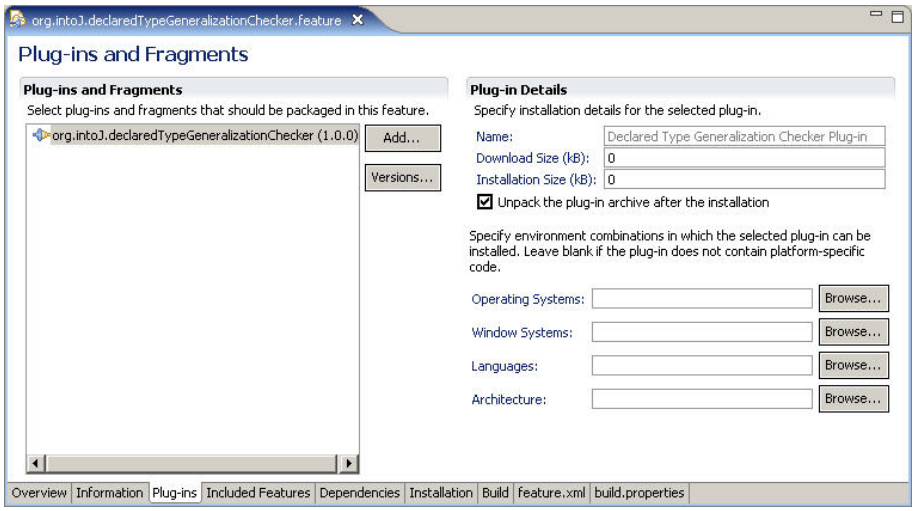

Abbildung B.4.: Dialog zum Zusammenfassen von Plug-Ins zu Features

werden auch die Lizenzbedingungen eines Plug-Ins festgelegt, somit müssen diese

[5]Java Development Tools
[6]Software Development Kit

nicht jedem einzelnen Plug-In beigelegt werden.
An dieser Stelle angelangt haben wir Konzepte kennen gelernt, wie Plug-Ins einerseits noch feiner unterteilt werden können und wie sie andererseits auf oberster Ebene wieder zusammengefasst werden. Bisher ist aber noch nicht klar, wie ein Plug-In zu einem Anwender kommt, um dort installiert zu werden. Es besteht die Möglichkeit, das Plug-In in ein ZIP-Archiv zu stecken, das beim Anwender entpackt und der Inhalt in die entsprechenden Verzeichnisse seiner Eclipse-Installation kopiert werden muss. Für viele Plug-Ins mag das ein praktikabler Weg sein, da man die ZIP-Datei recht einfach per E-Mail verschicken oder im Internet zum Download bereitstellen kann. Allerdings setzt dies beim Beutzer zumindest grundlegendes Wissen über Eclipse voraus. Ein weitaus komfortablerer Weg wird wieder von Eclipse selbst bereitgestellt.
Features können über Update-Sites veröffentlich werden. Unter einer Update-Site versteht man eine von der PDE generierte Internet-Seite, die Features zur Installation bereitstellt. Eine Update-Site wird mit der PDE als neues Projekt angelegt, mit einigen wenigen Dialogen lassen sich Features zum Projekt hinzufügen. Der Build des Projektes erzeugt die Verzeichnisstruktur und die entsprechenden HTML-Seiten. Die Update-Site enthält ein Manifest in der Datei *site.xml*. Der Update-Manager der Eclipse-Plattform kann mit Hilfe des Manifests die Features der Seite automatisch laden und installieren. Der Update-Manager wird über das Menü *Help - Software Updates - Find and Install ...* gestartet. Es kann geprüft werden, ob Updates zu installierten Features vorhanden sind, oder es kann eine neue Update-Site eingetragen werden, auf der nach neuen Features gesucht werden soll. Eine Update-Site muss nicht zwingend auf einem Server im Internet liegen, sie kann auch lokal auf dem Rechner des Anwenders gespeichert sein.
Nachdem ein neues Feature installiert wurde, sollte die Eclipse-Plattform neu gestartet werden. Bei manchen Features ist dies nicht erforderlich, jedoch ist das schwer zu erkennen, weshalb die Plattform einen Neustart meist empfiehlt. Bei diesem wird erreicht, dass die Referenzen der Plug-Ins neu gebunden werden.

B.5. Lebenszyklus

Der Lebenszyklus von Plug-Ins ist durch die OSGi-Spezifikation festgelegt. Ein Bundle kann sich in den Zuständen *INSTALLED*, *UNINSTALLED*, *RESOLVED*, *STARTING*, *ACTIVE* oder *STOPPING* befinden. Im Zustand *INSTALLED* sind alle

Bundles, sobald die Plattform gestartet wird. Es folgt eine Initilisierungsphase, nach der das Bundle den Zustand *RESOLVED* annimmt, wenn alle externen Referenzen des Bundles erfolgreich gebunden wurden. Durch einen Update-/Refresh-Mechanismus kann ein Bundle zur Laufzeit ausgetauscht werden und nimmt erneut den Zustand *INSTALLED* an. Im Zustand *RESOLVED* kann das Bundle gestartet, aber auch deinstalliert werden, was den Zustand *UNINSTALLED* zur Folge hat. Für eine detaillierte Betrachtung der Zustände sei auf die Spezifikation aus OSGi-Alliance (2006) verwiesen. Die folgende Abbildung illustriert die Zustände und die Zustandsübergänge:

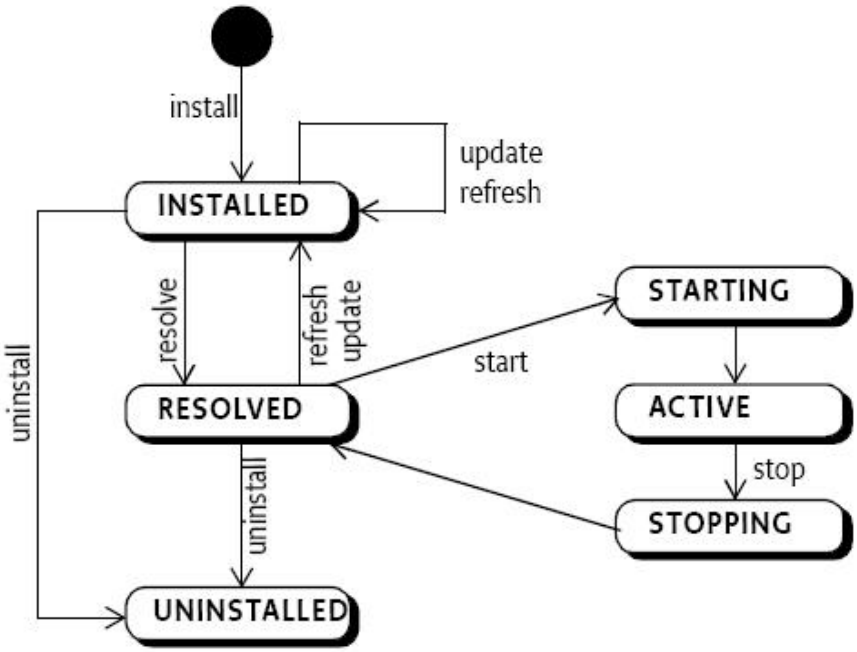

Abbildung B.5.: Lebenszyklus von Bundles gemäß OSGi-Standard
aus: OSGi-Alliance (2006)

Den Zustandsübergängen entsprechen in der Implementierung Methoden, mit denen der Lebenszyklus von Bundles durch die Plattform beeinflusst wird. Die Spezifikation sieht vor, dass diese Methoden ein Bundle-Objekt bereitstellen, das für jedes geladene Bundle erzeugt wird. Den Lebenszyklus steuert dann ein zentraler Bundle-Manager. Um den Start der Plattform nicht unnötig zu verzögern, gibt der OSGi-Standard vor, dass nicht alle Bundles sofort gestartet werden müssen. Der Standard empfiehlt verschiedene Start-Levels, in denen nur die dem aktuellen Level zugeordneten Bundles gestartet werden. Die Eclipse-Plattform weicht an dieser Stelle aber vom Standard ab und definiert das Lazy-Loading. Dies bedeutet, dass Plug-Ins nicht in einer festen

Reihenfolge gestartet werden, sondern erst, wenn zum ersten Mal auf die Funktionalität zugegriffen wird. Zwar müssen beim Start der Plattform gewisse Plug-Ins sofort geladen werden, aber bei den meisten genügt es, sie bei Bedarf zu starten. Natürlich verschlechtert dies beim ersten Aufruf die Zugriffszeit und führt zu einer gewissen Verzögerung der Operation, dafür bleibt die Dauer des Starts des Gesamtsystems auch mit wachsender Anzahl von Plug-Ins relativ konstant. Außerdem ist der Plug-In-Entwickler davon befreit, entscheiden zu müssen, wann sein Plug-In am günstigsten gestartet werden soll. Im Zweifelsfall würde er sich wahrscheinlich dafür entscheiden, dies so früh wie möglich zu tun. Darüber hinaus wird unnötige Ladezeit für Plug-Ins gespart, die nicht benutzt werden.
Das Herunterfahren der Plattform wird auch durch den zentralen Bundle-Manager gesteuert und erfolgt in umgekehrter Ladereihenfolge. Die konkrete Implementierung des OSGi-Bundle-Managers enthält die Klasse *org.eclipse.core.runtime.adaptor.EclipseStarter* der Eclipse-Runtime.

C. Code-Audits beim Compiler-Lauf

Das lateinische Wort 'audit' lässt sich mit dem Begriff Anhörung übersetzen. Ein Audit wird von einem speziell geschulten Auditor durchgeführt und dient der systematischen Untersuchung von Prozessabläufen. Ein Audit soll einen Vergleich zwischen dem erreichten Ist-Zustand und der ursprünglich definierten Zielsetzung ermöglichen. Grundlage für ein Audit sind Anforderungen und Richtlinien, anhand derer Probleme oder Verbesserungspotential aufgespürt werden sollen. Audits sind mittlerweile in vielen Bereichen ein Werkzeug zur Qualitätssicherung, z.B. im Finanzwesen, bei Produktionsabläufen, beim Umweltschutz etc. Im Softwareentwicklungsprozess spielen Audits auch eine immer größer werdende Rolle. In den nachfolgenden Abschnitten werden deshalb Strategien und Ziele von Code-Audits beschrieben. Darüber hinaus werden die Grundlagen für automatische Audits in Eclipse allgemein erläutert, die die Basis des Declared Type Generalization Checker Plug-Ins sind.

C.1. Strategie und Ziele

Audits werden je nach Anwendungsgebiet unterschiedlich durchgeführt. Während es beispielsweise bei einem IT-Sicherheits-Audit anhand eines Sicherheitskonzeptes[1] abzuhaken gilt, welche Vorgaben bereits erfüllt sind und welche noch nicht, sieht die Auditierung von Quelltexten wesentlich anders aus. Die Richtlinien, die für diese Audits die Grundlage bilden, sind oft komplex und abhängig vom jeweiligen Anwendungsfall. Die wichtigste Richtlinie ist die Syntax der verwendeten Programmiersprache selbst. Nur ein syntaktisch korrektes Programm kann überhaupt übersetzt werden. Dieses Audit übernimmt allerdings schon ein Compiler bzw. Interpreter während der Entwicklung. Die nächste Quelle von Richtlinien sind Design-Vorgaben für Quelltexte, die in vielen Firmen vorhanden sind. In solchen Design-Vorgaben sind beispielsweise

[1] Richtlinien zur IT-Sicherheit und zur Erstellung von Sicherheitskonzepten veröffentlicht das Bundesamt für Sicherheit in der Informationstechnik: http://www.bsi.bund.de

Regeln definiert, wie Variablen zu benennen sind, wie Kommentare im Quelltext auszusehen haben oder wie die Aufteilung von Klassen in Pakete zu erfolgen hat. Dieses Regelwerk existiert meist als gedrucktes Papierwerk und es wird erwartet (besser gesagt gehofft), dass sich alle Entwickler daran halten.
Ein weiterer Faktor, der das Aussehen von Quelltexten und damit den verwendeten Satz an Regeln bestimmt, ist das Projekt selbst. Wichtig sind hierbei die Kundenanforderungen an die Software und die davon abhängige Auswahl von Technologien, die zur Umsetzung verwendet werden. Für eine webbasierte Anwendung gelten andere Rahmenbedingungen als für eine Desktop-Anwendung. Eine kleine Anwendung zur Verwaltung der Portokasse rechtfertigt keine Client-Server-Architektur. Ausschlaggebend für ein späteres Audit sind solche Vorüberlegungen, die zu Beginn eines Projekts angestellt und im Idealfall niedergeschrieben wurden. Darüber hinaus ergeben sich auditierbare Regeln aus den Vorgaben, die man landläufig als gutes Design bezeichnet. Wichtigste Vertreter des guten Designs sind die Design-Patterns[2], die immer wiederkehrende Problemstellungen in der Softwareentwicklung auf einen definierten Satz von Mustern herunterbrechen, um, zumindest vom Ansatz her, standardisierte Lösungen aufzuzeigen. Wenngleich diese Muster kein Garant für gute oder verständliche Quelltexte sind, so sind sie doch ein Mittel, um Problemen einen Namen zu geben und unter den Entwicklern ein gemeinsames Vokabular einzuführen.
Ein weiterer Verteter des guten Designs ist die bereits zu Anfang dieser Arbeit vorgestellte Typgeneralisierung. Sie ermöglicht, lose gekoppelte Systeme zu entwickeln, indem möglichst keine konkreten Typen verwendet werden, sondern stattdessen solche, die den Ansprüchen der Minmalität genügen.
Der aufgezeigte Katalog an Richtlinien, die es bei der Code-Auditierung zu prüfen gilt, ist bei weitem nicht vollständig; er soll aber einen Anhaltspunkt geben, wie vielschichtig dieses Thema ist. Im weiteren werden Strategien der Auditierung diskutiert.
Regeln, die in Form von Papier vorliegen, können nur schwer geprüft werden, bzw. dies muss manuell durch eine Person erfolgen, die nicht der Entwickler selbst ist, denn dieser findet nur schwerlich Regelverstöße, die er ohnehin nicht hätte begehen sollen. Außerdem muss der Auditor über genügend Wissen vom Regelwerk, gegen das er prüfen muss, und über das Software-Projekt, das zu prüfen ist, verfügen. Eine solche Prüfung ist umständlich und zeitaufwändig, sie kann deshalb nicht kontinuierlich im Projekt erfolgen, sondern erst am Ende oder zumindest zu definierten Meilensteinen.

[2]siehe hierzu: Gamma u. a. (2001)

Jedoch wird eine Änderung in einer Software umso teurer, je später sie erfolgt.[3] Deshalb wäre es wünschenswert, dass eine Auditierung in kurzen Intervallen oder sogar während des Entwicklungsprozesses in der Entwicklungsumgebung erfolgt. Realistisch ist diese Forderung nur, wenn ein hoher Automatisierungsgrad erreicht werden kann. Ein hervorragendes Beispiel in dieser Richtung sind Unit-Tests. Hierbei lassen sich jederzeit kleine Funktionseinheiten automatisiert und wiederholbar prüfen. Der Entwickler erhält unmittelbar eine Rückmeldung, ob sich eine Code-Änderungen negativ auf andere Stellen im Programm ausgewirkt hat. Eine ähnliche Arbeitsweise ist auch für andere Aspekte bei der Programmierung wünschenswert, da viele der oben vorgestellten Anforderungen der Bequemlichkeit oder dem Zeitdruck geopfert werden, um viel zu spät mit enormen Aufwänden wieder beachtet zu werden (wenn überhaupt).

C.2. Grundlagen in Eclipse

Der Ansatz, automatische Code-Audits gleich in der Entwicklungsumgebung zu verankern, schafft die Grundlage dafür, dass die Audits vom Entwickler eigenverantwortlich immer wieder durchgeführt werden können. Die folgenden Abschnitte zeigen, welche Mechanismen und Erweiterungmöglichkeiten Eclipse bereitstellt, um Quelltexte prüfen zu können. Man wird sehen, dass Eclipse ermöglicht, unmittelbar in den Entwicklungsprozess einzugreifen, um schon nach kleinsten Änderungen Prüfungen durchzuführen. Der Entwickler kann so laufend auf Verbesserungspotential hingewiesen werden. Im Idealfall können sogar Lösungsmöglichkeiten angeboten werden, um den Quelltext zu überarbeiten und somit gestellte Anforderungen zu erfüllen.

C.2.1. Natures und Builders

Eine wichtige Komponente des Entwicklungsprozesses stellt der Builder dar. Builder werden immer dann aufgerufen, wenn Eclipse automatisch oder der Entwickler manuell die Anweisung gibt, die Quelltexte neu zu bauen. In diesem Zusammenhang bedeutet ein Programm zu bauen, seine Übersetzung anzustoßen. Deshalb ist auch der wichtigste Builder der Java-Builder, der den Java-Compiler für den aktuellen Quelltext aufruft. Die Funktionalität von Buildern ist aber nicht darauf beschränkt, Quelltex-

[3] siehe auch: Steimann u. a. (2005)

te zu übersetzen. Beispielsweise können Builder vor dem Übersetzungsvorgang Metaersetzungen oder Transformationen durchführen, aber auch nach der Übersetzung Prüfungen beginnen. Hierfür ist es natürlich wichtig, dass festgelegt werden kann, in welcher Reihenfolge Builder gestartet werden. Builder haben bei ihrem Aufruf Zugriff auf alle Ressourcen. Eclipse stellt sogar explizit Informationen bereit, welche Ressourcen sich seit dem letzten Lauf des Builders verändert haben. Um einen Builder in der Plattform bekannt zu machen und implementieren zu können, muss im Manifest eine entsprechende Extension für den Extension-Point *org.eclipse.core.resources.builders* eingetragen werden:

```
<extension id="org.intoJ.plugInBuilder"
           name="Builder dieses Plug-Ins"
           point="org.eclipse.core.resources.builders">
    <builder>
        <run class="org.intoJ.plugIn.Builder"/>
    </builder>
</extension>
```

Das Attribut *class* im Element *builder* gibt an, welche Klasse den Builder implementiert. Die Klasse muss zwingend eine Ableitung der Klasse *org.eclipse.core.resources.IncrementalProjectBuilder* sein. Diese Klasse enthält eine abstrakte Methode *build(...)*, die in der abgeleiteten Klasse überschrieben wird und durch die Plattform bei einem Build aufgerufen wird. Es ergibt sich folgendes Gerüst des Builders:

```
public class Builder extends IncrementalProjectBuilder {
    protected IProject[] build(int kind, Map args,
                               IProgressMonitor monitor)
    throws CoreException {
        IResourceDelta delta = getDelta(getProject());
        ...
    }

    protected void clean(IProgressMonitor monitor)
    throws CoreException {
        ...
    }
}
```

Die Methode *build(...)* wird immer aufgerufen, wenn der Entwickler einen Build explizit anstößt oder wenn Eclipse dies automatisch macht.[4] Der Parameter *kind* enthält dabei die Information, welche Build-Variante angefordert wurde. Der Wert

- *INCREMENTAL_BUILD* sagt aus, dass sich einige Ressourcen seit dem letzten Build geändert haben und es genügen würde, nur diese zu bearbeiten,
- *FULL_BUILD* wird übergeben, wenn angefordert wird alle Ressourcen neu zu bauen und
- *AUTO_BUILD* besagt, dass es sich um eine Anforderung von Eclipse handelt und nicht durch den Entwickler.[5]

Im Fall der Build-Art *INCREMENTAL_BUILD* kann mit der Methode *getDelta()* eine Differenz zwischen dem aktuellen Stand und dem letzten Build-Lauf abgerufen werden. Wie der Builder auf die verschiedenen Anforderungen reagiert, ist implementierungsabhängig; deshalb kann er trotz der Aufforderung *INCREMENTAL_BUILD* die Ressourcen komplett behandeln. Das Gerüst enthält außerdem die Methode *clean(...)*; sie wird immer dann aufgerufen, wenn der Entwickler die Anweisung erteilt, die Ergebnisse der Build-Läufe zu verwerfen.[6] Da die Methode in der Basisklasse nicht abstrakt ist, muss sie nicht überschrieben werden. Allerdings führt die Methode keine Operationen aus. Müssen beim Clean-Build aber Ressourcen entfernt werden, muss der Builder die Methode überschreiben.
Builder werden nicht direkt Eclipse-Projekten zugewiesen. Einem Projekt wird vielmehr eine Nature zugewiesen. Um der wörtlichen Übersetzung zu entsprechen, zeichnet eine Nature ein Projekt aus, sie verleiht ihm gewisse Eigenschaften. Jedem Java-Projekt ist auch die Java-Nature zugewiesen, um es als solches zu kennzeichnen. Die Java-Nature fügt dem Projekt auch den Java-Builder hinzu, damit es richtig übersetzt werden kann. In dieser Weise müssen auch alle anderen Builder über Natures mit einem Projekt verknüpft werden. Eine Nature wird wiederum über eine Extension angelegt, wie sie im folgenden Beispiel zu sehen ist:

```
<extension id="org.intoJ.plugInNature"
           name="Nature dieses Plug-Ins"
```

[4] Projektspezifisch kann eingestellt werden, dass bei jedem Speichern der Build gestartet wird.
[5] Automatische Build-Läufe sind allerdings abschaltbar, dann obliegt die Steuerung rein dem Entwickler selbst.
[6] Man nennt dies einen Clean-Build.

```
          point="org.eclipse.core.resources.natures">
      <builder id="org.intoJ.plugIn.Builder"/>
      <runtime>
         <run class="org.intoJ.plugIn.Nature"/>
      </runtime>
</extension>
```

Für eine Nature wird der Extension-Point *org.eclipse.core.resources.natures* erweitert. Ihre Definition gibt eine Klasse an, welche die Implementierung enthält. Darüber hinaus werden die Builder aufgeführt, die diese Nature enthält. Die Klasse der Nature muss das Interface *org.eclipse.core.resources.IProjectNature* implementieren. Somit hat eine Nature folgenden Aufbau:

```
public class DeclaredTypeGeneralizationCheckerNature implements
IProjectNature {
    private IProject project;

    public void configure() throws CoreException {
        ...
    }

    public void deconfigure() throws CoreException {
        ...
    }

    public IProject getProject() {
        return project;
    }

    public void setProject(IProject project) {
        this.project = project;
    }
}
```

Die Methoden *configure()* und *deconfigure()* werden von der Eclipse-Plattform aufgerufen, wenn die Nature einem Projekt hinzugefügt bzw. aus ihm entfernt wird. In der Methode *configure()* können Initialisierungen vorgenommen werden, um die

Nature zu benutzen. In erster Linie werden in den Methoden die Builder der Nature dem Projekt hinzugefügt bzw. entfernt. Ebenfalls beim Zuweisen der Nature wird die Methode *setProject(...)* aufgerufen. Sie übergibt der Nature eine Instanz des Eclipse-Projektes, dem sie zugewiesen wurde. Über diese Instanz kann später der Builder auf alle Ressourcen des Projekts zugreifen.

C.2.2. Parsen des Abstract-Syntax-Tree (AST)

Zunächst ist der Begriff des Abstract-Syntax-Tree selbst zu definieren. Ein AST ist die Darstellung eines linearen syntaxbehafteten Dokuments als logische Baumstruktur. Der AST entsteht beim Parsen[7] eines Quelldokuments. Durch die Transformation der flachen Darstellung in einen Baum lassen sich die Symbole des Quelltextes leicht traversieren. In den Baum können an beliebigen Stellen ohne großen Aufwand Knoten eingehängt, entfernt oder geändert werden. In Eclipse lässt sich ein AST für jede Kompilationseinheit erstellen. Dabei ist eine Kompilationseinheit mit einer Java-Datei gleichzusetzen. Der AST enthält alle Deklarationen und Ausdrücke der Java-Datei als Knoten. Ein AST wird deshalb schnell sehr groß und komplex. Der AST in Eclipse kennt bis zu 84 verschiedene Knotentypen, von einfachen Importdeklarationen über Variablendeklarationen bis hin zu If-Else-Blöcken. Bei der Bearbeitung des AST kommt das Visitor-Pattern[8] zum Einsatz. Eclipse stellt die abstrakte Klasse *org.eclipse.jdt.core.dom.ASTVisitor* zur Verfügung, die Deklarationen der *visit(...)*-Methoden für alle möglichen AST-Knoten enthält. Um einen Visitor zu implementieren, muss die abstrakte Klasse abgeleitet werden. Man überschreibt die entsprechenden *visit(...)*-Methoden für die Elemente des AST, bei deren Auffinden man benachrichtigt werden möchte. Das folgende Beispiel implementiert einen einfachen Visitor, der Methodendeklarationen im AST findet und beispielhaft den Methodennamen und den Rückgabewert ermittelt:

```
public class MethodASTVisitor extends ASTVisitor {
    public boolean visit(MethodDeclaration node) {
        String methodName = node.getName().getIdentifier();
        if (! node.isConstructor()) {
            String returnType = node.getReturnType2().toString();
        }
```

[7] Der Transformationsprozess in die Baumstruktur wird als Parsen bezeichnet.
[8] Für eine detaillierte Erklärung sei auf Gamma u. a. (2001) verwiesen.

```
        return true;
    }

    public void endVisit(MethodDeclaration node) {
        ...
    }
}
```

Hat eine *visit(...)*-Methode den Rückgabewert *true*, wird der Zweig des AST weiter durchwandert. Gibt sie hingegen *false* zurück, wird dieser Zweig nicht weiter untersucht. Die Traversierung geht auf die übergeordnete Hierarchiestufe zurück, um dort fortzufahren. Die Klasse *ASTVisitor* enthält zusätzlich noch für jeden Knotentyp eine Methode *endVisit(...)*, die aufgerufen wird, wenn aus der tieferliegenden Baumstruktur zurückgekehrt wird. Im obigen Beispiel wurden zunächst alle Knoten durchwandert, die der Methodendeklaration untergeordnet sind, bevor *endVisit(...)* aufgerufen wurde. Den Methoden des Visitors wird beim Aufruf der gefundene AST-Knoten übergeben. Aus diesem können Informationen zum Knoten abgerufen werden. In den Visitor-Methoden steckt nun die Logik, den Baum zu verändern oder Prüfungen auf diesem Knoten durchzuführen. Der Aufruf, der notwendig ist, um einen AST zu erstellen und den Visitor zu starten, wird in folgendem Beispiel gezeigt:

```
// Der Parser wird erzeugt, dabei muss angegeben werden, welche
// Java-Version das Quelldokument hat, JSL3 entspricht Java 1.5
ASTParser parser = ASTParser.newParser(AST.JLS3);
// Dem Parser wird mitgeteilt, dass er komplette
// Kompilationseinheiten verarbeiten soll, es wären auch einzelne
// Ausdrücke möglich
parser.setKind(ASTParser.K_COMPILATION_UNIT);
// Die zu parsende Kompilationseinheit wird an den Parser übergeben
parser.setSource(sourceUnit);
// Der AST wird erzeugt, wobei ein Progress-Monitor, der übergeben
// wird, Statusinformationen über den Fortschritt liefert
CompilationUnit unit =
    (CompilationUnit) parser.createAST(monitor);
// Dem erzeugten AST wird der Visitor übergeben, der seine Suche
// nach Methodendeklarationen beginnt
```

```
unit.accept(new MethodASTVisitor());
```

C.2.3. Problem-Marker

Bisher wurde gezeigt, wie man durch den Builder Projekte und die zugehörigen Kompilationseinheiten geliefert bekommt und wie man die gelieferten Einheiten effizient durchsuchen kann, um bestimmte Stellen Prüfungen zu unterziehen. Nur welchen Wert haben Prüfungen, wenn kein Ergebnis sichtbar wird? Eine Notlösung wäre, die Erkenntnisse der Prüfung in eine Datei zu schreiben. Dort würden sie aber leicht übersehen und hätten keinen direkten Bezug zum geprüften Quelltext. Glücklicherweise bietet Eclipse für dieses Problem auch Lösungen an. Durch Erweiterung interner Plug-Ins lassen sich Hinweise in einer Liste anzeigen, aber auch gezielt Markierungen im Quelltext setzen. Dieser Mechanismus ist aus der Java-Entwicklung bereits bestens bekannt. Der Java-Builder zeigt bei seinem Lauf Compiler-Fehler oder -Warnungen als Markierungen an. Fehler sind im Quelltext und in der Tabelle meist rot hervorgehoben, Warnungen hingegen gelb. Die Liste, die alle Markierungen enthält, dürfte bei jedem Entwickler stets geöffnet sein, da sie einen guten Überblick gibt, wo Probleme im Programm vorliegen, siehe Abbildung C.1:

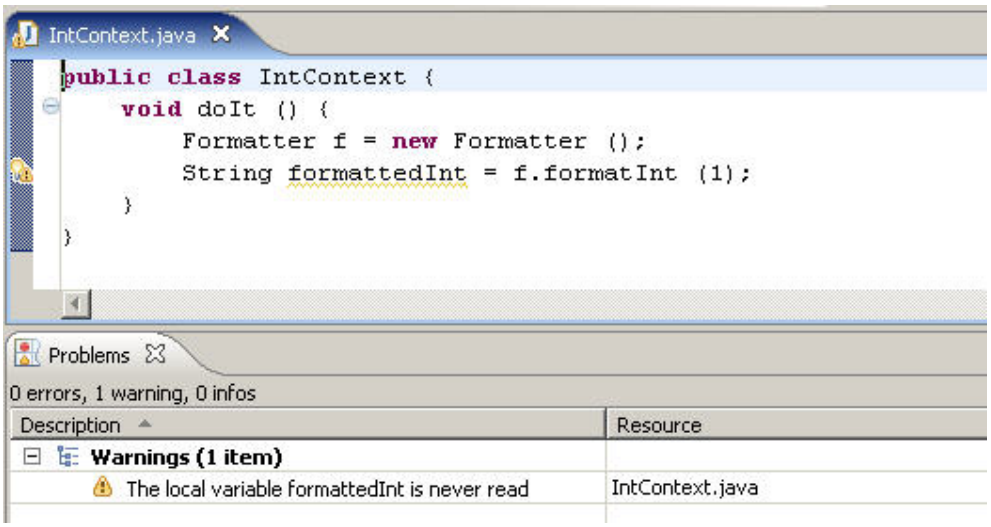

Abbildung C.1.: Anzeige einer Warnung im Quelltext und als Tabelle

Markierungen, die in der Liste angezeigt werden, bezeichnet man als Problem-Marker,

Markierungen im Quelltext sind Text-Marker. Um sie verwenden zu können, muss zunächst der entsprechende Extension-Point erweitert werden:

```
<extension id="org.intoJ.plugInMarker" name="Marker des Plug-Ins"
           point="org.eclipse.core.resources.markers">
    <super type="org.eclipse.core.resources.problemmarker"/>
    <super type="org.eclipse.core.resources.textmarker"/>
    <persistent value="true"/>
</extension>
```

Der hier zu erweiternde Extension-Point ist *org.eclipse.core.resources.markers*. Mit dem Element *super* wird angegeben, welche Markierungen von der Extension erzeugt werden. In diesem Beispiel werden gleichzeitig Problem- und Text-Marker gesetzt. Enthält das Element *persistent* als Attributwert *true*, werden die Markierungen persistent gespeichert und werden auch nach dem Neustart von Eclipse angezeigt. Neben den beiden bereits gezeigten Markierungsarten gibt es Bookmark- und Task-Marker. Sie unterscheiden sich nicht wesentlich vom Problem-Marker, sie werden lediglich in einer anderen Liste der Oberfläche angezeigt. Bookmark-Marker dienen rein als Lesezeichen, um Stellen im Quelltext schneller wiederzufinden, hingegen enthalten die Task-Marker Arbeitsanweisungen, die zu einem späteren Zeitpunkt noch auszuführen sind. Das Setzen einer Markierung sieht im Programm wie folgt aus:

```
IMarker marker =
    resource.createMarker("org.intoJ.plugInMarker");
// Hier wird die Position in der Datei angegeben, an der die Markierung
// beginnen soll
marker.setAttribute(IMarker.CHAR_START, startPosition);
// Hier die Endpostion
marker.setAttribute(IMarker.CHAR_END, endPosition);
// Der Schweregrad steuert, ob die Markierung rot oder gelb angezeigt
// wird
marker.setAttribute(IMarker.SEVERITY, IMarker.SEVERITY_WARNING);
// Der Text wird angezeigt und sollte eine aussagekräftige
// Beschreibung des Problems sein
marker.setAttribute(IMarker.MESSAGE, message);
// Die Zeilennummer wird in der Liste der Markierungen angezeigt
marker.setAttribute(IMarker.LINE_NUMBER, lineNumber);
```

Die Markierung wird gesetzt, indem auf einem Objekt, das vom Typ *org.eclipse.core.resources.IResource* ist, die Methode *createMarker()* mit einem eindeutigen Bezeichner aufgerufen wird. Der Bezeichner muss dem aus der Extension-Definition entsprechen. Durch diese eindeutige Verknüpfung zwischen Manifest und Programm erkennt die Plattform, dass in diesem Fall der Marker als Problem- und Text-Marker erzeugt werden muss. Das Objekt vom Typ *IResource* repräsentiert eine vollständige Datei, dabei muss es sich, im Gegensatz zu einer Kompilationseinheit, nicht mal um eine Java-Datei handeln. Die Markierung wird mit verschiedenen Attributen versehen, die das Erscheinen der Markierung in der Oberfläche steuern. Markierungen lassen sich auf ähnlich einfache Weise auch wieder entfernen:

```
// Von einem Objekt des Typs IResource können die Markierungen, die
// über den eindeutigen Bezeichner referenziert werden, entfernt
// werden. Der zweite Parameter steuert, ob auch Subtypen der zu
// löschenden Markierung entfernt werden sollen. Mit dem dritten
// Parameter legt man fest, ob nur Marker der aktuellen Resource
// gelöscht werden oder auch die von allen abhängigen Resourcen.
resource.deleteMarkers("org.intoJ.plugInMarker", false,
                       IResource.DEPTH_INFINITE);
```

Objekte vom Typ *IResource* implementieren die Methode *deleteMarkers*. Es ist aber nur möglich, alle Markierungen eines Typs innerhalb der Resource zu löschen. Das Löschen von ausgewählten Markierungen ist demnach nicht möglich. Objekte vom Typ *org.eclipse.core.resources.IProject*, die ein Java-Projekt repräsentieren, implementieren auch eine Methode *deleteMarkers()* mit den gleichen Parametern. Somit ist es möglich, alle Markierungen eines Typs in einem ganzen Projekt zu löschen, ohne durch die einzelnen Resourcen iterieren zu müssen.

C.2.4. Marker-Resolution

Wenn Quelltexte wild mit Markierungen zu Problemen übersät werden, ist es für einen Entwickler natürlich nützlich, zugleich eine Hilfestellung zu bekommen, wie er die markierten Probleme lösen kann. Dies wird durch sogenannte Marker-Resolutions ermöglicht. Diese werden definiert und bestimmten Markierungen zugewiesen. Tritt die Markierung dann im Quelltext auf, kann der Entwickler die Hilfestellung, den sogenannten Quick-Fix, über das Kontextmenü der Markierung oder durch Drücken

der Tastenkombination *STRG+1* aufrufen. Die Hilfe, die angeboten wird, ist kontextabhängig und kann, wenn sie denn angeboten wird, auch immer ausgeführt werden. Wenn mehrere Alternativen vorhanden sind, um das Problem zu lösen, muss der Entwickler selbst entscheiden. Die folgende Abbildung C.2 zeigt die Lösungsvorschläge, wenn eine benötigte Importdeklaration fehlt:

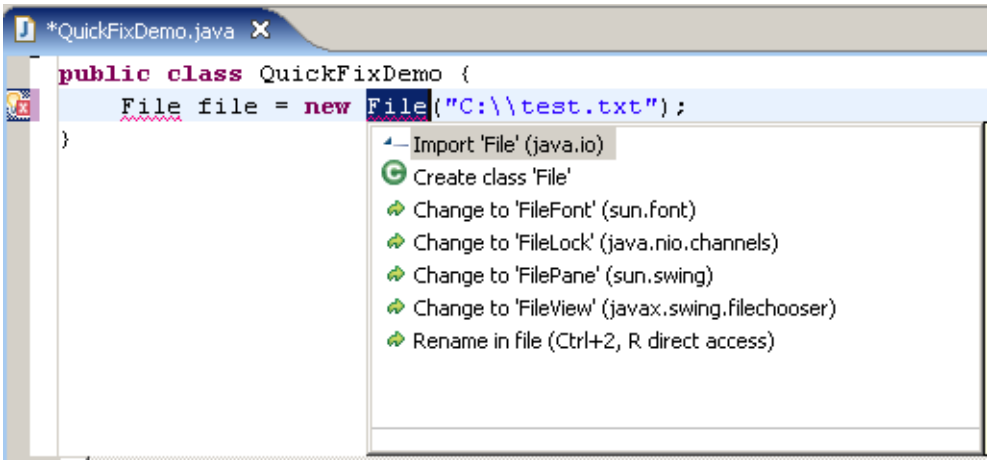

Abbildung C.2.: Anwendung eines Quick-Fix

Definiert wird die Marker-Resolution zunächst im Manifest wie folgt:

```
<extension id="org.intoJ.plugInMarkerResolution"
           name="MarkerResolution dieses Plug-Ins"
           point="org.eclipse.ui.ide.markerResolution">
    <markerResolutionGenerator class="org.intoJ.plugIn /
        .MarkerResolutionGenerator"
        markerType="org.intoJ.plugInMarker"/>
</extension>
```

Die Erweiterung des Extension-Points *org.eclipse.ui.ide.markerResolution* erfordert das Element *markerResolutionGenerator*, in dem mit dem Attribut *class* eine Klasse angegeben wird, die die Funktionalität des Quick-Fix liefert und das Interface *org.eclipse.ui.IMarkerResolutionGenerator2*[9] implementiert. Das Attribut *markerType* nennt den eindeutigen Bezeichner einer Markierung, bei deren Auftreten die eben

[9]Das Interface IMarkerResolutionGenerator2 ist ein schönes Beispiel der Anwendung der Stabilitätsregel aus Anhang B.1. Man hat im Laufe der Zeit erkannt, dass es aufwändig ist zu testen, ob ein Quick-Fix vorhanden ist, indem man ihn aufruft. Das Interface wurde daraufhin um die

angegebene Klasse den Quick-Fix bereitstellt. Für die Klasse ergibt sich folgendes Gerüst:

```
public class MarkerResolutionGenerator implements
IMarkerResolutionGenerator2 {
    public boolean hasResolutions(IMarker marker) {
        return true;
    }

    public IMarkerResolution[] getResolutions(IMarker marker) {
        ...
    }
}
```

Mit der Methode *hasResolutions(...)* kann zunächst getestet werden, ob für eine Markierung Quick-Fixes verfügbar sind. Ist dies der Fall, liefert die Methode *getResolutions(...)* ein Array mit Objekten vom Typ *org.eclipse.ui.IMarkerResolution* zurück (neuere Plug-Ins Objekte von *org.eclipse.ui.IMarkerResolution2*). Diese Objekte implementieren eine *run(...)*-Methode, die nebenläufig Aktionen ausführt, um das markierte Problem zu lösen. Wie im eingangs gezeigten Beispiel sind dies meist Änderungen des Quelltextes.
Im Gerüst einer entsprechenden Klasse befinden sich neben der *run-(...)*-Methode weitere Methoden, die Informationen zum Quick-Fix liefern. Dabei handelt es sich um den Text für die Beschriftung des Quick-Fix, eine Kurzbeschreibung und ein Bild, das ihn charakterisiert:

```
public class MarkerResolution implements IMarkerResolution2 {
    public String getLabel() {
        return "Quick-Fix für diese Markierung";
    }

    public String getDescription() {
        return "Beschreibung des Quick-Fix für diese Markierung";
```

Methode *hasResolutions()* erweitert, die *true* zurückliefert, wenn ein Quick-Fix implementiert ist. Da es unmöglich war, das bestehende Interface IMarkerResolutionGenerator zu ändern, wurde ein neues geschaffen, welches das bisherige um diese Methode erweitert, somit blieben alle bisherigen Plug-Ins lauffähig. Der Postfix *2* am Namen des neuen Interfaces ist zwar wenig elegant, erfüllt aber seinen Zweck.

```
    }

    public Image getImage() {
        return new Image(...);
    }

    public void run(IMarker marker) {
        ...
    }
}
```

D. Quelltexte des Plug-Ins

D.1. Property-Page

```
public class DeclaredTypeGeneralizationCheckerPropertyPage extends \
  PropertyPage {
    private Button activateBuilderSwitch;
    private Combo checkerSelectionComboBox;
    public static final QualifiedName CHECKER_PROPERTY_KEY = \
      new QualifiedName("org.intoJ.declaredTypeGeneralizationChecker", \
      "checkerSelectionProperty");

    public DeclaredTypeGeneralizationCheckerPropertyPage() {
        super();
    }
    private IProject getProject() {
        return (IProject) getElement();
    }
    protected Control createContents(Composite parent) {
        noDefaultAndApplyButton();
        Composite composite = new Composite(parent, SWT.NONE);
        GridLayout layout = new GridLayout();
        composite.setLayout(layout);
        GridData data = new GridData(GridData.FILL);
        data.grabExcessHorizontalSpace = true;
        composite.setLayoutData(data);

        addBuilderSwitch(composite);
        addSeparator(composite);
        addCheckerCombo(composite);
        if (! ExtensionValidator.isInferTypeLoaded()) {
```

```
            addSeparator(composite);
            addInferTypeLink(composite);
        }
        return composite;
    }
    private void addSeparator(Composite parent) {
        Label separator = new Label(parent, SWT.SEPARATOR |
          SWT.HORIZONTAL);
        GridData gridData = new GridData();
        gridData.horizontalAlignment = GridData.FILL;
        gridData.grabExcessHorizontalSpace = true;
        separator.setLayoutData(gridData);
    }
    private void addInferTypeLink(Composite parent) {
        ...
    }
    private void addBuilderSwitch (Composite parent) {
        activateBuilderSwitch= new Button(parent, SWT.CHECK);
        activateBuilderSwitch.setText(Messages.getString \
          ("PropertyPage.LabelActivateBuilder"));

        try {
            activateBuilderSwitch.setSelection(getProject(). \
              hasNature(DeclaredTypeGeneralizationCheckerNature.NATURE_ID));
        } catch (CoreException e) {
            ...
        }
    }
    private void addCheckerCombo (Composite parent) {
        String defaultSelection = null;
        try {
            defaultSelection = getPersitentCheckerSelection \
              ((IProject) getElement());
        }
        catch (CoreException e) {
            ...
        }
```

```
        Label label= new Label(parent, SWT.NONE);
        label.setText(Messages.getString("PropertyPage.LabelCheckerType"));
        checkerSelectionComboBox = new Combo(parent, SWT.READ_ONLY);
        IExtension[] extensions = ExtensionValidator. \
          getImplementingExtensions();
        for (int i = 0; i < extensions.length; i++) {
            if (ExtensionValidator.isInferTypeAdapter(extensions[i])) {
                if (ExtensionValidator.isInferTypeLoaded()) {
                    checkerSelectionComboBox.add(extensions[i].getLabel());
                    if (defaultSelection.equals(extensions[i]. \
                      getUniqueIdentifier())) {
                        checkerSelectionComboBox.select(checker \
                          SelectionComboBox.indexOf(extensions[i].getLabel()));
                    }
                }
            }
            else {
                checkerSelectionComboBox.add(extensions[i].getLabel());
                if (defaultSelection.equals(extensions[i]. \
                  getUniqueIdentifier())) {
                    checkerSelectionComboBox.select(checker \
                      SelectionComboBox.indexOf(extensions[i].getLabel()));
                }
            }
        }
    }
    public static String getPersitentCheckerSelection (IProject project) \
      throws CoreException {
        IResource resource = (IResource) project.getAdapter(IResource.class);
        String name = resource.getPersistentProperty \
          (DeclaredTypeGeneralizationCheckerPropertyPage.CHECKER_PROPERTY_KEY);

        if (name == null) {
            name = getDefaultCheckerSelection();
        }
        else {
```

```
            if (ExtensionValidator.isInferTypeAdapter(Platform. \
              getExtensionRegistry().getExtension(name))) {
                if (! ExtensionValidator.isInferTypeLoaded()) {
                    name = getDefaultCheckerSelection();
                }
            }
        }

        return name;
    }
    private static String getDefaultCheckerSelection () {
        return Platform.getExtensionRegistry().getExtension \
          ("org.intoJ.declaredTypeGeneralizationChecker. \
          generalizeTypeAdapter").getUniqueIdentifier();
    }
    private void setPersitentCheckerSelection (String name) throws \
      CoreException {
        IResource resource = (IResource) getElement(). \
          getAdapter(IResource.class);
        resource.setPersistentProperty(DeclaredTypeGeneralization \
          CheckerPropertyPage.CHECKER_PROPERTY_KEY, name);
    }
    public boolean performOk() {
        try {
            if (activateBuilderSwitch.getSelection()) {
                if (! getProject().hasNature(DeclaredType \
                  GeneralizationCheckerNature.NATURE_ID)) {
                    DeclaredTypeGeneralizationCheckerNature. \
                    addNatureToProject(getProject());
                }
            }
            else {
                if (getProject().hasNature(DeclaredTypeGeneralization \
                  CheckerNature.NATURE_ID)) {
                    DeclaredTypeGeneralizationCheckerNature. \
                    removeNatureFromProject(getProject());
                }
```

```
            }
        }
        catch (CoreException e) {
            ErrorDialog.openError(getShell(), Messages.getString \
              ("PropertyPage.ErrorTitle"), Messages.getString \
              ("PropertyPage.ErrorMessageBuilder"), e.getStatus());
        }

        String selectedChecker = checkerSelectionComboBox.getText();
        IExtension[] extensions = ExtensionValidator. \
          getImplementingExtensions();
        for (int i = 0; i < extensions.length; i++) {
            if (extensions[i].getLabel().equals(selectedChecker)) {
                try {
                    if (! extensions[i].getUniqueIdentifier(). \
                      equals(getPersitentCheckerSelection((IProject) \
                      getElement()))) {
                        setPersitentCheckerSelection(extensions[i]. \
                        getUniqueIdentifier());
                        getProject().build(IncrementalProjectBuilder. \
                          CLEAN_BUILD, DeclaredTypeGeneralization \
                          CheckerBuilder.BUILDER_ID, null, null);
                    }
                }
                catch (CoreException e) {
                    ...
                }
                break;
            }
        }
        return true;
    }
}
```

D.2. Nature und Builder

D.2.1. Klasse DeclaredTypeGeneralizationCheckerNature

```
public class DeclaredTypeGeneralizationCheckerNature implements \
  IProjectNature {
    private IProject project;
    public static final String NATURE_ID = "org.intoJ.declaredType \
      GeneralizationChecker.declaredTypeGeneralizationCheckerNature";

    public void configure() throws CoreException {...}
    public void deconfigure() throws CoreException {...}
    public IProject getProject() {...}
    public void setProject(IProject project) {...}
    public static boolean addNatureToProject(IProject project)
      throws CoreException {
        if (! project.isOpen()) {
            return false;
        }
        if (project.hasNature(NATURE_ID)) {
            return true;
        }
        IProjectDescription description = project.getDescription();
        List newIds = new ArrayList();
        newIds.addAll(Arrays.asList(description.getNatureIds()));
        newIds.add(NATURE_ID);
        description.setNatureIds((String[]) newIds.toArray(new \
          String[newIds.size()]));
        project.setDescription(description, null);
        return true;
    }
    public static boolean removeNatureFromProject(IProject project)
      throws CoreException {
        if (! project.isOpen()) {
            return false;
        }
        IProjectDescription description = project.getDescription();
```

```
        List newIds = new ArrayList();
        newIds.addAll(Arrays.asList(description.getNatureIds()));
        int index = newIds.indexOf(NATURE_ID);
        newIds.remove(index);
        description.setNatureIds((String[]) newIds.toArray(new \
          String[newIds.size()]));
        project.setDescription(description, null);
        return true;
    }
}
```

D.2.2. Klasse DeclaredTypeGeneralizationCheckerBuilder

```
public class DeclaredTypeGeneralizationCheckerBuilder extends \
  IncrementalProjectBuilder {
    protected Checker checker;
    public static final String BUILDER_ID = "org.intoJ.declared \
      TypeGeneralizationChecker.declaredTypeGeneralizationCheckerBuilder";

    protected IProject[] build(int kind, Map args, IProgressMonitor \
      monitor) throws CoreException {
        checker = Checker.create(getProject(), monitor, this);
        if (kind == FULL_BUILD) {
            fullBuild();
        }
        else {
            IResourceDelta delta = getDelta(getProject());
            if (delta == null) {
                fullBuild();
            }
            else {
                incrementalBuild(delta);
            }
        }
        return null;
    }
    private void clean(IProgressMonitor monitor) throws CoreException {
```

```
        if (! ProblemMarker.deleteMarkers(getProject())) {
            throw new CoreException (new Status (Status.ERROR,
              DeclaredTypeGeneralizationCheckerPlugin.getDefault(). \
              getBundle().getSymbolicName(), Status.ERROR, Messages. \
              getString("Builder.ErrorCleanBuild"),  null));
        }
    }
    private void fullBuild() throws CoreException {
        try {
            checker.performCheck();
        }
        catch (JavaModelException jme) {
            throw new CoreException (new Status (Status.ERROR,
              AstractionCheckerPlugin.getDefault().getBundle(). \
              getSymbolicName(), Status.ERROR, Messages.getString \
              ("Builder.ErrorChecking"), jme));
        }
    }
    protected void incrementalBuild(IResourceDelta delta)
      throws CoreException {
        delta.accept(new CheckerResourceDeltaVisitor(checker));
    }
    public static boolean addBuilderToProject(IProject project)
      throws CoreException {...}
    public static boolean removeBuilderFromProject(IProject project)
      throws CoreException {...}
    private static boolean hasBuilder(IProject project) {...}
}
```

D.3. Code-Prüfungs-Visitors

D.3.1. Klasse Checker

```
public class Checker {
    private static IProject project = null;
    private IJavaProject javaProject;
```

```
private IPackageFragment projectFragments[];
private static IProgressMonitor monitor;
private static IProgressMonitor subMonitor;
private static IncrementalProjectBuilder builder;
private static IDeclaredTypeGeneralizationChecker \
  externalChecker = null;

public static Checker create (IProject project, IProgressMonitor \
  monitor, IncrementalProjectBuilder builder) throws CoreException {
    Checker.monitor = monitor;
    Checker.builder = builder;
    return new Checker (project);
}
private Checker (IProject project) throws CoreException {
    Checker.project = project;
    javaProject = JavaCore.create(project);
    if (javaProject == null) {
        throw new CoreException (...);
    }
    try {
        projectFragments = javaProject.getPackageFragments();
    }
    catch (JavaModelException jme) {
        throw new CoreException (...);
    }

    initExternalChecker();
}
public void performCheck () throws CoreException {
    ICompilationUnit projectUnits[] = null;
    int numberOfDeclarationElements = 0;

    for (int i = 0; i < projectFragments.length; i++) {
        projectUnits = projectFragments[i].getCompilationUnits();
        for (int j = 0; j < projectUnits.length; j++) {
            numberOfDeclarationElements += performCheck( \
              projectUnits[j], false);
```

```
                if (isCheckCancelled()) {
                    return;
                }
            }
        }

        subMonitor = new SubProgressMonitor(monitor, \
          numberOfDeclarationElements);
        subMonitor.beginTask(Messages.getString("Nature.JobName"), \
          numberOfDeclarationElements);
        int numberOfProcessedDeclarationElements = 0;
        for (int i = 0; i < projectFragments.length; i++) {
            projectUnits = projectFragments[i].getCompilationUnits();
            for (int j = 0; j < projectUnits.length; j++) {
                numberOfProcessedDeclarationElements += performCheck( \
                  projectUnits[j], true);
                subMonitor.worked(numberOfProcessedDeclarationElements);
                if (isCheckCancelled()) {
                    return;
                }
            }
        }
        subMonitor.done();
    }
    public int performCheck (ICompilationUnit sourceUnit, boolean
action) throws CoreException {
        ASTParser parser = ASTParser.newParser(AST.JLS3);
        parser.setKind(ASTParser.K_COMPILATION_UNIT);
        parser.setSource(sourceUnit);

        CompilationUnit unit = (CompilationUnit) parser.createAST(null);
        CheckerASTVisitor visitor = new CheckerASTVisitor (this, \
          sourceUnit, sourceUnit.getUnderlyingResource(), unit, action);
        unit.accept(visitor);

        return visitor.getNumberOfDeclarationElements();
    }
```

```
    protected void handleField (FieldDeclaration node, ICompilationUnit
unit, IResource resource, CompilationUnit cUnit) {
        if (! node.getType().isPrimitiveType()) {
            String typeName = node.getType().toString();

            String fieldName = "";
            int fieldStart = 0;
            List fieldNamesList = node.fragments();
            Iterator fieldNameIterator = fieldNamesList.iterator();
            if (fieldNameIterator.hasNext()) {
                VariableDeclarationFragment fragment = \
                  (VariableDeclarationFragment) fieldNameIterator.next();
                fieldName = fragment.getName().getIdentifier();
                fieldStart = fragment.getName().getStartPosition();
            }

            int typeStart = node.getType().getStartPosition();

            SubProgressMonitor fieldSubMonitor = \
              new SubProgressMonitor(subMonitor, 1);
            fieldSubMonitor.beginTask(Messages.getString("Nature.JobName") \
              , 1);

            try {
                if (getExternalChecker().checkType(unit, typeName, \
                  typeStart, fieldName, fieldStart, fieldSubMonitor) == \
                    IDeclaredTypeGeneralizationChecker. \
                      GENERALIZATION_POSSIBLE) {
                    int lineNumber = cUnit.getLineNumber(typeStart);
                    ProblemMarker.prepareFieldMarker(resource, typeName, \
                      fieldName, typeStart, fieldStart, lineNumber);
                }
            }
            catch (Throwable t) {
                IStatus status = new Status (Status.ERROR,
                        DeclaredTypeGeneralizationCheckerPlugin. \
                          getDefault().getBundle().getSymbolicName(),
```

```
                    Status.ERROR, Messages.getString \
                      ("Checker.ErrorVisitor"), t);

            DeclaredTypeGeneralizationCheckerPlugin.getDefault(). \
              getLog().log(status);
        }

        fieldSubMonitor.done();
    }
}
protected void handleMethod (MethodDeclaration node,
  ICompilationUnit unit, IResource resource, CompilationUnit cUnit)
{...}
protected void handleVariable (VariableDeclarationStatement node,
  ICompilationUnit unit, IResource resource, CompilationUnit cUnit)
{...}
private static void initExternalChecker () throws CoreException {
    String extensionId = \
      DeclaredTypeGeneralizationCheckerPropertyPage. \
      getPersitentCheckerSelection(project);
    String classOfExternalChecker = ExtensionValidator. \
      getClassForInterfaceExtension(extensionId);

    try {
        externalChecker = (IDeclaredTypeGeneralizationChecker) \
          Class.forName(classOfExternalChecker).newInstance();
    }
    catch (Exception e) {
        throw new CoreException (...);
    }
}
public static IDeclaredTypeGeneralizationChecker \
  getExternalChecker) {
    if (externalChecker == null) {
        try {
            initExternalChecker ();
            return externalChecker;
```

```
            }
            catch (CoreException e) {
                return null;
            }
        }
        else {
            return externalChecker;
        }
    }
    public boolean isCheckCancelled() {
        if (monitor.isCanceled()) {
            builder.forgetLastBuiltState();
            return true;
        }
        else {
            return false;
        }
    }
    public static IProject getProject() {
        return project;
    }
```

D.3.2. Klasse CheckerASTVisitor

```
public class CheckerASTVisitor extends ASTVisitor {
    private Checker checker;
    private ICompilationUnit unit;
    private CompilationUnit cUnit;
    private IResource resource;
    private boolean action;
    private int numberOfDeclarationElements = 0;

    public CheckerASTVisitor (Checker checker, ICompilationUnit unit, \
      IResource resource, CompilationUnit cUnit, boolean action) {
        this.checker = checker;
        this.unit = unit;
        this.cUnit = cUnit;
```

```
        this.resource = resource;
        this.action = action;
    }
    public boolean visit(FieldDeclaration node) {
        if (! checker.isCheckCancelled()) {
            numberOfDeclarationElements++;

            if (action == true) {
                checker.handleField(node, unit, resource, cUnit);
            }
            return true;
        }
        else {
            return false;
        }
    }
    public boolean visit(MethodDeclaration node) {
        if (! checker.isCheckCancelled()) {
            numberOfDeclarationElements++;

            if (action == true) {
                checker.handleMethod(node, unit, resource, cUnit);
            }
            return true;
        }
        else {
            return false;
        }
    }
    public boolean visit(VariableDeclarationStatement node) {
        if (! checker.isCheckCancelled()) {
            numberOfDeclarationElements++;

            if (action == true) {
                checker.handleVariable(node, unit, resource, cUnit);
            }
            return true;
```

```
        }
        else {
            return false;
        }
    }
    public int getNumberOfDeclarationElements() {
        return numberOfDeclarationElements;
    }
}
```

D.3.3. Klasse CheckerResourceDeltaVisitor

```
public class CheckerResourceDeltaVisitor implements
IResourceDeltaVisitor {
    private Checker checker;

    public CheckerResourceDeltaVisitor (Checker checker) {
        this.checker = checker;
    }
    public boolean visit (IResourceDelta delta) {
        if (! (delta.getResource().getType() == IResource.FILE)) {
            return true;
        }
        if (! delta.getResource().getFileExtension().equalsIgnoreCase("java"))
        {
            return true;
        }

        try {
            IMarker deltaMarkers[] = delta.getResource().findMarkers( \
              ProblemMarker.MARKER_ID, false, IResource.DEPTH_ZERO);
            for (int i = 0; i < deltaMarkers.length; i++) {
                deltaMarkers[i].delete();
            }
            if (delta.getKind() == IResourceDelta.ADDED || delta.getKind() \
              == IResourceDelta.CHANGED) {
                ICompilationUnit unit = (ICompilationUnit) JavaCore.create \
```

```
                (delta.getResource());
            checker.performCheck(unit, true);
        }
    }
    catch (CoreException e) {
        return true;
    }

    return true;
  }
}
```

D.4. Problem-Marker

```
public class ProblemMarker {
    public static final String MARKER_ID = "org.intoJ.declaredType \
      GeneralizationChecker.declaredTypeGeneralizationCheckerProblemMarker";

    public static boolean deleteMarkers (IProject project) {
        try {
            project.deleteMarkers(MARKER_ID, false, IResource.DEPTH_INFINITE);
            return true;
        }
        catch (CoreException e) {
            return false;
        }
    }
    public static boolean prepareFieldOrVariableMarker (IResource resource, \
      String type, String name, int typeStartPosition, \
        int fieldStartPosition, int lineNumber) {
        String message;
        ArrayList<String> replacement = new ArrayList<String>();

        try {
            replacement.add(type);
            replacement.add(name);
```

```
            message = Messages.getString("Marker.Message", replacement);
            setMarker(resource, fieldStartPosition, fieldStartPosition + \
              name.length(), message, lineNumber);
            return true;
        }
        catch (CoreException e) {
            return false;
        }
    }
    public static boolean prepareReturnTypeMarker (IResource resource, \
      String type, String name, int typeStartPosition, int \
      nameStartPosition, int lineNumber) {...}
    public static boolean prepareMethodParameterMarker (IResource resource, \
      String type, String name, String methodName, int \
      typeStartPosition, int nameStartPosition, int lineNumber) {...}
    private static void setMarker (IResource resource, int start, int \
      end, String message, int lineNumber) throws CoreException {
        IMarker marker = resource.createMarker(MARKER_ID);
        marker.setAttribute(IMarker.CHAR_START, start);
        marker.setAttribute(IMarker.CHAR_END, end);
        marker.setAttribute(IMarker.SEVERITY, new Integer(IMarker. \
          SEVERITY_WARNING));
        marker.setAttribute(IMarker.MESSAGE, message);
        marker.setAttribute(IMarker.LINE_NUMBER, lineNumber);
    }
}
```

D.5. Marker-Resolution

D.5.1. Klasse MarkerResolutionGenerator

```
public class MarkerResolutionGenerator implements \
  IMarkerResolutionGenerator2 {
    private IDeclaredTypeGeneralizationChecker externalChecker;

    public boolean hasResolutions (IMarker marker) {
```

```
        if (Checker.getProject() != marker.getResource().getProject()) {
            try {
                Checker.create(marker.getResource().getProject(), null, \
                  null);
            }
            catch (CoreException e) {
                return false;
            }
        }
        externalChecker = Checker.getExternalChecker();

        return externalChecker.hasResolution();
    }
    public IMarkerResolution[] getResolutions (IMarker marker) {
        if (Checker.getProject() != marker.getResource().getProject()) {
            try {
                Checker.create(marker.getResource().getProject(), null, \
                  null);
            }
            catch (CoreException e) {
                return null;
            }
        }
        externalChecker = Checker.getExternalChecker();

        List<IMarkerResolution> resolutions = \
          new ArrayList<IMarkerResolution>();
        resolutions.add(externalChecker.getResolution());

        return (IMarkerResolution[]) resolutions.toArray \
          (new IMarkerResolution[resolutions.size()]);
    }
}
```

D.5.2. Klasse GeneralizeDeclaredTypeMarkerResolution

```
public class GeneralizeDeclaredTypeMarkerResolution implements \
  IMarkerResolution2 {
    public String getDescription() {
        return Messages.getString("GeneralizeTypeMarkerResolution.Description");
    }
    public Image getImage() {
        return AbstractUIPlugin.imageDescriptorFromPlugin \
          ("org.intoJ.declaredTypeGeneralizationChecker", \
          "icons/refactor.gif").createImage();
    }
    public String getLabel() {
        return Messages.getString("GeneralizeTypeMarkerResolution.Label");
    }
    public void run(IMarker marker) {
        try {
            int markerStartPosition = ((Integer) marker.getAttribute \
              (IMarker.CHAR_START)).intValue();
            int markerLength = ((Integer) marker.getAttribute \
              (IMarker.CHAR_END)).intValue() - markerStartPosition;
            ICompilationUnit unit = (ICompilationUnit) JavaCore.create \
              (marker.getResource());

            RefactoringExecutionStarter.startChangeTypeRefactoring(unit, \
              new Shell(), markerStartPosition, markerLength);
            marker.getResource().getProject().build(\
              DeclaredTypeGeneralizationCheckerBuilder.INCREMENTAL_BUILD, \
              DeclaredTypeGeneralizationCheckerBuilder.BUILDER_ID, null, null);

        }
        catch (CoreException e) {
            ErrorDialog.openError(new Shell(), Messages.getString \
            ("RefactoringError.Title"), Messages.getString \
              ("RefactoringError.Text"), e.getStatus());
        }
    }
}
```

D.5.3. Klasse InferTypeMarkerResolution

```
public class InferTypeMarkerResolution implements IMarkerResolution2
{
    public String getDescription() {
        return Messages.getString("InferTypeMarkerResolution.Description");
    }
    public Image getImage() {
        return AbstractUIPlugin.imageDescriptorFromPlugin \
          ("org.intoJ.inferType", "icons/inferType.gif").createImage();
    }
    public String getLabel() {
        return Messages.getString("InferTypeMarkerResolution.Label");
    }
    public void run(IMarker marker) {
        try {
            int markerStart = ((Integer) marker.getAttribute \
              (IMarker.CHAR_START)).intValue();
            int markerLength = ((Integer) marker.getAttribute \
              (IMarker.CHAR_END)).intValue() - markerStart;
            ITextSelection textSelection = new TextSelection(markerStart, \
              markerLength);

            IWorkbenchPage activePage = PlatformUI.getWorkbench(). \
              getActiveWorkbenchWindow().getActivePage();
            IEditorPart activeEditorPart = IDE.openEditor(activePage, \
              marker);

            InferTypeInEditorAction inferTypeAction = \
              new InferTypeInEditorAction();
            inferTypeAction.selectionChanged(null, textSelection);
            inferTypeAction.setActiveEditor(null, activeEditorPart);
            inferTypeAction.run(null);
        }
        catch (CoreException e) {
            ErrorDialog.openError(new Shell(), Messages.getString \
              ("RefactoringError.Title"), Messages.getString \
              ("RefactoringError.Text"), e.getStatus());
```

```
        }
    }
}
```

D.6. Hilfsklassen

D.6.1. Klasse Messages

```
public class Messages {
    private static final String BUNDLE_NAME = "org.intoJ.declaredType \
      GeneralizationChecker.ui.internal.messages";
    private static final ResourceBundle RESOURCE_BUNDLE = \
      ResourceBundle.getBundle(BUNDLE_NAME);

    public static String getString(String key) {
        try {
            return RESOURCE_BUNDLE.getString(key);
        }
        catch (MissingResourceException e) {
            return '!' + key + '!';
        }
    }
    public static String getString(String key, ArrayList replacement) {
        String value = getString(key);
        StringTokenizer tokenizer = new StringTokenizer (value);
        String token;
        StringBuffer buffer = new StringBuffer();

        Iterator replacementIterator = replacement.iterator();
        while (tokenizer.hasMoreTokens()) {
            token = tokenizer.nextToken();
            if (token.equals("%VALUE")) {
                if (replacementIterator.hasNext()) {
                    token = (String) replacementIterator.next();
                }
            }
```

```
            buffer.append(token +" ");
        }

        return buffer.toString();
    }
}
```

D.6.2. Klasse ExtensionValidator

```
public class ExtensionValidator {
    private static IExtensionRegistry registry = \
      Platform.getExtensionRegistry();
    private static IExtensionPoint point = registry.getExtensionPoint \
      ("org.intoJ.declaredTypeGeneralizationChecker. \
      declaredTypeGeneralizationCheckerInterface");

    public static IExtension[] getImplementingExtensions () {
        return point.getExtensions();
    }
    public static boolean isInferTypeAdapter (IExtension extension) {
        return extension.getUniqueIdentifier().equals("org.intoJ. \
          declaredTypeGeneralizationChecker.inferTypeAdapter");
    }
    public static boolean isInferTypeLoaded() {
        Bundle bundle = Platform.getBundle("org.intoJ.inferType");
        if (bundle != null) {
            if (bundle.getState() != Bundle.UNINSTALLED){
                return true;
            }
            else {
                return false;
            }
        }
        else {
            return false;
        }
    }
```

```
    public static String getExtensionLabel (IExtension extension) {
        return extension.getLabel();
    }
    public static IExtension getExtensionById (String id) {
        return registry.getExtension(id);
    }
    public static String getClassForInterfaceExtension (String id)
      throws CoreException {
        IExtension extension = ExtensionValidator.getExtensionById(id);
        IConfigurationElement[] extensionConfig = \
          extension.getConfigurationElements();
        for (int i = 0; i < extensionConfig.length; i++) {
            String attribute = extensionConfig[i].getAttribute("class");
            if (attribute != null) {
                return attribute;
            }
        }
        return null;
    }
}
```

Abbildungsverzeichnis

2.1. Typhierarchie der Java-Klasse Vector 10

3.1. Anwendung des Generalize Declared Type Refactorings 18
3.2. Anwendung des Infer Type Refactorings 19
3.3. Anwendung des Refactorings auf einen bereits bearbeiteten Typ . . . 20

4.1. Interface für Prüfalgorithmen 24
4.2. Adapter zwischen Interface und Prüfalgorithmus 26
4.3. Klassendiagramm des Plug-Ins in UML-Notation 29
4.4. Paketdiagramm der Plug-In-Struktur 30
4.5. Property-Page zur Steuerung des Plug-Ins 37
4.6. Eclipse Online-Hilfe des Plug-Ins 55
4.7. Beispiel der API-Dokumentation mit Javadoc 56

5.1. Gegenüberstellung der Prüfergebnisse in Abhängigkeit von der Projektgröße 63
5.2. Typgeneralisierungsvorschlag des Generalize Declared Type Refactorings 67
5.3. Typgeneralisierungsvorschlag des Infer Type Refactorings 68

A.1. Architektur der Eclipse-Plattform 79

B.1. Anlegen eines neuen Plug-In-Projekts 91
B.2. PDE-Editor für Manifest- und plugin.xml-Datei 92
B.3. Konfiguration der Plug-In-Testumgebung 93
B.4. Dialog zum Zusammenfassen von Plug-Ins zu Features 94
B.5. Lebenszyklus von Bundles gemäß OSGi-Standard 96

C.1. Anzeige einer Warnung im Quelltext und als Tabelle 106
C.2. Anwendung eines Quick-Fix 109

Tabellenverzeichnis

3.1. Refactorings in Eclipse 16

4.1. Aufteilung der Klassen des Plug-Ins in Pakete 31

5.1. Größe und Aufteilung des kommerziellen Projekts 61
5.2. Größe und Aufteilung von JUnit und JHotDraw 61
5.3. Vergleich der Laufzeiten zwischen beiden Algorithmen 62
5.4. Vergleich der Laufzeiten bei inkrementeller Prüfung 64
5.5. Gegenüberstellung der Anzahl gefundener generalisierbarer Deklarationen 65
5.6. Prozentualer Anteil der generalisierbaren Typdeklarationen 66

Literaturverzeichnis

[Arthorne 2004a] Arthorne, John:
How You've Changed! - Responding to resource changes in the Eclipse workspace.
http://www.eclipse.org/articles/Article-Resource-deltas/resource-deltas.html.
November 2004

[Arthorne 2004b] Arthorne, John:
Project Builders and Natures.
http://www.eclipse.org/articles/Article-Builders/builders.html.
November 2004

[Balzert 2000] Balzert, Helmut:
Lehrbuch der Software-Technik, Band 1 Software-Entwicklung.
2. Auflage.
Spektrum Akademischer Verlag, 2000. –
ISBN 3-8274-0480-0

[Bolour 2003] Bolour, Azad:
Notes on the Eclipse Plug-in Architecture.
http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html.
Juli 2003

[Buck 2006] Buck, Caroline:
Umschiffte Klippen - Erfahrungsbericht über die Entwicklung kommerzieller Eclipse-Plug-ins.
In: *Eclipse Magazin*
Volume 5 (2006), S. 78–81. –
ISSN 1861-2296

[Darwin 2002] Darwin, Ian F.:
Java Kochbuch.

O'Reilly, 2002. –
ISBN 3-89721-283-8

[Daum 2005] Daum, Berthold:
Java-Entwicklung mit Eclipse 3.1.
dpunkt.verlag, 2005. –
ISBN 3-89864-338-7

[Degenring 2004] Degenring, Arne:
Qualitätssicherung durch Code Auditing.
In: *Java Spektrum*
Volume 3 (2004), S. 36–40. –
ISSN 1431-4436

[Eclispe Foundation 2003] Eclispe Foundation, Inc.:
BYLAWS OF ECLIPSE FOUNDATION, INC.
http://www.eclipse.org/org/documents/Eclipse BYLAWS 2003_11_10 Final.pdf.
November 2003

[Edgar u. a. 2004] Edgar, Nick ; Haaland, Kevin ; Li, Jin ; Peter, Kimberley:
Eclipse User Interface Guidelines.
http://www.eclipse.org/articles/Article-UI-Guidelines/Contents.html.
Februar 2004

[Enns 2004] Enns, Raphael:
Refactoring in Eclipse.
http://www.cs.umanitoba.ca/∼eclipse/13-Refactoring.pdf.
Februar 2004

[Forster 2006] Forster, Florian:
InferType - Das automatische Extract Interface.
In: *Eclipse Magazin*
Volume 7 (2006), S. 46–47. –
ISSN 1861-2296

[Fowler 2004] Fowler, Martin:
Inversion of Control Containers and the Dependency Injection pattern.
http://www.martinfowler.com/articles/injection.html.
Januar 2004

[Fowler 2006] Fowler, Martin:
Refactorings in Alphabetical Order.
http://www.refactoring.com/catalog/index.html.
2006

[Gamma und Beck 2004] Gamma, Erich ; Beck, Kent:
Eclipse Erweitern - Prinzipien, Patterns und Plug-Ins.
Addison-Weseley, 2004. –
ISBN 3-8273-2238-3

[Gamma u. a. 2001] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John:
Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software.
Addison-Weseley, 2001. –
ISBN 3-8273-1862-9

[Gößner u. a. 2004] Gössner, Jens ; Mayer, Philip ; Steimann, Friedrich:
Interface utilization in the Java Development Kit.
In: *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, ACM Press, 2004, S. 1310–1315. –
ISBN 1-58113-812-1

[Goosens u. a. 2000] Goosens, Michel ; Mittelbach, Frank ; Samarin, Alexander:
Der LATEX-Begleiter.
Addison-Weseley, 2000. –
ISBN 3-8273-1689-8

[Harold und Means 2005] Harold, Elliotte R. ; Means, W. S.:
XML in a Nutshell.
3. Auflage.
O'Reilly, 2005. –
ISBN 3-89721-339-7

[Jeckle u. a. 2004] Jeckle, Mario ; Rupp, Chris ; Hahn, Jürgen ; Zengler, Barbara ; Queins, Stefan:
UML 2 glasklar.
Hanser, 2004. –

ISBN 3-446-22575-7

[JetBrains 2006] JetBrains, s.r.o.:
Refactoring.
http://www.jetbrains.com/idea/features/refactoring.html.
August 2006

[Lippert 2006] Lippert, Martin:
Was Eclipse im Innersten zusammenhält - Unter der Haube, Teil 1: Ein Blick auf die Eclipse Runtime.
In: *Eclipse Magazin*
Volume 5 (2006), S. 19–22. –
ISSN 1861-2296

[Marques 2005] Marques, Manoel:
Exploring Eclipse's ASTParser - How to use the parser to generate code.
http://www-128.ibm.com/developerworks/opensource/library/os-ast/?ca=dgr-lnxw97ASTParser.
April 2005

[Mayer 2003] Mayer, Philip:
Analyzing the use of interfaces in large OO projects.
In: *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM Press, 2003, S. 382–383. –
ISBN 1-58113-751-6

[McGaughey und Archer 2006] McGaughey, Skip ; Archer, Simon:
eclipse - Building Commercial-Quality Plug-ins.
Addison-Weseley, 2006. –
ISBN 0-321-42672-X

[Melhem und Glozic 2003] Melhem, Wassim ; Glozic, Dejan:
PDE Does Plug-ins.
http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html.
September 2003

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

2. Auflage.
Springer, 2005. –
ISBN 3-540-65410-0

[OSGi-Alliance 2006] OSGi-Alliance:
OSGi Service Platform Core Specification Release 4, Version 4.0.1.
http://osgi.org/documents/osgi_technology/download/r4-specs/r4.core.pdf.
Juli 2006

[Steimann u. a. 2005] Steimann, Friedrich ; Keller, Daniela ; Aziz Safi, B.:
Moderne Programmiertechniken und -methoden.
Vorlesung 01853 an der Fern-Universität Hagen.
2005

[Steimann und Mayer 2005] Steimann, Friedrich ; Mayer, Philip:
Patterns of Interface-Based Programming.
In: *Journal of Object Technology*
Volume 4 (2005), Nr. 5, S. 75–94. –
ISSN 1660-1769

[Steimann u. a. 2006] Steimann, Friedrich ; Mayer, Philip ; Meissner, Andreas:
Decoupling classes with inferred interfaces.
In: *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, ACM Press, 2006, S. 1404–1408. –
ISBN 1-59593-108-2

[Steimann u. a. 2003] Steimann, Friedrich ; Siberski, Wolf ; Kühne, Thomas:
Towards the systematic use of interfaces in Java programming.
In: *PPPJ '03: Proceedings of 2nd International Conference on the Principles and Practice of Programming in Java*, Computer Science Press, 2003, S. 13–17. –
ISBN 0-9544145-1-9

[Tip u. a. 2003] Tip, Frank ; Kiezun, Adam ; Bäumer, Dirk:
Refactoring for generalization using type constraints.
In: *SIGPLAN Not.*
Volume 38 (2003), Nr. 11, S. 13–26. –
ISSN 0362-1340

Index

.properties, 34

Abstract-Syntax-Tree, 23, 48, 104
Adapter, 25, 40
Agile Prozessmodelle, 9, 14
Anforderungsanalyse, 21
AST-Knoten, 105
AST-Visitor, 23, 26
Audit, 98
AUTO_BUILD, 102

Benutzerkontinuitäts-Regel, 82
Benutzeroberfläche, 82
Bookmark-Marker, 107
Builder, 23, 28, 36, 37, 100
Builder, inkrementell, 23
Bundle, 77, 78, 83, 89, 90
Bundle-Activator, 34, 84
Bundle-ClassPath, 34
Bundle-Localization, 34, 84
Bundle-Manager, 97
Bundle-ManifestVersion, 33, 84
Bundle-Name, 34, 84
Bundle-RequiredExecutionEnvironment, 84
Bundle-SymbolicName, 34, 84
Bundle-Vendor, 34, 84
Bundle-Version, 34, 84

Classloader, 78
Classpath, 78, 89
Clean-Build, 102
Code-Audit, 98, 100
Committer, 76
Compatibility-Layer, 78
Compiler, 23, 98, 100, 106

Datenflussanalyse, 10
Debugger, 92
Delegate, 87
Dependency Injection, 12
Design-Pattern, 99, 104

Eclipse, 7, 11, 13, 80
Eclipse-BuddyPolicy, 85
Eclipse-ExtensibleAPI, 85
Eclipse-Foundation, 76
Eclipse-Konsortium, 76
Eclipse-LazyStart, 85
Eclipse-PlatformFilter, 85
Eclipse-RegisterBuddy, 85
Enabler, 81
Erweiterer, 81
Erweiterungsregel, 81
Explizite Erweiterung, 24, 82, 86
Export-Package, 34, 84
Extension, 25, 35, 50, 85, 86, 90, 101

Extension-Point, 24, 31, 35, 49, 51, 71, 85, 86, 90, 101, 107
Extension-Registry, 25, 52, 90
Extract Interface, 13, 16, 18

Factory, 47
Feature, 57, 59, 94
feature.xml, 57
Fragment, 93
FULL_BUILD, 47, 102

Generalize Declared Type, 15, 16, 19, 21, 25, 43, 61, 68, 70
Generalize Type, 13
Generic Workbench, 78

HTML, 53, 55, 59, 95

IBM, 76
IDE, 14, 77
IExtension, 90
IExtensionPoint, 90
IExtensionRegistry, 90
Import-Package, 84
INCREMENTAL_BUILD, 102
Infer Type, 16, 18, 21, 25, 43, 61, 68, 70
Installation, 57, 58, 95
IntelliJ IDEA, 13
Interface, 6, 8, 15, 16, 20, 24, 40, 86, 103
Interface, Allgemeinheit, 9
Interface, Popularität, 9
Interpreter, 98

JAR-Datei, 93
Java-Builder, 23, 102, 106
Java-JDK, 8, 19
Java-Nature, 102
Javadoc, 55
JDBC, 60
JHotDraw, 61
JUnit, 61
JVM, 92

Klasse, 87
Kompilationseinheit, 104
Komponente, 6
Komponentenmodell, 77
Konformitätsregel, 81

Lazy-Loading, 96
Lazy-Loading-Regel, 81
Lebenszyklus, 95
Lokalisierung, 52

Manifest, 33, 83, 89, 92, 101
MANIFEST.MF, 33, 83, 89
Marker-Resolution, 108
messages.properties, 52
Metrik, 9

Nature, 28, 36, 37, 102

Objektorientierung, 86
Open-Source, 76
org.eclipse.core.resources.builders, 38, 101
org.eclipse.core.resources.IncrementalProjectBuilder, 39
org.eclipse.core.resources.IProject, 48, 108
org.eclipse.core.resources.IProjectDescription, 38
org.eclipse.core.resources.IProjectNature, 37, 103
org.eclipse.core.resources.IResource, 108

org.eclipse.core.resources.IResourceDelta, 49
org.eclipse.core.resources.markers, 49, 107
org.eclipse.core.resources.natures, 103
org.eclipse.core.runtime, 89
org.eclipse.core.runtime.adaptor. EclipseStarter, 97
org.eclipse.core.runtime.IProgressMonitor, 39, 41, 47
org.eclipse.jdt.core.dom.ASTVisitor, 104
org.eclipse.jdt.core.dom.CompilationUnit, 48
org.eclipse.jdt.core.ICompilationUnit, 40, 49
org.eclipse.jdt.core.IJavaProject, 48
org.eclipse.jdt.internal.corext.refactoring. structure.ChangeTypeRefactoring, 44
org.eclipse.ui, 89
org.eclipse.ui.actionSets, 87
org.eclipse.ui.dialogs.PreferencePage, 36
org.eclipse.ui.dialogs.PropertyPage, 36
org.eclipse.ui.ide.markerResolution, 51, 109
org.eclipse.ui.IMarkerResolution, 110
org.eclipse.ui.IMarkerResolution2, 41, 51, 110
org.eclipse.ui.IMarkerResolutionGenerator2, 51, 109
org.eclipse.ui.propertyPages, 35
org.inoJ.inferType.actions.Abstract InferTypeAction, 47
org.inoJ.inferType.InferTypeRunnable, 47
org.inoJ.inferType.model.model.Single DeclarationElement, 46
org.intoJ.declaredTypeGeneralization Checker.core.IDeclaredType GeneralizationChecker, 42
OSGi, 34, 77, 78, 83, 85, 95

Package, 28
Parsen, 104
PDE, 93
Plattform.class, 89
Plug-In, 7, 22, 80, 83
Plug-In-Development-Environment, 91
plugin.properties, 34
plugin.xml, 34, 53, 83, 85, 92
Problem-Marker, 27, 49, 106, 127
Programmanalyse, 11
Property, 28

Quick-Fix, 22, 27, 41, 50, 108

Refactoring, 9, 11, 12, 14, 15, 22
Reflection, 26
Require-Bundle, 34, 84, 89
Rich-Client-Plattform, 76, 78, 94

Schema, 41, 87
Schichtenregel, 81
Schnittstelle, 6
Sichere Plattform-Regel, 81
site.xml, 58, 95
Software-Fäulnis, 14
Softwarearchitektur, 14, 22, 60
Stabilitätsregel, 82, 86, 109
Subtyp, 71
Supertyp, 9, 71
Swing, 60

Symbol, 104

Task-Marker, 107
Text-Marker, 50, 107
Tracing, 92
Typdeklaration, 23
Type Constraints, 10, 16
Typgeneralisierung, 15, 16, 99
Typinferenz, 9, 10, 62

UML, 24, 28
Unit-Test, 100
Update-Site, 57, 58, 95
Use Supertype Where Possible, 13, 16, 71

Veröffentlicher, 81
Visitor, 23, 48, 104, 119
Visual Age for Java, 76

Webserver, 58

XML, 54, 85

Glossar

A

Agile Prozessmodelle Im Gegensatz zu strukturierten und damit starren Vorgehensmodellen wird seit einigen Jahren alternativ mit agilen Prozessen gearbeitet. Im Mittelpunkt stehen der Programmierer und seine Belange. Bekanntestes Vorgehensmodell ist das Extreme Programming, siehe hierfür auch Steimann u. a. (2005).

Anforderungsanalyse Die Anforderungsanalyse ist Teil des Softwareentwicklungsprozesses und findet in einem frühen Stadium statt. Sie erfolgt im Dialog zwischen Auftraggeber und Entwickler mit dem Ziel, Qualität und Produktivität der Entwicklung zu steigern. Anforderungen werden nach der Erfassung strukturiert und bewertet. Weiterführende Informationen sind u.a. auf http://www.requirements-engineering.org zu finden.

B

Bundle Eine OSGi-konforme Softwarekomponente wird als Bundle bezeichnet. Ein Bundle beinhaltet neben dem Code und den weiteren Ressourcen, z.B. Grafiken, auch Meta-Daten, die das Bundle beschreiben und definieren, das Manifest.

D

Debugging Ein Debugger ermöglicht die Fehlersuche in Programmen, indem der Ablauf schrittweise auf Quelltextebene nachvollzogen werden kann. Unter anderem können Inhalte von Variablen zur Laufzeit betrachtet werden.

E

Eclipse Ein Framework für Rich-Client-Applikationen, das durch seine Plug-In-Infrastruktur erweitert werden kann. Die bekannteste Anwendung ist die Entwicklungsumgebung für Software, ursprünglich nur für Java-Programme, das Eclipse SDK. Eclipse wird als als Open-Source-Projekt weiterentwickelt.

J

Java Objektorientierte Programmiersprache, die von der Firma Sun Microsystems entwickelt wurde. Java-Programme werden zunächst in Bytecode übersetzt, der dann in einer Laufzeitumgebung ausgeführt wird.

M

Minimales Interface Ein Interface ist minimal, wenn es nur die Methoden enthält, die in dem Kontext benötigt werden, in dem das Interface benutzt wird. Werden Klassen in mehreren Kontexten benutzt, ist es sinnvoll, wenn sie auch mehrere minimale Interfaces implementieren.

O

Open-Source Als Open-Source bezeichnet man Software, deren Quelltext zugänglich und lesbar ist. Abhängig vom Lizenzmodell kann der Quelltext bzw. die Software gegebenenfalls sogar kopiert und weiterverbreitet oder geändert werden.

OSGi OSGi steht für Open Service Gateway Initiative und stammt ursprünglich aus dem Embedded-Software-Bereich. Angestrebt wird, einen Standard für modulare und flexible Plattformen zu schaffen, deren Komponenten zur Laufzeit ausgetauscht und hinzugefügt werden können. Die detaillierte Spezifikation ist bei OSGi-Alliance (2006) einzusehen.

P

Programmanalyse ‘Program analysis offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at run-time when executing a program on a computer.’[1] Hierfür kennt man die Ansätze der Datenflussanalyse, Constraint-basierte Analyse, Abstrakte Interpretation und Typ- und Effekt-Systeme.

R

Refactoring ‘Refactoring is the process of modifying a program’s source code without changing its behaviour, with the objective of improving the program’s design’[2], siehe hierfür auch Steimann u. a. (2005).

Reflection Unter Reflection versteht man in der Programmierung die Fähigkeit eines ablaufenden Programms, Informationen über sich selbst, seine Methoden, seine Klassen etc. zu bekommen. Reflection bedeutet aber auch, dass sich ein Programm selbst modifizieren kann. Die Form der Reflection, die in Java vorkommt und nur die Informationen bereitstellt, wird als Introspection bezeichnet. Für eine Einführung in dieses Thema sei auf Steimann u. a. (2005) verwiesen.

Rich-Client-Plattform Rich-Client-Plattformen ermöglichen es, Anwendungen zu entwickeln, die modular aufgebaut sind und erweitert werden können. Dabei übernimmt die Plattform alle Lifecycle- und Standarddienste, die eigentliche Anwendungslogik stammt aus den ladbaren Modulen. Somit können unterschiedlíchste Anwendungen auf einer Plattform entwickelt werden.

S

Softwarearchitektur Die Erstellung der Architektur einer Software ist eine der ersten Phasen der Entwicklung. Balzert (2000) beschreibt die Architektur als ‘eine strukturierte oder hierarchische Anordnung der Systemkomponenten

[1] aus: Nielson u. a. (2005)
[2] aus: Tip u. a. (2003)

sowie Beschreibung ihrer Beziehungen'. Für weitere Details sei auch auf Balzert (2000) verwiesen.

T

Tracing Mittel zur Fehlersuche durch definierte Ausgaben im Programm. Beispielsweise wird beim Einsprung in eine Funktion eine Meldung mit Zeitpunkt und übergebenen Parameterwerten in eine Datei oder auf eine Konsole geschrieben.

U

UML Die Unified Modelliung Language ist ein Versuch, für die Modellierung von Software eine strukturierte grafische Sprache als Standard zu etablieren. Mittels UML lassen sich nicht nur starre Strukturen von Klassengeflechten modellieren, sondern auch dynamische Anwendungsfälle, Informationsflüsse und Interaktionen von Komponenten.

X

XML Die Extensible Markup Language dient zur Dokumentauszeichnung. XML definiert eine generische Syntax, um beliebige Daten mit lesbaren Markup-Elementen, den Tags, auszuzeichnen; siehe auch Harold und Means (2005).

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Markus Bach

Unterschwaningen, den 7. Februar 2007