Masterarbeit

Access Modifier Modifier – Ein Werkzeug zur Einstellung der Sichtbarkeit in Java-Programmen

von

Eric Großkinsky

Lehrgebiet Programmiersysteme
Fakultät für Informatik und Mathematik
FernUniversität in Hagen

Betreuer: Prof. Dr. F. Steimann

2007

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Ein Aspekt der objektorientierten Programmierung ist das Verbergen von Programmteilen hinter einer Schnittstelle. Auf die Klasse kann nur über ihre Schnittstelle zugegriffen werden, was zum einen die Benutzung einer Klasse vereinfacht und zum anderen helfen kann, inkonsistente Objektzustände zu verhindern. Ein weiterer Vorteil des Geheimnisprinzips ist, dass die verborgenen Programmteile geändert werden können, ohne dass ein Benutzer der Klasse seinen Programmcode anpassen muss. Nur die Schnittstellen müssen gleich bleiben. Aus diesen Gründen sollte die Schnittstelle einer Klasse so minimal wie möglich sein.

Die Sichtbarkeit einer Klasse und ihrer Elemente wie Methoden und Feldern werden in Java-Programmen durch einen Zugriffsmodifikator festgelegt. Die Schnittstelle einer Klasse ergibt sich dann aus der Sichtbarkeit all ihrer Elemente. Um nun die Schnittstelle zu minimieren, muss die Sichtbarkeit der Elemente auf die minimal notwendige Sichtbarkeit reduziert werden.

Folgendes Szenario ist vorstellbar: Ein Programmierer hat, z. B. aus Bequemlichkeit, für jedes Element zuerst die geringste Einschränkung der Sichtbarkeit gewählt, da er so von jeder Stelle darauf zugreifen kann. Er möchte nun die Schnittstellen seiner selbst erstellten Klassen minimieren. Er kann die Sichtbarkeit von Elementen dabei problemlos variieren, indem er einfach, ohne jede Werkzeugunterstützung, den Zugriffsmodifikator ändert. Allerdings muss er, damit die Sichtbarkeitsänderung zu keinem Übersetzungsfehler führt, alle Stellen im Programm berücksichtigen, an denen das Element benutzt wird. Wenn er die Bedingungen für jede Methode einer Klasse oder gar aller Klassen eines Projekts einzeln prüfen muss, dann ist das sehr mühsam und zeitaufwändig. Was durchaus schwerer wiegt als der bloße Zeitaufwand, ist die Tatsache, dass er sich unter Umständen nicht im Klaren darüber ist, dass eine einfache Modifikation der Sichtbarkeit auch die Semantik des Programms modifizieren kann.

Ein Ziel dieser Arbeit ist zu zeigen, wann die Sichtbarkeitsänderungen zulässig sind oder aber zu einer Modifikation der Semantik führen bzw. einen Übersetzungsfehler verursachen.

Um den Programmierer bei der Einschränkung der Schnittstelle zu unterstützen, soll deshalb der *Access-Modifier-Modifier* (*AMM*) entwickelt werden. Dieses Werkzeug soll ihm die zeitraubende Arbeit der Sichtbarkeitseinschränkung abnehmen und dabei unerwünschte Semantikänderungen verhindern.

Der Programmierer soll dabei das Werkzeug aus einer Entwicklungsumgebung aufrufen und bedienen können. Der *AMM* wird deshalb als eine Erweiterung der Programmierumgebung „Eclipse“ konzipiert. Die IDE Eclipse wurde gewählt, da sie sehr verbreitet ist und so der Kreis der potentiellen Benutzer steigt. Die Anwendung bietet einen sehr großen Funktionsumfang, ist sehr stabil, gut zu bedienen und als Open-Source-Projekt frei erhältlich. Zum Erfolg von Eclipse hat nicht minder beigetragen, dass es vielfältige Möglichkeiten bietet, es nach eigenen Wünschen und Bedürfnissen zu erweitern. Tatsächlich gibt es eine fast unüberschaubare Vielfalt an kommerziellen und frei erhältlichen Erweiterungen, den so genannten Plugins. Bei der Entwicklung von eigenen Refactoring-Werkzeugen für Java-Programme können die Fähigkeiten von Eclipse verwendet werden. Der Einsatz der Eclipse-Funktionalität kann den Entwicklungsaufwand beträchtlich verringern oder die Entwicklung, wie im Fall des *Access-Modifier-Modifier-Plugins*, sogar erst ermöglichen. So müssen Refactoring-Werkzeuge verschiedene Bedingungen prüfen, z. B. ob eine Methode eine andere Methode überschreibt. Mit Eclipse lässt sich der zu untersuchende Programmcode, der ja nur als Text vorliegt, in einen so genannten Syntaxbaum überführen, mit dem sich dann diese Bedingungen prüfen lassen.

1.2 Aufbau der Arbeit

Im folgenden Kapitel werden die Anforderungen an den *Access-Modifier-Modifier* weiter spezifiziert und einige Grundlagen geklärt. Danach werden in Kapitel 3 Eclipse kurz vorgestellt und einige wichtige Aspekte der Plugin-Entwicklung besprochen, die für die Entwicklung des *AMM-Plugins* relevant waren.

In Kapitel 4 werden Ansätze diskutiert, mit deren Hilfe geprüft werden kann, unter welchen Bedingungen ein Java-Element welche Sichtbarkeit annehmen darf. Ferner wird untersucht, wie die Sichtbarkeitsänderungen annotiert werden können, so dass z. B. auch Programmierteams, die an einer gemeinsamen Codebasis arbeiten, mit dem *AMM-Plugin* arbeiten können.

Die Implementierungsdetails des *AMM-Plugins* werden schließlich in Kapitel 5 besprochen. Einen Überblick über den Aufbau des Plugins liefert zunächst ein Klassendiagramm. Danach werden der Programmablauf und die Einzelheiten der wichtigsten Klassen erläutert.

In Kapitel 6 wird die Bedienung des *AMM-Plugins* aus Anwendersicht beschrieben, wobei auch auf Installationsdetails eingegangen wird. Die Schlussbetrachtungen in Kapitel 7 fassen die Ergebnisse zusammen und geben einen kurzen Ausblick auf mögliche weitere Untersuchungen des Themas.

2 Problemstellung

2.1 Überblick

Zuerst wird die Sichtbarkeit von Elementen in Java-Programmen genauer betrachtet und die Bedeutung der Begriffe „Öffnen“ und „Schließen“ erläutert. Danach werden einige Szenarien besprochen, die eine Werkzeugunterstützung notwendig bzw. wünschenswert machen. Aus diesen Szenarien werden schließlich die allgemeinen Anforderungen abgeleitet, die der *Access-Modifier-Modifier* erfüllen muss.

2.2 Zugriffskontrolle in Java

In Java gibt es die Zugriffsmodifikatoren *private*, *protected* und *public*. Wenn ein Element keinen Zugriffsmodifikator aufweist, hat es die Sichtbarkeit „default access“. Verbreitet ist auch die Bezeichnung „package access“ oder „package scope“, da die Sichtbarkeit auf ein Paket eingeschränkt wird [Ecke04] [Krüg00]. Im folgenden Text wird die Sichtbarkeit „default access“ zum Teil vereinfacht als *default* bezeichnet, wenn keine Missverständnisse auftreten können.

Wenn ein Benutzer direkten Zugriff auf ein Feld eines Objektes hat, kann er es mit einem Wert belegen, der nicht vorgesehen war. Objekte sollten deshalb selbst Kontrolle über ihre Daten erhalten. Dies wird erreicht, indem die Sichtbarkeit der Felder auf *private* gesetzt wird und auf die Felder dann nur indirekt über Methoden zugegriffen wird. (Diese Vorgehensweise ist in Java allerdings nicht zwingend vorgeschrieben, ist aber guter Programmierstil.)

Von welchen Programmstellen aus auf Elemente mit einer bestimmten Sichtbarkeit zugegriffen werden kann, ist detailliert in §6.6 der Java-Spezifikation [Gosl05] aufgeführt.

Stark vereinfacht können die Sichtbarkeiten folgendermaßen angegeben werden (nach [Krüg00]):

public – Die Methoden und Felder sind in der Klasse selbst, in Subklassen und für Aufrufer von Instanzen der Klasse sichtbar.

protected – Die Methoden und Felder sind in der Klasse selbst und in Subklassen sichtbar, sowie in Klassen, die sich im gleichen Paket wie die definierende Klasse befinden.

default – Die Methoden und Felder sind nur innerhalb des Paketes sichtbar, in dem sich die Klasse befindet.

private – Die Methoden und Felder sind nur in der Klasse selbst sichtbar.

Die Sichtbarkeit von Elementen kann also abgestuft werden. Unzweifelhaft ist *private* die geringste und *public* die größte Sichtbarkeit, die ein Element annehmen kann. Bei den Modifikatoren *protected* und *default* muss jedoch die Rangfolge genauer untersucht werden. Auf Elemente mit Sichtbarkeit *default* kann von Programmstellen zugegriffen werden, die sich innerhalb des gleichen Paketes befinden. Das gleiche gilt auch für Elemente mit Sichtbarkeit *protected*. Allerdings kann auf diese Elemente zusätzlich von allen Subklassen zugegriffen werden, auch wenn sich diese in einem anderen Paket befinden. *Protected* vergrößert also die Sichtbarkeit *default*. Damit lautet die Sichtbarkeitshierarchie[1], von der kleinsten Sichtbarkeit aufsteigend: *private*, *default*, *protected*, *public*.

2.3 Definition „Öffnen“ und „Schließen“ einer Klasse

Mit dem Ausdruck „Schließen einer Klasse“ ist folgendes gemeint: Die Sichtbarkeit jeder Methode einer Klasse wird auf die jeweils minimal mögliche Sichtbarkeit reduziert. Nach dem Schließen hat die Klasse ihre minimale Schnittstelle erreicht. Wenn eine Klasse innere Klassen enthält, werden die Methoden der inneren Klassen ebenfalls geschlossen. Die Felder einer Klasse werden im Rahmen dieser Arbeit nicht berücksichtigt.

Es wird im Folgenden angenommen, dass die Sichtbarkeit der Felder *private* beträgt und auf die Felder außerhalb der sie definierenden Klasse nur über Methoden zugegriffen

[1] Die Hierarchie gilt auch für die genauen Bedingungen nach §6.6 der Java-Spezifikation.

wird, z. B. über Getter- und Setter-Methoden[1]. Wird dies eingehalten, wird mit einer Sichtbarkeitsreduktion der Methoden auch der Zugriff auf die Felder eingeschränkt.

Beim „Öffnen einer Klasse“ ist es genau umgekehrt: Die Sichtbarkeit der Methoden einer Klasse wird auf die jeweils maximal zulässige Sichtbarkeit erhöht. Alternativ kann die Sichtbarkeit nur bis zu einer festzulegenden Sichtbarkeit erhöht werden. (Methoden mit einer höheren Sichtbarkeit werden hierbei aber nicht reduziert.) Innere Klassen einer Klasse werden ebenfalls geöffnet.

Schließen und Öffnen werden im folgenden Text auch mit den englischen Begriffen „Lock“ und „Unlock“ bezeichnet.

2.4 Anforderung an den *Access-Modifier-Modifier*

Allgemeine Anforderungen

Der *Access-Modifier-Modifier* soll Entwickler beim Schließen bzw. Öffnen einer Klasse unterstützen und sicherstellen, dass sich dadurch die Semantik des Programms nicht ändert. Weil er die Bedürfnisse von verschiedenen Benutzerkreisen erfüllen soll, müssen umfangreiche Einstellungen möglich sein. Mögliche Einsatzgebiete, aus denen sich weitere Anforderungen ergeben, werden im Folgenden diskutiert. Dabei wird auch untersucht, wann das Öffnen und Schließen einer Klasse sinnvoll ist und was dabei beachtet werden muss.

Software-Engineering-Methoden

Die meisten Vorgehensmodelle des Software Engineerings sehen eine Entwurfsphase vor, in der die Klassen entworfen und die Sichtbarkeit der Elemente festgelegt werden [Wint05]. Der Entwickler überlegt sich hierbei, von welchen anderen Klassen eine Methode aufgerufen oder überschrieben werden muss, in welchen Paketen sich die Klassen befinden und bestimmt daraufhin die Sichtbarkeit des Elements.

---

[1] Wenn auf die Felder direkt zugegriffen wird, können mit dem in Eclipse vorhandenen Refactoringwerkzeug „Encapsulate Field“ die Getter- und Setter-Methoden automatisch erzeugt und die Feldzugriffe durch Methodenaufrufe ersetzt werden. Die Sichtbarkeit der Felder wird dabei automatisch auf die Sichtbarkeit *private* gesetzt.

In der Methode des „Extreme Programming“ [Beck01] gibt es dagegen eine solche Entwurfsphase überhaupt nicht. Wenn ein Programmierer feststellt, dass eine Klasse um eine neue Funktionalität erweitert werden muss, dann führt er z. B. einfach neue Methoden ein. Er weiß jedoch unter Umständen noch nicht genau, von wo aus auf die Methode zugegriffen werden muss. Deshalb wird er die Sichtbarkeit der Methode aus Bequemlichkeit oftmals auf die höchstmögliche Sichtbarkeit setzen, so dass er von möglichst vielen Programmstellen auf sie zugreifen kann. Am Ende des Projekts sollen die Schnittstellen der Klassen aber möglichst minimal werden.

Allerdings werden auch bei den klassischen Modellen in der Entwurfsphase oftmals nicht alle Abhängigkeiten berücksichtigt, so dass die dort festgelegten Sichtbarkeiten geändert werden müssen. Außerdem ist der Entwurf ständigen Modifikationen ausgesetzt: Klassen werden hinzugefügt, in andere Pakete verschoben, neue Funktionalität wird hinzugefügt und alte entfernt.

Wenn eine Methode nun an einer Stelle aufgerufen werden soll, an der sie nicht sichtbar ist, muss der Programmierer erst umständlich zu der jeweiligen Klasse springen und die Sichtbarkeit von Hand anpassen. Zudem zeigt Eclipse bei der automatischen Codevervollständigung nur Elemente an, die auch an dieser Stelle sichtbar sind, so dass ein Benutzer, um von der Existenz einer unsichtbaren Methode zu erfahren, im Programmcode nachschauen muss.

Um dem Entwickler die Arbeit zu erleichtern, soll es deshalb möglich sein, eine Klasse bis zu einer bestimmten Sichtbarkeit zu öffnen. Er kann dann die Methoden der Klasse benutzen, ohne erst die oben geschilderten Umwege gehen zu müssen. Am Ende, oder wenn auf die Klasse nicht mehr zugegriffen werden muss, werden die Öffnungen, falls dies möglich ist, wieder rückgängig gemacht.

Es ist vorstellbar, dass der Entwickler nicht für jede Methode die minimal mögliche Sichtbarkeit wünscht, sondern dass nur die Öffnungen rückgängig gemacht werden sollen. Deshalb soll bei der Öffnung für jede Methode festgehalten werden, welche Sichtbarkeit sie vor der Öffnung hatte. Diese Informationen dienen dem Entwickler und auch dem *AMM* als Erinnerung daran, welche Methode am Ende wieder geschlossen werden sollte. Es soll im *AMM-Plugin* festgelegt werden können, dass bei Schließung der Klasse die Sichtbarkeit jeder Methode nur bis zu der Sichtbarkeit reduziert wird, die sie vor der

Öffnung aufwies. Mit dieser Einstellung kann verhindert werden, dass sich die ursprünglichen Schnittstellen durch den Einsatz des Werkzeugs verringern.

Framework und Bibliothek

Bei der Entwicklung eines Frameworks oder einer Bibliothek kann es erforderlich sein, dass die öffentliche Schnittstelle einer Klasse erhalten bleibt, damit ein Benutzer die für ihn wichtigen Funktionen überhaupt nutzen kann. Um mit einem Framework arbeiten zu können, müssen zudem oftmals von den im Framework vorhandenen Klassen neue Klassen abgeleitet werden. Bei Bibliotheken sollen Klassen um neue Funktionen erweitern werden können. Die neuen Subklassen müssen dazu auf bestimmte Methoden der Superklassen zugreifen oder sie überschreiben können.

Der Benutzer eines Frameworks bzw. einer Bibliothek arbeitet in der Regel mit den kompilierten Class-Dateien, so dass er die Sichtbarkeit von Elementen nicht ändern kann. Deshalb muss der Entwickler des Frameworks bzw. der Bibliothek dafür sorgen, dass eine Subklasse auf die benötigten Methoden zugreifen kann.

Damit beim Schließen einer Klasse die Sichtbarkeit dieser Methoden nicht geringer als die notwendige Sichtbarkeit wird, muss für jede Methode eine minimal zulässige Sichtbarkeit angegeben werden können. Der *AMM* berücksichtigt dann diese Angaben beim Schließen. Um eine minimale Sichtbarkeit einer Methode zu notieren, bieten sich z. B. Annotationen an.

In einer Klasse kann es Methoden geben, die nur dafür konzipiert wurden, innerhalb der eigenen Klasse aufgerufen zu werden. Es muss nun verhindert werden, dass diese Methoden beim Öffnen einer Klasse mitgeöffnet werden. Deshalb muss für jede Methode zusätzlich eine maximal zulässige Sichtbarkeit angegeben werden können.

Mehrere Benutzer

An größeren Programmierprojekten arbeiten zumeist mehrere Entwickler, die eine gemeinsame Codebasis benutzen. Es ist hier möglich, dass eine Klasse mehrmals nacheinander von verschiedenen Entwicklern geöffnet wird. Deshalb ist es wichtig, die Öffnungshierarchie, also wie die Methode geöffnet wurde, nachvollziehen zu können. Bei einer Öffnung sollte deshalb für jede Methode, zusätzlich zu der Sichtbarkeit die sie vor der Öffnung hatte, notiert werden, welcher Entwickler die Methode geöffnet hat. Beim

Schließen sollte dann im Werkzeug eingestellt werden können, dass nur die Methoden geschlossen werden, die ein bestimmter Entwickler geöffnet hat.

Programmier-Projekte

Damit ein Benutzer nicht jede Klasse einzeln öffnen oder schließen muss, ist es hilfreich, ein ganzes Eclipse-Projekt öffnen bzw. schließen zu können. Es muss dabei untersucht werden, ob die Reihenfolge, in der die Klassen geschlossen werden, von Bedeutung ist.

Hinweise mit Markern

In Eclipse können Entwickler mit Hilfe von Markern auf Probleme im Programmcode aufmerksam gemacht werden. Wenn bei der Übersetzung des Programmcodes z. B. eine Compiler-Warnung auftritt, versieht Eclipse die verursachende Stelle mit einem Marker, der den Grund der Warnung angibt (siehe Bild 2.1). Es bietet zudem eine automatische Lösung, „QuickFix“ genannt, an, mit dem das Problem behoben werden kann.

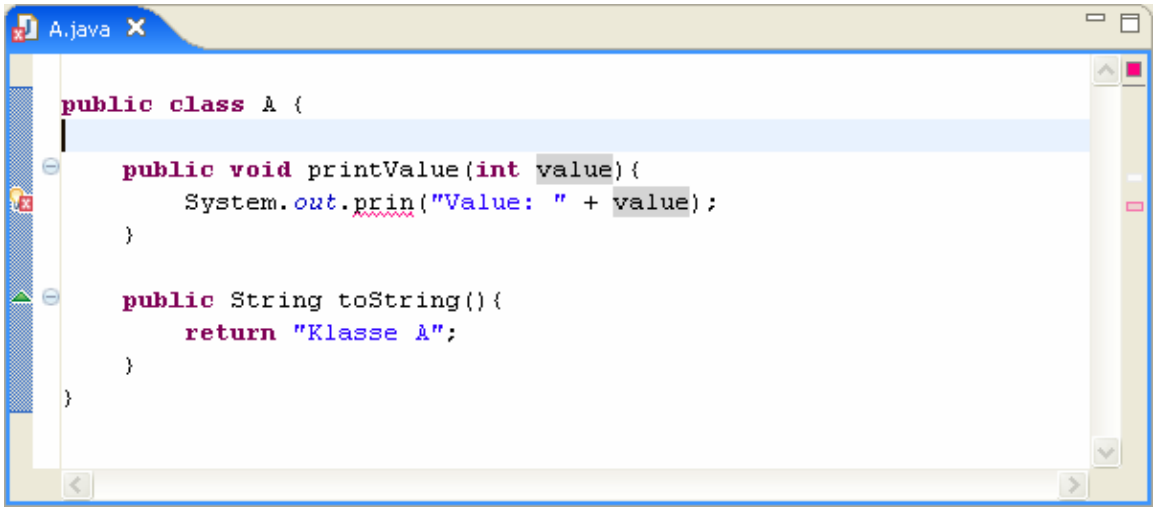

Bild 2.1: Die Piktogramme auf linken Leiste sind Marker: Der obere Marker zeigt einen Compilerfehler an, der untere, dass die Methode `toString()` eine überschreibende Methode ist.

Der *Access-Modifier-Modifier* soll dem Entwickler für den Fall, dass die Sichtbarkeit einer Methode eingeschränkt werden kann, etwas Gleichartiges zur Verfügung stellen. Folgendes soll also möglich sein: Wenn die Sichtbarkeit einer Methode reduziert werden kann, wird die Methode mit einem Marker annotiert. Dieser Marker zeigt die minimal mögliche Sichtbarkeit an und bietet einen QuickFix an, um die Sichtbarkeit auf die minimale Sichtbarkeit zu ändern und die notierte Öffnungshierarchie entsprechend anzupas-

sen. Sollte sich der Programmtext ändern, dann erfolgt eine erneute Prüfung und die Marker werden entsprechend angepasst.

3 Eclipse

3.1 Einführung

Vielen Benutzern ist Eclipse [Ecli07] in erster Linie als Integrated Development Tool (IDE) zur Entwicklung von Java-Programmen bekannt. Es ist aber weitaus mehr als nur ein IDE, es ist zudem ein Open-Source-Framework zur Entwicklung von Desktop-Applikationen. Eclipse dient dabei als Plattform, die gewisse Grundfunktionen zur Verfügung stellt und die fast beliebig erweitert werden kann. Eclipse weist so genannte Erweiterungspunkte auf, an denen sich die Erweiterungen, Plugins genannt, einklinken und so neue Funktionen zur Verfügung stellen können. Auch Plugins selbst können neue Erweiterungspunkte definieren, an denen sich wiederum andere Plugins einklinken können. Bild 3.1 zeigt den Aufbau der Eclipse-Plattform. Das Java-Development-Tooling (JDT) erweitert die Eclipse-Plattform um die Fähigkeit, Java-Programme zu erstellen. Wie im Bild zu erkennen ist, ist es selbst nur als ein Plugin konzipiert. Das JDT besteht aus einem Teil, der das User-Interface erweitert, und einem Teil, mit dem Javacode analysiert, manipuliert und kompiliert werden kann.

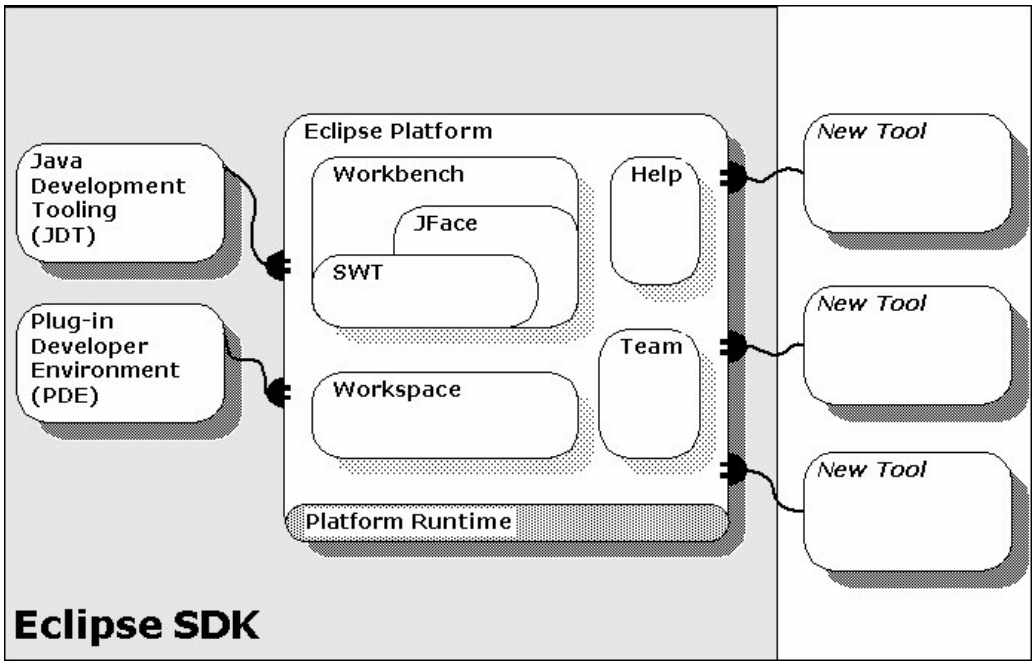

Bild 3.1: Aufbau der Eclipse-Plattform (aus [EHil06])

Da Eclipse hauptsächlich in Java geschrieben ist, erfolgt auch die Implementierung eines Plugins in Java. Um sich in die Erweiterungspunkte einklinken zu können, müssen meist

bestimmte Interfaces implementiert werden. Jedes Plugin benötigt eine Manifest-Datei, in der unter anderem festgelegt wird, in welche Erweiterungspunkte es sich einklinkt, welche Erweiterungspunkte es anderen zur Verfügung stellt und in welchem Archiv die Implementierung zu finden ist. [Bäum04] [Danj04]

3.2 Details der Plugin-Entwicklung

3.2.1 Literatur

Diese Arbeit erhebt keinen Anspruch, einen umfassenden Überblick über Eclipse oder die Entwicklung von Plugins mit Eclipse zu geben. Es wird nur auf einige Aspekte der Plugin-Entwicklung eingegangen, die für die Implementierung des *Access-Modifier-Modifier-Plugins* von besonderer Bedeutung waren. Gute Einführungen in die allgemeine Plugin-Entwicklung finden sich in [Danj04], [Bäum04], [Hint05] und [Clay04].

Aus der allgemeinen Eclipse-Plattform werden nachfolgend die Konzepte des Markers und des QuickFixes, aus dem JDT die des abstrakten Syntaxbaumes sowie der Suche nach Java-Elementen erläutert. Alle Erläuterungen beziehen sich auf die Version 3.2 von Eclipse.

3.2.2 Marker

Mit Hilfe von Markern können Stellen im Programmtext markiert werden. Wenn bei der Übersetzung des Programmcodes z. B. eine Compiler-Warnung auftritt, versieht das JDT von Eclipse die verursachende Stelle mit einem Marker, der den Grund der Warnung angibt. Eine Methode, die eine andere Methode überschreibt oder implementiert, wird im Quellcode-Editor mit einem Marker annotiert, der die Form eines grünen Dreiecks hat. Die Marker werden als Metadaten in einem gesonderten Bereich von Eclipse gespeichert.

Es gibt verschiedene Arten von Markern. Neue Marker können definiert werden, indem sie von vorhandenen Markern abgeleitet werden. Ein Marker kann auch von mehreren Markern abgeleitet werden. Dies ist möglich, da die Markertypen keine Java-Typen sind. Jeder Marker wird von einem Objekt des Typs `IMarker` repräsentiert. Ein Marker kann nicht nur im Editor sichtbar sein. So werden alle Marker, die vom Typ `org.ecl-`

ipse.core.resources.problemmarker abgeleitet wurden, in der Problems-View, einem Fenster in Eclipse, aufgelistet.

Jeder Marker hat bestimmte Attribute, z. B. besitzt ein Marker vom Typ org.eclipse.core.resources.textmarker die Attribute charStart, charEnd und lineNumber. Diese Attribute bestimmen die Stelle im Editor, an denen der Marker erscheinen soll und die Stellen an denen der Text unterstrichen werden soll. Wird im Programmtext eine neue Zeile oberhalb der Markierungsstelle eingefügt, werden diese Attributwerte automatisch angepasst. Abgeleitete Typen können zusätzliche Attribute definieren. Mit folgender Methode, kann ein Marker vom Typ textmarker erzeugt und einer Compilation-Unit hinzugefügt werden [Clay04].

```
public void markUnit(ICompilationUnit unit, String message, int charStart,
                     int charEnd) throws CoreException {
     //Erzeugen eines Markers vom Typ textmarker und
      //hinzufügen zu einer Ressource.
     IMarker marker = unit.getCorrespondingResource()
                      .createMarker("org.eclipse.core.resources.textmarker");
     //Attribute festlegen
     Map attributes = new HashMap(3);
     attributes.put(IMarker.MESSAGE, message);
     attributes.put(IMarker.CHAR_START, charStart);
     attributes.put(IMarker.CHAR_END, charEnd);

     marker.setAttributes(attributes);
}
```

3.2.3 QuickFix

Marker können dazu verwendet werden, Probleme im Programmtext anzuzeigen. Wenn der Mauszeiger über einen solchen Marker platziert wird, werden Informationen über die Art des Problems angezeigt. Wenn Eclipse oder ein Plugin für dieses Problem eine automatische Lösung, auch QuickFix genannt, anbietet, kann sich der Benutzer mit einem Klick auf den Marker die verfügbaren Lösungsvorschläge anzeigen lassen. Er kann einen Vorschlag auswählen und das Problem wird daraufhin automatisch behoben.

Ein Plugin kann eigene Lösungsvorschläge für einen Markertyp anbieten. Dafür muss es sich in den Erweiterungspunkt vom Typ org.eclipse.ui.ide.markerResolution

einklinken. Im Erweiterungspunkt wird angegeben, für welchen Markertyp Lösungen bereitgestellt werden können. Zusätzlich muss ein Generator implementiert werden, der entscheidet, ob für ein konkretes Problem eine Lösung angeboten werden kann. Der Generator muss dafür das Interface `IMarkerResolutionGenerator2` implementieren. Die Methode `hasResolutions` des Generators muss `true` zurückliefern, wenn er Lösungen anzubieten hat. Der Aufruf der Methode `getResolutions` liefert dann alle verfügbaren Lösungen zurück. Die Lösungen sind vom Typ `IMarkerResolution`. Um eine eigene Lösung anbieten zu können, muss dieses Interface implementiert werden. Eine Instanz dieses Typs enthält Information für den Benutzer, z. B. was genau am Programmtext geändert wird. Die implementierte `run`-Methode wird von Eclipse ausgeführt, wenn der Benutzer diesen Lösungsvorschlag ausgewählt hat.

3.2.4 AST

Um den Programmcode zu untersuchen gibt es in Eclipse mehrere Möglichkeiten. Die detailliertesten Informationen können mit dem Abstract Syntax Tree (AST) gewonnen werden. Der Programmcode wird dazu geparst und in eine Baumstruktur überführt, die aus Elementknoten besteht. Für jedes Element im Code gibt es einen entsprechenden Knoten. So gibt es Elementknoten für Methodendeklarationen, Zuweisungen oder Variabelnamen. Es gibt über 60 verschiedene Knotentypen, die alle Subtypen von `ASTNode` sind. Der AST einer Compilation-Unit mit 500 Zeilen kann um die zweitausend Knoten aufweisen.

Erzeugung des ASTs

Der Wurzelknoten eines ASTs ist vom Typ `CompilationUnit`[1]. Er enthält für jeden in der Compilation-Unit definierten Typ einen direkten Unterknoten, sowie Knoten für die Deklarierung eines Paketes und den Import von Typen. Ein Typknoten enthält wiederum Knoten für Methoden, Felder usw.

[1] Die Klasse `CompilationUnit` ist kein Subtyp von `ICompilationUnit`.

Einen AST kann man aus einer `ICompilationUnit` mit folgender Methode erzeugen:

```
public CompilationUnit createAST(ICompilationUnit unit,
                                 boolean resolveBindings) {
     ASTParser parser = ASTParser.newParser(AST.JLS3);
     parser.setKind(ASTParser.K_COMPILATION_UNIT);
     parser.setSource(unit);
     parser.setResolveBindings(resolveBindings);

     return (CompilationUnit) parser.createAST(/*ProgressMonitor*/null);
}
```

Ein `ASTParser` kann ASTs für verschiedene Java-Spezifikationen erzeugen. Mit der Angabe `AST.JLS3` wird ein AST gemäß der dritten Version der Java-Spezifikation, in der alle Neuerungen enthalten sind, die in Java 5 eingeführt wurden, erstellt. Bei der Erzeugung eines ASTs muss festgelegt werden, ob die sogenannten Bindungen mit aufgebaut werden sollen. Werden sie bei der Erzeugung nicht aufgebaut, kann später auch nicht auf sie zugegriffen werden. Die Bindungen enthalten zusätzliche Informationen über bestimmte Java-Elemente, wie Methoden, Typen und Variabeln.

Annotationen und Zugriffsmodifikatoren einer Methode im AST

Eine Methodendeklaration wird im AST durch den Typ `MethodDeclaration` repräsentiert. Die Signatur einer Methode kann zusätzlich zu den Modifikatoren wie *public*, *static* oder *final*, Annotationen aufweisen. Im AST sind die Knotentypen für die Modifikatoren und die Knotentypen für die Annotationen vom Typ `IExtendedModifier` abgeleitet. (In früheren Versionen des JDTs wurden die Modifikatoren durch einen Wert vom Typ `int` repräsentiert [Hint05]. In der neuen Version sind die Modifikatoren eigene AST-Knoten vom Typ `Modifier`.) Die Methode `MethodDeclaration#modifiers` liefert eine Liste zurück, die alle Modifikatoren und Annotationen enthält.

Ein Knoten, der eine Annotation repräsentiert, kann vom Typ `MarkerAnnotation`, `NormalAnnotation` oder `SingleMemberAnnotation` sein. Eine `MarkerAnnotation` zeichnet sich dadurch aus, dass sie keine Ausdrücke enthält. Als Beispiel sei hier die Annotation `Override` genannt. Eine `SingleMemberAnnotation` ist eine Annotation, die genau einen Ausdruck enthält. Alle anderen Annotationen werden durch einen Knoten vom Typ `NormalAnnotation` repräsentiert.

Bei der Auswertung einer Annotation ist zu beachten, dass eine Marker-Annotation nicht immer ein Knoten des Typs `MarkerAnnotation` ist. Die Marker-Annotation `Override` kann im Programmtext als `@Override` oder als `@Override()` auftreten. Beide Ausdrücke sind semantisch äquivalent. Allerdings wird `@Override()` von einem Knoten des Typs `NormalAnnotation` repräsentiert und `@Override` von einem Knoten vom Typ `MarkerAnnotation`. Zwischen den beiden Typen besteht keine Subtypbeziehung.

Erzeugung neuer Knoten

Oftmals ist es notwendig, neue Knoten zu erzeugen und in einen vorhandenen AST einzufügen. Um neue Knoten zu erzeugen, müssen die Factory-Methoden der Klasse `org.eclipse.jdt.core.AST` benutzt werden. (Der AST kann so auch von Grund auf neu aufgestellt werden, also ohne vorhandenen Programmcode zu parsen.) Da der Konstruktor eines Knotentyps die Sichtbarkeit *default-access* besitzt, kann ein Knoten nicht direkt erzeugt werden. Für jeden Knotentyp gibt es eine entsprechende Factory-Methode. Jeder AST besitzt eine eigene Instanz des Typs `AST`, die von jedem Knoten durch Aufruf der Methode `getAST` erhalten werden kann.

Knoten, die durch ein `AST`-Objekt eines ASTs neu erzeugt wurden oder vorhandene Knoten eines ASTs können nicht direkt in einen anderen AST eingefügt werden. Sie müssen vorher kopiert werden, wobei es auch möglich ist, ganze Subbäume zu kopieren. Mit der Methode `ASTNode.copySubtree(AST target, ASTNode node)` kann ein Knoten kopiert werden. Die Methode liefert eine tiefe Kopie des Subbaumes zurück, dessen Wurzel der übergebene AST-Knoten darstellt.

Änderung des ASTs

Der Programmcode kann auf verschiedene Arten geändert werden. Ein nahe liegendes Verfahren ist, den Programmcode als Text zu manipulieren. Dies ist ein Verfahren, welches auch das Refactoring-Framework von Eclipse anwendet, da es auch mit Java-Code funktioniert, der sich nicht in einer Compilation-Unit befindet, sondern z. B. in einer „Java Server Page“ (JSP) [Bäum07].

Ein alternatives Verfahren ist, den Programmcode über die Manipulation des aufgestellten ASTs zu ändern. Der AST wird hier geändert, indem Knoten neu hinzugefügt, verschoben oder gelöscht werden. Nachdem der AST geändert wurde, kann der dem AST

entsprechende Programmcode automatisch erzeugt werden. Bei der Erzeugung des Codes werden zuvor definierte Formatierungsregeln berücksichtigt. Die Änderung über den AST ist der einfachen Textmanipulation vorzuziehen. Denn meistens hängen die Änderungen von anderen Programmstellen oder Bedingungen ab, die mit dem AST besser berücksichtigt werden können. [Danj04] [Kuhn06]

Es gibt zwei Verfahren, wie sich Modifikationen an einem AST im Programmcode wieder finden. Das erste Verfahren arbeitet mit einem Objekt vom Typ `ASTRewrite`, der die Änderungen am AST protokolliert, der AST selbst wird dabei nicht verändert. Mit dem zweiten Verfahren kann der AST direkt manipuliert werden. (Es benutzt allerdings intern auch einen `ASTRewrite`.) Das erste Verfahren ist flexibler, da es z. B. bei einem Refactoring erlaubt, mit einem einzigen AST verschiedene Manipulationen zu untersuchen.

Folgendes Beispiel zeigt, wie die Sichtbarkeit einer Methode geändert werden kann. Weist eine Methode einen Zugriffsmodifikator vom Knotentyp `Modifier` auf, kann er mit der Methode `makePublic(Modifier)` in *public* geändert werden. Dazu wird zuerst eine Instanz eines `ASTRewrite`s erzeugt. Danach werden ihr die zu ändernde Eigenschaft und der neue Wert der Eigenschaft übergeben. Wenn mehrere Änderungen hintereinander durchgeführt werden sollen, muss immer die gleiche Instanz verwendet werden.

```
public void makePublic(Modifier modifier) {
     ASTRewrite rewriter = ASTRewrite.create(modifier.getAST());
     rewriter.set(modifier, Modifier.KEYWORD_PROPERTY,
                  ModifierKeyword.PUBLIC_KEYWORD, null);
}
```

Wenn die Sichtbarkeit der Methode *default-access* ist, ist kein Knoten vom Typ `Modifier` vorhanden, der geändert werden kann. Stattdessen müssen, wie im folgenden Programmtext gezeigt, ein Modifier-Knoten erzeugt und die Einfügung in den AST protokolliert werden. Wie schon erwähnt, wird der AST dabei nicht geändert. Mit einem `ListRewrite` kann der Knoten zu der übergebenen `MethodDeclaration` hinzugefügt werden. Ein `ListRewrite` wird immer dann benutzt, wenn die Eigenschaft eine Liste von Knoten ist.

```
public void makePublic(MethodDeclaration methodDecl, ASTRewrite rewriter){
     //Neuen Modifierknoten erzeugen
     Modifier newModifier = methodDecl.getAST()
                            .newModifier(ModifierKeyword.PUBLIC_KEYWORD);
     ListRewrite lrw = rewriter.getListRewrite(methodDecl,
                                       MethodDeclaration.MODIFIERS2_PROPERTY);
     //Einfügung des Knotens protokollieren
     lrw.insertLast(newModifier, null);
}
```

Suche im AST

Nun sollen bestimmte Informationen in einem AST gefunden werden, z. B. soll die Anzahl der Methoden in einer Compilation-Unit gefunden werden. Der Syntaxbaum kann am leichtesten mit Hilfe des Visitor-Patterns [Gamm95] durchsucht werden. Hierzu wird ein eigener Visitor von der Klasse `ASTVisitor` abgeleitet. Die Klasse `ASTVisitor` besitzt für jeden Knotentyp eine visit-Methode. Wird nach einem bestimmten Knotentyp gesucht, muss die entsprechende visit-Methode überschrieben werden. Wenn auch die Subknoten der gefunden Knoten untersucht werden sollen, muss die visit-Methode den Wert `true` zurückliefern. Um die Suche zu starten, wird die Methode `accept` des AST-Wurzelknotens aufgerufen und als Parameter der Visitor übergeben. Es ist durchaus üblich, die Visitors zu schachteln. Man sucht mit einem Visitor nach einem Knotentyp und innerhalb der visit-Methode wird auf dem gefundenen Knoten ein anderer Visitor aufgerufen. Damit können sonst kompliziert zu programmierende Suchen elegant gelöst werden.

3.2.5 SearchEngine

Um bestimmte Java-Elemente im Workspace zu finden, bietet sich die Klasse `SearchEngine` an. Sie unterstützt unter anderem die Suche nach Methodenreferenzen, nach Felddeklarationen und nach Klassen, die ein bestimmtes Interface implementieren [EHil06].

Alternativ können Java-Elemente auch mit Hilfe von ASTs gefunden werden. Dafür muss für jede Compilation-Unit der zugehörige AST aufgestellt und dann mit dem Vistor-Pattern durchsucht werden. Diese Vorgehensweise ist aber wesentlich aufwändiger, da hierzu alle Java-Elemente in den Arbeitsspeicher geladen werden müssen.

Um die `SearchEngine` benutzen zu können, werden einige Hilfsklassen benötigt. Mit einem `SearchPattern` wird definiert, nach welchen Java-Elementen gesucht werden soll. Eine Instanz des Typs `SearchScope` dient dazu, den Suchbereich einzuschränken. Die Suche kann so z. B. nur auf ein bestimmtes Projekt oder Paket beschränkt werden. Auch kann angegeben werden, ob Archive mit untersucht werden sollen. Ein Objekt vom Typ `SearchRequestor` wird von der `SearchEngine` benötigt, um die Suchergebnisse zu speichern. Da die Klasse abstrakt ist, muss eine konkrete Unterklasse gebildet werden.

Das folgende Beispiel soll das Zusammenspiel der Klassen verdeutlichen. Die Methode `searchMethodReferences` sucht im Projekt `project` nach Methodenaufrufen der Methode `method`. Die Suche wird auf Source-Folder eingeschränkt, d. h. nur Aufrufe, die in einer `ICompilationUnit` vorkommen werden berücksichtigt. Um die Ergebnisse zu speichern, wird eine anonyme Klasse von der abstrakten Klasse `SearchRequestor` abgeleitet. Die Suche selbst wird mit dem Aufruf der Methode `search` der `SearchEngine`-Instanz gestartet. Für jeden Treffer ruft die `SearchEngine` die Methode `acceptSearchMatch` der übergebenen `SearchRequestor`-Instanz auf. Das Suchergebnis wird in Form eines Objekts vom Typ `SearchMatch` übergeben. Das Objekt vom Typ `SearchMatch` enthält Informationen über die genaue Stelle des Fundes und über das Java-Element (dies kann z. B. eine Methode sein), in der der Fund aufgetreten ist. Die Suchergebnisse werden in der Liste `searchMatches` gespeichert.

```
public List<SearchMatch> searchMethodReferences(IJavaElement method,
                                  IJavaProject project) throws CoreException {

    // Suchbereich auf die SourceFolders im Projekt project einschränken
    IJavaSearchScope scope = SearchEngine.createJavaSearchScope(
                    new IJavaElement[] { project }, IJavaSearchScope.SOURCES);

    // Suche nach Methodenaufrufen von method
    SearchPattern pattern = SearchPattern.createPattern(method,
                                            IJavaSearchConstants.REFERENCES);

    final List<SearchMatch> searchMatches = new ArrayList<SearchMatch>();
```

```
    // Erzeugen des SearchRequestors
    SearchRequestor requestor = new SearchRequestor() {
        @Override
        public void acceptSearchMatch(SearchMatch match)
                                                throws CoreException {
            //Suchergebnis in Liste searchMatches speichern
            searchMatches.add(match);
        }
    };
    SearchEngine searchEngine = new SearchEngine();
    // Start der Suche
    searchEngine.search(pattern,
                        new SearchParticipant[] {
                           SearchEngine.getDefaultSearchParticipant() },
                        scope,requestor, null);

    return searchMatches;
}
```

3.3 Refactoring in Eclipse

Eclipse bietet eine beachtliche Anzahl an Refactoring-Werkzeugen. Während häufig genutzte Werkzeuge, wie z. B. die Umbenennung von Methoden und Feldern, sehr gut funktionieren, weisen andere Werkzeuge die eine oder andere Schwäche auf. Wenn nach dem Refactoring ein Compilerfehler auftritt, kann der Benutzer dies leicht erkennen und die Änderungen mit einem „Undo“ wieder rückgängig machen. Tückischer ist der Fall, wenn es zu keinem Compilerfehler kommt, aber die Semantik des Programms verändert wird. Als Beispiel sei hier das Werkzeug „Declared Type Refactoring“ erwähnt. Dieses hat unter anderem Probleme, überladene Methoden oder Methoden, die in Subklassen verschattet werden, richtig zu berücksichtigen. In seltenen Fällen kann die Anwendung dieses Werkzeugs deshalb eine Semantikänderung des refaktorisierten Programms verursachen. Diese Semantikänderungen lassen sich nur durch eigenes Testen, z. B. mit entsprechenden JUnit-Tests [Link05] aufdecken.

Im Artikel „Refactoring for Generalization using Type Constraints“ [Tip03], der die Grundlagen dieses Werkzeugs behandelt, wird erwähnt, dass die Grundlagen im Falle von überladenen Methoden noch nicht geklärt sind. Im Hilfetext ist von den Beschrän-

kungen dieses Refactoring-Werkzeugs nichts erwähnt, so dass ein Benutzer davon ausgehen könnte, dass es immer korrekt arbeitet. Die Schwächen und Beschränkungen der einzelnen Refactoring-Werkzeuge sollten nach Meinung des Verfassers deshalb im Hilfetext zumindest erwähnt werden.

4 Programmgesteuertes Öffnen und Schließen einer Klasse

4.1 Einführung

Das Öffnen und Schließen einer Klasse soll programmgesteuert erfolgen. Dafür muss ein Verfahren gefunden werden, mit dem die maximal mögliche bzw. minimal mögliche Sichtbarkeit von Methoden bestimmt werden kann. Eine Möglichkeit, die zulässige Sichtbarkeit zu ermitteln ist, Regeln aufzustellen, wann eine Methode welche Sichtbarkeit annehmen darf. Mit diesen Regeln kann dann die zulässige Sichtbarkeit jeder Methode bestimmt werden. Dieser Ansatz wird in Abschnitt 4.6 weiter verfolgt. Die Regeln selbst werden in den Abschnitten 4.4 und 4.5 besprochen. Wie wir sehen werden, ist eine Prüfung im Vorfeld aufwändig durchzuführen. Deshalb soll im nächsten Abschnitt zunächst ein anderer, einfacherer Ansatz untersucht werden.

4.2 Trial-and-Error-Ansatz

Bei dem Trial-and-Error-Ansatz wird die zulässige Sichtbarkeit einer Methode mit Hilfe eines Compilers gefunden. Die Idee, die dahinter steckt ist, dass eine nicht zulässige Sichtbarkeitsänderung auch einen Übersetzungsfehler zur Folge hat. Im folgenden Kapitel wird allerdings gezeigt werden, dass es, trotz fehlerfreier Übersetzung, zu einer Semantikänderung kommen kann, so dass der Ansatz modifiziert werden muss. Der Ansatz wird Trial-and-Error-Ansatz genannt, da die zulässige Sichtbarkeit durch Versuch und Irrtum bestimmt wird.

Im Folgenden wird die Vorgehensweise für das Schließen einer Klasse erläutert. Die Sichtbarkeit einer Methode wird hierzu Schritt für Schritt reduziert. Ist die Sichtbarkeit z. B. *public*, wird die Sichtbarkeit erst nach *protected*, dann nach *default* und zuletzt nach *private* geändert. Nach jedem Schritt wird das Projekt kompiliert. Wenn bei der Übersetzung Fehler auftreten, ist diese Sichtbarkeit nicht zulässig. Die letzte erfolgreich geprüfte Sichtbarkeit ergibt die minimal mögliche Sichtbarkeit der Methode, und die Suche kann abgebrochen werden.

Es kann auch eine andere Reihenfolge gewählt werden. Zuerst wird die Sichtbarkeit auf die kleinste Sichtbarkeit gesetzt. Treten keine Übersetzungsfehler auf, ist die minimale

Sichtbarkeit gefunden. Bei Übersetzungsfehlern wird die Sichtbarkeit schrittweise erhöht, bis die ursprüngliche Sichtbarkeit erreicht ist. Eine Methode mit Sichtbarkeit *public* wird z. B. erst nach *private*, dann nach *default* und dann nach *protected* geändert.

Welche Reihenfolge im Durchschnitt weniger Prüfungen erfordert, hängt davon ab, wie viele Methoden reduziert werden können. Können z. B. viele Methoden von *public* nach *private* geändert werden, ist die zweite Vorgehensweise effizienter. Wenn die Sichtbarkeit vieler Methoden nur um eine Stufe reduziert werden kann, ist die erste Vorgehensweise vorzuziehen. Wenn alle Methoden schon ihre minimal mögliche Sichtbarkeit aufweisen, benötigen beide Ansätze für jede Methode einen Versuch und damit insgesamt gleich viele Prüfungen.

Müsste nach einer Reduzierung der Sichtbarkeit jedes Mal das ganze Projekt kompiliert werden, wäre dies sehr aufwändig. Indem Werkzeuge zum automatisierten Erzeugen von Programmen aus Quelltexten, wie z. B. *Ant* [Ant07] eingesetzt werden, kann der Aufwand stark reduziert werden. *Ant* übersetzt nur die Quelldateien neu, die sich nach dem letzten Build geändert haben. Dabei benutzt *Ant* den Standard-Compiler javac von Sun. Aber auch dieser Aufwand lässt sich noch minimieren, indem der Eclipse-eigene Compiler eingesetzt wird. Dieser Compiler ist ein inkrementeller Compiler, d. h. es müssen nur die Teile einer Compilation-Unit übersetzt werden, die sich seit dem letzten Build geändert haben.

Wenn der Eclipse-Compiler bei der Übersetzung auf Probleme stößt, meldet er dies entweder mit einer Warnung oder einem Fehler. Warnungen werden erzeugt, wenn das Programm weiterhin ausgeführt werden kann. Wenn ein Problem auftritt, das laut Java-Spezifikation einen Übersetzungsfehler darstellt, wird immer ein Fehler gemeldet. Für andere Probleme kann in den Compiler-Einstellungen angegeben werden, ob sie als Fehler oder Warnungen ausgegeben oder ob sie ganz ignoriert werden sollen[1] [EHil06]. Ob die Sichtbarkeit einer Methode geändert werden kann, hängt vom Auftreten von Übersetzungsfehlern ab. Wenn bei einer bestimmten Compiler-Einstellung Warnungen als Über-

[1] Ein Beispiel: Wenn eine Methode eine andere überschreibt, kann sie mit der Annotation @Override versehen werden. Eine solche Annotation ist laut Java-Spezifikation allerdings nicht zwingend notwendig. Der Compiler kann nun so eingestellt werden, dass er einen Fehler erzeugt, wenn diese @Override-Annotation bei einer überschreibenden Methode fehlt.

setzungsfehler ausgegeben werden, kann unter Umständen eine andere maximale Sichtbarkeit erreicht werden, als wenn nur die Fehler laut Java-Spezifikation als Übersetzungsfehler gelten. Im Folgenden wird angenommen, dass nur die Fehler laut Spezifikation als Übersetzungsfehler angesehen werden.

Nun zur Vorgehensweise beim Öffnen einer Klasse. Die Sichtbarkeit einer Methode wird zuerst auf *public* gesetzt. Wenn nach der Änderung ein Übersetzungsfehler auftritt, wird schrittweise die jeweils nächst-niedrigere Sichtbarkeit geprüft. Der Fall, dass die Sichtbarkeit einer Methode nicht erhöht werden darf, dürfte eher selten auftreten. Wie in Abschnitt 4.5 gezeigt werden wird, kann die Öffnung einer Methode nur dann einen Übersetzungsfehler verursachen, wenn die Methode vor der Öffnung nicht in einem Subtyp sichtbar ist und es außerdem in diesem Subtyp eine Methode gibt, die den gleichen Namen aufweist.

4.3 Probleme durch den Trial-and-Error-Ansatz

4.3.1 Allgemeines

Der Trial-and-Error-Ansatz stellt nur sicher, dass eine Sichtbarkeitsänderung zulässig ist, d. h. dass sie zu keinem Übersetzungsfehler führt. Wenn aufgrund der Sichtbarkeitsänderung einer Methode allerdings die Semantik des Programms verändert wird, kann dies durch diesen Ansatz nicht aufgedeckt werden, da hierbei keine Compilerfehler auftreten.

Nach gründlicher Durchsicht der Java-Spezifikation [Gosl05] wurden vom Verfasser zwei Fälle gefunden, bei denen eine solche Semantikänderung auftreten kann: Wenn die geänderte Methode überschrieben wird oder überladen ist. Die beiden Fälle werden nachfolgend näher untersucht.

Es kann allerdings im Rahmen dieser Arbeit nicht bewiesen werden, dass es keine weiteren Fälle gibt, bei denen eine Sichtbarkeitsmodifikation einer Methode eine Semantikänderung hervorrufen kann.

4.3.2 Überschriebene Methoden

Die Tatsache, dass durch die Sichtbarkeitsmodifikation einer überschriebenen Methode eine Semantikänderung auftreten kann, ist nicht unbedingt auf den ersten Blick ersichtlich. Im folgenden Beispiel kann z. B. eine Änderung der Semantik nicht direkt hervorgerufen werden.

```
class A {
     public void m() {
          System.out.println("In Methode A#m");
     }
}

class B extends A {
     public void m() {
          System.out.println("In Methode B#m");
     }
     public static void main(String[] args) {
          A o = new B();
          o.m();
     }
}
```

Vor der Änderung wird die Methode `A#m` von der Methode `B#m` überschrieben. Das Objekt `o` ist vom statischen Typ `A` und vom dynamischen Typ `B`. An der Aufrufstelle wird deshalb `B#m` aufgerufen. Wenn die Sichtbarkeit der überschriebenen Methode `A#m` nun in *private* geändert wird, überschreibt die Methode `B#m` die Methode `A#m` nicht mehr. Die Methode `A#m` ist an der Aufrufstelle nicht mehr sichtbar, deshalb tritt beim Übersetzen des Programms ein Fehler auf.

Für den Fall, dass `A#m` nur innerhalb der Klasse `A` aufgerufen würde, also die Aufrufstelle nicht existierte, träte weder ein Übersetzungsfehler auf, noch käme es zu einer Änderung des Programmablaufs. Zwischen einer Methode im Supertyp und einer überschreibenden Methode im Subtyp besteht allerdings eine semantische Beziehung, die auch nach dem Schließen der Klassen erhalten bleiben soll. Deshalb sollte, wenn eine Methode *m* eine Methode in einem Supertyp überschreibt, dies auch nach der Veränderung der Sichtbarkeit der Fall sein.

Wenn eine Methode nur innerhalb der gleichen Klasse aufgerufen wird, könnte man meinen, dass sie auf *private* gesetzt werden könnte. Bei Vererbung kommt aber ein Fall ins Spiel, bei dem dies eine Modifikation der Semantik zur Folge hätte. Folgendes Beispiel macht dies deutlich.

```
class A {
     public void calculate() {
          m();
     }
     public void m() {
          System.out.println("In Methode A#m");
     }
}

class B extends A {
     public void m() {
          System.out.println("In Methode B#m");
     }
     public static void main(String[] args) {
          B b = new B();
          b.calculate();
     }
}
```

Die Methode `A#m` wird nur innerhalb der Methode `calculate` der Klasse `A` aufgerufen. `A#m` wird in der Subklasse `B` überschrieben. Wenn das Programm ausgeführt wird, wird in der Methode `calculate` die überschriebene Methode `B#m` aufgerufen. Nun wird versucht, die Sichtbarkeit der Methode `A#m` auf *private* zu reduzieren. Eine Reduktion auf *private* verursacht keinen Compilerfehler. Da die Methode `A#m` die Sichtbarkeit *private* hat, wird sie nicht mehr dynamisch gebunden und kann deshalb nicht mehr von `B#m` überschrieben werden. Beim Ausführen des Programms wird jetzt statt `B#m` die Methode `A#m` aufgerufen. Die Semantik des Programms hat sich verändert.

Daraus lässt sich folgern: Wenn eine Methode m_1 von einer Methode m_2 überschrieben wird, muss dies auch nach der Reduzierung der Sichtbarkeit von m_1 gelten.

Diese Bedingung wird immer verletzt, wenn die Sichtbarkeit einer überschriebenen Methode in *private* oder *default* geändert wird und sich die überschreibende Methode in einem anderen Paket befindet[1].

Eine Änderung der Sichtbarkeit einer überschreibenden Methode m_2 kann dagegen keinen Einfluss darauf haben, ob sie eine Methode m_1 (die sich in einer Superklasse befindet) überschreibt oder nicht. Wenn die Methode m_1, vor ihrer Änderung der Sichtbarkeit, die Methode m_2 überschrieben hat, ist dies auch nach der Änderung der Fall. Wenn die Sichtbarkeit von m_2 nach der Änderung kleiner wird als die Sichtbarkeit der überschriebenen Methode m_1, tritt ein Übersetzungsfehler auf.

Bei einer Erweiterung der Sichtbarkeit ist es gerade umgekehrt. Im einem weiteren Beispiel sei nun die Sichtbarkeit von `A#m` *private* und die Sichtbarkeit von `B#m` wieder *public*. Zwischen den Methoden `A#m` und `B#m` bestehe nun keine semantische Beziehung, sie haben unbeabsichtigt den gleichen Namen und die gleiche Signatur. `B#m` überschreibt nun die Methode `A#m` nicht. Wird das Programm ausgeführt, wird `A#m` aufgerufen. Wenn die Sichtbarkeit von `A#m` auf *public* erweitert wird, überschreibt `B#m` die Methode `A#m`. Nun wird im Hauptprogramm statt `A#m` die Methode `B#m` aufgerufen. Die Semantik des Programms hat sich verändert.

Daraus lässt sich folgern: Wenn eine Methode m_1 vor Erweiterung der Sichtbarkeit von einer Methode m_1 nicht überschrieben wird, darf dies auch nach der Erweiterung nicht der Fall sein.

4.3.3 Überladene Methoden

Wenn eine Methode überladen ist und ihre Sichtbarkeit geändert wird, kann es nur unter bestimmten Bedingungen zu Problemen kommen, die keinen Übersetzungsfehler erzeugen.

[1] Genau genommen gibt es bei der Änderung der Sichtbarkeit in *default* noch einen Fall, bei dem die Bedingung verletzt wird. Dieser wird in Abschnitt 5.4.7, bei der Erläuterung der Implementierungsdetails, genauer besprochen.

```
class A {
}
class B extends A {
     void calc() {
          System.out.println("In Methode calc()");
     }
     void calc(A a) {
          System.out.println("In Methode calc(A)");
     }
     void calc(B b) {
          System.out.println("In Methode calc(B)");
     }
}
class Main {
     public static void main(String[] args) {
          B b = new B();
          A a = new A();

          b.calc();   //Aufrufstelle 1
          b.calc(a);  //Aufrufstelle 2
          b.calc(b);  //Aufrufstelle 3
     }
}
```

Obige Klasse `B` enthält drei überladene Methoden. Wenn an einer Programmstelle, wie z. B. an der Aufrufstelle 1, die Methode `calc()` aufgerufen wird, und die Sichtbarkeit von `calc()` dann so verringert wird, dass die Methode dort nicht mehr sichtbar ist, tritt immer ein Übersetzungsfehler auf.

Anders liegt der Fall, wenn der Methode `calc` an einer Programmstelle, wie der Aufrufstelle 3, ein Objekt vom statischen Typ `B` übergeben wird. Da die Methode `calc(B)` spezieller als `calc(A)` ist, wird die Methode `calc(B)` aufgerufen. Wird jetzt die Sichtbarkeit der Methode `calc(B)`so verringert, dass sie nicht mehr an der Aufrufstelle sichtbar ist, wird stattdessen `calc(A)` aufgerufen. Die Semantik des Programms hat sich verändert. Wird die Sichtbarkeit der Methode `calc(B)` dann wieder erhöht, wird wieder `calc(A)` aufgerufen. Also kann es auch bei der Öffnung einer Methode zu einer Semantikänderung kommen.

Wenn die Sichtbarkeit der Methode `calc(A)` verringert werden soll, kann dagegen kein Konflikt mit `calc(B)` auftreten. Wird der Methode `calc` ein Objekt vom statischen Typ `A` übergeben (Aufrufstelle 2), wird `calc(A)` aufgerufen. Wird die Sichtbarkeit von `calc(A)` verringert, kann stattdessen nicht `calc(B)` aufgerufen werden. In diesem Fall tritt ein Übersetzungsfehler auf. Auch bei einer Erhöhung der Sichtbarkeit von `calc(A)` kann es, aufgrund der Methode `calc(B)`, nie zu einer Semantikänderung kommen. Wenn vor der Sichtbarkeitsänderung von `calc(A)` die Methode `calc(B)` aufgerufen wird, wird sie auch nach der Änderung aufgerufen.

Bedingungen für das Schließen

Es kann bei einer Sichtbarkeitsmodifikation zu Semantikänderungen kommen, wenn die geschlossene Methode m_1 durch eine Methode m_2 ersetzt[1] werden kann. Die Methode m_1 befinde sich in der Klasse C_1. Die Methode m_2 kann sich in der gleichen Klasse wie m_1 befinden oder in einem Super- oder Subtypen von C_1.

Wenn die Sichtbarkeit der Methode m_1 reduziert wird, kann es zu einem Ersetzen der Methode m_1 dann kommen, wenn vor der Modifikation der Sichtbarkeit im Programm an einer Stelle die Methode m_1 aufgerufen wird und an dieser Stelle m_2 sichtbar ist. Nach der Änderung ist die Methode m_1 an dieser Stelle nicht mehr sichtbar. Statt m_1 wird nun m_2 aufgerufen.

Eine solche Aufrufstelle kann es immer dann geben, wenn die Sichtbarkeit der Methode m_1 nach der Änderung kleiner als die Sichtbarkeit der Methode m_2 wird. Wenn die Methoden nach der Änderung nicht mehr überladen sind, kann es eine solche Stelle auch dann geben, wenn die Sichtbarkeit der Methode m_1 nicht kleiner als die der Methode m_2 wird. (Die Methoden können nicht mehr überladen sein, wenn m_2 in einem Subtypen von C_1 definiert ist und die Sichtbarkeit von m_1 z. B. in *private* geändert wird. Die problematische Aufrufstelle kann dann in diesem Subtypen liegen.)

[1] Die Bezeichnung „Ersetzung“ für den beschriebenen Vorgang stammt vom Autor dieser Arbeit.

Damit eine Methode m_1 durch eine Methode m_2 ersetzt werden kann, müssen folgende Zusammenhänge bestehen:

- Vor der Änderung der Sichtbarkeit muss die Methode m_1 durch die Methode m_2 überladen sein.
- Beide Methoden müssen die gleiche Anzahl an Parametern aufweisen. Die Anzahl der Parameter muss größer null sein.
- Der i-te Parametertyp (mit $1 \leq i \leq$ Parameteranzahl) der Methode m_2 muss den i-ten Parametertyp der Methode m_1 ersetzen können. Ein Parametertyp kann durch den gleichen Typ oder einen seiner Supertypen ersetzt werden. Ist der Typ ein primitiver Typ oder ein Wrappertyp, sind die Typen, die ihn zusätzlich ersetzen können, in der Tabelle 4.1 sowie der Tabelle 4.2 angegeben.

Damit ein Problem durch Überladung festgestellt wird, reicht es aus, dass es theoretisch eine Programmstelle geben könnte, an der eine Methode m_1 nach Reduktion ihrer Sichtbarkeit durch eine Methode m_2 ersetzt werden könnte. Es ist nicht notwendig, dass eine Stelle mit einem solchen Methodenaufruf tatsächlich existiert. Denn ein Programmierer könnte diesen Methodenaufruf auch nach einem Schließen der Klasse hinzufügen[1]. Er wäre sich unter Umständen nicht bewusst, dass statt der Methode m_1 nun die Methode m_2 aufgerufen werden würde. (Außerdem spart man sich mit dieser Annahme die aufwändige Suche nach Methodenreferenzen im gesamten Projekt.)

Bedingungen für das Öffnen

Bei einer Sichtbarkeitsmodifikation kann es zu Semantikänderungen kommen, wenn eine Methode m_2 durch die geöffnete Methode m_1 ersetzt werden kann. Die Methode m_1 befinde sich in der Klasse C_1. Die Methode m_2 kann sich in der gleichen Klasse wie m_1 befinden, in einem Supertypen von C_1 oder in einem Subtypen von C_1.

Wenn die Sichtbarkeit von m_1 vergrößert wird, kann bei folgendem Fall eine Semantikänderung auftreten. Im Programm wird, vor der Änderung, an einer Stelle die Methode m_2 aufgerufen, die Methode m_1 ist an dieser Stelle nicht sichtbar. Nach Erweiterung der

[1] Nach der Änderung können durch Hinzufügen von Methodenaufrufen auch Übersetzungsfehler auftreten, die ohne die Änderung nicht aufgetreten wären. Dies ist aber nicht so kritisch wie eine Semantikänderung, da ein Programmierer vom Compiler auf Übersetzungsfehler hingewiesen wird.

Sichtbarkeit ist m_1 an dieser Stelle nun sichtbar und es wird statt der Methode m_2 die Methode m_1 aufgerufen. Damit die Methode m_1 aufgerufen wird, muss m_1 für die Aufrufstelle spezieller als m_2 sein. Zwischen den Parametertypen der beiden Methoden müssen daher die gleichen Zusammenhänge bestehen, wie beim Schließen beschrieben.

Wenn die Methoden m_1 und m_2 vor sowie nach dem Öffnen überladen sind, ist eine solche Aufrufstelle nur möglich, wenn die Sichtbarkeit der Methode m_1 größer oder gleich der Sichtbarkeit der Methode m_2 wird. Vor der Änderung muss die Sichtbarkeit von m_1 kleiner als die von m_2 sein.

Sind die Methoden nur nach dem Öffnen überladen, kann es eine solche Aufrufstelle unabhängig von den Sichtbarkeiten von m_1 und m_2 geben. (In diesem Fall liegt m_2 in einem Subtypen von C_1 und m_1 ist nur nach dem Öffnen in diesem Subtyp sichtbar.)

Ersetzung der Parametertypen

Es sind hier die Subtypbeziehungen relevant, wie sie in §4.10.1 der Java-Spezifikation definiert sind. Diese schließen auch die Subtypbeziehung zwischen Primitiven ein. Die Subtypbeziehungen zwischen Primitiven lauten (siehe [Gosl05] Abschnitt 4.10.1):

```
double > float > long > int > short > byte
int > char
boolean
```

Ersetzung durch Autoboxing

Es gibt, außer durch Subtypbeziehung, noch eine weitere Möglichkeit, wie ein Parametertyp durch einen anderen ersetzt werden kann. Dies hängt mit dem in Java 1.5 eingeführten Autoboxing zusammen. Durch Autoboxing können an Stellen, an denen ein Wrappertyp erwartet wird, Primitive automatisch in einen Wrappertyp umgewandelt werden. Und an Stellen an denen ein primitiver Typ erwartet wird, werden Wrappertypen automatisch in Primitive umgewandelt.

```
public class A {
     void m(int n){
          System.out.println("In Methode m(int n) von Klasse A");
     }
}

public class B extends A {
     void m(Integer n){
          System.out.println("In Methode m(Integer n) von Klasse B");
     }

     public static void main(String[] args) {
          B a = new B();
          b.m(2);          ← Aufrufstelle
     }
}
```

Im obigen Beispiel wird an der Aufrufstelle die Methode `A#m(int)` aufgerufen, da sie spezieller als `B#m(Integer)` ist. Wird die Sichtbarkeit von `A#m(int)` auf *private* reduziert, ist sie an der Aufrufstelle nicht mehr sichtbar. Stattdessen wird `B#m(Integer)` aufgerufen. Dies ist durch Autoboxing möglich, da der Parameter vom Typ `int` automatisch in den Wrappertyp `Integer` umgewandelt wird.

In den folgenden Tabellen ist zusammengefasst, wann ein primitiver Typ von einem Wrappertyp, und wann ein Wrappertyp von einem Primitiven ersetzt werden kann. Die numerischen Wrappertypen sind alle direkt vom Typ `java.lang.Number` abgeleitet. Es gibt z. B. zwischen `Double` und `Float` keine Subtypbeziehung, wie sie zwischen den primitiven Typen `double` und `float` existiert. Deshalb kann der Typ `Float` auch nicht durch den Typ `Double` ersetzt werden.

Tabelle 4.1: Ersetzung eines primitiven Typs durch einen Wrappertyp

Primitiver Typ	Kann ersetzt werden durch Wrappertyp
`double`	`Double`
`float`	`Double, Float`
`long`	`Double, Float, Long`
`int`	`Double, Float, Long, Integer`
`short`	`Double, Float, Long, Integer, Short`
`byte`	`Double, Float, Long, Integer, Short, Byte`
`char`	`Double, Float, Long, Integer, Character`
`boolean`	`Boolean`

Tabelle 4.2: Ersetzung eines Wrappertyps durch einen primitiven Typ

Wrappertyp	Kann ersetzt werden durch Primitive vom Typ
`java.lang.Double`	`double`
`java.lang.Float`	`float`, Supertypen von `float`
`java.lang.Long`	`long`, Supertypen von `long`
`java.lang.Integer`	`int`, Supertypen von `int`
`java.lang.Short`	`short`, Supertypen von `short`
`java.lang.Byte`	`byte`, Supertypen von `byte`
`java.lang.Character`	`char`, Supertypen von `char`
`java.lang.Boolean`	`boolean`

Unberücksichtigte Sprachelemente

Mit der Version 5 von Java wurden Methoden mit variabler Parameteranzahl, „vararg“ genannt, eingeführt. Wie sich dieses neue Sprachelement auf die diskutierten Probleme durch überladene Methoden auswirkt, wurde aus Zeitgründen nicht untersucht.

4.3.4 Öffnungs- und Schließungsreihenfolge bei Vererbungshierarchien

Mit dem *Access-Modifier-Modifier* soll es möglich sein ein ganzes Java-Projekt zu schließen bzw. zu öffnen. Hierbei werden alle im Projekt vorhandenen Klassen geschlossen bzw. geöffnet. Die Klassen müssen in einer bestimmten Reihenfolge geschlossen oder geöffnet werden, damit sie ihre minimalen Schnittstellen erhalten. Ebenso kann es

notwendig sein, die Methoden innerhalb einer Klasse in einer bestimmten Reihenfolge zu schließen bzw. zu öffnen.

Überschreibende Methoden

Wenn Klassen zu einer Vererbungshierarchie gehören, ist es wichtig, die Klassen in der richtigen Reihenfolge zu schließen bzw. zu öffnen. Ein Beispiel soll dies verdeutlichen.

```
class A {
     public void m(){
     }
}

class B extends A{
     public void m(){
     }
}
```

Wenn die Klasse `B` zuerst geschlossen wird, kann die Sichtbarkeit von `B#m` nicht reduziert werden, da `B#m` die Methode `A#m` überschreibt. Denn die Sichtbarkeit von `A#m` ist *public*, und eine überschreibende Methode darf keine kleinere Sichtbarkeit haben als die überschriebene Methode.

Wenn die Klasse `A` zuerst geschlossen wird, kann die Sichtbarkeit von `A#m` in *default* geändert werden. Nachdem sie geschlossen wurde, kann die Sichtbarkeit von `B#m` in *default* geändert werden, da `A#m` jetzt diese Sichtbarkeit aufweist. Folgerung: Wenn eine Hierarchie geschlossen werden soll, muss eine Klasse vor ihren Subtypen geschlossen werden.

Die Sichtbarkeit der Methoden `A#m` und `B#m` sei jetzt *default*. Beide Klassen sollen geöffnet werden. Die Klasse `B` muss vor der Klasse `A` geöffnet werden, damit die Sichtbarkeit von `A#m` und `B#m` auf *public* erweitert werden kann. Beim Öffnen von Klassen ist die Reihenfolge also genau umgekehrt: Eine Klasse muss vor ihren Supertypen geöffnet werden.

Überladene Methoden, Fall Schließen

Eine Methode m_2, die eine Methode m_1 ersetzen kann, muss zuerst geschlossen werden. Wenn m_1 und m_2 die gleiche Sichtbarkeit aufweisen, darf die Sichtbarkeit von m_1 nur

reduziert werden, nachdem die Sichtbarkeit von m_2 reduziert wurde. (Wenn die Methode m_2 eine kleinere Sichtbarkeit als m_1 aufweist, kann die Sichtbarkeit von m_1 ohne Probleme auf die Sichtbarkeit von m_2 reduziert werden.)

Die Methode m_2 kann sich in der gleichen Klasse wie m_1, in einer Subklasse oder in einer Superklasse befinden. Es gibt also Fälle, in denen eine Superklasse vor einer Subklasse geschlossen werden muss, und Fälle in denen eine Subklasse vor einer Superklasse geschlossen werden muss. Es kann durchaus sein, dass eine Klasse C_1 vor ihrer Subklasse C_2, und die Subklasse C_2 vor der Klasse C_1 geschlossen werden muss. Dies ist der Fall, wenn sich in Klasse C_1 eine Methode m_1 befindet, die von einer Methode m_2 in Klasse C_2 ersetzt werden kann, und sich wiederum in Klasse C_2 eine Methode m_3 befindet, die von einer Methode m_4 in Klasse C_1 ersetzt werden kann.

Wenn eine Hierarchie geschlossen werden soll, muss eine Klasse bei dieser Vorgehensweise eventuell mehrmals untersucht werden. Im aufgeführten Fall würde zuerst Klasse C_1 geschlossen werden, dann Klasse C_2 und dann wiederum Klasse C_1. Beim ersten Schließen der Klasse C_1 kann nur die Methode m_4 geschlossen werden. Durch darauf folgendes Schließen der Klasse C_2 können die Methoden m_2 und m_4 geschlossen werden. Beim zweiten Schließen der Klasse C_1 kann nun die Methode m_1 geschlossen werden, da die Methode m_2 nun geschlossen ist und nicht mehr m_1 ersetzen kann.

Bei dieser Vorgehensweise gibt es Fälle, für die die minimale Sichtbarkeit nicht bestimmt werden kann. Eine Klasse `A` enthalte eine Methode `A#m(int)` und ihre Subklasse `B` die Methode `B#m(Integer)`. Beide Methoden weisen die gleiche Sichtbarkeit auf. Wenn die Klasse `A` geprüft wird, wird festgestellt, dass `A#m(int)` nicht reduziert werden kann, da sie durch `B#m(Integer)` ersetzt werden kann (siehe Regel S.13 aus Abschnitt 4.4.2). Bei Prüfung der Klasse `B` wird wiederum festgestellt, dass `B#m(Integer)` nicht reduziert werden kann, da sie durch `A#m(int)` ersetzt werden kann. Die Sichtbarkeit der Methoden kann in diesem Fall also nicht reduziert werden, wenn je eine Klasse für sich allein geschlossen wird, unabhängig von der Reihenfolge in der die Klassen geschlossen werden.

4.3.5 Erweiterung des Trial-and-Error-Ansatzes

Angesichts der oben aufgezeigten Fälle, bei denen es zu einer Semantikänderung des Programms kommen kann, kann der Trial-and-Error-Ansatz nicht allein eingesetzt werden. Es wird deshalb eine Kombination aus Trial-and-Error-Ansatz und einer Prüfung im Vorfeld gewählt. Bevor der Trial-and-Error-Ansatz zum Einsatz kommt, wird geprüft, ob es zu Semantikänderungen kommen kann. Bei möglichen Semantikänderungen wird die Sichtbarkeit dann nicht geändert.

4.4 Bedingungen für die Einschränkung der Sichtbarkeit

4.4.1 Allgemeines

Wenn eine Klasse geschlossen wird, muss die minimal mögliche Sichtbarkeit für ihre Methoden ermittelt werden. Wenn eine Methode z. B. von außerhalb des Paketes, indem sie definiert ist, aufgerufen wird, kann die Sichtbarkeit nur *public* betragen. Nachfolgend werden Regeln aufgestellt, mit denen für alle Fälle die minimale Sichtbarkeit bestimmt werden kann. Wenn die Regel nicht offensichtlich ist, werden eine kurze Begründung und der zugehörige Paragraph der Java-Spezifikation [Gosl05] angegeben.

Es wird angenommen, dass vor der Änderung der Sichtbarkeit keine Übersetzungsfehler vorliegen. Dies vereinfacht die zu untersuchenden Fälle beträchtlich. Die Verletzungen einiger der Regeln verursachen einen Übersetzungefehler, können also von einem Compiler erkannt werden. Die Regeln, die verhindern, dass sich die Semantik des Programms ändern kann, das Programm aber ohne Fehler übersetzt werden kann, basieren auf den Überlegungen des vorherigen Abschnitts.

4.4.2 Einschränkung der Sichtbarkeit von Methoden

Wenn die Sichtbarkeit einer Methode m_1 einer Klasse C_1 reduziert werden soll, müssen alle Regeln dieses Abschnitts eingehalten werden. C_1 kann auch eine innere, lokale oder anonyme Klasse sein.

Überschreiben oder Implementieren

Regel S.1: Wenn durch m_1 eine Methode eines Interfaces implementiert wird, muss die Sichtbarkeit von m_1 *public* sein. Begründung: Alle Methoden eines Interfaces sind implizit *public*, und eine implementierende Methode darf keine geringere Sichtbarkeit aufweisen als die implementierte Methode.

Regel S.2: Wenn m_1 eine Methode überschreibt, darf die Sichtbarkeit von m_1 nicht geringer werden, als die Sichtbarkeit der überschriebenen Methode.

Regel S.3: Wenn m_1 von einer Methode m_2 überschrieben wird, muss dies auch nach der Reduzierung der Sichtbarkeit von m_1 gelten, wenn m_2 mit @Override annotiert ist (siehe § 9.6.1.4)[1].

Methodenaufrufe

Für einen Methodenaufruf kann, aufgrund von Polymorphismus, zum Übersetzungszeitpunkt nicht immer bestimmt werden, welche Methode tatsächlich aufgerufen wird. Wird die Methode über eine Instanz aufgerufen, ist nachfolgend immer die Methode gemeint, die für den statischen Typ aufgerufen werden würde.

In §6.6 der Java-Spezifikation ist aufgeführt, an welcher Programmstelle eine Methode sichtbar ist. Daraus können die Regeln S.4 bis S.9 abgeleitet werden. Wenn Regel S.13 eingehalten wird, verursacht eine Verletzung dieser Regeln immer einen Übersetzungsfehler.

Regel S.4: Wenn m_1 nur innerhalb der gleichen Klasse aufgerufen wird, kann die Sichtbarkeit *private* sein, sofern keine anderen Regeln dagegen sprechen. Das gilt auch, wenn der Methodenaufruf auf einer Instanz dieser Klasse erfolgt.

Regel S.5: Falls m_1 nicht aus dem Rumpf der Klasse auf höchster Ebene („body of the top level class“), aber aus dem gleichen Paket aufgerufen wird, muss m_1 mindestens *default* sein, es sei denn folgender Fall liegt vor: Die Methode m_1 sei in Klasse C_1 definiert und der Aufruf erfolgt von Klasse C_2 aus. Wenn es eine Klasse C_n gibt, die ein Subtyp von C_1 und ein Supertyp von C_2 ist, dann muss die Sichtbarkeit von m_1 mindestens *pro-*

[1] Die Verletzung der Regel S.3 hat immer einen Übersetzungsfehler zur Folge. Die Regel S.3 ist in Regel S.12 enthalten. Sie muss nur gesondert berücksichtigt werden, wenn Regel S.12, welche eine Semantikänderung prüft, nicht berücksichtigt wird.

tected sein, wenn C_n sich nicht im gleichen Paket wie C_1 und C_2 befindet. Wenn die Sichtbarkeit von m_1 *default* beträgt, ist m_1 in diesem Fall in C_2 nicht sichtbar, da m_1 verdeckt wird.

Regel S.6: Wenn m_1 aus einem Subtyp aufgerufen wird, und der Subtyp sich nicht im gleichen Paket befindet, muss die Sichtbarkeit mindestens *protected* sein.

Regel S.7: Wird m_1 an einer Stelle aufgerufen, die sich nicht in einer Subklasse befindet und nicht im gleichen Paket, in dem sich C_1 befindet, muss die Sichtbarkeit *public* betragen. Auch wenn m_1 aus einem Subtyp aufgerufen wird, der sich nicht im gleichen Paket befindet, muss die Sichtbarkeit *public* betragen, wenn folgendes zutrifft: Der Methodenaufruf erfolgt auf einem Objekt, der Aufruf liegt aber nicht innerhalb des Programmcodes, der für die Implementierung des Objekts verantwortlich ist. (Wann ein Programmcode für die Implementierung verantwortlich ist, wird detailliert in § 6.6.2 der Spezifikation aufgeführt.)

Methodenaufrufe durch innere oder äußere Klassen

Regel S.8: Wenn der Methodenaufruf von m_1 in einer Klasse C_2 erfolgt, und die Klasse C_2 und C_1 eine gemeinsame äußere Klasse besitzen, darf die Sichtbarkeit von m_1 *private* betragen.

Regel S.9: Wenn m_1 innerhalb des Rumpfes der Klasse auf höchster Ebene („body of the top level class“) aufgerufen wird, darf die Sichtbarkeit *private* sein. Das gilt auch, wenn der Methodenaufruf auf einer Instanz der Klasse C_1 erfolgt.

Verschiedenes

Regel S.10: Wenn m_1 *abstract* ist, muss die Sichtbarkeit mindestens *default* betragen. Eine abstrakte Methode darf nicht *private* werden, da sie sonst in Subklassen nicht implementiert werden kann (siehe §8.4.3.1).

Regel S.11: Wenn m_1 eine Main-Methode ist, muss die Sichtbarkeit *public* sein. (Wenn die Sichtbarkeit einer Main-Methode nicht *public* ist, kann sie beim Programmstart nicht ausgeführt werden. Eine kleinere Sichtbarkeit als *public* hat allerdings keinen Übersetzungsfehler zur Folge.)

Semantikänderung durch Überladen und Überschreiben

Regel S.12: Wenn die Methode m_1 vor Reduzierung der Sichtbarkeit von einer Methode m_2 überschrieben wird, muss m_1 auch nach der Reduzierung von m_2 überschrieben werden.

Regel S.13: Die Änderung der Sichtbarkeit darf nicht durchgeführt werden, wenn es eine Programmstelle geben kann, an der – nach der Änderung – ein Methodenaufruf von m_1 durch den Aufruf einer anderen Methode m_2 ersetzt wird. Die Details hierzu wurden in Abschnitt 4.3.3 besprochen.

4.4.3 Einschränkung der Sichtbarkeit von Feldern

Da Felder nicht wie Methoden überladen oder überschrieben werden können, ist hier die Anzahl der Regeln für die Sichtbarkeitsreduktion geringer. Die Regeln S.4 bis S.9, die für die Aufrufe von Methoden gelten, gelten entsprechend auch für die Zugriffe auf Felder.

Bei einem Feldzugriff kann zur Übersetzungszeit immer bestimmt werden, auf welches Feld tatsächlich zugegriffen wird. Felder können sich allerdings gegenseitig verdecken. Durch die Verletzung der Regeln kommt es immer zu einem Übersetzungsfehler. Eine Semantikänderung ist dagegen nicht möglich.

4.5 Bedingungen für die Erweiterung der Sichtbarkeit

4.5.1 Allgemeines

Es soll nun untersucht werden, unter welchen Umständen die Sichtbarkeit von Methoden und Feldern erweitert werden kann. Für die Erweiterung gelten die gleichen allgemeinen Überlegungen, die für die Einschränkung der Sichtbarkeit in Abschnitt 4.4.1 dargelegt wurden.

4.5.2 Erweiterung der Sichtbarkeit von Methoden

Wenn die Sichtbarkeit einer Methode m_1 einer Klasse C_1 erhöht werden soll, müssen alle Regeln dieses Abschnitts eingehalten werden. C_1 kann auch eine innere, lokale oder anonyme Klasse sein.

Regel Ö.1: Wenn m_1 vor der Änderung nicht in einer Subklasse sichtbar war, darf sie dort auch nach der Änderung nicht sichtbar sein, wenn in der Subklasse eine Methode m_2 mit einer zu m_1 override-äquivalenten Signatur[1] definiert ist, und mindestens eine der folgenden Bedingungen gilt:

- m_2 weist eine kleinere Sichtbarkeit als m_1 auf.
- m_1 ist final.
- m_2 ist eine statische Methode.
- m_1 ist eine statische Methode.
- Der Rückgabewert von m_2 kann den Rückgabewert von m_1 nicht substituieren. (Der Rückgabewert von m_2 ist nicht „return-type-substitutable“, siehe §8.4.5 der Spezifikation.)
- m_2 kann mehr Checked-Exceptions werfen als die Methode m_1. (Siehe §8.4.6 der Spezifikation.)

Regel Ö.2: Wenn m_1 vor der Änderung nicht in einer Subklasse sichtbar war, darf sie auch nach der Änderung nicht sichtbar sein, falls in der Subklasse eine Methode m_2 definiert ist, die m_1 überlädt, aber der Most-Specific-Algorithmus ergibt, dass keine überladene Methode spezieller als die andere ist. (Siehe §15.12.2.5 der Spezifikation.)

Semantikänderung durch Überladen und Überschreiben

Regel Ö.3: Wenn eine Methode m_1 vor Erweiterung ihrer Sichtbarkeit nicht von einer Methode m_2 überschrieben wird, darf die Methode m_1 auch nach der Erweiterung nicht von m_2 überschrieben werden.

Regel Ö.4: Die Sichtbarkeit einer Methode m_1 darf nicht erhöht werden, wenn es eine Programmstelle geben kann, an der – nach der Änderung – ein Methodenaufruf von m_2 durch den Aufruf von m_1 ersetzt wird. Die Details hierzu wurden in Abschnitt 4.3.3 besprochen.

[1] Wann zwei Methoden-Signaturen override-äquivalent sind, ist in §8.4.2 der Spezifikation definiert.

4.5.3 Erweiterung der Sichtbarkeit von Feldern

Die Erweiterung der Sichtbarkeit von Feldern kann weder Fehler verursachen, noch zu Semantikänderungen führen. Denn Felder können nicht überschrieben oder überladen werden. Felder können lediglich verdeckt werden. Ein Beispiel: Wenn die Sichtbarkeit eines Feldes f_1 erhöht wird, ist es nach der Änderung möglicherweise in Subklassen sichtbar, in denen es vorher nicht sichtbar war. Ist in diesen Subklassen ein Feld f_2 mit gleichem Namen definiert, wird das Feld f_1 durch f_2 verdeckt. Wenn vor der Änderung an einer Programmstelle das Feld f_2 aufgerufen wurde, wird es auch nach der Änderung weiterhin aufgerufen.

4.6 Hintergrundprüfung

Der Trial-and-Error-Ansatz kann für eine Prüfung im Hintergrund nicht eingesetzt werden. Um die Bedingungen zu prüfen, ändert er den Programmcode und diese Änderungen werden im Editor sichtbar. Wenn sich, durch die Veränderung der Sichtbarkeit, Übersetzungsfehler ergeben, wird dies dem Benutzer zudem durch Marker angezeigt. Ein Benutzer würde also sehen, dass sich der Programmcode ohne sein zutun ständig verändern würde, und dass ständig Marker auftauchen und verschwinden würden. Es ist zwar möglich, die Ressourcen, also die Dateien in denen sich der Programmtext befindet, direkt zu modifizieren, allerdings verursacht dies wieder neue Probleme.

Da der Trial-and-Error-Ansatz nicht eingesetzt werden kann, muss ein anderes Verfahren gewählt werden. Dieses Verfahren muss, um die minimale Sichtbarkeit einer Methode bestimmen zu können, alle Regeln aus Abschnitt 4.4.2 prüfen. Der Programmcode darf dabei nicht geändert werden und der Benutzer muss während der Prüfung normal weiterarbeiten können. Dazu muss die Prüfung möglichst Ressourcen-sparend ablaufen. Die Regeln S.11, S.12 und S.13 müssen auch für den Trial-and-Error-Ansatz programmgesteuert geprüft werden. Die Programmteile, die diese Regeln prüfen, können auch für eine Prüfung im Hintergrund eingesetzt werden. Die programmgesteuerte Prüfung der Regeln S.1 bis S.10 muss dagegen neu implementiert werden. Bei der Umsetzung dieser Regeln geht es im Wesentlichen darum, alle Aufrufe einer Methode zu finden. Die mini-

male Sichtbarkeit hängt dann davon ab, an welchen Programmstellen die Aufrufe erfolgen. Implementierungsdetails finden sich in Abschnitt 5.5.

4.7 Steuerung und Dokumentation mit Hilfe von Annotationen

4.7.1 Die Annotationstypen AccessLock und AccessUnlock

Wie schon in Abschnitt 2 diskutiert, soll ein Anwender eine Klasse öffnen können. Am Ende sollen die geöffneten Methoden so weit wie möglich wieder geschlossen werden, damit das implizite Interface der Klasse nicht größer als nötig wird. Deshalb sollen die geöffneten Methoden mit Informationen versehen werden, um die Öffnung zu dokumentieren. Dazu bieten sich die Annotationen an, die in Java 1.5 eingeführt wurden. Um die Änderungshierarchie zu speichern, könnte man in Eclipse statt den Annotationen auch Eclipse-Marker[1] verwenden. Wenn in einem Programmierteam der Programmcode ausgetauscht würde, gingen die Marker-Informationen allerdings verloren, da die Marker nicht innerhalb des Programmtextes gespeichert werden. Deshalb eignen sich die Marker nur für Projekte mit einem einzelnen Programmierer. Die Marker-Informationen gingen auch verloren, wenn eine andere Programmierumgebung als Eclipse eingesetzt werden würde.

Zur Dokumentation der Öffnung werden neue Annotationstypen für Methoden eingeführt. Der Annotationstyp @AccessUnlock enthält Informationen über die Sichtbarkeit der Methode vor und nach der Öffnung und über den Autor, der die Öffnung durchgeführt hat. Um die Sichtbarkeit einer Methode zu dokumentieren, wird der Typ AccessModifier eingeführt. AccessModifier ist eine Enumeration, die die Elemente `PRIVATE`, `DEFAULT_ACCESS`, `PROTECTED` und `PUBLIC` besitzt. Das hat gegenüber einem einfachen String den Vorteil, dass es typsicher ist. Die Angabe des Autors ermöglicht es später, dass ein Programmierer nur die Öffnungen schließen kann, die er selbst angestoßen hat.

[1] Mit Hilfe von Markern können in Eclipse Programmstellen mit Information versehen werden. Marker wurden in Abschnitt 3.2.2 besprochen.

Eine Methode, deren Sichtbarkeit vom Autor Maier von *private* nach *default* geändert wurde, erhält folgende Annotation:

```
@AccessUnlock(from = AccessModifier.PRIVATE,
              to = AccessModifier.DEFAULT_ACCESS, author = "Maier")
```

Eine Schließung der Methode wird mit einer Annotation vom Typ @AccessLock versehen. Sie enthält Informationen über die Sichtbarkeit der Methode vor und nach der Schließung und den Autor, der die Schließung durchgeführt hat. Wenn eine Methode von *public* nach *private* geändert wird, wird dies mit folgender Annotation vermerkt:

```
@AccessLock(from = AccessModifier.PUBLIC,
            to = AccessModifier.PROTECTED, author = "Maier")
```

Wenn eine Methode zweimal hintereinander geöffnet wird, z. B. von einem Autor von *private* nach *protected* und dann von einem anderen Autor von *protected* nach *public*, kann eine Methode nicht mit zwei @AccessUnlock-Annotationen versehen werden. Denn ein Element kann nicht mit mehreren Annotationen des gleichen Typs annotiert werden. Deshalb müssen weitere Annotationstypen eingeführt werden, die je eine Liste von Annotationen des gleichen Typs enthalten. Eine Annotation vom Typ @Unlocks enthält alle Annotationen vom Typ @AccessUnlock, eine Annotation vom Typ @Locks alle Annotationen vom Typ @AccessLock. Wenn eine Methode nur mit einer @AccessLock-Annotation versehen wird, wird keine @Locks-Annotation eingefügt. Gleiches gilt für @AccessUnlock und @Unlocks. Eine Methode deren Sichtbarkeit von *private* nach *protected* und dann von *protected* nach *public* geändert wurde erhält folgende Annotation:

```
@Unlocks( {
    @AccessUnlock(from = AccessModifier.PRIVATE,
                  to = AccessModifier.PROTECTED, author = "Maier"),
    @AccessUnlock(from = AccessModifier.PROTECTED,
                  to = AccessModifier.PUBLIC, author = "Gross") })
```

4.7.2 Öffnungshistorie

Die Annotationen sollen es ermöglichen, die Historie der Öffnung nachvollziehen zu können. Deshalb werden folgende Bestimmungen eingeführt:

Für eine Öffnung gilt, dass eine geöffnete Methode immer mit einer @Unlock-Annotation versehen wird. Nach einer Schließung einer Methode wird in der Öffnungshierarchie nach Öffnungen gesucht, die durch die Schließung rückgängig gemacht werden. (Es können durch eine Schließung auch mehrere Öffnungsannotationen entfernt werden.) Diese werden aus der Hierarchie entfernt. Wird eine Methode geschlossen, wird eine Öffnungsannotation dann entfernt, wenn die Sichtbarkeit vor der zugehörigen Öffnung gleich oder größer der jetzigen Sichtbarkeit war. Mit der jetzigen Sichtbarkeit ist die Sichtbarkeit nach der Schließung gemeint. Wird die Sichtbarkeit einer Methode z. B. von *private* nach *public* und wieder zurück nach *private* geändert, enthält die Methode keine Annotationen mehr. Weist die Methode nach dem Entfernen der überflüssigen Öffnungsannotationen noch mindestens eine Öffnungsannotation auf, wird die Schließung mit einer @Lock-Annotation versehen. Ansonsten entfällt die Schließungsannotation.

4.7.3 Beispiel zur Öffnungshistorie

Die Methode `m` in einer Klasse hat die Sichtbarkeit *private*. Sie wird schrittweise geöffnet, erst nach *default-access*, dann nach *public*. Nach diesen zwei Öffnungen soll die Sichtbarkeit programmgesteuert geschlossen werden. Es wird vom Programm festgestellt, dass die Sichtbarkeit nach *default* geändert werden kann. Eine Öffnungsannotation kann entfernt werden. Nach Änderungen im Programmcode wird erneut eine Schließung durchgeführt. Da festgestellt wird, dass die Methode jetzt nur in der gleichen Klasse verwendet wird, wird die Sichtbarkeit nach *private* geändert. Im Folgenden sind die Annotationen der Methode nach jedem Schritt aufgeführt.

Die Methode vor der Änderung:

```
private void m() {
}
```

Die Annotation nach Änderung der Sichtbarkeit von *private* nach *default*:

```
@AccessUnlock(from = AccessModifier.PRIVATE,
              to = AccessModifier.DEFAULT_ACCESS, author = "Maier")
void m() {
}
```

Nach der Änderung von *default* nach *public*:

```
@Unlocks( {
     @AccessUnlock(from = AccessModifier.PRIVATE,
                   to = AccessModifier.DEFAULT_ACCESS, author = "Maier"),
     @AccessUnlock(from = AccessModifier.DEFAULT_ACCESS,
                   to = AccessModifier.PUBLIC, author = "Gross") })
public void m() {
}
```

Nach der Änderung von *public* nach *default*:

```
@AccessUnlock(from = AccessModifier.PRIVATE,
              to = AccessModifier.DEFAULT_ACCESS, author = "Maier")
void m() {
}
```

Nach der Änderung von *default* nach *private*:

```
private void m() {
}
```

Bei einem weiteren Beispiel erfolgt die Änderung einer Methode `m` von *private* nach *default*, von *default* nach *public* und von *public* nach *protected*. Dadurch entsteht folgende umfangreiche Annotation:

```
@Unlocks( {
      @AccessUnlock(from = AccessModifier.PRIVATE,
                    to = AccessModifier.DEFAULT_ACCESS, author = "Maier"),
      @AccessUnlock(from = AccessModifier.DEFAULT_ACCESS,
                    to = AccessModifier.PUBLIC, author = "Gross") })
@AccessLock(from = AccessModifier.PUBLIC,
            to = AccessModifier.PROTECTED, author = "Gross")
protected void m() {
}
```

Wie in den Beispielen zu sehen ist, können recht umfangreiche Annotationen entstehen. Eine Möglichkeit die Annotationen im Programmtext auszublenden, wie dies z. B. für import-Anweisungen möglich ist, gibt es in Eclipse zurzeit nicht. Solch umfangreiche Annotationen werden aber in der Praxis nicht oft vorkommen. In den meisten Fällen dürfte eine Methode nur einmal geöffnet werden. Wenn beim Schließen die Sichtbarkeit der Methoden dann wieder auf die Sichtbarkeit vor der Öffnung gesetzt wird, fällt dann die Öffnungsannotation weg, so dass keine Annotationen übrig bleiben. Deshalb wird eine Methode, die mit mehr als zwei Annotationen versehen ist nicht häufig auftreten.

4.7.4 Autorenhierarchie

An einem größeren Softwareprojekt arbeiten in der Regel mehrere Programmierer zusammen, die meist in Teams organisiert sind. Das spätere Plugin soll es Benutzern ermöglichen, nur die Methoden zu schließen, die sie selbst oder ein Mitglied ihres Teams geöffnet haben. Deshalb ist es hilfreich, wenn zu dem Namen des Programmierers das Team angegeben wird, indem er Mitglied ist. In einem Projekt gebe es die Teams T1 und T2. Diese Teams sind wiederum in mehrere Untergruppen aufgeteilt. Ein Autor mit Namen Maier sei Mitglied in der Untergruppe U1 des Teams T1. Seine Kennung lautet dann T1.U1.Maier. Dies ermöglicht einem Benutzer im späteren Plugin, alle geöffneten Methoden zu schließen, die er nur selbst, eine bestimmte Untergruppe oder ein bestimmtes Team geöffnet hat.

Wenn eine Methode nur vom selben Autor (bzw. Team) geschlossen werden darf, gelten folgende Bestimmungen: Ein Autor darf Methoden schließen, die er selbst geöffnet hat. Er darf sie nur bis zu einer Sichtbarkeit schließen, die sie vor der Öffnung hatten. Hat ein anderer Autor die Methode weiter geöffnet als er selbst, darf der Autor die Methode nicht schließen. Die Öffnung einer Methode unterliegt keinen Einschränkungen. Ein Autor darf eine Methode jederzeit öffnen.

Folgendes Beispiel soll die oben aufgestellten Regeln verdeutlichen. Ein Autor A öffnet die Methode von *private* nach *protected*. Danach öffnet der Autor B die Methode von *protected* nach *public*. Autor A darf die Methode nun nicht schließen, da sie nach seiner Öffnung von einem anderen Autor erneut geöffnet wurde. Erst wenn die Methode wieder die Sichtbarkeit *protected* oder *default* hat, darf Autor A die Sichtbarkeit der Methode bis

nach *private* verändern. Der Autor B darf die Methode nur bis zur Sichtbarkeit *protected* schließen.

4.7.5 Minimale und maximale Sichtbarkeit

Es werden zusätzliche Annotationen eingeführt, mit denen festlegt werden kann, welche minimale bzw. maximale Sichtbarkeit eine Methode durch eine programmgesteuerte Öffnung bzw. Schließung der Klasse annehmen darf.

Wenn einem Programmierer z. B. klar ist, dass auf eine Methode nicht von außerhalb eines Paketes zugegriffen werden soll, kann er mit einer dieser Annotationen festlegen, dass die maximale Sichtbarkeit *default* sein darf. Auch eine programmgesteuerte Öffnung der Klasse würde die Sichtbarkeit der Methode dann nicht höher als *default* setzen.

Anders liegt der Fall, wenn der Programmierer weiß, dass die Methode von Subklassen benötigt werden wird, die erst später hinzugefügt werden, wie dies z. B. bei Frameworks der Fall ist. Wenn die Methode außerhalb der Klasse zurzeit nicht benötigt wird, würde eine programmgesteuerte Schließung sie auf *private* setzen und so die Benutzung in Subklassen verhindern. Hier kann der Programmierer durch eine Annotation festlegen, dass die minimale Sichtbarkeit *protected* sein soll. Ein programmgesteuertes Schließen der Klasse würde die Sichtbarkeit der Methode dann nicht kleiner als *protected* setzen.

Mit dem Annotationstyp @MinAccess kann die minimale Sichtbarkeit festgelegt werden, die eine Methode annehmen darf, mit dem Annotationstyp @MaxAccess die maximale Sichtbarkeit. Der Parameter beider Annotationen ist vom Typ AccessModifier. Eine Methode, deren maximale Sichtbarkeit mit *protected* und deren minimale Sichtbarkeit mit *default* festgelegt ist, weist folgende Annotationen auf:

```
@MaxAccess(AccessModifier.PROTECTED)
@MinAccess(AccessModifier.DEFAULT_ACCESS)
public void m(){
}
```

5 Implementierung des *Access-Modifier-Modifiers*

5.1 Allgemeines

Der *Access-Modifier-Modifier* (*AMM*) setzt die in Kapitel 2 aufgestellten Anforderungen um. Er wurde als Plugin für Eclipse 3.2 konzipiert. In diesem Kapitel werden die Details der Implementierung besprochen, einen Überblick über alle Funktionen und die Bedienung des Plugins findet sich in Kapitel 6. Nachdem der Aufbau des Plugins erläutert wurde, wird der Programmablauf für das Schließen eines Projekts besprochen. Danach werden die wichtigsten Klassen vorgestellt, wobei auch Funktionsweisen und Besonderheiten des JDT erwähnt werden, wenn sie für die Umsetzung der Klasse notwendig waren und nicht schon in Kapitel 3.2 erläutert wurden. Die Umsetzung der *Hintergrundprüfung* wird in einem eigenen Kapitel am Ende besprochen. Die Funktionsweise einer Klasse wird so detailliert beschrieben, dass der grobe Programmablauf ersichtlich wird. Für das Verständnis der genauen Funktionsweise bleibt das Studium des Quellcodes dennoch unerlässlich.

Die Standardprüfung sowie die Hintergrundprüfung weisen einige Beschränkungen auf. Die Hintergrundprüfung kann z. B. in wenigen Fällen die minimale Sichtbarkeit nicht korrekt bestimmen. Diese Beschränkungen sind in Anhang 2 bzw. Anhang 3 aufgeführt. Auf das Testen des *AMM* wird in Anhang 1 eingegangen.

Die in Eclipse mitgelieferten Refactoring-Werkzeuge bieten ein einheitliches Bedienkonzept, das unter anderem eine Vorschau auf den geänderten Code ermöglicht. Die Werkzeuge nutzen dazu das Refactoring-Framework des JDTs [Arth04]. Dieses Framework wird vom *AMM* nicht benutzt, da keine ausreichende Dokumentation des Frameworks vorlag und der Aufwand für eine Verwendung des Frameworks in keiner Relation zu dem daraus gewonnen Nutzen gestanden hätte. Die mitgelieferten Refactoring-Werkzeuge müssen teilweise Bedingungen prüfen, die auch im *AMM-Plugin* geprüft werden müssen. Da die Werkzeuge aber z. B. nicht mit Problemen umgehen können, die durch überladene Methoden hervorgerufen werden (vgl. Abschnitt 3.3), wurden alle Prüfungen selbst implementiert. Für den Kern des *AMM* ist also kein Programmcode aus den vorhandenen Refactoring-Werkzeugen übernommen worden.

Für die Umsetzung des *AMM-Plugins* wurden die Fähigkeiten des JDTs genutzt. Das JDT und auch andere Bereiche von Eclipse sind in einen öffentlichen und einen internen Codeteil aufgegliedert. Laut [Clay04] soll auf internen Code nicht von außerhalb des Plugins, in der der Code definiert ist, zugegriffen werden, da sich die Schnittstellen in einer neueren Version von Eclipse drastisch ändern könnten. Eine Klasse befindet sich in einem internen Paket, wenn der Paketname ein „internal“ enthält. Ein Beispiel für ein internes Paket ist das Paket `org.eclipse.jdt.internal.compiler`. Alle anderen Klassen sind öffentlich und dürfen von außerhalb aufgerufen werden. Für diese öffentliche APIs gibt es feste Regeln, wie sich die Schnittstellen in einer neuen Version ändern dürfen. Deshalb wurde darauf geachtet, im *AMM-Plugin* möglichst nur öffentlich zugängliche Schnittstellen zu benutzen. Dies konnte nicht immer eingehalten werden. So werden einige statische Methoden der Klasse `org.eclipse.jdt.internal.corext.dom.Bindings` eingesetzt, für die es keine öffentlich zugängliche Alternative gibt.

5.2 Aufbau

Die UML-Klassendiagramme in Bild 5.1 sowie Bild 5.2 zeigen die wichtigsten Klassen des *AMM-Plugins* und ihre Beziehungen. Das Paket `de.gky.accessmodifiermodifier.core` enthält die eigentliche Anwendungslogik, die für das für Schließen und Öffnen notwendig ist. Die Klassen des Unterpaketes `problemcheckers` prüfen die Regeln aus den Abschnitten 4.4.2 sowie 4.5.2. Dazu verwenden diese die Hilfsklassen des Unterpaketes `util`.

Die *Hintergrundprüfung* wird von Klassen des Paketes `de.gky.accessmodifiermodifier.backgroundcheck` ermöglicht. Die Klassen, die benötigt werden, damit ein Benutzer Funktionen des *AMM-Plugins* aus Eclipse aufrufen kann finden sich im Paket `de.gky.accessmodifiermodifier.popup.actions`.

Im Paket `de.gky.accessmodifiermodifier` befindet sich der „Aktivator“ und eine Klasse mit Konfigurationseinstellungen. Das nicht abgebildete Paket `de.gky.accessmodifiermodifier.gui` enthält die Dialoge und Preferences-Seiten. Für die Erstellung der Dialoge wurde das Programm „Jigloo“ [Jigl06] verwendet.

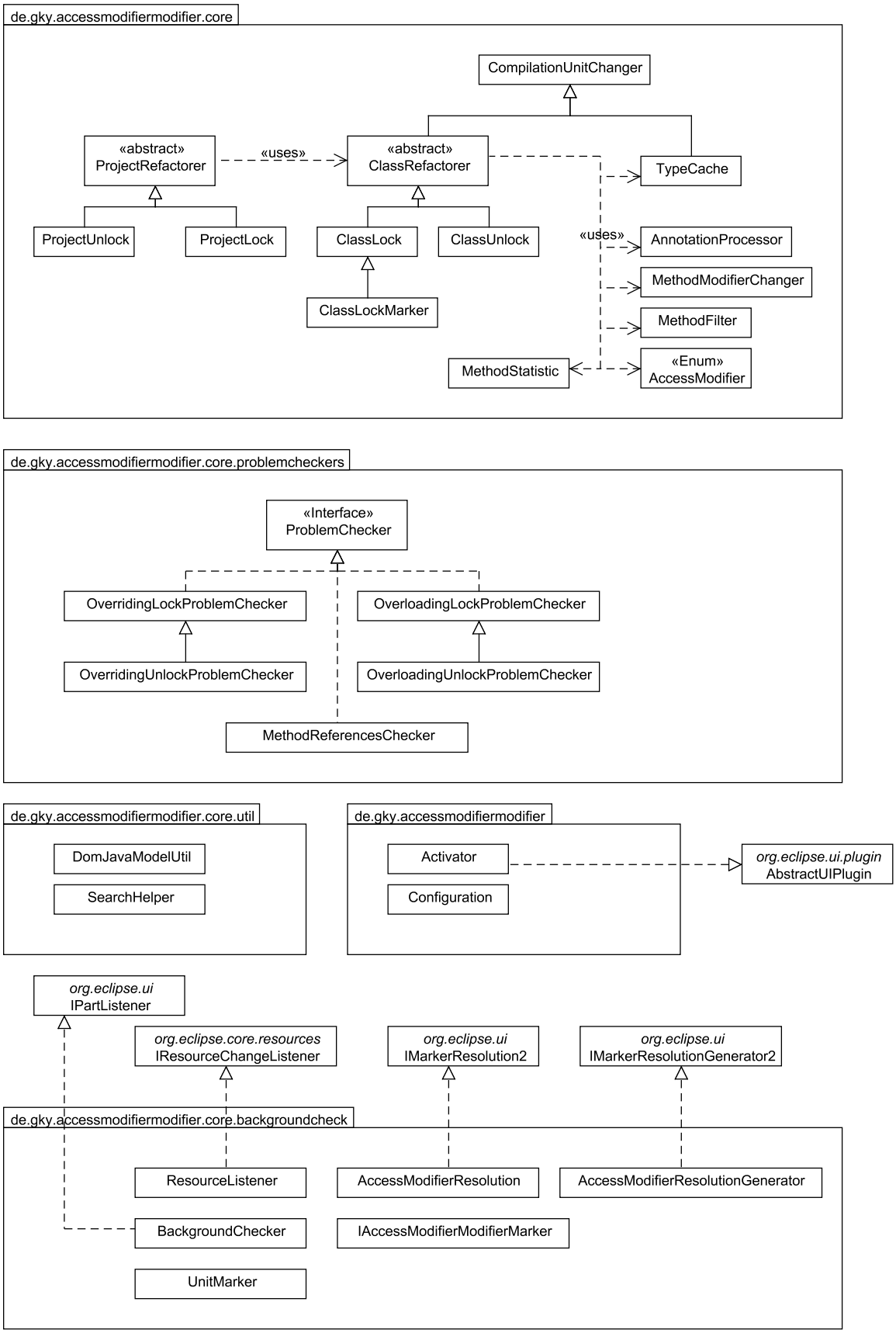

Bild 5.1: Klassendiagramm des *AMM-Plugins* (Teil 1)

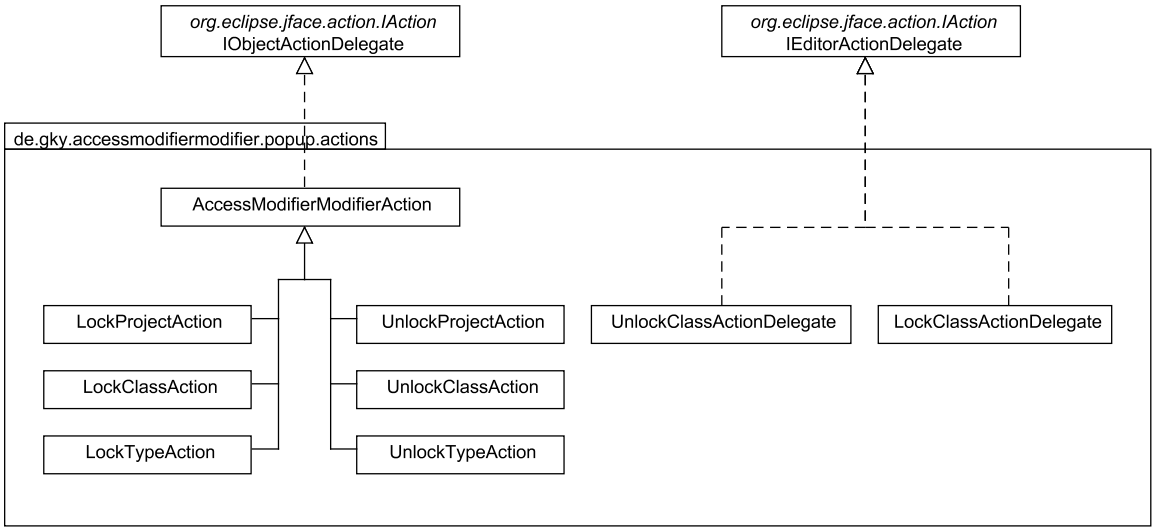

Bild 5.2: Klassendiagramm des *AMM-Plugins* (Teil 2)

5.3 Programmablauf

5.3.1 Schließen eines Projekts

Der Benutzer markiert im Package-Explorer das zu schließende Projekt und ruft mit der rechten Maustaste das Popup-Menü auf. Daraus wählt er die Menüoption „lock Projekt…“ aus. Eclipse erzeugt daraufhin eine Instanz der Klasse `LockProjectAction` und ruft ihre run-Methode auf. Eine Instanz der Klasse `ProjectLock` wird erzeugt, die alle Compilation-Units (diese sind vom Typ `ICompilationUnit`) eines Projekts schließt. Zuerst wird dem Benutzer ein Dialog angezeigt, mit dem Einstellungen vorgenommen werden können. Die Optionsmöglichkeiten sind in Abschnitt 6.2.3 beschrieben. Sie werden als Eclipse-Preferences gespeichert, so dass die Einstellungen erhalten bleiben. Ist eine `ICompilationUnit` an der Reihe, wird für die oberste Klasse (top-level Class) der weiteste Supertyp bestimmt, der auch in einer `ICompilationUnit` des Projekts, also als Quellcode, vorliegt. Dieser Typ wird einer Instanz der Klasse `ClassLock` übergeben. In der Klasse `ClassLock` werden zuerst die Vorbedingungen mit Hilfe der Klassen `OverridingLockProblemChecker` und `OverloadingLockProblemChecker` geprüft. Dann wird weiter die Trial-and-Error-Methode angewendet. Kann die Sichtbarkeit einer Methode reduziert werden, wird der Zugriffsmodifikator mit Hilfe der Klasse `MethodModifierChanger` geändert werden. Für die Auswertung und Anpassung der Annotationen ist die Klasse `AnnotationProcessor` zuständig. Da die Klasse

`ClassLock` ein Subtyp von `CompilationUnitChanger` ist, können mit ihr die vorgenommenen Änderungen in die entsprechende Datei zurück geschrieben werden.

5.3.2 Öffnen eines Projekts

Das Öffnen eines Projekts läuft entsprechend dem Schließen ab. Die verwendeten Klassen weisen hier statt der Bezeichnung „Lock“ die Bezeichnung „Unlock“ auf. So werden statt den Klassen `ProjectLock` und `ClassLock` die Klassen `ProjectUnlock` und `ClassUnlock` verwendet.

5.4 Funktionsweise der Klassen

5.4.1 Überblick über die Implementierung der Regeln für das Schließen und Öffnen

Die Klasse `OverridingLockProblemChecker` prüft die Regeln S.1, S.2, S.3, S.10 und S.12 aus Abschnitt 4.4.2. Die Regel S.3 wird dabei nicht explizit geprüft: wenn die Regel S.12 eingehalten wird, wird auch die Regel S.3 eingehalten. Die Klasse `OverloadingLockProblemChecker` ist für die Prüfung der Regel S.13 zuständig. Die Einhaltung der Regeln S.4 bis S.9 wird durch die Klasse `MethodReferencesChecker` sichergestellt. Die Regel S.11 wird direkt in der Klasse `ClassLock` bzw. `ClassLockMarker` geprüft.

Die Regeln Ö.1 und Ö.2 aus Abschnitt 4.5 werden nicht programmgesteuert geprüft. Wenn sie verletzt werden, tritt ein Übersetzungsfehler auf, der vom Trial-and-Error-Ansatz erkannt wird. Die Klasse `OverridingUnlockProblemChecker` prüft die Regel Ö.3, die Klasse `OverloadingUnlockProblemChecker` die Regel Ö.4.

5.4.2 Die Klasse ProjectLock und ProjectUnlock

Mit der Klasse `ProjectLock` kann ein ganzes Java-Projekt geschlossen, mit der Klasse `ProjectUnlock` kann es geöffnet werden. Die Klassen sind von der abstrakten Klasse `ProjectRefactorer` abgeleitet. Es wird davon ausgegangen, dass sich alle relevanten Klassen in dem zu refaktorisierenden Projekt befinden, also dass das Projekt nicht von

anderen Projekten abhängt. Wird die Methode `refactorProjectWithDialog` aufgerufen, wird dem Benutzer zuerst ein Dialog anzeigt, mit dem er Einstellungen vornehmen kann.

Die Klassen eines Projekts werden nacheinander refaktorisiert. Einer Instanz der Klasse `ClassLock` bzw. `ClassUnlock` wird immer nur ein einzelner zu schließender bzw. öffnender Typ übergeben. Die Instanz ermittelt dann die zugehörige Hierarchie und liefert eine Liste aller Typen zurück, die sie refaktorisiert hat. Wenn dann in einer Compilation-Unit eine Klasse ermittelt wird, die schon geschlossen wurde, wird sie nicht erneut geschlossen.

Die Schließung bzw. Öffnung wird im UIThread von Eclipse ausgeführt. Der Benutzer kann das Refactoring mit Hilfe des angezeigten Progress-Monitors abbrechen.

5.4.3 Die Klasse ClassLock

Mit der Klasse `ClassLock` kann eine einzige Klasse oder eine ganze Klassenhierarchie geschlossen werden. Sie ist von der abstrakten Klasse `ClassRefactorer` abgeleitet. Mit dem Aufruf der Methode `refactorClassHierarchy` wird die übergebene Klasse, ihre Subtypen und ihre Supertypen geschlossen. Die Klassen werden in absteigender Reihenfolge geschlossen werden, d. h. eine Klasse wird immer vor seinen Subtypen geschlossen. Die Supertypen und Subtypen werden mit Hilfe des JavaModels ermittelt. Sie sind Instanzen vom Typ `IType` und werden absteigend geordnet.

Bevor mit dem Refaktorisieren einer Klasse begonnen wird, wird der Inhalt der Compilation-Unit, in der sich die Klasse befindet, zwischengespeichert. Während des Refactorings können Exceptions auftreten. Beim Einlesen und Zurückschreiben können z. B. Ausnahmen vom Typ `CoreException` auftreten, wenn auf die benötigten Dateien nicht zugegriffen werden kann. Tritt eine Exception auf, wird sie abgefangen und es wird der zwischengespeicherte Programmtext wiederhergestellt. Das Refactoring wird dann an dieser Stelle abgebrochen.

Zuerst werden alle Methoden des Typs und der inneren Klassen ermittelt, wozu die Klasse `MethodFilter` verwendet wird. Anonyme Klassen werden nicht berücksichtigt. `MethodFilter` benutzt einen `ASTVisitor`, um die Methoden einer Klasse zu finden. Es

kann hier eingestellt werden, ob auch innere Klassen durchsucht werden sollen. Für jede so gefundene Methode wird ein Objekt vom Typ `MethodStatistic` erzeugt, das alle Daten speichert, die für die Methode im folgenden Refaktorisierungsprozess benötigt werden. Für das Schließen einer Klasse ist das Feld `possibleMinVisibility` der Klasse `MethodStatistic` relevant. Es enthält die minimale Sichtbarkeit, die die Methode annehmen darf und wird mit der Sichtbarkeit *private* initialisiert. Wenn eine der folgenden Prüfungen eine größere minimale Sichtbarkeit ergibt, wird das Feld entsprechend angepasst.

Für jede gefundene Methode wird nun die minimale Sichtbarkeit ermittelt. Wenn die Annotationen `MinAccess` und `MaxAccess` ausgewertet werden sollen, wird geprüft, ob die Methode eine solche Annotation aufweist. Die minimale Sichtbarkeit ist dann mindestens so groß wie die Sichtbarkeit, die in der `MinAccess`-Annotation festgelegt ist. Das Auslesen der Annotation wird von der Klasse `AnnotationProcessor` erledigt. Der Benutzer kann in den Einstellungen festlegen, dass eine Methode nur geschlossen werden darf, wenn sie vorher geöffnet worden ist, also wenn eine Öffnungsannotation vorliegt. Mit der Klasse `AnnotationProcessor` kann dann geprüft werden, ob eine Öffnungsannotation vorliegt. Wenn keine Öffnungsannotation vorliegt wird die Methode dann nicht geschlossen.

Weiter wird geprüft, ob die Regeln S.12 und S.13 aus Abschnitt 4.4.2 eingehalten werden. Sie können vor einer Änderung der Sichtbarkeit geprüft werden. Für die Prüfung werden Instanzen der Klassen `OverridingLockProblemChecker` und `OverloadingLockProblemChecker` eingesetzt, die beide vom Typ `ProblemChecker` sind. Klassen, die die Vorbedingungen prüfen, müssen das Interface `ProblemChecker` implementieren. Alle Instanzen vom Typ `ProblemChecker` werden in der Liste `checkers` gespeichert. Es können so leicht neue Prüfungen hinzugefügt werden. Bevor die Sichtbarkeit einer Methode geändert wird, ruft `ClassLock` die Methode `checkPreconditions` jeder Instanz in dieser Liste auf. (Nach einer Änderung wird die Methode `checkPostconditions` aufgerufen mit der die Nachbedingungen getestet werden können. Alle im Plugin vorhandenen Prüfungsklassen können die jeweiligen Bedingungen aber vor einer Änderung prüfen.)

Ist nach dieser Prüfung die minimal mögliche Sichtbarkeit immer noch kleiner als die ursprüngliche Sichtbarkeit der Methode, wird weiter mit dem Trial-and-Error-Verfahren geprüft. Beim Trial-and-Error-Verfahren wird die Sichtbarkeit der Methode schrittweise reduziert und ein nach jeder Änderung ein inkrementeller Compiler-Build angestoßen. Die Änderung des Zugriffsmodifikators im Programmcode erfolgt mit Hilfe der Klasse `MethodModifierChanger`. Meldet der Compiler nach einem Build einen Fehler, dann ist die minimal mögliche Sichtbarkeit um eine Stufe höher als die gerade geprüfte Sichtbarkeit. Tritt kein Compilerfehler auf, wird die nächst kleinere Sichtbarkeit geprüft. Bei einem Compilerfehler wird die Programmstelle an der der Fehler aufgetreten ist vom JDT mit einem Marker des Typs `org.eclipse.jdt.core.problem` versehen. Da der Fehler auch außerhalb der Klasse auftreten kann, in der sich die geänderte Methode befindet, wird im gesamten Projekt nach Markern dieses Typs gesucht. Der Marker vom Typ `problem` wird sowohl für Warnungen, als auch für Compilerfehler benutzt. Ist das Attribut `severity` des Markers gleich eins, liegt eine Warnung vor, bei zwei ein Compilerfehler.

Nach der Prüfung einer Methode wird die Änderung der Sichtbarkeit wieder rückgängig gemacht indem der ursprüngliche, vorher zwischengespeicherte Programmtext wieder eingelesen wird. Diese Vorgehensweise hat den großen Vorteil, dass der AST nicht bei jeder zu untersuchende Methode neu aufgebaut werden muss.

Nachdem alle Methoden geprüft wurden, wird die Sichtbarkeit jeder Methode auf die minimale ermittelte Sichtbarkeit gesetzt und die Annotationen der Methoden angepasst. Dazu wird wieder die Klasse `MethodModifierChanger` verwendet. Laut Javadoc sollten die durch einen `ASTRewrite` durchgeführten Änderungen entsprechend den eingestellten Standardregeln formatiert werden. Bei Annotation und den geänderten Zugriffsmodifikatoren funktioniert dies nicht zuverlässig. Der Programmtext wird deshalb erneut formatiert. Mit der statischen Methode `createCodeFormatter` der Klasse `org.eclipse.jdt.core.ToolFactory` erhält man ein Objekt vom Typ `CodeFormatter`, mit dem ein Dokument formatiert werden kann.

Wie in Abschnitt 4.3.4 besprochen, kann es beim Auftreten von überladenen Methoden erforderlich sein, Klassen mehrmals zu prüfen. Methoden, die wegen Überladungsproblemen nicht geändert werden konnten, werden in einer Liste gespeichert. Nachdem alle

Klassen der Hierarchie geprüft worden sind, werden die Klassen[1], in denen die Methoden der Liste definiert sind, erneut geprüft. Wenn das ursprüngliche Problem für eine Methode der Liste nun nicht mehr besteht, kann sie geschlossen werden. Wenn eine Methode dieser Liste eine andere Methode dieser Liste ersetzen kann, wird sichergestellt, dass sie vor dieser Methode geschlossen wird. (Um diese Bedingung einzuhalten, kann es vorkommen, dass eine Klasse mehrmals geschlossen werden muss.)

5.4.4 Die Klasse ClassUnlock

Mit der Klasse `ClassUnlock` kann eine einzelne Klasse oder eine ganze Klassenhierarchie geöffnet werden. Da `ClassUnlock` ebenfalls von der Klasse `ClassRefactorer` abgeleitet wurde, entspricht der Ablauf beim Öffnen weitgehend dem Ablauf beim Schließen, weshalb hier nur die Unterschiede aufgeführt werden. Die Vorbedingungen sind hier die Regeln Ö.3 und Ö.4 aus Abschnitt 4.5 und werden mit Instanzen der Klassen `OverridingUnlockProblemChecker` und `OverloadingUnlockProblemChecker` geprüft. Beim Trial-and-Error-Verfahren wird die Sichtbarkeit einer Methode zuerst auf *public* gesetzt. Wenn nach der Änderung ein Übersetzungsfehler auftritt, wird schrittweise die jeweilige nächst niedrigere Sichtbarkeit geprüft. Im Gegensatz zum Schließen erfolgt keine erneute Prüfung der Klassen, wenn Probleme durch überladene Methoden aufgetreten sind. (Der Schwerpunkt der vorliegenden Arbeit ist das Schließen einer Klasse, weshalb hierfür mehr Aufwand betrieben wird.)

5.4.5 Die Klasse TypeCache

Mit der Klasse `TypeCache` können die ASTs zwischengespeichert werden, die für die Prüfung der Vorbedingungen benötigt werden. Benötigt werden z. B. die ASTs der Compilation-Units, in denen sich die Subtypen der zu refaktorisierenden Klasse befinden. Klassen, die die Vorbedingungen prüfen, also das Interface `ProblemChecker` implementieren, brauchen teilweise Zugriff auf die ASTs derselben Compilation-Units. Ein AST einer Compilation-Unit wird von einer Instanz des Typs `TypeCache` aufgestellt und dann von ihr zwischengespeichert. Wird der AST erneut benötigt, wird der zwi-

[1] Tatsächlich müssten nur die Methoden aus der Liste erneut geprüft werden.

schengespeicherte AST zurückgeliefert. Mit der Methode `getAllSubtypeBindings(IType)` kann der Cache auch alle Subtypen eines Typs direkt ermitteln und sie als `ITypeBindings` zurückliefern.

Der Aufbau eines ASTs ist, besonders wenn die so genannten Bindungen berücksichtigt werden sollen, recht aufwändig.

Für den Aufbau eines ASTs mit Bindungen wurden, bei einem Rechner mit 1 GHz, Zeitspannen von 10 ms bis weit über 300 ms gemessen. Der Aufwand für die Erstellung des ASTs ist nicht nur höher, je umfangreicher die einzulesende Compilation-Unit ist: Um die Bindungen aufzustellen, müssen auch die Typen, die in der Compilation-Unit Verwendung finden, eingelesen werden. Dies sind z. B. alle Typen, die als Parametertypen vorkommen. Diese Typen verwenden wieder andere Typen und diese wiederum andere. Deshalb sollten die Bindungen im Allgemeinen beim Aufbau eines ASTs nur miterzeugt werden, wenn sie auch tatsächlich benötigt werden.

5.4.6 Die Klasse AnnotationProcessor

Der `AnnotationProcessor` wertet die Methoden-Annotationen, die für das Schließen und Öffnen vorgesehen sind, aus. Die vorhandenen Annotationen `@AccessLock` und `@AccessUnlock` werden in einer Liste gespeichert und später mit Hilfe der inneren Klasse `ParsedAnnotation` ausgewertet.

Wenn die Sichtbarkeit der Methode geändert wird, werden die Annotationen, wie in Abschnitt 4.6 beschrieben, angepasst. Diese Anpassungen werden mit Hilfe einer Instanz vom Typ `ASTRewrite` durchgeführt. Werden neue Annotationen der Typen `AccessLock` und `AccessUnlock` benötigt, werden diese von Grund auf neu erzeugt. (Alternativ könnte man auch die Annotation als String aufstellen und mit einem `ASTParser` parsen.) Die neuen Annotationen müssen in den AST an der richtigen Stelle eingefügt werden. Annotationen des *AMM-Plugins* werden immer vor anderen eventuell vorhandenen Annotationen, wie z. B. `@Override`, eingefügt.

5.4.7 Implementierung der Prüfung auf Überschreibungsprobleme – die Klasse OverridingLockProblemChecker

Mit der Klasse `OverridingLockProblemChecker` kann geprüft werden, ob bei der Schließung einer Methode die Regeln S.1, S.2, S.3, S.10 und S.12 aus Abschnitt 4.4.2 eingehalten werden. Die Prüfung der Regeln S.1, S.2, S.3 und S.10 ist nur für die *Hintergrundprüfung* unbedingt notwendig. Werden diese Regeln verletzt, tritt immer ein Übersetzungsfehler auf, der vom Trial-and-Error-Verfahren erkannt wird. Die zusätzliche Prüfung hilft aber auch das Trial-and-Error-Verfahren zu beschleunigen, da eine Verletzung einer dieser Regeln erkannt wird, ohne dass ein Build angestoßen werden muss. Die zusätzliche Prüfung selbst verursacht nur einen minimalen zusätzlichen Aufwand.

Die einfachste Vorgehensweise um sicherzustellen, dass die Regel S.12 eingehalten wird, ist, die Anzahl der Methoden zu bestimmen, die die Methode vor der Schließung überschreiben. Nach der Schließung wird dann geprüft, ob sich die Anzahl der überschreibenden Methoden geändert hat. Dieses Verfahren hat den Nachteil, dass es nur durchgeführt werden kann, wenn dabei die Sichtbarkeit geändert werden kann. Es kann deshalb für eine Prüfung im Hintergrund nicht angewendet werden. Da sowohl vor als auch nach der Schließung der AST der Subklassen aufgebaut werden muss, ist es zudem relativ aufwändig.

Gesucht wird also ein Verfahren, das die Prüfung der Regel S.12 durchführen kann, bevor die Methode geändert wurde. Nachfolgend werden Bedingungen aufgestellt, wann eine Methode m_2 eine Methode m_1 nach der Änderung weiterhin überschreibt. Wenn die Sichtbarkeit der Methode m_1 *private* wird, wird m_1 nicht mehr dynamisch gebunden und kann deshalb nicht mehr überschrieben werden. Wenn die Sichtbarkeit von m_1 *protected* wird, wird sie von m_2 weiterhin überschrieben. Denn eine Methode mit Sichtbarkeit *protected* ist in einem Subtyp sichtbar, wenn sie es auch vor der Änderung war.

Aufwändiger ist der Fall, wenn die Sichtbarkeit nach *default* geändert wird. Die Methode m_1 befinde sich in Klasse C_1, die überschreibende Methode m_2 befinde sich in Klasse C_2. Wenn sich die Klassen C_1 und C_2 in unterschiedlichen Paketen befinden, ist m_2 nach der Änderung nicht mehr in C_2 sichtbar und wird deshalb nicht mehr von m_1 überschrieben. Wenn sich C_1 und C_2 im gleichen Paket befinden, überschreibt m_2 die Methode m_1 auch nach der Änderung, bis auf folgenden seltenen Fall: Wenn es eine Klasse C_n gibt, die ein

Subtyp von C_1 und ein Supertyp von C_2 ist, und diese Klasse sich in einem anderen Paket befindet als C_1 und C_2, ist eine Methode mit der Sichtbarkeit *default*, die sich in C_1 befindet, nicht in C_2 sichtbar, auch wenn C_1 und C_2 sich im selben Paket befinden. Es wird also geprüft, ob alle Klassen die sich in der Subtyp-Hierarchie zwischen C_1 und C_2 befinden, sich auch in dem gleichen Paket wie C_1 befinden.

Implementierung der Regel S.12

Mit Hilfe einer Instanz der Klasse `TypeCache` werden alle Subtypen und ihre ASTs ermittelt. Für jede Methode m_i einer Subklasse wird untersucht, ob sie die zu ändernde Methode m_1 überschreibt. Mit der statischen Methode `findOverriddenMethod(IMethodBinding overriding, boolean testVisibility)` der Klasse `org.eclipse.jdt.internal.corext.dom.Bindings` kann die überschriebene Methode einer Methode ermittelt werden. Dann wird untersucht, ob eine so gefundene Methode mit der zu ändernden Methode m_1 übereinstimmt. Dies ist der Fall, wenn die zwei `IMethodBinding`-Objekte die gleiche Methode repräsentieren. Es wird nun untersucht, ob die vorher überschreibende Methode die geänderte Methode nicht mehr überschreibt. Dabei werden die in diesem Abschnitt aufgestellten Bedingungen verwendet.

Die Methoden, die verglichen werden sollen, befinden sich in unterschiedlichen ASTs. Es wird vom JDT nicht garantiert, dass zwei Variabeln vom Typ `IMethodBinding` aus unterschiedlichen ASTs, die die gleiche Methode repräsentieren, auch das gleiche Objekt referenzieren. Um zu bestimmen, ob zwei Methoden aus unterschiedlichen ASTs gleich sind, können die Keys der Methoden verglichen werden. Jedes Objekt vom Typ `IBinding` besitzt die Methode `getKey()` mit der sein Key ermittelt werden kann. Dieser Schlüssel ist vom Typ `String`. Wie genau der Key eines Objekts vom Typ `IBinding` aufgebaut ist, ist nicht genau spezifiziert. Jedes Objekt vom Typ `IMethodBinding`, das eine bestimmte Methode repräsentiert, liefert aber immer den gleichen Key zurück. Um die Keys zu vergleichen kann auch die Methode `IBinding#isEqualTo` verwendet werden.

Auch die Objekte vom Typ `IMethod` haben eine Methode `getKey()`, mit der ein String-Objekt zurückgegeben wird, das die Methode eindeutig identifiziert. Im Javadoc der Methode `getKey()` entsteht der Eindruck, dass die Keys eines Objekts vom Typ

`IMethod` und eines Objekts vom Typ `IMethodBinding` immer gleich sind, wenn sie die gleiche Methode repräsentieren. Es gibt allerdings Fälle, bei denen dies nicht zutrifft. Auch die Keys eines Objekts vom Typ `IType` und eines Objekts vom Typ `ITypeBinding`, die den gleichen Typ repräsentieren, sind nicht immer gleich.

Implementierung der Regeln S.1 und S.2

Die Prüfung der Regel S.2 ist recht einfach. Es werden alle durch m_1 überschriebenen oder implementierten Methoden ermittelt und jeweils geprüft, ob die geänderte Sichtbarkeit von m_1 kleiner als die Sichtbarkeit der überschriebenen Methode ist. Ist dies der Fall, wird nach der Änderung ein Übersetzungsfehler auftreten; ansonsten ist die Änderung zulässig.

Eine Methode eines Interfaces hat implizit die Sichtbarkeit *public*. Da eine Interface-Methode nicht explizit den Zugriffsmodifikator *public* aufweisen muss, ist der Test, ob eine Interface-Methode den Zugriffsmodifikator *public* aufweist, nicht immer positiv. Bei einem Interface darf man sich deshalb nicht auf die Prüfung der Zugriffsmodifikatoren verlassen. Wenn sich die überschriebene Methode in einem Interface befindet[1], kann stattdessen immer die Sichtbarkeit *public* angenommen werden.

5.4.8 Implementierung der Prüfung auf Überladungsprobleme – die Klasse OverloadingLockProblemChecker

Die Klasse `OverloadingLockProblemChecker` stellt die Einhaltung der Regel S.13 sicher. Die Methode m_1, deren Sichtbarkeit reduziert werden soll, befinde sich in Klasse C_1. Zuerst wird geprüft, ob in einem Supertyp von C_1 eine Methode m_2 vorhanden ist, die m_1 überlädt und ein Problem verursachen kann. Die Supertypen von C_1 werden aufsteigend durchsucht. Wenn eine überladene Methode gefunden wurde, muss zuerst untersucht werden, ob sie in der Klasse C_1 überhaupt sichtbar ist. (Zudem ergibt sich aus Regel S.13, dass nur Methoden Probleme verursachen können, die nach der Änderung eine höhere Sichtbarkeit aufweisen als die Methode m_1.) Wenn sie sichtbar ist, muss weiter geprüft werden, ob m_2 die Methode m_1 ersetzen kann. Dafür müssen die Parametertypen

[1] Genau genommen wird die Methode in diesem Fall implementiert und nicht überschrieben.

von m_1 und m_2 verglichen werden. Mit der Methode `ITypeBinding#isSubType-Compatible(ITypeBinding)` kann untersucht werden, ob ein Typ ein Subtyp eines anderen Typs ist. Die Dokumentation der Methode, die Javadoc, ist hier nicht ganz korrekt. Es heißt hier, dass Subtypbeziehung berücksichtigt werden, so wie sie in der Java Spezifikation [Gosl05] in Abschnitt 4.10 definiert sind. Die dort beschriebene Subtypbeziehung zwischen Primitiven wird allerdings von der Methode nicht berücksichtigt. Die Prüfung dieser Beziehung muss deshalb selbst implementiert werden. Beim Vergleich der Parameter muss zusätzlich berücksichtigt werden, dass ein primitiver Typ durch einen Wrappertyp und ein Wrappertyp durch einen primitiven Typ ersetzt werden kann.

Wenn nach Problemmethoden in einem Subtypen von C_1 gesucht wird, muss unter anderem geprüft werden, ob die Methode m_1 in dem Subtyp vor und nach der Änderung sichtbar ist. Die Methoden m_1 und m_2, d. h. die Instanzen vom Typ `IMethodBinding`, befinden sich zudem in verschiedenen ASTs. Deshalb muss im AST von Methode m_2 das Objekt vom Typ `IMethodBinding` gefunden werden, das die Methode m_1 repräsentiert. Dies ist notwendig, da mit der Methode `ITypeBinding#isSubTypeCompatible` nur Typen geprüft werden können, die aus dem gleichen AST stammen. Die Prüfung der beiden Methoden läuft dann genau wie bei den Supertypen ab.

5.4.9 Die Klasse OverloadingUnlockProblemChecker

Die Klasse `OverloadingUnlockProblemChecker` prüft die Regel Ö.4. Der Ablauf der Prüfung entspricht dem der Klasse `OverloadingLockProblemChecker`.

5.4.10 Die Klasse OverridingUnlockProblemChecker

Die Klasse `OverridingUnlockProblemChecker` prüft die Einhaltung der Regel Ö.3. Da diese Regel nur durch eine überschreibbare Methode m_1 verletzt werden kann, müssen Konstruktoren oder Methoden, die *final* oder *static* sind, nicht untersucht werden. Wenn eine Methode mit Sichtbarkeit *protected* geöffnet wird, kann die Regel ebenfalls nicht verletzt werden. Denn durch eine solche Öffnung kann sich die Menge der Methoden, die m_1 überschreiben, nicht ändern.

Für alle anderen Methoden werden die Subtypen nach Methoden m_2 durchsucht, die eine override-äquivalente Signatur aufweisen. Wenn eine Methode m_2 die Methode m_1 nicht überschreibt, wird ermittelt, ob dies nach der Sichtbarkeitsänderung der Fall ist. Dafür muss untersucht werden, ob die Methode m_1 nach der Änderung in dem Subtyp, in dem sich m_2 befindet, sichtbar ist.

5.5 Implementierung der *Hintergrundprüfung*

5.5.1 Ablauf der *Hintergrundprüfung*

Wird der Plugin gestartet, wird eine neue Instanz der Klasse `BackgroundChecker` erzeugt und als Listener beim `IPartService` angemeldet. Die *Hintergrundprüfung* ist anfangs noch nicht aktiviert. Sie wird erst aktiviert, wenn der Benutzer diese Option auf der Präferenzseite des Plugins ausgewählt hat. Die `BackgroundChecker`-Instanz wird darüber informiert und startet die *Hintergrundprüfung* für die Compilation-Unit des gerade aktiven Editors. Zur Prüfung wird eine Instanz der Klasse `ClassLockMarker` verwendet. Kann die Sichtbarkeit einer Methode reduziert werden, wird sie mit einem Marker vom Typ `AccessModifierModifier.accessModifierFixMarker` versehen, der auch im Editor sichtbar ist. Klickt der Benutzer auf diesen Marker, wird ihm ein QuickFix angeboten. Die Klassen, die für die Implementierung des QuickFix zuständig sind befinden sich im Paket `de.gky.accessmodifiermodifier.backgroundcheck`. Der QuickFix benutzt eine Instanz der Klasse `ClassLockMarker` um die Sichtbarkeit der Methode zu reduzieren und die Annotationen anzupassen.

5.5.2 Die Klasse BackgroundChecker

Die Klasse `BackgroundChecker` implementiert das Interface `IPartListener`. Wenn sich ein `IPartListener` bei einem `IPartService` angemeldet hat, wird er von „Part-Lifecycle-Events“ unterrichtet. Eine Referenz auf eine Instanz des Typs `IPartListener` erhält man mit der Anweisung: `PlatformUI.getWorkbench().getActiveWorkbenchWindow().getPartService()`. Die implementierte Methode `partActivated` des `IPartListeners` wird aufgerufen, wenn eine Workbenchkomponente aktiviert wird. Wenn es sich bei der Komponente um einen EditorPart handelt, wird in der

Methode eine eventuell laufende *Hintergrundprüfung* abgebrochen und eine neue Prüfung gestartet. Vor jeder Prüfung werden aus der aktuellen Compilation-Unit alle vom Plugin gesetzten Marker entfernt. Die Prüfung wird in einem Job im Hintergrund ausgeführt. Für die Prüfung wird eine Instanz der Klasse `ClassLockMarker` eingesetzt. Der Job wird mit der niedrigsten möglichen Priorität ausgeführt, damit ein Benutzer durch die *Hintergrundprüfung* nicht gestört wird. In der Progress-View, das die in Eclipse ablaufenden Operationen anzeigen kann, ist der Job sichtbar.

Ein Job kann in Eclipse eingesetzt werden, wenn ein Programmteil in einem neuen Thread ausgeführt werden soll. Gegenüber einfachen Threads bietet ein Job mehrere Vorteile. So können den Jobs verschiedene Prioritäten zugeordnet werden, mit denen Eclipse die Ausführung der Jobs und der restlichen Systemtätigkeit koordinieren kann. Ein Progress-Monitor kann zudem den Fortschritt eines Jobs anzeigen. [Danj04]

5.5.3 Die Klasse ClassLockMarker

Mit der Klasse `ClassLockMarker` kann die Prüfung auf minimale Sichtbarkeit im Hintergrund durchgeführt werden. Alle Methoden der übergebenen Compilation-Unit vom Typ `ICompilationUnit` werden geprüft. Eine Methode, deren Sichtbarkeit reduziert werden kann, wird mit einem Marker vom Typ `de.gky.accessModifierModifier.accessModifierFixMarker` versehen. Für die Suche nach Problemen durch überladene oder überschriebene Methoden werden die Klassen `OverloadingLockProblemChecker` und `OverridingLockProblemChecker` verwendet. Methodenaufrufe werden von der Klasse `MethodReferencesChecker` geprüft.

5.5.4 Die Klasse MethodReferencesChecker

Um nach Methodenaufrufen zu suchen, werden die Suchmöglichkeiten des JDT eingesetzt. Dies ist die schnellste und Arbeitsspeicher-sparende Möglichkeit, um nach Referenzen zu suchen. Die Suche nach Referenzen einer Methode bleibt dennoch recht aufwändig. Auf einem Rechner mit 1 GHz wurden bei Projekten mittlerer Größe für die Suche nach allen Referenzen einer Methode Zeitspannen von wenigen Millisekunden bis zu

mehreren Sekunden gemessen. Die Suchdauer hängt auch von der Größe des Projekts ab und auch, ob Eclipse schon einen Suchindex aufgebaut hat.

Um nach Referenzen von Methoden zu suchen, wird die Klasse `org.eclipse.jdt.core.search.SearchEngine` benutzt. Wenn mit der SearchEngine nach Referenzen einer Methode gesucht wird, werden nicht nur Referenzen der Methode gefunden, sondern auch Referenzen von Methoden, die sie überschreiben. In einigen Fällen kann die minimale Sichtbarkeit deshalb nicht richtig bestimmt werden, wie folgendes Beispiel zeigt.

```
package paket1;

public class A {
    public void m(){
    }
}

package paket1;

public class B extends A{
    public void m(){
    }
}

package paket2;

import paket1.A;
import paket1.B;
public class Z {
    public void methodInZ(){
         B b = new B();
         b.m();
    }
}
```

Für den obigen Programmcode soll die minimale Sichtbarkeit der Methode `A#m` gefunden werden. Die Sichtbarkeit der Methode `A#m` könnte in *default* geändert werden, da es auf sie keine Referenz außerhalb des gleichen Paketes gibt und sie, nach ihrer Änderung, von `B#m` weiterhin überschrieben werden muss. Die SearchEngine findet in der Klasse `Z` für `A#m` eine Referenz, da die Suche auch alle Referenzen von überschreibenden Methoden

zurückliefert. Allerdings wird in `methodInZ` nicht `A#m` sondern `B#m` aufgerufen. Da fälschlicherweise festgestellt wurde, dass `A#m` in einer Klasse außerhalb des gleichen Paketes aufgerufen wurde und die Klasse `Z` auch kein Subtyp von `A` ist, muss die Sichtbarkeit von `A#m` *public* betragen. Die Sichtbarkeit von `A#m` konnte also nicht reduziert werden. Mit dem zugehörigen AST kann in diesem Fall allerdings genau bestimmt werden, ob, statisch gesehen, `A#m` oder `B#m` aufgerufen wird[1]. Damit kann die Beschränkung der SearchEngine überwunden werden.

Im Falle einer Überladung eines Operators können manche Methodenaufrufe mit JavaSearch nicht gefunden werden. Da in Java nur der „+“-Operator überladen wurde, gibt es allerdings nur einen Fall. Wenn ein beliebiges Objekt mit dem „+“-Operator mit einem Objekt vom Typ `String` verknüpft wird, erfolgt ein impliziter Aufruf der Methode `toString`. Die Methode `toString` ist in der Klasse `Object` definiert, die ein Supertyp aller Klassen ist. Da die Sichtbarkeit von `toString` *public* beträgt, muss die Sichtbarkeit von überschreibenden Methoden ebenfalls *public* betragen. Da die Sichtbarkeit nicht verringert werden kann, stellt das Nichtauffinden dieser Referenzen kein Problem dar.

Details zur Implementierung

Mit Hilfe der SearchEngine werden alle Aufrufe einer Methode gefunden. Die Funktionsweise der SearchEngine wurde in Abschnitt 3.2.5 erläutert. Die Suche wird auf „Source Folder“ eingeschränkt[2], also Aufrufe, die in einer Compilation-Unit des Projekts vorkommen. Denn Methoden, die im Quellcode eine Projekts definiert sind, werden nur an Programmstellen aufgerufen, die sich ebenfalls in diesem Quellcode befinden und nicht etwa in Archiven.

Wie oben erläutert, werden alle gefundenen Aufrufe verworfen, die sich nicht genau auf die zu untersuchende Methode beziehen. Dafür wird für jede Compilation-Unit, in der ein Methodenaufruf gefunden wurde, der zugehörige AST aufgestellt. Mit Hilfe der Binding-Keys wird die Methode an der Aufrufstelle mit der zu untersuchenden Methode verglichen.

---

[1] Welche Methode dynamisch aufgerufen wird, steht erst zur Laufzeit fest.

[2] Ohne eine Einschränkung der Suche würden auch alle Archive und die API-Klassen durchsucht werden, was sehr aufwändig ist.

6 Bedienung des *Access-Modifier-Modifier-Plugins*

6.1 Installation

Das *AMM-Plugin* benötigt Eclipse der Version 3.2.1 oder höher sowie ein installiertes Java Runtime Environment (JRE) der Version 5 oder höher. Das Plugin wurde mit Eclipse der Version 3.2 entwickelt und getestet. Eine einwandfreie Funktion mit anderen Versionen kann nicht garantiert werden.

Um das Plugin zu installieren, muss die Datei „de.gky.AccessModifierModifier_1.0.0.jar“ in das Plugin-Verzeichnis von Eclipse kopiert werden. Wenn Eclipse gerade ausgeführt wird, muss es danach neu gestartet werden. Wurde Eclipse z. B. im Verzeichnis „C:\Eclipse“ installiert, ist das Plugin-Verzeichnis ein direkter Unterordner mit Namen „plugins“. Die Datei ist auf der der beiliegenden CD im Verzeichnis „AccessModifierModifier\Deployment“ zu finden. Ein Inhaltsverzeichnis der CD ist in Anhang 4 abgedruckt.

Jedes Projekt, das refaktorisiert werden soll, muss zudem Zugriff auf das Archiv „AMM_Annotations_1.0.0.jar“ erhalten. Dazu muss das Archiv dem Build-Path hinzugefügt werden. Es enthält die Annotationstypen, mit denen die Schließung bzw. Öffnung einer Methode markiert werden[1]. Um es hinzuzufügen, muss aus dem Hauptmenü „Project > Properties > Java Build Path“ gewählt werden. Durch Klicken des Buttons „Add External Jars…“ im Karteireiter „Libraries“ erscheint ein Dialog, mit dem das Archiv ausgewählt werden kann. Das Archiv befindet sich auf der CD im selben Verzeichnis wie das *AMM-Plugin*.

Wenn der Programmcode Annotationen des *AMM-Plugins* aufweist und nicht mit dem Eclipse-Compiler, sondern z. B. mit dem Sun-Compiler übersetzt werden soll, muss der zugehörige Classpath immer das Archiv „AMM_Annotations_1.0.0.jar“ enthalten.

Die vom *AMM* benötigten Annotationstypen sowie die Enumeration „AccessModifier“ befinden sich im Paket „de.gky.ammannotations“.

[1] Da das *AMM-Plugin* ebenfalls diese Annotationstypen aufweist, kann es alternativ verwendet werden.

6.2 Die *Standardprüfung*

6.2.1 Allgemeines

Plugins können nach dem Speichern einer Quellcodedatei zeitintensive Aktionen ausführen. Da die *Standardprüfung* den Quellcode bei einer Prüfung ständig verändert und speichert, sollten diese Aktionen deaktiviert werden. Dies gilt besonders für die *Hintergrundprüfung*.

6.2.2 Aufrufmöglichkeiten

Es besteht die Möglichkeit ein ganzes Projekt, nur eine einzige Klasse oder alle Klassen einer Compilation-Unit zu schließen bzw. zu öffnen. Nach dem Aufruf des Plugins erscheint jeweils ein Dialog, mit dem der Benutzer Einstellungen vornehmen kann. Diese Dialoge werden weiter unten erläutert. Bild 6.1 zeigt die im Folgenden benutzten Elemente in Eclipse.

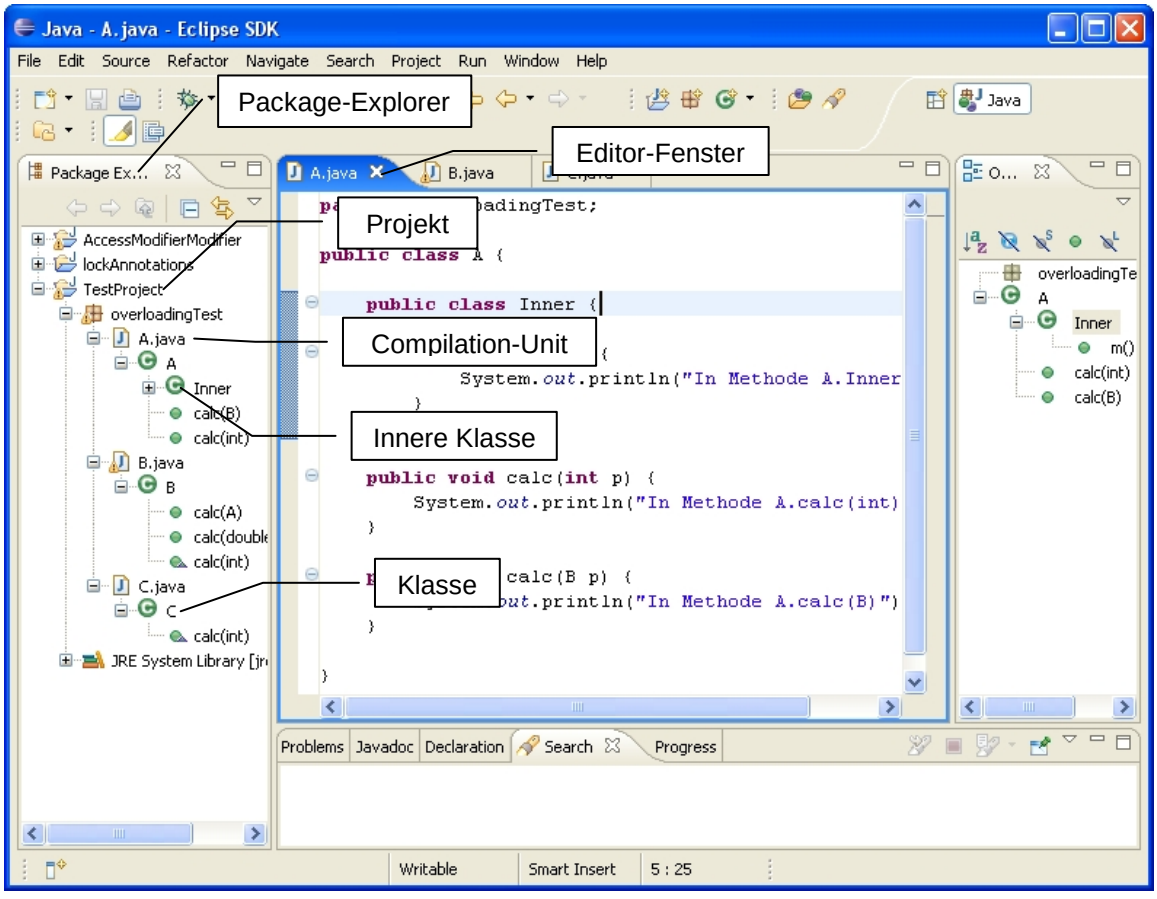

Bild 6.1: Elemente in Eclipse

Soll ein ganzes Projekt geschlossen werden, muss das Projekt im Package-Explorer markiert werden. Mit einem Klick auf die rechte Maustaste erscheint ein Popup-Menü. Aus dem Untermenü „AccessModfierModifier“ muss die Option „lock Project…“ ausgewählt werden. Die Option „unlock Project…“ wird ausgewählt, wenn das Projekt geöffnet werden soll.

Will ein Benutzer nur eine einzige Klasse schließen, wählt er im Package-Explorer die gewünschte Klasse aus. Diese Klasse kann auch eine innere Klasse sein. Mit dem Klick auf die rechte Maustaste erscheint ein Popup-Menü aus dem die Option „lock Class…“ des Untermenüs „AccessModifierModifier“ ausgewählt wird (siehe Bild 6.2). Wenn es sich bei der Klasse um eine innere Klasse handelt, wird die zugehörige äußere Klasse nicht geschlossen.

Eine Compilation-Unit kann auf zwei Arten geschlossen werden. Zum einen, indem analog zum Schließen einer Klasse, im Package-Explorer eine Compilation-Unit ausgewählt wird, zum anderen kann das Schließen auch vom Editor aus gestartet werden. Mit einem Klick auf die rechte Maustaste im aktuellen Editor-Fenster erscheint ein Kontextmenü. Der Menüpunkt „lock Classes…“ des Untermenüs „AccessModifierModifier“ startet das Schließen und ein Einstellungsdialog (siehe Bild 6.3) erscheint. Wenn die Compilation-Unit mehrere Klassen auf höchster Ebene aufweist, werden alle diese Klassen geschlossen und auch der Einstellungsdialog gilt für jede dieser Klassen[1]. Mit dem Menüpunkt „unlock Classes…“ können alle Klassen einer Compilation-Unit geöffnet werden.

[1] Üblicherweise hat eine Datei (Compilation-Unit) nur eine Klasse auf höchster Ebene.

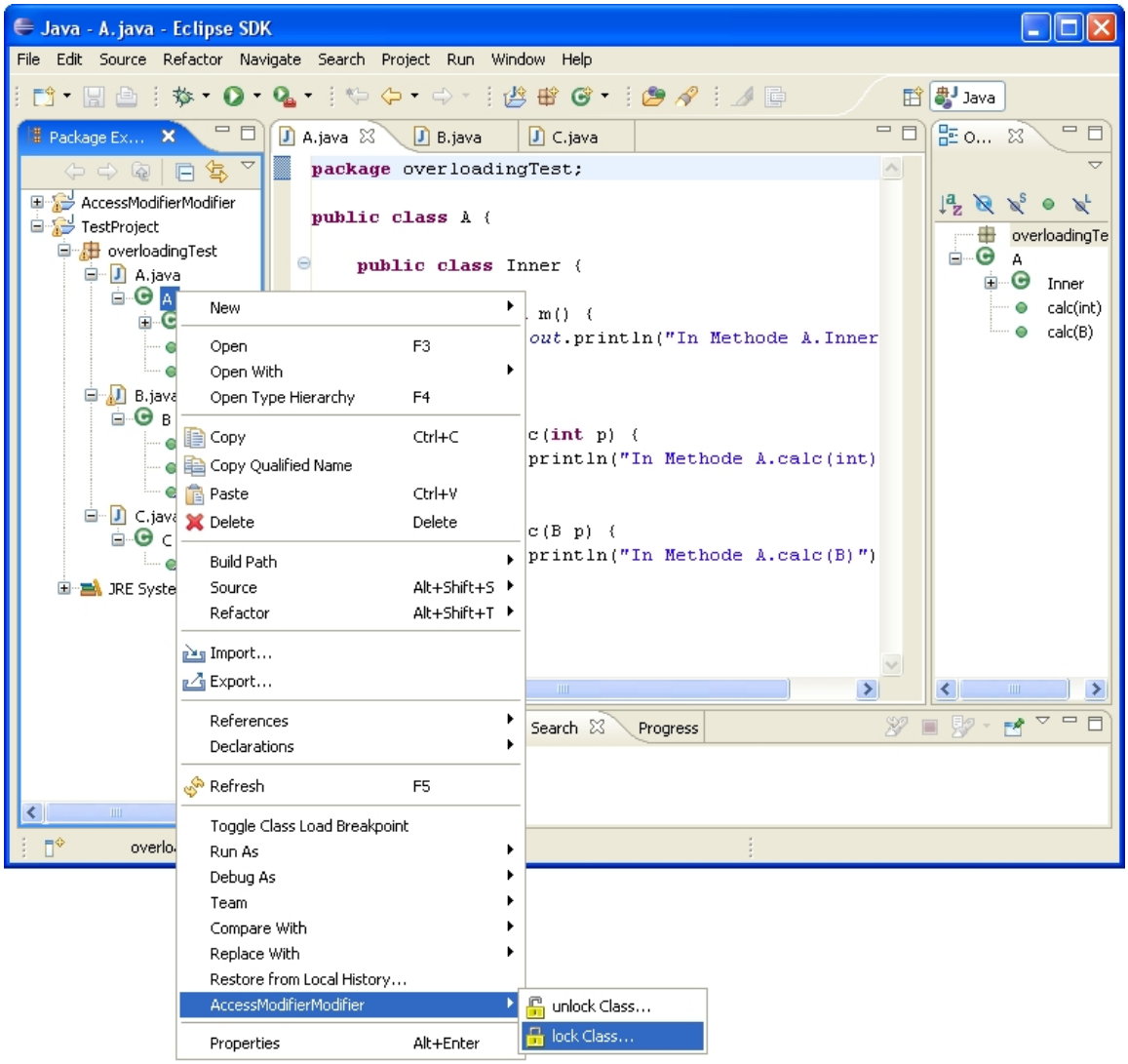

Bild 6.2: Menüauswahl

6.2.3 Einstellungsdialog für das Schließen einer Klasse

Bild 6.3 zeigt den Einstellungsdialog, der erscheint, wenn eine Klasse oder eine Compilation-Unit zum Schließen ausgewählt wurde. Der Benutzer kann auswählen, ob die ganze Hierarchie geschlossen werden soll. Hierarchie bedeutet hier, dass alle Klassen, die Super- oder Subtypen der ausgewählten Klasse sind, auch geschlossen werden. Ist ein ermittelter Subtyp ein anonymer Typ, wird er allerdings nicht geschlossen (vgl. Anhang 2). Von den Supertypen können vom *AMM-Plugin* selbstverständlich nur die Klassen geschlossen werden, die als Quellcode im Projekt vorliegen und nicht Teil einer Bibliothek sind. Die Klassen werden in der richtigen Reihenfolge bearbeitet. (Zur richtigen Reihenfolge beim Schließen siehe Abschnitt 4.3.4.) Weist eine Klasse innere Klassen auf, wer-

den diese ebenfalls geschlossen, solange es sich dabei nicht um anonyme Klassen handelt. Die Hierarchien dieser inneren Klassen werden beim Schließen nicht berücksichtigt. Ebenso wenig werden die Sub- bzw. Supertypen der inneren Klassen geschlossen, wenn diese nicht selbst innere Klassen der ausgewählten Klasse sind[1].

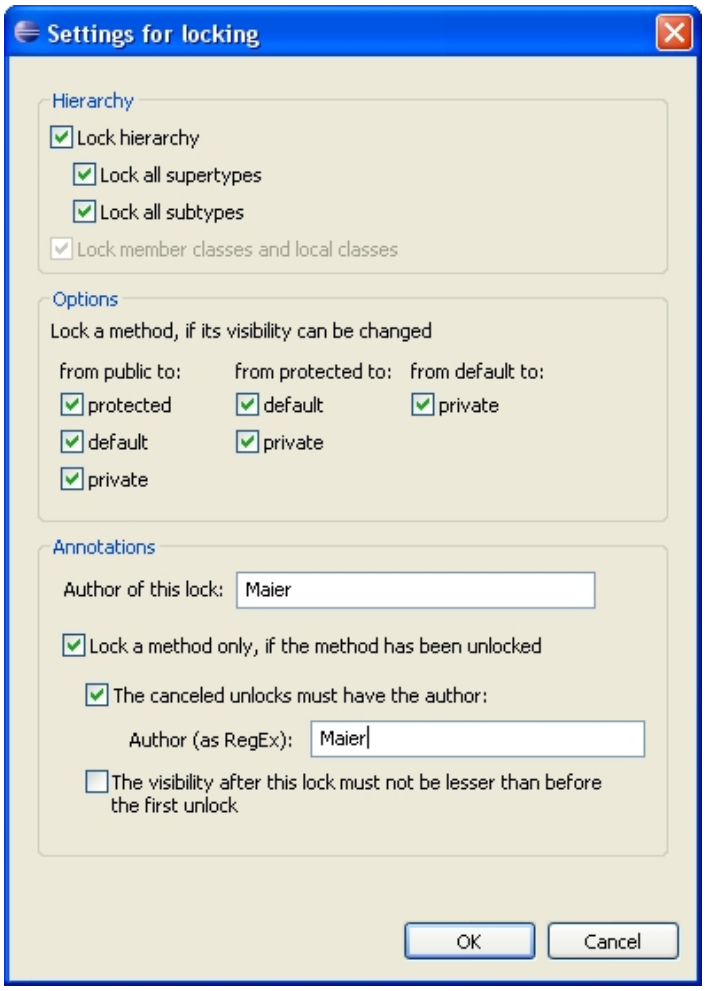

Bild 6.3: Dialog für das Schließen einer Klasse

Im Abschnitt „Options“ kann sehr detailliert festgelegt werden, welche Methoden tatsächlich geschlossen werden sollen. Mit diesen Einstellungen kann das Schließen an bestimmte Problemstellungen angepasst werden. In einem Fall will z. B. ein Benutzer, dass alle öffentlichen Schnittstellen einer Klasse erhalten bleiben. In einem anderen Fall sollen Methoden mit der Sichtbarkeit *protected* auch weiterhin in Subtypen überschrieben werden können. Dies alles kann eingestellt werden.

[1] Es ist aber möglich, im Package-Explorer eine innere Klasse auszuwählen und sie mit „lock Class …“ zu schließen. Dann können ihre Sub- und Supertypen beim Schließen mit geschlossen werden.

Die Schließung einer Methode kann von der ursprünglichen Sichtbarkeit der Methode abhängig gemacht werden. Der Benutzer kann festlegen, dass Methoden, deren Sichtbarkeit *public* bzw. *protected* oder *default* beträgt, nur geschlossen werden sollen, wenn sie nach der Schließung eine bestimmte Sichtbarkeit aufweisen. Wenn die dann ermittelte minimal mögliche Sichtbarkeit nicht ausgewählt wurde, wird die Sichtbarkeit auf die nächst größere ausgewählte Sichtbarkeit gesetzt.

Ein Beispiel soll diese Einstellungen verdeutlichen: In der Option „from public to:“ wurden nur die Sichtbarkeiten „protected“ und „private“ markiert; in der Option „from protected to:“ und „from default to:“ wurde nichts ausgewählt. Stellt das *AMM-Plugin* nun fest, dass die minimale Sichtbarkeit einer Methode, mit der ursprünglichen Sichtbarkeit *public*, *default* beträgt, wird die Sichtbarkeit nur auf *protected* gesetzt, da *default* nicht ausgewählt wurde. Alle Methoden, die eine ursprüngliche Sichtbarkeit von *protected* oder *default* aufweisen, werden vom *AMM-Plugin* nicht berücksichtigt.

Wird eine Methode geschlossen, werden vorhandene Annotationen angepasst und gegebenenfalls wird eine neue Schließungsannotation hinzugefügt. Die Schließungsannotation enthält den Namen des Autors, der für die Schließung verantwortlich ist. In der Gruppe „Annotations“ kann der Name des Autors im Feld „Author of this lock“ angegeben werden.

Der Benutzer kann weiter festlegen, dass eine Methode nur dann geschlossen werden soll, wenn sie vorher geöffnet worden ist. Eine Methode wurde geöffnet, wenn sie mit einer entsprechenden Öffnungsannotation versehen ist. Der Benutzer kann diese Auswahl weiter einschränken und nur die Methoden berücksichtigen, die von einem bestimmten Autor geöffnet worden sind. Der Autor wird als regulärer Ausdruck („Regular Expression“) angegeben. So ist es möglich, bestimmte Gruppen von Autoren zu berücksichtigen. Ein Programmierer einer Arbeitsgruppe hat z. B. die Kennung „gruppeA.MichaelMaier“. Mit der Angabe von „gruppeA.*“ werden alle Autoren der Kategorie „gruppeA“ berücksichtigt. Wenn der Autor Mitglied in mehreren Gruppen ist, können mit Angabe von „.*MichaelMaier“ alle Methoden, die der Autor Michael Maier geöffnet hat, geschlossen werden.

Zusätzlich kann festgelegt werden, dass die Sichtbarkeit einer Methode nur bis zu der Sichtbarkeit reduziert werden soll, die sie vor dem (ersten) Öffnen hatte.

6.2.4 Einstellungsdialog für das Öffnen einer Klasse

Wenn eine Klasse zum Öffnen ausgewählt wurde, erscheint der Dialog aus Bild 6.4. In diesem Dialog kann eingestellt werden, ob nur die Klasse selbst, oder die ganze Hierarchie geöffnet werden soll. Die Optionen für die Berücksichtigung der Hierarchie sind die gleichen wie beim Schließungsdialog. Weiter kann der Benutzer einstellen, ob eine geöffnete Methode mit einer Öffnungsannotation versehen werden soll. Als Autor wird in der Annotation der Name vermerkt, der im Feld „Author of the unlock“ angegeben wurde.

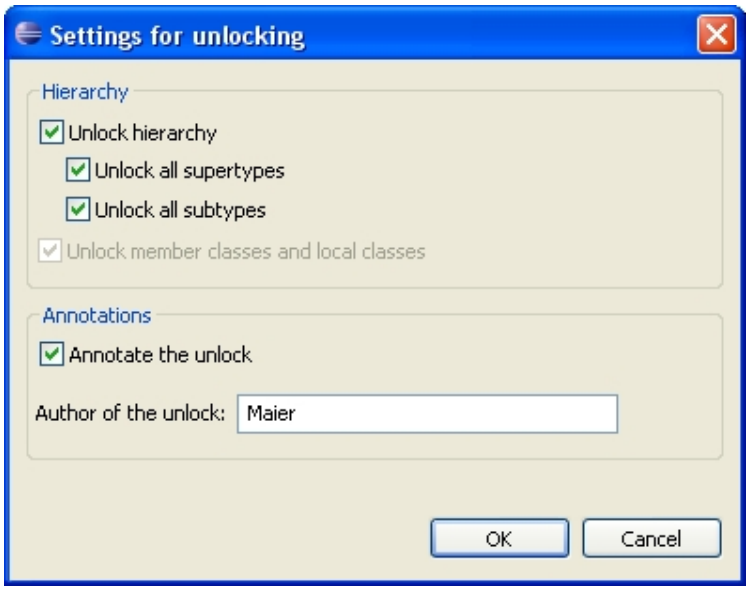

Bild 6.4: Dialog für das Öffnen einer Klasse

6.2.5 Einstellungsdialog für das Öffnen bzw. Schließen eines Projekts

Wenn ein ganzes Projekt geöffnet bzw. geschlossen wird, werden alle Klassen in diesem Projekt geöffnet bzw. geschlossen. Die Klassen werden in der richtigen Reihenfolge bearbeitet. Der Dialog für das Schließen eines Projekts gleicht dem Dialog für das Schließen einer Klasse, nur fehlen ihm die Einstellungsmöglichkeiten in der Gruppe „Hierarchy“. Entsprechendes gilt für den Öffnungsdialog.

6.3 Die *Hintergrundprüfung*

6.3.1 Bedienung

Nach dem Start von Eclipse ist die *Hintergrundprüfung* ausgeschaltet. Sie kann im Preferences-Dialog aktiviert werden. Wenn sie aktiviert ist, wird die Compilation-Unit geprüft, die sich im aktiven Editor-Fenster befindet. Kann die Sichtbarkeit einer in dieser Compilation-Unit enthaltenen Methode verringert werden, annotiert das *AMM-Plugin* sie mit einem Marker. Durch überstreichen des Markers mit dem Mauszeiger wird die ermittelte minimale Sichtbarkeit der zugehörigen Methode angezeigt (siehe Bild 6.5).

Jeder Marker stellt eine programmgesteuerte Lösung (QuickFix) zur Verfügung, welche die Sichtbarkeit der Methode auf die ermittelte minimale Sichtbarkeit reduzieren kann. Mit einem Mausklick auf den Marker erscheint im Menü ein Eintrag, um den QuickFix zu starten (siehe Bild 6.6). Eventuell vorhandene Öffnungsannotationen werden bei der Änderung angepasst und gegebenenfalls wird eine neue Schließungsannotation hinzugefügt (vgl. Bild 6.7).

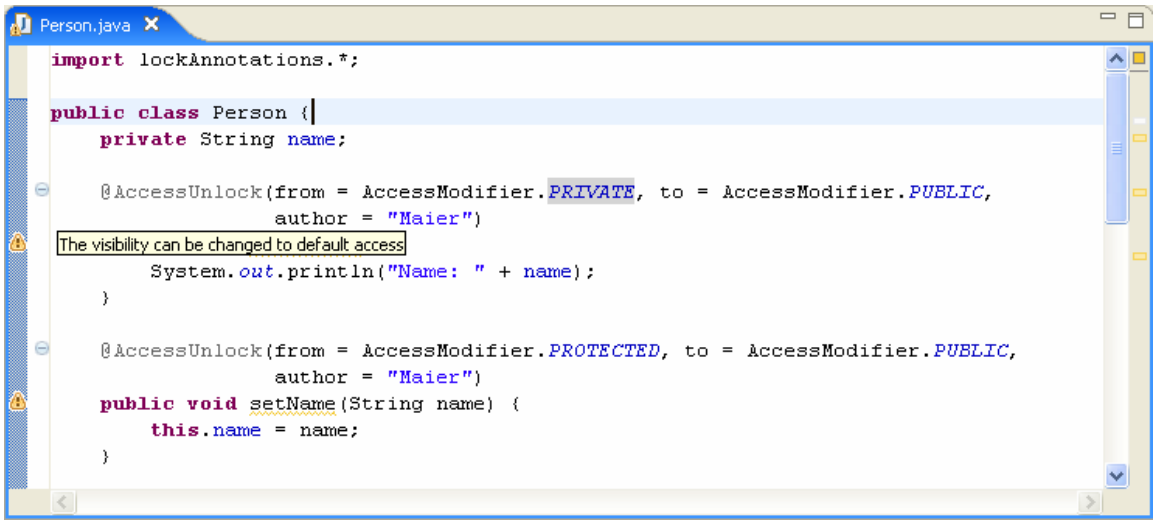

Bild 6.5: Mit *AMM*-Markern versehene Methoden. Der Info-Text des oberen Markers wird gerade angezeigt. Er weist den Benutzer darauf hin, dass die Sichtbarkeit der Methode `printName()` nach *default-access* geändert werden kann.

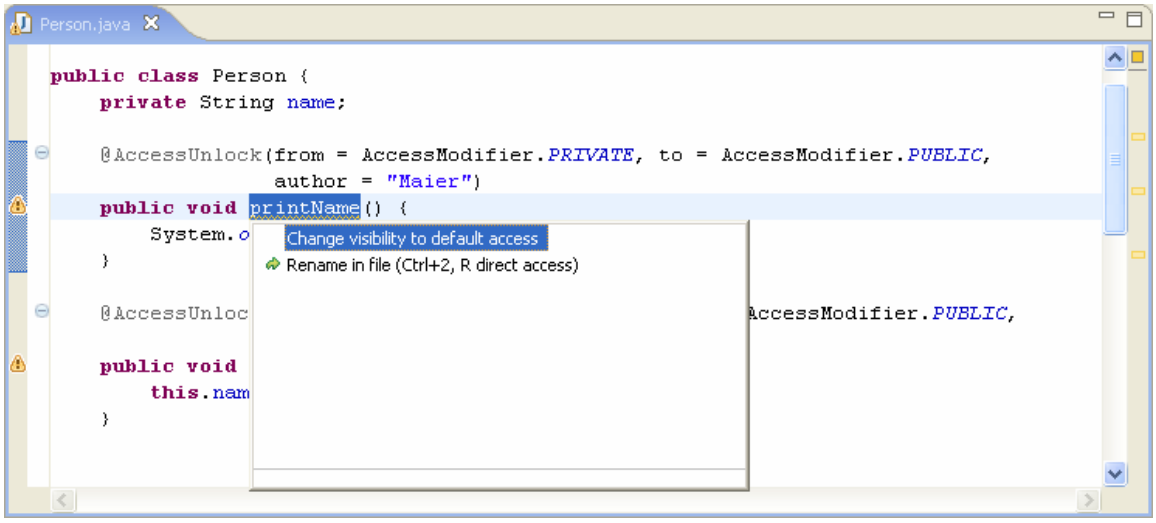

Bild 6.6: Der QuickFix für die Methode `printName()` wird ausgewählt.

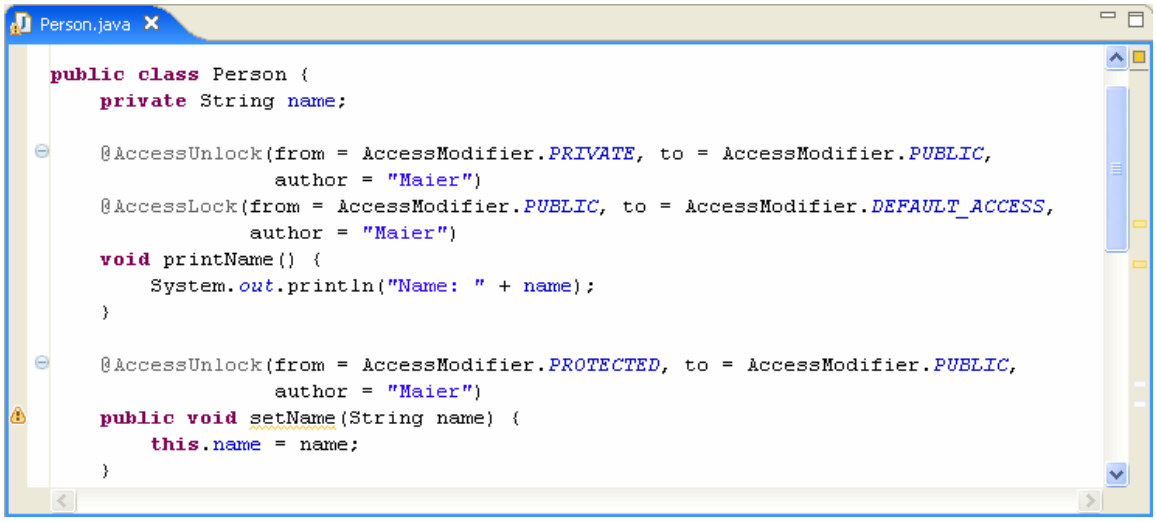

Bild 6.7: Der QuickFix hat die Sichtbarkeit der Methode `printName()` von *public* nach *default* geändert und die Annotationen angepasst.

Jedesmal wenn ein Editor-Fenster[1] aktiviert oder der Inhalt des aktiven Fensters gespeichert wird, startet eine neue Prüfung der gerade angezeigten Compilation-Unit. Durch diese Vorgehensweise werden auch Änderungen berücksichtigt, die in anderen Compilation-Units vorgenommen wurden. Da die Prüfung, besonders bei großen Projekten, recht aufwändig ist, kann es einige Zeit dauern, bis alle Methoden einer Compilation-Unit geprüft sind.

[1] In Eclipse können Editoren-Fenster auch als Registerkarten (Tabs) angeordnet sein. Registerkarten verhalten sich hier genauso wie einzelne Fenster.

Folgendes Szenario soll die Vorgehensweise deutlich machen: Der Benutzer will eine Compilation-Unit bearbeiten und öffnet sie im Editor. Durch die Aktivierung des Editor-Fensters startet die *Hintergrundprüfung*. Für eine Methode *m* mit Sichtbarkeit *public* ermittelt diese die minimale Sichtbarkeit *private*, da sie nur innerhalb der sie definierenden Klasse aufgerufen wird. Der Benutzer will nun eine andere, schon im Editor geöffnete, Compilation-Unit bearbeiten und aktiviert das entsprechende Editor-Fenster. Er fügt an einer Programmstelle eine Referenz der Methode *m* hinzu, wodurch die minimale Sichtbarkeit der Methode *m default* wird. Wenn der Benutzer wieder zur ersten Compilation-Unit wechselt, erfolgt (durch erneutes Aktivieren des Fensters) eine neue Prüfung. Diese stellt sicher, dass bei der Ermittelung der minimalen Sichtbarkeit die hinzugefügte Referenz berücksichtigt wird. Der Marker der Methode *m* zeigt nun als minimale Sichtbarkeit *default* an.

6.3.2 Einstellungen

Die Einstellungen für die *Hintergrundprüfung* kann der Benutzer im Preferences-Dialog vornehmen (siehe Bild 6.8). Dazu wählt er aus der Menüleiste „Window > Preferences…“ und klickt im erscheinenden Dialog in der linken Spalte auf die Option „Access-ModifierModifier“.

Die nun angezeigte Preferences-Seite enthält ausschließlich Einstellungen für die *Hintergrundprüfung*. Mit der Option „Activate background checking“ kann die Prüfung gestartet werden. Der Benutzer kann einstellen, dass nur Methoden, deren Sichtbarkeit auf eine bestimmte Sichtbarkeit reduziert werden kann, markiert werden[1]. Das ist hilfreich, wenn er z. B. nur wissen will, welche Methoden die Sichtbarkeit *private* annehmen können. (Ohne diese Einstellungsmöglichkeit würden auch immer Methoden mit einem Marker versehen, die nach *protected* oder *default* geändert werden könnten. Dann müsste er sich erst mühsam durch die Marker klicken um die gewünschten Methoden zu finden.)

Mit der Option „Consider annotation @MinAccess“ kann der Benutzer festlegen, ob eine vorhandene Annotation @MinAccess bei der Untersuchung der minimalen Sichtbarkeit

[1] Die Einstellmöglichkeiten sind hier geringer als beim obigen Schließungsdialog, da ein Benutzer die Sichtbarkeit einer Methode sieht, bevor er den QuickFix aktiviert. Gleiches gilt auch für die Optionen, die die Annotationen betreffen.

berücksichtigt werden soll. Durch Deaktivierung dieser Option, kann die tatsächliche minimale Sichtbarkeit von Methoden angezeigt werden, die mit @MinAccess annotiert sind.

Im Feld „Author of the lock“ kann der Benutzer seinen Name angegeben. Der angegebene Name erscheint in allen Schließungsannotation, die durch die QuickFixes hinzufügt werden. (Um eine Methode mit dem QuickFix schließen zu können, muss der hier angegebene Name nicht mit dem Namen des Autors einer vorhandenen Öffnungsannotation übereinstimmen.)

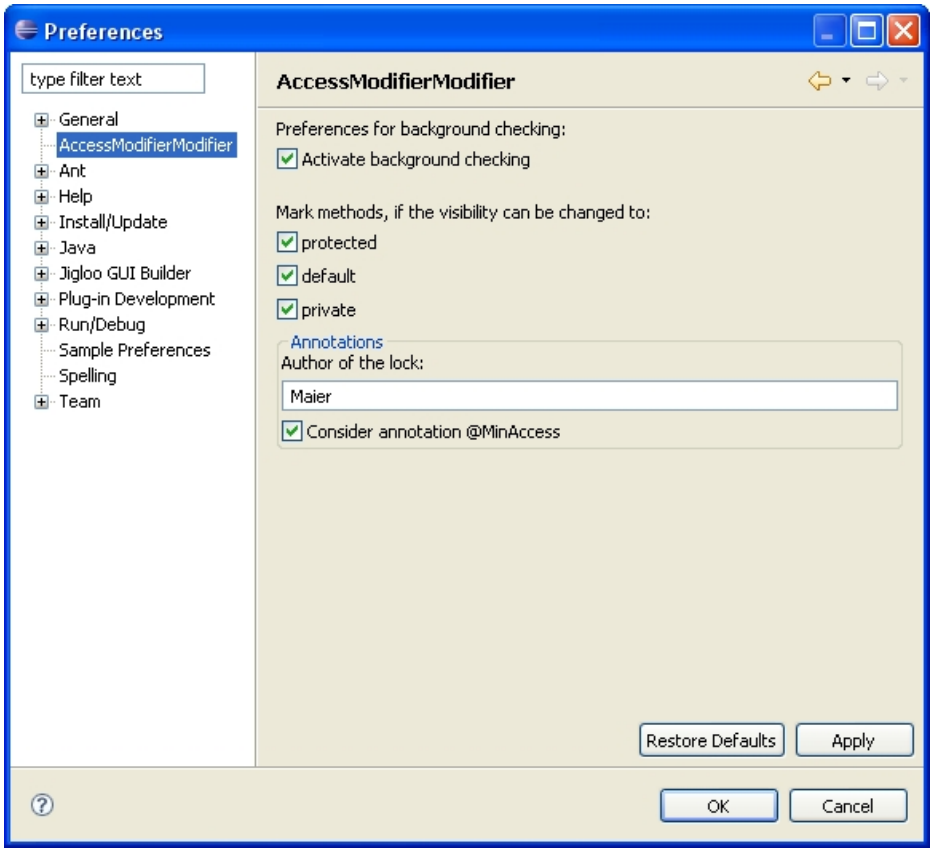

Bild 6.8: Einstellungen für die *Hintergrundprüfung*

7 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde die Entwicklung des *Access-Modifier-Modifiers* dokumentiert, das heißt eines „Refactoring-Werkzeugs“, das als Plugin für die Version 3.2 von Eclipse konzipiert wurde. Der *AMM* kann sowohl die minimal als auch die maximal zulässige Sichtbarkeit von Methoden ermitteln und sie entsprechend anpassen.

Es wurden zunächst zwei Ansätze vorgestellt, wie dies erfolgen kann. Der erste Ansatz, basierend auf einem Trial-and-Error-Vorgehen, versuchte das Problem unter Zuhilfenahme eines Compilers zu lösen. Da festgestellt wurde, dass mit diesem Ansatz keine Semantikänderungen erkannt werden können, musste er erweitert werden. Es wurde daher untersucht, unter welchen Umständen eine Sichtbarkeitsänderung einer Methode zu einer Modifikation der Semantik führen kann. Zusammenfassend kann gesagt werden, dass sich der Ablauf eines Programms nur dann ändern kann, wenn die geänderte Methode selbst überschrieben oder überladen ist.

Der zweite Ansatz verzichtete auf den Einsatz eines Compilers. Es wurde nun versucht die Prüfung aller Bedingungen selbst zu implementieren. Dafür wurden Regeln aufgestellt, unter welchen Bedingungen die Sichtbarkeit einer Methode erhöht oder verringert werden kann.

Eine Dokumentation der Hierarchie von Sichtbarkeitsänderungen wurde mit Hilfe von Annotationen erreicht. Sie ermöglichen es, festzustellen, welcher Entwickler die Sichtbarkeit eines Elements wie geändert hat. Mit weiteren Annotationen kann festgelegt werden, in welcher Weise die Sichtbarkeit einer Methode durch den *AMM* modifiziert werden darf.

Der *AMM* bietet eine *Standardprüfung* und eine *Hintergrundprüfung* an. In der *Standardprüfung* können eine einzelne Klasse, eine Compilation-Unit oder ein komplettes Eclipse-Projekt geschlossen bzw. geöffnet werden. Der Benutzer kann hier einstellen, welche Bedingungen Methoden erfüllen müssen, um für eine Änderung in Frage zu kommen. Die *Standardprüfung* basiert dabei auf dem erweiterten ersten Ansatz.

Mit der *Hintergrundprüfung* kann die minimal zulässige Sichtbarkeit von Methoden geprüft werden, während der Programmierer weiterarbeitet. Die minimale Sichtbarkeit wird ihm mit Hilfe von Eclipse-Info-Markern angezeigt. Letztere stellen ihm automatische

Lösungen, sogenannte „QuickFixes“, zur Verfügung, mit denen die Sichtbarkeit angepasst werden kann.

Da der Trial-and-Error-Ansatz für eine Prüfung im Hintergrund nicht geeignet war, mussten hier auch alle Regeln, die bei einem Schließen zu einem Übersetzungsfehler führen, implementiert werden. Die Prüfung auf eine Änderung der Semantik konnte allerdings von der *Standardprüfung* übernommen werden. Für die Prüfung auf mögliche Semantikänderungen und Übersetzungsfehler wurde die Funktionalität des JDT von Eclipse genutzt. Mit dem JDT war es möglich, Programmcode in einen Syntaxbaum zu überführen und damit die Einhaltung der aufgestellten Regeln zu überprüfen.

Ausblick

Nicht alle Ideen für den *AMM* konnten im Rahmen dieser Arbeit auch umgesetzt werden. So können in der derzeitigen Fassung zwar die Schnittstellen von Klassen minimiert bzw. maximiert werden, die Sichtbarkeit von Klassen selbst kann aber nicht eingeschränkt werden. Um dies zu ermöglichen, müsste zuerst untersucht werden, unter welchen Bedingungen die Sichtbarkeitsmodifikation einer Klasse zu Übersetzungsfehlern bzw. Semantikänderungen führen kann.

Die *Hintergrundprüfung* kann zurzeit die minimale Sichtbarkeit von Methoden in wenigen Fällen nicht korrekt bestimmen. Sie müsste so ergänzt werden, dass sie auch in diesen Fällen das richtige Ergebnis liefert.

Zudem konnten nicht alle Neuerungen berücksichtigt werden, die in der Version 5 der Programmiersprache Java eingeführt wurden. So werden z. B. Enumerationen, die ebenso wie Klassen, eine Schnittstelle aufweisen, beim Schließen nicht berücksichtigt.

Das JDT von Eclipse bietet für die standardmäßig mitgelieferten Refactoring-Werkzeuge eine einheitliche Bedienungsoberfläche an. Aus Gründen der Einheitlichkeit könnte der *AMM-Plugin* in diese eingebunden werden.

Der *AMM*, als Eclipse-Plugin konzipiert, kann nur verwendet werden, wenn als Entwicklungsumgebung Eclipse eingesetzt wird. Um den *AMM* auch anderen Benutzerkreisen zugänglich zu machen, könnte er für andere IDEs, wie z. B. „Netbeans“ [Netb07] portiert werden. Da der *AMM* viele Funktionen von Eclipse benutzt und deshalb stark an Eclipse gekoppelt ist, ist dies allerdings kein leichtes Unterfangen.

ANHANG

Anhang 1: Funktionstests

Die nach dem Refactoring erwartete Sichtbarkeit einer Methode kann mit Hilfe von Annotationen vermerkt werden. Diese können mit programmgesteuerten Tests ausgewertet werden. Die Annotation @TestProjectMin gibt die minimale Sichtbarkeit an, falls das Projekt als Ganzes geschlossen wird. (Beim Schließen eines Projekts werden alle äußeren Klassen in der richtigen Reihenfolge geschlossen.) Mit der Annotation @TestBackgroundMin kann die Sichtbarkeit vermerkt werden, die die *Hintergrundprüfung* ermitteln sollte.

Um die *Standardprüfung* zu testen, muss ein Projekt zuerst, wie in Abschnitt 6.2.1 beschrieben, geschlossen werden. Denn dieser Test prüft nur, ob die aktuelle Sichtbarkeit mit der angegebenen übereinstimmt. Beim Test der *Hintergrundprüfung* darf das Projekt dagegen nicht geschlossen sein. Denn hier wird bei Testausführung die *Hintergrundprüfung* gestartet und ihre Ergebnisse mit den Annotationen verglichen.

Wenn eine Methode keine @TestBackgroundMin-Annotation aufweist, wertet der Test der *Hintergrundprüfung* eine eventuell vorhandene @TestProjectMin-Annotation aus. Die Annotation @TestBackgroundMin muss also nur angegeben werden, wenn die *Hintergrundprüfung* eine andere minimale Sichtbarkeit als die *Standardprüfung* ermittelt.

Der Zugriff auf das Debug-Menü, mit dem die Tests gestartet werden, ist nur in der Debug-Version möglich. Es wird aufgerufen, indem im Package-Explorer auf eine Compilation-Unit mit der rechten Maustaste geklickt wird. Die Testergebnisse werden nur auf der Konsole ausgegeben.

Auf der beiliegenden CD befindet sich eine Auswahl an Eclipse-Projekten, mit denen das *AMM-Plugin* getestet wurde.

Anhang 2: Beschränkungen der *Standardprüfung*

Klassensichtbarkeit

Die *Standardprüfung* kann die Schnittstelle einer Klasse anpassen, indem sie die Sichtbarkeit von Methoden ändert. Eine Klasse weist in Java selbst eine Sichtbarkeit auf, die modifiziert werden kann. Allerdings verändert die *Standardprüfung* bei dem Schließen bzw. Öffnen einer Klasse diese Klassensichtbarkeit nicht. Nachfolgend wird kurz skizziert, ohne Anspruch auf Vollständigkeit, welche Unterschiede sich daraus ergeben.

Das unterschiedliche Verhalten ist gering, wenn jede Klasse mindestens einen expliziten Konstruktor aufweist. Denn dann steht der implizite Standardkonstruktor, dessen Sichtbarkeit nicht modifizierbar ist, nicht mehr zur Verfügung. Beim Schließen bzw. Öffnen werden dann alle (expliziten) Konstruktoren einer Klasse geschlossen, so dass sich die Klasse nahezu so verhält, als sei die Sichtbarkeit der Klasse selbst minimiert worden.

Ein Beispiel: Die Klasse *C* weist die Sichtbarkeit *public* auf und enthält einen expliziten Konstruktor und mehrere Methoden, die alle die Sichtbarkeit *public* besitzen. Die Klasse ist an allen Programmstellen sichtbar und kann folglich an diesen instanziiert werden. Die Klasse wird nun geschlossen. Nehmen wir an, die Sichtbarkeit des Konstruktors und aller Methoden können nach *default* geändert werden. Die Klasse kann jetzt nur noch an Programmstellen instanziiert werden, die sich im selben Paket befinden. Zudem sind auch alle ihre (selbst definierten) Methoden außerhalb des Paketes nicht mehr sichtbar.

An einer Programmstelle außerhalb des Paketes kann es aber weiterhin eine Referenz des Typs *C* geben, die auf eine Instanz dieses Typs zeigt. Allerdings können mit dieser Referenz nur Methoden aufgerufen werden, die die Klasse *C* aus Supertypen geerbt hat und die zudem an der Programmstelle auch sichtbar sind.

Wenn die Supertypen, von denen diese Methoden geerbt wurden, an dieser Stelle selbst sichtbar sind, ergibt sich daraus kein abweichendes Verhalten. Denn dann kann es, auch wenn die Klasse selbst geschlossen wird, an der Stelle immer eine Referenz geben, die den Typ eines dieser Supertypen aufweist.

Anders liegt der Fall, wenn diese Supertypen an der Stelle nicht sichtbar sind. Denn dann kann, wenn die Klasse selbst geschlossen wird, nie auf die Methoden dieser Supertypen zugegriffen werden.

Anonyme Klassen

Anonyme Klassen werden von der *Standardprüfung* nicht geschlossen bzw. geöffnet. Über die Instanz, die mit einer anonymen Klasse erzeugt wurde, kann nur über die Schnittstelle ihrer Superklasse (bzw. des implementierten Interfaces) zugegriffen werden. Eine Einschränkung der Sichtbarkeit ist bei anonymen Klassen deshalb nicht notwendig. Lediglich beim Öffnen einer Hierarchie kann es u. U. zu geringen Abweichungen kommen: Wenn die komplette Hierarchie geöffnet werden soll, können Methoden, die in anonymen Klassen überschrieben werden, durch diese Einschränkung eventuell nicht geöffnet werden. Da der Schwerpunkt dieser Arbeit in der Minimierung von Schnittstellen liegt, scheint diese Beschränkung beim Öffnen vertretbar.

Die Klassen `OverridingLockProblemChecker` und `OverloadingLockProblemChecker` berücksichtigen bei Prüfung einer zu schließenden Klasse aber auch anonyme Subklassen.

Berücksichtigte Klassen

Bei der *Standardprüfung* (sowie der *Hintergrundprüfung*) werden immer nur die Klassen berücksichtigt, die sich im selben Eclipse-Projekt befinden. Diese Beschränkung wurde aus Performance-Gründen festgelegt.

Bekannte Bugs

Beim Testen der *Standardprüfung* sind nur wenige Bugs aufgefallen, die zudem vom Verfasser nicht als kritisch eingestuft werden:

- Die Hierarchie einer Klasse *C* soll geschlossen werden. Es gibt in einer Compilation-Unit zwei lokale Klassen, die beide denselben Name aufweisen und die beide Subtypen der Klasse *C* sind. Dann wird nur eine dieser lokalen Klassen geschlossen.
- Wenn eine Compilation-Unit geschlossen wird, erscheint für jede Klasse auf höchster Ebene ein eigener Refactoring-Progress-Monitor.

Anhang 3: Beschränkungen der *Hintergrundprüfung*

Es gibt einige Fälle, in denen die *Hintergrundprüfung* die minimale Sichtbarkeit einer Methode nicht, oder nicht richtig ermitteln kann. Diese sind im Folgenden aufgeführt.

Anonyme und lokale Klassen

- Wenn sich die zu schließende Methode in einer generischen Klasse befindet und in einer anonymen Klasse überschrieben wird, kann die minimale Sichtbarkeit nicht richtig bestimmt werden.
- Methoden von anonymen und lokalen Klassen werden nicht untersucht. Diese Klassen können nur in der sie definierenden Klasse benutzt werden, so dass eine Reduktion ihrer Schnittstellen nicht zwingend notwendig ist.

Konstruktoren

- Wenn ein Konstruktor von einem impliziten Konstruktor einer Subklasse aufgerufen wird, ermittelt die *Hintergrundprüfung* für ersteren auch dann die minimale Sichtbarkeit *protected*, wenn sich die Klasse und die Subklasse im gleichen Paket befinden.

Paketwechsel innerhalb der Klassenhierarchie

- Wenn der in Regel S.5 aus Abschnitt 4.4.2 aufgeführte Sonderfall auftritt, wird für die minimale Sichtbarkeit *default* ermittelt. Das Ergebnis müsste allerdings *protected* sein. Dieser Fall kann nur dann auftreten, wenn innerhalb einer Klassenhierarchie das Paket gewechselt und wieder in das gleiche Paket zurück gewechselt wird. Da dieses Hin- und Zurückwechseln auf schlechtes Programmdesign schließen lässt, scheint es vertretbar, es zumindest bei der *Hintergrundprüfung* nicht zu berücksichtigen.

Anhang 4: Inhaltsverzeichnis der beiliegenden CD

Die folgende Liste zeigt eine Auswahl der auf der beiliegenden CD enthaltenen Dateien.

de.gky.AccessModifierModifier_1.0.0.jar	–	Das *Access-Modifier-Modifier-Plugin* in der Auslieferungsversion.
AMM_Annotations_1.0.0.jar	–	Archiv mit den kompilierten Annotationstypen, die vom *AMM-Plugin* benötigt werden.
AccessModifierModifierProjekt1.0.0_ReleaseVersion.zip	–	Die Release-Version des *AMM* als archiviertes Eclipse-Plugin-Projekt. Die Auslieferungsversion basiert auf dieser Version.
AccessModifierModifierProjekt1.0.0_DebugVersion.zip	–	Die Debug-Version des *AMM* als archiviertes Eclipse-Plugin-Projekt. Diese Version enthält ein Debug-Menü, mit dem Tests gestartet werden können.
TestEclipseProjekte.zip	–	Archivierte Eclipse-Projekte, mit denen der *AMM* getestet wurde (siehe Anhang 1).
eclipse-SDK-3.2.1-win32.zip	–	Eclipse der Version 3.2.1 – Installationsdatei für MS-Windows
Inhaltsverzeichnis.txt	–	Aktuelles Inhaltsverzeichnis der CD

Die auf der CD enthaltenen archivierten Plugin-Projekte können leicht auf eine Eclipse-Instanz übertragen werden. Dazu muss aus dem Hauptmenü „File > Import > General > Existing Projects into Workspace“ gewählt werden. In dem erscheinenden Dialog kann dann das Archiv mit den zu importierenden Projekten ausgewählt werden.

Literaturverzeichnis

[Ant07] o.V.: *Homepage des Ant-Projekts*, [online] URL: http://ant.apache.org (Stand 10.06.2007)

[Arth04] Arthorne, John; Laffra, Chris: *Official Eclipse 3.0 FAQ*, Boston: Addison-Wesley, 2004.

[Bäum07] Bäumer, D. et al.: *Integrating Refactoring Support into a Java Development Tool*, [online] URL: http://pag.csail.mit.edu/~akiezun/companion.pdf (Stand 10.06.2007)

[Bäum04] Bäumer, D.; Megert, D.;Weinand, A.:*Eclipse Plug-Ins – Entwickeln und publizieren*, Artikel im Internet, 2004, URL: http://medien.informatik.fh-fulda.de/Members/milde/lehre/projektplanung/weinand_OS_01_04.pdf (Stand 10.06.2007)

[Beck01] Beck, Kent: *Extreme programming explained: embrace change*, Boston, Mass.: Addison-Wesley, 2001.

[Clay04] Clayberg, Eric; Rubel, Dan: *eclipse – Building Commercial-Quality Plug-ins*, Boston: Addison-Wesley, 2004.

[Danj04] D'Anjou, Jim et al.: *The Java developer's guide to Eclipse – 2nd ed.*, Boston: Addison-Wesley, 2004.

[Ecke04] Eckel, B.: *Thinking in Java – 3rd Edition*, Prentice Hall, 2004.

[Ecli07] o.V.: *Homepage des Eclipse-Projekts*, [online] URL: http://www.eclipse.org (Stand 10.06.2007)

[EHil06] o.V.: *Benutzerhilfe von Eclipse 3.2*, 2006.

[Fowl00] Fowler, Martin: *Refactoring – Wie sie das Design vorhandener Software verbessern*, München: Addison-Wesley, 2000.

[Gamm95] Gamma, Erich et al.: *Design patterns: elements of reusable object-oriented software*, Reading, Mass.: Addison-Wesley, 1995.

[Gosl05] Gosling, James et al.: *The Java Language Specification, Third Edition*, Boston: Addison-Wesley, 2005.

[Hint05] Hinterwäller, Bodo: *Metamodell-basierte Spezifikation von Refactorings*, Diplomarbeit am Institut für Softwaretechnik der Universität Koblenz-Landau, 2005.

[Jigl06] o.V.: *Jigloo 3.9.0: SWT/Swing GUI Builder for Eclipse and WebSphere*, Cloudgarden, 2006, [online] URL: http://www.cloudgarden.com/jigloo (Stand 10.06.2007)

[Krüg00] Krüger, G.: *Java 2 - Handbuch der Java-Programmierung*, 2. Auflage, München: Addison-Wesley, 2000.

[Kuhn06] Kuhn, T.; Thomann, O.; *Abstract Syntax Tree*, [online] URL: http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST-/index.html (Stand 10.06.2007)

[Link05] Link, Johannes: *Softwaretests mit JUnit: Techniken der testgetriebenen Entwicklung*, 2. Aufl., Heidelberg: dpunkt, 2005.

[Netb07] o.V.: *Netbeans Homepage*, [online] URL: http://www.netbeans.org (Stand 10.06.2007)

[Tip03] Tip, F. et. al.: *Refactoring for generalization using type constraints*. In: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications. OOPSLA '03. ACM Press, New York.

[Wint05] Winter, Mario: *Methodische objektorientierte Softwareentwicklung: eine Integration klassischer und moderner Entwicklungskonzepte*, 1. Auflage, Heidelberg: dpunkt-Verl., 2005.

Ehrenwörtliche Erklärung

Ich versichere, dass ich die vorliegende Masterarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt sowie Zitate kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mannheim, den 19. August 2007 gez.