Institut für Softwaretechnik, Fachbereich Informatik,

Universität Koblenz-Landau, Abteilung Koblenz

Diplomarbeit

Metamodell-basierte Spezifikation von Refactorings

Betreuer:

Prof. Dr. Jürgen Ebert

Dr. Andreas Winter

27. Juni 2005

Bodo Hinterwäller

Backhausstraße 3, D-56340 Dachsenhausen,

bodo@hinterwaeller.de

Zusammenfassung

Die Implementation eines Softwareprodukts erfordert kontinuierliche Änderungen am zugrundeliegenden Sourcecode. Folglich lässt es sich kaum vermeiden, dass der entwickelte Programmcode zunehmend unstrukturierter wird. Abhilfe kann Refactoring schaffen, welches ein methodisches Vorgehen zur Überarbeitung von existierendem Programmcode bietet.

Ziel dieser Arbeit ist es, ein Verfahren zur Metamodell-basierten Spezifikation von Refactorings zu entwickeln. Abschließend folgt die Implementation eines Eclipse Plug-Ins, welches die zuvor beschriebenen Modell-Transformationen durchführt und dementsprechende Sourcecode-Änderungen vornimmt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1
Themendarstellung

Die vorliegende Diplomarbeit beschäftigt sich mit der Metamodell-basierten Spezifikation von Refactorings zur Durchführung von Sourcecode-Änderungen. Dieses Kapitel beschreibt die Motivation und Zielsetzung dieser Arbeit und gibt einen Überblick über das geplante Vorgehen.

1.1 Motivation

Die Implementation eines Softwareprodukts erfordert kontinuierliche Änderungen des Sourcecodes um weitere Funktionalitäten hinzuzufügen oder Fehler zu beheben. Daher lässt es sich kaum vermeiden, dass der entwickelte Programmcode zunehmend unstrukturierter wird. Einzelne Methoden werden durch Hinzufügen zusätzlicher Codezeilen verkompliziert, ebenso erzeugen Copy-Paste-Aktionen des Programmierers einen unübersichtlichen Programmtext. Abhilfe kann Refactoring schaffen, welches ein methodisches Vorgehen zur Überarbeitung eines existierenden Programmcodes ermöglicht.

Refactoring steht für das Verändern eines Sourcecodes zur Verbesserung der Lesbarkeit, Struktur und Wartbarkeit – ohne Änderungen an der Funktionalität vorzunehmen. Ziel ist es, einen strukturierten Quellcode zu erzeugen, Redundanzen zu entfernen und die Wiederverwendbarkeit des Codes zu fördern. Zu diesem Zweck beschreibt Martin Fowler [Fowl99] in seinem Referenzwerk eine Vielzahl von möglichen Refactoring-Maßnahmen, die zur Codeverbesserung beitragen können.

Mittlerweile existieren verschiedene Werkzeuge, die einen Entwickler bei der Durchführung von Refactorings unterstützen. Allerdings basieren diese Änderungen stets auf dem vorliegenden Programmcode. Eine Metamodell-basierte Spezifikation von Refactorings hingegen ermöglicht das Anwenden von Sourcecode-Transformationen auf einer höheren Abstraktionsebene und somit unabhängig von einer vorliegenden Implementierung.

1.2 Zielsetzung

Im Rahmen dieser Diplomarbeit soll ein Verfahren entwickelt werden, welches Sourcecode-Transformationen auf Metamodell-Ebene ermöglicht und sich an dem in [EbKu97] vorgestellten "Extract-Transform-Rewrite"-Ansatz orientiert. Das geplante Vorgehen zur Sourcecode-Änderung auf Modell-Ebene lässt sich anhand folgender Abb. 1-1 schematisch veranschaulichen: Ein zu manipulierender Sourcecode wird durch ein entsprechendes Modell repräsentiert. Dieses Modell ist Instanz eines Metamodells. Unter der Voraussetzung, dass das Sourcecode-Modell die benötigten Informationen bereitstellt, können Code-Transformationen sodann auf Basis des extrahierten Modells unter Anwendung einer Transformationsspezifikation beschrieben und durchgeführt werden. Das Resultat ist ein transformiertes Modell, welches ebenfalls eine Instanz des gemeinsamen Metamodells darstellt und abschließend zur Erzeugung des refaktorisierten Sourcecodes verwendet werden kann.

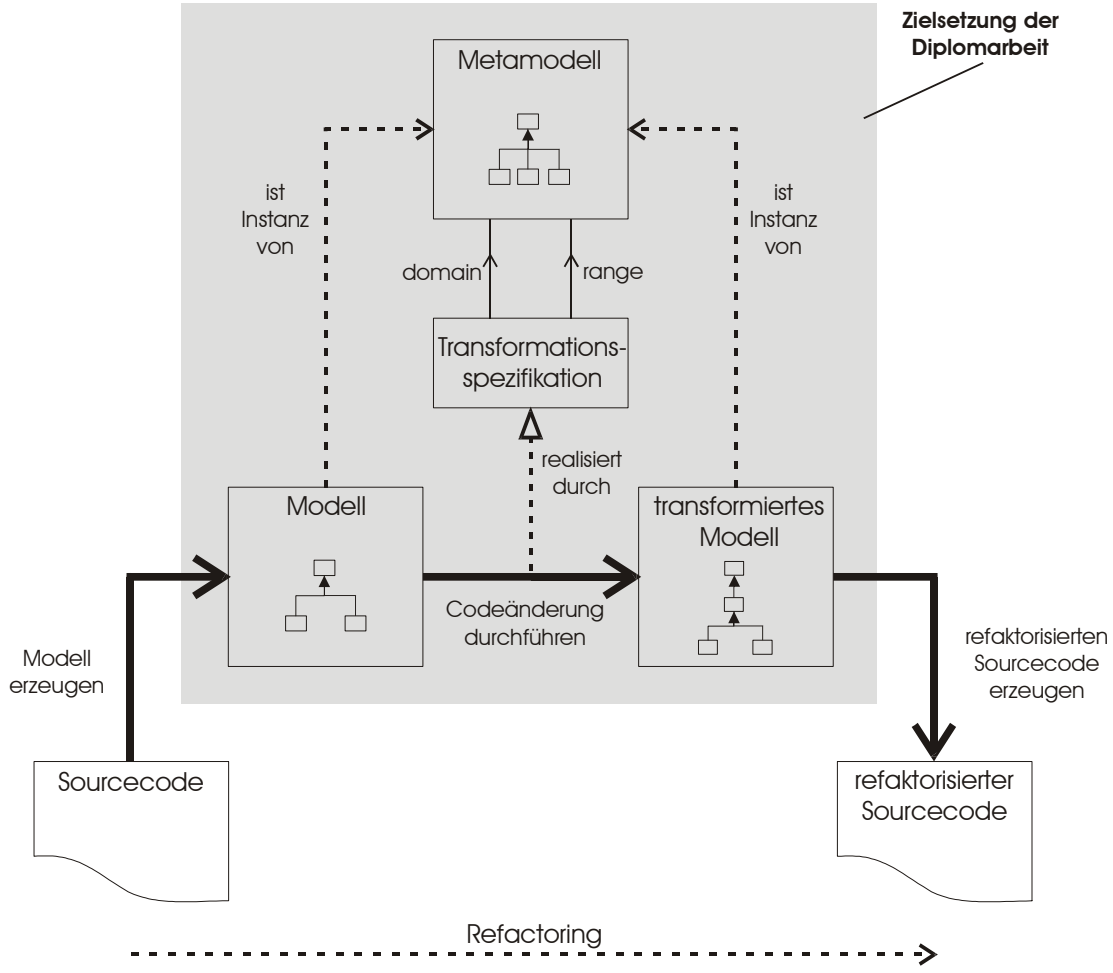


Abb. 1-1: Zielsetzung

Um die Adäquatheit des Metamodells und des Transformationsbeschreibungsansatzes demonstrieren zu können, wird exemplarisch ein Refactoring aus Fowlers Katalog ausgewählt sowie die damit verbundenen Sourcecode-Änderungen auf Modell-Ebene beschrieben und spezifiziert. Abschließend erfolgt die Implementation eines Eclipse Plug-Ins, welches basierend auf den Ergebnissen dieser Arbeit eine Sourcecode-Änderung unter Anwendung des ausgewählten Refactorings auf Modell-Ebene ermöglicht.

1.3 Vorgehen

In Kapitel 2 wird die Entwicklungsumgebung "Eclipse" und das zugrunde liegende Plug-In-Konzept vorgestellt, sodass die Ergebnisse dieser Diplomarbeit zu einem späteren Zeitpunkt in Form einer Eclipse-Komponente zur Verfügung gestellt werden können. Der Einsatz dieser Entwicklungsumgebung bietet sich an, da Eclipse selbst eine Komponente enthält, die das Refaktorisieren eines Sourcecodes unterstützt und somit nützliche Informationen für diese Diplomarbeit liefern kann. Zudem existiert eine Vielzahl frei verfügbarer sowie kommerzieller Eclipse-Erweiterungen, die ebenfalls von Interesse sein könnten. Auch die Einsicht des verfügbaren Sourcecodes wird für den weiteren Verlauf hilfreich sein.

Kapitel 3 beschreibt das Finden eines geeigneten Metamodells und bietet eine einführende Erläuterung der in Eclipse enthaltenen Java Development Tools (JDT) sowie der dort zur Verfügung stehenden Funktionalität zur Erzeugung eines Sourcecode-Modells. Auf dieser Grundlage werden im darauf folgenden Kapitel 4 die Anwendung eines ausgewählten Refactorings an einem Sourcecode-Beispiel dargestellt und die notwendigen Transformationen an dem korrespondierenden Sourcecode-Modell erläutert.

In Kapitel 5 erfolgen die Zerlegung des Refactorings in Einzelschritte sowie eine Erläuterung der in jedem Teilschritt durchzuführenden Modell-Transformationen. Kapitel 6 beinhaltet eine Einführung in die Sprache PROGRES und der zugehörenden Konzepte zur Beschreibung von Graphtransformationen, anhand derer das Refactoring in eine formale Spezifikation überführt wird. Weiterhin werden in Kapitel 7 die Sprachen GRAL und GReQL vorgestellt, auf deren Basis eine algorithmische Spezifikation des ausgewählten Refactorings erfolgt.

Damit endet die Darstellung einer Sourcecode-Änderung auf Modell-Ebene und Kapitel 8 erläutert die abschließende Implementation eines Eclipse Plug-Ins, welches die erstellte Spezifikation zur Ausführung bringt und somit die Anwendung des ausgewählten Refactorings auf Modell-Ebene ermöglicht.

Abschließend folgen in Kapitel 9 eine Zusammenfassung der Ergebnisse dieser Diplomarbeit sowie ein Ausblick auf mögliche zukünftige Weiterentwicklungen.

Kapitel 2

Das Plug-In-Konzept von Eclipse

Das vorliegende Kapitel vermittelt einen Einblick in die Architektur sowie Erweiterbarkeit der Eclipse-Plattform und schafft somit die Grundlage zur Entwicklung eines eigenen Plug-Ins, um abschließend die Ergebnisse dieser Diplomarbeit implementieren zu können.

2.1 Eclipse-Entwicklungsumgebung

Eclipse ist eine universelle, frei verfügbare Open-Source-Werkzeugplattform und hat sich im Java-Umfeld als leistungsstarke Entwicklungsumgebung etabliert [Ecli04]. Im Gegensatz zu anderen Programmierumgebungen besteht Eclipse aus einem kleinen Kern mit einer unbegrenzten Anzahl von Erweiterungsmöglichkeiten in Form von Plug-Ins [BaMe04]. Die Eclipse-Plattform, welche die Kernkomponente der IDE darstellt, die Plug-In-Entwicklungsumgebung (`PDE`) sowie Java-Entwicklungswerkzeuge (`JDT`) bilden die Basiskomponenten von Eclipse (siehe Abb. 2-1).

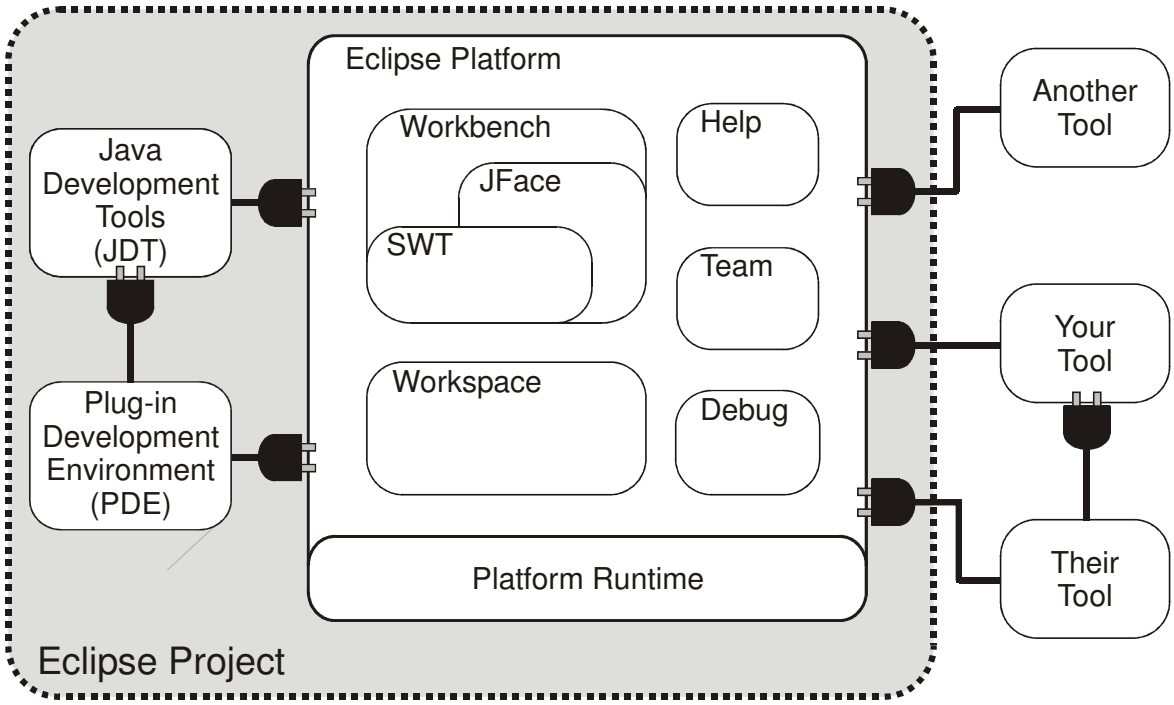


Abb. 2-1: Eclipse-Architektur [Jdtp04]

Die Eclipse-Plattform selbst setzt sich wiederum aus verschiedenen Elementen zusammen: Die Laufzeitumgebung (`Platform Runtime`) übernimmt das Plug-In-Management, im `Workspace` werden sämtliche Daten und Projekte eines Benutzers verwaltet, die `Workbench` ist das User Interface von Eclipse und basiert auf dem Standard Widget Toolkit (`SWT`) sowie `JFace`, einem GUI auf SWT-Basis. Weiterhin stehen ein umfangreiches Hilfesystem (`Help`), Debugging-Funktionalität (`Debug`) sowie eine Unterstützung für die Team-Programmentwicklung (`Team`) zur Verfügung.

2.2 Komponentenmodell

Eclipse ist aber mehr als eine reine Java-IDE. Durch die Möglichkeit, Plug-Ins für Eclipse zu entwickeln, kann prinzipiell jede beliebige Funktionalität integriert werden. Neben den Basis-Komponenten, aus denen die Eclipse-Umgebung aufgebaut ist, existiert mittlerweile eine Vielzahl von frei verfügbaren sowie kommerziellen Plug-Ins, welche nahezu an allen Stellen der Entwicklungsplattform andocken können [Wart04]. So sind beispielsweise Komponenten verfügbar, um UML-Modelle zu zeichnen, Sourcecode-Metrik-Analysen durchzuführen oder aspektorientierte Java-Programme mit dem Quasi-Standard AspectJ zu entwickeln. Daher bietet das Komponentenmodell die Möglichkeit, die Eclipse-Umgebung in vielfältiger Weise an die Bedürfnisse eines jeweiligen Benutzers anzupassen (siehe Abb. 2-2).

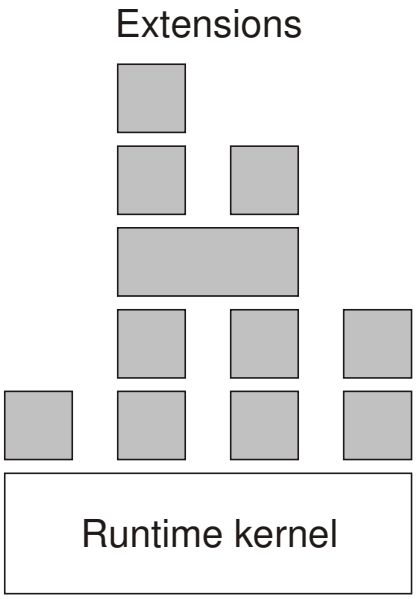


Abb. 2-2: Erweiterbarkeit von Eclipse [GaBe04]

2.3 Erweiterungspunkte

Plug-Ins stellen die kleinste erweiterbare und installierbare Einheit dar und können als Behälter für eine Menge von Erweiterungen (im Eclipse-Kontext "extensions" oder "contributions" genannt) angesehen werden [BaWe04]. Weiterhin kann jedes Plug-In sogenannte Erweiterungspunkte definieren, über die es von anderen Komponenten erweitert werden kann (siehe Abb. 2-3).

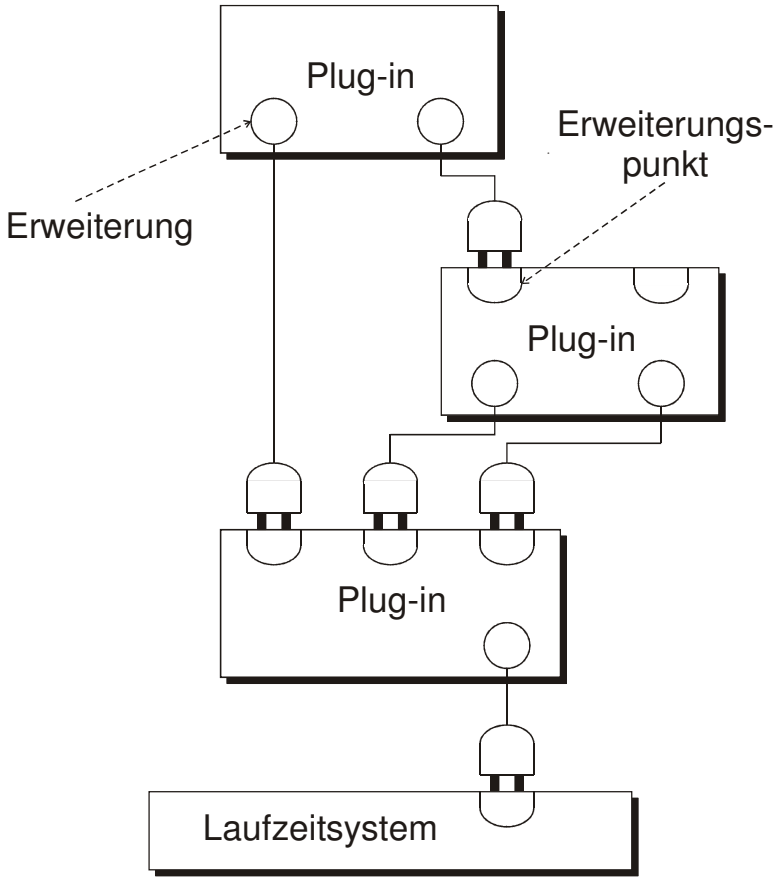


Abb. 2-3: Eclipse-Komponentenmodell [Daum04]

Selbstverständlich wurde auch die Java-Umgebung aus einer Reihe von Plug-Ins entwickelt. Diese erweitern die von Eclipse zur Verfügung gestellten Basiskomponenten, indem auf entsprechende Erweiterungspunkte aufgesetzt wird. So bietet Eclipse beispielsweise einen inkrementellen Java-Compiler, der auf dem inkrementellen Builder der Plattform aufsetzt. Speichert ein Benutzer Änderungen in einer Java-Datei, werden diese Modifikationen dem Builder mitgeteilt und von diesem übersetzt [Weye04].

2.4 OSGi-Standard

Bei der Entwicklung von Eclipse Plug-Ins kann auf bereits existierende Komponenten über deren Erweiterungspunkte zugegriffen werden. Für eine einheitliche Darstellung der Komponenten und um den Zugriff auf diese zu vereinfachen, entsprechen ab der Eclipse-Version 3.0 die Plug-In-Formate dem OSGi-Standard [Open04]. Die OSGi ist eine Initiative zur Standardisierung von Diensten, die auf lokalen Netzwerken und eingebetteten Geräten zum Einsatz kommen [Daum04]. Für Eclipse bedeutet dies, dass das Komponenten-Format fortan auf einem offenen Standard beruht und Plug-Ins aus einem implementierenden und einem deklarativen Teil in Form einer Manifest-Datei bestehen müssen.

Der Kern von Eclipse erfüllt die Rolle eines OSGi-Servers, auf welchem die einzelnen Plug-Ins als OSGi-Dienste laufen. Zu diesem Zweck müssen die Dienste das Interface `BundleActivator` implementieren. Dieses Interface beinhaltet eine `start()`-Methode, über die sich ein Dienst am OSGi-Server registriert, sowie eine `stop()`-Methode zum Abmelden vom Server. In Eclipse erledigt dies die abstrakte Klasse `Plugin`, welche einen `BundleActivator` implementiert. Somit muss man sich als Plug-In-Entwickler kaum um OSGi-Konformität kümmern. Damit auch ältere Plug-Ins weiterhin benutzt werden können, wurde zusätzlich eine Kompatibilitätsschicht eingebaut [Daum04].

Durch die Trennung in deklarative Beschreibung und Implementation der Komponenten kann das aufwändige Laden der konkreten Implementierung von Plug-Ins solange von der Eclipse-Umgebung verzögert werden, bis der Code auch tatsächlich benötigt wird. Gamma/Beck [GaBe04] bezeichnen dies als *lazy loading rule*: "Contributions are only loaded when they are needed". Insbesondere dieses Konzept des verzögerten Ladens ermöglicht es, die Skalierbarkeit der Eclipse-Umgebung zu gewährleisten. Immerhin besteht die Entwicklungsplattform in ihrer aktuellen Version aus über 80 Komponenten. Und kommerzielle Erweiterungen wie etwa "WebSphere Studio Application Developer" von IBM fügen nochmals mehrere hundert Plug-Ins hinzu [BaMe04].

2.5 Ablaufkern und Eclipse-Komponenten

Der Eclipse-Kern ist ebenfalls als Plug-In angelegt und beinhaltet Funktionalitäten, die nötig sind um weitere Komponenten einbinden zu können. Die hierfür benötigten Klassen und Interfaces befinden sich im Paket `org.eclipse.core.runtime`. Dazu gehört auch die abstrakte Klasse `Plugin` (siehe Abb. 2-4), welche Methoden für das Plug-In-Management beinhaltet. Die Entwicklung eines Eclipse Plug-Ins fängt daher normalerweise mit der Implementierung einer Unterklasse dieser abstrakten Klasse an [Bolo03]. Eine Beschreibung der Klasse `Plugin` sowie Beispielcode für ein Plug-In-Rumpfprogramm befinden sich in der zugehörigen API-Referenz, welche über das Eclipse-Hilfesystem[1] aufgerufen werden kann.

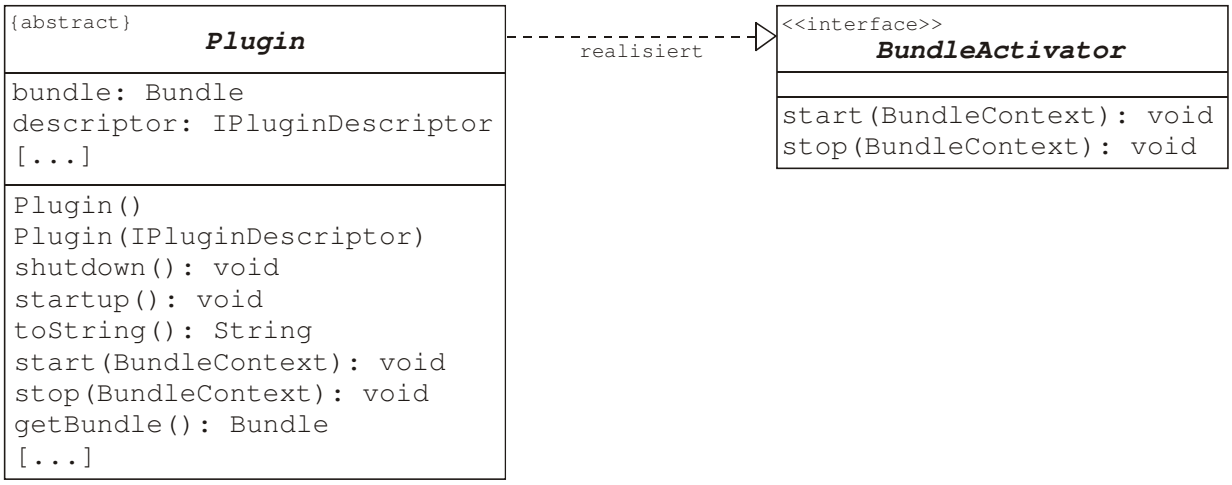


Abb. 2-4: Abstrakte Klasse Plugin und Interface BundleActivator

Ziel der Entwicklung von Plug-Ins ist es nicht, die vorhandenen Komponenten zu ersetzen, sondern vielmehr die gegebene Funktionalität zu erweitern. Gamma/Beck [GaBe04] beschreiben dieses Paradigma als *sharing rule:* "Add, don't replace". Da die Eclipse-Umgebung aus einzelnen Komponenten zusammengesetzt ist, besteht die Möglichkeit, an nahezu jeder beliebigen Stelle aufzusetzen und neue Komponenten hinzuzufügen. Insbesondere kann man bei der Plug-In-Entwicklung auf die Funktionalität existierender Eclipse-Komponenten zurückgreifen. Dazu gehören unter anderem die Ressourcenverwaltung

[1] Aufruf des Hilfesystems über das Menü `Help` → `Help Contents` der Eclipse-Umgebung. Die Plugin-Beschreibung ist unter `Platform Plug-In Developer Guide` → `Reference` → `API Reference` → `org.eclipse.core.runtime` → `Class Summary` → `Plugin` zu finden.

(Projekte, Verzeichnisse, Dateien) der Eclipse-Umgebung, diverse GUI-Komponenten wie Editoren und Views sowie das Hilfesystem [Daum04].

Um auf bereits vorhandene Komponenten der Eclipse-Umgebung aufzusetzen bietet sich ein Blick in die Liste der zur Verfügung stehenden Erweiterungspunkte an. In der "Extension Points Reference" des Eclipse-Hilfesystems befindet sich eine Übersicht[2] aller von Eclipse zur Verfügung gestellten Erweiterungspunkte (siehe Abb. 2-5). Diese sind in verschiedene Kategorien eingeteilt, um einem Entwickler die Suche zu erleichtern. Zur Identifikation der Erweiterungspunkte von Komponenten fremder Hersteller muss natürlich auf eine Dokumentation des entsprechenden Anbieters zurückgegriffen werden.

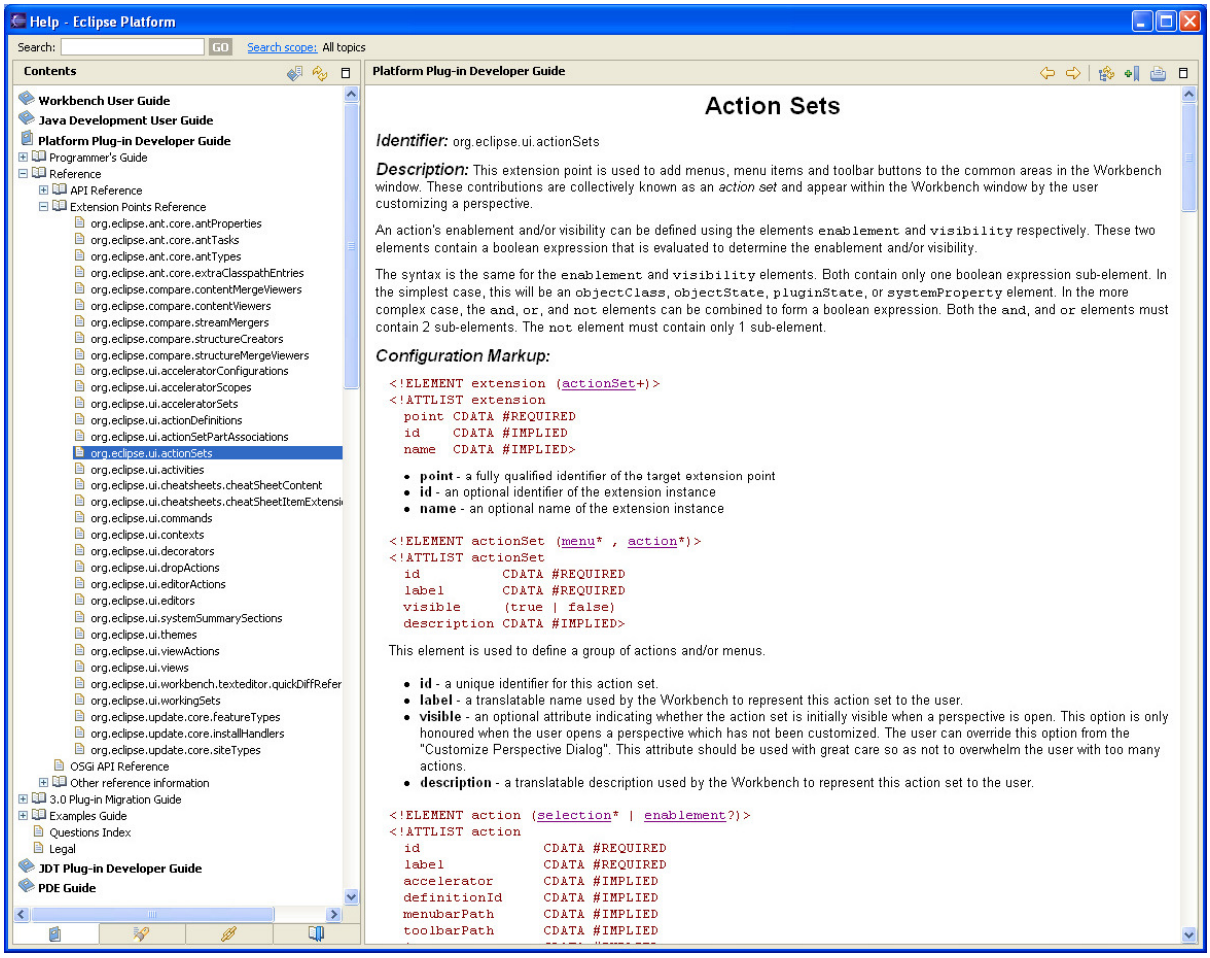


Abb. 2-5: Extension Points Reference

[2] Unter `Platform Plug-In Developer Guide` → `Reference` → `Extension Points Reference` des Eclipse-Hilfesystems befindet sich die Übersicht der Erweiterungspunkte.

Das Paket `org.eclipse.ui` und seine Unterpakete sind hauptsächlich für den Aufbau der Eclipse-Benutzeroberfläche zuständig. Hierüber werden Komponenten wie Views, Texteditoren und Formulare zur Verfügung gestellt [Daum04]. Über die vordefinierten Erweiterungspunkte dieser Plug-Ins kann die Eclipse-Umgebung um neue Elemente erweitert werden. So wird beispielsweise der Erweiterungspunkt `org.eclipse.ui.actionSets` dazu benutzt, so genannte "Action Sets" wie Menüelemente und Schaltflächen in die Eclipse-Oberfläche einzufügen. Eine Beschreibung zusätzlicher Erweiterungspunkte, insbesondere für die Eclipse-Benutzeroberfläche, bietet der "Platform Plug-in Developer Guide" [Plat04].

2.6 Plug-In Manifest

Jede Eclipse-Installation enthält einen Unterordner `plugins`. Alle dort abgelegten Komponenten stehen der Eclipse-Plattform und somit einem Benutzer zur Verfügung. Jedes Plug-In wird in ein eigenes Verzeichnis installiert und beinhaltet diverse Dateien, unter anderem das Plug-In-Manifest [Bolo03]. Dies ist der deklarative Teil einer Eclipse-Komponente.

Das Plug-In-Manifest ist eine XML-Datei, welche stets den Namen `plugin.xml` trägt und das zentrale Element einer Komponente darstellt. Über das Manifest wird die Einbettung des Plug-Ins in die Eclipse-Oberfläche beschrieben und dem Ablaufkern mitgeteilt, was nötig ist um eine Komponente zu aktivieren [Bolo03]. Einen Ausschnitt der Manifest-Datei des Plug-Ins `org.eclipse.ui` zeigt Abb. 2-6. Hier ist zu erkennen, dass ein Manifest in verschiedene Bereiche eingeteilt ist.

```
<plugin
   id="org.eclipse.ui"
   class="org.eclipse.ui.internal.UIPlugin">
   …
   <runtime>
      <library name="ui.jar"/>
      …
   </runtime>
   <requires>
      <import plugin="org.eclipse.core.runtime"/>
      …
   </requires>
   <extension-point
      id="actionSets"
      name="Action Sets"
      schema="schema/actionSets.exsd"/>
   …
</plugin>
```

Abb. 2-6: plugin.xml-Datei der Refactor-Komponente

Das `<plugin>`-Element definiert den Rumpf der XML-Datei. Über das zugehörige Attribut `id` wird einer Komponente eine systemweit eindeutige Bezeichnung zugewiesen und `class` legt den Namen der Java-Klasse fest, welche das Plug-In implementiert. Der `<runtime>`-Abschnitt beinhaltet Informationen für den Eclipse-Ablaufkern. Hier wird beispielsweise über das `<library>`-Element der Name der Bibliothek angegeben, welche die implementierenden Klassen der Komponente beinhaltet. Im `<requires>`-Bereich werden Abhängigkeiten zu anderen Plug-Ins beschrieben, insbesondere kann über `<import>` festgelegt werden, welche zusätzlichen Plug-Ins zur Ausführung der beschriebenen Komponente benötigt werden.

Definiert eine Komponente eigene Erweiterungspunkte, können diese über `<extension-point>` angegeben und mit `id`, `name` und `schema` versehen werden. Über `schema` wird eine XML-Datei referenziert, welche die Schemaspezifikation für diesen Erweiterungspunkt bereitstellt. Diese Spezifikationsdatei kann neben der Beschreibung der benötigten Tags auch eine Dokumentation enthalten, die Entwickler bei der Erzeugung von Erweiterungen unterstützt. Weitere hilfreiche Informationen über den Aufbau der `plugin.xml` befinden sich in der Plug-In Manifest-Beschreibung [Epla04].

2.7 Erstellen von Plug-Ins

Eine Einführung zur Erstellung von Plug-Ins bietet das "Hello World"-Beispiel von Gamma/Beck [GaBe04]. Ziel ist es, die Eclipse-Umgebung um eine Schaltfläche zu erweitern. Um das Beispiel nicht unnötig zu verkomplizieren, wird auf die Erstellung einer komplexen Komponente verzichtet. Stattdessen wird durch Druck auf die Schaltfläche lediglich eine Instanz eines einfachen Dialogfensters erzeugt und dem Benutzer angezeigt.

2.7.1 Erweiterungspunkt identifizieren

In einem ersten Schritt ist das Plug-In der Eclipse-Umgebung zu identifizieren, welches die benötigte Funktionalität bietet, eine zusätzliche Schaltfläche einfügen zu können und einen entsprechenden Erweiterungspunkt anbietet, über den diese Komponente erweitert werden kann. Eine Übersicht liefert die bereits erwähnte "Extension Points Reference" des Eclipse-Hilfesystems. Hierüber lässt sich `org.eclipse.ui.actionSets` als zu bevorzugenden

Erweiterungspunkt identifizieren, welcher für das Erzeugen von Menüs und Schaltflächen genutzt werden kann. Die zugehörige Dokumentation bietet eine detaillierte Beschreibung des Erweiterungspunkts und erläutert die Syntax zur Erstellung eines "Action Sets" in der Manifest-Datei.

2.7.2 Manifest-Datei erstellen

Mittels der Manifest-Datei wird die Erweiterung der Eclipse-Oberfläche um die zusätzlich einzufügende Schaltfläche vorgenommen. Den entsprechenden Ausschnitt der `plugin.xml`-Datei zeigt Abb. 2-7.

```
…
<plugin
  id="org.eclipse.contribution.hello"
  class="org.eclipse.contribution.hello.HelloPlugin">
  …
  <requires>
     <import plugin="org.eclipse.ui"/>
  </requires>
  <extension
    point="org.eclipse.ui.actionSets">
    <actionSet
      label="Hello Action Set"
      id="org.eclipse.contribution.hello.actionSet">
      <action
        label="Hello"
        class="org.eclipse.contribution.hello.HelloAction"
        toolbarPath="helloGroup"
        id="org.eclipse.contribution.hello.HelloAction">
      </action>
    </actionSet>
  </extension>
</plugin>
```

Abb. 2-7: plugin.xml

Das `id`-Attribut im `<plugin>`-Abschnitt weist dem zu erzeugenden Plug-In einen eindeutigen Bezeichner zu und `class` beinhaltet einen Verweis auf die zugehörige Plug-In-Klasse. Über `<requires>` wird die Basis-Komponente `org.eclipse.ui` importiert, da diese den Erweiterungspunkt `actionSets` zur Verfügung stellt.

Den weitaus wichtigeren Teil des Manifests stellt jedoch der `<extension>`-Abschnitt dar. Das Attribut `point` legt `org.eclipse.ui.actionSets` als Erweiterungspunkt fest, auf den das neue Plug-In aufsetzen wird. Über `<actionSet>` wird eine neue Aktionsmenge definiert, wobei `label` einen Klartext-Namen und `id` einen eindeutigen Bezeichner festlegt.

Zuständig für das Erzeugen einer Schaltfläche ist der Bereich `<action>`. Hier wird anhand des `class`-Attributs auf die implementierende Java-Klasse der neuen Komponenten verwiesen, und `toolbarPath` legt die Position der Schaltfläche in der Eclipse-Umgebung fest. Existiert der hier angegebene Bereich noch nicht in der Schaltflächenleiste, wird er neu angelegt und steht damit für zukünftig einzufügende Schaltflächen zur Verfügung. Damit wäre die Erstellung des Komponenten-Manifests abgeschlossen.

2.7.3 Aktionsklasse implementieren

Weiterhin kann aus der "Extension Points Reference" entnommen werden, dass die Komponenten-Klasse das Interface `IWorkbenchWindowActionDelegate` implementieren muss. Dieses Interface ist zuständig für das Durchführen der Schaltflächenaktion. Ein Blick in die API-Referenz zeigt, dass `IWorkbenchWindowActionDelegate` von `IActionDelegate` abgeleitet und hier die Methode `run()` zu finden ist. Diese wird aufgerufen, sobald eine Aktion ausgeführt wird. Demnach ist die `run()`-Methode die richtige Stelle, um ein Dialogfenster zu erzeugen und eine Nachricht auszugeben (siehe Abb. 2-8).

```
…
public class HelloAction implements IWorkbenchWindowActionDelegate {
  …
  public void run(IAction action) {
    MessageDialog.openInformation(null, null, "Hello, Eclipse world");
  }
}
```

Abb. 2-8: HelloAction.java

2.7.4 Erweiterung und Erweiterungspunkt

Die Ähnlichkeit der Begriffe Erweiterung und Erweiterungspunkt verdient eine Anmerkung:
Das Tag `<extension>` in der Manifest-Datei liefert einen Hinweis darauf, dass die aktuell betrachtete Komponente eine Erweiterung einer anderen Komponente vornimmt. Der benutzte Erweiterungspunkt wird durch Angabe von `point="…"` referenziert. Andererseits dient der Tag `<extension-point>` der Definition eines eigenen Erweiterungspunkts, über welchen eine Komponente von anderen Plug-Ins erweitert werden kann. Die Syntax zur Erzeugung einer Erweiterung für diesen Erweiterungspunkt wird in einer Schema-Definition hinterlegt.

2.8 Details der Plug-In-Architektur

Im Folgenden wird das Zusammenwirken zweier Plug-Ins detaillierter erläutert. Als Grundlage dient das zuvor beschriebene Plug-In-Beispiel, welches über den Erweiterungspunkt `actionSets` die Komponente `org.eclipse.ui` erweitert. Abb. 2-9 zeigt die beiden beteiligten Plug-Ins und stellt die Beziehungen untereinander sowie die erforderlichen Elemente jeder Komponente dar.

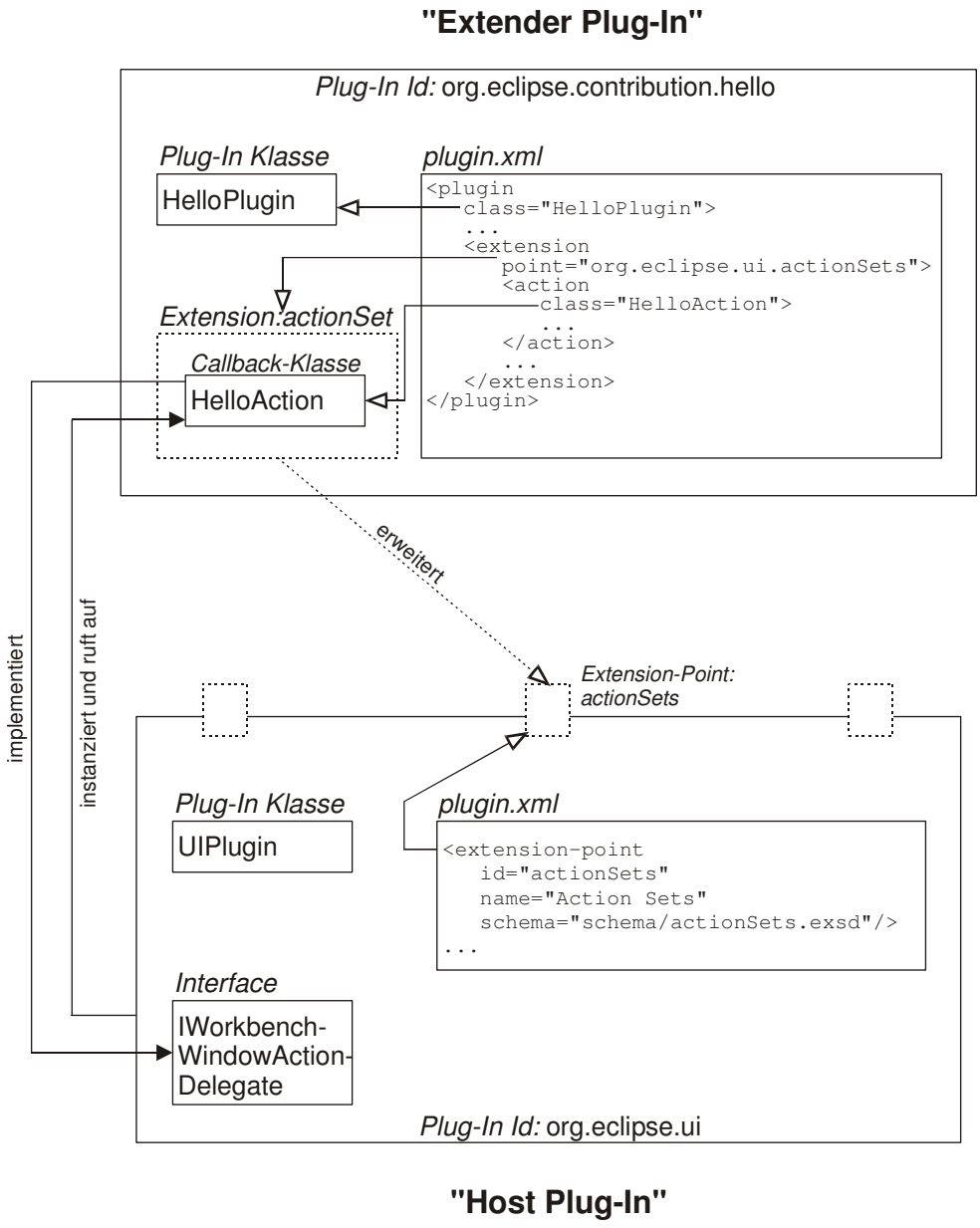


Abb. 2-9: Host- und Extender Plug-In

Erweitert ein Plug-In eine andere Komponente, kommt zwischen den Beteiligten ein sogenannter Vertrag zustande. Im Kontext dieses Erweiterungs-Vertrags können verschiedene Elemente identifiziert werden: *Host Plug-In*, *Extender Plug-In* sowie *Callback-Objekte*.

2.8.1 Host Plug-In

Das *Host Plug-In* stellt Erweiterungspunkte zur Verfügung, über welche das Verhalten beziehungsweise die Funktion der Komponente durch andere Plug-Ins erweitert werden kann. Im vorliegenden Beispiel definiert das Host Plug-In `org.eclipse.ui` unter anderem den Erweiterungspunkt `actionSets`. Die Deklaration erfolgt in der Manifest-Datei *plugin.xml* durch das XML-Element `<extension-point>`. Die Syntax zur Deklaration von Menüelementen oder Schaltflächen, die der Eclipse Benutzeroberfläche hinzugefügt werden können, ist in einer Schema-Definition, in diesem Fall namens `actionSets.exsd`, hinterlegt. Diese spezifiziert, wie die Komponente zu erweitern ist, welche Attribute dem Host Plug-In zu übergeben sind und welches *Interface* zu implementieren ist. Das Host Plug-In selbst beinhaltet neben diesem Interface eine Menge von Klassen, welche die Funktionalität der Komponente bereitstellen, sowie eine Plug-In Klasse. Die Hauptfunktion dieser *Plug-In Klasse* ist es, während der Komponenten-Aktivierung beziehungsweise -Deaktivierung spezielle Aufgaben zu erledigen wie beispielsweise das Reservieren oder Freigeben von Ressourcen. Wird diese Funktionalität für eine Komponente nicht benötigt, erübrigt sich das Erstellen einer solchen Plug-In Klasse und die Eclipse Runtime erzeugt ein "default plug-in runtime object" [Bolo03].

2.8.2 Extender Plug-In

Das *Extender Plug-In* erzeugt eine Erweiterung des Host Plug-Ins durch Angabe des XML-Elements `<extension>` in der *plugin.xml* und referenziert den betreffenden Erweiterungspunkt über das Attribut `point`. Die weitere Deklaration einer Erweiterung wird gemäß der im Host Plug-In hinterlegten Schemadefinition des korrespondierenden Erweiterungspunkts erstellt. Im vorliegenden Beispiel erfordert die Schemadefinition des Erweiterungspunkts `actionSets` unter anderem die Angabe einer Klasse, welche aufgerufen wird sobald eine Schaltflächenaktion eintritt. Dies übernimmt die Klasse `HelloAction`, welche zugleich das Interface `IWorkbenchWindowActionDelegate` des Host Plug-Ins implementiert. Sind während des Startens oder Beendens des Extender Plug-Ins Aktionen auszuführen, muss hier ebenfalls eine *Plug-In-Klasse* (`HelloPlugin`) erzeugt werden.

2.8.3 Callback-Objekte

Über *Callback-Objekte* findet die Kommunikation zwischen Host- und Extender Plug-In statt [Bolo03]. Callback-Objekte sind – im Gegensatz zu Plug-Ins – normale Java-Objekte, die durch das Host Plug-In aufgerufen werden sobald bestimmte Ereignisse eintreten. Das Interface für Callback-Objekte wird typischerweise seitens des Host Plug-Ins angeboten und innerhalb des Extender Plug-Ins implementiert. Die Existenz von Callback-Objekten wird bestimmt durch die Schema-Definition eines Erweiterungspunkts. Im Fall von `actionSets` muss daher innerhalb der Manifest-Datei des Extender Plug-Ins die entsprechende *Callback-Klasse* (`HelloAction`) benannt werden, welche das Interface `IWorkbenchWindowActionDelegate` des Host Plug-Ins implementiert.

2.8.4 Interner Ablauf

Der interne Ablauf gestaltet sich folgendermaßen: Wird ein Plug-In aktiviert, müssen alle Extender Plug-Ins identifiziert werden, welche die definierten Erweiterungspunkte des Host Plug-Ins nutzen. Im konkreten Fall von `org.eclipse.ui` müssen demnach alle Komponenten gefunden werden, die beispielsweise den Erweiterungspunkt `actionSets` referenzieren. Dies geschieht über den Zugriff auf das Plug-In Repository von Eclipse, welches während des Startens der Entwicklungsumgebung alle zur Verfügung stehenden Plug-Ins registriert. Hierüber kann das Extender Plug-In `org.eclipse.contribution.hello` ermittelt und die Deklaration der entsprechenden Erweiterung verarbeitet werden. Aus den extrahierten Informationen wird daraufhin eine Schaltfläche erzeugt und in die Eclipse-Oberfläche integriert. Wird nun seitens des Host Plug-Ins eine Schaltflächenaktion registriert, kann das korrespondierende Callback-Objekt instanziert, die dort implementierte Methode `run()` aufgerufen und somit die seitens des Extender Plug-Ins gewünschte Button-Reaktion ausgeführt werden. Abb. 2-10 zeigt den soeben beschrieben Ablauf nochmals in Form eines Sequenzdiagramms.

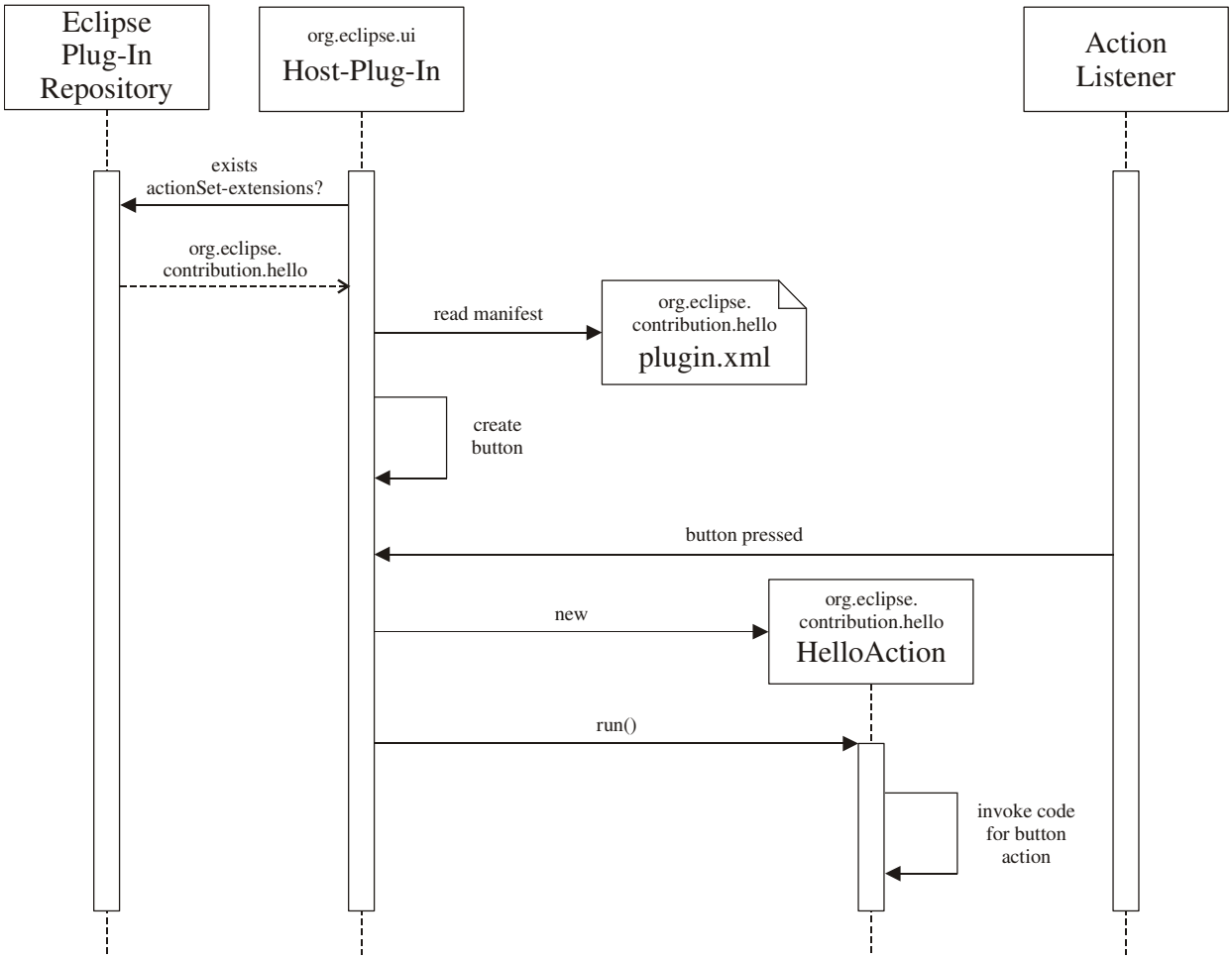


Abb. 2-10: Plug-In Sequenzdiagramm

Selbstverständlich liefert diese Beschreibung lediglich einen oberflächigen Eindruck der Erweiterungsverarbeitung, da intern verschiedene Implementierungstechniken angewandt werden und an dieser Stelle nicht näher erläutert werden können. Daher sei für einen detaillierteren Einblick auf [Bolo03] verwiesen, wo das Erzeugen von Erweiterungspunkten an einem anschaulichen Beispiel schrittweise erläutert sowie das Identifizieren und Verarbeiten der Manifest-Deklarationen von Erweiterungen anhand von Codefragmenten verdeutlicht wird.

2.9 Plug-In Development Environment (PDE)

Für die Eclipse-Plattform sind inzwischen eine Reihe nützlicher Plug-Ins[3] entstanden. Natürlich müssen diese nicht, insbesondere wenn es sich um komplexe und umfangreiche Komponenten handelt, ohne Unterstützung seitens Eclipse erstellt werden. Um Entwickler von Plug-Ins bei ihrer Arbeit behilflich zu sein, verfügt Eclipse über die "Plug-In Development Environment"-Komponente (PDE). Diese besteht aus einem Assistenten zur Erstellung von Plug-In Projekten, speziellen Editoren für die Manifest- sowie Schemadefinitionsdatei und stellt eine zweite Workbench (*runtime workbench*) zum Testen eines in der Entwicklung befindlichen Plug-Ins bereit [Pdeg04]. Weiterhin verwendet PDE die bei der Erzeugung von Erweiterungspunkten erstellte Schemadefiniton und unterstützt auf diese Weise die Entwicklung von Plug-Ins, indem die für den benutzten Erweiterungspunkt benötigten Tags abgefragt werden. Mit PDE ist es daher möglich, Eclipse Plug-Ins auf komfortable Weise zu entwickeln.

[3] Im Laufe der Zeit wurden eine Vielzahl von Komponenten für die Eclipse-Umgebung entwickelt und stehen Benutzern meist frei zugänglich zur Verfügung. Da beinahe täglich neue Plug-Ins auftauchen, lohnt sich ein regelmäßiger Blick auf folgende Internetseiten:
www.eclipse.org, *www.eclipse-plugins.info*, *www.eclipseproject.de*.

Kapitel 3
Sourcecode-Modell

Thema des vorliegenden Kapitels ist eine einführende Beschreibung der in Eclipse enthaltenen Java Development Tools sowie der zur Verfügung stehenden Funktionalität zur Erzeugung eines abstrakten Syntaxbaums. Dieser ermöglicht die modellhafte Darstellung eines Sourcecodes und bildet somit die Basis zur Beschreibung von Codetransformationen auf Metamodell-Ebene.

Ein anfänglicher Blick auf das Eclipse Modeling Framework[4] (EMF) zeigte, dass die dort vorhandenen Modellierungsmöglichkeiten unzureichend zur Beschreibung von Sourcecode sind. EMF bietet lediglich Modellierungskonzepte analog zu Klassendiagrammen und verfügt daher nicht über den benötigten Detaillierungsgrad. Aufgrund dessen ist EMF für das weitere Vorgehen ungeeignet und es muss eine Möglichkeit gefunden werden, die Struktur eines Java-Programms zu extrahieren, um auf diesem Weg ein geeignetes Metamodell zu finden. Daher werden im Folgenden die in Eclipse enthaltenen Java Development Tools (JDT) betrachtet.

3.1 Java Development Tools (JDT)

Die Java-Entwicklungswerkzeuge (JDT) beinhalten Komponenten, mit denen auf Basis der Eclipse-Plattform eine vollständig integrierte Java-Entwicklungsumgebung entsteht. Die Plug-Ins verfügen über APIs, so dass diese Komponenten von Drittanbietern genutzt werden können. Die von JDT bereitgestellten Funktionalitäten untergliedern sich in drei Haupt-Kategorien: *JDT User Interface*, *JDT Debug* und *JDT Core* [Jdtp04].

Das Paket *JDT User Interface* implementiert die Java-spezifischen Aspekte der Eclipse-Benutzeroberfläche, insbesondere die verschiedenen Sichten auf ein Java-Projekt, bietet Assistenten zur Erstellung von Java-Elementen an und stellt einen Java Editor bereit. Vorhanden sind auch Refactoring-Tools zum Suchen, Umbenennen und Ändern von Referenzen.

[4] siehe http://www.eclipse.org/emf/

Das Paket *JDT Debug* stellt diverse Komponenten zum Debuggen von Java-Code zur Verfügung und erlaubt die Ausführung von Programmcode auf einer benutzerdefinierten virtuellen Maschine, sofern diese konform zur Java Plattform Debugger Architektur ist [Weye04].

Besonders hilfreich zur Repräsentation der Java-Sourcecode-Struktur sind die Klassen des Pakets *JDT Core*. Diese sind für die Definition der Infrastruktur unterhalb der Eclipse-Benutzeroberfläche zuständig und bieten daher verschiedene Möglichkeiten zur Analyse einer Java-Datei [Danj04]. Dazu zählt unter anderem ein Java-Modell, welches APIs zur Navigation im Java-Elementbaum bietet und über das Paket `org.eclipse.jdt.core` zur Verfügung steht. Eine Unterstützung zur Erstellung eines abstrakten Syntaxbaums (AST), der zur Untersuchung der Struktur einer Java-Datei bis hinab auf Token-Ebene verwendet werden kann, bietet das Paket `org.eclipse.jdt.core.dom`.

3.2 JDT-Core: Java-Modell

Informationen über die interne Struktur eines Java-Quellcodes lassen sich über das von JDT zur Verfügung gestellte Java-Modell extrahieren. Das Paket `org.eclipse.jdt.core` definiert diejenigen Klassen, welche für den Aufbau eines Java-Programms benötigt werden. Die Struktur wird aus dem Klassenpfad des zu untersuchenden Java-Projekts abgeleitet und steht als hierarchisches Objektmodell zur Verfügung [Jdtp04]. Die verschiedenen im Java-Modell enthaltenen Elemente zeigt Abb. 3-1.

Element	**Description**
IJavaModel	Represents the root Java element, corresponding to the workspace. The parent of all projects with the Java nature. It also gives you access to the projects without the java nature.
IJavaProject	Represents a Java project in the workspace. (Child of IJavaModel)
IPackageFragmentRoot	Represents a set of package fragments, and maps the fragments to an underlying resource which is either a folder, JAR, or ZIP file. (Child of IJavaProject)
IPackageFragment	Represents the portion of the workspace that corresponds to an entire package, or a portion of the package. (Child of IPackageFragmentRoot)
ICompilationUnit	Represents a Java source (.java) file. (Child of IPackageFragment)

Abb. 3-1: Elemente des JDT Java-Modells [Jdtp04]

Element	Description
IPackageDeclaration	Represents a package declaration in a compilation unit. (Child of ICompilationUnit)
IImportContainer	Represents the collection of package import declarations in a compilation unit. (Child of ICompilationUnit)
IImportDeclaration	Represents a single package import declaration. (Child of IImportContainer)
IType	Represents either a source type inside a compilation unit, or a binary type inside a class file.
IField	Represents a field inside a type. (Child of IType)
IMethod	Represents a method or constructor inside a type. (Child of IType)
IInitializer	Represents a static or instance initializer inside a type. (Child of IType)
IClassFile	Represents a compiled (binary) type. (Child of IPackageFragment)

Abb. 3-1: Elemente des JDT Java-Modells [Jdtp04] (Forts.)

Einige der Elemente werden durch den "Package Explorer" der Eclipse-Oberfläche visualisiert (siehe Abb. 3-2), andere korrespondieren mit den Elementen, aus welchen eine Java-Datei (sog. Compilation Unit) besteht und sind im "Outline View" der Eclipse-Umgebung wiederzufinden (siehe Abb. 3-3).

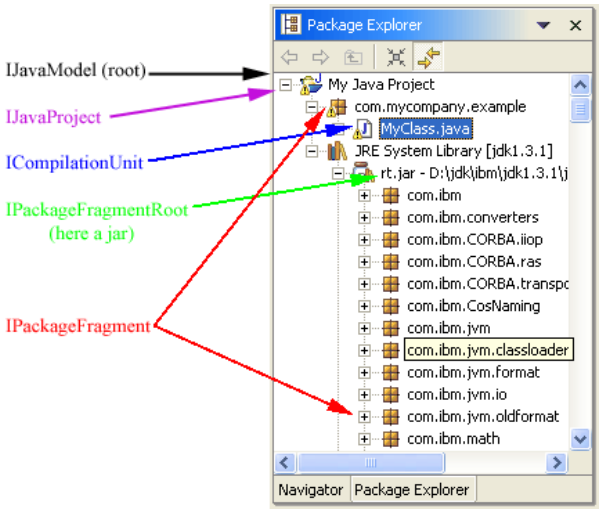


Abb. 3-2: Elemente im Package Explorer

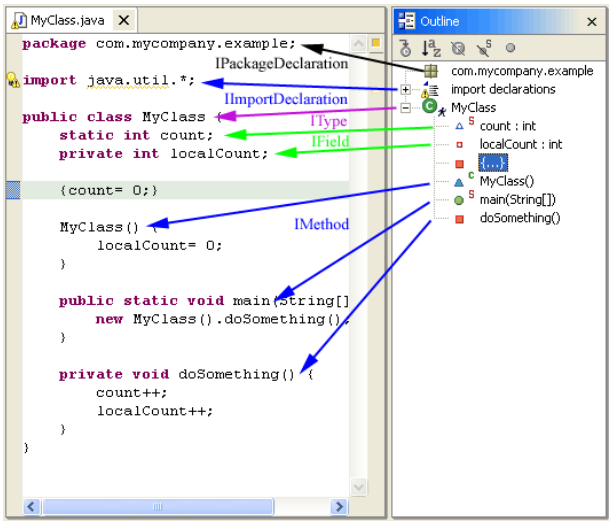


Abb. 3-3: Elemente im Outline View

Ein Eclipse Plug-In zum Extrahieren der internen Struktur eines Java-Programms wurde im Rahmen des *Mrs.G*-Projekts an der University of California, Irvine unter Leitung von Prof. Susan Elliot Sim entwickelt [Mrsg04]. Insbesondere die von Sukanya Ratanotayanon

entwickelte *JavaExtractor*-Komponente traversiert das Java Modell eines Quellcodes und stellt die Struktur in Form eines GXL-Dokuments dar. Eine Betrachtung dieses Plug-Ins ergab allerdings, dass das von JDT zur Verfügung gestellte Java-Modell nicht über den benötigten Detaillierungsgrad zur Beschreibung von Transformationen auf dieser Abstraktionsebene bietet. Daher wird im Folgenden ein Sourcecode-Modell in Gestalt eines abstrakten Syntaxbaums betrachtet.

3.3 JDT-Core: Abstrakter Syntaxbaum

Compiler verwenden im Allgemeinen eine abstrakte Syntax, um die interne Struktur eines Programms darzustellen. Typische Elemente dieser Syntax sind beispielsweise Anweisungen, Ausdrücke oder Bezeichner. Entspricht die Repräsentation einer Baumstruktur, wird diese als abstrakter Syntaxbaum (*engl.* abstract syntax tree, AST) bezeichnet [BaMe04].

Ein AST unterstützt die syntaktische Analyse eines Java-Quellcodes und steht einem Benutzer durch das Paket `org.eclipse.jdt.core.dom` zur Verfügung. Anhand eines ASTs lassen sich im Gegensatz zum Java-Modell eine Vielzahl weiterer Elemente identifizieren, unter anderem lokale Variablen oder if-Anweisungen innerhalb von Methoden. Insgesamt stehen über sechzig Java-Sprachelemente zur Verfügung [Danj04]. Allerdings repräsentiert ein AST nicht alle Elemente der Java-Sprachspezifikation: Der JDT-Syntaxbaum ignoriert Kommentare und enthält folglich nicht genau den gleichen Inhalt wie der geparste Java-Quellcode [Shav04].

Der AST enthält einen Wurzelknoten, der eine Java-Datei (sog. Compilation Unit) repräsentiert. Dieser besitzt Unterknoten für die in der Datei deklarierten Typen. Die Unterknoten eines Typknoten sind wiederum Knoten für Felder, Methoden oder geschachtelte Typen [BaMe04]. Dazu gehören beispielsweise `MethodDeclaration`, `ExpressionStatement` oder `SimpleName`, welche Elemente einer `CompilationUnit` darstellen.

3.3.1 AST-Parser

Das Erzeugen eines ASTs erfolgt durch einen `ASTParser`. Typischerweise wird der Syntaxbaum für eine in der Eclipse-Umgebung geöffnete Java-Datei generiert. Der `ASTParser` ermöglicht es einem Benutzer außerdem einen abstrakten Syntaxbaum für eine class-Datei oder

eine externe Datei anzulegen. Eine weitere nützliche Eigenschaft ist das Erstellen eines benutzerspezifischen ASTs, welcher ausschließlich Elemente enthält, die für weitere Betrachtungen nötig sind. So besteht die Möglichkeit, lediglich Informationen über eine einzelne Methode zu sammeln, ohne den gesamten AST einer Java-Datei anfertigen zu müssen [ArLa04].

3.3.2 AST-Visitor

Ein bereits erstellter AST kann mit einem Visitor traversiert werden. JDT stellt für diesen Zweck einen `ASTVisitor` bereit, welcher für jeden Knotentyp eine entsprechende `visit()`- sowie eine `endVisit()`-Methode beinhaltet [BaMe04]. `Visit()` wird aufgerufen, bevor ein Knoten traversiert wird und liefert standardmäßig `true` als Rückgabewert. Wird die Methode überschrieben und `false` zurückgegeben, erfolgt kein Besuch des entsprechenden Knotens. Ist das Traversieren eines Element-Knotens und seiner Nachfolger abgeschlossen, wird die Methode `endVisit()` aufgerufen und es können entsprechende Benutzer-Aktionen durchgeführt werden.

Ein Beispiel für die Implementierung eines Visitors zur Traversierung eines ASTs sowie Auswertung spezieller Elementknoten zeigt Abb. 3-4. Es werden die in jeder Methode vorkommenden Methodenaufrufe gezählt und beim Verlassen des Methodendeklarations-Knotens die Summe der Aufrufe in einer Konsole ausgegeben [GaBe04].

```
class CallsPerMethodVisitor extends ASTVisitor {
  int callCount = 0;

  public boolean visit(MethodDeclaration node) {
    callCount = 0;
    return super.visit(node);
  }

  public void endVisit(MethodDeclaration node) {
    System.out.println("Calls in " + node.getName() + ":" + callCount);
  }

  public boolean visit(MethodInvocation node) {
    callCount++;
    return true;
  }
}
```

Abb. 3-4: CallsPerMethod-Visitor

Für den Fall, dass ein Visitor nicht an speziellen Elementen interessiert ist, stehen zwei allgemeine Methoden zur Verfügung: `preVisit()` und `postVisit()`. Diese beiden Methoden werden vor dem Betreten beziehungsweise nach dem Verlassen eines jeden Elementknotens aufgerufen. Ohne die `visit()`-Methode eines jeden Elements überschreiben zu müssen, können Benutzer `preVisit()` und `postVisit()` verwenden, um eine allgemeine Traversierung eines ASTs zu erreichen [GaBe04]. Im folgenden Beispiel (siehe Abb. 3-5) wird die Implementierung eines Visitors veranschaulicht, welcher `postVisit()` überschreibt und beim Verlassen eines jeden Elementknotens einen Zähler inkrementiert [GaBe04].

CountingVisitor:

```
class CountingVisitor extends ASTVisitor {
  int count = 0;
  public void postVisit(ASTNode node) {
    count++;
  }
  public int getCount() {
    return count;
  }
}
```

Aufruf:

```
CompilationUnit cu;
CountingVisitor visitor = new CountingVisitor();
cu.accept(visitor);
System.out.println(visitor.getCount());
```

Abb. 3-5: Counting-Visitor und entsprechender Aufruf

3.4 Bindungen

Eine Bindung (*engl.* binding) ist eine Zuordnung von Attributen zu einem Bezeichner. Über eine Variablenbindung wird insbesondere einer Variablen x ihr entsprechender Typ (bspw. integer) zugewiesen. Diese Zuordnung wird typischerweise von einem Compiler erzeugt und in einer Liste (sog. Symboltabelle) hinterlegt, welche die Bezeichner und die ihnen im Programm zugewiesenen Bedeutungen, wie beispielsweise ihren Datentyp sowie ihr Vorkommen enthält.

Neben den Elementen, aus denen eine Java-Datei besteht, beinhaltet der JDT-AST zusätzlich Informationen über Pakete, Methoden, Variablen sowie Typen in Form von Bindungen. Diese Informationen werden beim Generieren des ASTs gesammelt und entsprechenden Java-Elementen zugeordnet. Anhand von Bindungen ist unter anderem die Unterscheidung zwischen lokaler und globaler Deklaration einer Variablen möglich.

3.5 Visualisierung eines ASTs

Eine visuelle Darstellung eines Abstract Syntax Trees ermöglicht das Eclipse Plug-In "ASTView". Diese Komponente kann über die Eclipse-Entwicklerseite geladen und in die Eclipse-Umgebung eingebunden werden [Astv05]. ASTView erstellt den Syntaxbaum zu einer ausgewählten Java-Datei und zeigt diesen in einem Bereich der Eclipse-Oberfläche an. Den von ASTView erzeugten Syntaxbaum des Counting-Visitors aus Kapitel 3.2 ist in Abb. 3-6 zu sehen.

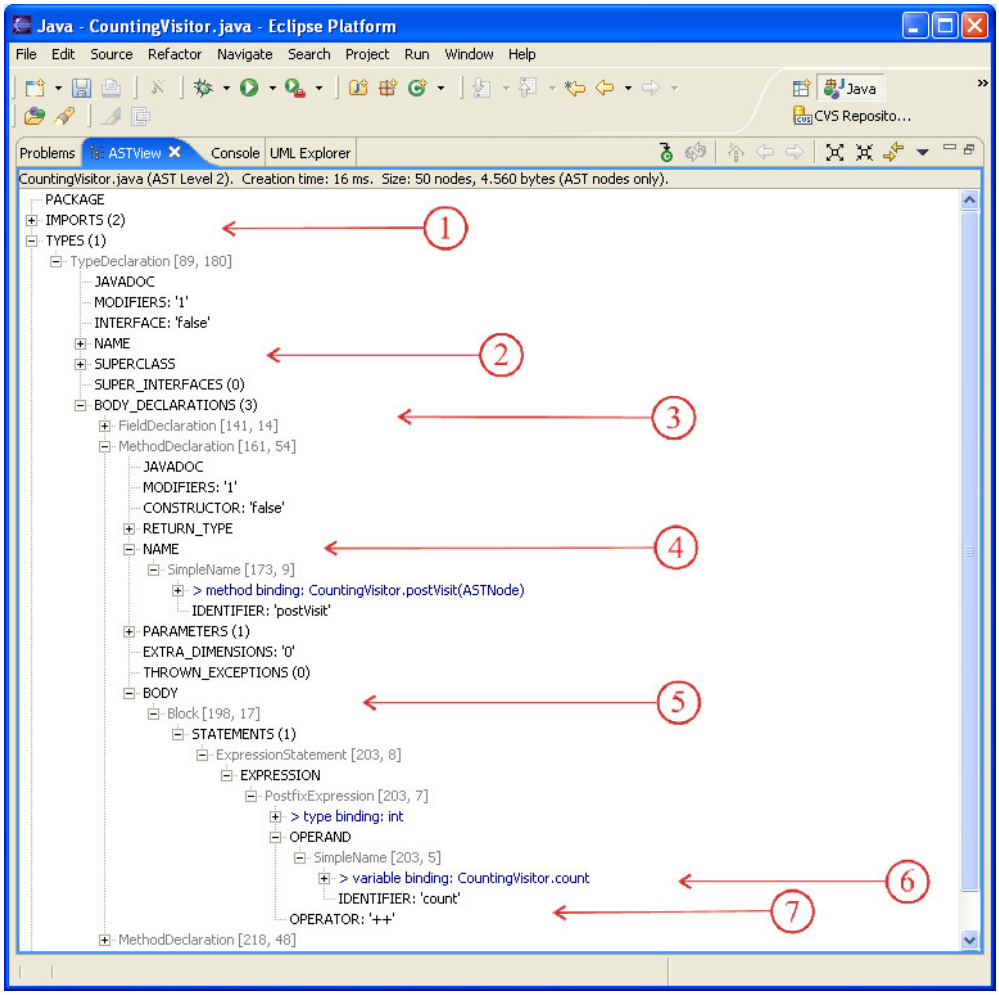


Abb. 3-6: Abstract Syntax Tree zu CountingVisitor.java

Ausgehend von einer `CompilationUnit` beschreibt der AST die in einer Source-Datei enthaltenen Paket-, Import- und Typdeklarationen (1). Eine Typdeklaration beinhaltet Informationen über den Aufbau der Klasse, insbesondere bezeichnet der Unterknoten

`Superclass` (2) die Elternklasse von "CountingVisitor". Weiterhin ist eine Klassenrumpfdeklaration (3) vorhanden, die sich aus `FieldDeclaration` und `MethodDeclaration` zusammensetzt. Innerhalb der Felddeklaration ist die globale Definition der Variablen "count" hinterlegt. Die Methodendeklaration hingegen beinhaltet unter anderem den Namen sowie den Rückgabetyp der Methode "postVisit()" (4). Weiterhin sind Informationen über den Aufbau des Methodenrumpfs vorhanden (5). Hier ist beispielsweise zu erkennen, dass sich die Methode "postVisit()" aus einem Postfix-Ausdruck, bestehend aus dem Bezeichner "count" und dem Operator "++" (7), zusammensetzt.

Neben der Darstellung der Struktur eines Java-Programms enthält der JDT-AST verschiedene Bindungsinformationen. So ist beispielsweise in der Variablen-Bindungsinformation des `SimpleName`-Elements "count" (6) durch Angabe von "CountingVisitor.count" ein Verweis auf dessen globale Deklaration hinterlegt.

3.6 Sourcecode-Manipulation

Auch Code-Manipulationen sind anhand eines ASTs möglich. Aus diesem Grund beinhaltet jeder Knotentyp geeignete Zugriffmethoden. Beispielsweise stellt ein Methoden-Deklarations-Element die beiden Methoden `getBody()` und `setBody()` zur Verfügung. Hierüber kann einerseits der Rumpf einer Methode ausgelesen, andererseits aber auch Änderungen durchgeführt werden.

Eine weitere Manipulationsmöglichkeit bietet der Einsatz des von JDT zur Verfügung gestellten `ASTRewriter`, welcher im Paket `org.eclipse.jdt.core.dom.rewrite` enthalten ist. Dieser sammelt Modifikationsbeschreibungen für einzelne Knoten und übersetzt die Änderungen, sodass sie zu einem späteren Zeitpunkt mit dem Original-Sourcecode zusammengeführt werden können.

Zusätzliche Informationen bezüglich Code-Modifikationen auf Basis des JDT-ASTs befinden sich im Kapitel "Manipulating Java Code" des JDT Plug-In Developer Guide [Jdtp04]. Die dort ebenfalls vorgestellten Codefragmente sind für die abschließende Implementation eines Eclipse Plug-Ins insbesondere zur Erzeugung des modifizierten Sourcecodes maßgeblich.

3.7 Java-Grammatik

Der JDT-AST ermöglicht neben einer Darstellung der syntaktischen Struktur einer Java-Datei das Identifizieren von Bindungen. Damit gewährleistet er einen detaillierten Einblick in die zugrunde liegende Gestalt eines Java-Quellcodes und stellt somit ein umfassendes Modell in Form eines Graphen zur Verfügung. Der Aufbau eines Syntaxbaums wird bestimmt durch das Java-Metamodell, welches durch eine kontextfreie Grammatik beschrieben werden kann.

Eine nützliche Hilfe zur Herleitung dieser Java-Grammatik ist das Klassendiagramm des JDT-Core-Pakets. Ein solches kann mit "Together® – Edition for Eclipse" der Firma Borland® teilweise aus dem Sourcecode erzeugt werden [BoTo05]. Together® ist ein kommerzielles Eclipse-Plug-In, welches eine integrierte UML-Modellierungsumgebung zum Entwurf neuer Applikationen sowie zur Extraktion von Entwurfsinformationen bereits erstellter Projekte bietet und über die Seite des Herstellers als Testversion zur Verfügung steht.

Einen Ausschnitt aus dem von "Together" erzeugten AST-Klassendiagramm zeigt Abb. 3-7. Hierüber lässt sich die Generalisierungshierarchie des Pakets `org.eclipse.jdt.core.dom` erkennen sowie `ASTNode` als Wurzelklasse aller Java-Elemente identifizieren.

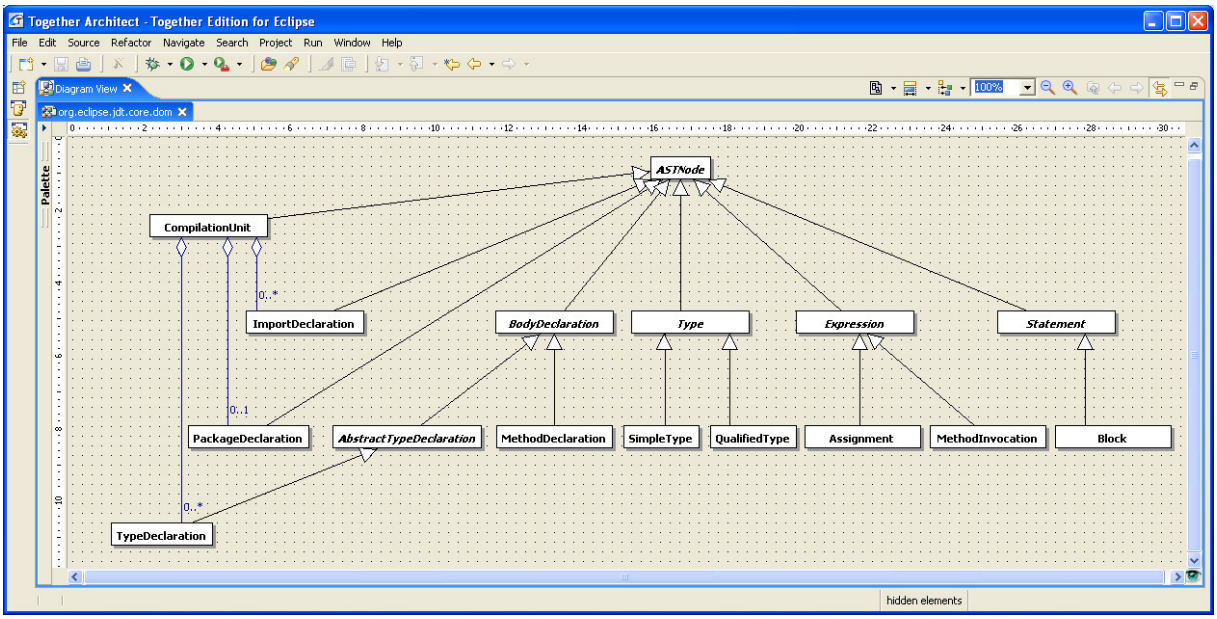


Abb. 3-7: Klassendiagramm zu org.eclipse.jdt.core.dom

Die Grammatik bestimmt die Gestalt von Java-Programmen und ist eine Anleitung für den Aufbau von Modellen, wie beispielsweise eines ASTs. Basierend auf den Klassen des Pakets `org.eclipse.jdt.core.dom` sowie unter Zuhilfenahme der entsprechenden API-Referenz lässt sich die Java-Grammatik extrahieren. Sie basiert auf JDT-Release 3.0 und bezieht sich auf JDK 1.4.

Die in EBNF vorliegende Java-Grammatik zeigt auszugsweise Abb. 3-8. Hier sind ausgehend von einer Compilation Unit diejenigen Elemente dargestellt, welche ebenfalls in dem von *ASTView* erzeugten Syntaxbaum (siehe Abb. 3-6) zu erkennen sind. Die eckigen Klammern stehen für die optionale Existenz eines Ausdrucks, die geschweiften Klammern bezeichnen ein n-maliges (n≥0) Vorkommen, der senkrechte Strich beschreibt die alternative Darstellung und Terminalsymbole werden durch Hochkommas aufgeführt. Eine Übersicht der kompletten Java-Grammatik ist in Anhang A hinterlegt.

```
CompilationUnit::=          [ PackageDeclaration ] { ImportDeclaration }{ TypeDeclaration | ';' }

PackageDeclaration::=       'package' Name ';'

ImportDeclaration::=        'import' Name [ '.*' ] ';'

TypeDeclaration::=          ClassDeclaration | InterfaceDeclaration

ClassDeclaration::=         [ Javadoc ] { Modifier } 'class' Identifier
                                [ extends Type ]
                                [ implements Type { ',' Type } ]
                                '{' { ClassBodyDeclaration | ';' } '}'

ClassBodyDeclaration::=     BodyDeclaration

BodyDeclaration::=          AbstractTypeDeclaration | AnnotationTypeMemberDeclaration |
                            EnumConstantDeclaration | FieldDeclaration | Initializer |
                             MethodDeclaration | ConstructorDeclaration

MethodDeclaration::=        [ Javadoc ] { Modifier } ( Type | 'void' ) Identifier
                                '(' [ FormalParameter { ',' FormalParameter } ] ')'
                                [ 'throws' TypeName { ',' TypeName } ] ( Block | ';' )
```

Abb. 3-8: Auszug aus Java-Grammatik

3.8 AST-Metamodell

Die visuelle Darstellung des Metamodells zeigt auszugsweise Abb. 3-9. Es beschreibt die Gestalt des zu einer Compilation Unit korrespondieren ASTs. Das Metamodell ist somit eine Aufbauanleitung für alle JDT-ASTs, d.h. ein zu einer vorliegenden Java-Datei generierter AST ist eine Instanz dieses Metamodells.

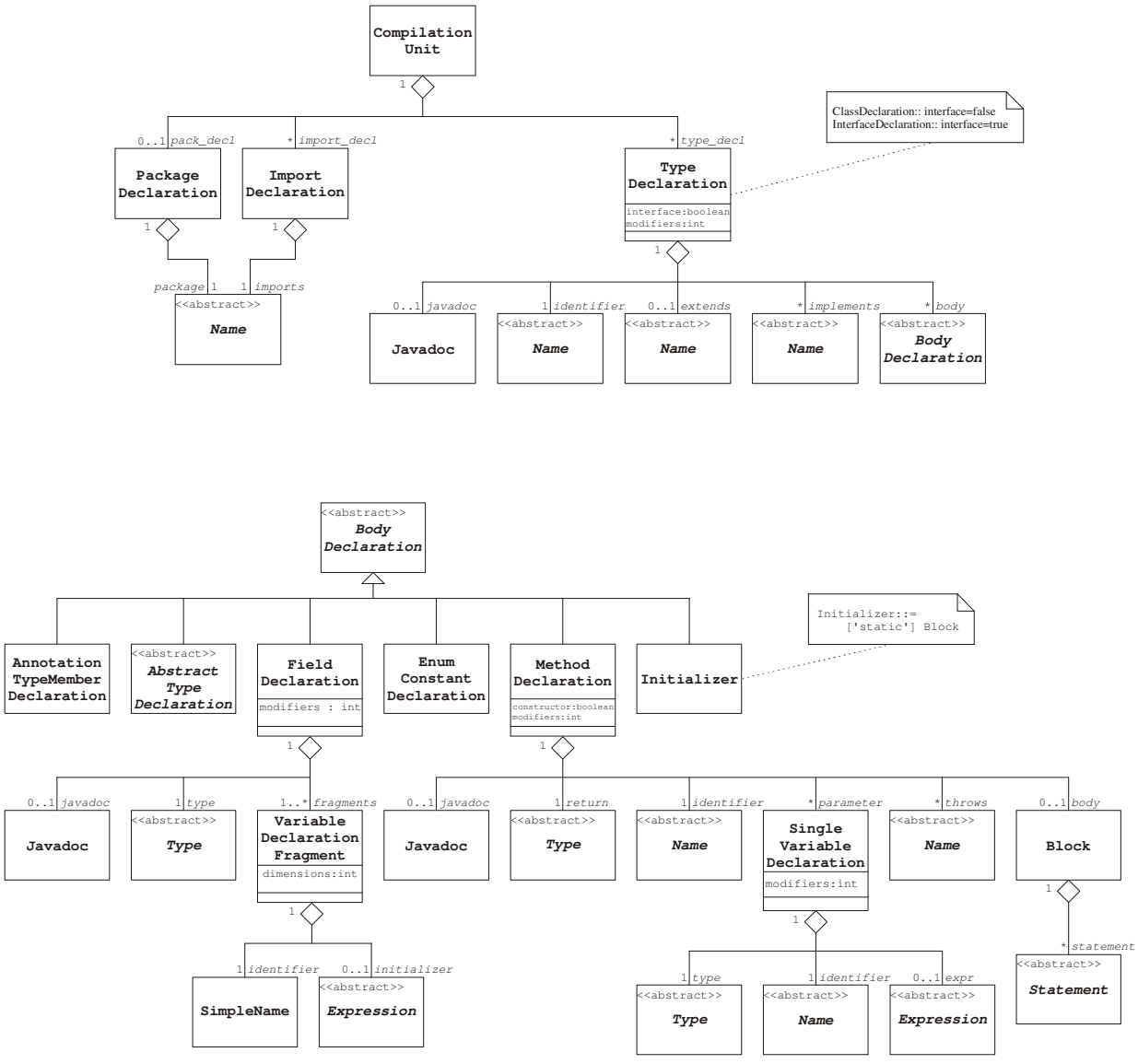


Abb. 3-9: Auszug aus AST-Metamodell

Das Metamodell beschreibt beispielsweise die Struktur des in Abb. 3-6 dargestellten Syntaxbaums korrespondierend zu `CountingVisitor.java`. Die AST-Knoten auf höchster Hierarchieebene repräsentieren *PackageDeclaration*, *ImportDeclaration* sowie *TypeDeclaration* der Compilation Unit.

Die *TypeDeclaration* besteht aus den AST-Knoten *Javadoc* und *BodyDeclaration* sowie *Name*-Elementen, welche den Identifier, die Superklasse sowie das Superinterface der korrespondierenden Java-Datei repräsentieren. Das Attribut *modifiers* des AST-Knotens *TypeDeclaration* beschreibt die Modifikatoren kodiert als Integer-Wert und *interface* verweist auf eine vorliegende Klassen- oder Interfacedefinition.

Die *BodyDeclaration* besteht aus *FieldDeclaration* und *MethodDeclaration*, welche wiederum weitere Kindknoten enthält: *Javadoc* repräsentiert vorhandene Java-Dokumentation, *Type* beschreibt den Rückgabetyp der Methode, die beiden *Name*-Elemente bezeichnen den Methodenname sowie die Throws-Exception, *FormalParameter* enthält die Argumente der Methode und *Block* stellt den Methodenrumpf dar. Das Attribut *modifiers* kodiert den Modifikator als Integer-Wert und *constructor* weist auf eine vorliegende Methoden- oder Konstruktordefinition hin.

Das vorliegende Metamodell, welches in vollem Umfang in Anhang B zu finden ist, bildet die Basis zur Beschreibung von Code-Transformationen. Ein extrahierter AST einer Java-Datei stellt eine Instanz dieses Metamodells dar und repräsentiert den Java-Sourcecode als Graph. Anhand dieses ASTs können nun durchzuführende Refactorings beschrieben und durchgeführt werden.

3.9 Angepasste Zielsetzung

Mit den jetzt bisher gesammelten Erkenntnissen lässt sich die in Kapitel 1 beschriebene Zielsetzung folgendermaßen konkretisieren (siehe Abb. 3-10):

Zu einem gegebenen Sourcecode wird ein abstrakter Syntaxbaum generiert, welcher eine Instanz des Metamodells ist. Für verschiedene Refactorings sind entsprechende Informationen

aus dem Syntaxbaum zu extrahieren. Sourcecode-Änderungen finden sodann auf Basis des ASTs statt und werden beschrieben durch eine Transformationsspezifikation. Das Ergebnis ist ein transformierter AST, welcher selbstverständlich ebenfalls eine Instanz des vorliegenden Metamodells ist. Aus diesem AST kann nun der refaktorisierte Sourcecode erzeugt werden.

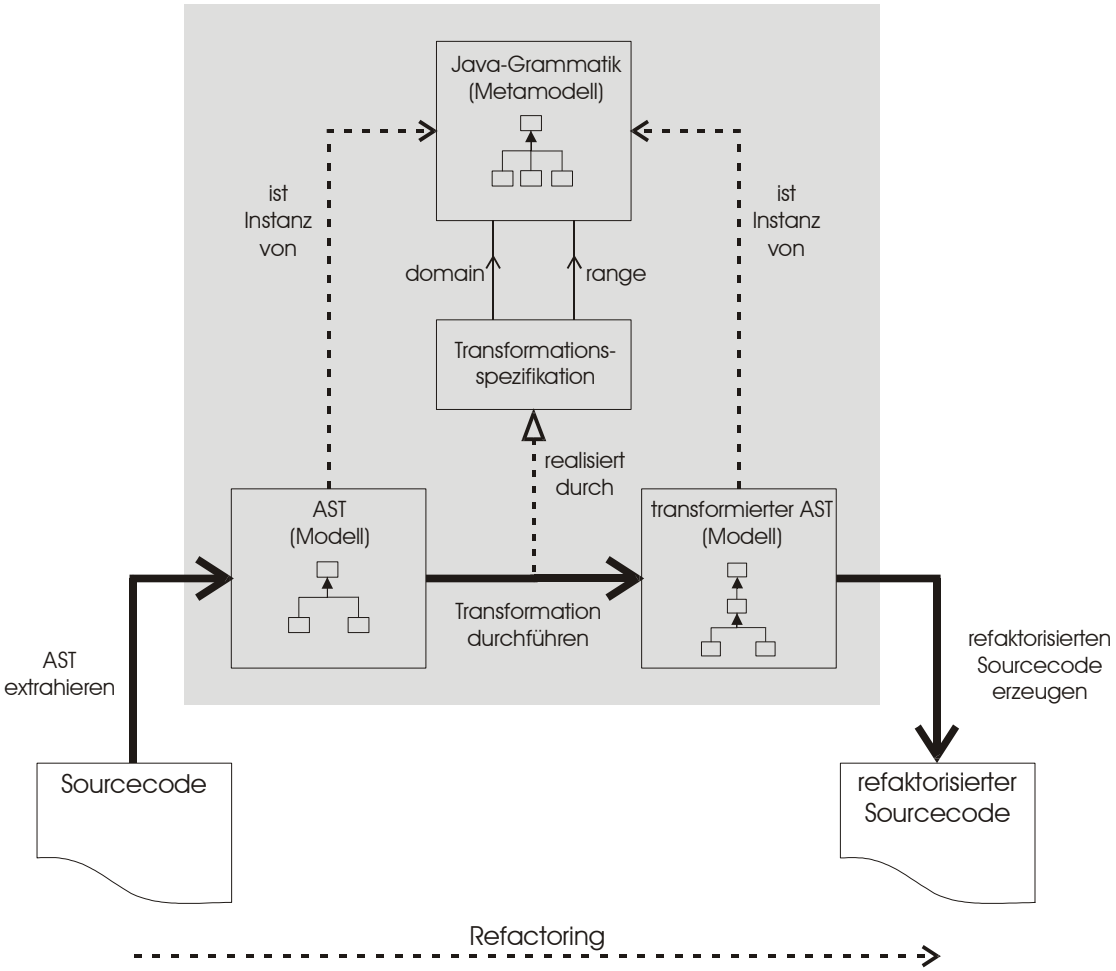


Abb. 3-10: Angepasste Zielsetzung

Kapitel 4
Refactoring auf Metamodell-Ebene

Das vorliegende Kapitel bietet eine Einführung in die Thematik des Refactoring und erläutert die Anwendung eines ausgewählten Refactorings an einem konkreten Beispiel auf Sourcecode-Ebene. Anschließend erfolgt eine Beschreibung dieser Sourcecode-Änderung auf Basis des entsprechenden Syntaxbaums sowie eine Erläuterung der damit verbundenen Modell-Transformationen.

4.1 Refactoring

Refactoring ist eine Technik der Softwareverbesserung und beschreibt ein methodisches Vorgehen zur Bearbeitung eines existierenden Programmcodes. Inhalt des Refactorings ist aber nicht die Fehlerkorrektur oder Erweiterung einer Applikation sondern vielmehr die Verbesserung der Struktur eines Codes, ohne dabei das funktionale Verhalten eines Programms zu verändern [Scho03]. Das Ziel von Refactoring ist es, einen strukturierten Quellcode zu erzeugen und Redundanzen zu entfernen, um dadurch die Verständlichkeit und Wartbarkeit zu fördern.

Refactoring hat seinen Ursprung im Smalltalk-Umfeld und wurde erstmals in der Dissertation von William Opdyke (siehe [Opdy92]) vorgestellt. Auf der Grundlage dieser Arbeit entwickelten John Brant und Don Roberts einen *Refactoring Browser* (siehe [BrRo96]) für Smalltalk-Programme. Durch die Veröffentlichung des Refactoring-Buches von Martin Fowler (siehe [Fowl99]) und der Integration der Refactoring-Technik in den Softwareentwicklungsprozess *Extreme Programming* hat diese Methode auch ihren Einzug in andere Programmiersprachen gefunden [ScVi04]. Die Popularität von Refactoring wird deutlich durch die in aktuellen Softwareentwicklungsumgebungen enthaltene Refactoring-Unterstützung – darunter auch das Refactoring Plug-In für Java der Eclipse-Plattform.

Ein erster Schritt vor der Anwendung von Refactoring ist das Erkennen von Konstellationen im Programmtext, die einer Änderung bedürfen (sog. *Bad Smells*). Dazu zählen unter anderem identische Codefragmente, die an verschiedenen Stellen im Code auftreten, nicht aussagekräftige Namen für Methoden und Variablen, sehr lange Methoden, die es erschweren, die Programmlogik zu verstehen sowie sehr große Klassen, die zahlreiche Daten verwalten und zu viel Funktionalität bereitstellen [Fowl99]. Mittlerweile wurde eine Vielzahl verschiedener Refactorings benannt, welche ein Code-Fragment ausgehend von einem definierten Zustand mittels einer Folge von Transformationen in einen definierten Endzustand überführen [Scho03]. Viele dieser Refactorings sind in dem Standardwerk von Martin Fowler beschrieben und stehen online in einem Refactoring-Katalog[5] inklusive Beschreibung und Beispiel zur Verfügung.

Grundsätzlich nehmen Refactorings lediglich kleine Änderungen am Programmcode vor. Allerdings besteht jederzeit die Gefahr, dass sich Fehler einschleichen und die Funktionalität beeinträchtigt wird. Abhilfe können Komponententests schaffen, indem sie die funktionalen Eigenschaften nach Durchführung eines Refactorings prüfen und auf Korrektheitsfehler hinweisen [Fisc02]. Für den Aufbau solcher Tests bietet sich unter anderem das von Erich Gamma und Kent Beck entwickelte UnitTest-Framework[6] an, welches insbesondere durch den Einsatz in Eclipse unter dem Namen JUnit für Java-Programmcode bekannt wurde. Mit JUnit lassen sich Tests bequem mit einer grafischen Oberfläche nach jedem einzelnen Schritt eines Refactorings ausführen [ScVi04].

4.2 Extract Method-Refactoring

Eines des nützlichsten und am häufigsten angewandten Refactorings ist *Extract Method*. Es beschreibt das Herausziehen einzelner Codezeilen aus einer langen Methode, die mehrere Aufgaben erledigt, und unterstützt somit eine strukturiertere und verständlichere Darstellung einer Methode. Die extrahierten Programmzeilen werden in eine neue Methode eingefügt, welche einen prägnanten Methodennamen erhält und innerhalb der langen Methode an entsprechender Stelle aufgerufen wird [ScVi04].

[5] siehe http://www.refactoring.com/catalog
[6] siehe http://www.junit.org

4.2.1 Vorgehen

Als Indiz zur Anwendung von *Extract Method* nennt Fowler das Vorkommen von Kommentaren in langen Methoden beziehungsweise die Notwendigkeit, Kommentare einzufügen, um die Funktionsweise zu erläutern. Neben dieser Information beschreibt Fowler das schrittweise Vorgehen zur Durchführung einer Sourcecode-Änderung folgendermaßen:

1. Create a new method, and name it after the intention of the method.

2. Copy the extracted code from the source method into the new target method.

3. Scan the extracted code for references to any variables that are local in scope to the source method. These are local variables and parameters to the method.

4. See whether any temporary variables are used only within this extracted code. If so, declare them in the target method as temporary variables.

5. Look to see whether any of these local-scope variables are modified by the extracted code. If one variable is modified, see whether you can treat the extracted code as a query and assign the result to the variable concerned. […]

6. Pass into the target method as parameters local-scope variables that are read from the extracted code.

7. Replace the extracted code in the source method with a call to the target method.

Abb. 4-1: Vorgehen des Extract Method-Refactorings [Fowl99]

4.2.2 Beispiel

An folgender Methode (siehe Abb. 4-2) lässt sich das *Extract Method*-Refactoring veranschaulichen. Das dargestellte Codefragment ist Bestandteil eines Beispielprogramms aus [Fowl99], welches Rechnungen für Kunden einer Videothek erstellt und ausdruckt. Aus der Methode `printOwing()`, die in einer Klasse namens `Customer` angesiedelt ist, soll derjenige Code extrahiert werden, welcher für die Ausgabe des Banners zuständig ist. Da keinerlei

Variablen betroffen sind, handelt es sich um ein triviales Vorgehen: Die betreffenden Codezeilen werden in eine neue Methode `printBanner()` verschoben sowie ein entsprechender Methodenaufruf in `printOwing()` eingefügt (siehe Abb. 4-3).

```
void printOwing() {
  Enumeration e = _orders.elements();
  double outstanding = 0.0;

  //print banner
  System.out.println("*****************");
  System.out.println("* Customer Owes *");
  System.out.println("*****************");

  //calculate outstanding
  while (e.hasMoreElements()) {
    Order each = (Order) e.nextElement();
    outstanding += each.getAmount();
  }

  //print details
  System.out.println("name: " + _name);
  System.out.println("amount: " + outstanding);
}
```

Abb. 4-2: Beispielcode

```
void printOwing() {
  Enumeration e = _orders.elements();
  double outstanding = 0.0;

  printBanner();

  //calculate outstanding
  while (e.hasMoreElements()) {
    Order each = (Order) e.nextElement();
    outstanding += each.getAmount();
  }

  //print details
  System.out.println("name: " + _name);
  System.out.println("amount: " + outstanding);
}

void printBanner() {
  //print banner
  System.out.println("*****************");
  System.out.println("* Customer Owes *");
  System.out.println("*****************");
}
```

Abb. 4-3: Refaktorisierter Beispielcode

4.3 Refactoring-Unterstützung in Eclipse

Zur Unterstützung der Implementierung von Java-Programmcode bietet die Eclipse-Umgebung ebenfalls eine Refactoring-Komponente an. Diese kann über den Menüpunkt *Refactor* oder über ein Popup-Menü für ein markiertes Programmelement aufgerufen werden.

Neben einer Vielzahl von zur Verfügung stehenden Refactorings wird *Extract Method* ebenfalls zur Verfügung gestellt. Durch Markieren der zu extrahierenden Codezeilen und Aufruf des *Extract Method*-Refactorings wird ein Benutzerdialog geöffnet. Nach Eingabe eines Bezeichners für die zu erstellende Methode zeigt eine Vorschau das zu erwartende Ergebnis in einem Dialogfenster an (siehe Abb. 4-4).

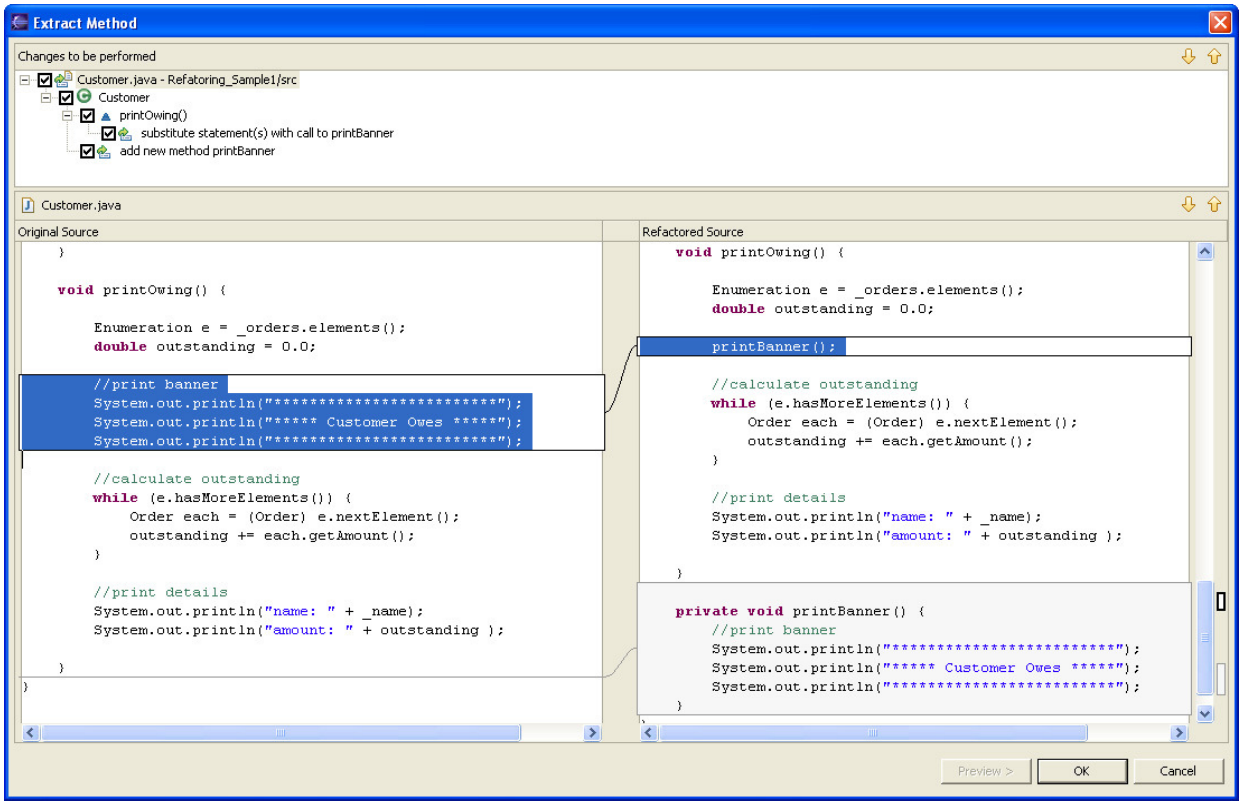


Abb. 4-4: Extract Method-Refactoring in Eclipse

4.4 Erzeugen eines ASTs

Vor der eigentlichen Erzeugung eines ASTs wird vorab ein für die Traversierung des Syntaxbaums und die Ausgabe der einzelnen Knoten benötigter Visitor von `ASTVisitor` abgeleitet (siehe Abb. 4-5). Die Ausgabe der Knoten dient der Veranschaulichung des Aufbaus des ASTs und wird für folgende Beispiele hilfreich sein.

```
public class ASTPrinter extends ASTVisitor {

  public void preVisit(ASTNode node) {
    ASTNode tempNode = node.getParent();
    while (tempNode != null) {
      System.out.print('\t');
      tempNode = tempNode.getParent();
    }
    System.out.println(node.getClass());
  }

}
```

Abb. 4-5: Visitor zur Erzeugung der AST-Ausgabe

Die Methode `preVisit()` wird vor Betreten eines jeden Knotens des ASTs aufgerufen und erzeugt eine Einrückung des entsprechenden Knotens in der Standardausgabe gemäß seiner Position in der AST-Hierarchie. Ist die Einrückung erfolgt, wird der Knotentyp über `node.getClass()` ausgegeben.

Das Erzeugen des ASTs aus einer vorliegenden Compilation Unit sowie das Zuweisen des zuvor erstellten Visitors zeigt Abb 4-6. Weitere Informationen, insbesondere über die zur Verfügung stehenden Methoden, bietet die Dokumentation[7] der Klasse `ASTParser`.

```
ICompilationUnit icu  = …
ASTParser parser = ASTParser.newParser(AST.JLS2);
parser.setSource(icu);
parser.setResolveBinding(true);
CompilationUnit cu = (CompilationUnit) parser.createAST(null);
ASTPrinter astprinter = new ASTPrinter();
cu.accept(astprinter);
```

Abb. 4-6: Erzeugen des Parsers und Zuweisen des Visitors

Die Methode `newParser()` generiert eine neue Instanz der Klasse `ASTParser`, welcher durch `setSource()` ein zu parsender Java-Sourcecode übergeben wird. Der Aufruf von `createAST()` erzeugt den gewünschten abstrakten Syntaxbaum und `accept()` erlaubt das Traversieren des ASTs anhand des zuvor erzeugten Visitors.

[7] Zu finden unter `JDT-Plug-In Developer Guide` → `Reference` → `API-Reference` → `org.eclipse.jdt.core.dom` → `Class Summary` → `ASTParser` des Eclipse-Hilfesystems.

Der Zugriff auf die Compilation Unit, welche in Abb. 4-6 über die Methode `setSource()` dem Parser übergeben wird, kann etwa durch Markieren der entsprechenden Java-Datei (welche die zu parsende Klasse oder Methode enthält) im Package Explorer der Eclipse-Oberfläche geschehen. Über diese Selektion kann auf das benötigte Java-Element (in diesem Fall ein Element vom Typ `ICompilationUnit`) zugegriffen werden.

Die Methode `setResolveBinding()` ermöglicht das Einbeziehen von Bindungsinformationen während der Erzeugung des Syntaxbaums. Über Bindungsinformationen ist es beispielsweise möglich, denjenigen Knoten zu finden, der einen bestimmten Bezeichner deklariert [Shav04]. Wird `setResolveBinding()` auf `true` gesetzt, geben diejenigen AST-Knoten, die die Methode `resolveBinding()` implementieren, einen Subtyp von `IBinding` zurück. Die zur Verfügung stehenden Bindungen sind vom Typ `IMethodBinding`, `IPackageBinding`, `ITypeBinding` sowie `IVariableBinding`. Über entsprechende Methoden kann sodann auf Bindungsinformationen von konkreten AST-Knoten zugegriffen werden.

Den zur Methode `printOwing()` aus dem Beispielcode (siehe Abb. 4-2) extrahierten AST zeigt nachfolgende Abb. 4-7. Hier sind diejenigen AST-Knoten in Fettdruck dargestellt, welche mit den in der rechten Spalte aufgeführten Sourcecode-Zeilen korrespondieren. Auf die Darstellung von Bindungsinformationen wurde in diesem Fall verzichtet, da diese das Erscheinungsbild des Syntaxbaums unnötig verkomplizieren. Im unteren Teil des nachfolgend dargestellten ASTs sind die durch die AST-Knoten repräsentierten Sourcecode-Elemente detaillierter aufgeführt.

```
CompilationUnit
|--TypeDeclaration
   |--MethodDeclaration                          void printOwing()
      |--PrimitiveType
      |--SimpleName
      |--Block
         |--VariableDeclarationStatement         Enumeration e = _orders.elements()
         |  |--SimpleType
         |  |  |--SimpleName
         |  |--VariableDeclarationFragment
         |     |--SimpleName
         |     |--MethodInvocation
         |        |--SimpleName
         |        |--SimpleName
         |--VariableDeclarationStatement         double outstanding = 0.0
         |  |--PrimitiveType
         |  |--VariableDeclarationFragment
         |     |--SimpleName
         |     |--NumberLiteral
         |--ExpressionStatement                  System.out.println("*****************")
         |  |--MethodInvocation
         |     |--QualifiedName
         |     |  |--SimpleName
         |     |  |--SimpleName
         |     |--SimpleName
         |     |--StringLiteral
         |--ExpressionStatement                  System.out.println("* Customer Owes *")
         |  |--MethodInvocation
         |     |--QualifiedName
         |     |  |--SimpleName
         |     |  |--SimpleName
         |     |--SimpleName
         |     |--StringLiteral
         |--ExpressionStatement                  System.out.println("*****************")
         |  |--MethodInvocation
         |     |--QualifiedName
         |     |  |--SimpleName
         |     |  |--SimpleName
         |     |--SimpleName
         |     |--StringLiteral
         |--WhileStatement                       while (e.hasMoreElements())
         |  |--MethodInvocation
         |  |  |--SimpleName
         |  |  |--SimpleName
         |  |--Block
         |     |--VariableDeclarationStatement   Order each = (Order) e.nextElement()
         |     |  |--SimpleType
         |     |  |  |--SimpleName
         |     |  |--VariableDeclarationFragment
         |     |     |--SimpleName
         |     |     |--CastExpression
         |     |        |--SimpleType
         |     |        |  |--SimpleName
         |     |        |--MethodInvocation
         |     |           |--SimpleName
         |     |           |--SimpleName
         |     |--ExpressionStatement            outstanding += each.getAmount()
         |        |--Assignment
         |           |--SimpleName
         |           |--MethodInvocation
         |              |--SimpleName
         |              |--SimpleName
         |--ExpressionStatement                  System.out.println("name: " + _name)
         |  |--MethodInvocation                     System.out.println()
         |     |--QualifiedName                        System.out
         |     |  |--SimpleName                           system
         |     |  |--SimpleName                           out
         |     |--SimpleName                           println
         |     |--InfixExpression                      "name: " + _name
         |        |--StringLiteral                        "name: "
         |        |--SimpleName                           _name
         |--ExpressionStatement                  System.out.println("amount: " + outstanding)
            |--MethodInvocation                     System.out.println()
               |--QualifiedName                        System.out
               |  |--SimpleName                           system
               |  |--SimpleName                           out
               |--SimpleName                           println
               |--InfixExpression                      "amount: " + outstanding
                  |--StringLiteral                        "amount: "
                  |--SimpleName                           outstanding
```

Abb. 4-7: AST-Knoten und korrespondierender Beispielcode

4.5 Extract Method-Refactoring auf AST-Basis

Nachdem nun das Vorgehen zur Erzeugung eines ASTs zu einer gegebenen Compilation Unit bekannt ist, lässt sich dieser als Basis zur Durchführung von Refactorings heranziehen. Analog des *Extract Method*-Beispiels aus Kap. 4.2.2 wird nachfolgend die dort beschriebene Code-Manipulation an dem korrespondierenden Syntaxbaum erläutert.

4.5.1 Extract Method-Refactoring ohne lokale Variablen

Eine graphisch vereinfachte Darstellung des zuvor dargestellten ASTs bietet Abb. 4-8. Hier sind lediglich die für das durchzuführende Refactoring wesentlichen Knoten abgebildet. Die farblich gekennzeichneten Elemente entsprechen den Expression Statements, welche durch das *Extract Method*-Refactoring in eine neue Methode ausgelagert werden.

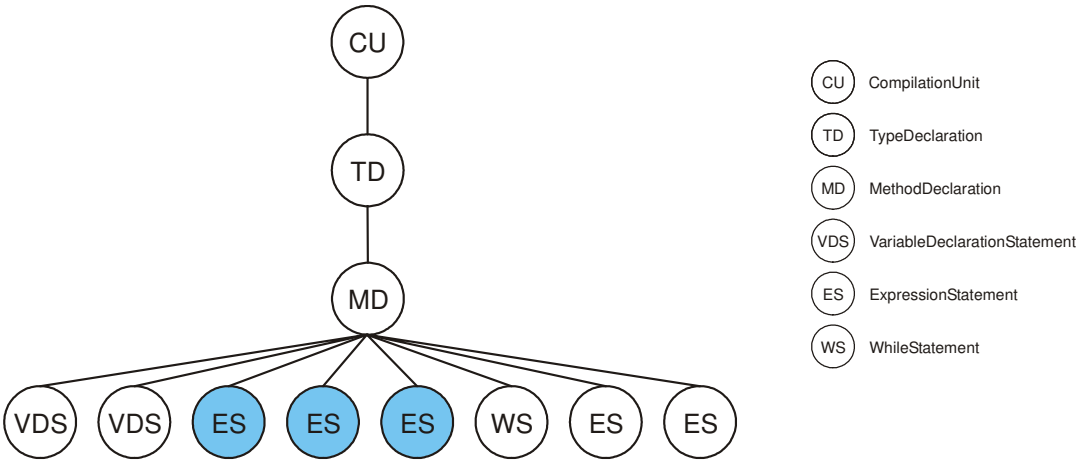
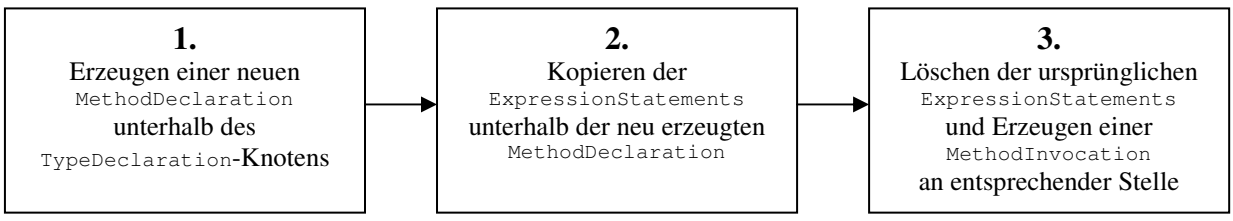


Abb. 4-8: Vereinfachte AST-Darstellung des Beispielcodes

Das *Extract Method*-Refactoring verläuft auf Basis des ASTs folgendermaßen:



Die ersten beiden Schritte korrespondieren mit den entsprechenden Aktionen in Fowlers Vorgehen (siehe Abb. 4-1). Da keinerlei Variablen von dem hier beschriebenen Refactoring betroffen sind, können Fowlers Vorgehenspunkte 3 bis 6 ausgelassen werden. Der dritte Schritt entspricht wiederum der letzten Aktion in Fowlers Vorgehen. Das Ergebnis der beschriebenen AST-Transformation zeigt Abb. 4-9.

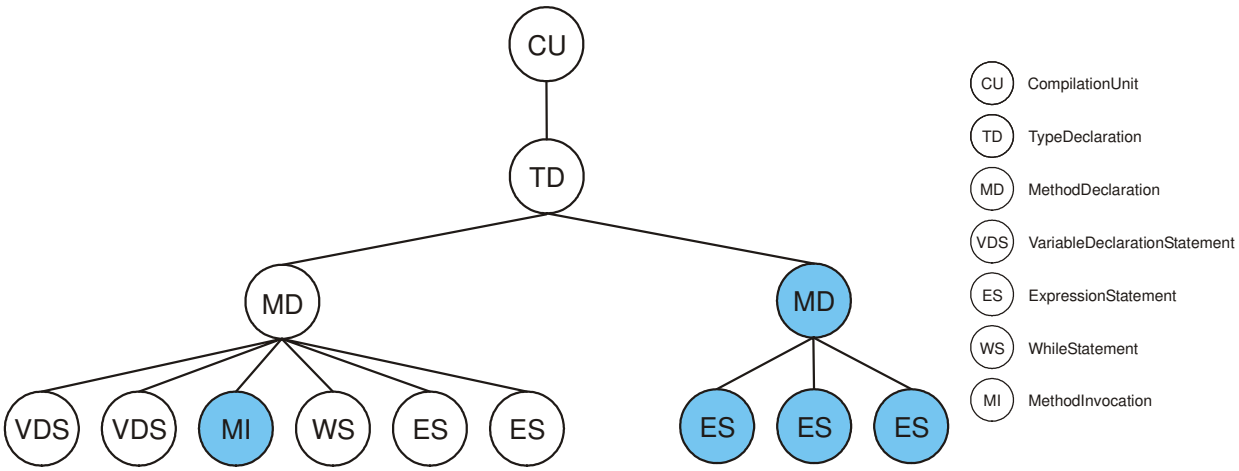


Abb. 4-9: AST nach Anwendung des Extract Method-Refactorings

Natürlich sind bei der Transformation einige Details zu beachten, die aufgrund der gewählten Abstraktionsebene des ASTs aus Abb. 4-8 nicht zu erkennen sind. Beispielsweise muss zusätzlich zur Erzeugung des neuen `MethodDeclaration`-Knotens ein `SimpleName`-Element generiert werden, welches den Methodenbezeichner darstellt. Weiterhin ist ein `Block`-Element zur Repräsemtation des Methodenrumpfs hinzuzufügen. Zudem beinhaltet das Verschieben der `ExpressionStatements` ein Kopieren des gesamten Unterbaums.

Die Anwendung dieses "trivialen" *Extract Method*-Beispiels hat gezeigt, dass einfache Codetransformationen durchaus an einem abstrakten Syntaxbaum beschrieben werden können. Interessanter wird es jedoch, sobald Variablen von einem durchzuführenden *Extract Method*-Refactoring betroffen sind. In diesem Fall kann auf die Angabe beziehungsweise Betrachtung der vorhandenen Bindungsinformationen nicht mehr verzichtet werden. Das Vorgehen wird daher an einem weiteren *Extract Method*-Beispiel verdeutlicht.

4.5.2 Extract Method-Refactoring mit lokalen Variablen

Das folgende Beispiel führt ein weiteres *Extract Method*-Refactoring am bereits refaktorisierten Beispielcode (siehe Abb. 4-3) durch. Die Detailausgabe soll ebenfalls in eine neue Methode `printDetails()` ausgelagert werden. Auf die globale Variable `_name` ist keine Rücksicht zu nehmen, da auf sie auch von der neuen Methode zugegriffen werden kann. Die lokale Variable `outstanding` hingegen muss als Parameter an die neue Methode übergeben werden, damit sie auch dort verfügbar ist. Zur Veranschaulichung zeigt Abb. 4-10 den Ausgangscode sowie das Ergebnis nach erneuter Anwendung des *Extract Method*-Refactorings.

```
void printOwing() {
  Enumeration e = _orders.elements();
  double outstanding = 0.0;

  printBanner();

  //calculate outstanding
  while (e.hasMoreElements()) {
    Order each = (Order) e.nextElement();
    outstanding += each.getAmount();
  }

  //print details
  System.out.println("name: " + _name);
  System.out.println("amount: " + outstanding);
}

void printBanner() {
  //print banner
  System.out.println("*****************");
  System.out.println("* Customer Owes *");
  System.out.println("*****************");
}
```

```
void printOwing() {
  Enumeration e = _orders.elements();
  double outstanding = 0.0;

  printBanner();

  //calculate outstanding
  while (e.hasMoreElements()) {
    Order each = (Order) e.nextElement();
    outstanding += each.getAmount();
  }

  printDetails(outstanding);


}

void printBanner() {
  //print banner
  System.out.println("*****************");
  System.out.println("* Customer Owes *");
  System.out.println("*****************");
}

void printDetails(double outstanding) {
  //print details
  System.out.println("name: " + _name);
  System.out.println("amount: " + outstanding);
}
```

Abb. 4-10: Extract Method-Refactoring mit Variablen

Zur Durchführung des *Extract Method*-Refactorings ist es notwendig zu erkennen, ob und welche Variablen von der durchzuführenden Sourcecode-Änderung betroffen sind. Zu diesem Zweck müssen die im AST enthaltenen Bindungsinformationen analysiert werden. Da die beiden Variablen `_name` und `outstanding` durch einen AST-Knoten vom Typ `SimpleName` repräsentiert werden (siehe Abb. 4-7) kann die Methode `resolveBinding()` aufgerufen werden. Zur Veranschaulichung zeigt Abb. 4-11 die Vererbungshierarchie von `SimpleName` sowie `IBinding`.

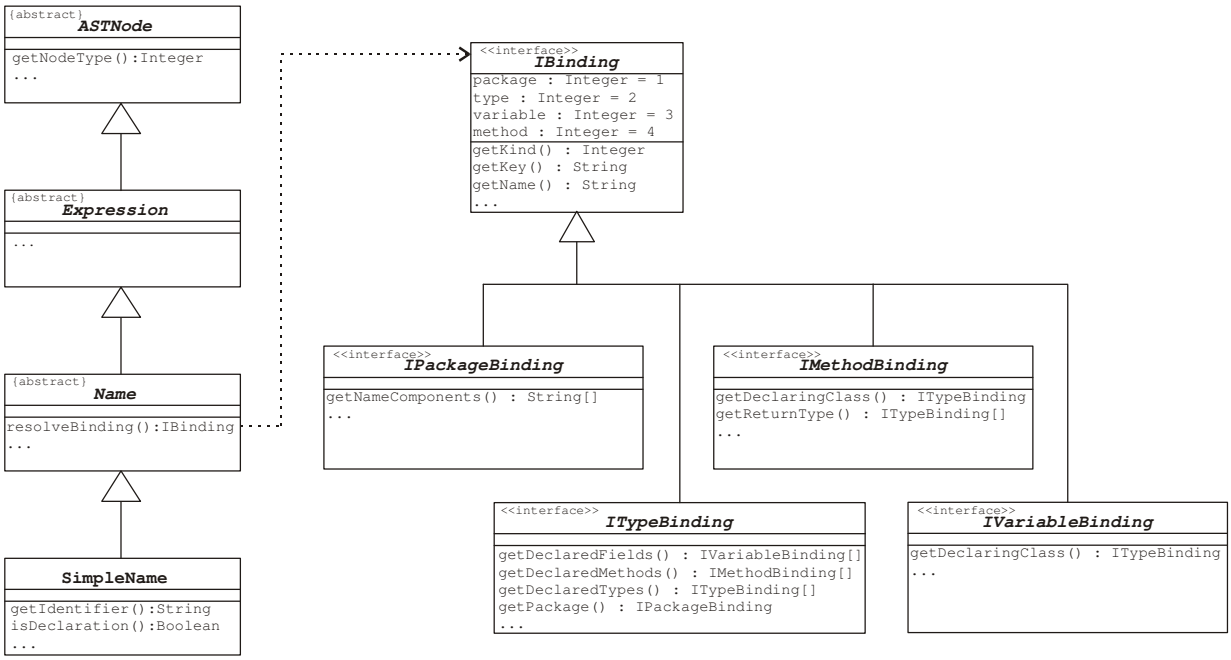


Abb. 4-11: Klasse SimpleName und Interface IBinding

Beispielsweise liefert der Aufruf von `resolveBinding()` für den `SimpleName`-Knoten, welcher die Variable `_name` repräsentiert, als Rückgabe ein `IBinding`. Diese Bindungsinformation lässt sich durch Aufruf von `getKind()` als eine Variablenbindung identifizieren. Über die Methoden `getName()`, `getKey()` sowie `getDeclaringClass()` lassen sich die weiteren benötigten Informationen extrahieren.

Abb. 4-12 zeigt denjenigen AST, welcher den in Abb. 4-10 dargestellten Ausgangs-Sourcecode repräsentiert und auf dessen Grundlage das Refactoring durchgeführt wird. Die zu verschiebenden `ExpressionStatements` inklusive ihrer Unterbäume sind farblich markiert. Zudem sind die Bindungsinformationen für `_name` und `outstanding` an den jeweiligen Knoten vermerkt.

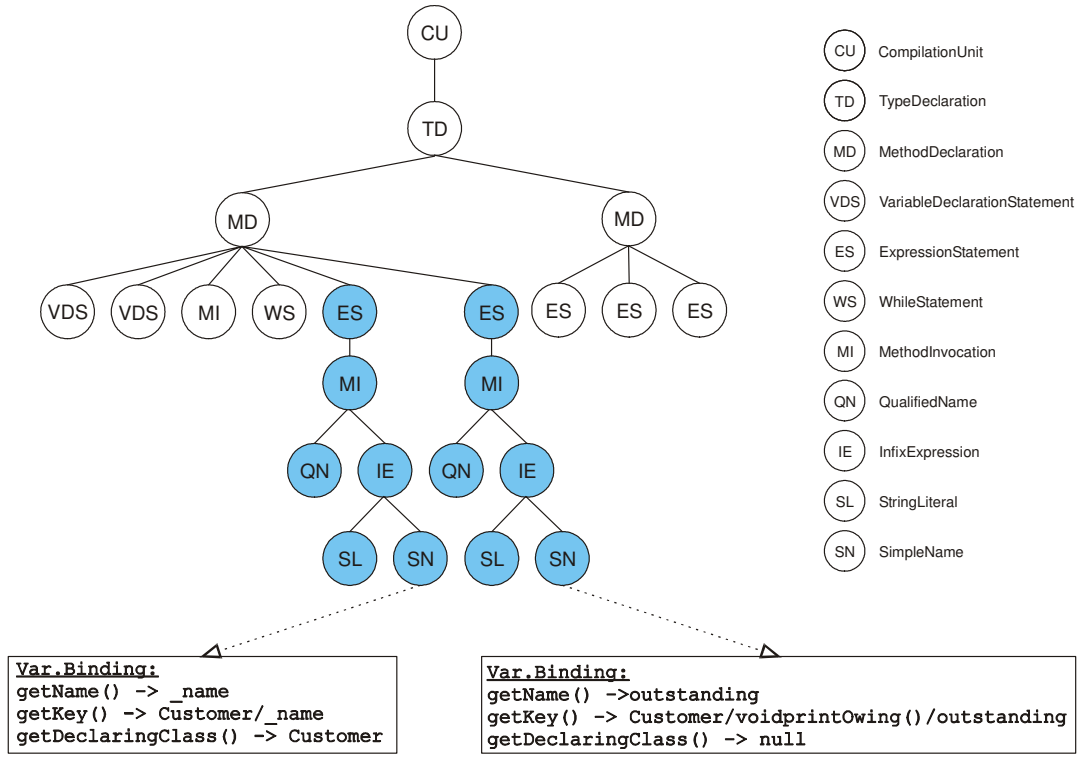
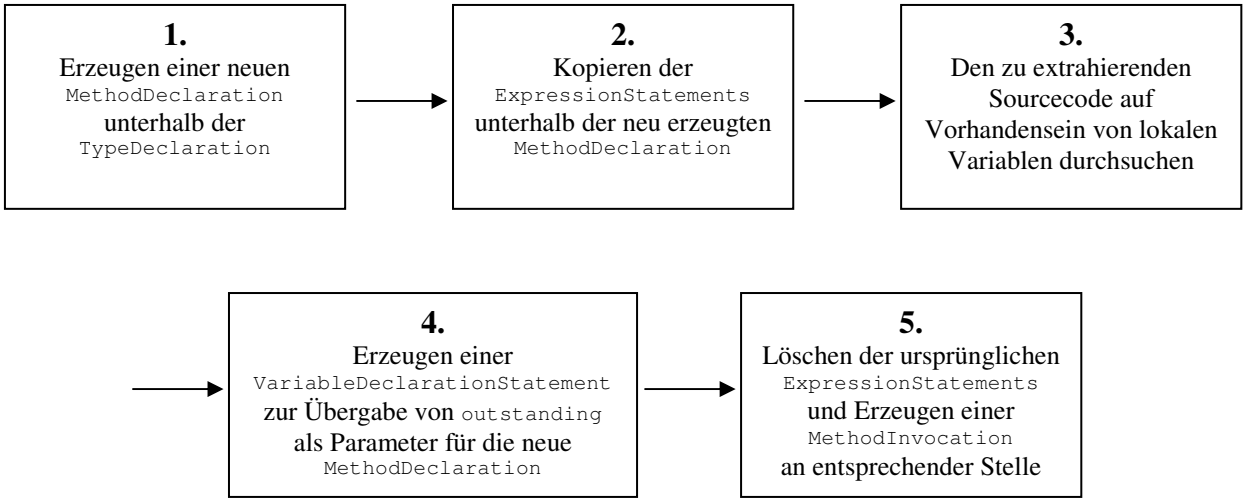


Abb. 4-12: AST und Bindungsinformationen

Einen Bezeichner für eine Bindung liefert `getKey()`. Hierüber lassen sich alle Bindungen, die ausgehend von einem gegebenen AST-Knoten erreichbar sind, eindeutig identifizieren. Für die Variable `outstanding`, welche aufgrund der Rückgabe eines Nullwerts nach Aufruf von `getDeclaringClass()` als lokal deklariert erkannt wird, kann anhand `getKey()` der Ursprung der Deklaration ermittelt werden. Dieser liegt in der Klasse `Customer` innerhalb der Methode `printOwing()`. Für `_name` hingegen liefert `getDeclaringClass()` den Namen der Klasse `Customer`, woraufhin auf eine globale Deklaration dieser Variablen geschlossen werden kann.

Aus dem vorliegenden Syntaxbaum und unter Betrachtung der enthalten Bindungsinformationen kann nun nach Fowlers Vorgehen erneut das Erstellen des transformieren ASTs erfolgen:



Augenscheinlich liefert der vorliegende AST inklusive der enthaltenen Bindungsinformationen genügend Details, um eine neue Methode zu erzeugen, die Detailausgabe in diese Methode zu verschieben und aufgrund des Vorhandenseins einer lokalen Variablen diese per Parameter an die neue Methode zu übergeben. Das Ergebnis der AST-Transformation zeigt Abb. 4-13.

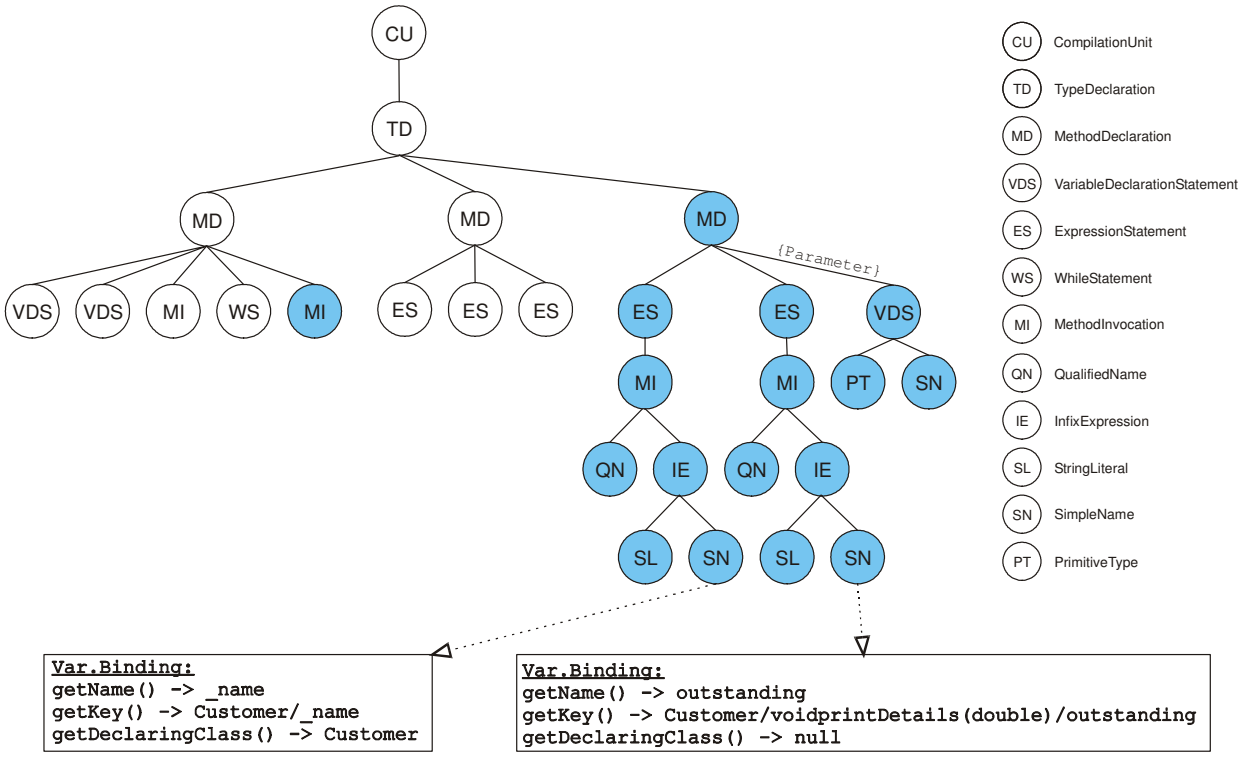


Abb. 4-13: Transformierter AST

4.5.3 Extract Method-Refactoring mit Variablen-Rückgabe

Abschließend sei bemerkt, dass Fowlers Beschreibung des *Extract Method*-Refactorings einige weitere Schritte beinhaltet:

1. Wird innerhalb der neuen Methode der Wert einer Variablen verändert und dieser für weitere Ausführungen im ursprünglichen Sourcecode benötigt, muss die Variable anhand eines *return*-Statements zurückgegeben werden. Dies wird ermöglicht durch Erzeugen von zusätzlichen entsprechenden AST-Knoten unterhalb der neuen Methodendeklaration.

2. Zudem muss sodann anstelle eines einfachen Methodenaufrufs eine Zuweisung an die lokale Variable der Form `localVar = newMethodCall` erfolgen. Dies ist ebenfalls mittels einer weiteren AST-Transformation durch Erzeugen von zusätzlichen AST-Knoten möglich.

Werden zudem mehrere Variablen in der ausgelagerten Methode verändert, müssten diese allesamt an den ursprünglichen Sourcecode zurückgegeben werden. Da die Java-Syntax jedoch lediglich das Vorhandensein eines einzelnen Rückgabe-Werts ermöglicht, muss vorab eine entsprechende Prüfung auf Basis des vorliegenden ASTs erfolgen und die Ausführung des *Extract Method*-Refactorings unterbunden werden.

Auf diese weiteren Schritte zur vollständigen Anwendung des *Extract Method*-Refactorings wird im weiteren Verlauf nicht näher eingegangen, da diese die Beschreibung und Spezifikation des Refactorings unnötig verkomplizieren und es offensichtlich ist, dass auch die zur Durchführung dieser weiteren Refactoring-Schritte benötigten Sourcecode-Informationen auf Grundlage eines vorliegenden ASTs ermittelt werden können.

Kapitel 5
Extract Method-Refactoring im Detail

Das vorliegende Kapitel beschreibt die Zerlegung des *Extract Method*-Refactorings in Einzelschritte und erläutert die notwendigen AST-Transformationen während der Durchführung des Refactorings.

5.1 Zerlegung

Ein Refactoring auf Modell-Ebene setzt sich aus einer Reihe von Einzelschritten in Form von zu erfüllenden Vorbedingungen und anschließend durchzuführenden Graphtransformationen zusammen. Daher ist es sinnvoll, das *Extract Method*-Refactoring in Teilschritte zu zerlegen und diejenigen Bedingungen zu identifizieren, welche vor Ausführung der einzelnen Codemanipulationen erfüllt sein müssen. Das Vorgehen des *Extract Method*-Refactorings wird nachfolgend schrittweise beschrieben, wobei die in Klammern stehenden Ziffern die identifizierten Bedingungen und Teiltransformationen kennzeichnen.

5.1.1 Beschreibung der Teilschritte

Diejenigen Codezeilen, welche in eine neue Methode verschoben werden sollen, müssen seitens des Benutzers markiert werden (1). Weiterhin ist ein Bezeichner für die zu erzeugende Methode anzugeben (2). Nun kann ermittelt werden, ob es sich um einen gültigen Methodenbezeichner im Kontext der vorliegenden Compilation Unit handelt. Zu diesem Zweck sind einerseits die bereits vorhanden Methoden zu extrahieren (3) und andererseits die Methoden der Superklassen, soweit vorhanden, zu untersuchen (4). Existiert der gewählte Bezeichner der neu zu erzeugenden Methode bereits oder werden dadurch ererbte Methoden überschrieben, ist das Refactoring abzubrechen (5). Ansonsten kann die neue Methode erzeugt werden (6).

Jetzt muss der zu verschiebende Sourcecode lediglich auf das Vorhandensein lokaler Variablen überprüft werden (7). Sind solche vorhanden, werden diese als Parameter an die neue Methode übergeben (8). Ist diese Graphtransformation abgeschlossen oder konnten keine lokalen

Variablen identifiziert werden, wird anhand einer weiteren Graphtransformation der Methodenaufruf eingefügt (9). Abschließend können die markierten Codezeilen in die neue Methode verschoben werden (10), ebenfalls realisiert durch eine letzte Teilgraphmanipulation.

5.1.2 Aktivitätsdiagramm

Die soeben beschriebenen Einzelschritte zeigt Abb. 5-1 nochmals in graphischer Form. Für jede Teilaktion des *Extract Method*-Refactorings, repräsentiert durch die entsprechende Nummerierung, existiert in dem nachfolgenden Aktivitätsdiagramm ein zugehöriger Knoten.

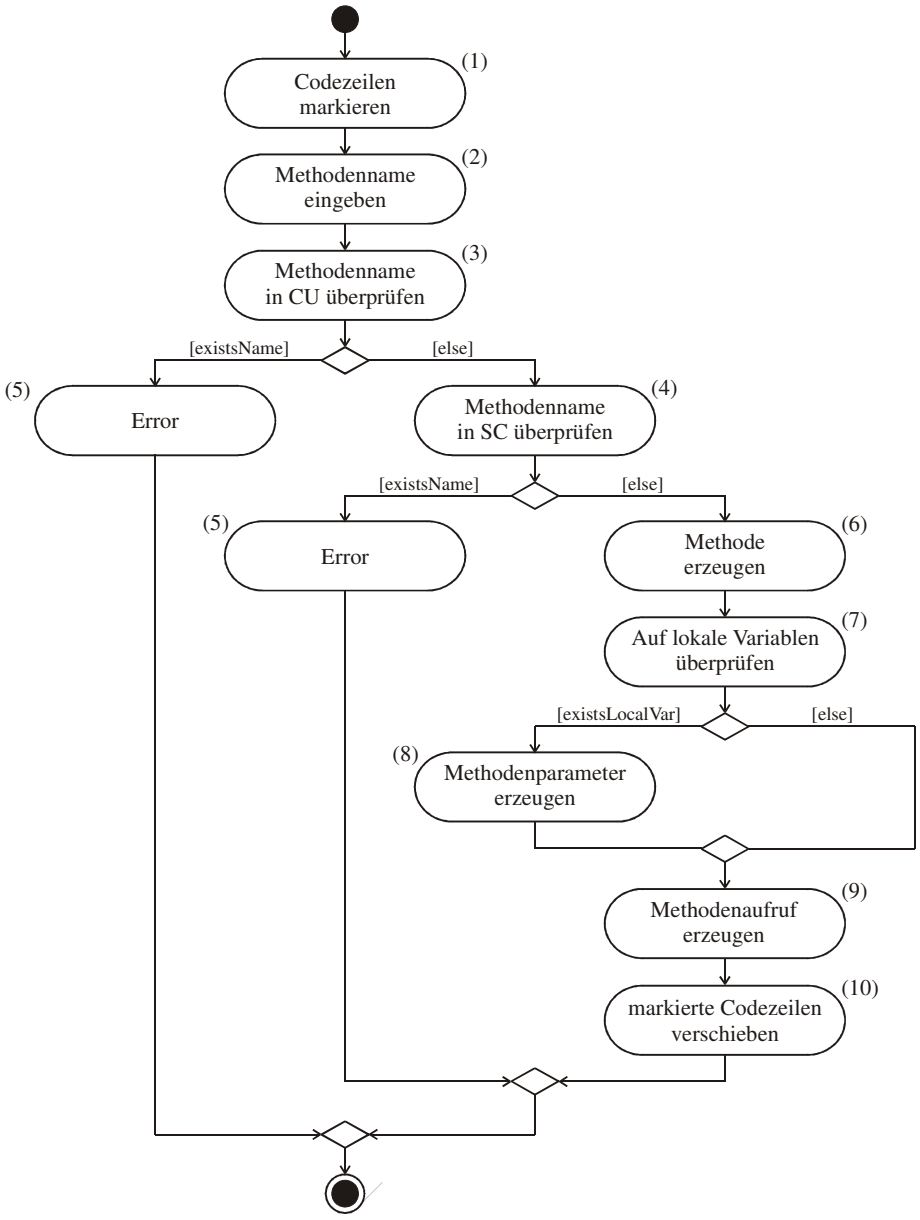


Abb. 5-1: Teilschritte des Extract Method-Refactorings

Dieses Ablaufdiagramm bildet die Basis für die weitere Beschreibung des *Extract Method*-Refactorings. Die Teilschritte "Codezeilen markieren" (1) sowie "Methodenname eingeben" (2) werden im weiteren Verlauf nicht detaillierter betrachtet. Hierbei handelt es sich um Benutzereingaben, die vor der Durchführung des *Extract Method*-Refactorings bereits zur Verfügung stehen. Ebenfalls wird die Fehlerbehandlung (5) nicht weiter ausgeführt, da im Folgenden lediglich die einzelnen Refactoring-Teilschritte erläutert werden.

5.1.3 Beispiel-AST

Zur Veranschaulichung der durchzuführenden Graphanfragen und -transformationen ist in Abb. 5-3 derjenige AST auszugsweise dargestellt, welcher das bereits vorgestellte *Extract Method*-Beispiel (siehe Abb. 5-2) repräsentiert. Er basiert auf dem AST-Metamodell (siehe Anhang B) und zeigt lediglich diejenigen Knoten und Kanten, welche für die Beschreibung des *Extract Method*-Refactorings von Bedeutung sind. Zur Orientierung kennzeichnet die Nummerierung der Statement-Knoten innerhalb des ASTs die entsprechende Codezeile im zugrundeliegenden Beispielcode.

```
void printOwing() {
 [1] Enumeration e = _orders.elements();
 [2] double outstanding = 0.0;

 [3] System.out.println("*****************");
 [4] System.out.println("* Customer Owes *");
 [5] System.out.println("*****************");

 [6] while (e.hasMoreElements()) {
       Order each = (Order) e.nextElement();
       outstanding += each.getAmount();
     }

 [7] System.out.println("name: " + _name);
 [8] System.out.println("amount: " + outstanding);
}
```

Abb. 5-2: Beispielcode zum Extract Method-Refactoring

Die grundlegenden Teilschritte (3) bis (10) des *Extract Method*-Refactorings, welche die zu überprüfenden Vorbedingungen und Graphtransformationen beinhalten, werden nachfolgend schrittweise auf Basis des ASTs aus Abb. 5-3 erläutert. Diese müssen später zur vollständigen Beschreibung des Refactorings in eine formale Spezifikation überführt werden.

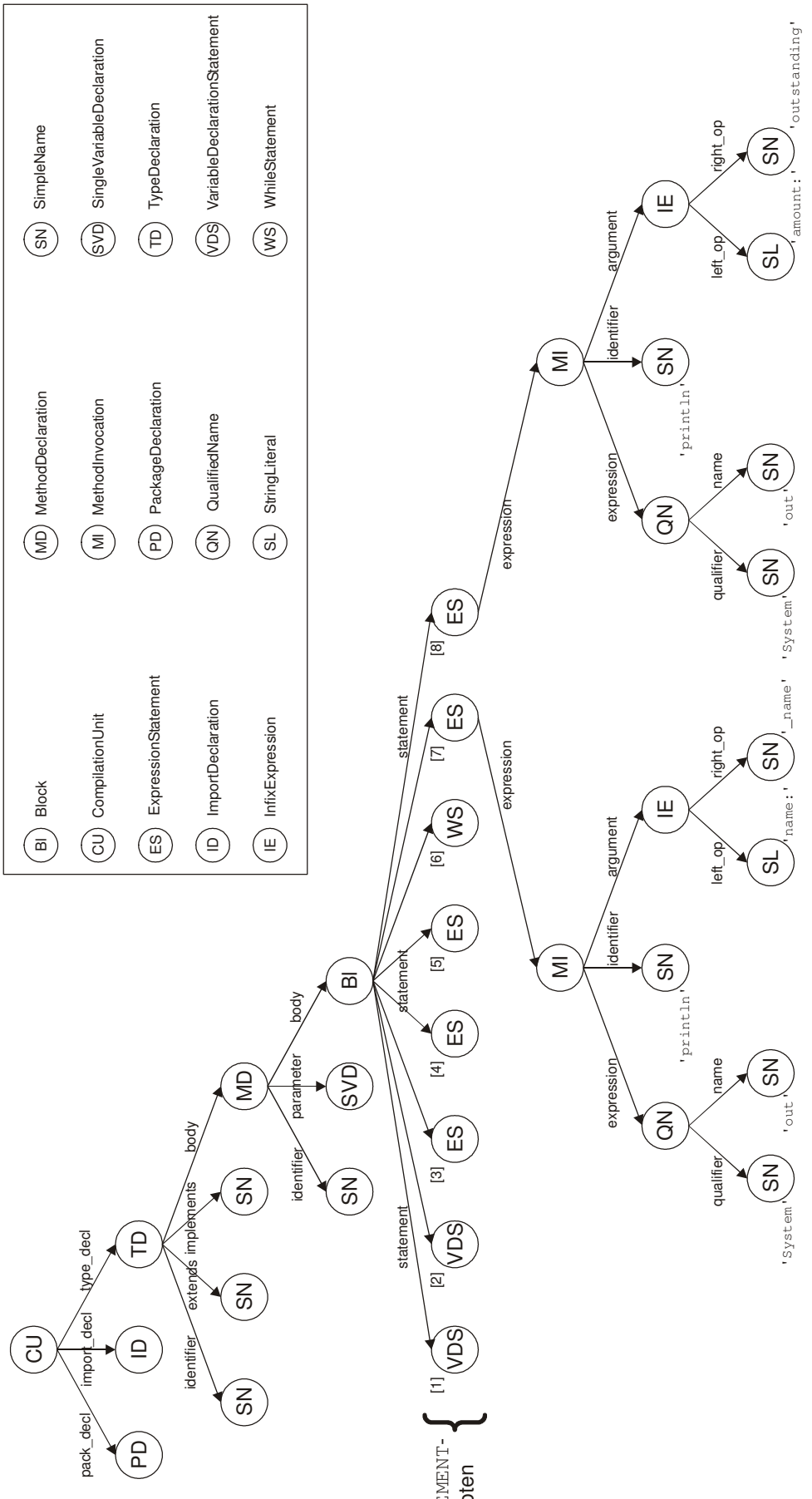


Abb. 5-3: AST gemäß Extract Method-Beispiel

5.2 Gültigkeit des Methodenbezeichners in Compilation Unit

Zur Durchführung des *Extract Method*-Refactorings ist es nötig, die Gültigkeit des vom Benutzer gewählten Bezeichners für die neu zu erstellende Methode zu überprüfen. Der gewählte Methodenbezeichner ist korrekt, sofern er der von Java geforderten Syntax entspricht (was im Folgenden als gegeben vorausgesetzt wird) sowie im Kontext der vorliegenden Compilation Unit noch nicht vorhanden ist.

Daher wird in diesem Refactoring-Teilschritt (3) zunächst kontrolliert, ob der gewählte Methodenbezeichner innerhalb der aktuellen Compilation Unit bereits vergeben ist. Zu diesem Zweck muss überprüft werden, ob im vorliegenden AST ein `MethodDeclaration`-Knoten existiert, der über eine `identifier`-Kante mit einem `SimpleName`-Knoten verbunden ist, welcher den Namen einer Methode repräsentiert und mit dem gewählten Methodenbezeichner übereinstimmt. Ist eine solche Konstellation vorhanden, muss eine entsprechende Fehlerbehandlung (5) durchgeführt und der Benutzer benachrichtigt werden. Anderenfalls kann der nächste Teilschritt eingeleitet werden.

5.3 Gültigkeit des Methodenbezeichners in Superklasse

Gemäß dem Teilschritt (4) ist zur Durchführung des *Extract Method*-Refactorings zusätzlich zu überprüfen, ob das Erzeugen der neuen Methode eine bereits durch Vererbung vorhandene Methode mit gleichem Bezeichner überschreibt. Daher ist es nötig, die Methoden der Superklassen in der Vererbungshierarchie zu identifizieren und mit dem gewählten Bezeichner zu vergleichen.

In einem vorliegenden AST ist zu diesem Zweck ausgehend von einem `TypDeclaration`-Knoten derjenige `SimpleName`-Knoten zu identifizieren, welcher über eine `extends`-Kante zu erreichen ist und somit den Namen der Superklasse repräsentiert. Hier können anhand der enthaltenen Bindungsinformationen die benötigten Angaben extrahiert werden. Auf eine detaillierte Beschreibung wird an dieser Stelle jedoch verzichtet, da sich das Vorgehen anhand der später folgenden Spezifikation anschaulicher erläutern lässt.

Existiert der gewählte Methodenbezeichner bereits in einer der Superklassen, muss wiederum eine entsprechende Fehlerbehandlung (5) eingeleitet werden. Anderenfalls wird der nächste Teilschritt durchgeführt.

5.4 Methodendeklaration erzeugen

Ein weiterer Teilschritt des *Extract Method*-Refactorings erfordert das Erzeugen der neuen Methodendeklaration (6). Dies wird selbstverständlich nur dann durchgeführt, wenn die Gültigkeit des gewählten Methodenbezeichners gewährleistet ist, was anhand der zuvor beschriebenen Bedingungen (3) und (4) überprüft werden kann.

Zur Deklaration einer neuen Methode müssen dem AST einige zusätzliche Knoten hinzugefügt werden, darunter ein `MethodDeclaration`-Knoten, welcher als Kindelement des entsprechenden `TypeDeclaration`-Knotens eingefügt wird. Weiterhin wird eine neue `identifier`-Kante benötigt, welche einen `SimpleName`-Knoten anbindet, der den Methodenbezeichner repräsentiert. Der gewählte Methodenbezeichner wird abschließend dem entsprechenden Attribut des `SimpleName`-Knotens zugewiesen. Damit ist das Erzeugen der neuen Methodendeklaration abgeschlossen und eine erste AST-Transformation wurde durchgeführt.

5.5 Vorkommen lokaler Variablen

Das weitere Vorgehen zur Durchführung des *Extract Method*-Refactorings beinhaltet das Überprüfen auf ein Vorkommen lokaler Variablen innerhalb der zu refaktorisierenden Codezeilen. Liegt ein solcher Fall vor, müssen diese lokalen Variablen in einem folgenden Refactoring-Teilschritt an die neue Methode übergeben werden, indem die Methodendeklaration um entsprechende Parameter erweitert wird.

Die Überprüfung auf AST-Basis gemäß dem Refactoring-Teilschritt (7) gestaltet sich folgendermaßen: Ausgehend von einem `MethodDeclaration`-Knoten, der diejenige Methode mit den zu verschiebenden Codezeilen repräsentiert, muss eine `body`-Kante zu einem `Block`-

Knoten existieren, welcher wiederum über eine ausgehende `statement`-Kante mit einem `STATEMENT`-Knoten verbunden ist. Dies ist ein abstrakter Knoten, welcher einen Platzhalter für eine Vielzahl von möglichen Statement-Knoten wie `VariableDeclarationStatement`-, `WhileStatement`- oder `ExpressionStatement`-Knoten darstellt. Während der Durchführung des *Extract Method*-Refactorings können mehrere dieser `STATEMENT`-Knoten durch den Benutzer markiert sein, d.h. sie repräsentieren die zu refaktorisierenden Codezeilen innerhalb einer Methode.

Ausgehend von diesen `STATEMENT`-Knoten existiert eventuell ein Pfad, welcher über verschiedene Kanten letztendlich an einem `NAME`-Knoten endet. Dieser ebenfalls abstrakte Knoten repräsentiert wiederum einen `SimpleName`- oder `QualifiedName`-Knoten. Kann von dem gefundenen `NAME`-Knoten eine ausgehende `binding`-Kante zu einem `IVariableBinding`-Knoten identifiziert werden, handelt es sich bei dem gefundenen Element um eine Variable. Ist das `declaringClass`-Attribut des zugehörigen Bindungsknotens zudem mit `null` belegt, wurde damit das Vorkommen einer lokalen Variable entdeckt. Diese muss in einem weiteren Refactoring-Teilschritt über einen entsprechenden Methodenparameter der neuen Methode übergeben werden. Das Vorgehen zur Identifizierung lokaler Variablen wird während der später durchzuführenden Spezifikation eingehender behandelt.

5.6 Methodenparameter erzeugen

Konnten innerhalb der zu refaktorisierenden Codezeilen lokale Variablen ermittelt werden, müssen diese an die neue Methode als Argumente übergeben werden. Daher ist es nötig, gemäß dem Refactoring-Teilschritt (8) die zuvor erzeugte Methodendeklaration um entsprechende Parameter zu erweitern.

Zu diesem Zweck muss ausgehend von dem neuen `MethodDeclaration`-Knoten eine `parameter`-Kante erzeugt werden, welche einen `SingleVariableDeclaration`-Knoten anbindet. Dieser Knoten wird wiederum über eine ausgehende `identifier`-Kante mit einem `SimpleName`-Knoten verbunden, welcher den Bezeichner des Methodenparameters repräsentiert. Zudem wird ein `Type`-Knoten erzeugt, der über eine `type`-Kante ebenfalls mit dem `SingleVariableDeclaration`-Knoten verbunden ist und den Parametertyp darstellt. Die

benötigten Informationen zur Erzeugung dieser Knoten kann aus den zuvor identifizierten AST-Knoten, welche die lokalen Variablen repräsentieren, entnommen werden. Damit ist dieser Refactoring-Teilschritt ebenfalls abgeschlossen und der Methodenaufruf kann erzeugt werden.

5.7 Methodenaufruf erzeugen

In einem nächsten *Extract Method*-Teilschritt wird entsprechend dem Refactoring-Vorgehen (9) der Aufruf der zuvor neu erzeugten Methode eingefügt. Dies beinhaltet das Identifizieren der markierten Statements, d.h. derjenigen Codezeilen, welche später in die neue Methode ausgelagert werden. An dieser Position muss der Aufruf der zuvor neu erzeugten Methode erfolgen.

Daher muss im vorliegenden AST vor den entsprechenden `STATEMENT`-Knoten ein zusätzlicher `ExpressionStatement`-Knoten eingefügt werden, welcher durch eine `statement`-Kante mit dem zugehörigen `Block`-Knoten verbunden wird. Von diesem `ExpressionStatement`-Knoten wird eine ausgehende `expression`-Kante zu einem `MethodInvocation`-Knoten erzeugt. Dieser wird wiederum über eine `identifier`-Kante mit einem `SimpleName`-Knoten gepaart, welcher den Methodenaufruf der zuvor erzeugten neuen Methode darstellt. Über `argument`-Kanten werden entsprechend der Anzahl der Methodenparameter, d.h. der zuvor identifizierten lokalen Variablen, zusätzliche `SimpleName`-Knoten hinzugefügt. Damit ist dieser Teilschritt ebenfalls abgeschlossen und die markierten Codezeilen können in die neue Methode verschoben werden.

5.8 Markierte Codezeilen verschieben

Nachdem der Methodenaufruf an passender Stelle eingefügt wurde, kann gemäß dem letzten *Extract Method*-Teilschritt (10) das Verschieben der markierten Codezeilen in die zuvor erzeugte neue Methode erfolgen.

Zu diesem Zweck muss unterhalb des `MethodDeclaration`-Knotens, der die neu erzeugte Methode repräsentiert, ein `Block`-Knoten erzeugt werden, welcher über eine eingehende `body`-

Kante mit der entsprechenden Methodendeklaration verbunden wird und somit den Methodenrumpf darstellt. Unterhalb dieses Knotens können die markierten `STATEMENT`-Knoten inklusive ihrer eingehenden `statement`-Kanten und kompletten Unterbäume verschoben werden. Damit ist auch dieser letzte Teilschritt abgeschlossen und das *Extract Method*-Refactoring beendet.

5.9 Pseudocode des Extract Method-Refactorings

Die Zerlegung eines Refactorings in Teilschritte bietet den Vorteil, einzelne Bedingungen und Graphtransformationen identifizieren zu können. Am Beispiel des *Extract Method*-Refactorings konnten auf diese Weise die durchzuführenden Graphtransformationen und entsprechenden Vorbedingungen schrittweise erläutert werden. Zusammenfassend lässt sich das *Extract Method*-Refactoring analog dem in Abb. 5-1 vorgestellten Vorgehen durch folgenden Pseudocode (siehe Abb. 5-4) beschreiben, wobei der Eingabeparameter `marking` die markierten Codezeilen in Form einer Menge von `STATEMENT`-Knoten repräsentiert und `methodName` einer Zeichenkette gemäß dem gewählten Methodenbezeichner entspricht.

```
extractMethodRefactoring(marking : Set of STATEMENT, methodName : string) {
  if existiert Methodenbezeichner in Compilation Unit then error;
  else
    if existiert Methodenbezeichner in Superklasse then error;
    else
      erzeuge Methodendeklaration;
      if existieren lokale Variablen then erzeuge Methodenparameter;
      erzeuge Methodenaufruf;
      verschiebe markierte Codezeilen;
    end;
  end;
}
```

Abb. 5-4: Pseudocode des Extract Method-Refactorings

Die in diesem Kapitel vorgestellten Einzelschritte des *Extract Method*-Refactorings können nun in eine formale Spezifikation überführt werden. Zu diesem Zweck wird im Folgenden die Sprache PROGRES vorgestellt.

Kapitel 6
Transformations-Spezifikation mit PROGRES

Das vorliegende Kapitel erläutert die Sprache PROGRES sowie deren Konzepte zur Beschreibung von Graphtransformationen, welche sodann zur Spezifikation des zuvor beschriebenen *Extract Method*-Refactorings eingesetzt werden.

6.1 PROGRES

PROGRES (*PROgramming Graph REwriting Systems*) ist eine high-level Programmiersprache, welche auf gerichteten, attributierten Graphen basiert und unter anderem Konzepte zur Beschreibung von Graphtransformationen bietet [ScWi99]. PROGRES wurde an der RWTH Aachen entwickelt und hat sich im Laufe der Zeit zu einer Programmierumgebung entwickelt, welche für Solaris- und Linux-Plattformen zur Verfügung steht.

Basierend auf einer Mischung von graphischen und textuellen Elementen stellt PROGRES eine hybrid-visuelle Sprache dar. So beinhaltet beispielsweise eine Graphtransformations-spezifikation einen fortlaufenden Text, innerhalb dessen zur Konstruktion von linker und rechter Seite einer Transformationsregel eine graphische Notation verwendet wird, welche wiederum mit textuellen Angaben versehen sein kann [ScWi99].

6.1.1 Knotentypen

Zur Definition der statischen Struktur eines Graphschemas stehen verschiedene Komponenten zur Verfügung, darunter Knoten- und Kantentypen. *Knotentypen* dienen zur Typisierung eines Knotens und beschreiben die Eigenschaften ihrer Instanzen in Form von Attributen. Sie werden in PROGRES vor allem dazu eingesetzt, diejenigen Knotenklassen auszuzeichnen, für die es in der beschriebenen Domäne konkrete Knoteninstanzen geben kann [Zuen96].

Knotentypen werden graphisch als Rechteck mit abgerundeten Ecken notiert. Die textuelle Notation zur Definition eines Knotentyps hingegen hat die Form

```
node_type aNodeType : aClass end;
```

Dabei stellt *aNodeType* den Knotentyp-Bezeichner dar und *aClass* repräsentiert die zugehörige Knotenklasse.

6.1.2 Knotenklassen

Knotenklassen hingegen dienen der einmaligen Definition von gleichen Eigenschaften für verschiedene Knotentypen, welche sodann die Attribute von der entsprechenden Knotenklasse erben. Dieses Konzept entspricht dem bekannten Vererbungsprinzip objekt-orientierter Programmiersprachen. Knotenklassen, zu denen es keine Knotentypen gibt, sind sogenannte abstrakte Klassen und werden dementsprechend nicht instanziert. Sie stellen lediglich der Zusammenfassung gemeinsamer Eigenschaften in der Vererbungshierarchie dar [Zuen96].

Die graphische Repräsentation erfolgt durch Rechtecke und die Vererbungshierarchie wird durch gerichtete Kanten mit einer unausgefüllten Pfeilspitze von Subklasse in Richtung Superklasse gekennzeichnet. Die textuelle Notation erfolgt in der Form

```
node_class aClass is_a SuperClass1, SuperClass2,… end;
```

Dabei repräsentiert *aClass* den Knotenklassenbezeichner und nach dem Schlüsselwort `is_a` folgt eine Auflistung eventuell vorhandener *SuperClasses*.

6.1.3 Kantentypen

Kantentypen werden einerseits zur Kennzeichnung einer Kante herangezogen und erlauben andererseits das Identifizieren von gültigen Start- und Endknotentypen. Zudem ist die Definition von Kardinalitäten in `[min:max]`-Notation zulässig. Allerdings ist in PROGRES das Zuweisen von Attributen zu Kantentypen nicht vorgesehen.

Die graphische Notation eines Kantentyps entspricht einer durchgehenden Linie zuzüglich einer ausgefüllten Pfeilspitze zur Angabe der Richtung. Die textuelle Notation hat das Format

```
edge_type anEdgeType : aSourceClass [min:max] -> aTargetClass [min:max];
```

mit *anEdgeType* zur Benennung des Kantentyps, *aSourceClass* und *aTargetClass* zur Festlegung des Start- und Endknotentyps sowie *[min:max]* zur Angabe der Kardinalitäten.

6.1.4 Attribute

Zur Datenmodellierung stellt PROGRES *Attribute* zur Verfügung, welche in drei Arten unterteilt sind: *intrinsic*-, *meta*- sowie *derived*-Attribute. Attribute der Sorte *intrinsic* stellen lokale Variablen der einzelnen Knoteninstanzen eines Graphen dar und erlauben somit das Hinterlegen zusätzlicher Informationen an den einzelnen Knoten. *Meta*-Attribute unterscheiden sich von intrinsic-Attributen dadurch, dass ihnen im operationalen Teil einer Spezifikation keine Werte zugewiesen werden dürfen. Der Wert ergibt sich immer aus der für die zugehörige Knotenklasse gültigen Initialisierungsvorschrift. Wie bei meta-Attributen wird der Wert bei *derived*-Attributen ausschließlich durch die zugehörigen Berechnungsvorschriften definiert. Im Unterschied zu den beiden anderen Arten darf aber in Berechnungen für derived-Attribute beliebig auf die Werte anderer Attribute im Graphen zugegriffen werden [Zuen96].

Graphisch werden diese Attribute innerhalb des zugehörigen Knotentypelements dargestellt. Die textuelle Definition eines intrinsic-Attributs für eine Knotenklasse hat beispielsweise die Gestalt

```
node_class aClass
  intrinsic attr1 : string = 'x';
            attr2 : integer;
end;
```

wobei *attr1* und *attr2* die Attributbezeichner darstellen, welchen ein entsprechender Typ zugewiesen wird. Optional kann ein Attribut zudem mit einem Initialwert belegt werden.

6.1.5 Typkonzept

Alle Elemente der modellierten Graphstruktur, wie Knoten, Attribute und Kanten, gehören jeweils zu genau einem Typ, der die Eigenschaften dieser Elemente festlegt. Für Knoten sieht PROGRES ein zweistufiges Typkonzept vor: Jeder Knoten gehört zu genau einem Knotentyp, welcher wiederum genau einer Knotenklasse angehört, die die Eigenschaften des Knotentyps (und damit der Instanzen eines Knotentyps) festlegt [Zuen96].

6.2 AST-Graphschema in PROGRES-Notation

Das nachfolgend dargestellte Graphschema veranschaulicht nochmals die soeben erläuterten PROGRES-Sprachelemente in graphischer Notation (siehe Abb. 6-1) und beschreibt ausgehend von einer Compilation Unit einen AST auf höchster Hierarchieebene.

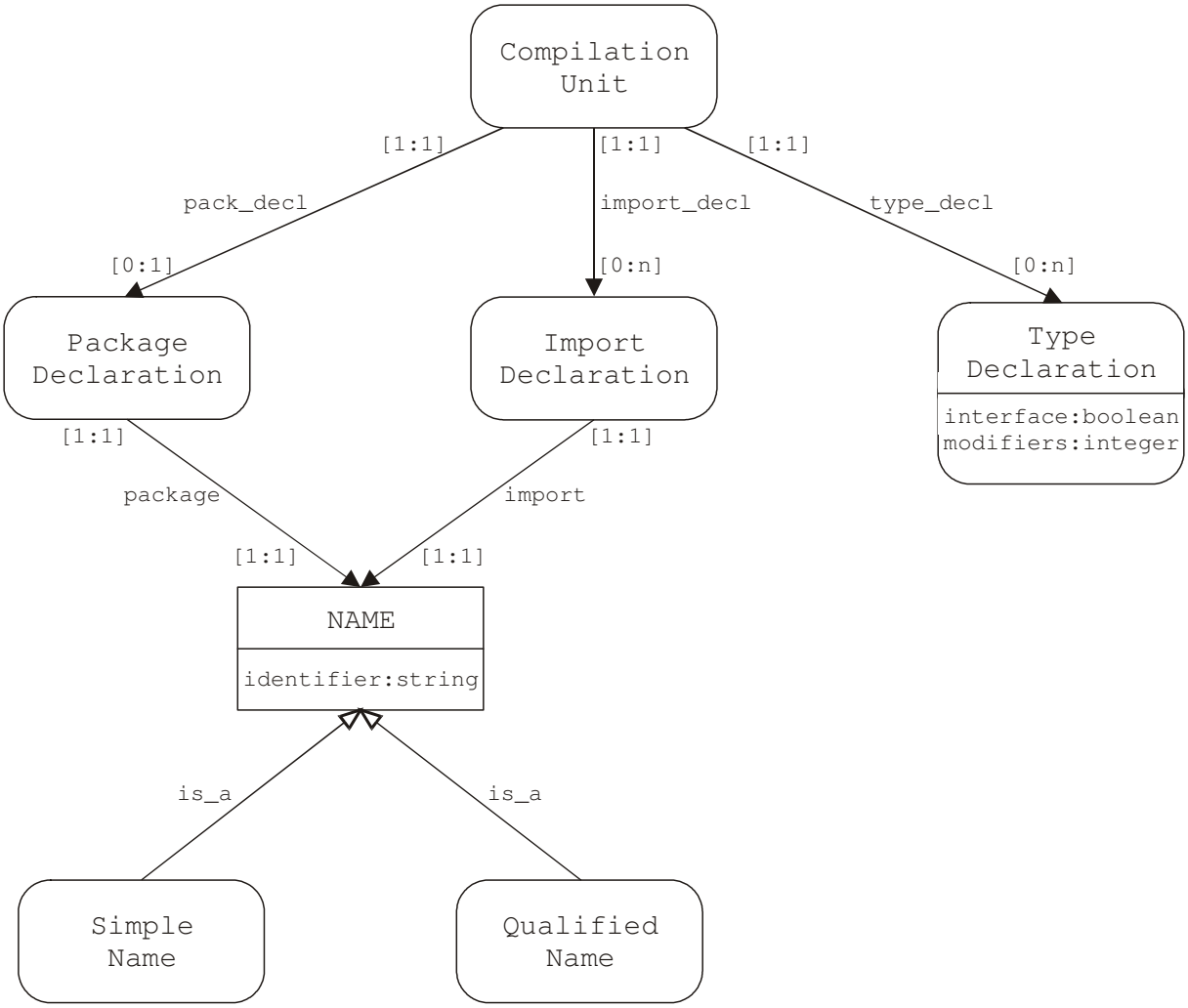


Abb. 6-1: Graphische Darstellung des AST-Graphschemas

Die textuelle Notation der Knoten und Kanten entsprechend dem in Abb. 6-1 dargestellten Graphschema ist nachfolgend (siehe Abb. 6-2) dargestellt.

Knoten:

```
node_type CompilationUnit : COMPILATIONUNIT end;
node_type PackageDeclaration : PACKAGEDECLARATION end;
node_type ImportDeclaration : IMPORTDECLARATION end;
node_type TypeDeclaration : TYPEDECLARATION
    intrinsic interface : boolean;
              modifiers : integer;
end;
node_class NAME
    intrinsic identifier : string;
end;
node_type SimpleName : NAME end;
node_type QualifiedName : NAME end;
```

Kanten:

```
edge_type pack_decl : CompilationUnit [1:1] -> PackageDeclaration [0:1];
edge_type import_decl : CompilationUnit [1:1] -> ImportDeclaration [0:n];
edge_type type_decl : CompilationUnit [1:1] -> TypeDeclaration [0:n];
edge_type package : PackageDeclaration [1:1] -> NAME [1:1];
edge_type import : ImportDeclaration [1:1] -> NAME [1:1];
```

Abb. 6-2: Textuelle Notation des AST-Graphschemas

Im AST-Graphschema enthält der Knotentyp *CompilationUnit* drei ausgehende Kanten (*pack_decl*, *import_decl* sowie *type_decl*), welche inklusive der angegebenen Kardinalitäten die Gestalt eines ASTs auf höchster Hierarchieebene beschreiben. Anhand der Kantentyp-Definition lässt sich spezifizieren, welcher Klasse die Start- und Zielknoten einer Kante angehören. Für den Kantentyp *type_decl* wird beispielsweise der Knotentyp *CompilationUnit* als Start- sowie *TypeDeclaration* als Zielknoten festgelegt. Die Angabe der Kardinalitäten lässt weitere Aussagen zu: Für einen Knotentyp *CompilationUnit* können mehrere ausgehende Kanten zu verschiedenen *TypeDeclaration*-Knotentypen existieren. Umgekehrt existiert zu jeder *TypeDeclaration* lediglich eine *CompilationUnit*, welche durch eine Rückwärtstraversierung der eingehenden Kante *type_decl* identifiziert werden kann.

Die angesprochene Vererbungshierarchie lässt sich anhand der Knotenklassendefinition von *Name* erkennen. Diese Klasse beinhaltet das String-Attribut *identifier*, welches an die Knotentypen *SimpleName* und *QualifiedName* vererbt wird. Über *Name.identifier* kann auf den

Wert des Attributs zugegriffen werden. In Attributberechnungsvorschriften kann der aktuelle Knoten über *self* angesprochen werden. So liefert beispielsweise für *Name* der Ausdruck *self.identifer* den hinterlegten Namen.

6.3 Pfadausdrücke in PROGRES

Pfadausdrücke können in PROGRES innerhalb einer Schemaspezifikation zur Formulierung von Attributberechnungsvorschriften sowie zur Beschreibung von Graphtraversierungen benutzt werden [Zuen96]. Die einfachste Form eines Pfadausdrucks ist die Kantentraversierungsoperation `-edgeType1->` beziehungsweise `<-edgeType2-` zur Navigation entlang eines Kantentyps. Die Konkatenation mehrerer Pfadprädikate ermöglicht der `&`-Operator. So gelangt man bspw. über den Pfadausdruck `-import_decl-> & -import->` ausgehend von einer `CompilationUnit` zu denjenigen `NAME`-Knoten, welche die Imports einer Java-Datei repräsentieren (siehe Abb. 6-1).

Die textuelle Definition eines Pfads unter Anwendung des `&`-Operators hat die folgende allgemeine Gestalt

```
path aPathName : aSourceClass -> aTargetClass =
    -edgeType1-> & -edgeType2->
end;
```

Dabei repräsentiert *aPathName* den Pfadbezeichner und *aSourceClass* sowie *aTargetClass* stellen die Anfangs- beziehungsweise Endknotenklasse des Pfades dar. Der eigentliche Pfadausdruck folgt nach dem Gleichheitszeichen.

Anstelle einer Konkatenation können selbstverständlich auch andere Operatoren wie `and`, `or` und `but_not` verwendet werden. Eine Verzweigung kann anhand eines *Branch*-Operators aufgrund der Auswertung eines Pfadprädikats realisiert werden, und zur Iteration von Teilpfaden stehen die Operatoren `*` sowie `+` zur Verfügung.

Pfade bieten demnach eine Möglichkeit, Beziehungen zwischen Knoten zu beschreiben sowie ausgehend von einem Startknoten die Menge der über einen beschriebenen Pfad zu erreichenden Endknoten zu ermitteln.

Über *Restriktionen* ist es weiterhin möglich, während einer Pfadauswertung zusätzliche Anforderungen zu formulieren. So kann anhand `def -Edge1->` beispielsweise überprüft werden, ob vom aktuellen Knoten eine Kante des Typs *Edge1* traversiert werden kann und es wird gegebenenfalls der Zielknoten zurückgeliefert. Weitere Möglichkeiten zur Restriktion bietet die Typeinschränkung durch den `:`-Operator sowie die Beschreibung von Einschränkungen anhand boolescher Ausdrücke mittels des `valid`-Operators.

Restriktionen können graphisch durch einen Doppelpfeil mit annotierter Einschränkung dargestellt werden, wobei die Pfeilspitze auf das betreffende Element gerichtet ist. Alternativ kann bei einer komplexen Restriktion eine textuelle Definition außerhalb der graphischen Repräsentation vorgenommen werden.

6.4 Tests in PROGRESS

Tests stellen eine allgemeine, graphisch notierte Suchanfrage an einen Graphen dar, liefern als Ergebnis einen Wahrheitswert und können daher zur Modellierung von Fallunterscheidungen herangezogen werden [Zuen96]. Neben den bereits bekannten Elementen wie Knoten, Kanten, Attribute, Pfadausdrücken und Restriktionen können Tests attributwertige, knotenwertige sowie knotentypwertige Parameter enthalten. Weiterhin ist die Darstellung optionaler Knoten (dargestellt durch ein gestricheltes Rechteck), optionaler Knotenmengen (dargestellt durch überlappende, gestrichelte Rechtecke) und obligatorischer Knotenmengen (dargestellt durch überlappende, durchgezogene Rechtecke) möglich. Zusätzlich können Graphmuster negative Knoten, Kanten und Pfade, notiert durch eine x-förmige Streichung, enthalten, welche eine Möglichkeit zur Formulierung von Ausschlussbedingungen ermöglichen.

Das folgende Test-Beispiel (siehe Abb. 6-3) enthält einige der soeben erwähnten Komponenten. Für eine detailliertere Erläuterung wird auf [Zuen96] verwiesen, welcher verschiedene dieser Elemente an einem Beispiel erklärt.

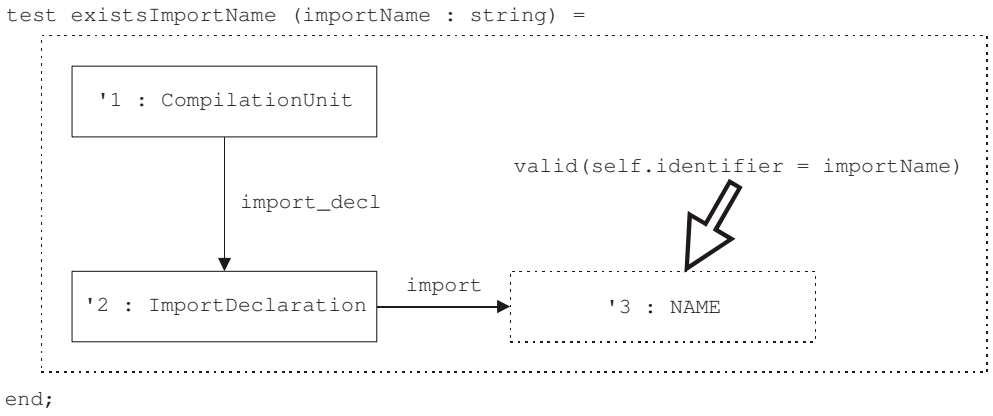


Abb. 6-3: PROGRES-Test

Der Test `existsImportName` überprüft, ob der über den Parameter `importName` angegebene Name eines Imports in einem Graphen vorhanden ist. Zu diesem Zweck wird eine Suche nach einem Vorkommen des angegebenen Musters in einem vorliegenden Graphen gestartet.

Ausgehend von Knoten `'1`, welcher eine `CompilationUnit` repräsentiert und über eine `import_decl`-Kante mit einem `ImportDeclaration`-Knoten verbunden ist, muss weiterhin eine `import`-Kante vorhanden sein, welche den Knoten `'2` mit einem optionalen `NAME`-Knotentyp verbindet. Existieren entsprechende Knoten für das vorgegebene Muster und ist die Restriktion `self.identifier=importName` für den Knoten `'3` gültig, ist der Test gelungen und eine importierte Klasse entsprechend `importName` ist in dieser Compilation Unit vorhanden. Ansonsten schlägt dieser Test fehl und es liegt ein negatives Ergebnis vor.

In Tests werden Knotenelemente zur Vereinfachung der graphischen Repräsentation allesamt als Rechtecke dargestellt. Lediglich anhand der Groß- und Kleinschreibung der Knotenbezeichner kann erkannt werden, ob es sich um einen Knotentyp oder eine Knotenklasse handelt.

Im Gegensatz zur Definition von Pfaden, welche grundsätzlich die Angabe eines vorgegeben Start- und Endknoten erfordern, ist dies bei Tests nicht der Fall. Hier wird kein Ausgangsknoten benötigt und lediglich nach dem angegebenen Muster im vorliegenden Graphen gesucht [Zuen96].

Alle in Tests vorkommenden Elemente sind auch innerhalb der nachfolgend beschriebenen Produktionen anwendbar. Im Unterschied zu Tests, welche zur Formulierung von Graphanfragen und zur Mustersuche herangezogen werden, ist es die Aufgabe von Produktionen, einen gegebenen Graphen zu transformieren.

6.5 Produktionen in PROGRES

Transformationen bilden die Basis zur Erzeugung und Manipulation schemakonsistenter Graphen und stellen den operationalen Teil einer Spezifikation dar. Transformationsregeln, sogenannte *Produktionen*, bestehen aus einer linken (*engl.* left hand side, LHS) und einer rechten (*engl.* right hand side, RHS) Regelseite. Eine auf diese Weise modellierte Graphveränderung ergibt sich dabei aus einem Vergleich der beiden Regelseiten, welche in PROGRES meist untereinander notiert werden [Zuen96].

Eine Produktion sucht nach dem in der linken Regelseite aufgeführten Muster innerhalb dem zugrundeliegenden Graphen und ersetzt dieses, soweit auffindbar und vorhandene Restriktionen erfüllbar, durch die in der rechten Regelseite aufgeführten Elemente. Demnach ist das Ausführen einer Produktion gleichzusetzen mit einer Mustersuche und anschließender Teilgraphersetzung.

Zur Veranschaulichung zeigt das folgende Produktionsbeispiel (siehe Abb. 6-4) eine Transformationsregel zur Erzeugung eines weiteren Import-Knotens innerhalb eines ASTs. Grundlage ist das in Abb. 6-1 dargestellte AST-Graphschema.

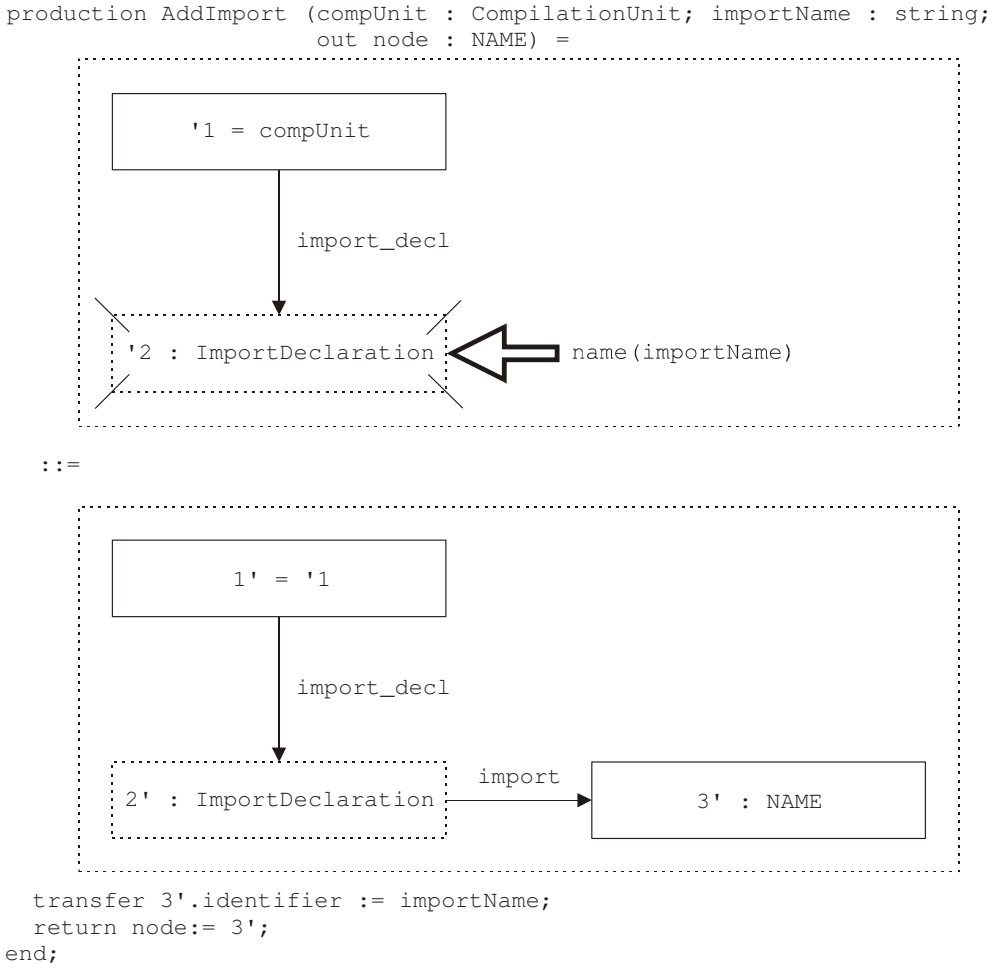


Abb. 6-4: PROGRES-Produktion

Die linke Regelseite der Produktion `AddImport` besteht aus einem Knoten `'1`, der die durch den Parameter angegebene Compilation Unit kennzeichnet. Dies wird festgelegt durch die Knoteninschrift `'1 = compUnit` und den Parameter `compUnit : CompilationUnit`. Der zweite Produktionsparameter `importName : string` wird innerhalb der Restriktion als Name für den neuen Knoten eine Rolle spielen. Weiterhin ist ein optionaler Knoten `'2` vom Typ `ImportDeclaration` enthalten, der von `'1` aus über eine `import_decl`-Kante erreichbar sein soll und das neu einzufügende Import-Element modelliert. Da der Knoten zu diesem Zeitpunkt noch nicht vorhanden sein darf, wird er als negativer Knoten dargestellt und die zu erfüllende Restriktion durch einen Doppelpfeil annotiert.

In der rechten Regelseite wird der Knoten `'1` der linken Regelseite wieder aufgeführt. Dies wird ausgedrückt durch die Knoteninschriften der Form `i'='i`. Solche Knoten bleiben während

der Transformationsausführung erhalten und werden in unveränderter Form in den neuen Graphzustand übernommen. Alle anderen Elemente, die zuvor in der linken Seite einer Produktion aufgeführt wurden und in der rechten Regelseite nicht mehr enthalten sind, werden im Zuge der Teilgraphersetzung gelöscht, d.h. die entsprechenden Knoten und Kanten werden aus dem Graph entfernt [Zuen96].

Falls nun der optionale Knoten `'2` erstellt werden konnte, wird zusätzlich ein Knoten `3'` der Klasse `NAME` erzeugt und eine neue `import`-Kante zwischen den Knoten `2'` und `3'` eingefügt. Dem Attribut `identifier` des Knotens `3'` wird zudem der Eingabeparameter `importName` zugeordnet. Dies ermöglicht die `transfer`-Zuweisung am Ende der Produktion.

Abschließend wird noch der `return`-Teil der Produktion abgearbeitet. Hier wird der zuvor über `out name : NAME` festgelegte Rückgabeparameter der Produktion mit dem Knoten `3'` belegt, welcher den Namen des neuen Import-Elements enthält.

Damit endet die Übersicht der wichtigsten PROGRES-Bestandteile und es kann nun eine Transformations-Spezifikation des *Extract Method*-Refactorings vorgenommen werden. Zusätzliche Informationen, insbesondere über weitere PROGRES-Elemente sowie die PROGRES-Syntax, sind über die Projekt-Homepage der RWTH Aachen[8] zu finden.

6.6 Spezifikation des Extract Method-Refactorings

Anhand der von PROGRES zur Verfügung gestellten Konzepte zur Beschreibung von Graphtransformationen werden nun die graphverändernden Manipulationen des *Extract Method*-Refactorings, welches in Kapitel 5 in Einzelschritte zerlegt wurde, formal durch PROGRES-Produktionen definiert. Die identifizierten Bedingungen hingegen können in Form von PROGRES-Tests formuliert und überprüft werden. Analog dem *Extract Method*-Vorgehen aus Abb. 5-1 werden im Folgenden die einzelnen Teilschritte in eine formale Spezifikation überführt.

[8] siehe *www-i3.informatik.rwth-aachen.de* → Research → PROGRES

6.6.1 Gültigkeit des Methodenbezeichners in Compilation Unit

Folgender Test (siehe Abb. 6-5) ermöglicht die Überprüfung auf Gültigkeit des gewählten Methodenbezeichners im Kontext der vorliegenden Compilation Unit gemäß dem in Kapitel 5.2 beschriebenen Vorgehen und bezieht sich somit auf Teilschritt (3) im *Extract Method*-Aktivitätsdiagramm.

Auf Grundlage des vorliegenden ASTs überprüft der Test `existsMethodNameInCU`, ob der vom Benutzer eingegebene Bezeichner `methodName` bereits für eine Methode in der vorliegenden Compilation Unit verwendet wird. Zu diesem Zweck wird nach dem vorgegeben Graphmuster innerhalb des ASTs gesucht.

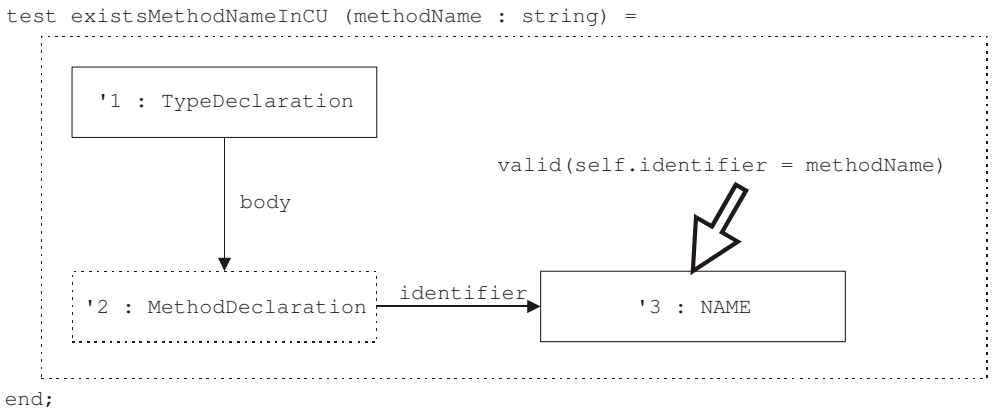


Abb. 6-5: existsMethodNameInCU-Test

Ausgehend von einem `TypeDeclaration`-Knotentyp muss eine `body`-Kante zu einem optionalen `MethodDeclaration`-Knotentyp existieren. Weiterhin muss ein `NAME`-Element, welches den Methodenbezeichner repräsentiert, über eine `identifier`-Kante ausgehend von der Methodendeklaration erreichbar sein. Anhand der Restriktion `valid(self.identifier = methodName)` wird überprüft, ob der Name der ermittelten Methode mit dem eingegebenen Methodenbezeichner übereinstimmt.

Da im günstigsten Fall kein Teilgraph entsprechend dem Suchmuster zu finden ist, existiert kein Vorkommen eines `MethodDeclaration`-Knotens und der Test scheitert. Aus diesem Grund wird Knoten '2 als optional vorkommend dargestellt. Liefert der Test hingegen ein positives Ergebnis, d.h. es konnte ein passender Teilgraph ermittelt werden, ist der gewählte

Methodenbezeichner bereits vergeben und es muss eine Fehlermeldung ausgegeben werden. Die Fehler-Behandlung wird ebenfalls als gegeben vorausgesetzt und ferner nicht näher spezifiziert.

6.6.2 Gültigkeit des Methodenbezeichners in Superklasse

Der folgende Test (siehe Abb. 6-6) entspricht der *Extract Method*-Aktivität (4) und sucht nach dem Vorkommen des angegebenen Methodenbezeichners in einer eventuell vorhandenen Superklasse der Compilation Unit. Zu diesem Zweck muss auf die im AST enthaltenen Bindungsinformationen zugegriffen werden, welche die benötigten Angaben enthalten.

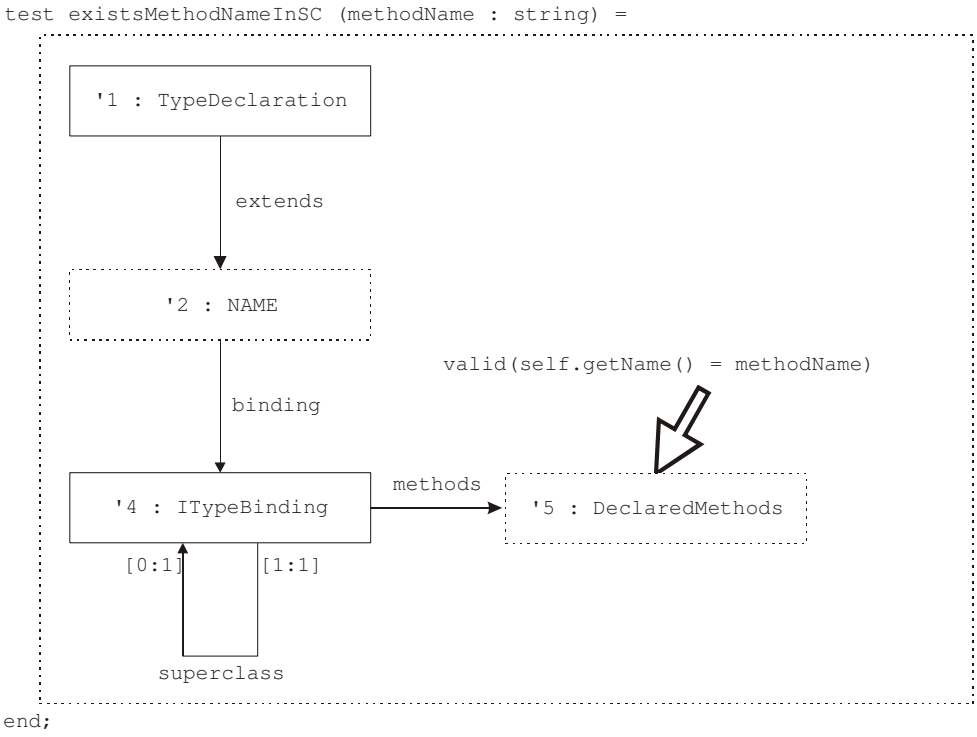


Abb. 6-6: existsMethodNameInSC-Test

Der Test `existsMethodNameInSC` sucht nach einem Subgraph folgender Gestalt: Ausgehend von einer `TypeDeclaration` muss eine `extends`-Kante zu einem `Name`-Knoten existieren, welcher die Superklasse repräsentiert. Ein solches Vorkommen ist optional, da keine Elternklasse vorhanden sein muss. Ansonsten können die Bindungsinformationen dazu genutzt werden, die Methoden der Superklasse zu identifizieren und diese mit dem gewünschten Methodenbezeichner zu vergleichen. Anhand der `superclass`-Kante kann in der Vererbungshierarchie aufgestiegen und die bereitgestellten Methoden der jeweils höheren Klasse zu einem

Vergleich herangezogen werden. Kann kein Teilgraph gefunden werden, der dem angegebenen Suchmuster entspricht, bricht der Test ab und der gewählte Methodenbezeichner ist insofern gültig, dass durch die Erzeugung einer neuen Methode keine geerbte Methode überschrieben wird.

Das Aufsteigen in der Vererbungshierarchie erfolgt durch eine Iteration über die vorhandenen `superclass`-Kanten innerhalb der Bindungsinformationen. Durch mehrmaliges Durchlaufen wird jeweils zu einer nächsthöheren Superklasse verzweigt. Die Darstellung in PROGRES bereitet jedoch Probleme, da Kanteniterationen nicht vorgesehen sind. Daher müsste auf alternative PROGRES-Konstrukte zurückgegriffen werden, was eine umfassendere Einarbeitung in diese Sprache erfordert und aus zeitlichen Gründen unterlassen wird.

6.6.3 Methodendeklaration erzeugen

Das Erzeugen einer neuen Methode anhand des zuvor gewählten Bezeichners erledigt folgende Produktion (siehe Abb. 6-7), die somit Aktivität (6) des *Extract Method*-Refactorings entspricht.

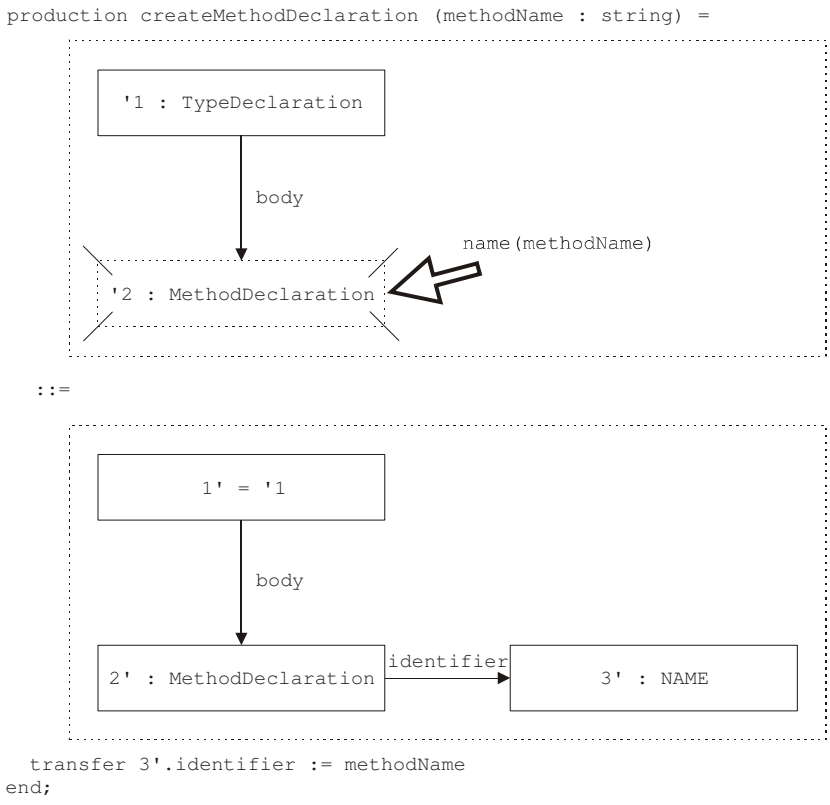


Abb. 6-7: createMethodDeclaration-Produktion

Die linke Regelseite der Produktion `createMethodDeclaration` ist entsprechend dem Produktionsbeispiel zur Erzeugung eines neuen Knotens aus Abb. 6-4 aufgebaut. Der optionale negative Knoten `'2` modelliert die zu diesem Zeitpunkt noch nicht existierende Methodendeklaration, welche durch die angegebene Restriktion bestimmt wird.

Durch die rechte Regelseite werden alle zuvor aufgeführten Knoten und Kanten unverändert übernommen und zusätzlich ein neuer `MethodDeclaration`-Knotentyp erzeugt. Ausgehend von dieser Methodendeklaration werden dem Graphen eine neue `identifier`-Kante sowie ein `NAME`-Element hinzugefügt, wobei letzteres den Methodenbezeichner repräsentiert. Die abschließende Zuweisung des Eingabeparameters `methodName` an das Attribut `identifier` des Knotens `3'` geschieht im textuellen Bereich am Ende der Produktion.

Werden die Tests `existsMethodNameInCU` und `existsMethodNameInSC` ausgeführt und enden beide aufgrund von nicht auffindbaren Subgraphen, die den vorgegebenen Suchmustern entsprechen, kann nachfolgend die soeben definierte Produktion zur Erzeugung eines neuen Methoden-Knotens gefahrlos ausgeführt werden. Mit anderen Worten lassen sich anhand von Tests diejenigen Vorbedingungen überprüfen, welche vor Anwendung einer Produktion gelten beziehungsweise erfüllt sein müssen.

6.6.4 Transaktionen

Zur Verknüpfung von Tests und Produktionen stellt PROGRES sogenannte Transaktionen bereit. Transaktionen erlauben beispielsweise eine sequentielle Ausführung, dargestellt durch den &-Operator in Form von `Prod1 & Prod2 & Prod3...` und können Fallunterscheidungen sowie Iterationen enthalten. Für weitere Operatoren wird an dieser Stelle auf die Syntaxbeschreibung von PROGRES verwiesen.

Nachfolgend zeigt Abb. 6-8 ansatzweise diejenige Transaktion, welche das *Extract Method*-Refactoring beschreibt und die bisher spezifizierten Tests sowie die Produktion zur Erzeugung der neuen Methodendeklaration beinhaltet. Sie repräsentiert damit die im Aktivitätsdiagramm (siehe Abb. 5-1) enthaltenen Aktivitäten (3) bis (6).

```
transaction extractMethod (methodName : string) =
  choose
    when existsMethodNameInCU(methodName) then error;
  else
    choose
      when existsMethodNameInSC(methodName) then error;
    else
      createMethodDeclaration(methodName);
    end;
  end;
end;
```

Abb. 6-8: Extract Method-Transaktion

Die Transaktion `extractMethod` erhält als Eingabeparameter den Namen der zu erzeugenden Methode und beginnt mit einer Fallunterscheidung, dargestellt durch das Schlüsselwort `choose`. Diese beinhaltet eine bedingte Alternative der Form `when Test then Prod`. Liefert die erste Bedingung ein positives Ergebnis, wird eine Fehlerbehandlung, angedeutet durch *error*, eingeleitet. Ansonsten wird der entsprechende `else`-Zweig abgearbeitet und eine zweite Fallunterscheidung anhand eines weiteren Tests durchgeführt. Kommt hier ein positives Ergebnis zustande, wird der weitere Ablauf mit einer Fehlerausgabe abgebrochen. Anderenfalls erfolgt ein Sprung in den zugehörenden `else`-Zweig und die erste Produktion wird gestartet.

Anhand dieser Transaktion wird deutlich, wie Tests und Produktionen miteinander verkettet werden können. Zur vollständigen Beschreibung des *Extract Method*-Refactoring sind noch einige weitere Abfragen und Graphmanipulationen zu definieren, welche später die soeben erstellte *Extract Method*-Transaktion vervollständigen werden.

6.6.5 Vorkommen lokaler Variablen

Aufgabe des nächsten Tests (siehe Abb. 6-9) ist das Überprüfen auf Existenz von lokalen Variablen innerhalb der zu verschiebenden Codezeilen. Dies entspricht Aktivität (7) im vorliegenden *Extract Method*-Aktivitätsdiagramm. Hierzu muss wiederum auf die entsprechenden Bindungsinformationen zugegriffen werden, um eine Entscheidung treffen zu können. Weiterhin wird innerhalb dieses Tests ein PROGRES-Pfadausdruck eingesetzt, um komplexe Kantenzüge auf einfache Weise darstellen zu können.

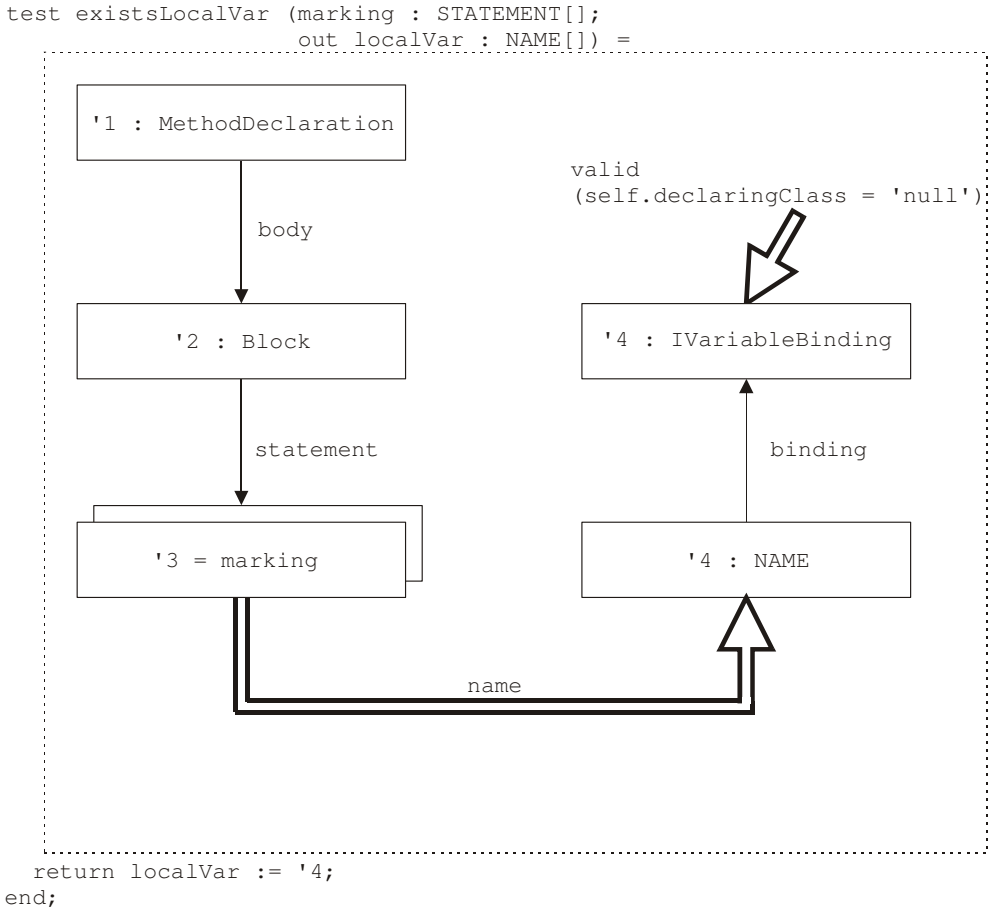


Abb. 6-9: existsLocalVar-Test

Als Eingabeparameter erhält der Test `existsLocalVar` diejenigen `STATEMENT`-Knoten, welche in eine neue Methode zu verschieben sind und dementsprechend zuvor markiert wurden. Diese werden durch das Array `marking` repräsentiert. Als Rückgabe des Tests sind die String-Arrays `name` und `type` vorgesehen, welchen während der Testausführung der Name sowie der Typ einer eventuell ermittelten lokalen Variable zugeordnet wird.

Die zu verschiebenden Codezeilen befinden sich stets innerhalb des Rumpfs einer Methode und sind dort in Form von `STATEMENT`-Elementen hinterlegt. Diese Aussage wird im Suchmuster durch die Knoten `'1` bis `'3` dargestellt. Anhand der Knoteninschrift `'3 = marking` wird sichergestellt, dass lediglich diejenigen `STATEMENT`-Knoten in die weitere Betrachtung einbezogen werden, welche die markierten Codezeilen repräsentieren.

Um die Erstellung des Tests zu vereinfachen wird ein `name`-Pfad zwischen einer `STATEMENT`- und einer `NAME`-Knotenklasse eingeführt. Der Pfad stellt einen Kantenzug dar, welcher diejenigen Zielknoten `'4` enthält, die den Bezeichner einer Variablen darstellen.

Wie aus der Java-Grammatik (und auch dem Metamodell) zu entnehmen ist, repräsentiert ein Statement eine Vielzahl möglicher Statement-Konstrukte. Die Definition des `name`-Pfades umfasst daher eine Oder-Verknüpfung aller (Sub-)Pfade, welche ausgehend von den verschiedenen Statement-Konstrukten die nötigen Kanten bis hinab zu den entsprechenden `NAME`-Knoten traversieren. Auf eine formale Definition wird aufgrund der Komplexität an dieser Stelle verzichtet und die Existenz des `name`-Pfades als gegeben vorausgesetzt.

Sofern ein entsprechender `NAME`-Knoten mit dem vorliegenden Suchmuster übereinstimmt, können die zugehörigen Bindungsinformationen, repräsentiert durch die Knoten `'5` und `'6`, betrachtet werden. Wie bereits in Kapitel 4.5 beschrieben befinden sich dort diejenigen Informationen, welche die Entscheidung über das Vorhandensein einer lokalen Variable ermöglichen.

Verläuft der Test erfolgreich, d.h. konnten dem Suchmuster entsprechende Subgraphen ermittelt und die an Knoten `'6` notierte Restriktion `self.declaringClass = 'null'` erfüllt werden, wurden eine oder mehrere lokale Variablen identifiziert. Diese müssen später als Parameter der neuen Methode bekanntgegeben werden. Daher werden sie im `return`-Bereich des Tests entsprechend gespeichert und stehen somit für die Produktion zur Erzeugung des Methodenparameters zur Verfügung.

6.6.6 Methodenparameter erzeugen

Sind lokale Variablen von dem durchzuführenden *Extract Method*-Refactoring betroffen, müssen diese anhand von Methodenparametern innerhalb der neuen Methode bekanntgegeben werden. Dies entspricht Aktivität (8) und wird von folgender Produktion (siehe Abb. 6-10) erledigt.

Die linke Regelseite beschreibt die Suche nach dem Teilgraphen, welcher die zuvor neu erzeugte Methode mit entsprechendem Methodenbezeichner beinhaltet. Dies setzt

selbstverständlich voraus, dass ein solches Element durch die Produktion `createMethodDeclaration` im Zuge des *Extract Method*-Refactorings erstellt wurde.

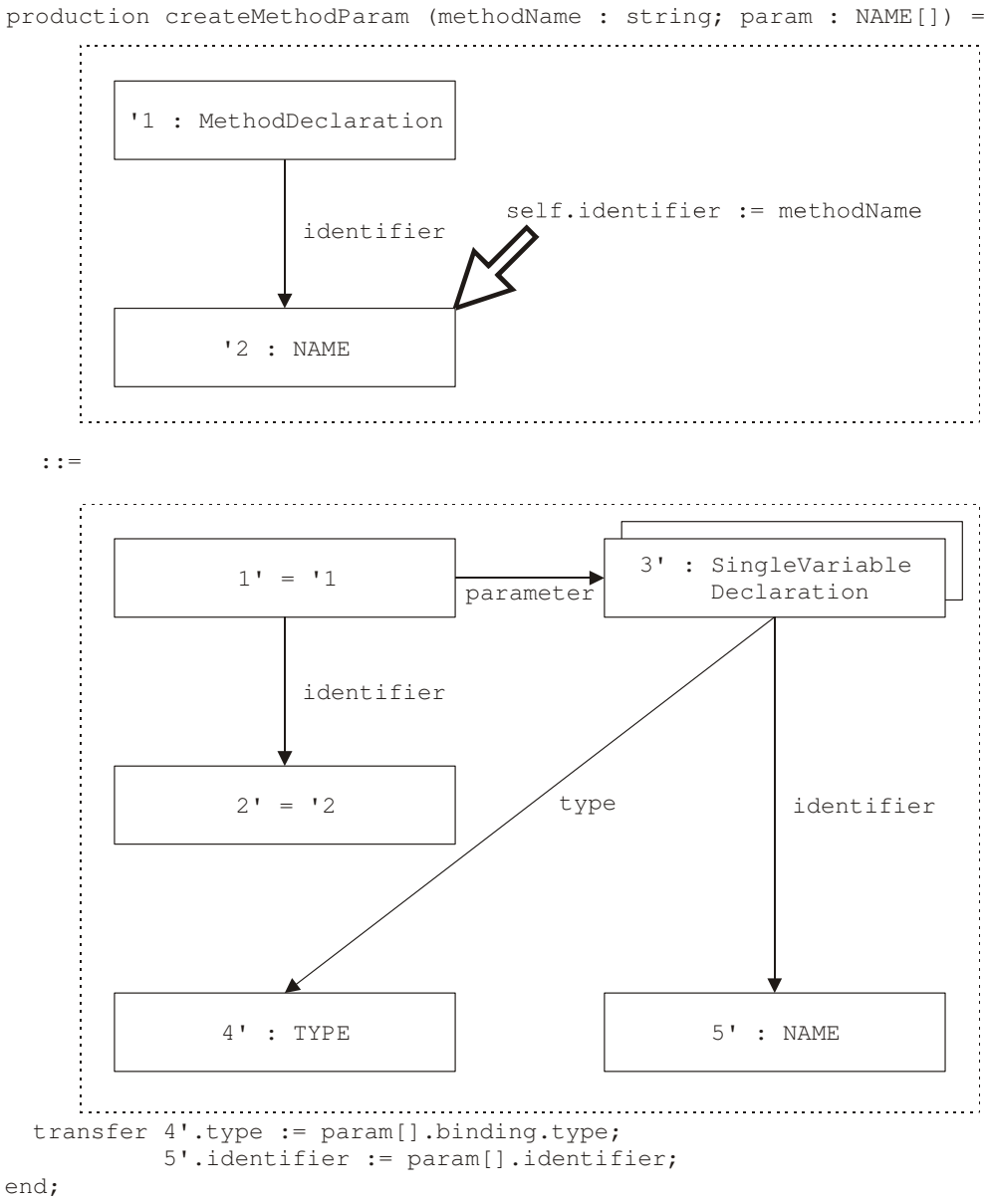


Abb. 6-10: createMethodParam-Produktion

Die rechte Regelseite der Produktion erzeugt gemäß dem als Eingabeparameter übergebenen `param`-Arrays eine entsprechende Anzahl von `SingleVariableDeclaration`-Knoten, welche die Parameter der Methode repräsentieren. Zusätzlich werden dem existierenden AST-Graphen `TYPE`- und `NAME`-Knoten hinzugefügt, welche Parameter-Typ und -Name darstellen. Die Zuweisung an die entsprechenden Attribute wird durch den `transfer`-Bereich am Ende der

Produktion vorgenommen. Damit ist ein weiterer Refactoring-Teilschritt abgeschlossen und es kann der Methodenaufruf erzeugt werden.

6.6.7 Methodenaufruf erzeugen

Die folgende Produktion `createMethodCall` (siehe Abb. 6-11) erzeugt einen Methodenaufruf für die zuvor erstellte Methodendeklaration und entspricht damit Aktivität (9) des *Extract Method*-Aktivitätsdiagramms.

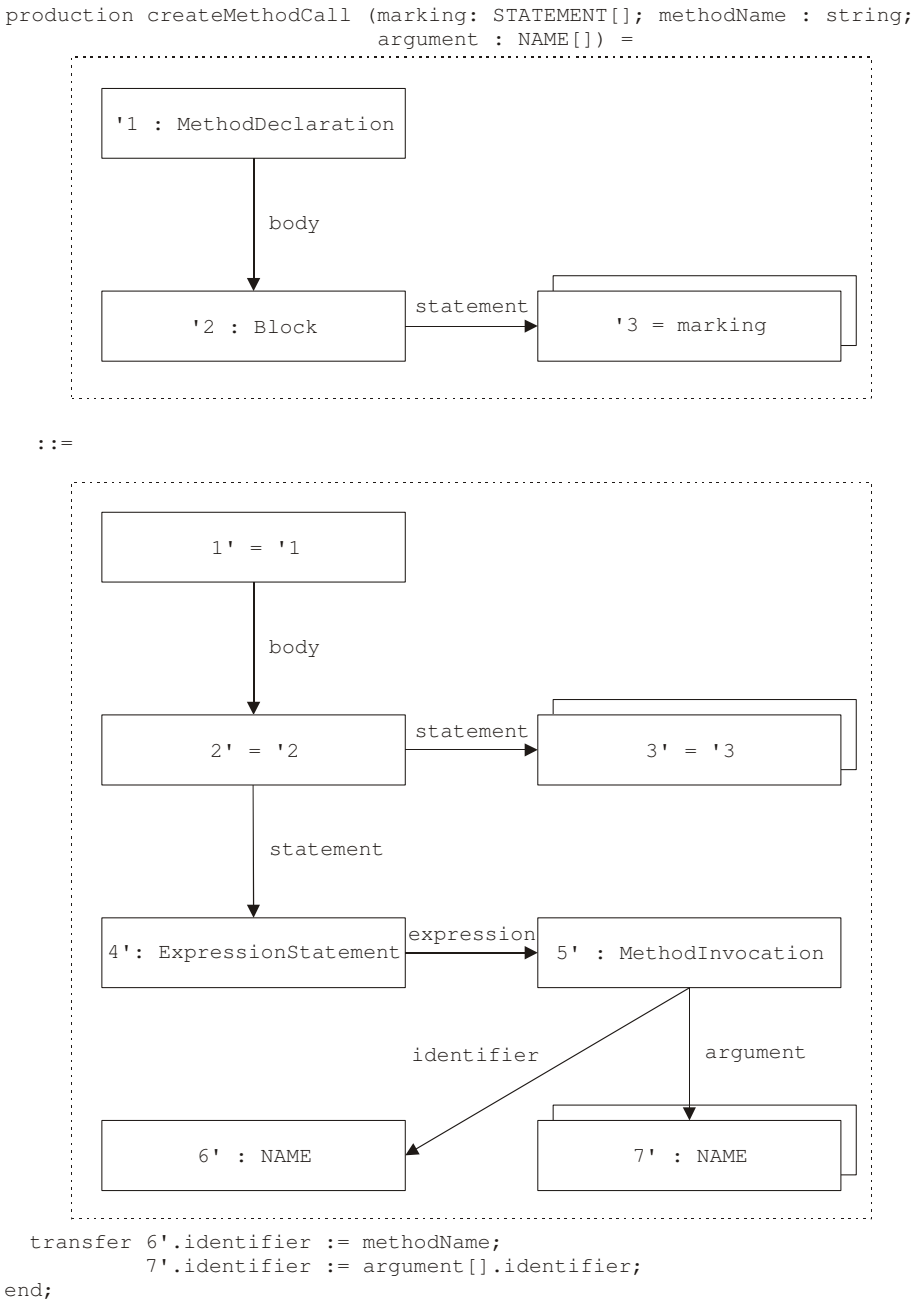


Abb. 6-11: createMethodCall-Produktion

Die linke Regelseite sucht nach denjenigen `STATEMENT`-Elementen, welche durch das *Extract Method*-Refactoring in eine neue Methode ausgelagert werden sollen und bereits markiert vorliegen.

Die rechte Regelseite hinterlässt den gefundenen Teilgraphen unverändert, da die selektierten Codezeilen und dementsprechend die markierten Statements erst in einer folgenden Graphtransformation verschoben werden. Vielmehr wird über eine zusätzliche `statement`-Kante ein neuer `ExpressionStatement`-Knoten eingebunden Eine `expression`-Kante verbindet diesen mit einem `MethodInvocation`-Knoten zur Repräsentation des Methodenaufrufs. Eine von dort ausgehende `identifier`-Kante stellt eine Beziehung zu einem `NAME`-Element her, welches den Bezeichner der aufzurufenden Methode kennzeichnet. Weitere `NAME`-Knoten, entsprechend der Anzahl der Argumente des Methodeaufrufs, sind zu erstellen und anhand `argument`-Kanten mit dem `MethodInvocation`-Knoten zu verbinden.

Zur Angabe der Position des einzufügenden Methodenaufrufs vor den markieren `STATEMENT`-Knoten wäre eine geordnete Kantenfolge hilfreich. Da eine solche Reihenfolge jedoch nicht in PROGRES vorgesehen ist und stattdessen eine Knotenliste durch Hinzufügen zusätzlicher Kanten erzeugt werden müsste, wird an dieser Stelle auf eine genauere Spezifikation verzichtet.

Damit ist die Durchführung dieser Produktion abgeschlossen und ein Aufruf der zuvor erstellten neuen Methode wurde in den AST-Graphen durch Hinzufügen zusätzlicher Knoten erzeugt. Der letzte Schritt des *Extract Method*-Refactorings wird durch eine weitere Graphtransformation eingeleitet.

6.6.8 Markierte Codezeilen verschieben

Aufgabe der Produktion `moveMarkedCode` (siehe Abb. 6-12) ist es, die markierten Codezeilen in den Rumpf der neuen Methode zu verschieben. Dieser Schritt wird entgegen der Beschreibung des *Extract Method*-Refactorings erst nach einem Erzeugen des Methodenaufrufs ausgeführt, da anderenfalls die selektierten Statement-Elemente bereits verschoben wurden und die Position des zu erstellenden Methodenaufrufs nicht mehr identifiziert werden kann. Ebenso kann auf das Erstellen einer Kopie der markierten Codezeilen analog dem *Extract Method*-Vorgehen verzichtet und die betreffenden Elemente lediglich an ihre neue Position verschoben werden.

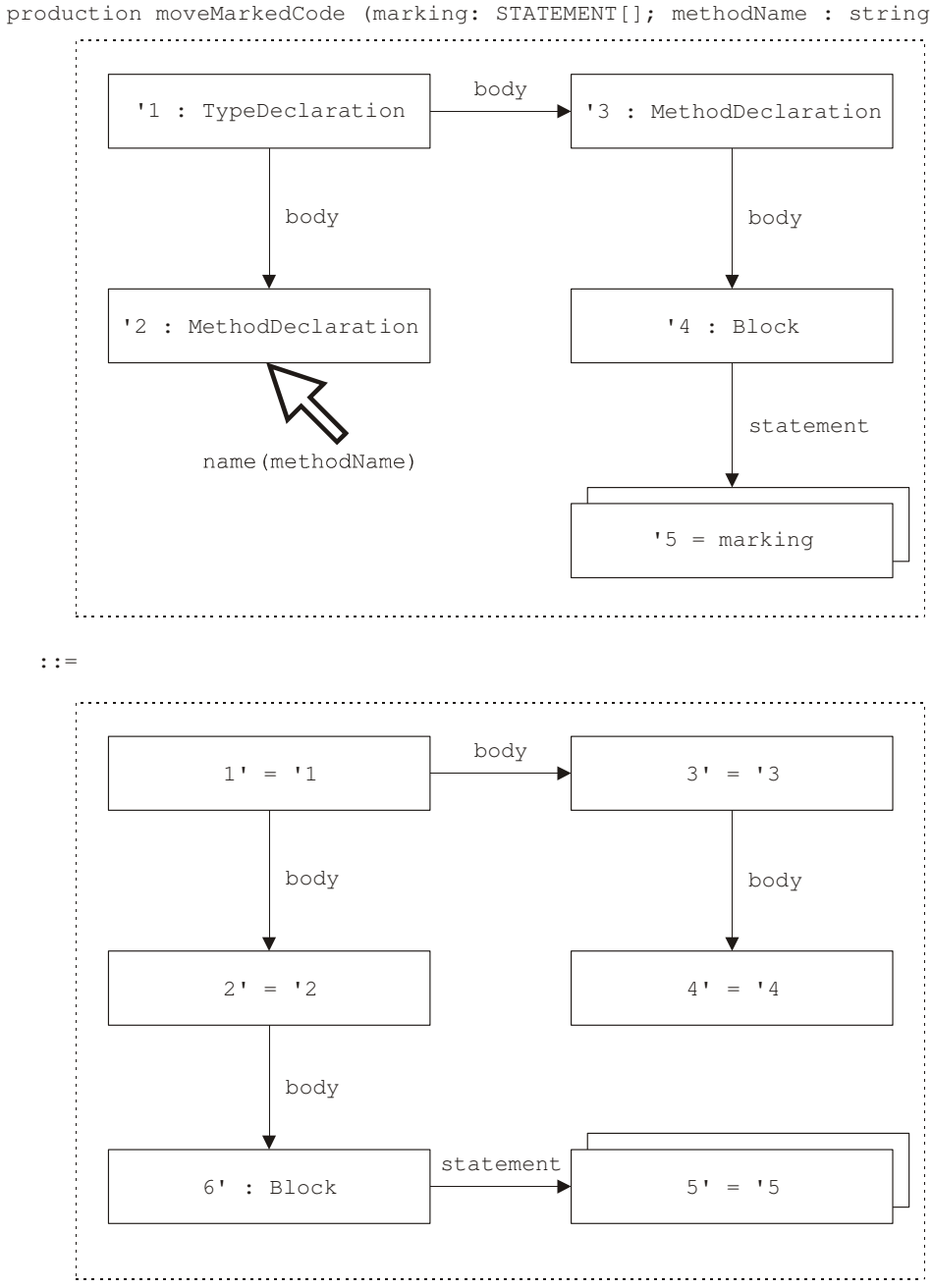


Abb. 6-12: moveMarkedCode-Produktion

Die linke Regelseite der Produktion sucht nach einem Teilgraphen, welcher innerhalb einer `MethodDeclaration` die zu verschiebenden `STATEMENT`-Elemente enthält – dargestellt durch die Knoten `'3` bis `'5`. Weiterhin ist derjenige `MethodDeclaration`-Knoten zu identifizieren, welcher die zuvor erzeugte Methodendeklaration repräsentiert und somit das Ziel der zu verschiebenden Elemente verkörpert. Dieser wird anhand Knoten `'2` modelliert.

Die Anwendung der rechten Regelseite entfernt die `statement`-Kanten und die damit verbundenen Knoten aus dem Rumpf der ursprünglichen Methode. Ausgehend von Knoten `2'`, welcher die neue Methodendeklaration repräsentiert, wird eine neue `body`-Kante erzeugt, die den Rumpf der neuen Methode – dargestellt durch den `Block`-Knoten `6'` – dem Graphen hinzufügt. In diesen Methodenrumpf werden sodann die selektierten Codezeilen verschoben. Dies wird erreicht durch Erzeugen neuer `statement`-Kanten, welche eine Verbindung zu den durch die linke Regelseite markierten `STATEMENT`-Elementen herstellen – angedeutet durch die Knoteninschrift `5'='5`.

Damit ist das Verschieben der betroffenen AST-Elemente abgeschlossen und somit der letzte Schritt in der Ausführung des *Extract Method*-Refactorings erfolgt. Nachfolgend kann nun die vollständige *Extract Method*-Transaktion erstellt werden.

6.6.9 Extract Method-Transaktion

Nachdem letztendlich alle zuvor beschriebenen Aktivitäten in Form von Tests und Produktionen auf Basis von PROGRES-Sprachelementen spezifiziert wurden, kann die bereits in Abb. 6-8 erläuterte Transaktion vervollständigt werden. Die nachfolgende dargestellte Transaktion (siehe Abb. 6-13) stellt nun endlich eine formelle Transformationsbeschreibung für das *Extract Method*-Refactoring dar.

```
transaction extractMethod (marking : STATEMENT[]; methodName : string) =
  use localVar : NAME[];
  do
    choose
      when existsMethodNameInCU(methodName)
        then error;
    else
      choose
        when existsMethodNameInSC(methodName)
          then error;
      else
        createMethodDeclaration(methodName);
      & choose
          when existLocalVar(marking, out localVar)
            then createMethodParam(methodName, localVar);
        end;
        createMethodCall(marking, methodName, localVar);
      & moveMarkedCode(marking, methodName);
      end;
    end;
  end;
end;
```

Abb. 6-13: Extract Method-Transaktion

Anhand des `use`-Statements werden die Variablen `localVar` und `localVarType` für den `do`-Bereich der Transaktion deklariert. Ihnen werden die Rückgabeparameter des Tests `existsLocalVar` zugeordnet, damit sie später für die Produktionen `createMethodParam` sowie `createMethodCall` zur Verfügung stehen und an diese übergeben werden können.

Die Transaktion führt eine sequentielle Abarbeitung der spezifizierten Produktionen durch, wobei die einzelnen Graphtransformationen ausgehend von bestimmten Vorbedingungen, dargestellt in Form von Tests, aufgerufen werden.

Im ersten `choose`-Abschnitt wird die Graphtransformation zur Erzeugung einer neuen Methodendeklaration (`createMethodDeclaration`) lediglich dann ausgeführt, wenn der als Eingabeparameter übergebene Methodenbezeichner (`methodName`) im Kontext der Compilation Unit sowie eventuell vorhandener Superklassen gültig ist. Dies gewährleistet die Ausführung der Tests `existsMethodNameInCU` und `existsMethodNameInSC`. Die Gültigkeit bezieht sich auf doppelt vorhandene Methodenbezeichner in einer Java-Klasse sowie auf das Überschreiben von Methoden, welche bereits in einer Superklasse deklariert wurden und somit in einer Java-Klasse durch Vererbung bereitstehen. In diesem Fall muss eine Fehlerbehandlung ausgeführt und der Benutzer entsprechend benachrichtigt werden.

Der zweite `choose`-Bereich überprüft die zu verschiebenden Codezeilen auf das Vorhandensein von lokalen Variablen anhand des Tests `existsLocalVar`. Konnten solche identifiziert werden, müssen sie per Methodenparameter für den Code innerhalb der neuen Methode verfügbar gemacht werden. Daher ist aus den gefundenen lokalen Variablen und dessen Typen eine Parameterdefinition durch Anwendung der Produktion `createMethodParam` zu erzeugen. Anschließend sowie für den Fall, dass keine lokalen Variablen vorkommen, wird der Methodenaufruf anhand der Produktion `createMethodCall` erzeugt.

Die letzte Produktion `moveMarkedCode` verschiebt den markierten Codebereich in den Rumpf der neu erzeugten Methode, welche durch den zuvor hinzugefügten Methodenaufruf ausgeführt wird. Damit ist die Transaktion abgeschlossen und es liegt ein modifizierter AST-Graph vor, dessen Struktur gemäß dem *Extract Method*-Refactoring in einen neuen konsistenten Zustand überführt wurde.

6.7 Anmerkung

Die anschauliche graphische Darstellung der PROGRES-Tests und -Produktionen stellt zwar einen klaren Vorteil dar, andererseits ist das Erstellen dieser Konstrukte nur unter Einsatz eines unterstützenden Modellierungstools effizient zu bewerkstelligen. Zudem bereitet die Beschreibung von Kanteniterationen (siehe Abb. 6-6) Probleme. Da Kanten in PROGRES zudem ungeordnet vorliegen, kann auch die Produktion zur Einfügung des Aufrufs der neuen Methode an einer bestimmten Position nur unter Anwendung komplizierter Konstrukte dargestellt werden (siehe Abb. 6-11). Daher wird im folgenden Kapitel eine weitere Spezifikation des *Extract Method*-Refactorings unter Verwendung der Sprachen GRAL und GReQL vorgenommen.

Kapitel 7
Transformations-Spezifikation mit GRAL/GReQL

Das vorliegende Kapitel bietet einen Überblick über die Sprachen GRAL und GReQL, auf deren Basis sodann eine algorithmische Beschreibung von Graphanfragen und -transformationen zur Spezifikation des *Extract Method*-Refactorings erfolgt.

7.1 GRAL

Die Spezifikationssprache GRAL (GRAph specification Language) wurde 1991 am Institut für Softwaretechnik der Universität Koblenz erstellt und seither in zahlreichen Projekten eingesetzt sowie weiterentwickelt. Das Resultat ist die aktuell vorliegende Sprachversion 2.0, dessen vollständige Spezifikation in [Fran97] zu finden ist.

Die Sprache GRAL dient zur Spezifikation von Restriktionen auf *TGraphen* (siehe [EbFr94]). Ein *TGraph* ist ein angeordneter gerichteter Graph, der aus typisierten Knoten und Kanten besteht, welche allesamt durch Attribute beschrieben werden können [Dahm98]. Ein *angeordneter gerichteter* Graph wird definiert durch seine Knoten- und Kantenmenge sowie eine Inzidenzfunktion, welche jedem Knoten eine *geordnete* Folge der ein- und ausgehenden Kanten zuordnet. Die *Typisierung* von Knoten und Kanten erfolgt, indem den Elementen ein entsprechender Typbezeichner zugewiesen wird. Weiterhin kann durch eine *Attributierung* jedem Knoten und jeder Kante eine Menge von Attributinstanzen zugewiesen werden, welche in Form von Tupeln, bestehend aus Attributbezeichner und zugehörigem Attributwert, vorliegen [EbFr94].

Da GRAL eine $\mathbb{Z}$-ähnliche Sprache (siehe [Spiv92]) darstellt, ist sie an Prädikatenlogik und Mengenlehre orientiert. GRAL-Prädikate sind in polynomineller Zeit testbar und erlauben somit ein effizientes Überprüfen von in GRAL formulierten Ausdrücken [Eber96]. Eine GRAL-Spezifikation setzt sich aus einer Reihe von GRAL-Prädikaten zusammen, welche die Struktur und Eigenschaft eines Graphen sowie dessen Kanten- und Knotenelemente beschreiben.

7.1.1 Prädikate

Ein GRAL-Prädikat besteht entweder aus einer logischen Konstante (*true*, *false*), einem Prädikat-Term oder einem strukturierten Prädikat [Fran97]. *Prädikat-Terme* setzen sich aus einem Prädikat-Symbol und zugehörigen Argumenten zusammen, wobei die Symbole zur Beschreibung von Knoten- und Kanteneigenschaften (*isIsolated*, *isRoot*, ...) sowie Struktureigenschaften (*isConnected*, *isTree*, …) dienen[9]. *Strukturierte Prädikate* hingegen bauen sich aus (komplexen) Prädikaten, Quantoren (∃, ∀) und logischen Operatoren (∧, ∨, …) auf und haben in der allgemeinsten Form folgende syntaktische Struktur:

quantifier variableDeclarations | *variableContraints* • *predicate*

7.1.2 Pfadbeschreibungen

Ein weiteres Merkmal von GRAL ist die Möglichkeit Aussagen über Pfade zu treffen, welches durch Pfad-Beschreibungen ermöglicht wird. Eine Pfad-Beschreibung ist ein regulärer Ausdruck, der zusammengefasst wie folgt induktiv definiert ist [Fran97]:

1. Eine Pfad-Beschreibung besteht aus einem Kantensymbol (⇀, ↼, ⇌), optional annotiert mit einem Kantentyp (wie $⇀_{edge1}$) und gefolgt von einem •-Symbol mit annotiertem Knotentyp (wie $⇀_{edge1}\ •_{vertex1}$).

2. Für zwei Pfad-Beschreibungen p_1 und p_2 gilt:
   a. die Sequenz p_1p_2,
   b. die Iteration p_1^* oder p_1^+ sowie
   c. die Alternative $(p_1 \mid p_2)$

   sind gültige Pfad-Beschreibungen.

7.1.3 Pfad-Prädikate

In einer GRAL-Spezifikation werden Pfad-Beschreibung zur Definition von Pfad-Prädikaten und Pfad-Ausdrücken verwendet. Ein *Pfad-Prädikat* ist ein spezielles GRAL-Prädikat, welches Erreichbarkeitsrestriktion beschreibt und somit eine Aussage bezüglich zweier Knoten eines

[9] Viele der am häufigsten benötigten effizienten Graphfunktionen sind bereits in GRAL definiert und stehen innerhalb einer GRAL-Spezifikation zur Verfügung.

vorliegenden Graphen trifft. Ein Pfad-Prädikat besteht aus einer Pfad-Beschreibung, die in diesem Fall als Prädikatsymbol verwendet wird, sowie zugehörigen Argumenten [Fran97].

So lässt sich beispielsweise anhand des Pfad-Prädikats

$$v \rightarrow_{edgeType1} \rightarrow_{edgeType2} w$$

ausdrücken, dass die beiden Knoten *v* und *w* durch einen Pfad verbunden sind, welcher bei *v* startet, aus einer Vorwärtskante vom Typ *edgeType1* gefolgt von einer Kante vom Typ *edgeType2* besteht und bei *w* endet.

7.1.4 Pfad-Ausdrücke

Ein *Pfad-Ausdruck* hingegen dient zur Ermittlung derjenigen Knotenmenge, welche ausgehend von einem Startknoten über einen Pfad zu erreichen ist. Ein Pfad-Ausdruck besteht aus einer Pfad-Beschreibung, die in diesem Fall als Funktionssymbol vorliegt, sowie zugehörigen Argumenten [Fran97]. So lassen sich beispielsweise durch den Pfadausdruck

$$v \rightarrow_{edgeType1} \rightarrow_{edgeType2}$$

diejenigen Knoten identifizieren, welche ausgehend von Startknoten *v* über einen Pfad bestehend aus einer Kante vom Typ *edgeType1* gefolgt von einer Kante vom Typ *edgeType2* zu erreichen sind.

Hiermit endet der Überblick über die Spezifikationssprache GRAL, da die bisher erläuterten Konzepte die nötigen Informationen bereitstellen, um die Teilschritte des *Extract Method*-Refactorings nun anhand von GRAL spezifizieren zu können. Ergänzend wird im folgenden Abschnitt die Query-Sprache GReQL vorgestellt, damit auch Graphanfragen auf einfache Weise notiert werden können.

7.2 GReQL

GReQL (GUPRO-Repository Query Language) ist eine reine Anfragesprache[10] für TGraphen und wurde im Rahmen des GUPRO-Projekts[11] am Institut für Softwaretechnik der Universität Koblenz entwickelt. GReQL basiert im Wesentlichen auf der Sprache GRAL und liegt momentan in der Sprachversion 1.3 vor, deren Beschreibung und Spezifikation in [KaKu01] zu finden ist.

7.2.1 Anfragen

GreQL-Anfragen, sogenannte *FWR-Ausdrücke*, haben das Format

```
FROM declList
[WITH predicate]
REPORT reportList END
```

und liefern als Ergebnis stets eine Multimenge (sog. *Bag*) von Ergebnistupeln [KaKu01]. Eine Anfrage kann Knoten- und Kantenbezeichner oder -typen, Attributwerte sowie Pfade beinhalten.

Das Schlüsselwort `FROM` leitet den Deklarationsteil eines FWR-Ausdrucks ein. Hier werden in einer *declList* diejenigen Variablen vereinbart, welche innerhalb der weiteren GreQL-Anfrage benutzt werden. Als Wertebereich können alle im Schema vorkommenden Knoten (V{*nodeType*}) und Kanten (E{*edgeType*}) verwendet werden.

Das Schlüsselwort `WITH` stellt den optionalen Bedingungsteil einer Anfrage dar, wobei *predicates* die zu überprüfende Aussage formuliert. Der Aufbau dieser Prädikate entspricht der bereits in GRAL vorgestellten Syntax.

Die Ergebnisbeschreibung, beginnend mit dem Schlüsselwort `REPORT`, legt die Gestalt des Anfrage-Ergebnisses fest. Der Rückgabeparameter einer GReQL-Query wird durch *reportList* repräsentiert und besteht aus einer kommagetrennten Liste beliebiger Ausdrücke. Standardmäßig liefert `REPORT` eine Multimenge als Ergebnis, d.h. Werte können mehrfach vorkommen. Alternativ kann mit der Angabe von `REPORT SET` die Rückgabe einer einfachen

[10] Im Gegensatz zu bspw. SQL sind in GReQL keine Möglichkeiten zur Manipulation vorhanden.
[11] siehe http://www.gupro.de

Wertmenge erzwungen werden, die keine doppelten Elemente enthält. Das nachfolgende Schlüsselwort `END` kennzeichnet schließlich das Ende der Graphanfrage.

Zur Auswertung einer GReQL-Anfrage wird konzeptionell für jede mögliche Belegung der deklarierten Variablen überprüft, ob das angegebene Prädikat erfüllt ist, und dementsprechend die im Ergebnisbereich spezifizierten Ausgabeparameter belegt [KaKu01]. Zudem ist es möglich, komplexe Anfragen durch Schachtelung zu erzeugen. Da im `WITH`- und `REPORT`-Bereich einer Query beliebige GReQL-Ausdrücke verwendet werden dürfen, können hier wiederum FWR-Ausdrücke auftreten.

7.2.2 USING-Klausel

Die Einführung freier Variablen, die sodann in GReQL-Ausdrücken verwendet werden können, ermöglicht die `USING`-Klausel. Dieser Mechanismus gestattet beispielsweise das Erzeugen parametrisierbarer GReQL-Anfragen, die somit einem Funktionsaufruf entsprechen. Die allgemeinste Gestalt einer derartigen GReQL-Anfrage ist nachfolgend dargestellt:

```
USING variableList :
      FROM declList
      [WITH predicate]
      REPORT reportList END
```

Somit ist es möglicht, die in der *`variableList`* deklarierten Variablen in dem nachfolgenden FWR-Ausruck verwenden zu können.

7.2.3 Pfadbeschreibungen

Ein weiteres wichtiges Sprachkonstrukt von GReQL sind Pfadbeschreibungen, welche bereits ausführlich in GRAL vorgestellt wurden. Die Notation der Kantensymbole wird in GReQL folgendermaßen dargestellt:

`-->{`*`edgeType`*`}`, `<--{`*`edgeType`*`}` oder `<-->{`*`edgeType`*`}`

Dabei bestimmt die Richtung des Pfeilsymbols die Beziehungsrichtung und *`edgeType`* repräsentiert die optionale Angabe eines Kantentyps. Analog zu GRAL werden auch GReQL-Pfadbeschreibungen zur Definition von Pfad-Prädikaten und -Ausdrücken verwendet.

Im weiteren Verlauf wird diese soeben vorgestellte ASCII-Notation auch für GRAL-Ausdrücke verwendet, um eine einheitliche Darstellung zu gewährleisten. Die erläuterten Grundlagen zur Erstellung von GReQL-Anfragen sollten ausreichen, um hierauf aufbauend im folgenden Abschnitt die einzelnen *Extract Method*- Spezifikationen formulieren zu können. Daher wird für weitere Informationen zur Erstellung von Graphanfragen auf die GReQL-Sprachreferenz und -Syntax verwiesen (siehe [KaKu01]).

7.3 Spezifikation des Extract Method-Refactorings

Nachdem die wichtigsten Merkmale der Spezifikationssprache GRAL sowie der Graphanfragesprache GReQL vorgestellt wurden, können diese zur Spezifikation eines Refactorings auf Metamodell-Ebene eingesetzt werden.

Ziel dieses Abschnitts ist es, die in Kapitel 5 vorgestellten Teilschritte zur Beschreibung des *Extract Method*-Refactorings formal durch Anwendung der Sprachen GRAL und GReQL zu spezifizieren. Die nötigen Graphtransformationen werden in Java-Pseudocode unter Verwendung der in der GRAL/GReQL-Bibliothek vorhandenen Funktionen zum Zugriff auf Graphen und deren Elemente formuliert. Wie bereits erwähnt, handelt es sich dabei um effiziente Operationen, d.h. ihr Ergebnis ist in polynomineller Zeit zu berechnen. Analog der einzelnen *Extract Method*-Schritte (siehe Abb. 5-1) werden diese nachfolgend in eine algorithmische Spezifikation überführt.

7.3.1 Gültigkeit des Methodenbezeichners in Compilation Unit

Zur Durchführung des *Extract Method*-Refactorings ist es nötig, die Gültigkeit des vom Benutzer gewählten Bezeichners für die neu zu erstellende Methode zu überprüfen. Zu diesem Zweck wird überprüft, ob im vorliegenden AST ein Methodendeklarations-Knoten existiert, dessen Bezeichner-Knoten ein Namens-Attribut enthält, welches mit dem gewählten Methodenbezeichner übereinstimmt. Folgende Spezifikation (siehe Abb.7-1) beschreibt diesen Sachverhalt anhand eines GRAL-Ausdrucks.

```
private boolean existsMethodNameInCU(methodName : string) {
    EXISTS td : TypeDeclaration, name : NAME @
           td -->{body} -->{identifier} name
           AND name.identifier = methodName
}
```

Abb. 7-1: existsMethodNameInCU-Spezifikation

Die Funktion `existsMethodNameInCU` überprüft, ob es eine Belegung gibt, die das angegebene GRAL-Pfadprädikat erfüllt. Dies ist gegeben, falls ausgehend vom einem `TypeDeclaration`-Knoten `td` ein Pfad, bestehend aus einer `body`- sowie einer folgenden `identifier`-Kante, existiert, welcher an einem `name`-Knoten endet, dessen `identifier`-Attribut dem gewählten Methodenbezeichner `methodName` entspricht. Kann eine solche Belegung im vorliegenden AST gefunden werden, ist der gewählte Methodenbezeichner ungültig und es muss eine Fehlerbehandlung eingeleitet werden, da das *Extract Method*-Refactoring in diesem Fall nicht durchgeführt werden kann.

7.3.2 Gültigkeit des Methodenbezeichners in Superklasse

Zur Durchführung des *Extract Method*-Refactorings ist zusätzlich zu überprüfen, ob das Erzeugen der neuen Methode eine bereits durch Vererbung vorhandene Methode mit gleichem Bezeichner überschreibt. Daher ist es nötig, die Methoden der Superklassen in der Vererbungshierarchie zu identifizieren und mit dem gewählten Bezeichner zu vergleichen. Die benötigten Informationen lassen sich aus dem AST durch Zugriff auf die enthaltenen Bindungsinformationen extrahieren. Entsprechend dem in Abb. 6-6 dargestellten PROGRES-Test kann das Vorgehen durch folgende Spezifikation (siehe Abb. 7-2) beschrieben werden:

```
private boolean existsMethodNameInSC(methodName : string) {
    EXISTS td : TypeDeclaration, dm : DeclaredMethods @
           td -->{extends} -->{binding} -->{superclass}* -->{methods} dm
           AND dm.name = methodName
}
```

Abb. 7-2: existsMethodNameInSC-Spezifikation

Die Funktion `existsMethodNameInSC` enthält ein GRAL-Pfadprädikat, dessen Erfüllbarkeit zu überprüfen ist. Es beschreibt die Existenz eines Pfades, welcher einen `TypeDeclaration`-Knoten `td` mit einem `DeclaredMethods`-Knoten `dm` verbindet. Ein solcher Pfad beginnt mit einer `extends`- gefolgt von einer `binding`-Kante. An diesem Punkt ist die Superklasse einer Compilation Unit erreicht. Über beliebig viele `superclass`-Kanten kann nun in der vorliegenden Vererbungshierarchie aufgestiegen werden. Letztendlich folgt eine `methods`-Kante, welche die Pfadbeschreibung abschließt und an einem Knotenelement `dm` endet. Dieses enthält eine Liste aller in der entsprechenden Superklasse vorhandenen Methoden, auf welche durch `dm.name` zugegriffen werden kann. Das Prädikat `dm.name = methodName` stellt somit ein Vergleich der gefundenen Methoden mit dem Bezeichner der neu zu erzeugenden Methode dar. Kann eine entsprechende Belegung im zugrundeliegenden AST gefunden werden, ist der Methodenbezeichner bereits in einer Klasse der Generalisierungshierarchie vorhanden. In diesem Fall muss wiederum eine Fehlerbehandlung eingeleitet werden und die Durchführung des *Extract Method*-Refactorings wird beendet.

7.3.3 Methodendeklaration erzeugen

Ein weiterer Teilschritt des *Extract Method*-Refactorings erfordert das Erzeugen der neuen Methodendeklaration. Dies wird selbstverständlich nur dann durchgeführt, wenn die Gültigkeit des gewählten Methodenbezeichners gewährleistet ist, was anhand der zuvor beschriebenen Tests überprüft wurde.

Das eigentliche Erzeugen einer Methode lässt sich durch folgendes Pattern (siehe Abb. 7-3) veranschaulichen. Es beschreibt auf der linken Seite denjenigen Teilgraphen des ASTs, an dessen Position die Transformation durchzuführen ist. Die rechte Seite hingegen stellt die Gestalt des ASTs nach Erzeugung einer Methodendeklaration dar.

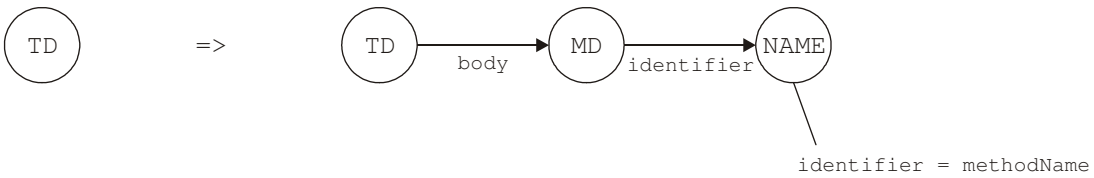


Abb. 7-3: createMethodDeclaration-Pattern

Der `TypeDeclaration`-Knoten `TD` der linken Patternseite enthält diejenige Methode, welche durch das *Extract Method*-Refactoring umstrukturiert werden soll. An dieser Stelle muss anhand von zusätzlichen Knotenelementen die Deklaration der neuen Methode erfolgen.

Die rechte Seite beschreibt die Gestalt des ASTs nach Hinzufügen der neuen Methodendeklaration. Der Knoten `TD` ist nun durch eine ausgehende `body`-Kante mit dem neuen `MethodDeclaration`-Knoten `MD` verbunden und eine `identifier`-Kante führt weiter zu einem `NAME`-Knoten, dessen `identifier`-Attribut mit dem Wert `methodName` belegt ist.

Die einzelnen Transformationsschritte zur Erzeugung der Kanten und Knoten lassen sich durch folgenden Pseudocode (siehe Abb. 7-4) gemäß der bereits vorgestellten PROGRES-Produktion (siehe Abb. 6-7) beschreiben.

```
private void createMethodDeclaration(startVertex : TypeDeclaration,
                                      methodName : string) {
[1]  assert scheme;
[2]  v = G.createVertex{MethodDeclaration}();
[3]  w = G.createVertex{NAME}();
[4]  G.createEdge{body}(startVertex, v);
[5]  G.createEdge{identifier}(v, w);
[6]  w.setAttribute(identifier, methodName);
[7]  assert scheme;
}
```

Abb. 7-4: createMethodDeclaration-Spezifikation

Der Methode `createMethodDeclaration` werden als Eingabeparameter der `TypeDeclaration`-Startknoten für die durchzuführenden Transformationen (`startVertex`) sowie der Bezeichner der neuen Methode (`methodName`) übergeben. Da eine Compilation Unit mehrere Typdeklarationen beinhalten kann (entsprechend der Anzahl der in einer Java-Datei deklarierten Klassen), muss zuvor derjenige `TypeDeclaration`-Knoten identifiziert werden, welcher den zu refaktorisierenden Programmcode umfasst. Dieser lässt sich bereits im Vorfeld durch eine AST-Traversierung ermitteln. Ein trivialer Fall liegt vor, wenn lediglich eine einzelne Typdeklaration vorhanden ist.

Der Rumpf der Methode `createMethodDeclaration` setzt sich zusammen aus einer Reihe von graphverändernden Operationen (`[2]` bis `[6]`), welche durch Zusicherungen (`[1]` und `[7]`) eingerahmt sind. So stellt `assert scheme` eine Zusicherung dar, welche besagt, dass die linke Patternseite vor Ausführung der Transformationen dem vorgegebenen AST-Schema entspricht.

Während der Ausführung der einzelnen Transformationsschritte befindet sich der Syntaxbaum in einem inkonsistenten Zustand, da seine Gestalt von der zugrundeliegenden AST-Schemadefinition abweicht. Erst nachdem alle Transformationen durchgeführt und der Programmablauf terminiert wurde, erreicht der AST wieder einen konsistenten Zustand und die abschließende Zusicherung, angedeutet durch `assert scheme`, ist erfüllt. Dies bedeutet, dass der Syntaxbaum nun wieder in einem dem AST-Schema entsprechenden konsistenten Zustand vorliegt und somit die Graphtransformation abgeschlossen ist. Die Assertions symbolisieren somit eine Klammerung für Transaktionen, wie sie aus dem Datenbanken-Bereich bekannt sind.

Die Transformation des ASTs besteht aus einzelnen Graphoperationen zur Erzeugung von Kanten und Knoten, welche durch das Präfix `G.` gekennzeichnet sind. So erzeugt

```
G.createVertex{nodetype}()
```

ein neues Knotenelement vom Typ *nodeType*. Anhand dieser Methode werden in `[2]` und `[3]` die neuen `MethodDeclaration`- sowie `NAME`-Knoten erstellt.

Die Erzeugung von gerichteten Kanten geschieht durch Aufruf der Graphoperation

```
G.createEdge{edgeType}(startVertex, endVertex)
```

Diese generiert eine neue Kante vom Typ *edgeType* zwischen dem Anfangsknoten *startVertex* und dem Zielknoten *endVertex*. So wird in `[4]` eine `body`-Kante zwischen dem Typdeklarations-Startknoten `startVertex` und dem Knoten `v` erstellt, welcher den zuvor erzeugten Methodendeklarations-Knoten enthält. Analog wird in `[5]` eine zusätzliche `identifier`-Kante zwischen den Knoten `v` und `w` generiert, welche die Verbindung zwischen der Methodendeklaration und dem zugehörigen Methodenbezeichner darstellt.

Die Wertzuweisung eines Attributs erfolgt anhand der Methode

```
nodeType.setAttribute(attribute, value)
```

Dabei wird dem Attribut `attribute` eines Knotens `nodeType` der Wert `value` zugeordnet. So kann in `[6]` dem `idenitifier`-Attribut des zuvor generierten `NAME`-Knotens, repräsentiert durch die Variable `w`, der neue Methodenbezeichner `methodName` zugewiesen werden. Damit ist die AST-Transformation abgeschlossen und der Graph befindet sich wieder in einem konsistenten Zustand.

7.3.4 Vorkommen lokaler Variablen

Das weitere Vorgehen zur Durchführung des *Extract Method*-Refactorings beinhaltet ein Überprüfen auf Vorkommen lokaler Variablen innerhalb der zu refaktorisierenden Codezeilen. Liegt ein solcher Fall vor, müssen diese lokalen Variablen in einem weiteren Refactoring-Teilschritt als Argumente an die neue Methode übergeben werden.

Die Überprüfung auf AST-Basis gestaltet sich folgendermaßen: Ausgehend von einem `MethodDeclaration`-Knoten muss eine `body`-Kante zu einem `Block`-Knoten existieren, welcher wiederum über eine ausgehende `statement`-Kante mit einem abstrakten `STATEMENT`-Knoten verbunden ist.

Ausgehend von diesen `STATEMENT`-Knoten existiert eventuell ein Pfad, welcher über verschiedene Kanten letztendlich an einem `NAME`-Knoten endet. Dieser ebenfalls abstrakte Knoten repräsentiert wiederum einen `SimpleName`- oder `QualifiedName`-Knoten. Kann von dem gefundenen `NAME`-Knoten eine ausgehende `binding`-Kante zu einem `IVariableBinding`-Knoten identifiziert werden, handelt es sich bei dem gefundenen Element um eine Variable. Ist das `declaringClass`-Attribut des Bindungsknoten zudem mit `null` belegt, wurde damit das Vorkommen einer lokalen Variable entdeckt.

Zur Veranschaulichung des soeben beschriebenen Vorgehens zeigt Abb. 7-5 nochmals auszugsweise den AST entsprechend dem *Extract Method*-Beispiel aus Kapitel 4. Lediglich derjenige AST-Teilbaum, welcher die zu verschiebenden Codezeilen enthält, ist inklusive seiner Kindknoten dargestellt. An den `STATEMENT`-Knoten, die die verschiedenen Statements wie `ExpressionsStatement` oder `WhileStatement` repräsentieren, ist durch die

Nummerierung ein Verweis auf die entsprechende Codezeile annotiert. Neben den farblich markierten `SimpleName`-Knoten, welche einen Bezeichner innerhalb eines Statements darstellen, sind auch die `VariableBinding`-Knoten[12] farblich gekennzeichnet. Diese enthalten die nötigen Informationen, um die Existenz einer lokalen Variable bestimmen zu können.

Das zuvor beschriebene Vorgehen zur Identifizierung einer lokalen Variable kann anhand des ASTs in Abb. 7-5 folgendermaßen verdeutlicht werden: Ausgehend von einem `MethodDeclaration`-Knoten führt eine `body`-Kante zu einem `Block`-Knoten. Dieser ist beispielsweise über eine `statement`-Kante mit dem `STATEMENT`-Knoten verbunden, welcher durch einen `ExpressionStatement`-Knoten (8) repräsentiert wird. Von hier führt ein Pfad bestehend aus `expression`-, `argument`- und `right_op`-Kante zu einem `NAME`-Knoten. Dieser ist vom Typ `SimpleName` und repräsentiert die Variable `outstanding`. Über eine `binding`-Kante führt ein Weg zu dem zugehörigen `VariableBinding`-Knoten, dessen `declaringClass`-Attribut den Wert `null` enthält. Somit ist die Variable `outstanding` als lokal vorkommend identifiziert.

Die gerade erläuterte Methode am Beispiel eines Expression Statements lässt sich analog auf alle anderen Statement-Elemente übertragen. Lediglich der zu wählende Pfad zwischen `STATEMENT`- und `NAME`-Knoten ist unterschiedlich. Ein Problem bereitet allerdings das Erkennen einer Variablen, da diese ebenso wie beispielsweise ein Methodenbezeichner durch `SimpleName`-Knoten repräsentiert werden. Daher müssen in einem AST, ausgehend von einem markierten `STATEMENT`-Knoten, sämtliche `SimpleName`-Elemente identifiziert und die enthaltenen Bindungsinformationen überprüft werden. Bei einer späteren Implementierung lässt sich dieses Problem effizient anhand eines Visitors lösen: Dieser traversiert alle `SimpleName`-Knoten und ermittelt diejenigen Elemente, deren Variablen-Bindungsinformation auf eine lokale Deklaration hindeutet.

[12] Anmerkung: In einem AST werden Bindings nicht durch Knoten repräsentiert. Der Zugriff erfolgt über Methodenaufrufe. Die Knotendarstellung dient daher lediglich zur vereinfachten Darstellung.

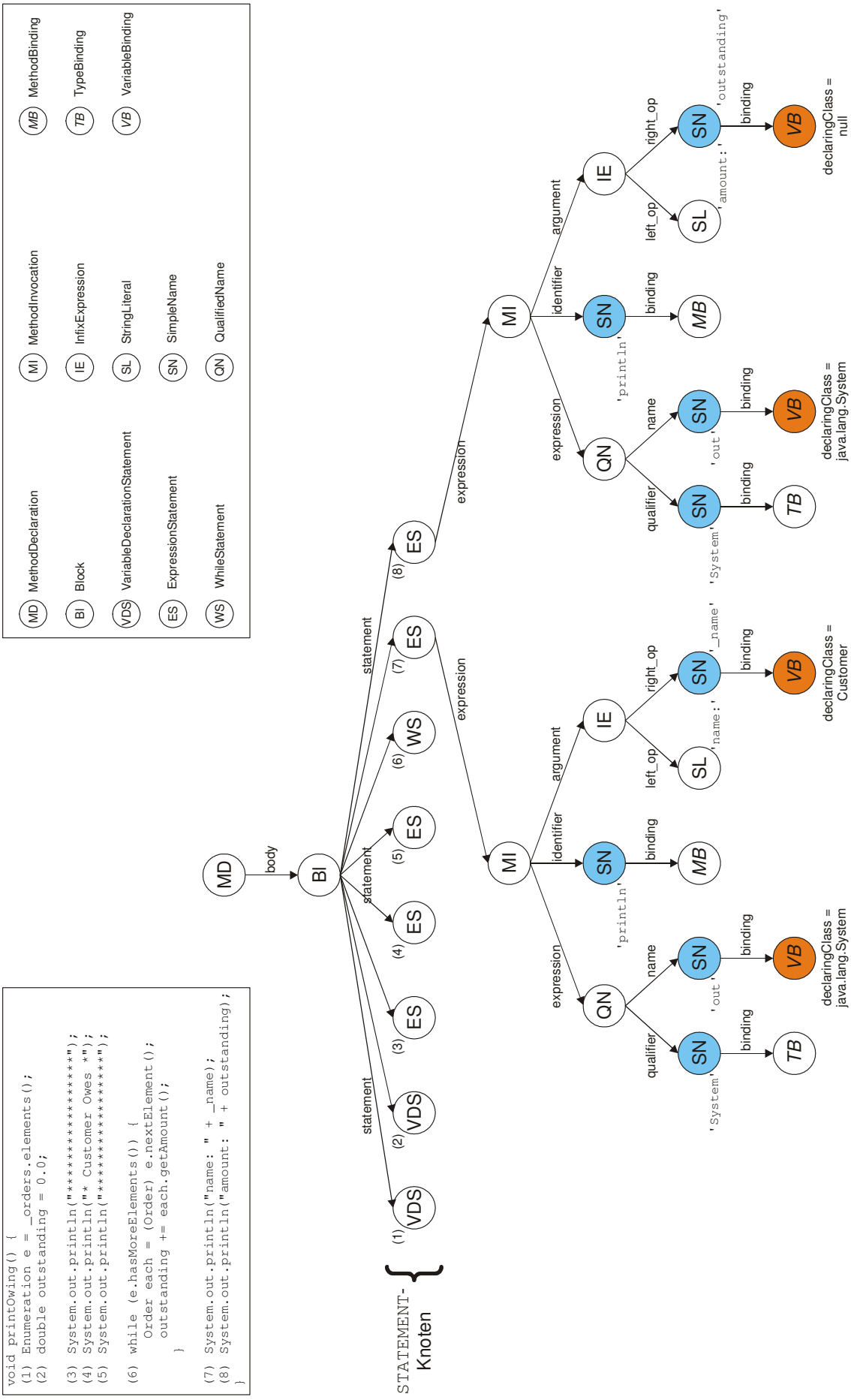


Abb. 7-5: AST gemäß Extract Method-Beispiel

Da es sich bei der Identifizierung einer lokalen Variable um eine Graphanfrage mit anschließender Rückgabe eines entsprechenden Elements handelt, kann dies durch folgende GReQL-Anfrage (siehe Abb. 7-6) spezifiziert werden.

```
private SEQ(V{NAME}) existsLocalVar(marking : SEQ(V{STATEMENT})) {
    USING marking:
      FROM md : V{MethodDeclaration}, name : V{NAME}
      WITH
        EXISTS vb : V{VariableBinding}, m : marking @
            md (-->{body}) (-->{statement}) m (-->*) name (-->{binding}) vb
            AND vb.declaringClass = 'null'
      REPORT name END
}
```

Abb. 7-6: existsLocalVar-Spezifikation

Die Funktion `existsLocalVar` erhält als Eingabeparameter eine Sequenz von `STATEMENT`-Knotentypen, welche die markierten Codezeilen beinhaltet und anhand der `USING`-Klausel für den nachfolgenden FWR-Ausdruck zur Verfügung gestellt wird. Die Deklaration der Variablen `md` und `name`, welche die Methodendeklaration sowie den Variablenbezeichner repräsentieren, erfolgt im `FROM`-Bereich des FWR-Ausdrucks. Anschließend wird im `WITH`-Abschnitt nach einer Belegung für das dort angegebene Prädikat im vorliegenden AST gesucht. Dieses Prädikat drückt das zuvor skizzierte Vorgehen zur Suche nach einer lokalen Variablen aus. Kann eine Prädikat-Belegung gefunden werden, wird das entsprechende Element im `REPORT`-Bereich der Variable `name` zugewiesen. Der Rückgabeparameter der Funktion `existsLocalVar` ist daher eine Sequenz von `NAME`-Knotentypen, da mehrere lokale Variablen in den markierten Statements vorkommen können. Damit endet die Spezifikation dieses weiteren *Extract Method*-Teilschritts.

7.3.5 Methodenparameter erzeugen

Kommen innerhalb der zu refaktorisierenden Codezeilen lokale Variablen vor, müssen diese an die neue Methode als Argumente übergeben werden. Daher ist es nötig, die zuvor erzeugte Methodendeklaration um die entsprechenden Parameter zu erweitern. Das folgende Pattern (siehe Abb. 7-7) veranschaulicht die zu diesem Zweck durchzuführende Graphtransformation.

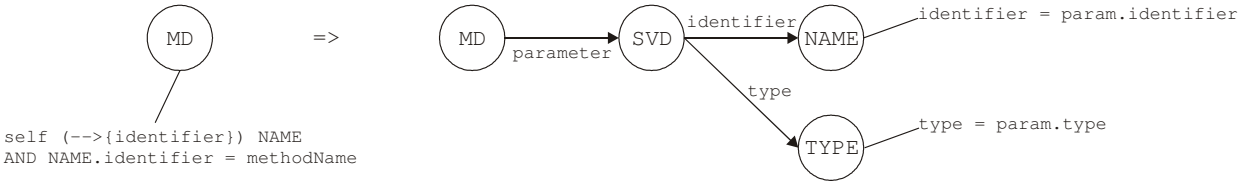


Abb. 7-7: createMethodParam-Pattern

Die linke Patternseite beschreibt denjenigen `MethodDeclaration`-Knoten, welcher die zuvor erzeugte neue Methode mit Bezeichner `methodName` repräsentiert. Dies wird ersichtlich durch die annotierte Restriktion in Form eines GRAL-Ausdrucks. Die rechte Seite hingegen beschreibt die Gestalt des ASTs nach durchgeführter Graphtransformation. Hier existiert nun zusätzlich eine `parameter`-Kante, welche den `MethodDeclaration`-Knoten mit einem neuen `SingleVariableDeclaration`-Knoten verbindet. Dieser repräsentiert einen Parameter der Methodendeklaration und ist weiterhin über eine `identifier`-Kante mit einem `NAME`-Knoten sowie einer `type`-Kante mit einem `TYPE`-Knoten verbunden. Diese beiden Knoten stellen den Parameterbezeichner sowie -typ dar und ihre Attribute sind entsprechend dem einzufügenden Parameter belegt.

Die einzelnen Schritte zur Erzeugung der Kanten und Knoten sind in Abb. 7-8 spezifiziert und entsprechen der im vorherigen Kapitel erstellten PROGRES-Produktion (siehe Abb. 6-10).

```
private void createMethodParam(startVertex : MethodDeclaration,
                               param : SEQ(V{NAME})) {
    assert scheme;
    for (int i = 0; i < param.size(); i++) {
        u = G.createVertex{SingleVariableDeclaration}();
        G.createEdge{parameter}(startVertex, u);
        v = G.createVertex{TYPE}();
        w = G.createVertex{NAME}();
        G.createEdge{type}(u, v);
        G.createEdge{identifier}(u, w);
        v.setAttribute(type, param.elementAt(i).type);
        w.setAttribute(identifier, param.elementAt(i).identifier);
    }
    assert scheme;
}
```

Abb. 7-8: createMethodParam-Spezifikation

Die Graphtransformation `createMethodParam` benötigt als Eingabeparameter den `MethodDeclaration`-Knoten `startVertex`, der die zuvor erzeugte neue Methodendeklaration repräsentiert und zusätzlich eine `param`-Sequenz, welche die zuvor identifizierten lokalen Variablen in Form von `NAME`-Knoten enthält. Anhand dieser übergebenen Parameter kann der Refactoring-Teilschritt durchgeführt werden. Da die in der Spezifikation vorkommenden Elemente bereits zuvor ausführlich erläutert wurden, wird auf eine erneute Erklärung verzichtet und dieser Refactoring-Teilschritt als abgeschlossen betrachtet.

7.3.6 Methodenaufruf erzeugen

In einem nächsten *Extract Method*-Teilschritt wird der Aufruf der zuvor neu erzeugten Methode generiert. Dies beinhaltet das Identifizieren der markierten Statements, d.h. derjenigen Codezeilen, welche später in die neue Methode ausgelagert werden. An dieser Position muss der Aufruf der zuvor neu erzeugten Methoden erfolgen. Die Gestalt des ASTs vor und nach dieser Graphtransformation ist in nachfolgendem Pattern (siehe Abb. 7-9) dargestellt. Auf der rechten Schemaseite ist zu erkennen, dass der Aufruf der neuen Methode vor den markierten Statements eingefügt wurde. Da GRAL/GReQL auf angeordneten Graphen basiert, bereitet diese Beschreibung im Gegensatz zu PROGRES keine Probleme.

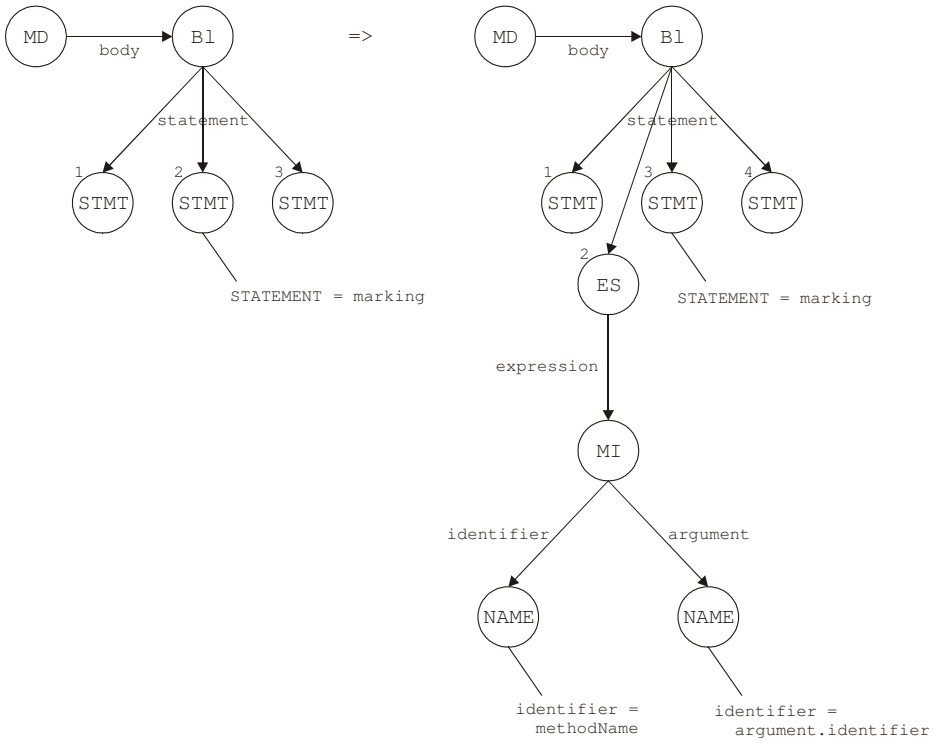


Abb. 7-9: createMethodCall-Pattern

Die Spezifikation der im vorherigen Schema beschriebenen Graphtransformation ist nachfolgend (siehe Abb. 7-10) dargestellt und bezieht sich auf die entsprechende PROGRES-Produktion (siehe Abb. 6-11) aus dem vorherigen Kapitel.

```
private void createMethodCall(startVertex : MethodDeclaration,
                              marking : SEQ(V{STATEMENT}),
                              methodName : string, argument : SEQ(V{NAME})) {
 [1]  assert scheme;
 [2]  u = omega(startVertex.firstOut{body}());
 [3]  v = G.createVertex{ExpressionStatement}();
 [4]  e = G.createEdge{statement}(u, v);
 [5]  f = G.edgeFromTo(u, marking.firstElement());
 [6]  u.putBefore(pos(f, G.allOutEdges(u)), e);
 [7]  w = G.createVertex{MethodInvocation}();
 [8]  x = G.createVertex{NAME}();
 [9]  G.createEdge{expression}(v, w);
[10]  G.createEdge{identifier}(w, x);
[11]  x.setAttribute(identifier, methodName);
[12]  for (int i = 0; i < argument.size(); i++) {
[13]      y = G.createVertex{NAME}();
[14]      G.createEdge{argument}(w, y);
[15]      y.setAttribute(identifier, argument.elementAt(i).identifier);
[16]  }
[17]  assert scheme;
}
```

Abb. 7-10: createMethodCall-Spezifikation

Die Methode `createMethodCall` erhält als `startVertex`-Parameter denjenigen `MethodDeclaration`-Knoten, welcher die zu verschiebenden Codezeilen beinhaltet. Diese wiederum werden anhand der `marking`-Sequenz der Methode übergeben. Weitere Eingabeparameter sind `methodName`, stellvertretend für den neuen Methodenbezeichner sowie eine `argument`-Sequenz, welche eventuell zuvor ermittelte lokale Variablen enthält und dementsprechend als Argumente an die neue Methode weitergeleitet werden.

Der Ablauf ist erneut durch Zusicherungen (`[1]` und `[17]`) eingerahmt, deren Funktion bereits zuvor erläutert wurde. In `[2]` wird ausgehend von `startVertex`, welcher die Methodendeklaration repräsentiert, der zugehörige Rumpf in Form eines `Block`-Knotens ermittelt. Durch `startVertex.firstOut{body}()` wird die einzige vorhandene ausgehende

`body`-Kante ermittelt, per `omega()` der durch diese `body`-Kante verbundene `Block`-Knoten zurückgegeben und dieser sodann in der Variable `u` hinterlegt. In `[3]` erfolgt das Erzeugen eines `ExpressionStatement`-Knotens, welcher durch eine `statement`-Kante `[4]` mit dem soeben ermittelten `Block`-Knoten `u` verbunden wird.

In den folgenden beiden Zeilen wird der gerade erzeugten `statement`-Kante die korrekte Position innerhalb der Methodendeklaration zugeordnet, damit der einzufügende Methodenaufruf an passender Stelle im Programmcode erscheint. Zu diesem Zweck wird in `[5]` diejenige `statement`-Kante ermittelt, welche ausgehend vom `MethodDeclaration`-Knoten eine Verbindung zu dem ersten markierten `STATEMENT`-Knoten herstellt. Die durch den Aufruf von `G.edgeFromTo(startVertex, marking[0])` identifizierte Kante wird sodann in der Variablen `f` gespeichert. In `[6]` wird durch `G.allOutEdges(startVertex)` eine Liste aller aus dem `MethodDeclaration`-Knoten ausgehenden Kanten zurückgeliefert. Daraufhin wird durch `pos(f, G.allOutEdges(startVertex))` die Position der zuvor ermittelten `statement`-Kante `f` in dieser Liste bestimmt. Der Aufruf der Funktion *`node`*`.putBefore(`*`position`*`, `*`edge`*`)` bewirkt ein Positionieren der Kante *`edge`* vor der angegebenen *`position`*-Markierung in der Kantenliste des Knotens *`node`*.

Damit ist in `[6]` die Lage des Methodenaufrufs innerhalb des ASTs festgelegt und in `[7]` bis `[10]` werden die weiteren benötigten Knoten und Kanten erzeugt sowie der entsprechende Methodenbezeichner für den Aufruf durch `methodName` in `[11]` zugewiesen. In `[12]` bis `[16]` erfolgt gemäß der Länge der `argument`-Sequenz ein Erzeugen von Knoten und Kanten als Argumente des Methodenaufrufs. Hiermit endet die Graphtransformation und ein weiterer *Extract Method*-Schritt ist abgeschlossen.

7.3.7 Markierte Codezeilen verschieben

Ziel des letzten *Extract Method*-Teilschritts ist das Verschieben der markierten Codezeilen in die neue Methode. Zu diesem Zweck müssen im vorliegenden AST die markierten `STATEMENT`-Knoten sowie die zugehörigen eingehenden `statement`-Kanten in den Rumpf der neuen Methode verschoben werden. Dazu ist es lediglich notwendig, den eingehenden `statement`-Kanten einen neuen Anfangsknoten zuzuweisen. Die Gestalt des ASTs vor und nach dieser durchzuführenden Graphtransformation ist in folgendem Pattern (siehe Abb. 7-11) dargestellt und entspricht der im letzten Kapitel erstellten PROGRES-Produktion (siehe Abb. 6-12).

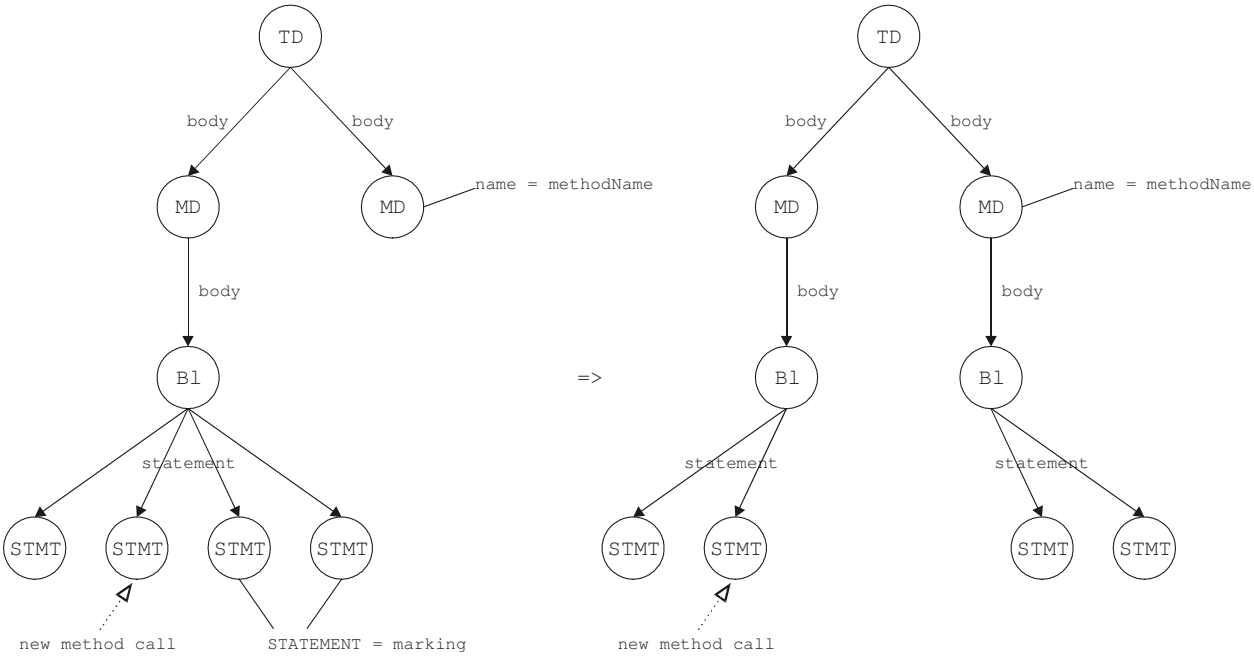


Abb. 7-11: moveMarkedCode-Pattern

Die dem Pattern entsprechende Spezifikation ist nachfolgend (siehe Abb. 7-12) aufgeführt. Als Eingabeparameter benötigt diese Graphtransformation wiederum die Angabe der markierten `STATEMENT`-Knoten in Form der bereits bekannten `marking`-Sequenz. Zusätzlich wird derjenige `MethodDeclaration`-Knoten als `startVertex` übergeben, welcher die zuvor neu erzeugte Methodendeklaration repräsentiert und während der durchzuführenden Codeverschiebung um weitere Knoten- und Kantenelemente erweitert wird.

```
private void moveMarkedCode(startVertex : MethodDeclaration,
                            marking : SEQ(V{STATEMENT})) {
[1]  assert scheme;
[2]  u = G.createVertex{Block}();
[3]  G.createEdge{body}(startVertex, u);
[4]  for (int i = 0; i < marking.size(); i++) {
[5]      e = marking.elementAt(i).firstIn{statement}();
[6]      e.alpha() = u;
[7]  }
[8]  assert scheme;
}
```

Abb. 7-12: moveMarkedCode-Spezifikation

Der Rumpf der Methode `moveMarkedCode` ist wiederum durch die Assertions `[1]` und `[8]` eingerahmt. In `[2]` wird ein neuer `Block`-Knoten erzeugt, welcher den Rumpf der zuvor neu erzeugten Methode darstellt, und in `[3]` durch eine `body`-Kante mit der Methodendeklaration verbunden. `[4]` enthält eine Iteration über die zu verschiebenden Codezeilen, welche durch die Elemente der `marking`-Sequenz repräsentiert werden. Für jedes Sequenz-Element wird in `[5]` die einzige eingehende `statement`-Kante ermittelt und in `[6]` der zuvor erzeugte `Block`-Knoten als Kantenanfangsknoten zugewiesen.

Damit endet die letzte Graphtransformation und es wurden in der neuen Methodendeklaration diejenigen Elemente eingefügt, welche in einem AST die zu verschiebenden Codezeilen repräsentieren. Zugleich wurden die entsprechenden Elemente aus der ursprünglichen Methode entfernt. Zusammenfassend sind nun alle *Extract Method*-Teilschritte spezifiziert. Doch bevor der Pseudocode zur Ausführung des Refactorings vorgestellt wird, folgt die Deklaration einiger Hilfsfunktionen.

7.3.8 Hilfsfunktionen

Zur Vereinfachung der Spezifikation der einzelnen Refactoring-Teilschritte wurde den Methoden, welche die Graphtransformationen beschreiben, jeweils ein entsprechender `startVertex`-Parameter übergeben. So erwartet `createMethodDeclaration` einen entsprechenden `TypeDeclaration`-Knoten, `createMethodParam` und `moveMarkedCode` denjenigen `MethodDeclaration`-Knoten, welcher die neue Methodendeklaration repräsentiert und `createMethodCall` den `MethodDeclaration`-Knoten, der die zu verschiebenden Codezeilen beinhaltet. Diese verschiedenen Startknoten können anhand folgender Hilfsfunktionen aus den dem *Extract Method*-Refactoring zur Verfügung stehenden Informationen, genauer gesagt den markierten Codezeilen (`marking`-Sequenz) und dem Bezeichner der neuen Methode (`methodName`), ermittelt werden.

Damit die Graphtraversierungen der einzelnen Hilfsfunktionen nachzuvollziehen sind, zeigt Abb. 7-13 eine schematische Darstellung des ASTs, welcher die Basis zur Durchführung des *Extract Method*-Refactorings bildet. Hier sind ebenfalls die markierten Codezeilen (`marking`) und der gewählte Methodenbezeichner (`methodName`) dargestellt.

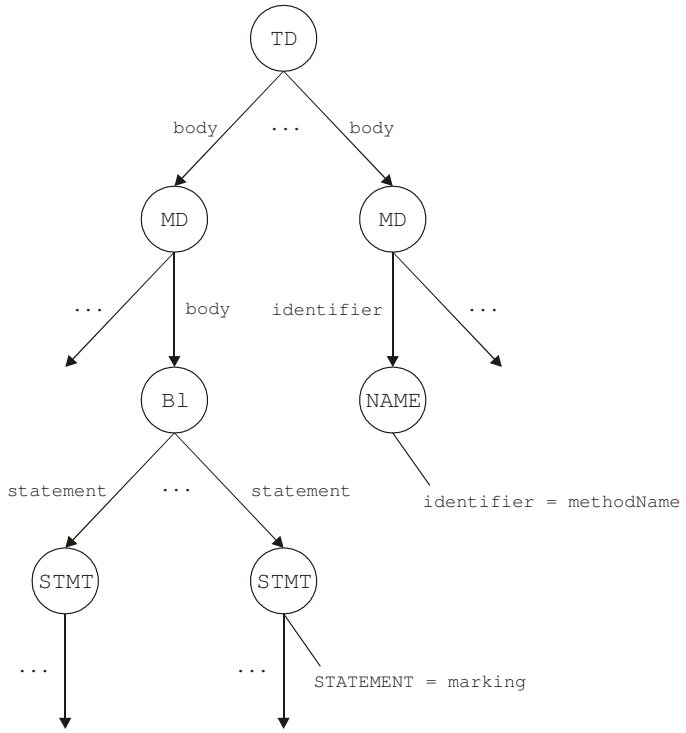


Abb. 7-13: AST-Schema

Hilfsfunktion 'getTargetMD'

Die Funktion `getTargetMD` (siehe Abb. 7-14) ermittelt den `MethodDeclaration`-Knoten, welcher die neue Methode mit Bezeichner `methodName` repräsentiert und berechnet somit den `startVertex`-Parameter für die Graphtransformationen `createMethodParam` und `moveMarkedCode`.

```
private MethodDeclaration getTargetMD(methodName : string) {
 [1]  for all edges e1 of Type body do begin
 [2]      v1 = e1.omega();
 [3]      if v1.hasType(MethodDeclaration) then begin
 [4]          for all outEdges e2 of v1 of Type identifier do begin
 [5]             v2 = e2.omega();
 [6]             if v2.identifier = methodName then
 [7]               return v1;
 [8]          end;
 [9]      end;
[10]  end;
}
```

Abb. 7-14: getTargetMD-Hilfsfunktion

Die Hilfsfunktion `getTargetMD` traversiert in `[1]` schrittweise alle `body`-Kanten und speichert in `[2]` den Kantenendknoten in der Variable `v1`. Entspricht in `[3]` dieser Endknoten einem Knoten vom Typ `MethodDeclaration`, werden in `[4]` alle von diesem ausgehenden `Identifier`-Kanten betrachtet. Der Endknoten einer `identifier`-Kante wird in `[5]` wiederum gespeichert und in `[6]` dessen `identifier`-Attribut mit dem Eingabeparameter `methodName` verglichen. Stimmen die Werte überein, wurde ein dem gesuchten Methodenbezeichner entsprechender `NAME`-Knoten identifiziert. Der zugehörige `MethodDeclaration`-Knoten ist noch in der Variable `v1` zwischengespeichert und liefert in `[7]` somit die Rückgabe der Funktion.

Hilfsfunktion 'getSourceMD'

Die Funktion *getSourceMD* (siehe Abb. 7-15) ermittelt anhand der `marking`-Sequenz denjenigen `MethodDeclaration`-Knoten, dessen Rumpf die zu verschiebenden Codezeilen in Form von `STATEMENT`-Knoten enthält. Das Ergebnis liefert somit den `startVertex`-Eingabeparameter der Graphtransformation `createMethodCall`.

```
private MethodDeclaration getSourceMD(marking : SEQ(V{STATEMENT})) {
[1]  e1 = marking.firstElement().firstIn{statement}();
[2]  v1 = e1.alpha();
[3]  e2 = v1.firstIn{body}();
[4]  v2 = e2.alpha();
[5]  return v2;
}
```

Abb. 7-15: getSourceMD-Hilfsfunktion

Die Hilfsfunktion *getSourceMD* extrahiert in `[1]` die (einzige) eingehende `statement`-Kante des ersten `STATEMENT`-Elements der `marking`-Sequenz. Anschließend wird in `[2]` der Startknoten dieser Kante angefordert, welcher nach Abb. 7-13 einen `Block`-Knoten darstellt. Ausgehend von diesem Knoten wird in `[3]` die (einzige) eingehende `body`-Kante ermittelt, in `[4]` der entsprechende Anfangsknoten vom Typ `MethodDeclaration` gespeichert und in `[5]` als Rückgabe der Funktion übergeben.

Hilfsfunktion 'getSourceTD'

Die Funktion *getSourceTD* liefert anhand der `marking`-Sequenz denjenigen `TypeDeclaration`-Knoten, welcher die Methode mit den zu verschiebenden Programmzeilen enthält und unterhalb dessen die neue Methode zu erzeugen ist. Das Ergebnis entspricht zugleich dem `startVertex`-Eingabeparameter der Graphtransformation *createMethodDeclaration*.

```
private TypeDeclaration getSourceTD(marking : SEQ(V{STATEMENT})) {
[1]  v1 = getSourceMD(marking);
[2]  e1 = v1.firstIn{body}();
[3]  v2 = e1.alpha();
[4]  return v2;
}
```

Abb. 7-16: getSourceTD-Hilfsfunktion

Wie in Abb. 7-13 zu erkennen ist, kann der benötigte `TypeDeclaration`-Knoten beispielsweise durch eine Rückwärtstraversierung der `body`-Kante ermittelt werden, die diejenige `MethodDeclaration` anbindet, welche die zu verschiebenden Codezeilen beinhaltet. Da dieser `MethodDeclaration`-Knoten bereits durch die Hilfsfunktion *getSourceMD* berechnet wird, erfolgt in `[1]` vorab ein entsprechender Aufruf. Anhand des zurückgegebenen Knotens kann in `[2]` auf die dort eingehende `body`-Kante zugegriffen werden. Der Anfangsknoten der `body`-Kante entspricht dem gesuchten `TypeDeclaration`-Knoten (`[3]`), welcher sodann in `[4]` den Rückgabewert der Funktion bildet.

7.3.9 Extract Method-Transaktion

Auf Basis der zuvor spezifizierten *Extract Method*-Teilschritte und den erläuterten Hilfsfunktionen kann abschließend das *Extract Method*-Refactoring in Pseudocode (siehe Abb. 7-17) notiert werden.

Die Ausführungsreihenfolge der Teilschritte des *Extract Method*-Refactorings erfolgt gemäß dem in Kapitel 5 vorgestellten Ablaufdiagramm (siehe Abb. 5-1) und unterscheidet sich von der mittels PROGRES erstellten *Extract Method*-Transaktion (siehe Abb. 6-12) lediglich durch die Aufrufe der Hilfsfunktionen. Daher wird an dieser Stelle auf eine ausführliche

Beschreibung verzichtet und der Programmablauf aufgrund der bereits erfolgten Erklärung der einzelnen Methoden und Funktionen nicht weiter erläutert.

```
public void extractMethodRefactoring(marking : SEQ(V{STATEMENT}),
                                      methodName : string) {
    localVar = SEQ(V{NAME});
    if existsMethodNameInCU(methodName) then error;
    else begin
        if existsMethodNameInSC(methodName) then error;
        else begin
            sourceTD = getSourceTD(marking);
            createMethodDeclaration(sourceTD, methodName);
            localVar = existLocalVar(marking);
            if (localVar.size() <> 0) then begin
                targetMD = getTargetMD(methodName);
                createMethodParam(targetMD, localVar);
            end;
            sourceMD = getSourceMD(marking);
            createMethodCall(sourceMD, marking, methodName, localVar);
            moveMarkedCode(targetMD, marking);
        end;
    end;
}
```

Abb. 7-17: Extract Method-Transaktion in Pseudocode

Die Anwendung der *Extract Method*-Transaktion auf einen AST transformiert diesen gemäß dem beschriebenen Vorgehen des *Extract Method*-Refactorings. Ein auf diese Weise modifizierter AST enthält offensichtlich ebenfalls die benötigten Informationen, um den refaktorisierten Sourcecode herleiten zu können.

Eine konkrete Anwendung des *Extract Method*-Refactorings auf AST-Basis und somit die Umsetzung der erstellten *Extract Method*-Spezifikation wird abschließend durch das im folgenden Kapitel beschriebene *ASTRefactor* Plug-In demonstriert.

Kapitel 8
Implementation des ASTRefactor Plug-Ins

Nachdem in den vorangegangenen Kapiteln die Durchführung des *Extract Method*-Refactorings auf AST-Basis schrittweise erläutert und spezifiziert wurde, können die Ergebnisse nun in Form eines Eclipse Plug-Ins implementiert werden.

Das vorliegende Kapitel beginnt mit einer Anforderungsbeschreibung an das zu entwickelnde Plug-In. Danach folgt eine Erläuterung der vorzunehmenden Erweiterungen zur Integration des Plug-Ins in die Eclipse-Umgebung. Nach einer Vorstellung der entwickelten Klassen wird die Kernfunktionalität des Plug-Ins zur Transformation des ASTs sowie das Übertragen dieser Manipulation auf den zugrundeliegenden Sourcecode erläutert. Weiterhin folgt eine Ablaufbeschreibung des *Extract Method*-Refactorings sowie abschließend eine Darstellung der Anwendung des *ASTRefactor* Plug-Ins.

8.1 Anforderungen

Der Aufruf des zu entwickelnden *ASTRefactor* Plug-Ins soll – analog der in Eclipse bereits enthaltenen Refactoring-Unterstützung – über verschiedene Menüs ermöglicht werden: Einerseits anhand eines zusätzlichen Menüs in der Menüleiste der Eclipse-Oberfläche, andererseits mittels eines weiteren Eintrags im Kontextmenü des Java-Editors.

Auf Basis der seitens eines Benutzers markierten Codezeilen wird sodann das *Extract Method*-Refactoring auf AST-Ebene durchgeführt. Zu diesem Zweck wird in einem ersten Dialogfenster der ursprüngliche Sourcecode angezeigt sowie die Eingabe eines neuen Methodenbezeichners ermöglicht. Daraufhin folgt die Ausführung des Refactorings und das Dialogfenster wird um eine Vorschau des geänderten Sourcecodes erweitert. Bestätigt der Benutzer diese Transformation, werden die Änderungen auf den zugrundeliegenden Sourcecode übertragen.

Weiterhin soll das Plug-In die Möglichkeit bieten, den AST in textueller Gestalt in einem separaten Fenster anzuzeigen. Da der Syntaxbaum zur Durchführung des Refactorings benötigt wird und daher ohnehin erzeugt werden muss, bietet sich diese zusätzliche Funktionalität geradezu an. Der Aufruf soll ebenfalls über die Menüs an den zuvor genannten Positionen möglich sein, zusätzlich aber auch anhand eines Eintrags im Kontextmenü für eine Java-Sourcedatei, welche beispielsweise im Package Explorer aufgeführt ist.

8.2 Plug-In Manifest

Die Datei `plugin.xml` (siehe Anhang C) beschreibt die Einbettung des *ASTRefactor* Plug-Ins in die Eclipse-Umgebung. Entsprechend den Anforderungen stellt das zu entwickelnde Plug-In ein Hauptmenü 'ASTRefactor' zur Verfügung, welches nach Bedarf eine Kombination der Untermenüpunkte 'Show AST' und 'Extract Method' enthält.

8.2.1 Erweiterung des Java Editor-Kontextmenüs

Das *ASTRefactor* Plug-In erzeugt einen zusätzlichen Eintrag im Kontextmenü des Java-Editors, sodass zu einer im Editor angezeigten Java-Datei einerseits der zugehörige AST angezeigt sowie andererseits das *Extract Method*-Refactoring aufgerufen werden kann. Den entsprechenden Abschnitt innerhalb der Manifest-Datei `plugin.xml` zur Definition dieser Erweiterung zeigt folgende Abb. 8-1.

```
...
<extension
   point="org.eclipse.ui.popupMenus">
   <viewerContribution
      ...
      targetID="#CompilationUnitEditorContext">
      <menu
         ...
         label="ASTRefactor">
      </menu>
      <action
         label="Show AST"
         ...
         class="de.hinterwaeller.astrefactor.ASTView"/>
      <action
         label="Extract Method"
         ...
         class="de.hinterwaeller.astrefactor.ExtractMethodRefactoring"/>
   </viewerContribution>
</extension>
...
```

Abb. 8-1: Erweiterung des Editor-Kontextmenüs

Anhand des Erweiterungspunkts `org.eclipse.ui.popupMenus`[13] können zusätzliche Aktionen in bereits existierende Kontextmenüs integriert werden. Unter Verwendung einer `viewerContribution` ist es insbesondere möglich, die Kontextmenüs von Views und Editoren der Eclipse-Umgebung zu erweitern. Die Angabe der `targetID` referenziert dabei das entsprechende Element, wobei der Zugriff auf das Editor-Kontextmenü beispielsweise durch die Bezeichnung `#CompilationUnitEditorContext` erfolgt.

Das Erstellen des neuen Menüs findet im `menu`-Abschnitt statt, in dem unter anderem der darzustellende Menütext durch die Angabe von `label="ASTRefactor"` festgelegt wird. Die Menüelemente 'Show AST' und 'Extract Method' werden in den nachfolgenden `action`-Bereichen definiert. Zusätzlich zur Festlegung des Menütextes durch das `label`-Attribut erfolgt anhand des `class`-Attributs die Zuweisung derjenigen Klasse, welche bei Eintreten einer Menüaktion aufgerufen wird.

8.2.2 Erweiterung des Kontextmenüs für ICompilationUnit-Elemente

Weiterhin wird ein Menü-Eintrag erstellt, welcher lediglich innerhalb des Kontextmenüs einer `ICompilationUnit` erscheint und somit das Anzeigen des zu einer Java-Sourcedatei zugehörigen ASTs ermöglicht. Den entsprechenden Abschnitt zur Erzeugung eines solchen Kontextmenü-Elements ist in Abb. 8-2 dargestellt.

```
...
<extension
   point="org.eclipse.ui.popupMenus">
   <objectContribution
      ...
      objectClass="org.eclipse.jdt.core.ICompilationUnit">
      <menu
         ...
         label="ASTRefactor">
      </menu>
      <action
         label="Show AST"
         ...
         class="de.hinterwaeller.astrefactor.ASTView"/>
   </objectContribution>
</extension>
...
```

Abb. 8-2: Erweiterung des Kontextmenüs für ICompilationUnit-Elemente

[13] Dokumentation zu finden unter `Platform Plug-In Developer Guide` → `Reference` → `Extension Points Reference` → `org.eclipse.ui.popupMenus` des Eclipse-Hilfesystems.

Zu diesem Zweck kann ebenfalls der Erweiterungspunkt `org.eclipse.ui.popupMenus` genutzt werden. Jedoch wird jetzt eine `objectContribution` erzeugt, welche auf den durch `objectClass` referenzierten Objekttyp reagiert und einen Kontextmenü-Eintrag lediglich für Elemente dieses Typs anzeigt. Da eine Java-Source-Datei im Eclipse-Kontext durch eine `ICompilationUnit`[14] repräsentiert wird, bietet sich die Angabe dieses Referenzobjekts geradezu an. Weiterhin folgt die Definition des Menüs 'ASTRefactor' sowie des Menüelements 'Show AST' unter Bekanntgabe der implementierenden Klasse für diese Menüaktion. Damit ist es nun möglich, die Visualisierung des ASTs zu einer beispielsweise im Package Explorer der Eclipse-Umgebung angezeigten Java-Datei aufzurufen.

8.2.3 Erweiterung der Eclipse-Menüleiste

Das Erstellen des 'ASTRefactor'-Menüs in der Menüleiste der Eclipse-Oberfläche erfolgt durch den Erweiterungspunkt `org.eclipse.ui.actionSets` (siehe Abb. 8-3). Da die Verwendung dieses Erweiterungspunkts bereits ausführlich in Kapitel 2.7 erläutert wurde, wird an dieser Stelle auf eine weitere Ausführung verzichtet.

```
...
<extension
   point="org.eclipse.ui.actionSets">
   <actionSet
      ...
      <menu
         ...
         label="ASTRefactor">
      </menu>
      <action
         label="Show AST"
         ...
         class="de.hinterwaeller.astrefactor.ASTView"/>
      <action
         label="Extract Method"
         ...
         class="de.hinterwaeller.astrefactor.ExtractMethodRefactoring"/>
   </actionSet>
</extension>
...
```

Abb. 8-3: Erweiterung der Workbench-Menüleiste

[14] Dokumentation zu finden unter `JDT Plug-In Developer Guide` → `Reference` → `API Reference` → `org.eclipse.jdt.core` → `ICompilationUnit` des Eclipse-Hilfesystems.

8.3 Klassen des Plug-Ins

Eine Übersicht derjenigen Klassen, welche die Funktionalität des *ASTRefactor* Plug-Ins implementieren, bietet das folgende Klassendiagramm (siehe Abb. 8-4). Sie sind allesamt in dem Paket `de.hinterwaeller.astrefactor` enthalten und werden nachfolgend kurz erläutert. Eine Auflistung der jeweils zugehörigen Variablen und Methoden bietet die im Anhang D enthaltene Übersicht der einzelnen Klassen. Ein zusätzlicher Blick in den entsprechenden Sourcecode liefert weitere Details bezüglich der Implementierung.

Die Klasse `ASTCreator` ist für das Erzeugen des ASTs zu einer vorliegenden Java-Sourcedatei zuständig. Die visuelle Darstellung dieses ASTs erfolgt hauptsächlich anhand der Klassen `ASTView` und `ASTViewVisitor`. Für die Hauptfunktionalität des Plug-Ins, nämlich die Durchführung des *Extract Method*-Refactoring auf Metamodell-Ebene, sind insbesondere die Klassen `ExtractMethodRefactoring`, `FindLocalVarVisitor` und `FindStatementVisitor` zuständig, wobei die Benutzerinteraktion durch die Klasse `ExtractMethodDialog` ermöglicht wird. Die Veranschaulichung des schrittweisen Ablaufs des *Extract Method*-Refactorings innerhalb des *ASTRefactor* Plug-Ins sowie die Erläuterung des Zusammenwirkens der beteiligten Klassen erfolgt in Kapitel 8.5.

Zusätzlich ist die Klasse `ASTRefactorPlugin` enthalten, welche die Plug-In-Klasse von *ASTRefactor* darstellt. Sie ist von `AbstractUIPlugin` abgeleitet und bietet somit die benötigten Methoden zur Integration des Plug-Ins in die Eclipse-Plattform. Auf diese Klasse wird im weiteren Verlauf nicht näher eingegangen und daher auf die Erläuterung der Funktion von Plug-In-Klassen in Kapitel 2.8 sowie die API-Referenz von `AbstractUIPlugin`[15] verwiesen.

[15] Dokumentation zu finden unter `Platform Plug-In Developer Guide` → `Reference` → `API Reference` → `org.eclipse.ui.plugin` → `AbstractUIPlugin` des Eclipse-Hilfesystems.

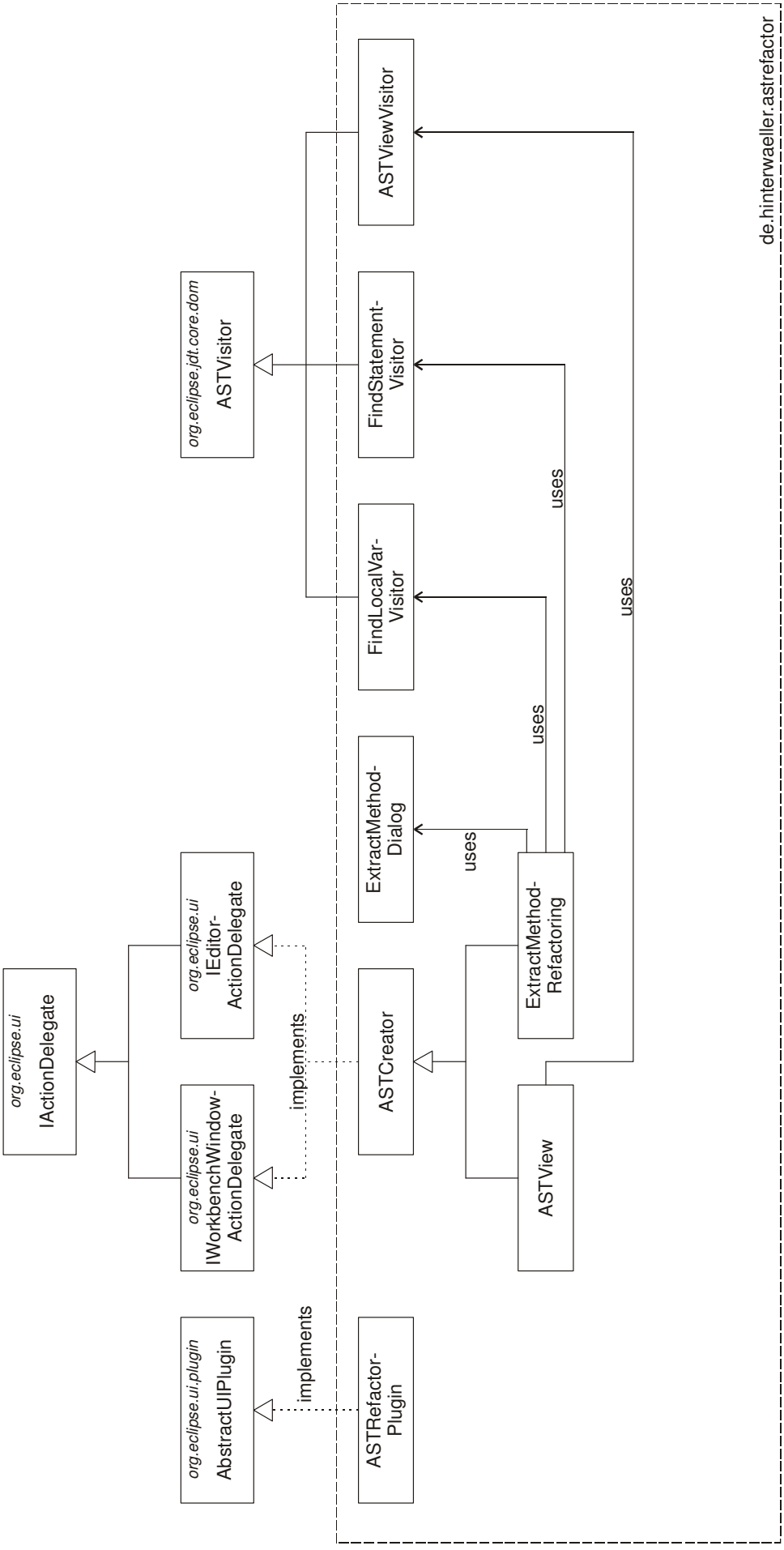


Abb. 8-4: ASTRefactor-Klassendiagramm

8.3.1 ASTCreator

Die Klasse `ASTCreator` bildet die Elternklasse von `ASTView` und `ExtractMethodRefactoring`. Sie implementiert das Interface `IWorkbenchWindow-ActionDelegate`, welches von dem Erweiterungspunkt `org.eclipse.ui.actionSets` zur Erzeugung des zusätzlichen Menüpunkts in der Eclipse-Menüleiste erwartet wird. Weiterhin implementiert die Klasse `ASTCreator` das Interface `IEditorActionDelegate` des Erweiterungspunkts `org.eclipse.ui.popupMenus` zur Bereitstellung des neuen Eintrags im Kontextmenü des Java-Editors. Beide Interfaces sind von der Schnittstelle `IActionDelegate` abgeleitet, welche die Methodendeklaration `run()` für den entsprechenden Aktionsaufruf bei Aktivierung eines Menüeintrags beinhaltet. Zudem ist dort die Methode `selection-Changed()` definiert, worüber sich das aktuell selektierte Element der Eclipse-Oberfläche ermitteln lässt.

Die Hauptfunktion der Klasse `ASTCreator` ist das Ermitteln der zugrundeliegenden Compilation Unit (d.h. der Java-Sourcedatei) entsprechend der gegenwärtig aktiven Selektion eines Elements der Eclipse-Oberfläche sowie das Erzeugen des zugehörigen abstrakten Syntaxbaums. Dieser kann durch die Klasse `ASTView` angezeigt werden beziehungsweise bildet die Basis zur Durchführung des Refactorings anhand der Klasse `ExtractMethodRefactoring`.

8.3.2 ASTView

Die Klasse `ASTView` ist abgeleitet von `ASTCreator` und wird instanziert, sobald ein Benutzer die Funktion 'Show AST' über einen Menüeintrag aufruft. `ASTView` übergibt den von `ASTCreator` erzeugten Syntaxbaum entsprechend der aktuell gewählten Compilation Unit an `ASTViewVisitor` und erhält als Rückgabe den Syntaxbaum in textueller Form, welcher sodann in einem separaten Fenster angezeigt wird.

8.3.3 ASTViewVisitor

Die Klasse `ASTViewVisitor` ist abgeleitet von `ASTVisitor` des JDT-Pakets und traversiert alle Knoten eines übergebenen ASTs. Für jeden besuchten Knoten wird eine Zeichenkette erstellt, welche aus entsprechend der Position des Knotens in der Hierarchieebene des Syntaxbaums vorangestellten Leerzeichen gefolgt vom Typbezeichner des aktuellen Elements besteht. Diese Zeichenketten werden zeilenweise in einen Textpuffer eingetragen, sodass nach Beendigung

der Traversierung eine textuelle Darstellung des ASTs vorliegt, welche sodann zur Visualisierung durch die Klasse `ASTView` zur Verfügung steht.

8.3.4 ExtractMethodRefactoring

Die Klasse `ExtractMethodRefactoring` ist ebenfalls abgeleitet von `ASTCreator` und wird instanziert, sobald ein Benutzer die Funktion 'Extract Method' über einen entsprechenden Menüpunkt aufruft. `ExtractMethodRefactoring` ist hauptsächlich für die Durchführung des *Extract Method*-Refactorings auf AST-Basis zuständig und enthält dementsprechend die in Kapitel 7 vorgestellte *Extract Method*-Transaktion (siehe Abb. 7-17).

`ExtractMethodRefactoring` nutzt die Funktionalitäten der Klassen `FindStatementVisitor` und `FindLocalVarVisitor`, um vorab die Durchführbarkeit des Refactorings zu gewährleisten und die benötigten Informationen aus dem vorliegenden AST zu extrahieren. Anhand der Klasse `ExtractMethodDialog` wird ein Dialogfenster aufgerufen, worüber der Benutzer den für das Refactoring notwendigen Methodenbezeichner eingeben und die Durchführung des Refactorings veranlassen kann.

8.3.5 FindStatementVisitor

Die Klasse `FindStatementVisitor` ist abgeleitet von `ASTVisitor` und erhält als Eingabe die durch den Benutzer innerhalb des Java-Editors markierte Textselektion. Anhand der Position dieser Markierung werden die entsprechenden Statement-Knoten des zugrundeliegenden ASTs ermittelt. Umschließt eine Markierung ein Statement nicht völlig oder befindet sich ein Statement nicht innerhalb eines Methodenrumpfs, wird dies der aufrufenden Klasse `ExtractMethodRefactoring` mitgeteilt und dort eine entsprechende Fehlermeldung ausgegeben. Ansonsten liefert `FindStatementVisitor` diejenigen Statement-Knoten, welche die Eingabe des durchzuführenden *Extract Method*-Refactorings bilden.

8.3.6 FindLocalVarVisitor

Die Klasse `FindLocalVarVisitor` ist ebenfalls abgeleitet von `ASTVisitor` und ermittelt die lokalen Variablen, deren Wert innerhalb der markierten Statements verändert wird. Diese Variablen müssen zu einem späteren Zeitpunkt per Rückgabeparameter aus der neuen Methode zurückgegeben werden. Können mehrere solche Variablen identifiziert werden, erstellt

`ExtractMethodRefactoring` eine Fehlermeldung, da in Java lediglich die Rückgabe eines einzelnen Parameters erlaubt ist.

Ferner erledigt `FindLocalVarVisitor` das Identifizieren von lokalen Variablen innerhalb der zu refaktorisierenden Statements. Diese Variablen müssen später per Parameter an die neue Methode übergeben werden. Dementsprechend erzeugt `ExtractMethodRefactoring` während der Durchführung der *Extract Method*-Transaktion die Argumente der neuen Methodendeklaration.

8.3.7 ExtractMethodDialog

Die Klasse `ExtractMethodDialog` erstellt einen Benutzerdialog zur Durchführung des Refactorings und wird von `ExtractMethodRefactoring` aufgerufen. Nach Eingabe eines Bezeichners für die neu zu erstellende Methode kann die *Extract Method*-Transaktion gestartet und abschließend eine Vorschau des refaktorisierten Sourcecodes betrachtet werden.

8.4 Implementation der AST-Manipulation

Im Folgenden werden einige Details zur Manipulation des Java-Sourcecodes auf AST-Ebene erläutert, da dies die Kernfunktionalität des *ASTRefactor* Plug-Ins bildet. Das Erzeugen des Syntaxbaums zu einer vorliegenden Java-Sourcedatei anhand der JDT-Klasse `ASTParser` wurde bereits in Kapitel 4.4 erläutert. Der folgende Codeauszug (siehe Abb. 8-5) stellt das allgemeine Vorgehen zur Erzeugung einer neuen Methodendeklaration unter Anwendung der von JDT zur Verfügung gestellten Funktionalitäten dar und ist in angepasster Form in der `ExtractMethodRefactoring`-Klasse enthalten. Die einzelnen Schritte werden nachfolgend erläutert sowie an entsprechender Stelle auf die im *ASTRefactor* Plug-In enthaltenen Klassen und deren Methoden verwiesen.

Das Aufzeichnen der Änderungen am vorliegenden Syntaxbaum sowie das Übertragen der vorgenommenen Manipulationen auf den Sourcecode ermöglichen entsprechende JDT-Funktionalitäten, welche allerdings nicht eingehender betrachtet wurden, da dies mit einer zeitaufwändigen Einsicht des JDT-Sourcecodes verbunden gewesen wäre.

Für detailliertere Informationen bezüglich der Manipulation von Java-Sourcecode wird daher auf das entsprechende Tutorial[16] innerhalb der JDT-Dokumentation verwiesen.

```
     /* init */
[1]  String methodName = ...
[2]  CompilationUnit astRoot = ...
[3]  astRoot.recordModifications();

     /* create method declaration */
[4]  TypeDeclaration currentTD = ...
[5]  AST currentAST = astRoot.getAST();
[6]  MethodDeclaration newMD = currentAST.newMethodDeclaration();
[7]  SimpleName newSN = currentAST.newSimpleName(methodName);
[8]  newMD.setName(newSN);
[9]  currentTD.bodyDeclarations().add(newMD);

     /* rewrite ast modifications */
[10] ICompilationUnit icu = ...
[11] String javaSource = icu.getBuffer().getContents();
[12] Document tempJavaSourceDoc = new Document(javaSource);
[13] TextEdit changes = astRoot.rewrite(tempJavaSourceDoc, null);
[14] changes.apply(tempJavaSourceDoc);
[15] String modifiedJavaSource = tempJavaSourceDoc.get();
[16] icu.getBuffer().setContents(modifiedJavaSource);
```

Abb. 8-5: Sourcecode-Manipulation auf AST-Ebene

8.4.1 Initialisierung

Der Bezeichner der neuen Methode wird von der Klasse `ExtractMethodDialog` geliefert und in `[1]` der Variablen `methodName` zugewiesen. Auf den Syntaxbaum, welcher durch Hinzufügen der neuen Methodendeklaration transformiert wird, kann in `[2]` anhand `astRoot` zugegriffen werden. Die Variable `astRoot` beinhaltet die Wurzel des ASTs, welcher zuvor von der Klasse `ASTCreator` erzeugt wurde, und ist entsprechend der zugrundeliegenden Java-Sourcedatei vom Typ `CompilationUnit`. In `[3]` erfolgt durch `recordModifications()` das Aktivieren der Änderungsverfolgung an dem durch die Wurzel `astRoot` repräsentierten Syntaxbaum. Diese Änderungen werden intern gespeichert und können nach Beendigung der Manipulation zur weiteren Verarbeitung abgerufen werden.

[16] Zu finden im Eclipse-Hilfesystem unter `JDT Plug-In Developer Guide → Programmer's Guide → JDT Core → Manipulating Java code`.

8.4.2 Erzeugen der Methodendeklaration

In den Zeilen `[4]` bis `[9]` erfolgt die Erstellung der zusätzlichen Methodendeklaration auf AST-Ebene. Zu diesem Zweck wird anfangs in `[4]` derjenige Typdeklarations-Knoten, unterhalb dessen die neue Methodendeklaration einzufügen ist, ermittelt und in der Variablen `currentTD` zwischengespeichert. Im *ASTRefactor* Plug-In wird dies durch Aufruf der Methode `getCurrentTypeDeclaration()` der Klasse `ExtractMethodRefactoring` erledigt.

In `[5]` wird anschließend per `getAST()`, aufgerufen über die Wurzel des aktuellen Syntaxbaums, als Resultat ein Objekt vom Typ `AST` zurückgegeben und in der Variablen `currentAST` hinterlegt. Diese Instanz der Klasse `AST` ist der Repräsentant des gesamten Syntaxbaums inklusive aller enthaltenen Knoten-Elemente, d.h. die einzelnen Knoten eines Syntaxbaums sind allesamt im Besitz dieses Repräsentanten. Die in der Klasse `AST` deklarierten Methoden dienen dazu, einen abstrakten Syntaxbaum aufzubauen, wobei für jedes vorkommende AST-Knoten-Element eine entsprechende Methode zur Verfügung steht. Ein auf diese Weise erzeugter Knoten geht daher unweigerlich in den Besitz des entsprechenden Repräsentanten über und seine Zugehörigkeit ist eindeutig festgelegt. Somit kann in `[6]` durch `currentAST.newMethodDeclaration()` ein neuer Methodendeklarations-Knoten erzeugt werden, der sodann dem aktuellen AST angehört und in der Variablen `newMD` zwischengespeichert wird.

In `[7]` erzeugt `newSimpleName()` auf die gleiche Art einen weiteren Knoten, welcher innerhalb des ASTs den Bezeichner der neuen Methode repräsentiert und dementsprechend mit `methodName` initialisiert wird. In `[8]` folgt mittels `setName()` die Zuweisung als Bezeichner für die zuvor erzeugte Methodendeklaration und in `[9]` wird abschließend der vorab ermittelten Typdeklaration anhand `bodyDeclarations().add()` die neue Methodendeklaration eingefügt.

8.4.3 Übertragen der AST-Transformationen auf den Sourcecode

Nachdem die AST-Transformation abgeschlossen ist, müssen die durchgeführten Änderungen auf den Sourcecode übertragen werden. Dies findet in den Zeilen `[10]` bis `[16]` statt und beginnt mit dem Ermitteln der Java-Datei, deren zugehöriger AST soeben manipuliert wurde. Im Eclipse-Kontext wird diese durch eine `ICompilationUnit` repräsentiert und kann im *ASTRefactor* Plug-In durch den Aufruf von `getJavaSource()` der Klasse `ASTCreator`

abgerufen werden. In `[11]` wird mittels `getBuffer().getContents()` der enthaltene Sourcecode in Form einer Zeichenkette extrahiert und in der Variablen `javaSource` abgelegt.

Danach folgt in `[12]` die Erzeugung eines `Document` auf Basis der Sourcecode-Zeichenkette sowie das Zuweisen an die temporäre Variable `tempJavaSourceDoc`. Über `astRoot.rewrite()` erfolgt in `[13]` auf Grundlage des soeben erzeugten Sourcecode-Dokuments eine Konvertierung der intern aufgezeichneten AST-Modifikationen. Das Ergebnis ist ein `TextEdit`-Objekt, welches die vorzunehmenden Sourcecode-Änderungen in textueller Form beinhaltet. In `[14]` werden diese Änderungen mittels `apply()` auf das Sourcecode-Dokument übertragen und in `[15]` der Inhalt des modifizierten Dokuments anhand `get()` der String-Variablen `modifiedJavaSource` zugewiesen.

Abschließend wird der modifizierte Java-Sourcecode durch `getBuffer().setContents()` in `[16]` auf die durch eine `ICompilationUnit` repräsentierte Java-Datei übertragen und der ursprünglich vorhandene Sourcecode überschrieben. Als Resultat erscheint der refaktorisierte Sourcecode in dem entsprechenden Java-Editor und die Transformation auf AST-Ebene ist somit abgeschlossen.

8.5 Ablauf des Extract Method-Refactorings

Im Folgenden wird der Ablauf des *Extract Method*-Refactorings anhand des *ASTRefactor* Plug-Ins schrittweise erläutert. Dabei wird davon ausgegangen, dass keine Exceptions aufgrund von Fehlern ausgelöst werden und auch die eigentliche *Extract Method*-Transaktion fehlerfrei durchgeführt wird, d.h. der vom Benutzer eingegebene neue Methodennamen existiert weder in der aktuellen Compilation Unit noch in einer der Superklassen.

Zur Veranschaulichung zeigt folgendes Sequenzdiagramm (siehe Abb. 8-6) die Instanzen der beteiligten Klassen sowie den Benutzer, welcher die Ausführung des Refactorings durch seine Eingaben steuert. Weiterhin sind die Kernfunktionen der entsprechenden Klassen aufgeführt, welche maßgeblich an der Durchführung des Refactorings beteiligt sind.

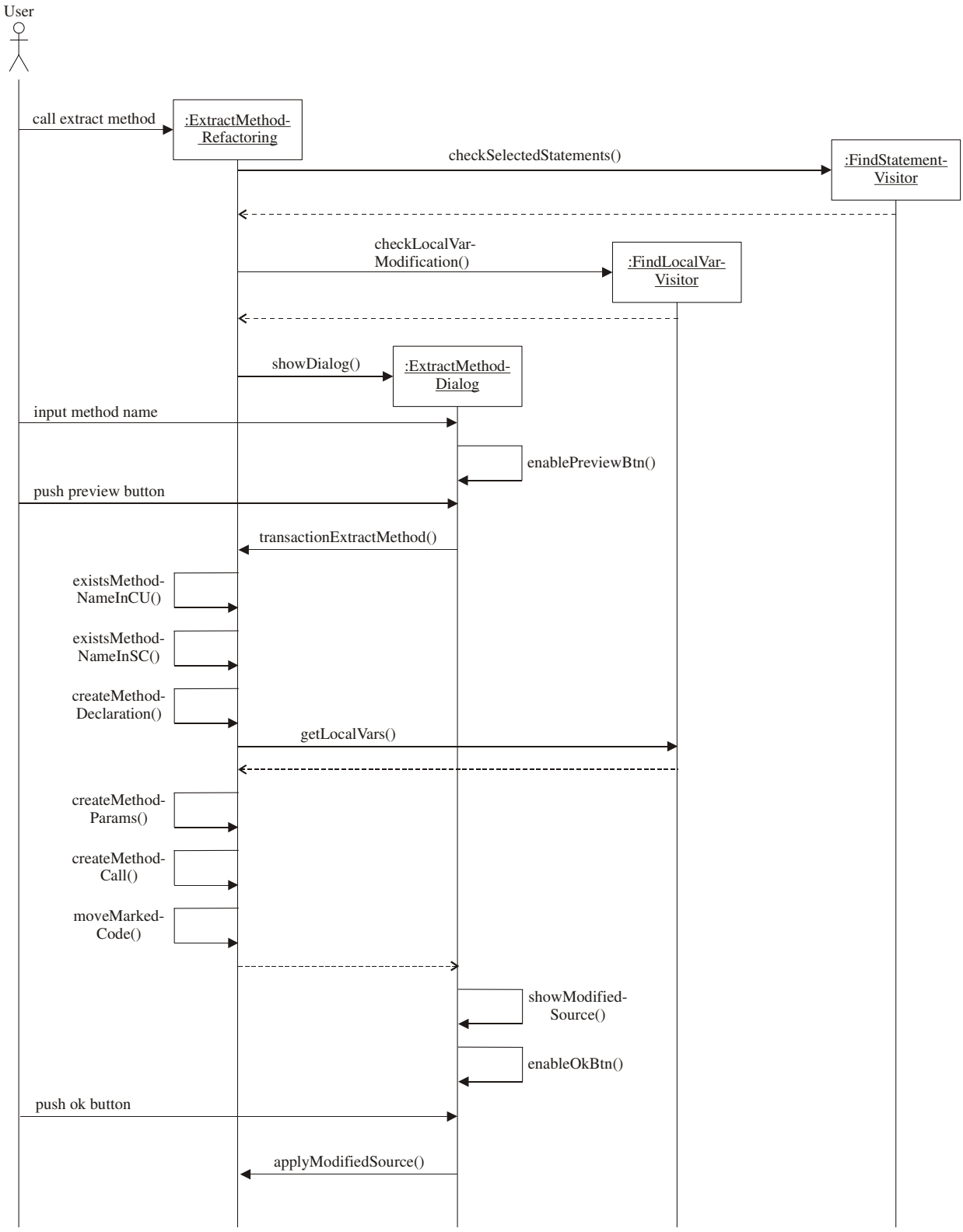


Abb. 8-6: ASTRefactor-Sequenzdiagramm

Der Ablauf gestaltet sich folgendermaßen: Ruft ein Benutzer (`User`) das *Extract Method*-Refactoring auf (`call extract method`), wird durch die Eclipse-Umgebung eine Instanz der Klasse `ExtractMethodRefactoring` erzeugt und die Methode `run()` ausgeführt. Diese beinhaltet ein Überprüfen der seitens des Benutzers markierten Codezeilen (`checkSelectedStatements()`) durch die Klasse `FindStatementVisitor`. Wurden gültige Statements selektiert, erfolgt eine Überprüfung auf wertverändernde Zuweisungen an die lokalen Variablen innerhalb der markierten Statements (`checkLocalVarModification()`) anhand der Klasse `FindLocalVarVisitor`.

Wurde die bisherige Ausführung durch keinerlei Ausnahmen unterbrochen, liegt ein gültiger Zustand zur Ausführung des *Extract Method*-Refactorings vor. Daher wird nun eine Instanz der Klasse `ExtractMethodDialog` erzeugt und die Methode `showDialog()` aufgerufen.

`ExtractMethodDialog` öffnet ein Dialogfenster und ermöglicht dem Benutzer die Eingabe eines Bezeichners für die neu zu erstellende Methode. Gibt der Benutzer einen Methodenbezeichner ein (`input method name`), wird die 'Preview'-Schaltfläche des Dialogfensters aktiviert (`enablePreviewBtn()`) und auf eine Vorschau-Anforderung des Benutzers gewartet.

Erfolgt eine solche Anforderung, wird das *Extract Method*-Refactoring anhand der in der Klasse `ExtractMethodRefactoring` enthaltenen Methode `transactionExtractMethod()` (siehe Abb. 8-7) auf AST-Basis schrittweise ausgeführt.

```
protected boolean transactionExtractMethod(String methodName) {
   if (existsMethodNameInCU(methodName)) {
      error(methodName, "compilation unit");
      return false;
   }
   if (existsMethodNameInSC(methodName)) {
      error(methodName, "superclass");
      return false;
   }
   createMethodDeclaration(methodName);
   Vector localVars = getLocalVars();
   if (localVars.size() > 0) {
      createMethodParam(localVars);
   }
   createMethodCall(methodName, localVars);
   moveMarkedCode();
   return true;
}
```

Abb. 8-7: Extract Method-Transaktion

Die Methode `transactionExtractMethod()` erwartet als Eingabe den Bezeichner der neu zu erzeugenden Methode, welcher durch den Benutzerdialog `ExtractMethodDialog` übergeben wird. Auf die zusätzlich benötigten Informationen zur Durchführung des Refactorings wie den zu transformierenden Syntaxbaum sowie die markierten Statements wird anhand der entsprechend in dieser Klasse deklarierten Instanzvariablen zugegriffen. Mittels des Boolean-Rückgabewerts der *Extract Method*-Transaktion wird `ExtractMethodDialog` über den Verlauf des Refactorings benachrichtigt, damit dort entsprechende Maßnahmen zur Darstellung des Dialogfensters ergriffen werden können.

Der Ablauf von `transactionExtractMethod()` gestaltet sich wie folgt: Durch `existMethodNameInCU()` wird überprüft, ob der neue Methodenbezeichner innerhalb der vorliegenden Compilation Unit bereits existiert. Ist dies der Fall, wird eine entsprechende Fehlermeldung ausgegeben und die Verarbeitung abgebrochen. Anderenfalls wird per `existsMethodNameInSC()` das Vorkommen des Methodenbezeichners in der Superklasse kontrolliert und demzufolge entweder die Transaktion mit einer Fehlermeldung beendet oder die Ausführung fortgesetzt.

Aufgrund des bereits eingegebenen Methodenbezeichners kann nun durch den Aufruf von `createMethodDeclaration()` die neue Methodendeklaration auf AST-Basis erzeugt werden. Mit `getLocalVars()` werden anhand von `FindLocalVarVisitor` die in den zu refaktorisierenden Statements vorhandenen lokalen Variablen ermittelt, in Form eines Vektors zurückgegeben und für die weitere Durchführung des Refactorings zwischengespeichert. Enthält dieser Vektor Elemente, d.h. es existieren lokale Variablen, müssen dementsprechend mittels `createMethodParams()` die Parameter für die neu erzeugte Methodendeklaration erzeugt werden.

Abschließend wird mit `createMethodCall()` der benötigte Aufruf der neuen Methode erzeugt und durch `moveMarkedCode()` das Verschieben der markierten Statements durchgeführt. Damit ist das *Extract Method*-Refactoring auf AST-Basis abgeschlossen und die erfolgreiche Terminierung der Transaktion wird `ExtractMethodDialog` anhand des positiven Rückgabewerts mitgeteilt.

Die Methode `showModifiedSource()` von `ExtractMethodDialog` veranlasst das Anzeigen des refaktorisierten Sourcecodes im Dialogfenster und aktiviert die OK-Schaltfläche (`enableOkBtn()`). Bestätigt der Benutzer die Sourcecode-Transformationen (`push ok button`), wird `applyModifiedSource()` aufgerufen und das Dialogfenster geschlossen. Damit werden die Änderungen wirksam, d.h. die Sourcecode-Transformationen werden auf die zugrunde liegende Compilation Unit übertragen, und das Refactoring auf AST-Basis ist abgeschlossen.

8.6 Verwendung des Plug-Ins

Das erstellte *ASTRefactor* Plug-In wurde unter Eclipse 3.0.1 auf Basis von Java 1.4.1 entwickelt und getestet. Nachdem der Ordner `de.hinterwaeller.astrefactor` in das Plug-In-Verzeichnis der Eclipse-Installation eingefügt wurde, kann die *ASTRefactor*-Komponente nach dem Start der Eclipse-Entwicklungsumgebung verwendet werden.

Nachfolgend wird das Vorgehen zur Durchführung des *Extract Method*-Refactorings sowie das Anzeigen des zu einer Java-Sourcedatei korrespondierenden Syntaxbaums erläutert. Abschließend folgt eine Beschreibung der funktionalen Eigenschaften des *ASTRefactor* Plug-Ins.

8.6.1 Funktion 'Extract Method'

Aufgabe des *ASTRefactor* Plug-Ins ist es, das *Extract Method*-Refactoring auf die zuvor durch einen Benutzer markierten Codezeilen anzuwenden. Der Aufruf des Refactorings kann entweder über den Menüpunkt 'Extract Method' des Menüs 'ASTRefactor' in der Menüleiste der Eclipse-Oberfläche oder einen entsprechenden Eintrag im Kontextmenü des Java-Editors erfolgen. Wurden seitens des Benutzers gültige Codezeilen markiert, öffnet sich ein Dialogfenster, welches den Sourcecode in einem separaten Bereich anzeigt und die Eingabe eines Bezeichners für die neu zu erzeugende Methode ermöglicht (siehe Abb. 8-8).

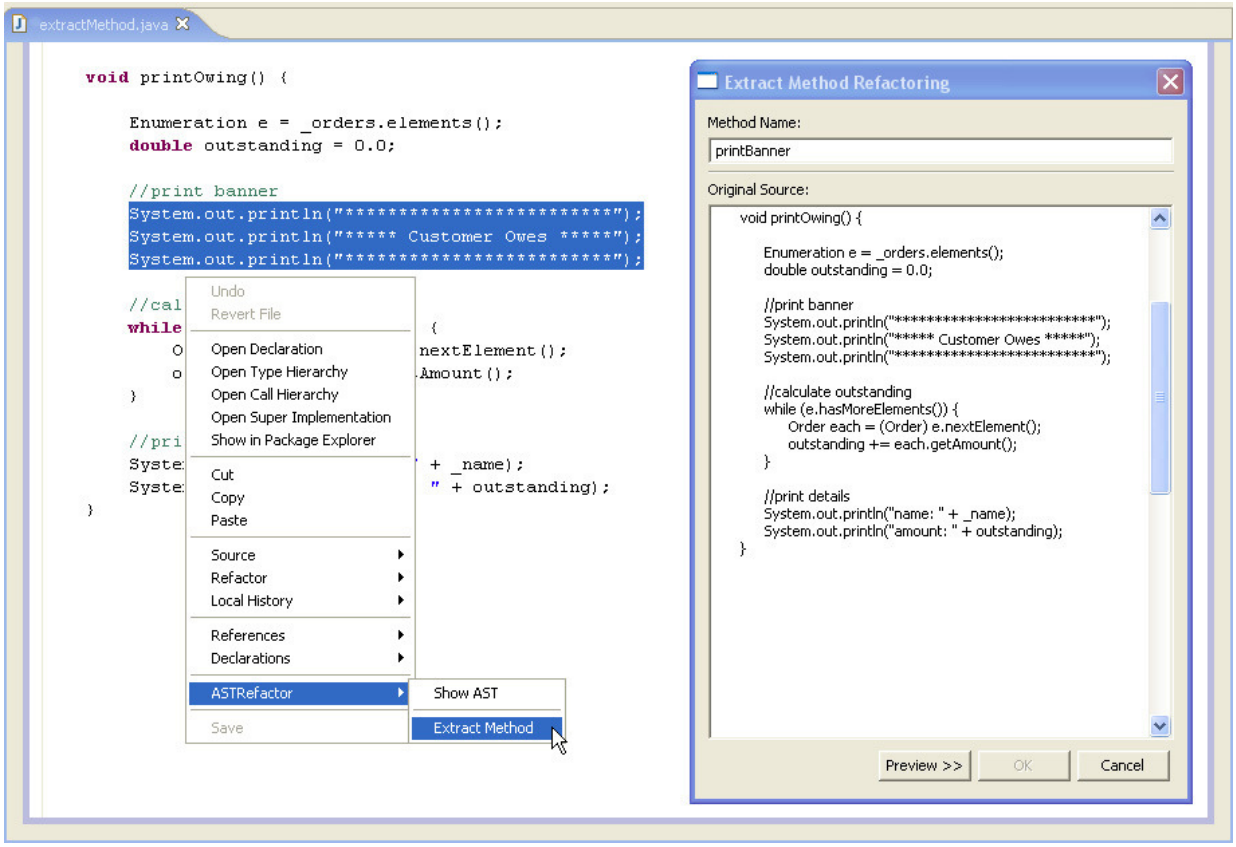


Abb. 8-8: Dialogfenster des ASTRefactor Plug-Ins

Nach Eingabe eines Methodenbezeichners und Betätigung der Schaltfläche 'Preview' wird das *Extract Method*-Refactoring auf AST-Basis durchgeführt und eine Vorschau des zu erwartenden Ergebnisses in einem weiteren Bereich des Dialogfensters angezeigt (siehe Abb. 8-9). Bestätigt der Benutzer die Ausführung des Refactorings, wird die AST-Transformation auf den Sourcecode übertragen und der refaktorisierte Code anschließend im Java-Editor angezeigt.

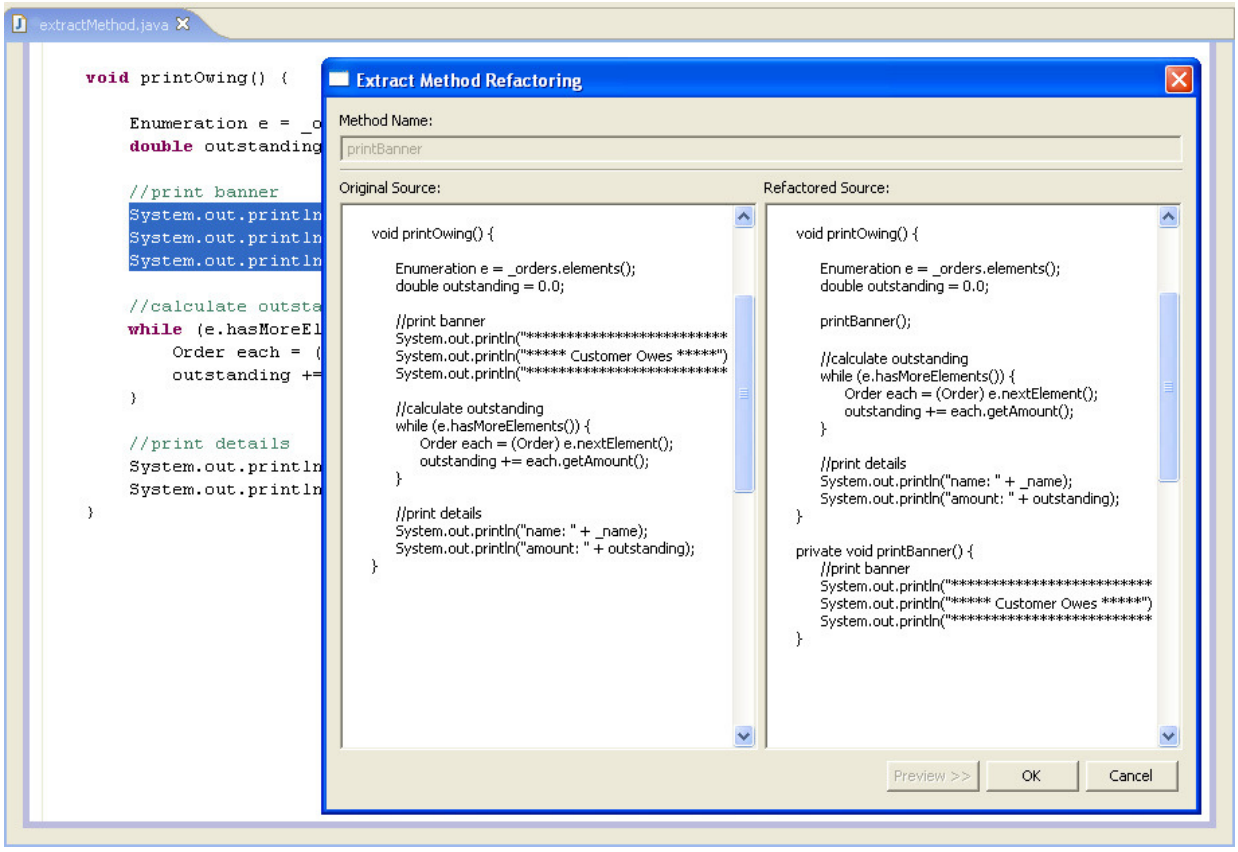


Abb. 8-9: Vorschau-Dialogfenster des ASTRefactor Plug-Ins

8.6.2 Funktion 'Show AST'

Das *ASTRefactor* Plug-In bietet die Möglichkeit, den Syntaxbaum zu einer vorliegenden Java-Sourcedatei in einem separaten Fenster anzuzeigen (siehe Abb. 8-10). Anhand des Menüelements 'Show AST' des Menüs 'ASTRefactor' der Eclipse-Menüleiste beziehungsweise des Editor-Kontextmenüs wird eine textuelle Darstellung des entsprechenden ASTs geöffnet, wobei insbesondere die Darstellung der *SimpleName*-Elemente mitsamt den repräsentierten Bezeichnern erfolgt. Zusätzlich kann die Visualisierung des ASTs über das Kontextmenü einer im Package Explorer aufgeführten Java-Datei aufgerufen werden.

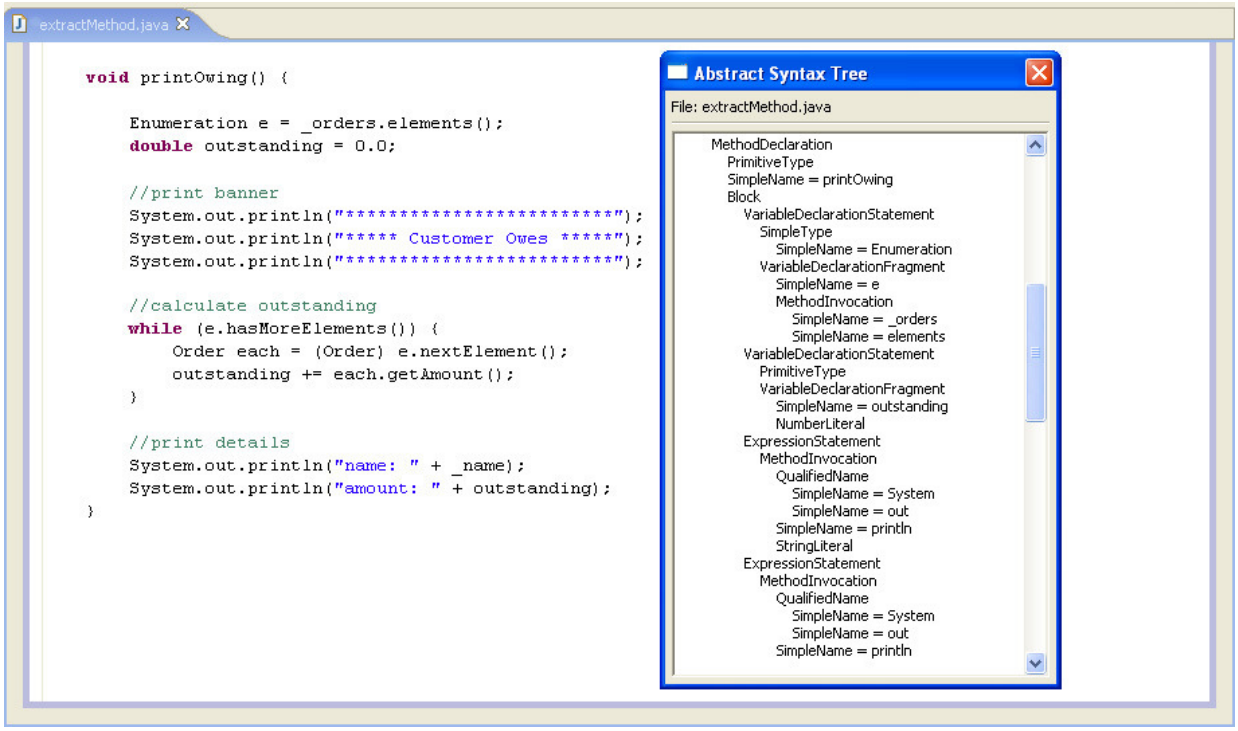


Abb. 8-10: AST-Darstellung des ASTRefactor Plug-Ins

8.6.3 Funktionale Erweiterung

Das *ASTRefactor* Plug-In bietet eine zusätzliche Funktionalität, welche während der Beschreibung und Spezifikation des Refactorings nicht betrachtet wurde. Wie bereits in den vorangegangenen Kapiteln erwähnt, umfasst das von Fowler beschriebene *Extract Method*-Refactoring einen weiteren Schritt: Das Erzeugen eines Rückgabeparameters für die neue Methodendeklaration, sofern in den zu refaktorisierenden Codezeilen eine lokale Variable vorkommt, deren Wert durch eine Zuweisung geändert wird und daher wieder an die ursprüngliche Methode zurückgegeben werden muss.

Die Umsetzung dieser Funktionalität konnte folgendermaßen realisiert werden: `FindLocalVarVisitor` liefert diejenigen Variablen, deren Wert innerhalb der markierten Statements verändert wird. Existieren mehrere solcher Variablen, wird eine entsprechende Fehlermeldung ausgegeben, da lediglich ein Methoden-Rückgabeparameter erlaubt ist. Bei Vorhandensein von lediglich einer einzelnen Variablen wird innerhalb der *Extract Method*-Teiltransaktion `createMethodDeclaration()` der vorliegende Return-Typ ermittelt und dementsprechend in die neu zu erzeugende Methodendeklaration integriert. Weiterhin wird

während der Ausführung der Teiltransaktion `createMethodCall()` anstelle eines einfachen Methodenaufrufs eine Zuweisung der Form `localVariable = newMethodCall` erstellt und somit die Rückgabe der wertveränderten Variable in die ursprüngliche Methode erreicht. Abschließend erfolgt in der Teiltransaktion `moveMarkedCode()` nach dem Verschieben der markierten Statements in den Rumpf der neuen Methode das Erzeugen des zugehörenden Return-Statements. Zur Veranschaulichung zeit Abb. 8-11 die Anwendung des *Extract Method*-Refactorings auf die While-Schleife in dem bereits bekannten Beispielcode.

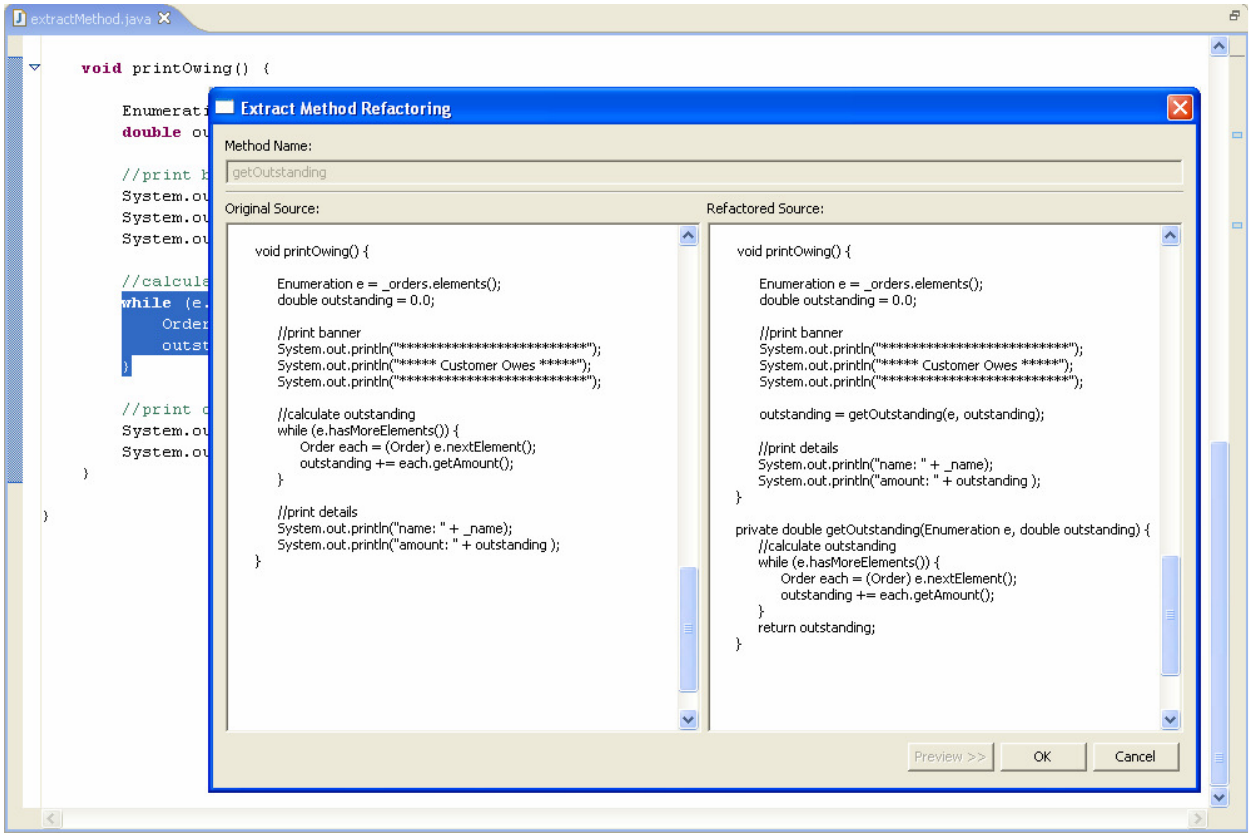


Abb. 8-11: Rückgabe der wertgeänderten Variable

Kapitel 9
Abschließende Betrachtung

Dieses Kapitel bietet einen Rückblick auf die vorliegende Diplomarbeit und stellt die erarbeiteten Ergebnisse zusammenfassend dar. Abschließend erfolgt ein Ausblick auf mögliche weiterführende Ausführungen auf Grundlage der vorliegenden Ergebnisse.

9.1 Rückblick

Während der Suche nach einem geeigneten Modell, welches einen Sourcecode in zweckdienlicher Form repräsentiert und auf dessen Basis Transformationen durchgeführt werden können, wurden das in JDT enthaltene Java-Modell sowie das darauf basierende *Mrs.G* Plug-In untersucht. Sogleich stellte sich aber heraus, dass der vorliegende Detailierungsgrad nicht genügend Informationen zur Beschreibung von Transformationen auf der gewünschten Abstraktionsebene bieten würde. Daher erfolgte ein Blick auf die ebenfalls in JDT enthaltene Funktionalität zur Erzeugung eines abstrakten Syntaxbaums. Die von diesem Modell gebotene Darstellung eines Sourcecodes bis hinab auf Token-Ebene sowie die zusätzlich enthaltenen Bindungsinformationen ließen den Schluss zu, dass dieser JDT-AST alle benötigten Angaben für den weiteren Verlauf bereitstellen würde. Daraufhin folgte das Erstellen der zugrunde liegenden Java-Grammatik unter Zuhilfenahme der AST-Dokumentation sowie dem Klassendiagramm des AST-Pakets. Aus dieser Grammatik konnte ein Metamodell, welches den Aufbau eines Syntaxbaums beschreibt, abgeleitet werden. Dieses Metamodell bildete fortan die Grundlage zur Beschreibung von Sourcecode-Manipulationen auf AST-Ebene.

Nach einer Vorstellung des Refactoring-Konzepts wurde anhand des ausgewählten *Extract Method*-Refactorings eine Sourcecode-Änderung auf AST-Basis an einem konkreten Beispiel beschrieben. Die nachfolgende Zerlegung des *Extract Method*-Refactorings in Einzelschritte diente vor allem dazu, Vorbedingungen sowie Teiltransformationen zu identifizieren und lieferte somit Erkenntnisse über die zur Durchführung des *Extract Method*-Refactorings benötigten Sourcecode-Informationen. Dazu zählten insbesondere die in einer Klasse global deklarierten Variablen, lokale Variablendeklarationen innerhalb einer Methode, bereits

vorhandene Methodenbezeichner sowie damit verbundene Angaben über den Aufbau der Vererbungshierarchie. Während der Beschreibung der einzelnen Refactoring-Schritte auf Syntaxbaum-Basis konnte zudem gezeigt werden, dass der von JDT zur Verfügung gestellte AST inklusive der enthaltenen Bindungsinformationen die nötigen Angaben bereitstellte, um Codemanipulationen auf Modell-Ebene beschreiben und durchführen zu können.

Letztendlich fehlte eine formale Spezifikation zur Durchführung des *Extract Method*-Refactorings auf AST-Ebene. Daher wurde die Sprache PROGRES einschließlich ihrer kennzeichnenden Konzepte erläutert, welche eine Möglichkeit zur formalen Definition von Graphtransformationen und damit zur Beschreibung von Codemanipulationen auf Modell-Ebene bot. Die Spezifikation einiger Teilschritte des *Extract Method*-Refactorings bereitete jedoch Probleme, denn die Darstellung von Kontrollstrukturen mittels Kanteniteratoren wird seitens PROGRES nicht unterstützt. Ebenso ließ die fehlende Ordnung der Kantenfolge eine nur unzureichende Spezifikation des *Extract Method*-Refactorings zu.

Aufgrund dieser auf den zugrundeliegenden Graphgrammatik-Ansatz zurückzuführenden Unzulänglichkeiten wurde sodann nach einer alternativen Möglichkeit zur Spezifikation des Refactorings gesucht. Daraus resultierte die Einführung der Spezifikationssprache GRAL und der auf dieser aufbauenden Anfragesprache GReQL, welche gemeinsam zur algorithmischen Spezifikation des *Extract Method*-Refactorings eingesetzt wurden. Unter Verwendung dieser Sprachen und zusätzlicher Kontrollstrukturen ließen sich verschiedene Sachverhalte im Gegensatz zu PROGRES eleganter und einfacher ausdrücken, was insbesondere auf die bereits in GRAL/GReQL vorhandenen Funktionen zur Traversierung eines Graphen und Zugriffsmethoden auf Kanten und Knoten zurückzuführen war. Zudem war es jetzt möglich, Kanten in einer geordneten Folge zu betrachten sowie Kanteniteratoren einzusetzen, wodurch die Beschreibung des Kontrollflusses ermöglicht wurde. Einen weiteren Vorteil gegenüber der aufwändigen graphischen PROGRES-Notation bot darüber hinaus die Darstellung der GRAL/GReQL-Konstrukte in ASCII-Notation.

Somit war das Ziel dieser Diplomarbeit erreicht, denn es konnte ein Vorgehen skizziert werden, welches Sourcecode-Änderungen mittels Transformationen eines entsprechenden Sourcecode-Modells an einem komplexen Fallbeispiel ermöglichte.

Dieses Vorgehen lässt sich folgendermaßen zusammenfassen: Aus einem vorliegenden Sourcecode wird ein abstrakter Syntaxbaum extrahiert, welcher eine Instanz des gefundenen Metamodells ist und inklusive der zur Verfügung stehenden Bindungsinformationen detaillierte Angaben über den zugrunde liegenden Programmcode enthält. Auf dieser Basis können Sourcecode-Änderungen durchgeführt werden, wobei die Spezifikation der AST-Transformationen operational durch einen GRAL/GReQL-Ausdrücke verwendenden Pseudocode erfolgt. Ein auf diese Weise veränderter AST ist ebenfalls eine Instanz des gemeinsamen Metamodells und enthält offensichtlich die nötigen Informationen, um hieraus den refaktorisierten Sourcecode erzeugen zu können. Zur Veranschaulichung zeigt Abb. 9-1 das Verfahren nochmals in graphischer Form.

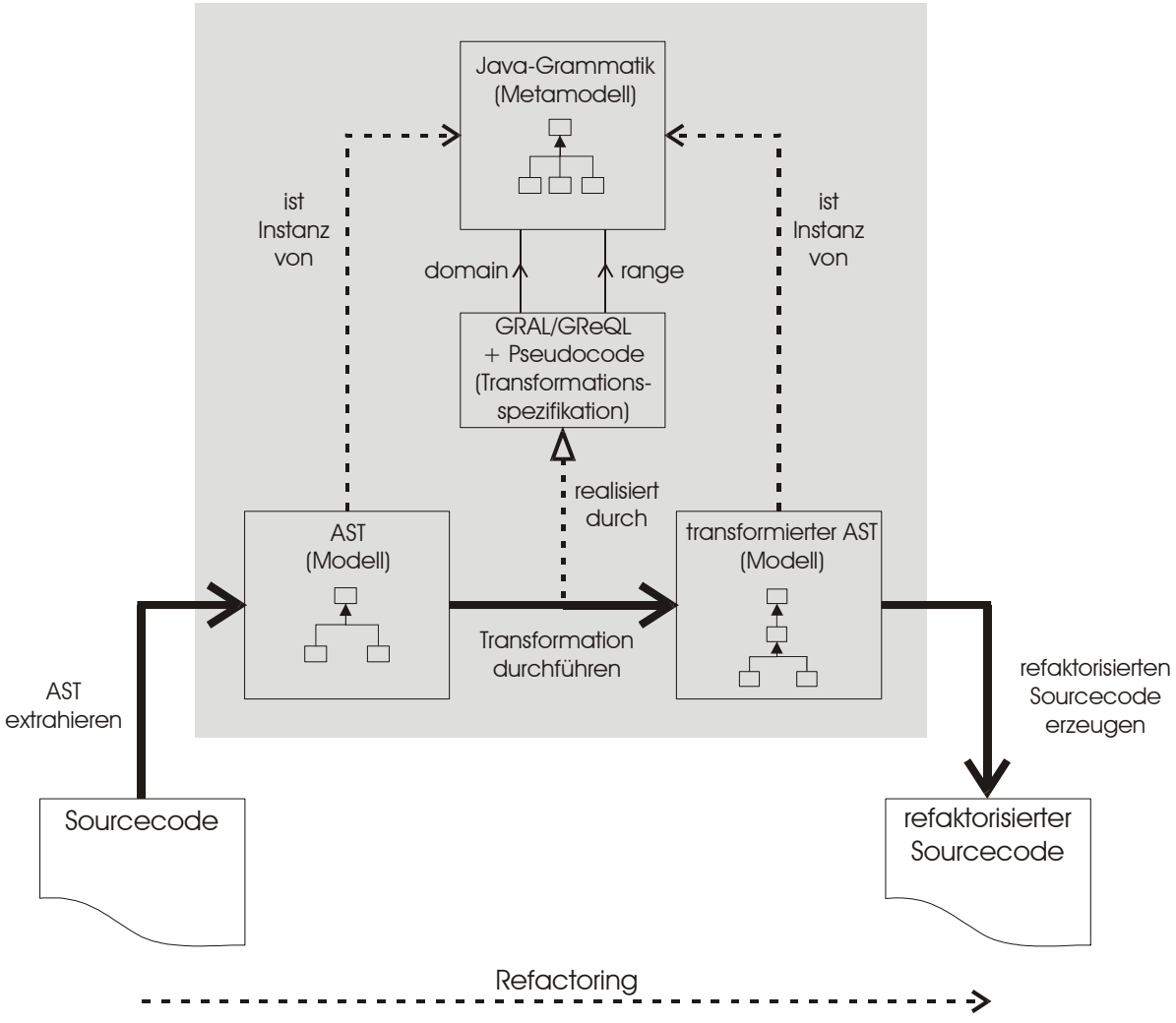


Abb. 9-1: Sourcecode-Änderung durch Modell-Transformation

9.2 Allgemeines Vorgehen zur Entwicklung eines Refactoring Plug-Ins

Mit einer anfänglichen Darstellung des Plug-In-Konzepts der Entwicklungsumgebung Eclipse sowie einer Erläuterung zur Implementation von Komponenten unter Einbeziehung bereits vorhandener Erweiterungspunkte wurde vorab die Grundlage zur Erstellung eines eigenen Plug-Ins geschaffen. Das *ASTRefactor* Plug-In, dessen Vorgehen zur Transformation eines Syntaxbaums aus der *Extract Method*-Spezifikation abgeleitet wurde, repräsentiert somit eine prototypische Umsetzung des erarbeiteten Konzepts.

Das konkrete Vorgehen zur Entwicklung des *ASTRefactor* Plug-Ins, welches ein Metamodell-basiertes Refactoring für Java-Sourcecode ermöglicht, lässt sich folgendermaßen verallgemeinern:

1. Anfänglich ist ein zu implementierendes Refactoring in seine Einzelschritte zu zerlegen.
2. Auf Grundlage der Zerlegung können die enthaltenen Vorbedingungen und Teiltransformationen identifiziert werden.
3. Diese sind sodann entsprechend dem JDT-AST-Metamodell operational zu spezifizieren und abschließend in einer Refactoring-Transaktion zusammenzufassen[17].
4. Abschließend kann das Plug-In erzeugt und die benötigten Menüpunkte sowie die aufzurufende Klasse in der Manifest-Datei deklariert werden[18].

Ein auf diese Weise erstelltes Refactoring Plug-In kann sodann zur Änderung von Java-Sourcecode eingesetzt werden. Der nachfolgend notwendige Syntaxcheck des refaktorisierten Sourcecodes wird dabei durch den in Eclipse enthaltenen inkrementellen Compiler vorgenommen. Zur Überprüfung der verhaltensbewahrenden Programmtransformation können abschließend entsprechende Tests, beispielsweise unter Einsatz von JUnit, durchgeführt werden.

[17] Zusätzlich sollten auch sprachspezifische Eigenschaften, wie beispielsweise das in Java maximal zulässige Vorkommen eines einzigen Methoden-Rückgabeparameters, betrachtet und entsprechend spezifiziert werden.

[18] Diese Klasse bietet somit die Funktionalität zur Durchführung des Refactorings und enthält dementsprechend die zuvor formulierte Refactoring-Transaktion, deren Implementation aus der Spezifikation hergeleitet wird.

Der soeben skizzierte Ansatz zur Entwicklung von Metamodell-basierten Refactorings ist dahingehend erweiterbar, das dargestellte Verfahren in ein Werkzeug zur Generierung von Erweiterungen für Eclipse zu integrieren, welches aus einer zuvor erstellten Refactoring-Spezifikation die entsprechende Java-Implementation erzeugt und somit weitere Sourcecode-Transformationen auf Modell-Ebene in Form von Plug-Ins zur Verfügung stellt. Mit diesem Refactoring-Generator wäre es somit auf einfache Art möglich, die Eclipse-Umgebung um zusätzliche Refactorings zu ergänzen.

9.3 Anforderungen an einen Metamodell-basierten Transformationsansatz

Aus den gesammelten Erfahrungen sowie dem soeben skizzierten allgemeinen Vorgehen zur Erstellung eines Refactoring Plug-Ins lassen sich zudem folgende grundlegende Anforderungen an einen Metamodell-basierten Transformationsansatz herleiten:

Das Metamodell muss alle zur Durchführung von Refactorings benötigten Informationen bereitstellen. Dazu zählen neben dem strukturellen Aufbau eines vorliegenden Sourcecodes insbesondere zusätzliche Typinformationen und semantische Details. Diese gestatten beispielsweise das Identifizieren von lokalen und globalen Variablendeklarationen, geben Hinweise auf bereits vorhandene Bezeichner sowie deren Sichtbarkeit, enthalten Angaben über Methodensignaturen und ermöglichen eine Einsicht in die vorliegende Vererbungshierarchie.

Weiterhin muss die Spezifikationssprache die nötigen Konzepte zur Beschreibung von Graphtransformationen bieten. Dazu sollte neben Pfadausdrücken und Kontrollflussbeschreibungen auch das Verwenden von Graphpattern in Betracht gezogen werden. Außerdem sollte es möglich sein, die Spezifikation der Graphtransformation auf die entsprechende Zielsprache übertragen zu können. Ferner muss gewährleistet sein, dass das Ergebnis einer Graphtransformation einen konsistenten Endzustand erreicht, d.h. der transformierte Graph wiederum eine Instanz des Metamodells darstellt und somit der Syntax der Zielsprache entspricht.

9.4 Ausblick

Die vorliegende Diplomarbeit hat gezeigt, wie sich die Anwendung eines konkreten Refactorings auf Basis eines Syntaxbaums inklusive der enthaltenen Bindungsinformationen beschreiben und spezifizieren lässt. Allerdings basiert das Vorgehen auf dem von JDT zur Verfügung gestellten AST und ist daher an der Programmiersprache Java orientiert. Das Übertragen auf ein allgemeineres Vorgehen und somit ein Distanzieren von den eingesetzten JDT-Funktionalitäten beinhaltet einige Aspekte, welche weiterführende Untersuchungen erfordern.

Zur Verallgemeinerung des Vorgehens ist es unerlässlich, sich von dem bisher betrachteten JDT-AST-Modell abzuwenden und den Sourcecode anhand eines geeigneten Modells – beispielsweise in Form eines TGraphen – darzustellen, welches die benötigten Konzepte zur Repräsentation der zur Durchführung von verschiedenen Refactorings erforderlichen Informationen enthält. Als Grundlage zur Erstellung eines entsprechenden Metamodells wird insbesondere das Zerlegen weiterer Refactorings hilfreich sein, um die Liste der benötigten Sourcecode-Informationen zu erweitern.

Aus diesem Graphen, welcher sodann einen zu verändernden Sourcecode repräsentiert, können mittels einer Anfragesprache – wie beispielsweise GReQL – die zu transformierenden Elemente extrahiert und im Zuge des durchzuführenden Refactorings entsprechend umgestaltet werden. Diese Manipulationen können anschließend in den ursprünglichen Graphen integriert werden und auf Basis des Metamodells kann eine Konsistenzüberprüfung erfolgen.

Anhand des Metamodells kann zudem die Spezifikation weiterer Refactorings in Form von graphverändernden Transformationen erfolgen. Allerdings sollte ein Verfahren erarbeitet werden, welches aus einer deklarativen Refactoring-Spezifikation die entsprechende Graph-Transformation automatisch generiert, denn im Gegensatz zur Anfertigung des *ASTRefactor* Plug-Ins sollte es vermieden werden, den Programmcode zur Durchführung des Refactorings aus der *Extract Method*-Spezifikation abzuleiten und entsprechend anpassen zu müssen. Weiterhin sollte es möglich sein, den refaktorisierten Sourcecode aus dem geänderten Graphen herzuleiten, ohne auf die in JDT vorhandenen Funktionalitäten zurückzugreifen.

Somit wäre eine Verallgemeinerung des in dieser Arbeit beschriebenen Vorgehens erreicht und die bisher genutzten JDT-Funktionalitäten wären durch generische Konzepte zu ersetzen. Dies resultiert in einem allgemeinen Framework zur Manipulation eines existierenden Sourcecodes, welches aus drei charakteristischen Vorgehensschritten besteht (vgl. Abb. 9-1):

1. Extraktion der Sourcecode-Informationen in einen Graphen
2. Transformation des Graphen
3. Herleiten des geänderten Sourcecodes aus dem transformierten Graphen

Weitere Untersuchungen, insbesondere im Hinblick auf das Identifizieren von Graph-Patterns zur automatisierten Anwendung verschiedener Refactorings, welche das Eingreifen eines Benutzers auf ein Minimum reduzieren, sowie das Übertragen des konzeptuellen Frameworks auf verschiedene Programmiersprachen wären darüber hinaus zusätzlich interessante zu verwirklichende Ziele.

Anhang A

Java-Grammatik

Die nachfolgende Grammatik wurde aus den Klassen des Pakets `org.eclipse.jdt.core.dom` sowie unter Zuhilfenahme der zugehörenden API-Referenz hergeleitet. Der Wurzelknoten eines ASTs bildet dabei stets ein `CompilationUnit`-Element. Die weiteren Grammatik-Elemente sind nachfolgend in alphabetischer Reihenfolge aufgeführt.

CompilationUnit ::= [PackageDeclaration] { ImportDeclaration } { TypeDeclaration | ';' }

AbstractTypeDeclaration ::= TypeDeclaration | EnumDeclaration | AnnotationTypeDeclaration

Annotation ::= MarkerAnnotation | NormalAnnotation | SingleMemberAnnotation

AnnotationTypeBodyDeclaration ::= AnnotationTypeMemberDeclaration | FieldDeclaration |
TypeDeclaration | EnumDeclaration |
AnnotationTypeDeclaration

AnnotationTypeDeclaration ::= [Javadoc] { ExtendedModifier } '@' 'interface' Identifier
'{' { AnnotationTypeBodyDeclaration | ';' } '}'

AnnotationTypeMemberDeclaration ::= [Javadoc] { ExtendedModifier }
Type Identifier '(' ')' ['default' Expression] ';'

AnonymousClassDeclaration ::= '{' ClassBodyDeclaration '}'

ArrayAccess ::= Expression [Expression]

ArrayCreation ::= 'new' PrimitiveType '[' Expression ']' { '[' Expression ']' } { '[' ']' } |
'new' TypeName '[' Expression ']' { '[' Expression ']' } { '[' ']' } |
'new' PrimitiveType '[' ']' { '[' ']' } ArrayInitializer |
'new' TypeName '[' ']' { '[' ']' } ArrayInitializer

ArrayInitializer ::= '{' [Expression { ',' Expression}] '}'

ArrayType ::= Type '[' ']'

AssertStatement ::= 'assert' Expression [':' Expression] ';'

Assignment ::= Expression AssignmentOperator Expression

AssignmentOperator ::= '=' | '+=' | '-=' | '*=' | '/=' | '&=' | '|=' | '^=' | '%=' | '<<=' | '>>=' | '>>>='

```
Block ::=     '{' { Statement } '}'

BlockComment ::=     <block comment AST node type>

BodyDeclaration ::=     AbstractTypeDeclaration | AnnotationTypeMemberDeclaration |
                        EnumConstantDeclaration | FieldDeclaration | Initializer |
                        MethodDeclaration | ConstructorDeclaration

BooleanLiteral ::=     'true' | 'false'

BreakStatement ::=     'break' [ Identifier ] ';'

CastExpression ::=     '(' Type ')' Expression

CatchClause ::=     'catch' '(' FormalParameter ')' Block

CharacterLiteral ::=     <character literal nodes>

ClassBodyDeclaration ::=     BodyDeclaration

ClassDeclaration ::=     [ Javadoc ] { Modifier } 'class' Identifier
                            [ 'extends' Type]
                            [ 'implements Type' { ',' Type } ]
                            '{' { ClassBodyDeclaration | ';' } '}'

ClassInstanceCreation ::=     [ Expression . ] 'new' Name '(' [ Expression { ',' Expression } ] ')'
                                [ AnonymousClassDeclaration ]

ClassName ::=     [ Identifier '.' ] Identifier

Comment ::=     BlockComment | Javadoc | LineComment

ConditionalExpression ::=     Expression '?' Expression ':' Expression

ConstructorDeclaration ::=     [ Javadoc ] { Modifier } Identifier
                                '(' [ FormalParameter { ',' FormalParameter } ] ')'
                                ['throws' TypeName { ',' TypeName } ] Block

ConstructorInvocation ::=     'this' '(' [ Expression { ',' Expression } ] ')' ';'

ContinueStatement ::=     'continue' [ Identifier ] ';'

Digit ::=     '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

DocElement ::=     TextElement | Name | MethodRef | MemberRef | '{' TagElement '}'

DoStatement ::=     'do' Statement 'while' '(' Expression ')' ';'

EmptyStatement ::=     ';'
```

```
EnhancedForStatement ::=     'for' '(' FormalParameter ':' Expression ')' Statement

EnumConstantDeclaration ::=     [ Javadoc ] { ExtendedModifier } Identifier
                                    [ '(' [ Expression { ',' Expression } ] ')' ]
                                    [ '{' { ClassBodyDeclaration | ';' } '}' ]

EnumDeclaration ::=     [ Javadoc ] { ExtendedModifier } 'enum' Identifier
                           [ 'implements' Type { ',' Type } ]
                           '{' [ EnumConstantDeclaration { ',' EnumConstantDeclaration } ]
                           [ ';' { ClassBodyDeclaration | ';' } ] '}'

Expression ::=     Annotation | ArrayAccess | ArrayCreation | ArrayInitializer | Assignment |
                   BooleanLiteral | CastExpression | CharacterLiteral | ClassInstanceCreation |
                   ConditionalExpression | FieldAccess | InfixExpression | InstanceofExpression |
                   MethodInvocation | Name | NullLiteral | NumberLiteral |
                   ParenthesizedExpression | PostfixExpression | PrefixExpression | StringLiteral |
                   SuperFieldAccess | SuperMethodInvocation | ThisExpression | TypeLiteral |
                   VariableDeclarationExpression

ExpressionStatement ::=     StatementExpression ';'

ExtendedModifier ::=     Modifier | Annotation

FieldAccess ::=     Expression '.' Identifier

FieldDeclaration ::=     [Javadoc] { ExtendedModifier } Type VariableDeclarationFragment
                            { ',' VariableDeclarationFragment } ';'

ForInit ::=     '(' VariableDeclarationExpression ')' { Expression {',' Expression } }

FormalParameter ::=     ['final'] Type VariableDeclarationFragment

ForStatement ::=     'for' '(' [ ForInit ] ';' [ Expression ] ';' [ ForUpdate ] ')' Statement

ForUpdate ::=     Expression { ',' Expression }

Identifier ::=     IdentifierName

IdentifierName ::=     Letter { Letter | Digit }

IfStatement ::=     'if' '(' Expression ')' Statement [ 'else' Statement]

ImportDeclaration ::=     'import' Name [ '.*' ] ';'

InfixExpression ::=     Expression InfixOperator Expression { InfixOperator Expression }

InfixOperator ::=     '*' | '/' | '%' | '+' | '-' | '<<' | '>>' | '>>>' | '<' | '>' | '<=' | '>=' | '==' |'!=' | '^' | '&' | '|' |
                      '&&' | '||'

Initializer ::=     [ 'static' ] Block
```

InstanceofExpression ::= Expression 'instanceof' Type

InterfaceBodyDeclaration ::= BodyDeclaration

InterfaceDeclaration ::= [Javadoc] { Modifier } 'interface' Identifier
['extends' Type { ',' Type }]
{ '{' InterfaceBodyDeclaration | ';' '}' }

Javadoc ::= '/**' { TagElement } '*/'

LabeledStatement ::= Identifier ':' Statement

Letter ::= `<a unicode-letter>`

LineComment ::= <end-of-line comment AST node type>

MarkerAnnotation ::= '@' TypeName

MemberRef ::= [Name] '#' Identifier

MemberValuePair ::= SimpleName '=' Expression

MethodDeclaration ::= [Javadoc] { Modifier } (Type | 'void') Identifier
'(' [SingleVariableDeclaration { ',' SingleVariableDeclaration }] ')'
['throws' TypeName { ',' TypeName }] (Block | ';')

MethodInvocation ::= [Expression '.'] Identifier '(' [Expression { ',' Expression }] ')'

MethodRef ::= [Name] '#' Identifier '(' [MethodRefParameter | { ',' MethodRefParameter }] ')'

MethodRefParameter ::= Type [Identifier]

Modifier ::= 'public' | 'protected' | 'private' | 'static' | 'abstract' | 'final' | 'native' | 'synchronized' |
'transient' | 'volatile' | 'strictfp'

Name ::= SimpleName | QualifiedName

NormalAnnotation ::= '@' TypeName '(' [MemberValuePair { ',' MemberValuePair }] ')'

NullLiteral ::= <null literal node>

NumberLiteral ::= <number literal node>

PackageDeclaration ::= 'package' Name ';'

ParameterizedType ::= Type '<' Type { ',' Type } '>'

ParenthesizedExpression ::= '(' Expression ')'

PostfixExpression ::= Expression PostfixOperator

PostfixOperator ::= '++' | '--'

PrefixExpression ::= PrefixOperator Expression

PrefixOperator ::= '++' | '--' | '+' | '-' | '~' | '!'

PrimitiveType ::= 'byte' | 'short' | 'char' | 'int' | 'long' | 'float' | 'double' | 'boolean' | 'void'

QualifiedName ::= Name '.' SimpleName

QualifiedType ::= Type '.' SimpleName

ReturnStatement ::= 'return' [Expression] ';'

SingleMemberAnnotation ::= '@' TypeName '(' Expression ')'

SimpleName ::= Identifier

SimpleType ::= TypeName

SingleVariableDeclaration ::= { Modifier } Type Identifier { '[' ']' } ['=' Expression]

Statement ::= AssertStatement | Block | BreakStatement | ConstructorInvocation |
ContinueStatement | DoStatement | EmptyStatement | EnhancedForStatement |
ExpressionStatement | ForStatement | IfStatement | LabeledStatement |
ReturnStatement | SuperConstructorInvocation | SwitchCase | SwitchStatement |
SynchronizedStatement | ThrowStatement | TryStatement |
TypeDeclarationStatement | VariableDeclarationStatement | WhileStatement

StatementExpression ::= Expression

StringLiteral ::= <string literal node>

SuperConstructorInvocation ::= [Expression '.'] 'super' '(' [Expression { ',' Expression }] ')' ';'

SuperFieldAccess ::= [ClassName '.'] 'super' '.' Identifier

SuperMethodInvocation ::= [ClassName '.'] 'super' '.' Identifier
'(' [Expression { ',' Expression }] ')'

SwitchCase ::= 'case' Expression ':' | 'default' ':'

SwitchStatement ::= 'switch' '(' Expression ')' '{' { SwitchCase | Statement } '}'

SynchronizedStatement ::= 'synchronized' '(' Expression ')' Block

TagElement ::= ['@' Identifier] { DocElement }

TextElement ::= <sequence of characters not including a close comment delimiter '*/'>

ThisExpression ::= [ClassName '.'] 'this'

ThrowStatement ::= 'throw' Expression ';'

TryStatement ::= 'try' Block { CatchClause } ['finally' Block]

Type ::= ArrayType | ParameterizedType | PrimitiveType | QualifiedType | SimpleType

TypeDeclaration ::= ClassDeclaration | InterfaceDeclaration

TypeDeclarationStatement ::= TypeDeclaration

TypeLiteral ::= (Type | 'void') '.class'

TypeName ::= [Identifier '.'] Identifier

TypeParameter ::= TypeVariable ['extends' Type { '&' Type }]

TypeVariable ::= Identifier

VariableDeclaration ::= SingleVariableDeclaration | VariableDeclarationFragment

VariableDeclarationExpression ::= { Modifier } Type VariableDeclarationFragment
{ ',' VariableDeclarationFragment }

VariableDeclarationFragment ::= SimpleName { '[' ']' } ['=' Expression]

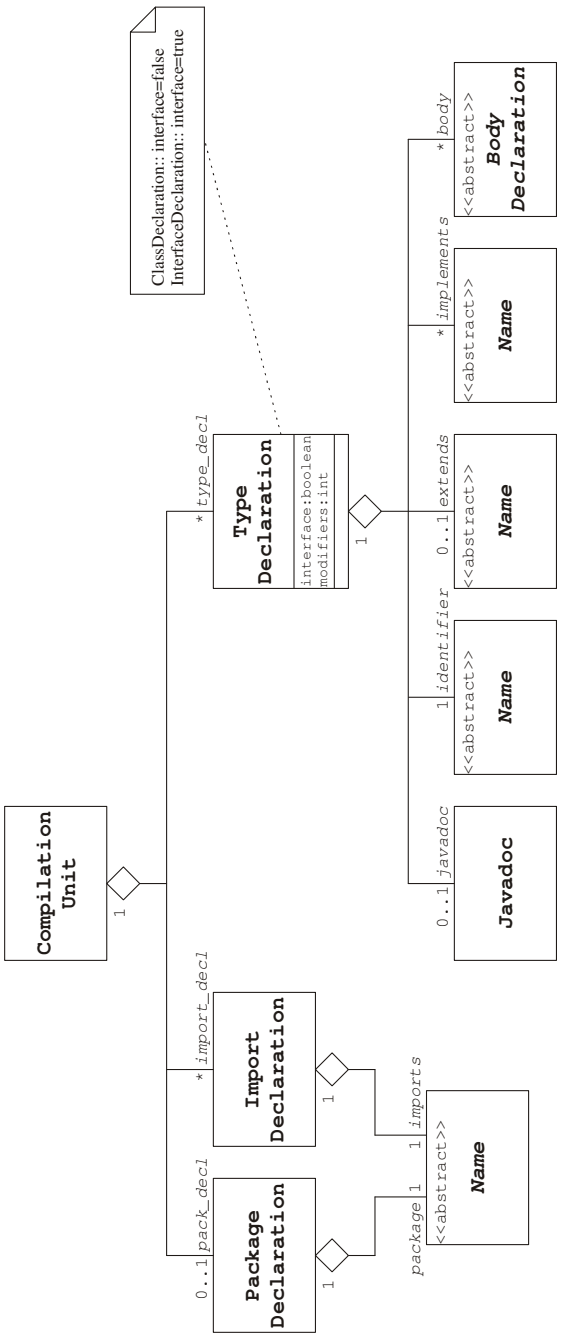
VariableDeclarationStatement ::= { Modifier } Type VariableDeclarationFragment
{ ',' VariableDeclarationFragment } ';'

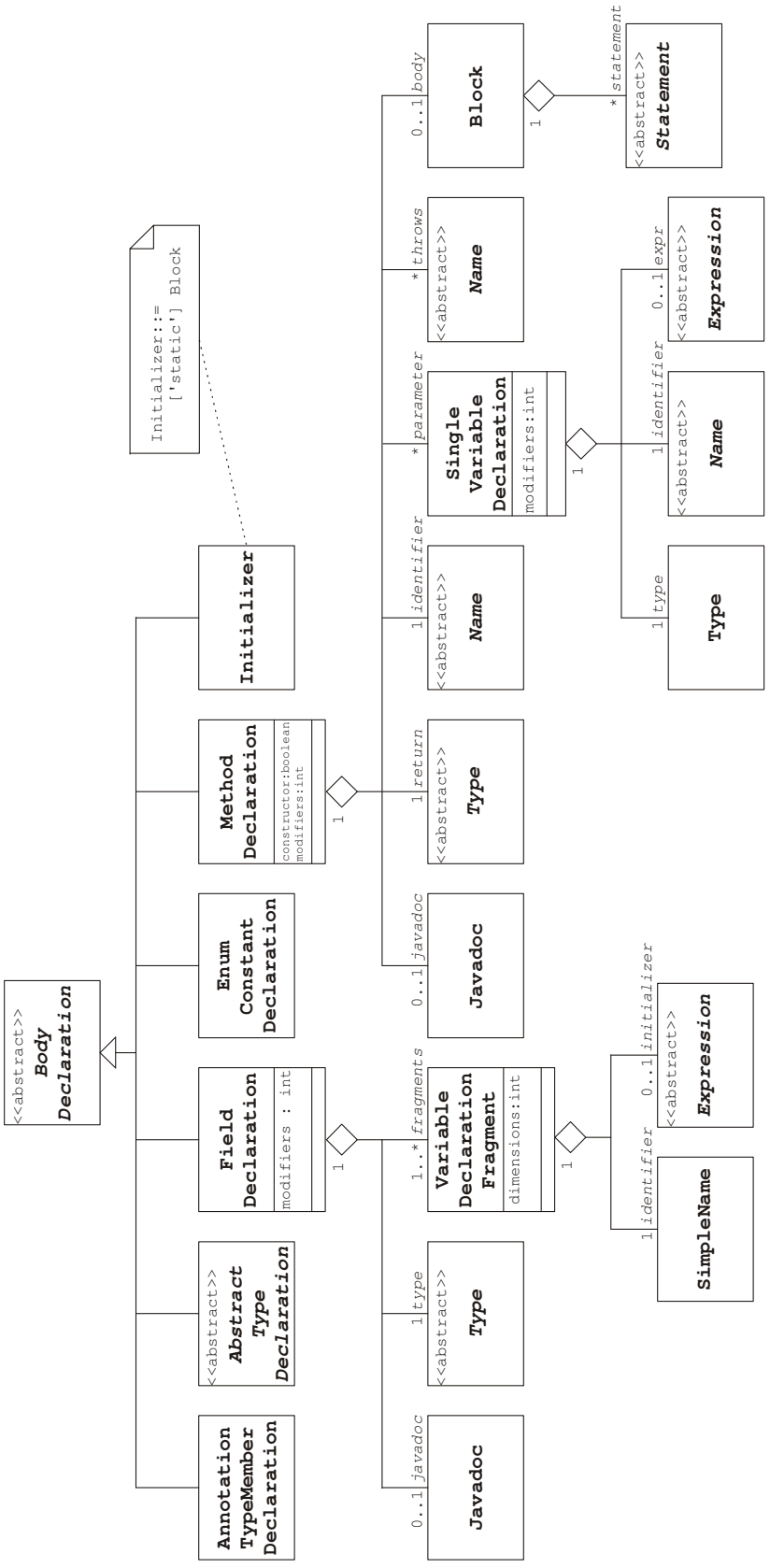
WhileStatement ::= 'while' '(' Expression ')' Statement

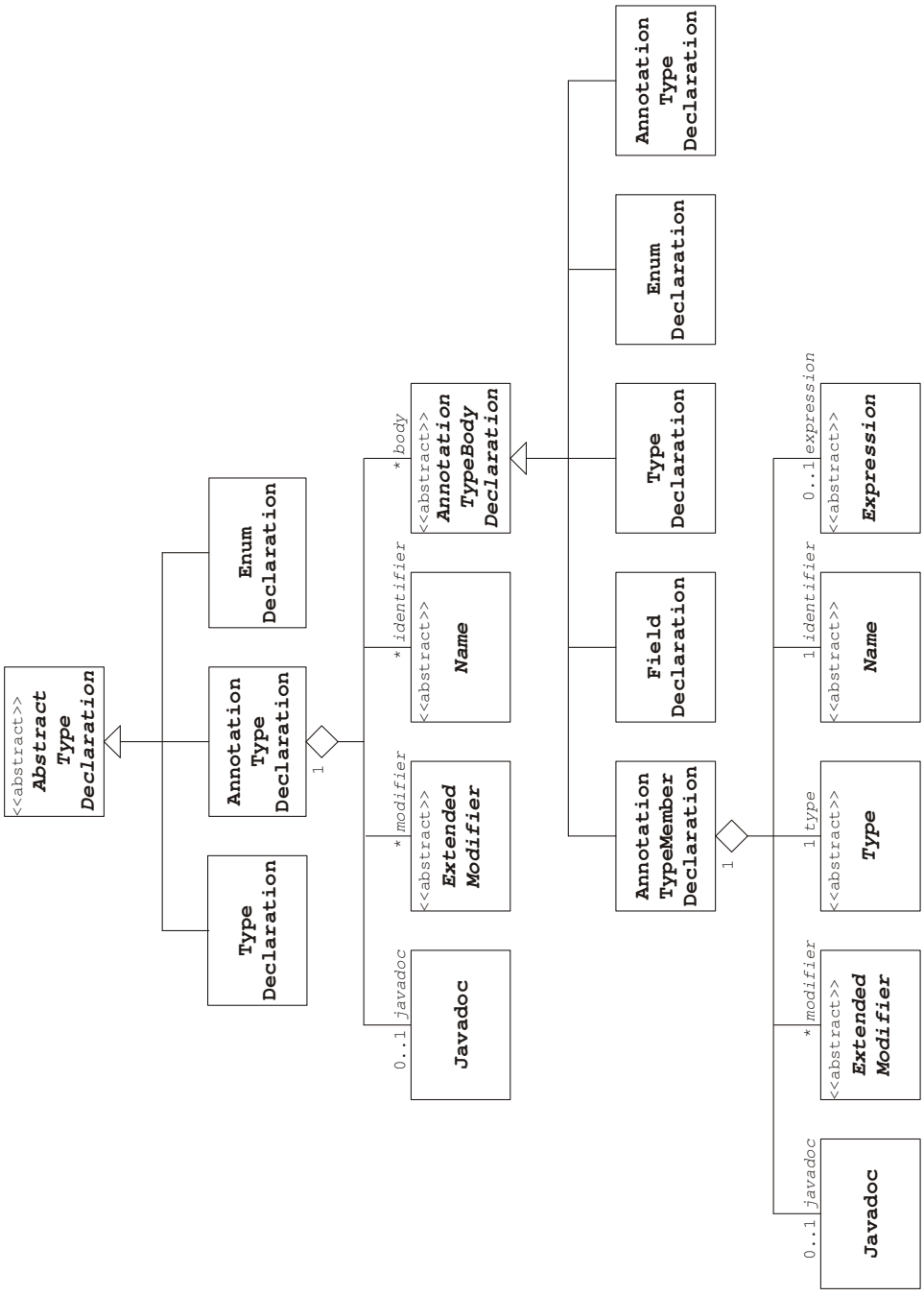
Anhang B
AST-Metamodell

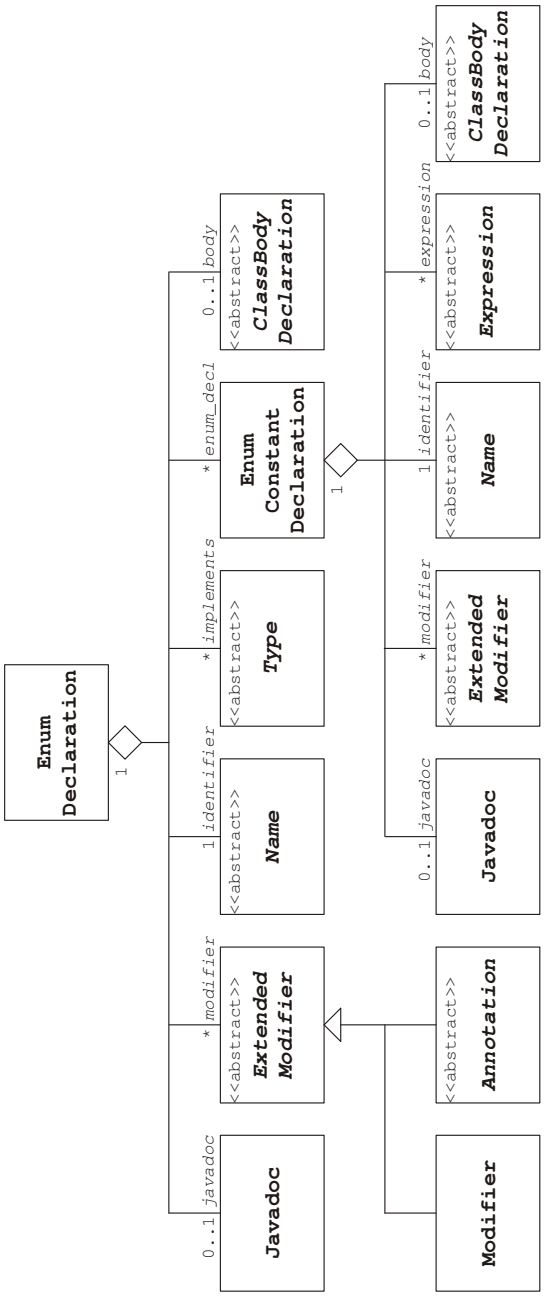
Das folgende Metamodell repräsentiert den Aufbau eines ASTs ausgehend von einer Compilation Unit und wurde auf Basis der Java-Grammatik sowie verschiedener erzeugter JDT-ASTs generiert. Auf die Erstellung eines Teilmodells für einen Javadoc-Knoten wurde bewusst verzichtet, da dieser keine Auswirkung auf die durchzuführenden Graphtransformationen hat und daher nicht weiter betrachtet wird.

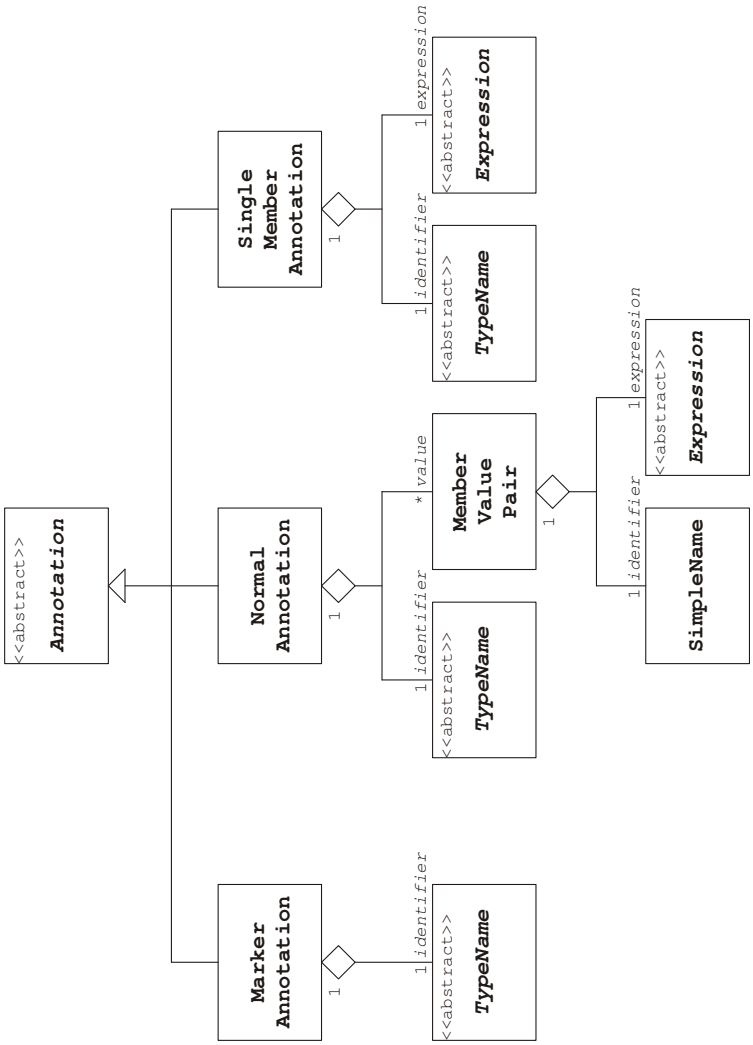
Die mit `<abstract>` gekennzeichneten Elemente entsprechen abstrakten Klassen innerhalb des `org.eclipse.jdt.core.dom`-Pakets, weshalb diese Knoten nie in einem Syntaxbaum vorkommen. Sie dienen lediglich zur Generalisierung verschiedener spezialisierter Elemente. So ist beispielsweise *`Name`* ein abstraktes Element, welches einen Repräsentant für die konkreten Instanzen `SimpleName` und `QualifiedName` darstellt.

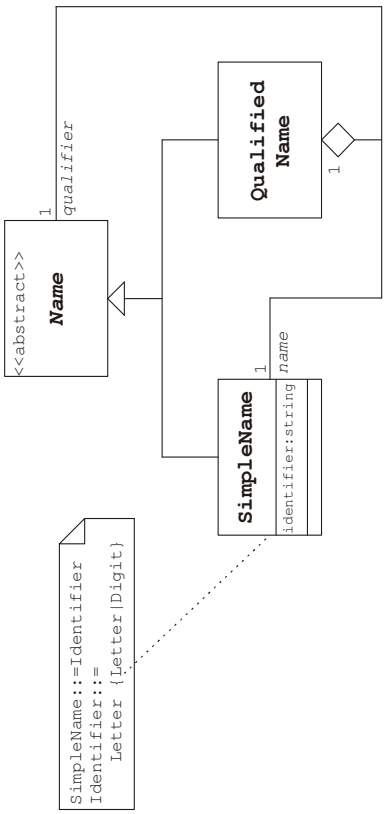

ClassDeclaration:: interface=false
InterfaceDeclaration:: interface=true
Compilation Unit
1
*
type_decl
Type Declaration
interface:boolean
modifiers:int
1
0..1
extends
<<abstract>>
Name
*
implements
<<abstract>>
Name
*
body
<<abstract>>
Body Declaration
1
identifier
<<abstract>>
Name
0..1
javadoc
Javadoc
*
import_decl
Import Declaration
1
1
imports
<<abstract>>
Name
0..1
pack_decl
Package Declaration
1
1
package

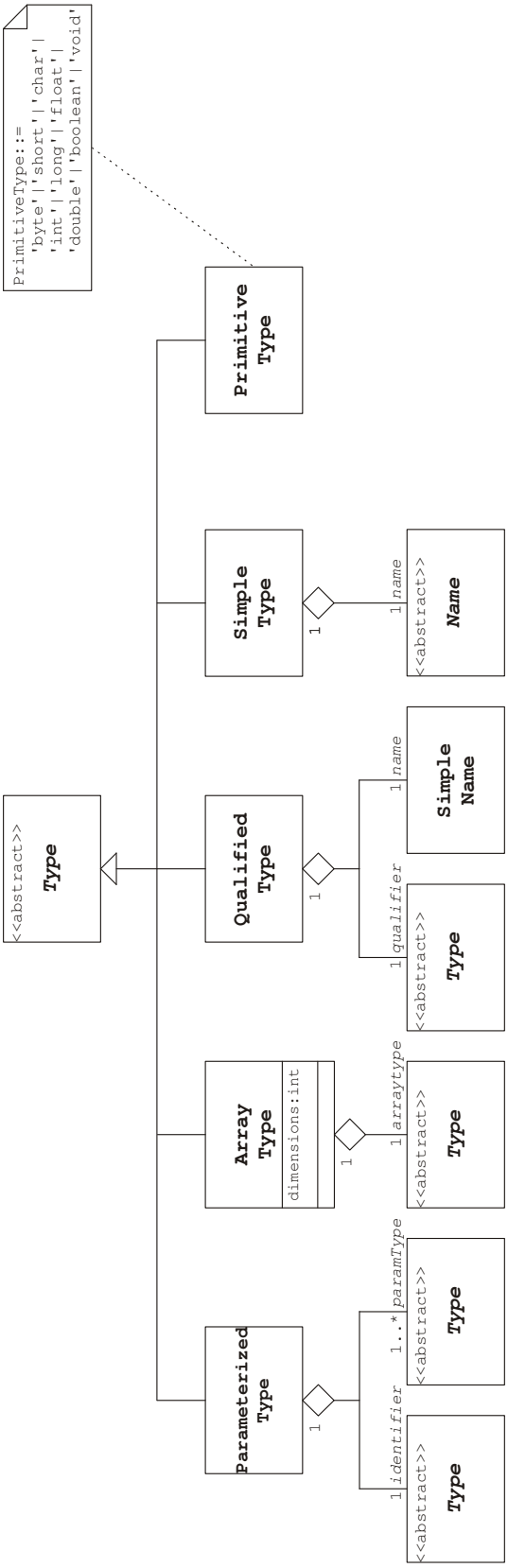

<<abstract>>
Body
Declaration
Initializer
Method
Declaration
constructor:boolean
modifiers:int
Enum
Constant
Declaration
Field
Declaration
modifiers : int
<<abstract>>
Abstract
Type
Declaration
Annotation
TypeMember
Declaration
Initializer::=
['static'] Block
1 identifier
<<abstract>>
Name
1 return
<<abstract>>
Type
0..1 javadoc
Javadoc
1..* fragments
Variable
Declaration
Fragment
dimensions:int
1 type
<<abstract>>
Type
0..1 javadoc
Javadoc
1 identifier
SimpleName
0..1 initializer
<<abstract>>
Expression
* parameter
Single
Variable
Declaration
modifiers:int
1 identifier
<<abstract>>
Name
0..1 expr
<<abstract>>
Expression
1 type
Type
* throws
<<abstract>>
Name
0..1 body
Block
* statement
<<abstract>>
Statement

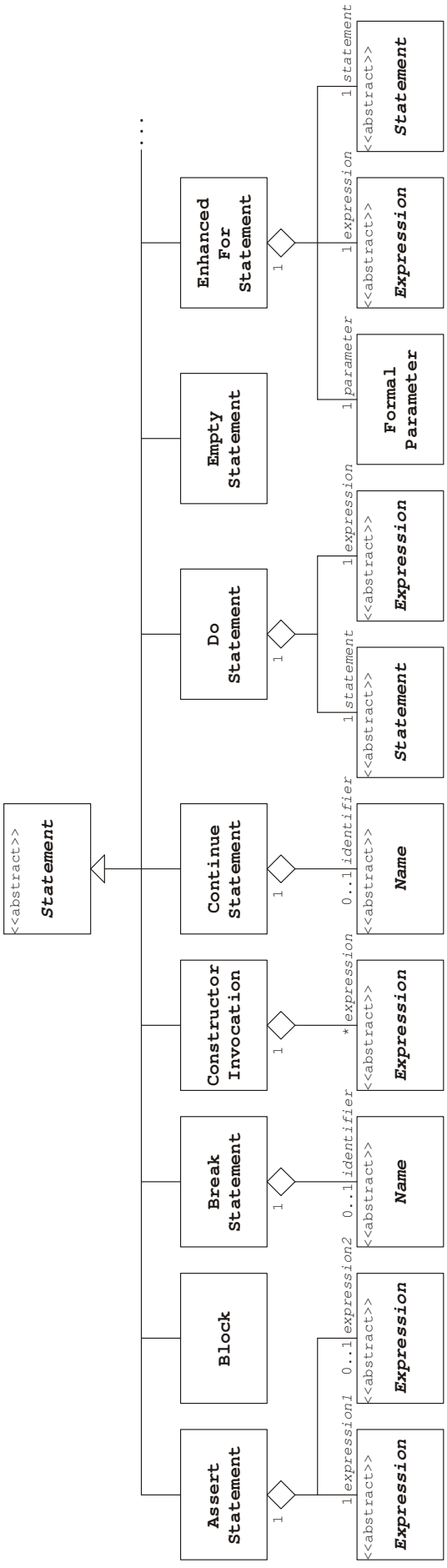

<<abstract>>
Abstract Type Declaration
Type Declaration
Annotation Type Declaration
Enum Declaration
1
* modifier
<<abstract>>
Extended Modifier
0..1 javadoc
Javadoc
* identifier
<<abstract>>
Name
* body
<<abstract>>
Annotation TypeBody Declaration
Type Declaration
Field Declaration
Annotation TypeMember Declaration
Enum Declaration
Annotation Type Declaration
1
1 type
<<abstract>>
Type
* modifier
<<abstract>>
Extended Modifier
0..1 javadoc
Javadoc
1 identifier
<<abstract>>
Name
0..1 expression
<<abstract>>
Expression

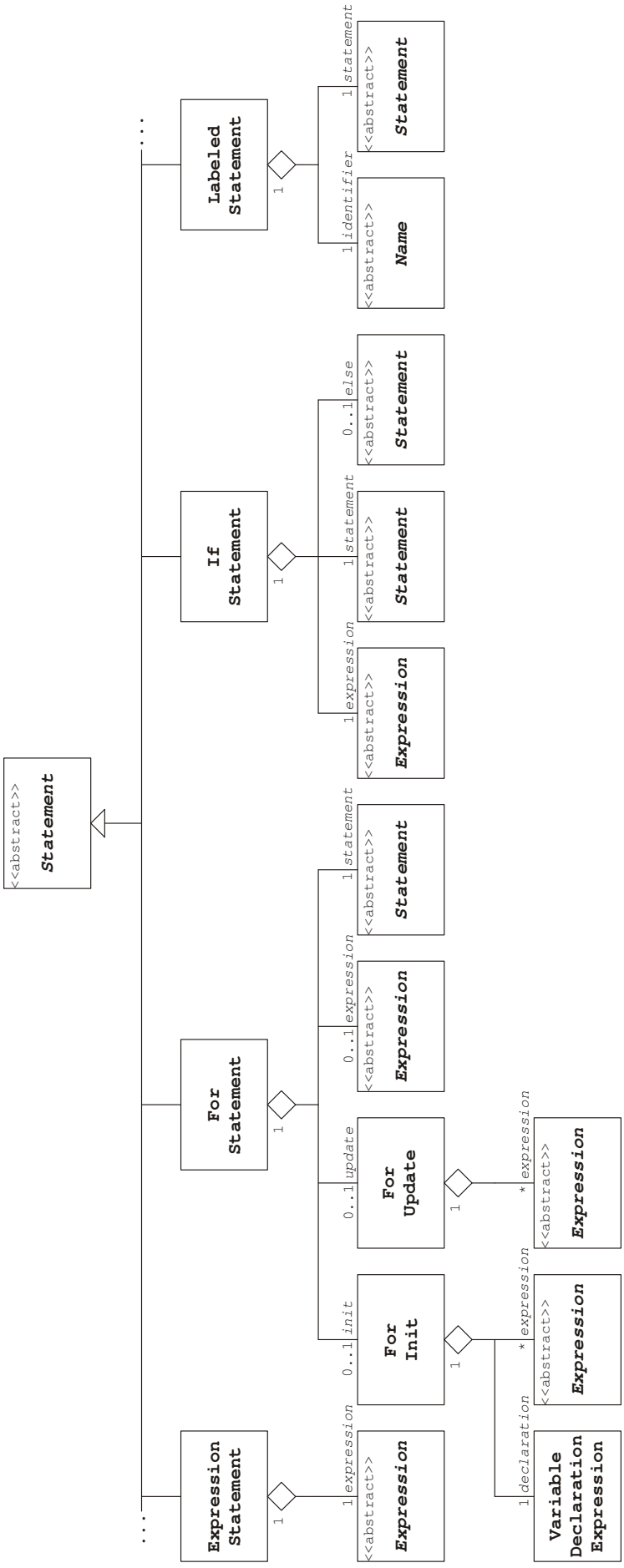

Enum Declaration
1
0..1 javadoc
Javadoc
* modifier
<<abstract>> Extended Modifier
<<abstract>> Annotation
Modifier
1 identifier
<<abstract>> Name
* implements
<<abstract>> Type
* enum_decl
Enum Constant Declaration
0..1 body
<<abstract>> ClassBody Declaration
0..1 javadoc
Javadoc
* modifier
<<abstract>> Extended Modifier
1 identifier
<<abstract>> Name
* expression
<<abstract>> Expression
0..1 body
<<abstract>> ClassBody Declaration

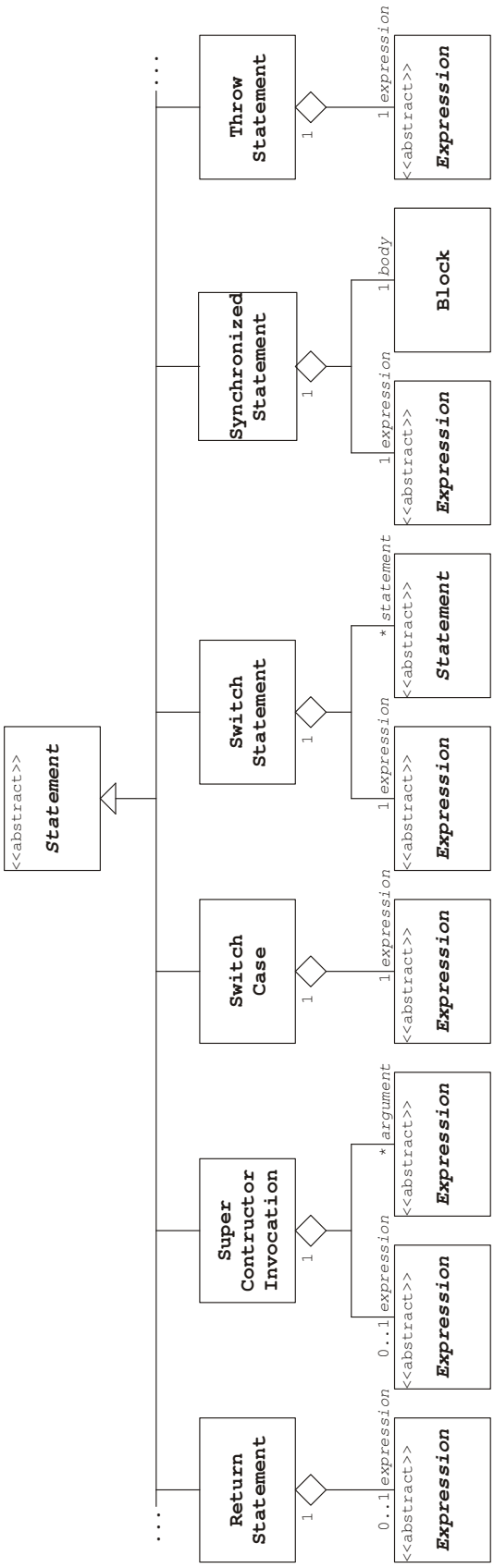

<<abstract>>
Annotation
Normal Annotation
Marker Annotation
Single Member Annotation
Member Value Pair
SimpleName
<<abstract>>
TypeName
<<abstract>>
Expression
identifier
value
expression
1
*

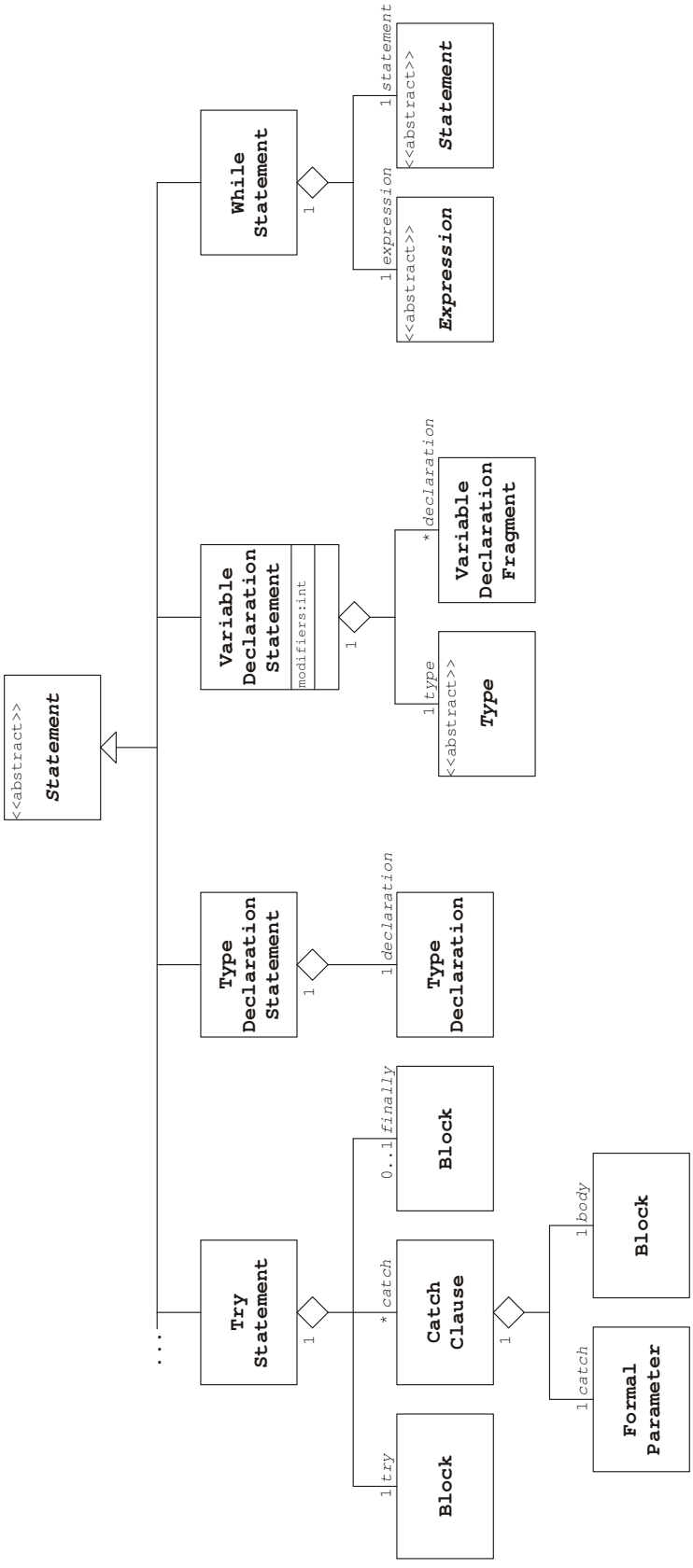

SimpleName:::=Identifier
Identifier:::=
Letter {Letter|Digit}
SimpleName
identifier:string
name
1
Qualified Name
1
qualifier
1
<<abstract>>
Name

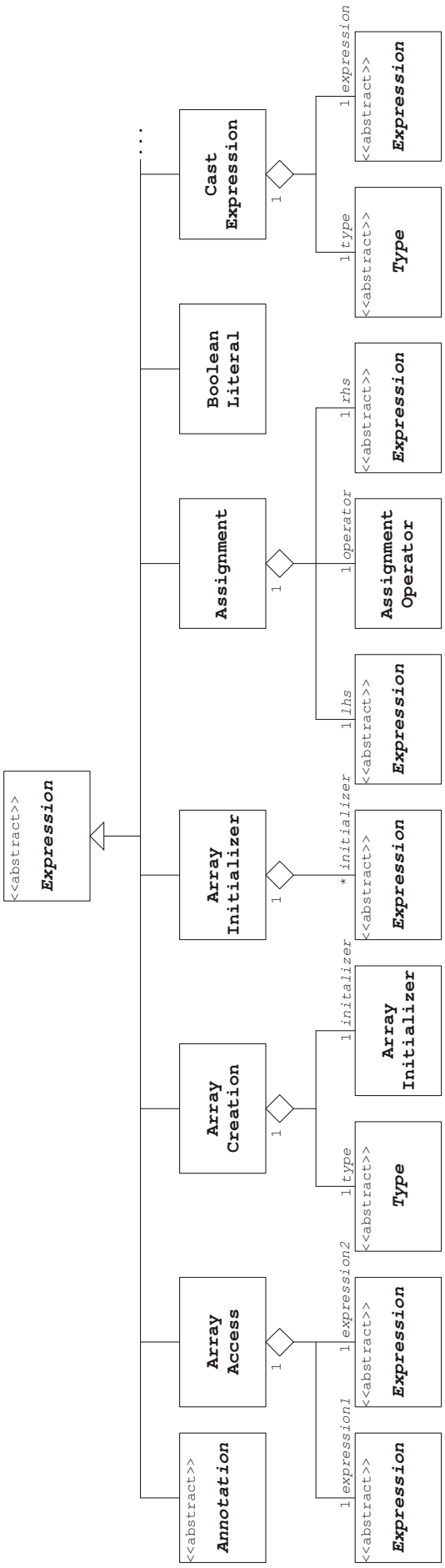

PrimitiveType::=
'byte'|'short'|'char'|
'int'|'long'|'float'|
'double'|'boolean'|'void'
<<abstract>>
Type
Primitive Type
Simple Type
Name
Qualified Type
Simple Name
Array Type
dimensions:int
Parameterized Type
1
1..*
name
qualifier
arraytype
paramType
identifier

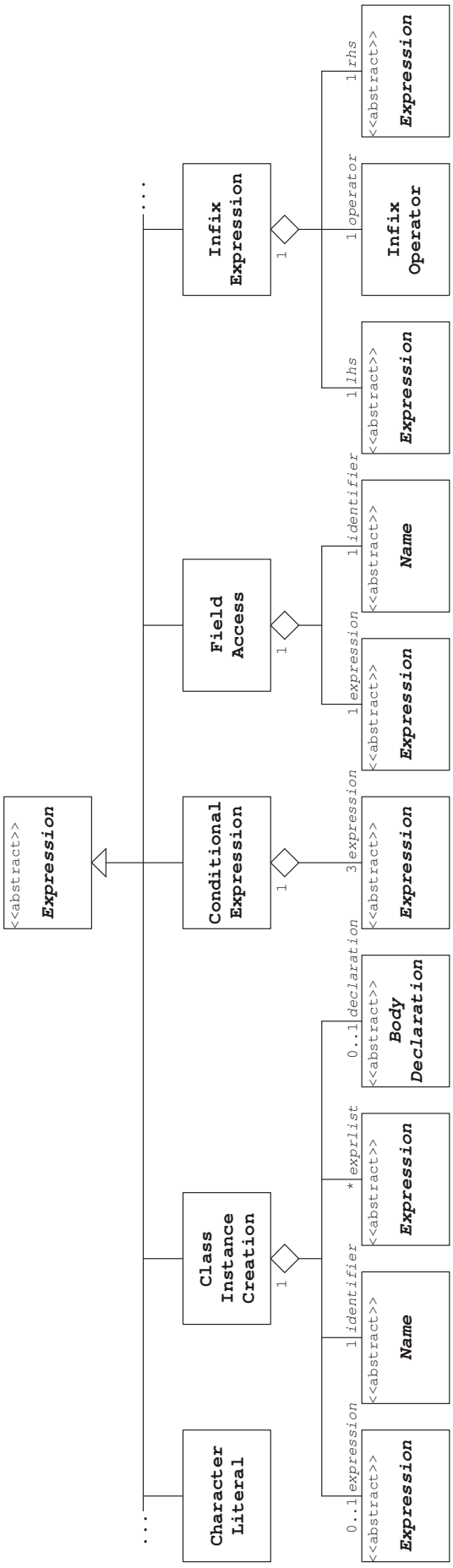

<<abstract>>
Statement
Assert Statement
Block
Break Statement
Constructor Invocation
Continue Statement
Do Statement
Empty Statement
Enhanced For Statement
...
<<abstract>>
Expression
<<abstract>>
Name
<<abstract>>
Statement
<<abstract>>
Expression
Formal Parameter
1
0..1
*
expression1
expression2
identifier
statement
parameter

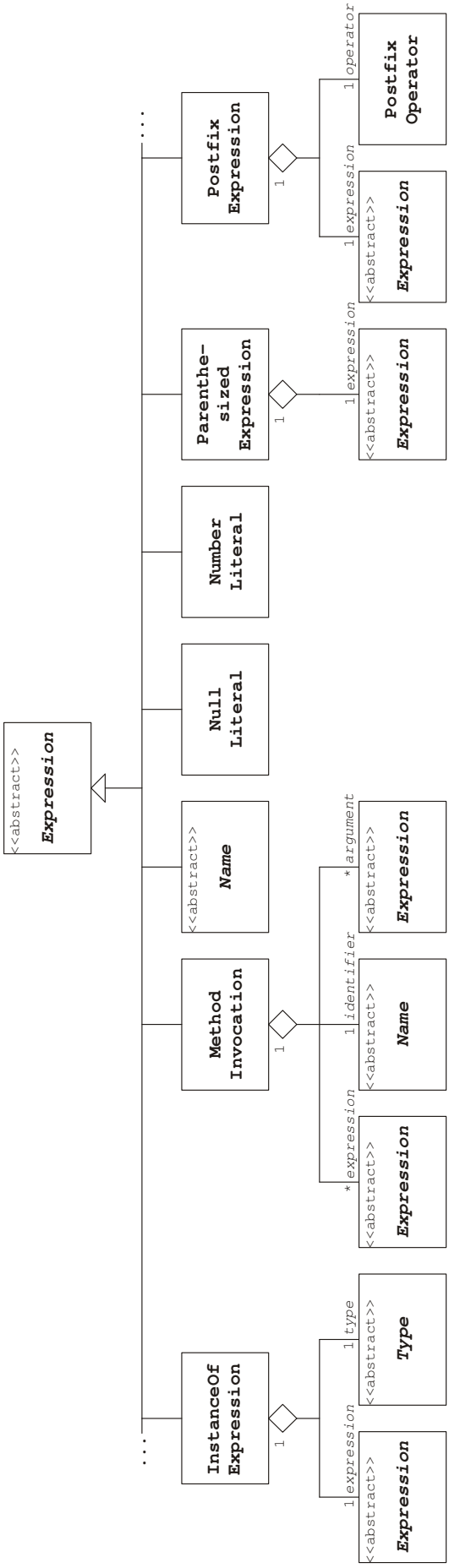

...
<<abstract>>
Statement
Expression Statement
1
1 expression
<<abstract>>
Expression
For Statement
1
0..1 init
For Init
1
* expression
<<abstract>>
Expression
1 declaration
Variable Declaration Expression
0..1 update
For Update
1
* expression
<<abstract>>
Expression
0..1 expression
<<abstract>>
Expression
1 statement
<<abstract>>
Statement
If Statement
1
1 expression
<<abstract>>
Expression
1 statement
<<abstract>>
Statement
0..1 else
<<abstract>>
Statement
Labeled Statement
1
1 identifier
<<abstract>>
Name
1 statement
<<abstract>>
Statement
...

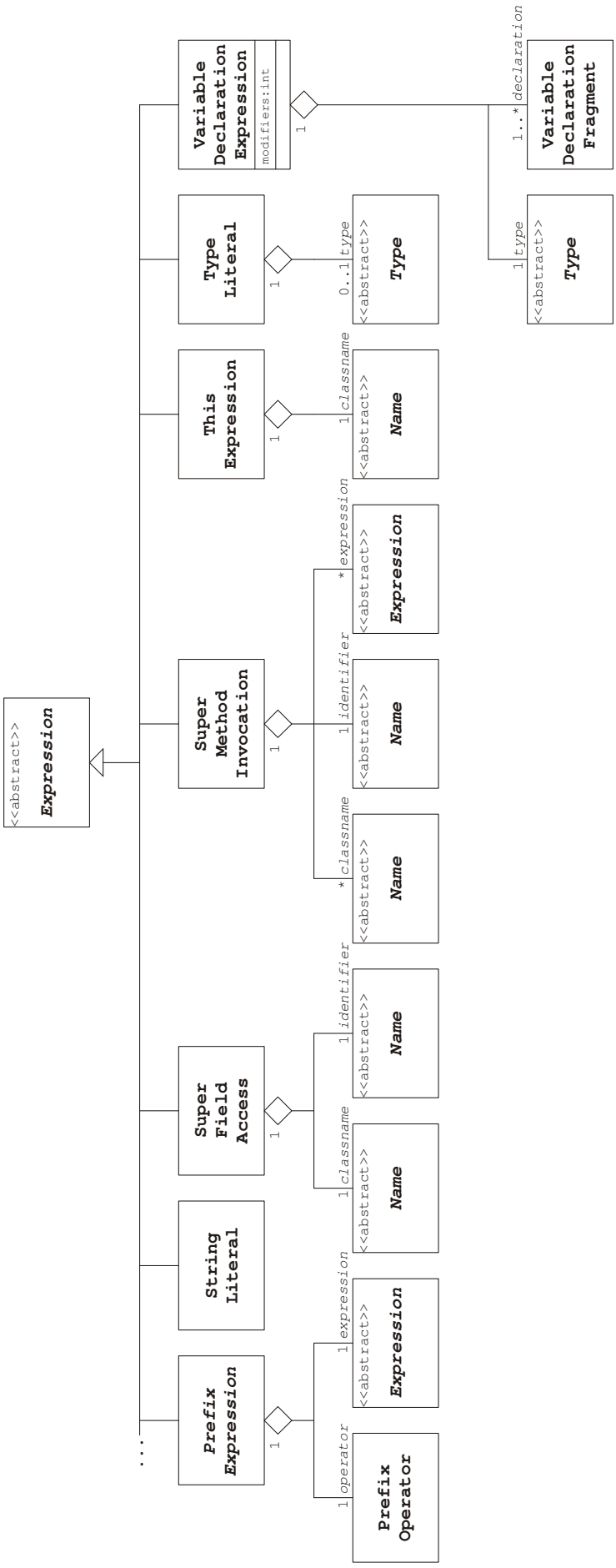

«abstract»
Statement
...
...
Return Statement
Super Contructor Invocation
Switch Case
Switch Statement
Synchronized Statement
Throw Statement
Block
«abstract» Expression
expression
argument
statement
body
1
0..1
*


<<abstract>>
Statement
...
While Statement
1
1 expression
<<abstract>>
Expression
1 statement
<<abstract>>
Statement
Variable Declaration Statement
modifiers:int
1
1 type
<<abstract>>
Type
* declaration
Variable Declaration Fragment
Type Declaration Statement
1
1 declaration
Type Declaration
Try Statement
1
1 try
Block
* catch
Catch Clause
0..1 finally
Block
1
1 catch
Formal Parameter
1 body
Block


<<abstract>>
Expression
...
Annotation
Array Access
Array Creation
Array Initializer
Assignment
Boolean Literal
Cast Expression
1 expression1
1 expression2
1 type
1 initializer
* initializer
1 lhs
1 operator
1 rhs
1 type
1 expression
Assignment Operator
Type


<<abstract>>
Expression
...
Character Literal
Class Instance Creation
1
identifier
1
<<abstract>>
Name
expression
0..1
<<abstract>>
Expression
exprlist
*
<<abstract>>
Expression
declaration
0..1
<<abstract>>
Body Declaration
Conditional Expression
1
expression
3
<<abstract>>
Expression
Field Access
1
expression
1
<<abstract>>
Expression
identifier
1
<<abstract>>
Name
Infix Expression
1
lhs
1
<<abstract>>
Expression
operator
1
Infix Operator
rhs
1
<<abstract>>
Expression
...


<<abstract>>
Expression
...
Postfix Expression
Parenthe-sized Expression
Number Literal
Null Literal
<<abstract>>
Name
Method Invocation
InstanceOf Expression
...
1
operator
1
Postfix Operator
1
expression
1
<<abstract>>
Expression
1
expression
1
<<abstract>>
Expression
1
identifier
*
argument
*
expression
1
<<abstract>>
Name
<<abstract>>
Expression
<<abstract>>
Expression
1
type
1
<<abstract>>
Type
1
expression
1
<<abstract>>
Expression


<<abstract>>
Expression
...
Prefix Expression
String Literal
Super Field Access
Super Method Invocation
This Expression
Type Literal
Variable Declaration Expression
modifiers:int
Variable Declaration Fragment
<<abstract>>
Type
<<abstract>>
Name
<<abstract>>
Expression
<<abstract>>
Name
<<abstract>>
Name
<<abstract>>
Name
<<abstract>>
Expression
Prefix Operator
operator
expression
classname
identifier
classname
identifier
expression
classname
declaration
type
type
1
0..1
1..*
*

Anhang C

ASTRefactor Plug-In: Manifest-Datei

de.hinterwaeller.astrefactor
plugin.xml

```
<plugin
   id="de.hinterwaeller.astrefactor"
   name="ASTRefactor Plug-in"
   version="1.0.0"
   provider-name=""
   class="de.hinterwaeller.astrefactor.ASTRefactorPlugin">

   <runtime>
      <library name="astrefactor.jar">
         <export name="*"/>
      </library>
   </runtime>

   <requires>
      <import plugin="org.eclipse.ui"/>
      <import plugin="org.eclipse.core.runtime"/>
      <import plugin="org.eclipse.jdt.core"/>
      <import plugin="org.eclipse.core.resources"/>
      <import plugin="org.eclipse.ui.ide"/>
      <import plugin="org.eclipse.jface.text"/>
   </requires>

   <extension
      point="org.eclipse.ui.popupMenus">
      <viewerContribution
         targetID="#CompilationUnitEditorContext"
         id="de.hinterwaeller.astrefactor.viewerContribution">
         <menu
               label="ASTRefactor"
               path="additions"
               id="astrefactor.viewerpopupmenu">
               <separator name="popupseparator1"/>
               <separator name="popupseparator2"/>
            </menu>
            <action
            label="Show AST"
               class="de.hinterwaeller.astrefactor.ASTView"
               menubarPath="astrefactor.viewerpopupmenu/popupseparator1"
               id="astrefactor.viewershowAST"/>
            <action
               label="Extract Method"
               class="de.hinterwaeller.astrefactor.ExtractMethodRefactoring"
               menubarPath="astrefactor.viewerpopupmenu/popupseparator2"
               id="astrefactor.viewerextractmethod"/>
      </viewerContribution>
```

```
      <objectContribution
         adaptable="false"
         objectClass="org.eclipse.jdt.core.ICompilationUnit"
         id="de.hinterwaeller.astrefactor.objectContribution">
         <menu
            label="ASTRefactor"
            path="additions"
            id="astrefactor.objectpopupmenu">
            <separator name="popupseparator"/>
         </menu>
         <action
            label="ShowAST"
            class="de.hinterwaeller.astrefactor.ASTView"
            menubarPath="astrefactor.objectpopupmenu/popupseparator"
            id="astrefactor.objectshowAST"/>
      </objectContribution>
   </extension>

   <extension
      point="org.eclipse.ui.actionSets">
      <actionSet
         label="ASTRefactor"
         visible="true"
         id="de.hinterwaeller.astrefactor.menuContribution">
         <menu
            label="ASTRefactor"
            id="astrefactor.menu">
            <separator name="menuseparator1"/>
            <separator name="menuseparator2"/>
         </menu>
         <action
            label="Show AST"
            class="de.hinterwaeller.astrefactor.ASTView"
            menubarPath="astrefactor.menu/menuseparator1"
            id="astrefactor.showAST"/>
         <action
            label="Extract Method"
            class="de.hinterwaeller.astrefactor.ExtractMethodRefactoring"
            menubarPath="astrefactor.menu/menuseparator2"
            id="astrefactor.extractmethod"/>
      </actionSet>
   </extension>

</plugin>
```

Anhang D

ASTRefactor Plug-In: Klassenübersicht

de.hinterwaeller.astrefactor

Class ASTRefactorPlugin

public class **ASTRefactorPlugin**
extends AbstractUIPlugin

Constructor Summary

ASTRefactorPlugin()

Field Summary

private static ASTRefactorPlugin	**plugin**	*shared instance*
private ResourceBundle	**resourceBundle**	*resource bundle*

Method Summary

public static ASTRefactorPlugin	**getDefault()**	*Returns the shared instance.*
public static String	**getResourceString(String key)**	*Returns the string from the plugin's resource bundle, or 'key' if not found.*
public static String	**getPluginId()**	*Returns the plugin's id.*
public ResourceBundle	**getResourceBundle()**	*Returns the plugin's resource bundle.*
public void	**start(BundleContext context) throws Exception**	*This method is called upon plug-in activation.*
public void	**stop(BundleContext context) throws Exception**	*This method is called when the plug-in is stopped.*

de.hinterwaeller.astrefactor

Class ASTCreator

public class **ASTCreator**
extends java.lang.Object
implements IActionDelegate, IWorkbenchWindowActionDelegate, IEditorActionDelegate

Constructor Summary

ASTCreator()

Field Summary

private CompilationUnit	**fASTRoot**	*current AST root node*
private ICompilationUnit	**fJavaSource**	*current java source*
private ITextSelection	**fTextSelection**	*current text selection*
private IWorkbenchWindow	**fWbWnd**	*active workbench window*

Method Summary

private void	**createAST(ICompilationUnit javaSource) throws CoreException**	*Creates AST from java source.*
public void	**dispose()**	*Disposes this action delegate.*
protected IWorkbenchWindow	**getActiveWorkbenchWindow() throws CoreException**	*Returns active workbench window.*
protected CompilationUnit	**getASTRoot()**	*Returns current AST root node.*
private void	**getCorrespondingJavaSource(ISelection selection)**	*Gets corresponding java source from current selection.*
protected IStatus	**getErrorStatus(String message, Throwable th)**	*Returns error status for exception and displays message dialog.*
private IOpenable	**getJavaInput(IEditorPart part)**	*Returns java input for active editor.*
protected ICompilationUnit	**getJavaSource()**	*Returns current java source.*
protected ITextSelection	**getTextSelection()**	*Returns selected text.*
public void	**init(IWorkbenchWindow window)**	*Initializes this action delegate with the workbench window it will work in.*
public void	**run(IAction action)**	*Performs this action.*
public void	**selectionChanged(IAction action, ISelection selection)**	*Notifies this action delegate that the selection in the workbench has changed.*
public void	**setActiveEditor(IAction action, IEditorPart targetEditor)**	*Sets the active editor for the delegate.*

de.hinterwaeller.astrefactor

Class ASTView

public class **ASTView**
extends **ASTCreator**

Constructor Summary

ASTView()

Field Summary

private GridData	**fGridData**	*data for GridLayout*

Method Summary

public void	**run(IAction action)**	*Called on menu-action 'show AST'. Calls run() from superclass ASTCreator.*
private void	**showAST()**	*Runs visitor and creates new window to display (text)-ast.*

de.hinterwaeller.astrefactor

Class ExtractMethodRefactoring

public class **ExtractMethodRefactoring**
extends **ASTCreator**

Constructor Summary

ExtractMethodRefactoring()

Field Summary

private CompilationUnit	**fASTRoot**	*created AST root node*
private ExtractMethodDialog	**fDialog**	*extract method dialog*
private Vector	**fMarkedStatements**	*marked statements*
private Document	**fModifiedSourceDoc**	*modified source*
private Vector	**fModifiedVars**	*modified variables inside marked statements*
private MethodDeclaration	**fNewMethodDeclaration**	*new method declaration*

Method Summary

protected void	**applyModifiedSource()**	*Applies the ast-changes and modifies sourcecode.*
private void	**checkLocalVarModification() throws CoreException**	*Checks if local variable are modified inside statement.*
private void	**checkSelectedStatements(CompilationUnit astRoot, ITextSelection textSelection) throws CoreException**	*Checks validity of selected statements.*
private void	**createMethodCall(String methodName, Vector argument)**	*Creates method call.*
private void	**createMethodDeclaration(String methodName)**	*Creates new method declaration.*
private void	**createMethodParam(Vector param)**	*Creates method parameters.*
private void	**error(String methodName, String source)**	*Error-handling if methodname already exists in compilation unit or superclass.*
private boolean	**existsMethodNameInCU(String methodName)**	*Checks if methodname exists already in compilation unit.*
private boolean	**existsMethodNameInSC(String methodName)**	*Checks if methodname exists in compilation unit's superclass(es).*
private TypeDeclaration	**getCurrentMethodDeclaration()**	*Returns corresponding MethodDeclaration-node of marked statements.*
private TypeDeclaration	**getCurrentTypeDeclaration()**	*Returns corresponding TypeDeclaration-node of marked statements.*
private Vector	**getLocalVars()**	*Returns local variable vector.*
protected Document	**getRefactoredSource()**	*Creates refactored sourcecode from modified ast and returns it.*
private void	**moveMarkedCode()**	*Moves marked code (statements).*
public void	**run(IAction action)**	*Called on menu-action 'extract method'. Calls run() from superclass ASTCreator.*
protected boolean	**transactionExtractMethod(String methodName)**	*Extract method refactoring transaction.*

de.hinterwaeller.astrefactor

Class ExtractMethodDialog

public class **ExtractMethodDialog**
extends java.lang.Object

Constructor Summary

ExtractMethodDialog(ExtractMethodRefactoring caller)

Field Summary

private IWorkbenchWindow	**fActiveWindow**	*active workbench window*
private Button	**fBtnOk**	*ok button*
private Button	**fBtnPreview**	*preview button*
private ExtractMethodRefactoring	**fCaller**	*object calling this dialog*
private int	**fColumns**	*dialog's current columns*
private GridData	**fGridData**	*griddata for gridlayout*
private String	**fNewMethodName**	*new method name*
private ICompilationUnit	**fOrigSource**	*source compilation unit*
private Shell	**fShell**	*current shell*
private Text	**fTxtFldMethodName**	*new methods name textfield*

Method Summary

private void	**createButtons(final Shell shell)**	*Creates dialog's buttons and listeners.*
private void	**createMethodNameArea(Shell shell)**	*Creates dialog's method name area.*
private void	**createOrigSourceLabel(Shell shell)**	*Creates dialog's original source label.*
private void	**createOrigSourceTextField(String origSource, Shell shell)**	*Creates dialog's original source textfield area.*
private void	**createRefactoredSourceLabel(Shell shell)**	*Creates dialog's refactored source label.*
private void	**createRefactoredSourceTextField(Document refactoredSource, Shell shell)**	*Creates dialog's refactored source textfield area.*
private void	**createSeparator(Shell shell)**	*Creates dialog's separator.*
private void	**disablePreviewBtn()**	*Disables preview button.*
private void	**enableOkBtn()**	*Enables ok button.*
private void	**enablePreviewBtn()**	*Enables preview button.*
private void	**resetDlg()**	*Resets dialog.*
private void	**setMethodNameInTextField()**	*Sets method name in textfield.*
public void	**showDialog(ICompilationUnit source, Document modifiedSource, IWorkbenchWindow activeWindow)**	*Shows extract method dialog window.*
private void	**showModifiedSource()**	*Updates some dialog-elements and shows refactored source preview in additionally textview.*

de.hinterwaeller.astrefactor

Class FindLocalVarVisitor

public class **FindLocalVarVisitor**
extends org.eclipse.jdt.core.dom.ASTVisitor

Constructor Summary

FindLocalVarVisitor()

Field Summary

private Vector	**fDeclaringNode**	*binding-declaring node*
private Vector	**fLocalDeclaredVars**	*detected local variable declaration inside statements*
private Vector	**fLocalVars**	*detected local variables*
private Vector	**fModifiedVars**	*local variables that are modified by an assignment*

Method Summary

public static Vector	**getLocalVars(Vector stmts)**	*Creates visitor and returns local variable-vector.*
public static Vector	**getModifiedVars(Vector stmts)**	*Creates visitor and returns modified local variable-vector.*
private void	**insertDeclaringNode(SimpleName node, IVariableBinding varBinding)**	*Inserts binding-declaring node in vector.*
private void	**insertLocalDeclaredVar(SimpleName node)**	*Inserts local declared variable-node in vector.*
private void	**insertLocalVar(SimpleName node)**	*Inserts local variable-node in vector (no duplicates).*
private void	**insertModifiedVar(SimpleName node)**	*Inserts modified local variable in vector (no duplicates).*
public boolean	**visit(SimpleName node)**	*Visits all SimpleName-nodes and detects local variable.*

de.hinterwaeller.astrefactor

Class FindStatementVisitor

public class **FindStatementVisitor**
extends org.eclipse.jdt.core.dom.ASTVisitor

Constructor Summary

FindStatementVisitor(int offset, int length)

Field Summary

private Statement	**fCoveringStmt**	*covering statement*
private Statement	**fCoveredStmt**	*covered statement*
private int	**fEnd**	*end of selection in AST*
private static int	**fErrorStatus**	*error-status (0 = no error, >0 = error)*
private int	**fStart**	*start of selection in AST*
private Vector	**fStmtVector**	*selected AST-statements*

Method Summary

public static int	**getErrorStatus()**	*Returns current error status.*
public static Vector	**getMarkedStatements(CompilationUnit astRoot, int offset, int length)**	*Creates visitor and returns selection-covered statements-vector.*
private void	**insertStatementNode(Statement node)**	*Inserts selection-covered statement in statement-vector (no duplicates).*
public void	**preVisit(ASTNode node)**	*Visits all AST-nodes and detects selection-covered statement-nodes.*

de.hinterwaeller.astrefactor

Class ASTViewVisitor

public class **ASTViewVisitor**
extends org.eclipse.jdt.core.dom.ASTVisitor

Constructor Summary

ASTViewVisitor()

Field Summary

private StringBuffer	**fStringBuf**	*text-AST*

Method Summary

private StringBuffer	**getASTString()**	*Returns text-AST as stringbuffer.*
private void	**insertNode(String nodeString)**	*Inserts node-string in stringbuffer.*
public static StringBuffer	**perform(CompilationUnit cu)**	*Creates visitor and returns text-AST.*
public void	**preVisit(ASTNode node)**	*Visits all AST-nodes and creates text-AST.*

Literatur- und Quellenverzeichnis

[ArLa04] Arthorne, Laffra: **Official Eclipse 3.0 FAQ**, Addison Wesley, 2004.
(vgl. www.eclipsefaq.org)
[Stand: Jan. 2005]

[Astv05] **Eclipse AST View Plugin.**
(vgl. dev.eclipse.org/viewcvs/index.cgi/checkout/jdt-ui home/plugins/
org.eclipse.jdt.astview/index.html)
[Stand: Jan. 2005]

[BaMe04] Bäumer, Megert, Weinand: **Eclipse Plug-Ins – Entwickeln und Publizieren**,
in: ObjektSpektrum 01/2004.
(vgl. www.sigs.de/publications/os/2004/01/weinand_OS_01_04.pdf)
[Stand: Nov. 2004]

[BaWe04] Bäumer, Weinand: **Eclipse-Technologie als Basis für die Anwendungs-entwicklung**, in: JavaMagazin Sonderausgabe „Eclipse 3.0“, Herbst 2004, S. 25-37.

[Bolo03] Bolour, A.: **Notes on the Eclipse plug-in architecture**, Eclipse Corner Article.
(vgl. www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html)
[Stand: Nov. 2004]

[BoTo05] **Together® Edition for Eclipse**, Borland ®, 2005.
(vgl. www.borland.com/together)
[Stand: Jan. 2005]

[BrRo96] Brant, Roberts: **Refactoring Browser**, 1996.
(vgl. st-www.cs.uiuc.edu/users/brant/Refactory/RefactoringBrowser.html)
[Stand: März 2005]

[Dahm98] Dahm, et.al.: **TGraphen und EER-Schemata formale Grundlagen**, Fachbericht Informatik 12/98, Universität Koblenz-Landau, Institut für Informatik, Koblenz, 1998.

[Danj04] D'Anjou, et.al.: **The Java Developer's Guide to Eclipse, 2nd Edition**, Addison Wesley, 2004.

[Daum04] Daum, B.: **Java-Entwicklung mit Eclipse 3**, dpunkt-Verlag, 2004.

[Eber96] Ebert, et.al.: **Graph Based Modeling and Implementation with EER/GRAL**, Fachbericht Informatik 11/96, Universität Koblenz-Landau, Institut für Informatik, Koblenz, 1996.

[EbFr94] Ebert, Franzke: **A Declarative Approach to Graph Based Modeling**, Fachbericht Informatik 3/94, Universität Koblenz-Landau, Institut für Informatik, Koblenz, 1994.

[EbKu97] Ebert, Kullbach, Panse: **The Extract-Transform-Rewrite Cycle – A Step towards MetaCARE**, Fachbericht Informatik 28/97, Universität Koblenz-Landau, Institut für Informatik, Koblenz, 1997.

[Ecli04] **Eclipse Foundation.**
(vgl. www.eclipse.org)
[Stand: Nov. 2004]

[Epla04] **Eclipse platform plug-in manifest**, Eclipse Foundation.
(vgl. www.eclipse.org/documentation/html/plugins/org.eclipse.platform.doc.isv/doc/reference/misc/rplugman.html)
[Stand: Nov. 2004]

[Fisc02] Fischer, O.: **Aus alt mach neu – Refactoring verbessert Software**,
in: iX 04/2002, S.151.

[Fowl99] Fowler, M.: **Refactoring – improving the design of existing code**, Addison Wesley, 1999.

[Fran97] Franzke, A.: **GRAL 2.0: A Reference Manual**, Fachbericht Informatik 3/97, Universität Koblenz-Landau, Institut für Informatik, Koblenz, 1997.

[GaBe04] Gamma, Beck: **Contributing to Eclipse: Principles, Patterns, and Plug-Ins**, Addison Wesley, 2004.

[Jdtp04] **JDT Plug-in Developer Guide**, Eclipse Foundation.
(vgl. www.eclipse.org/documentation/pdf/org.eclipse.jdt.doc.isv_3.0.1.pdf)
[Stand: Jan. 2005]

[KaKu01] Kamp, Kullbach: **GReQL – Eine Anfragesprache für das GUPRO-Repository – Sprachbeschreibung (Version 1.3)**, Projektbericht 8/01, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 2001.

[Mrsg04] **Mrs.G - A GXL plugin for eclipse.**
(vgl. calla.ics.uci.edu/mrsg/index.php)
[Stand: Nov. 2004]

[Opdy92] Opdyke, W. F.: **Refactoring Object-Oriented Frameworks**, PhD Thesis, University of Illinois at Urbana-Champaign, 1992.
(vgl. ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z)
[Stand: Feb. 2005]

[Open04] **Open Service Gateway Initiative**.
(vgl. www.osgi.org)

[Pdeg04] **PDE Guide**, www.eclipse.org
(vgl. www.eclipse.org/documentation/pdf/org.eclipse.pde.doc.user_3.0.1.pdf)
[Stand: Nov. 2004]

[Plat04] **Platform Plug-in Developer Guide**, www.eclipse.org
(vgl. www.eclipse.org/documentation/pdf/org.eclipse.platform.doc.isv_3.0.1.pdf)
[Stand: Nov. 2004]

[Scho03] Schoen, T.: **Runderneuerung – Ein Überblick über Refactoring und dessen Einsatz bei der Java-Entwicklung**, in: JavaMagazin 08/2003.
(vgl. www.javamagazin.de/itr/online_artikel/psecom,id,391,nodeid,11.html)
[Stand: Feb. 2005]

[ScVi04] Schiffer, Violka: **Spielzimmer aufräumen – Refaktorisieren macht Quellcode lesbarer**, in: c't 17/2003, S.204.

[ScWi99] Schürr, Winter, Zündorf: **PROGRES: Language and Environment**,
in: G. Rozenberg (ed.): Handbook on Graph Grammar: Applications, Vol. 2, Singapur, 1999: World Scientific, 487-550.
(vgl. www.es.tu-darmstadt.de/english/research/publications/download/Handbook-II-13.pdf)
[Stand: März 2005]

[Shav04] Shavor, et.al.: **Eclipse – Anwendungen und Plug-Ins mit Java entwickeln**, Addison Wesley, 2004.

[Spiv92] Spivey, J. M.: **The Z Notation – A Reference Manual**, Prentice Hall, 1992.
(vgl. spivey.oriel.ox.ac.uk/~mike/zrm/zrm.pdf)
[Stand: April 2005]

[Wart04] Wartala, R.: **Java-Entwicklung mit Eclipse 3.0**, in: iX-Konferenz „Eclipse 3.0 – Get plugged in", Heidelberg, 15. Juni 2004.
(vgl. www.wartala.de/download/EclipseTutorialFinal.pdf)
[Stand: Nov. 2004]

[Weye04] Weyerhäuser, M.: **Die Programmierumgebung Eclipse**, in: JavaSpektrum CeBIT-Sonderausgabe 2003.
(vgl. www.sigs.de/publications/js/2003/cebit/weyerhaeuser_JS_cebit_03.pdf)
[Stand: Nov. 2004]

[Zuen96] Zündorf, A.: **Eine Entwicklungsumgebung für PROgrammierte GRaphErsetzungsSysteme - Spezifikation, Implementierung und Verwendung**, Dissertation, RWTH Aachen, Vieweg, 1996.

Erklärung zur Urheberschaft

Hiermit erkläre ich, dass ich die vorliegende Arbeit eigenständig und ausschließlich unter Bezugnahme auf die im Text genannten Quellen erstellt habe.

Dachsenhausen, im Juni 2005

_____

(Bodo Hinterwäller)